

## 2.4 Player

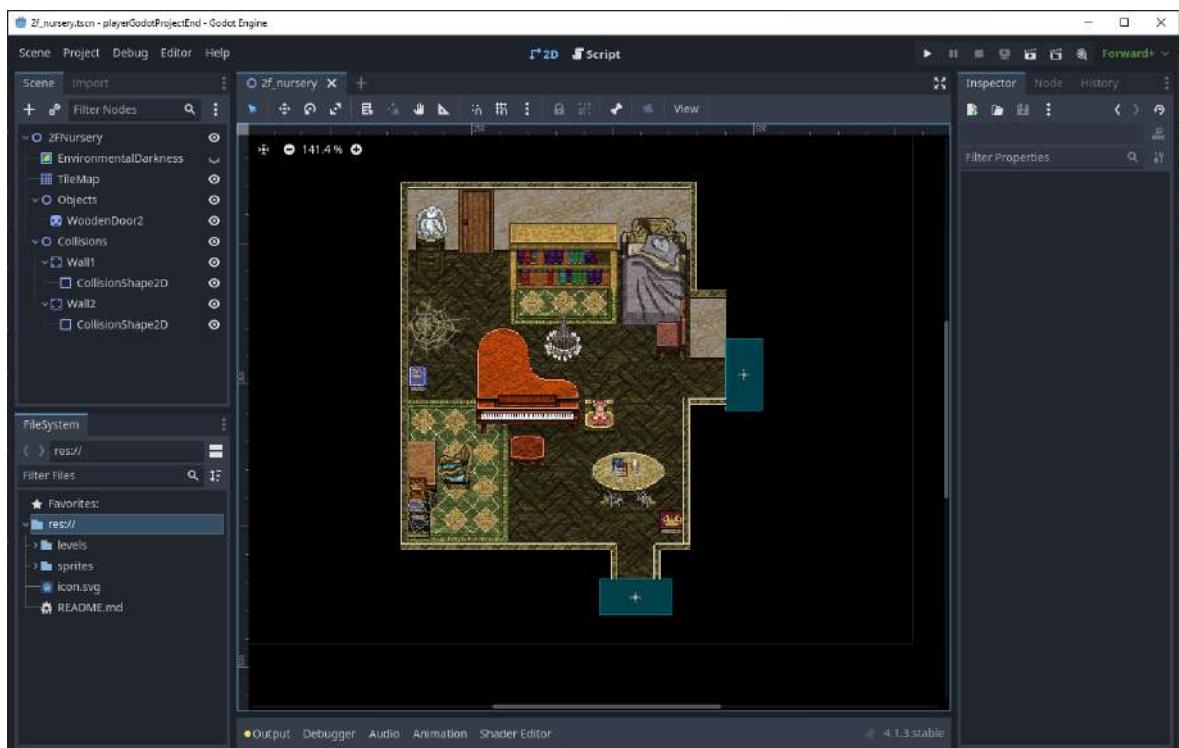
In questa sezione vedremo finalmente come si realizza uno dei pilastri fondanti di un videogioco: il **Player**.

Nelle prossime pagine impareremo come programmare la logica necessaria perché un utente possa controllare il nostro giocatore all'interno del gioco; far "collidere" il player con gli oggetti del mondo come muri, mobili, altri personaggi, ecc...; animare le azioni del player come la camminata; e, infine, gestire gli attacchi e la morte del giocatore stesso.

I progetti relativi a questa sezione sono disponibili nella repository [01-godot-project-player](#)<sup>37</sup> su GitHub, appartenente alla pagina con l'elenco delle repository di questo documento, il cui link è fornito all'inizio del capitolo. Il progetto di partenza per seguire gli argomenti della sezione è **Player [Start]**, situato nella cartella [//01-godot-project-player/start](#), mentre il progetto Godot completo è **Player [End]**, disponibile nella cartella [//01-godot-project-player/end](#). Infine, la sezione degli esercizi è pensata per essere svolta partendo dal progetto **Player [Exercise]** nella cartella [//01-godot-project-player/exercise](#).

### 2.4.1 Creazione della Player Scene

Aprendo il progetto **Player [Start]** che utilizzeremo come base di questo capitolo, ci troveremo davanti la seguente scena



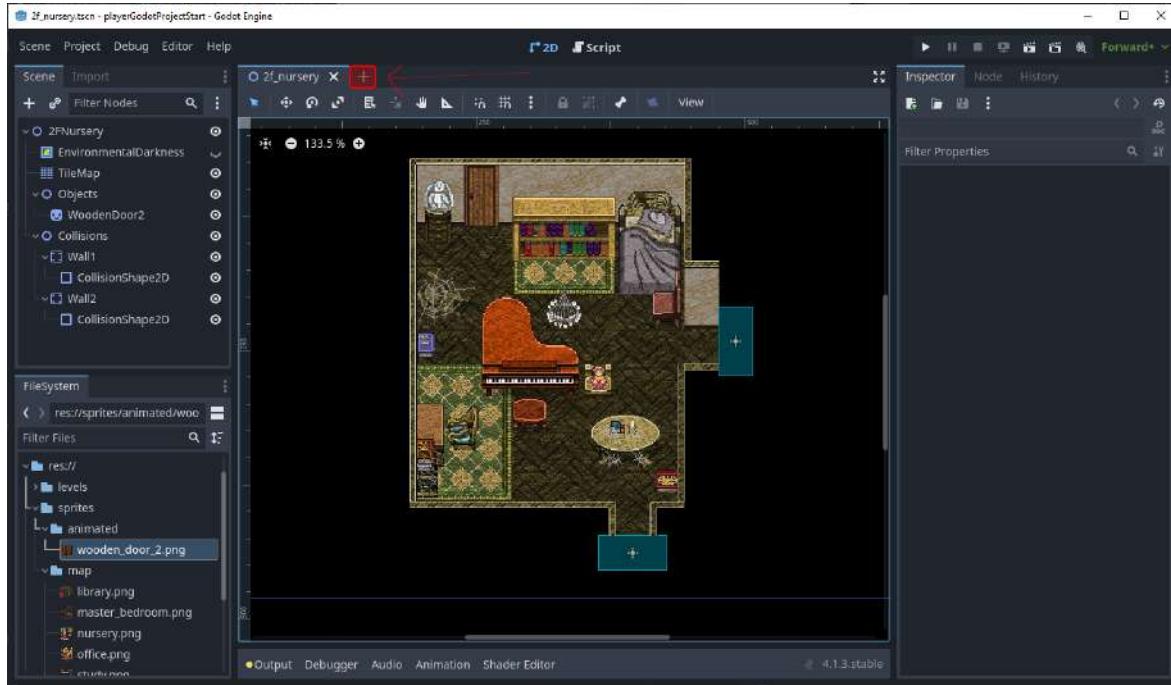
la quale è un livello del nostro videogioco.

Per adesso, ignoriamo i nodi come **EnvironmentalDarkness**, **TileMap**, **Objects**, **WoodenDoor2**, ecc..., che vedremo più avanti, utili solamente per creare la stanza.

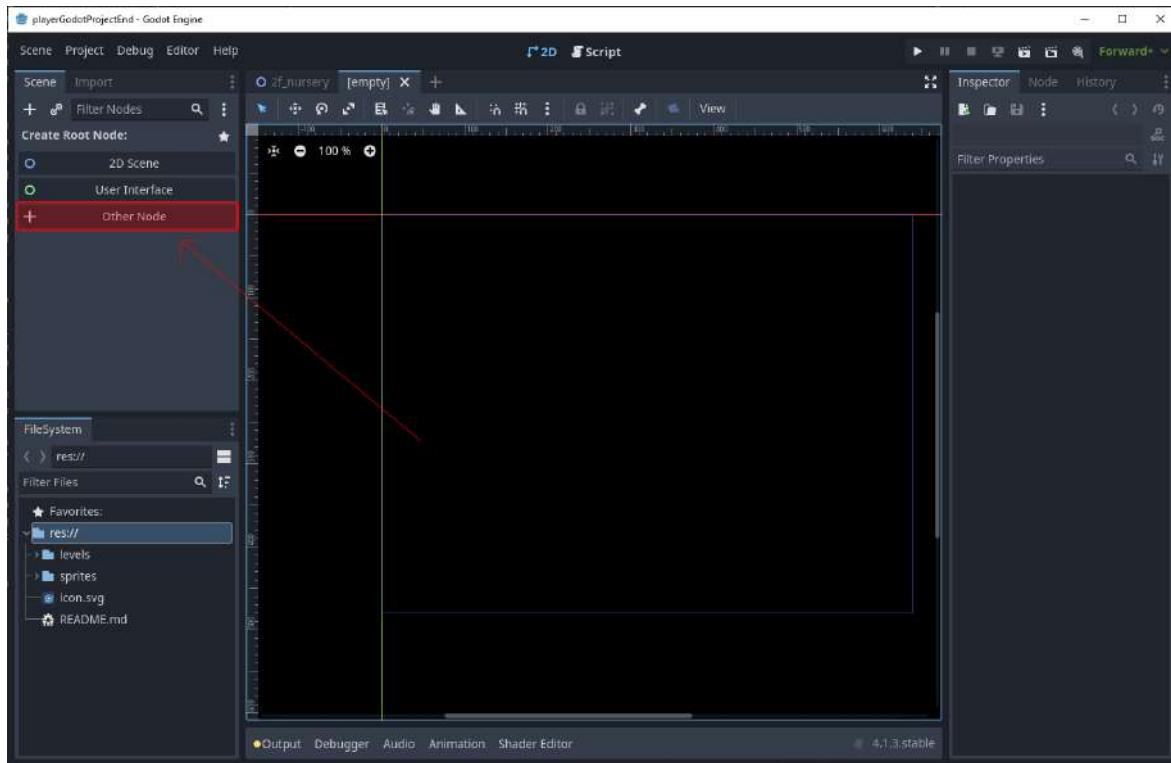
<sup>37</sup><https://github.com/rpg-course-godot/01-godot-project-player.git>

Come possiamo vedere la scena è pressocché completa, pronta per essere esplorata. Ma ci manca un player per poterlo fare!

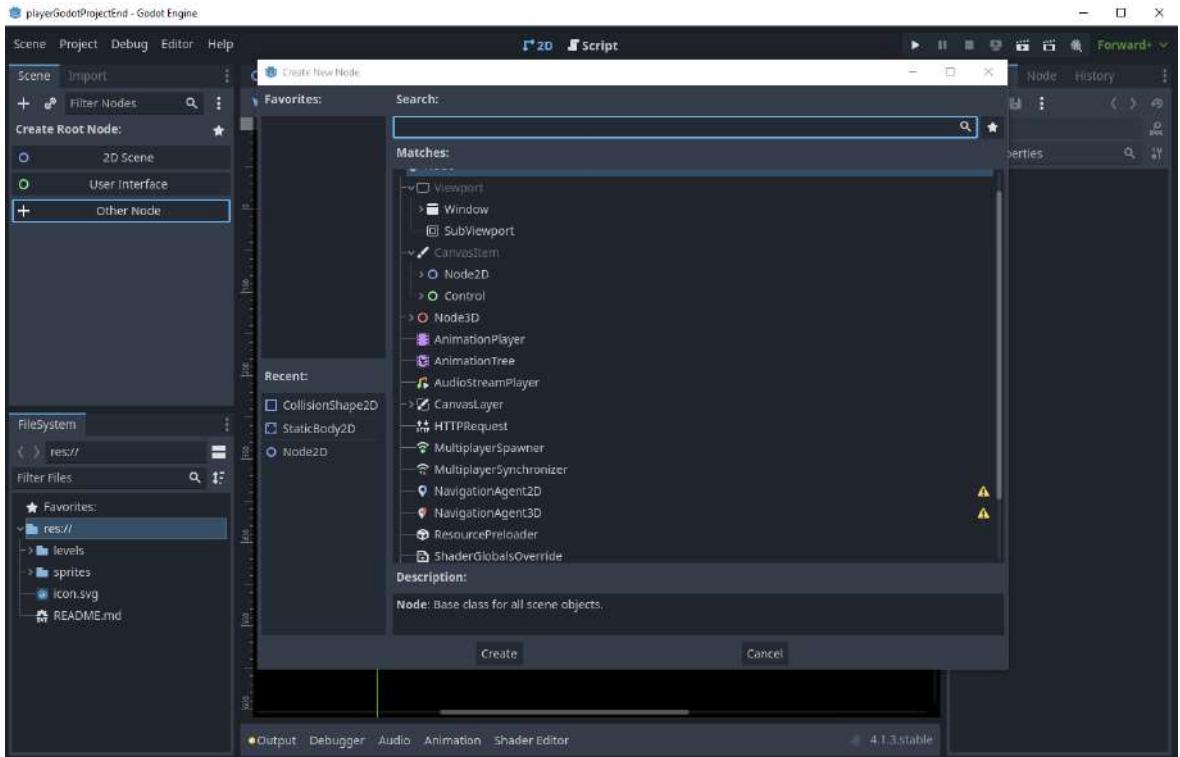
Per creare la nostra scena **Player**, clicchiamo il **+** di fianco alla tab della scena **2f\_nursery**



clicchiamo dunque la voce **Other Node** dalla nuova finestra che ci appare

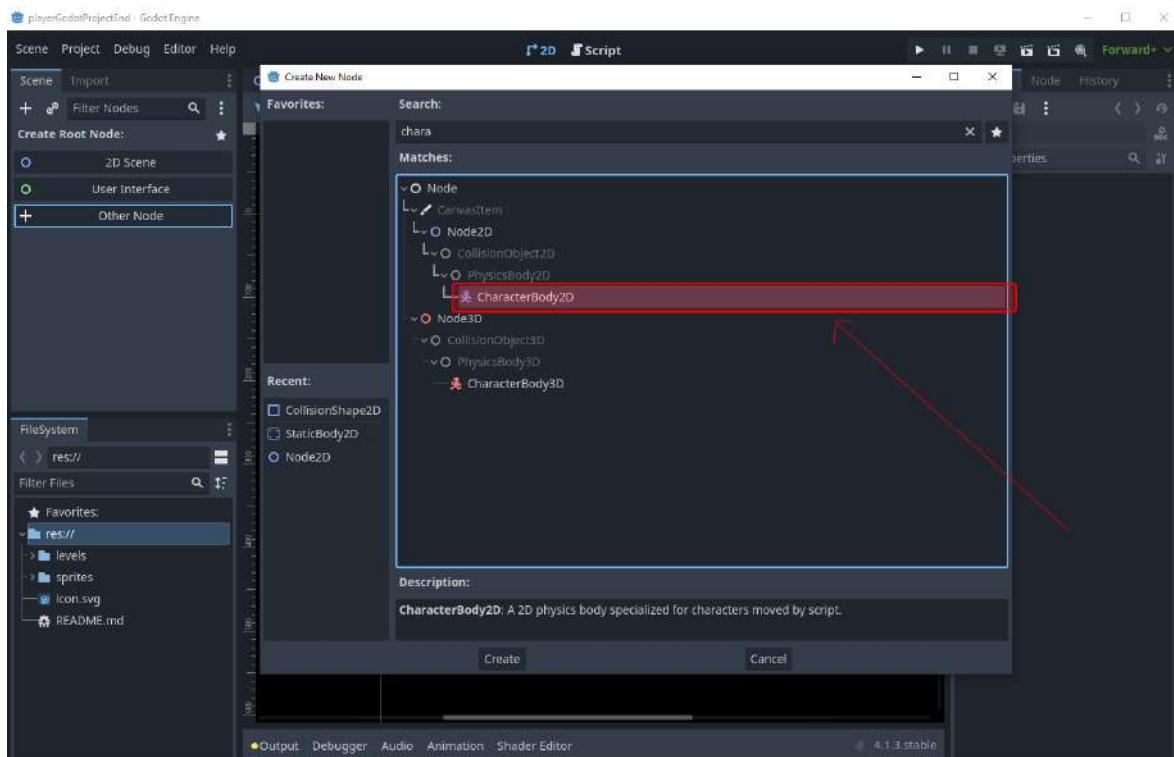


si aprirà così la finestra **Create New Node**

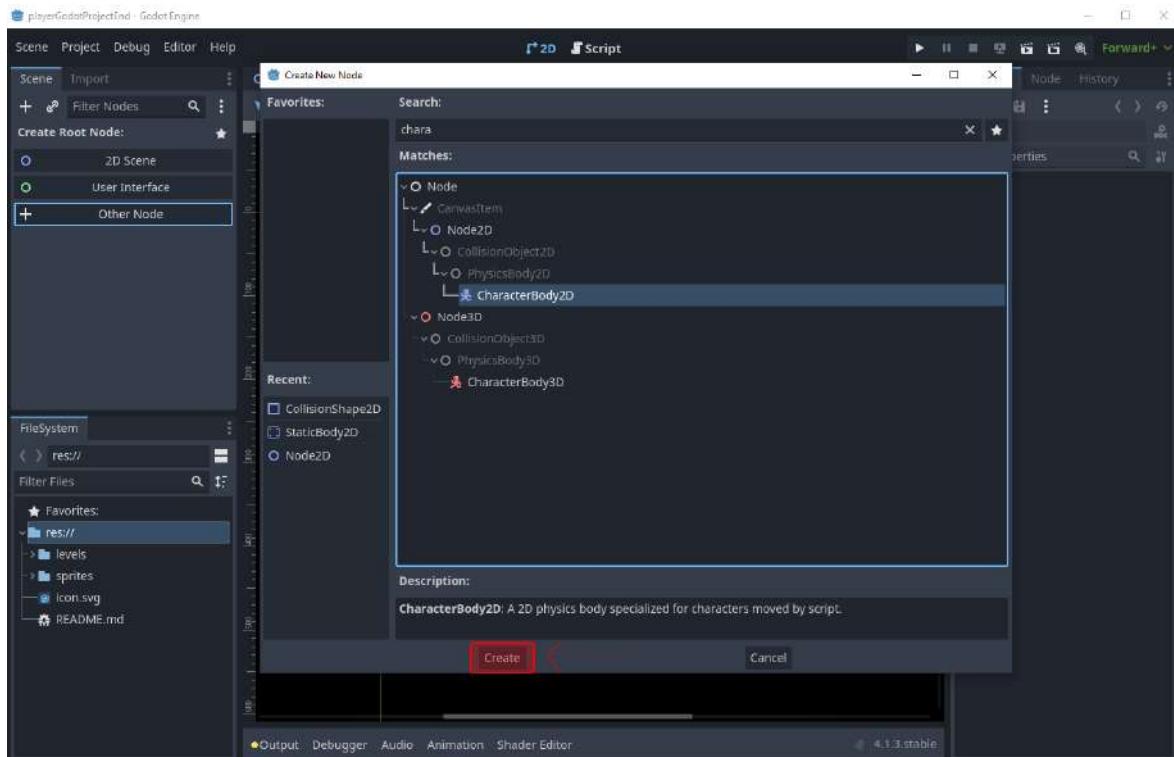


selezioniamo il nodo **CharacterBody2D**.

La classe **CharacterBody2D** è una classe specializzata per corpi fisici che sono pensati per essere controllati dall'utente. I nodi **CharacterBody2D** non sono affatto soggetti alla fisica ma possono influenzare gli altri corpi fisici sul loro percorso.

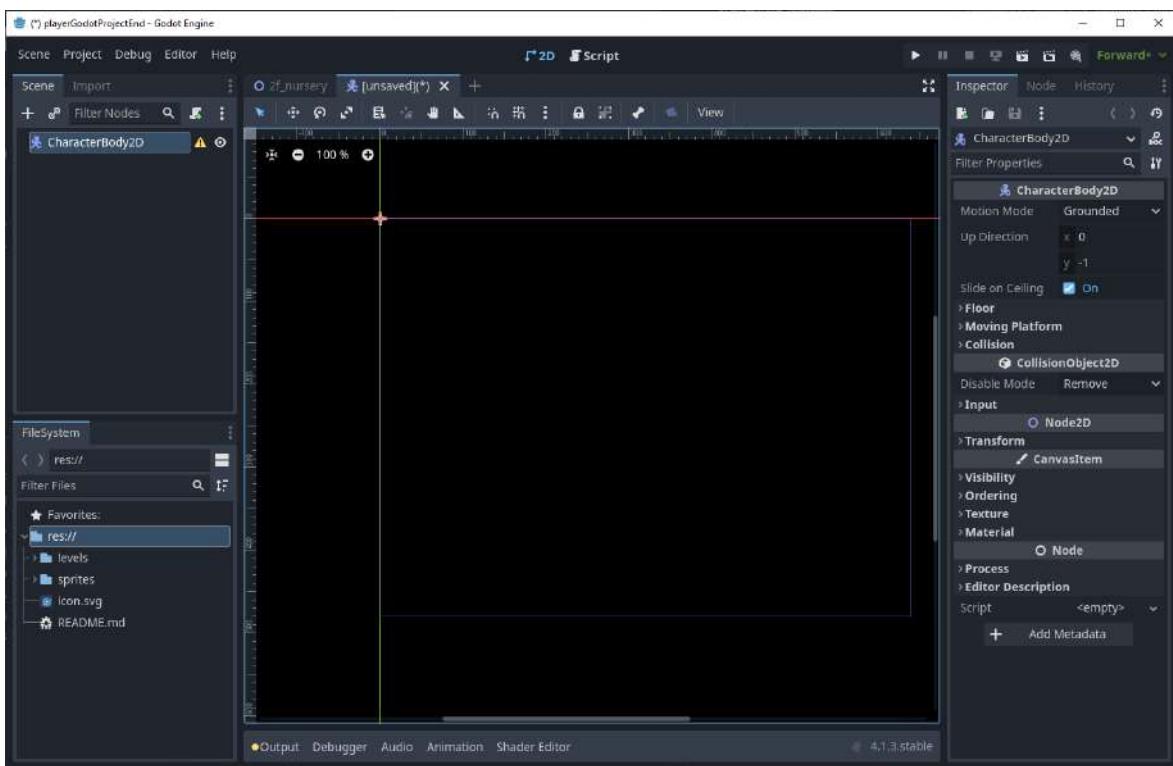


e clicchiamo su **Create**

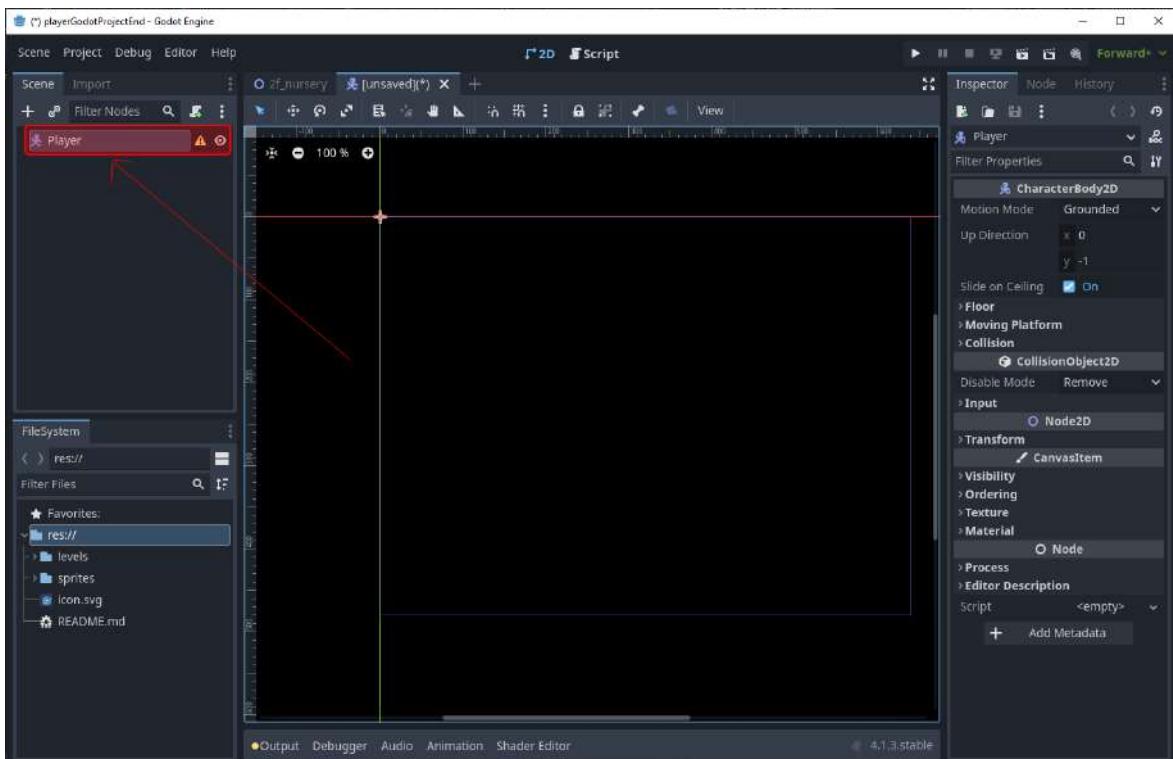


per creare la nuova scena

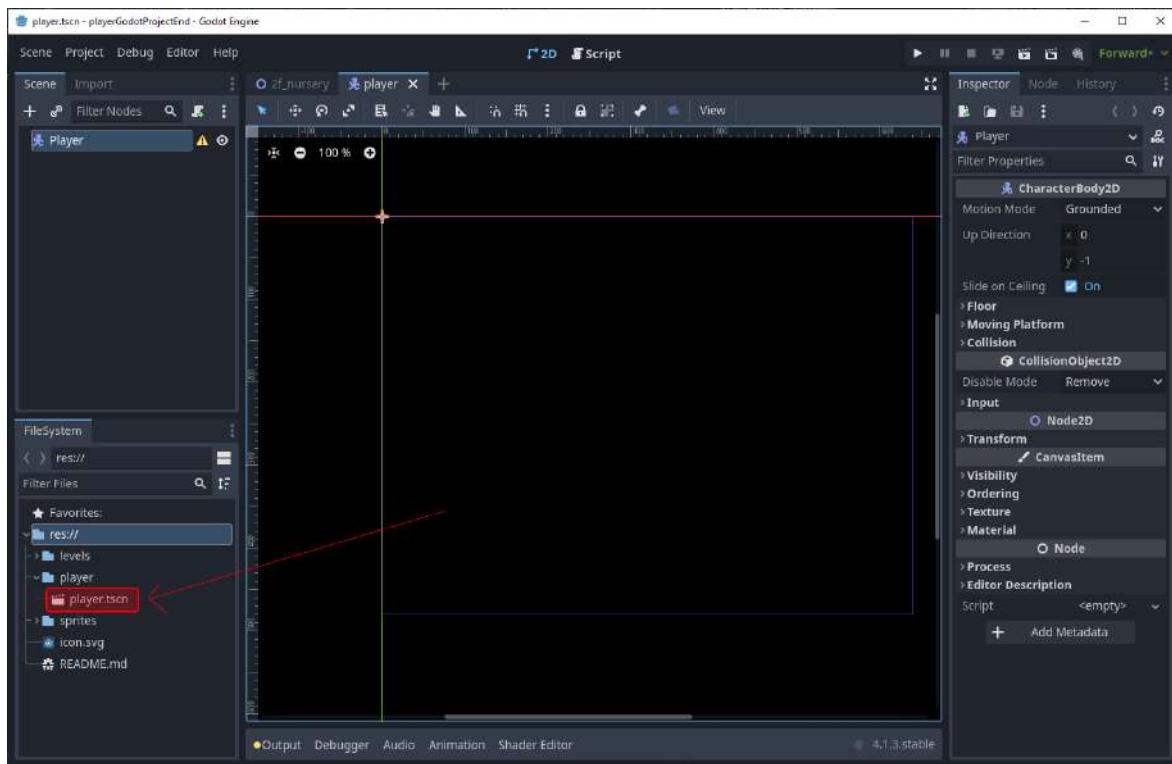
## CHAPTER 2. GODOT



rinominiamo dunque il nodo **CharacterBody2D** con **Player**



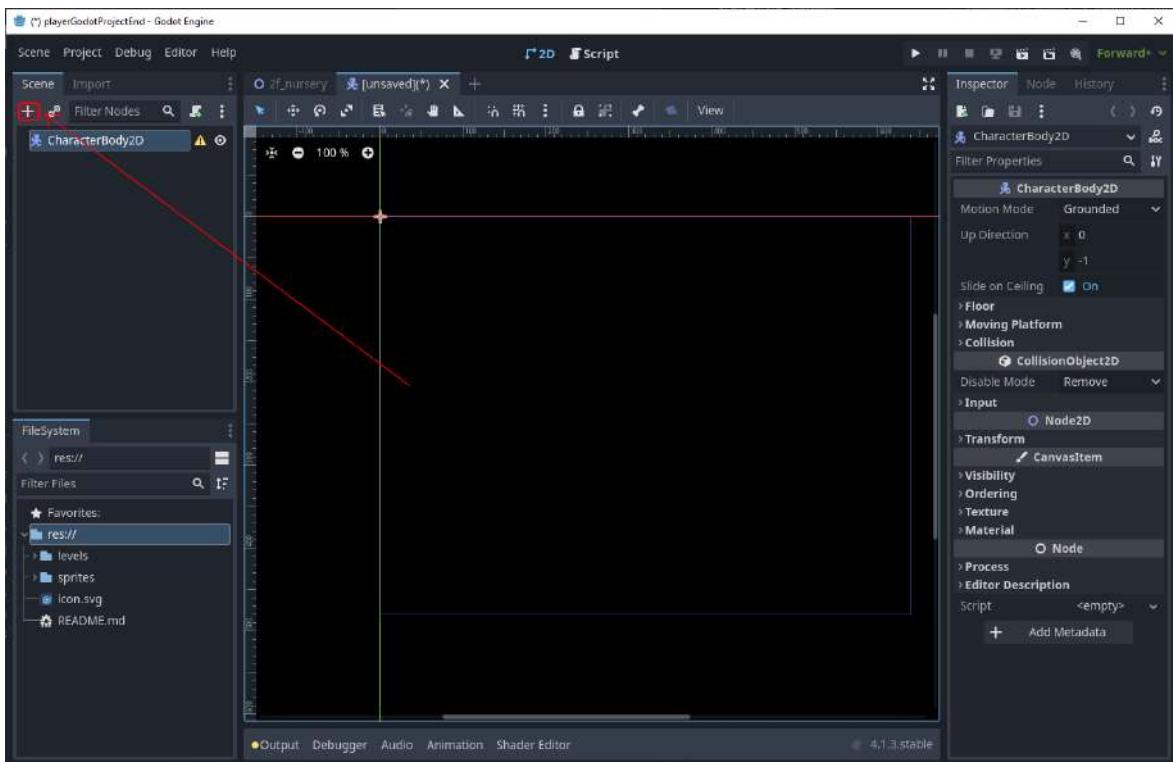
e salviamolo dunque in una nuova cartella **player** tramite **CTRL + S**



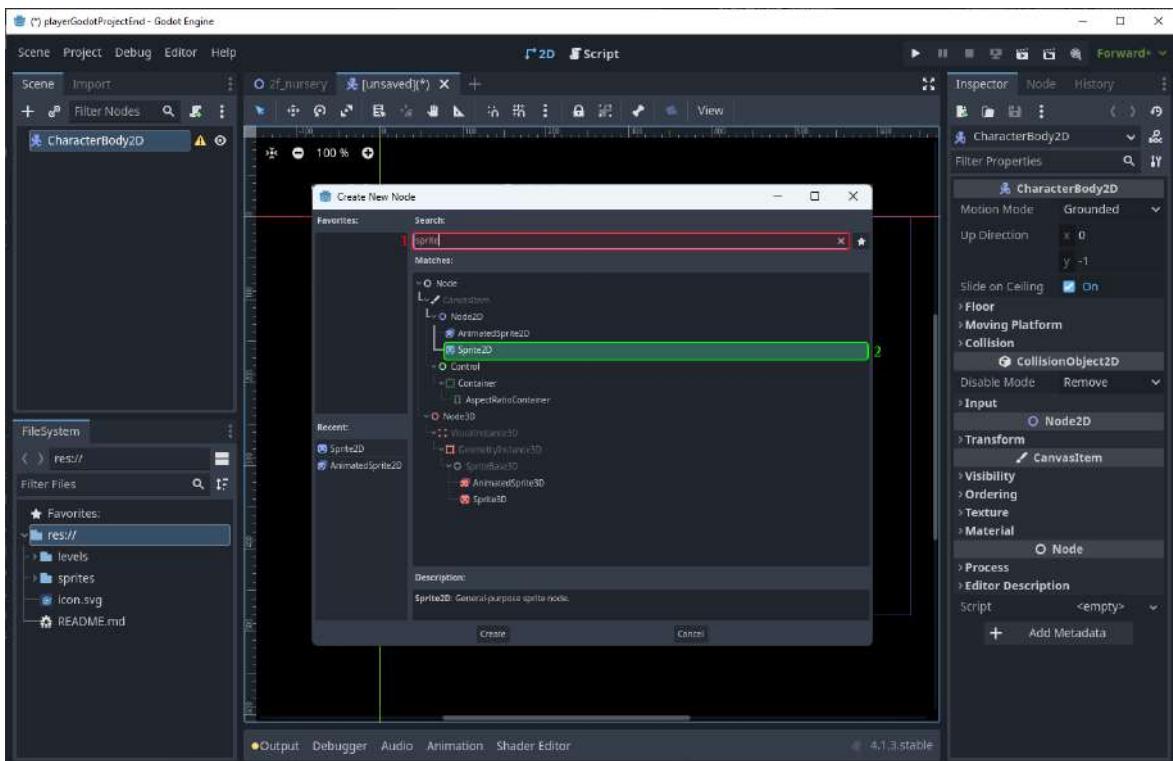
Abbiamo appena creato una nuova scena del personaggio, ma manca ancora un passo per poterlo vedere in "pixel e colori".

#### 2.4.2 Aggiunta di una Texture alla Player Scene

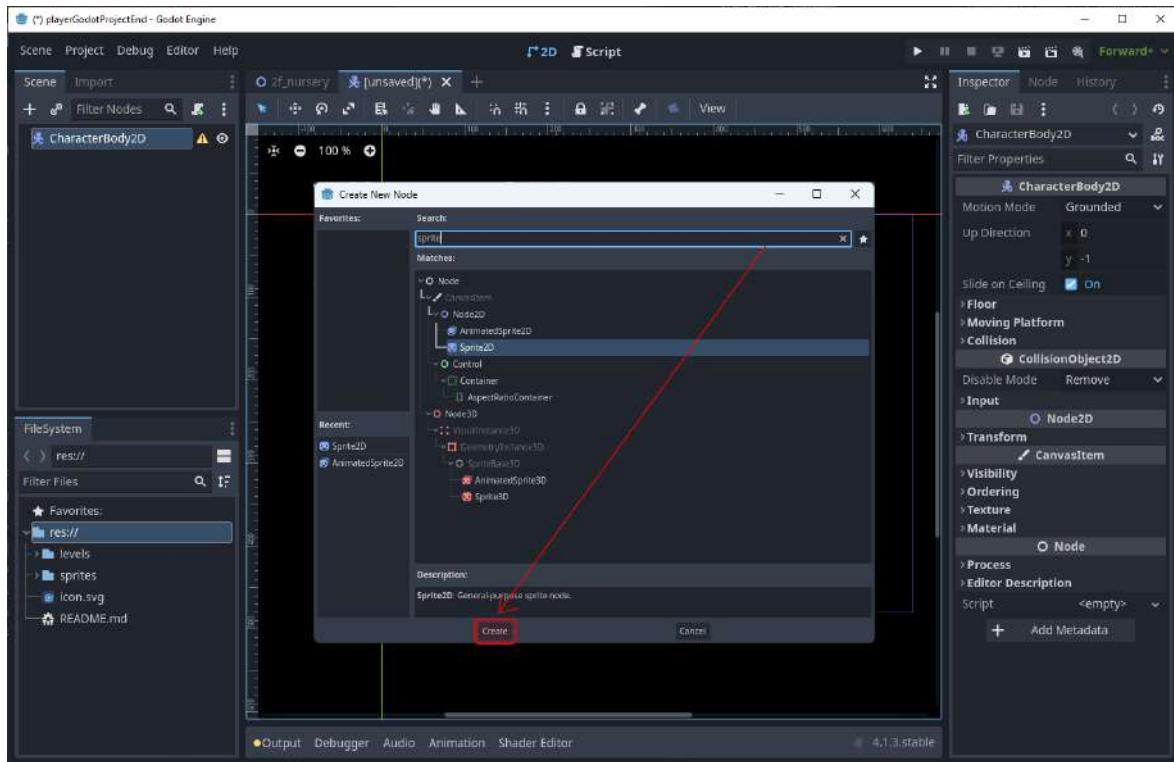
Aggiungiamo un nodo **Sprite2D** come figlio di **Player** cliccando sul pulsante con il **+**



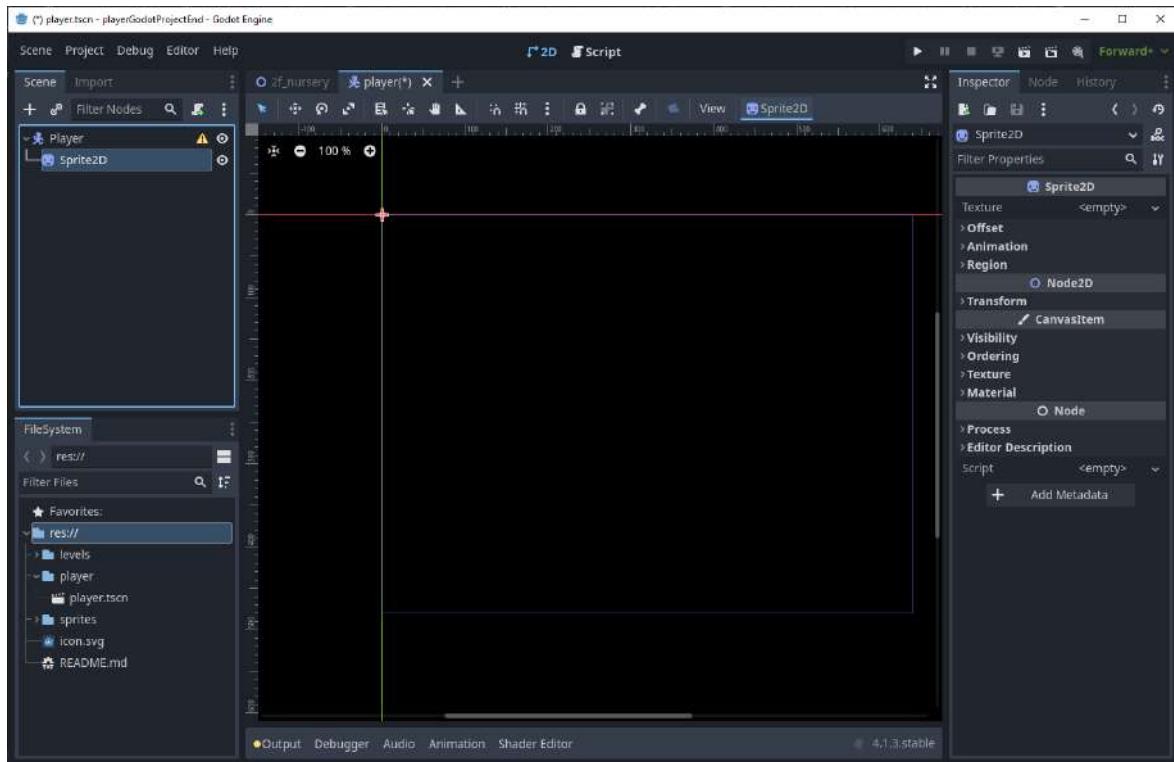
cercando e selezionando il nodo **Sprite2D**



e cliccando su **Create**



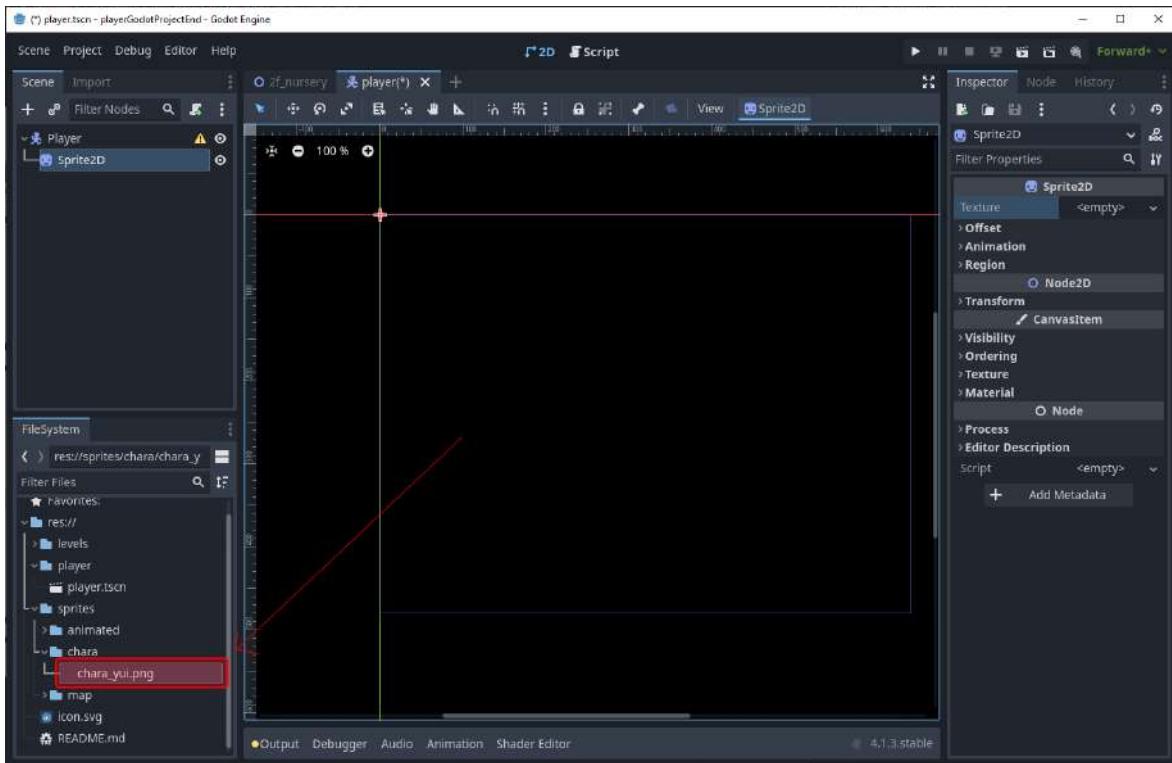
per aggiungere il nodo **Sprite2D** come figlio del nodo **Player**



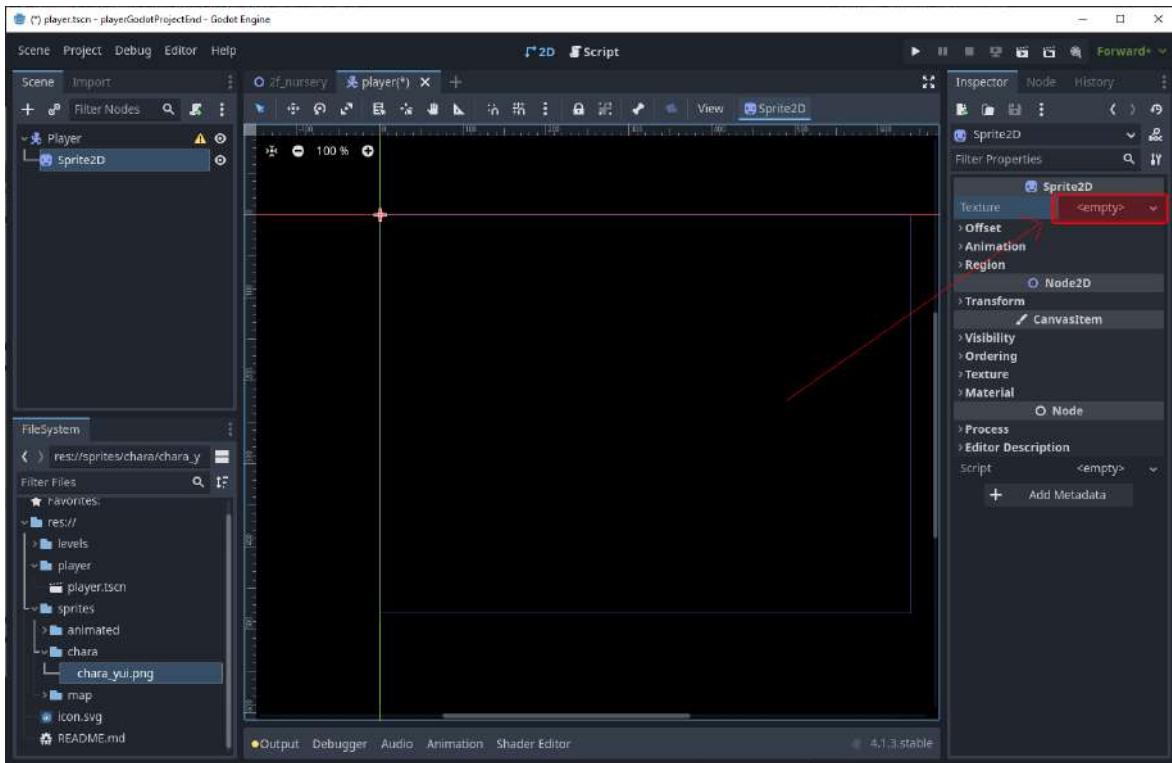
La classe **Sprite2D** è un nodo che visualizza una texture 2D. La texture visualizzata può essere una regione di un texture atlas più grande o un frame di uno spritesheet.

Trasciniamo ora lo spritesheet del player del nostro videogioco, dalla cartella del

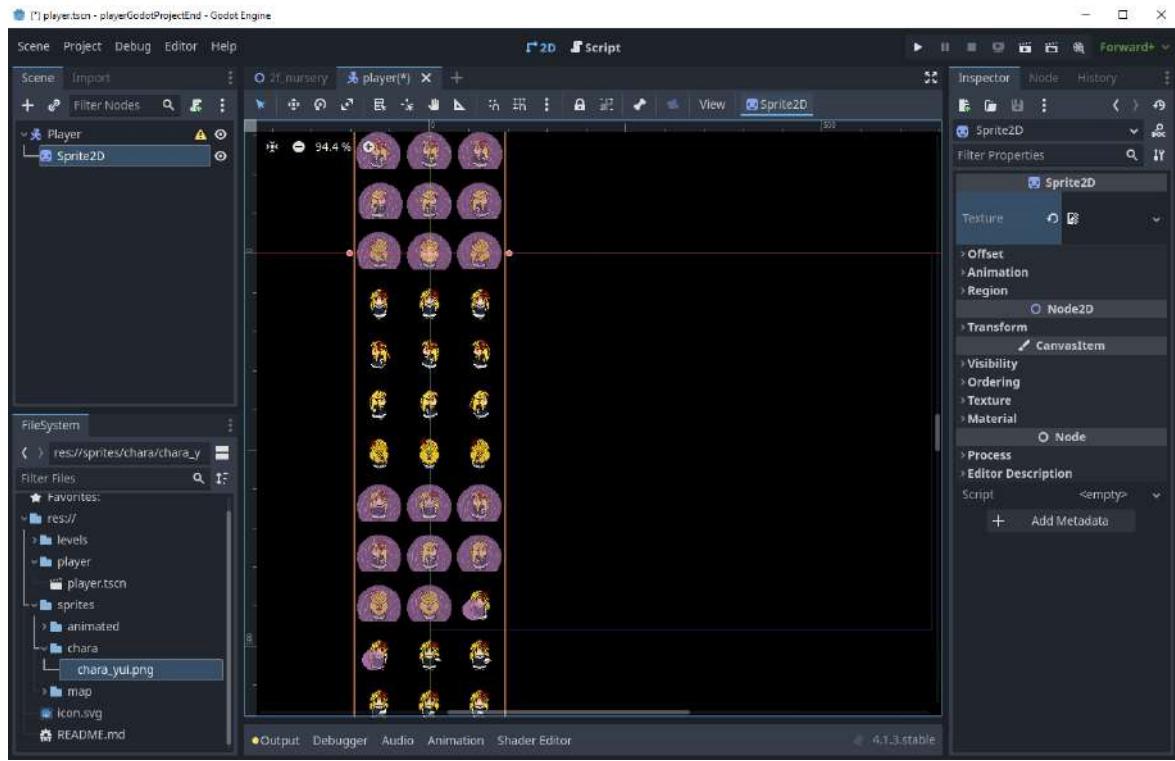
FileSystem dock `res://sprites/chara/chara_yui.png`



fino alla voce `<empty>` di `Texture` dell'inspector dock a destra

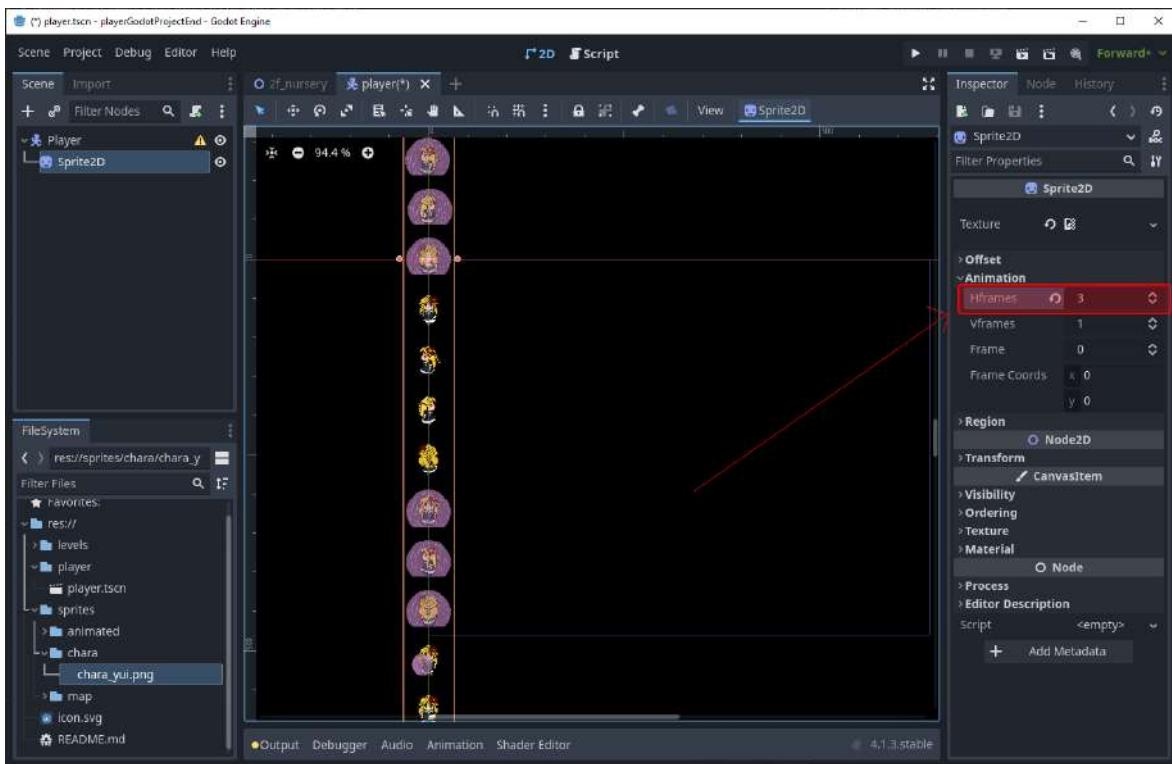


così facendo, dovremmo finalmente vedere una serie di immagini comparire a schermo

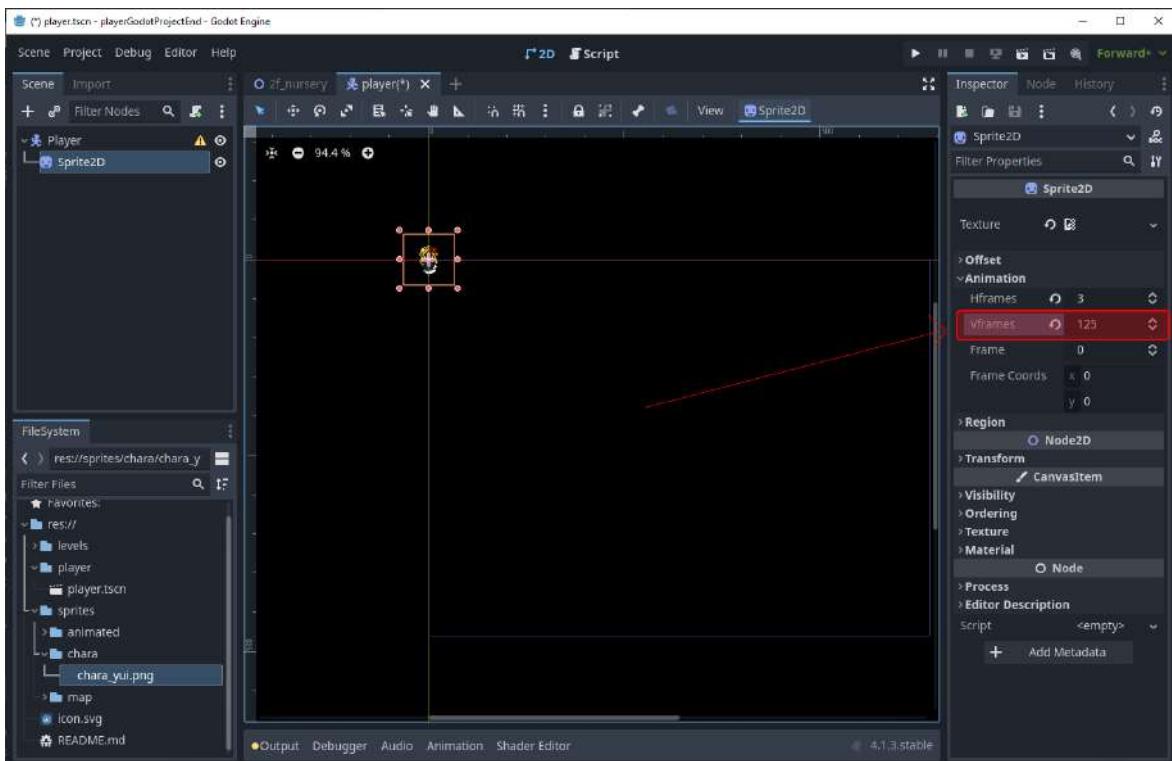


Quello che stiamo utilizzando per importare la texture del nostro personaggio è uno *spritesheet*, ovvero, una griglia al cui interno sono presenti diversi disegni (detti "sprite") che rappresentano il personaggio e che, riprodotti in successione, formano un'animazione di quest'ultimo.

Per poter però vedere a schermo una sola sprite alla volta, invece che tutte insieme, sotto la voce **Animation** del nostro Inspector dock, impostiamo il valore di **Hframes** a **3**

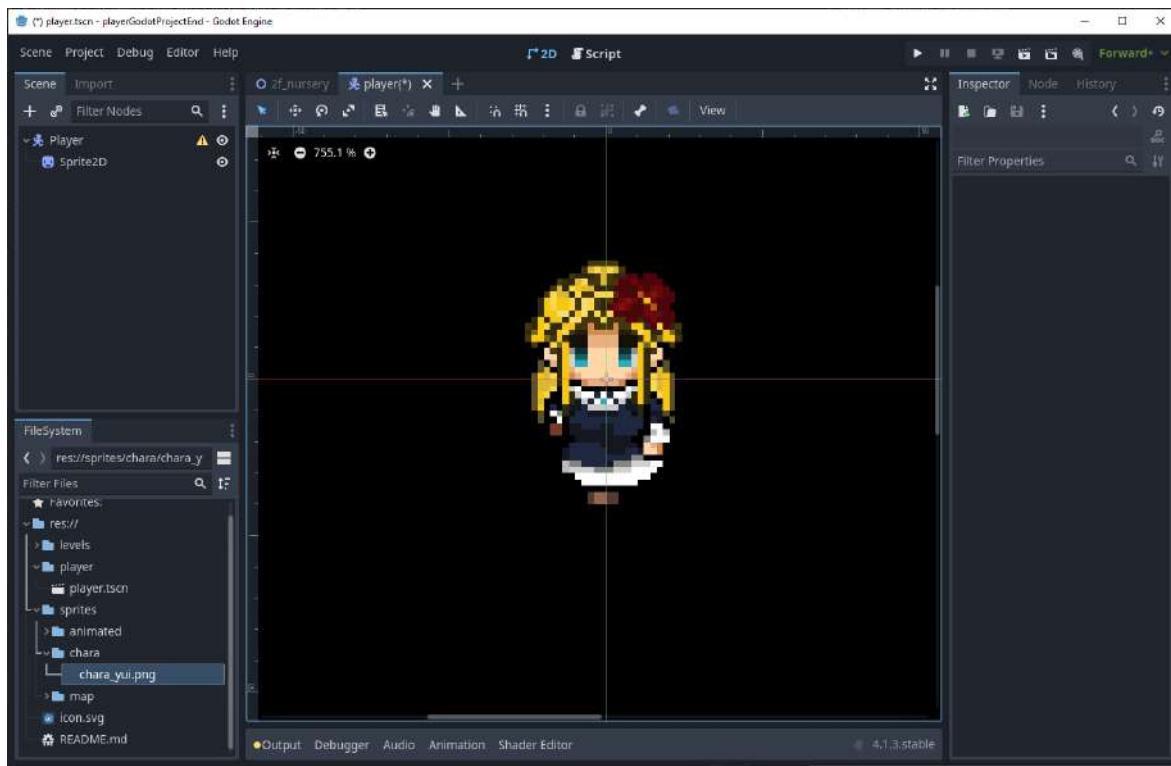


e quello di **Vframes** a **125**

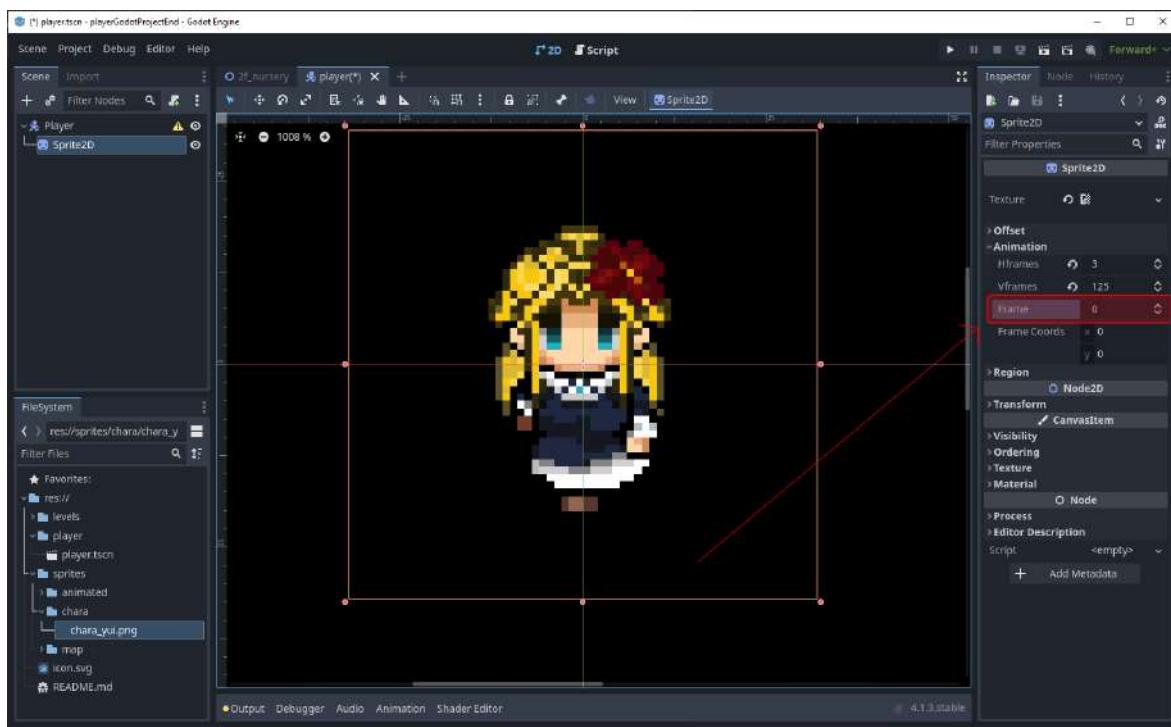


Come si ha già intuito, **Hframes** rappresenta il numero di colonne dello spritesheet, mentre **Vframes** il numero di righe (difatti il nostro spritesheet è formato da 3x125 sprite).

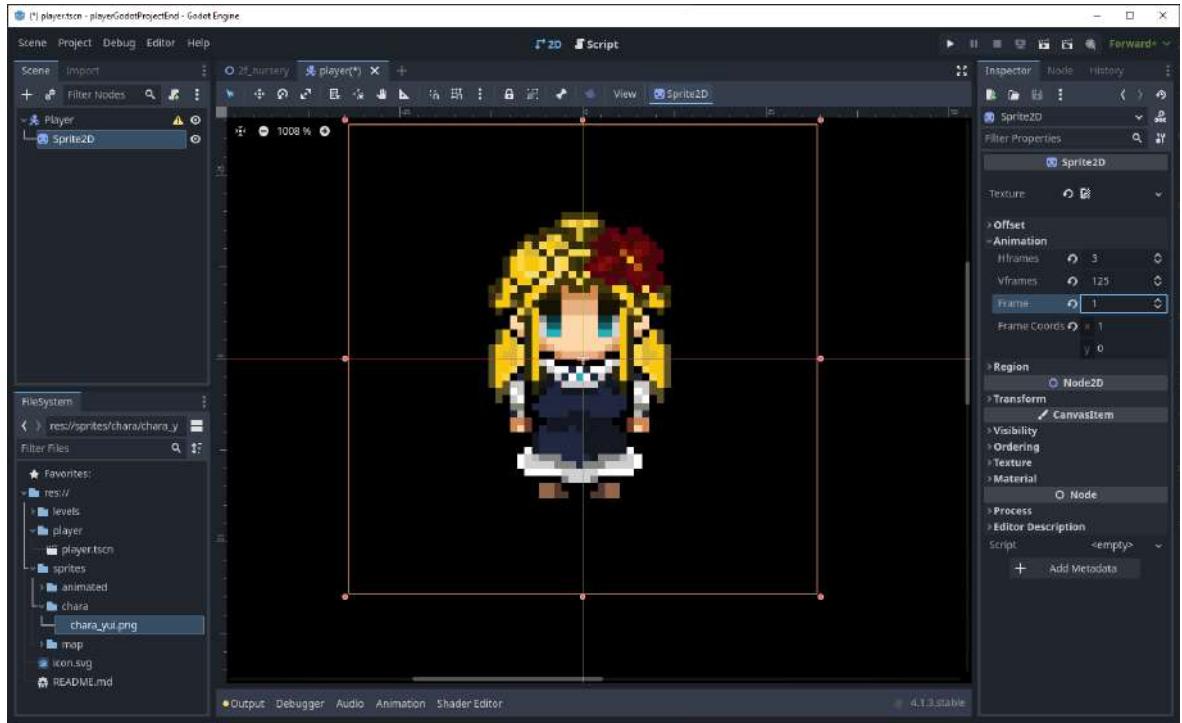
Ora che abbiamo aggiunto una texture alla scena, il nostro personaggio inizia a prendere forma!



Per cambiare il frame visualizzato basta cambiare il valore dell'opzione **Frame** dell'Inspector dock.

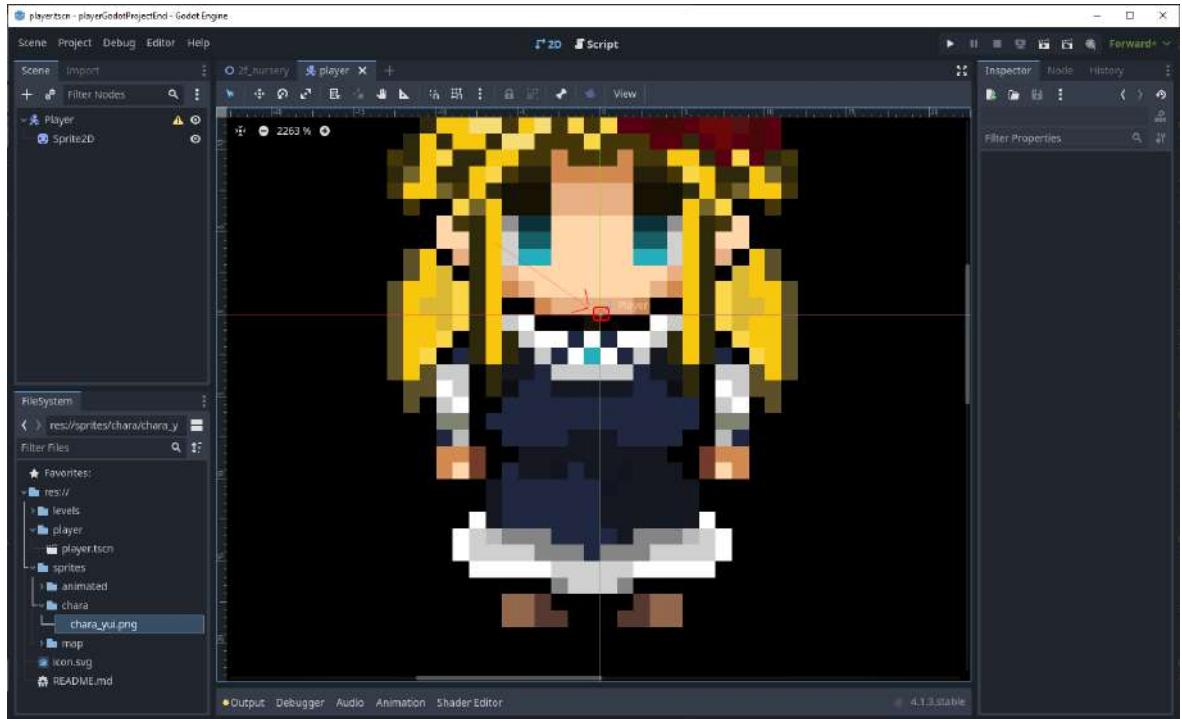


Proviamo dunque ad impostarlo su **1**



come possiamo vedere, il nostro personaggio ha cambiato posa.

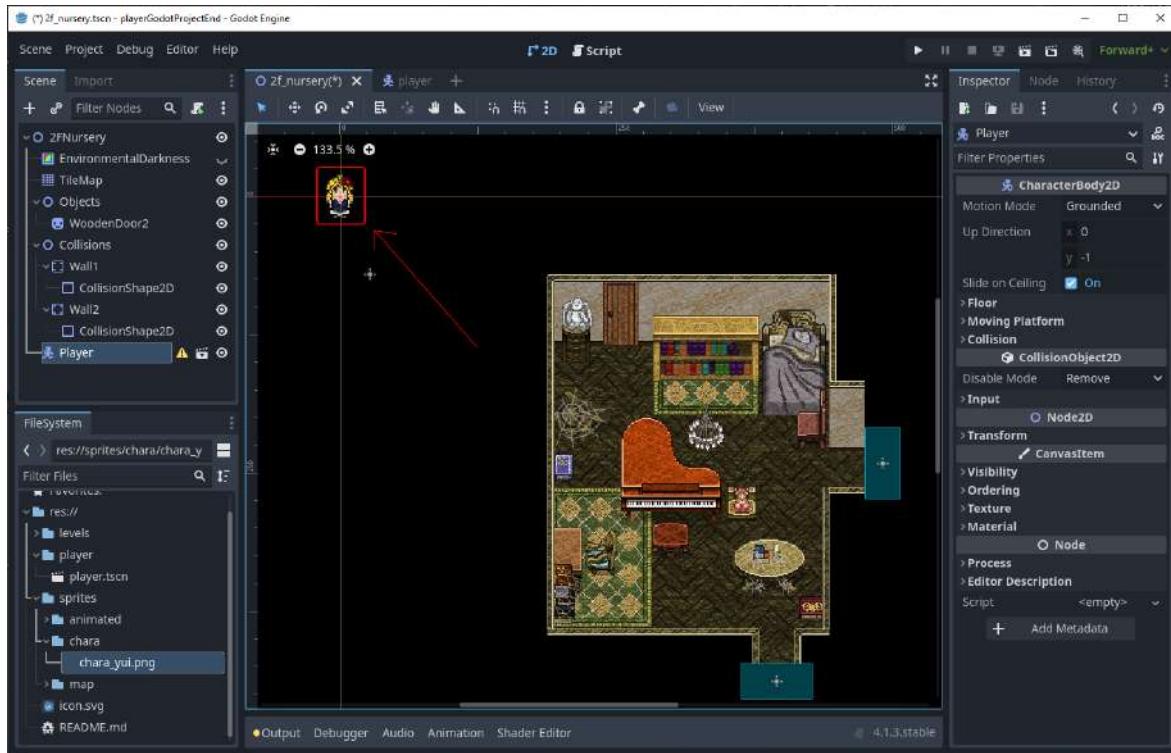
Un altro aspetto a cui dobbiamo prestare attenzione, è il punto di origine della nostra scena **Player**, ovvero, il punto di intersezione dei 4 assi.



Al momento, il nostro personaggio ha la linea di origine all'altezza della gola. Se

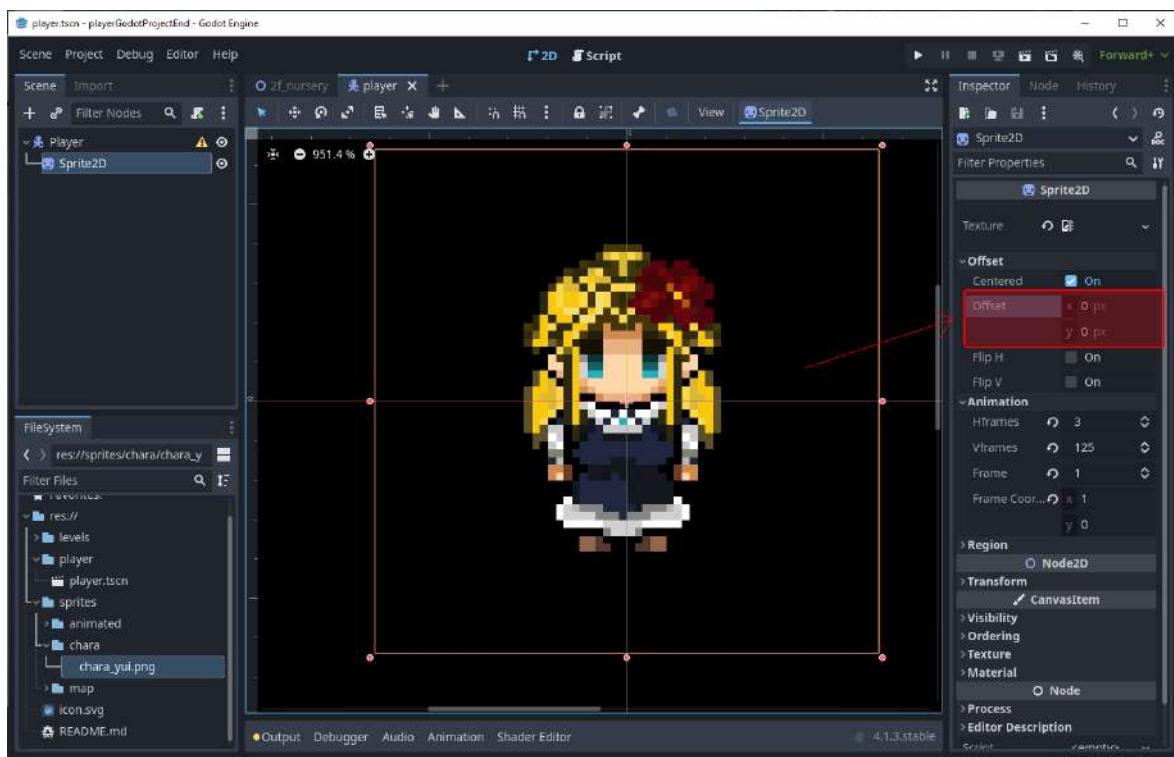
pensassimo a questa come al pavimento, è come se il player avesse tutto il corpo tranne la testa sottoterra.

Un modo lampante per capire che la linea di origine non è impostata in modo corretto, è provare a importare la scena **Player** all'interno di un'altra scena come ad esempio **2f\_nursery**

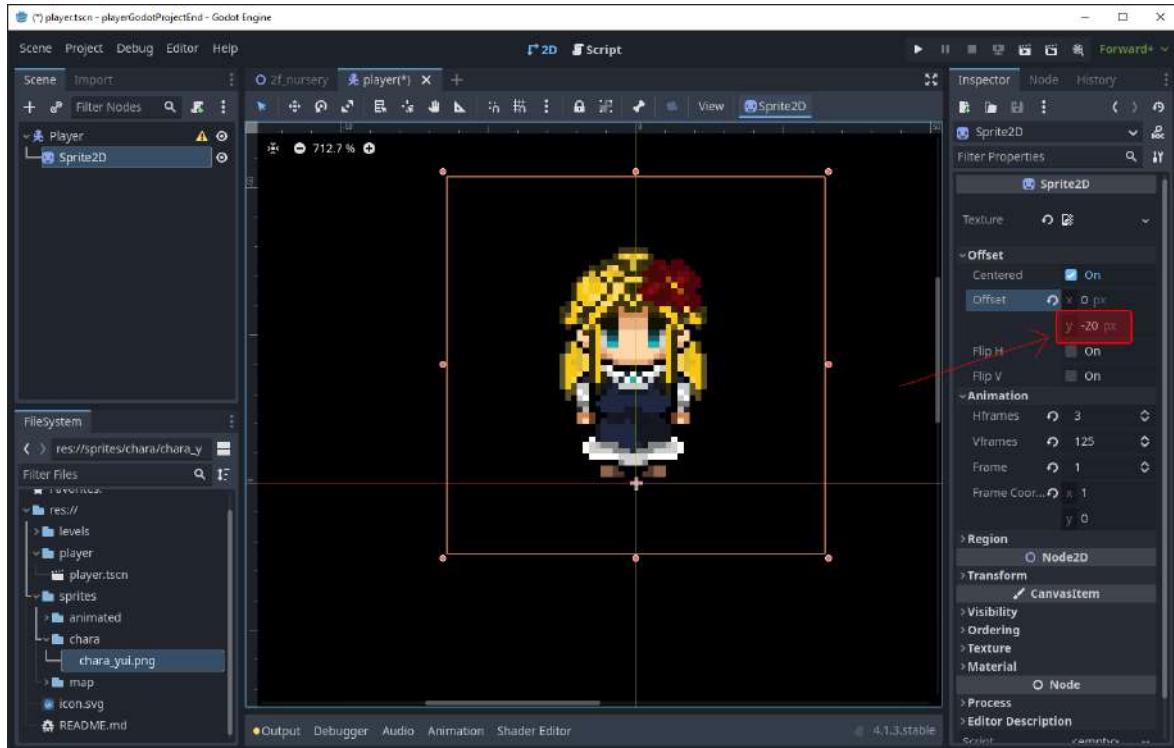


Come visibile a colpo d'occhio, c'è chiaramente qualcosa che non va.

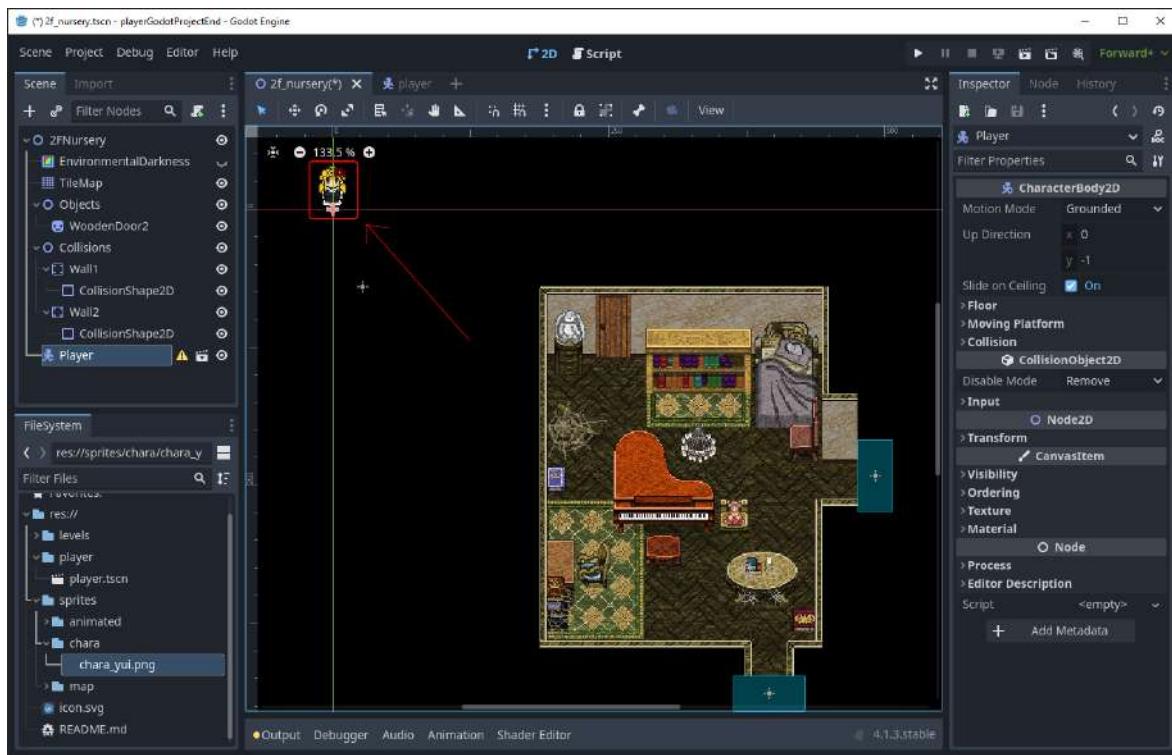
Per risolvere questo problema, dobbiamo cambiare i valori dell'**Offset** sotto la voce **Offset** dell'Inspector dock.



Impostando il valore della **y** a **-20**, infatti, il player avrà i piedi sulla nostra linea di origine, come è giusto che sia.

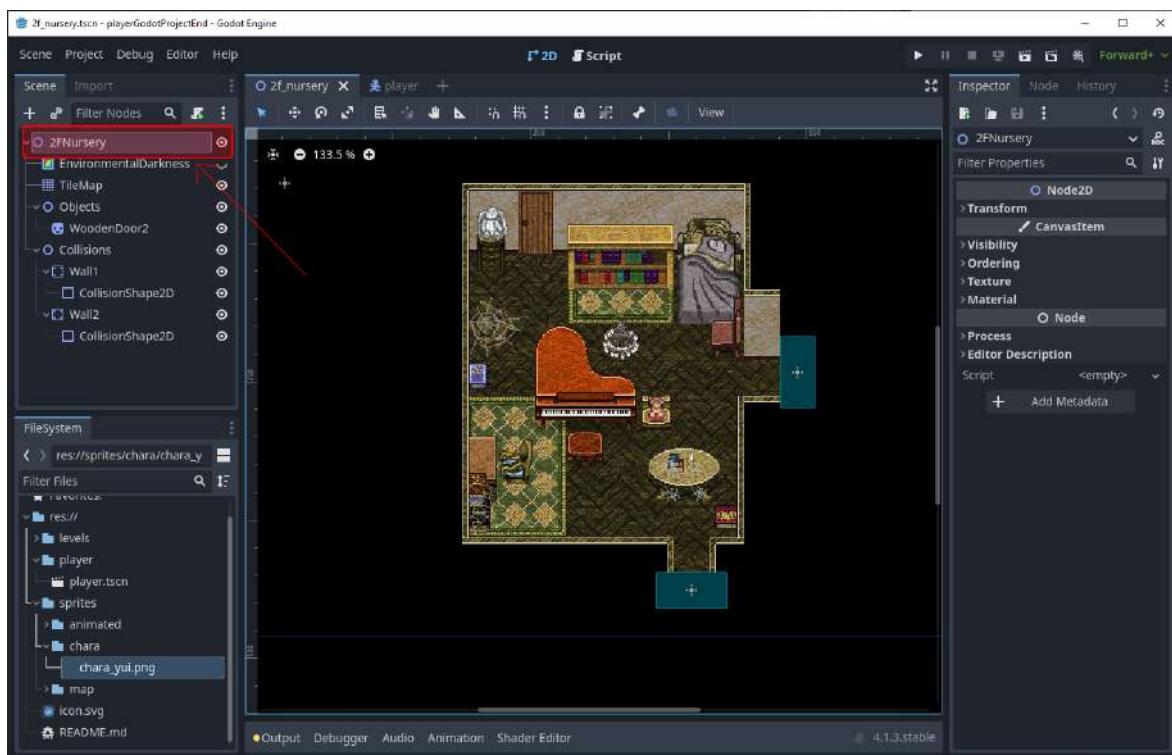


La conferma viene anche dalla prova di import del player nella scena **2f\_nursery**

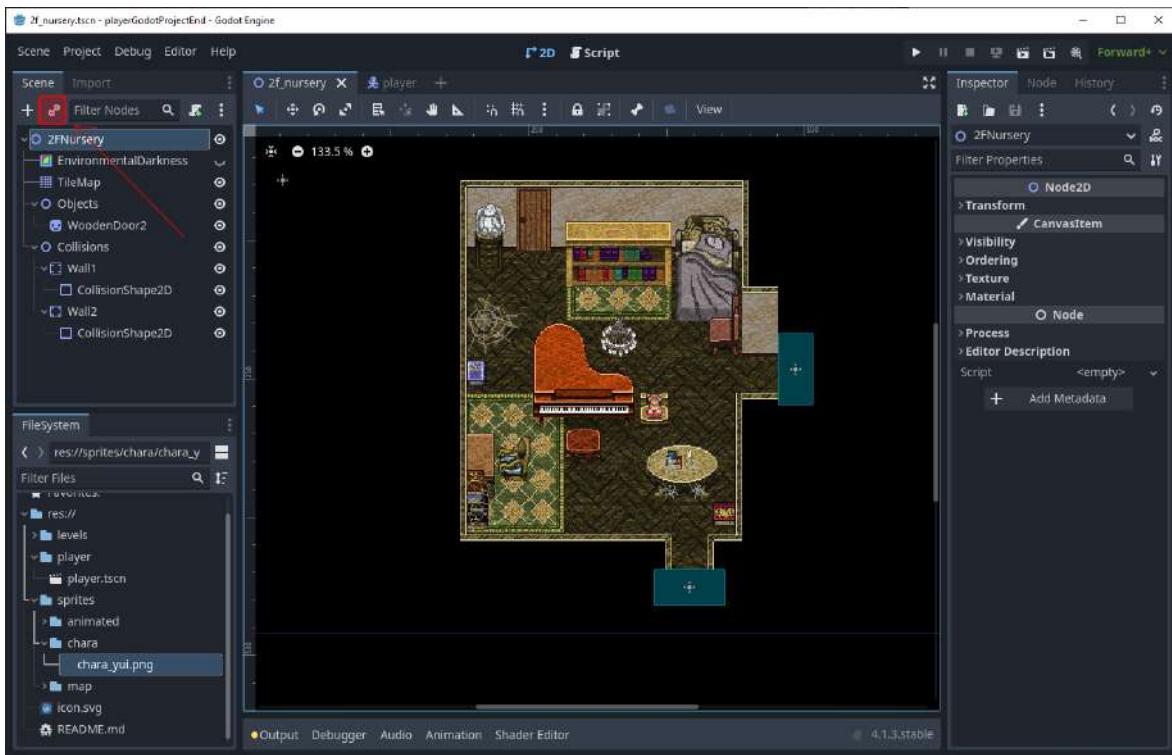


### 2.4.3 Muovere il Player

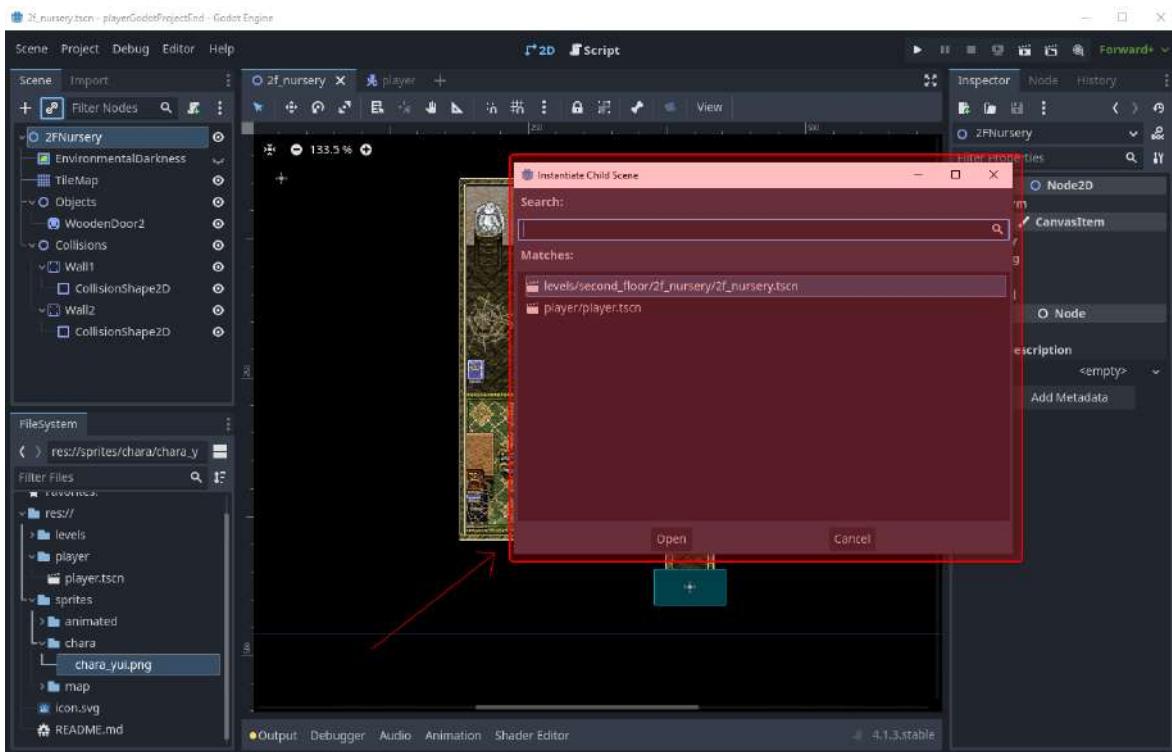
Per importare la nostra scena **Player** in quella **2f\_nursery**, torniamo nella tab della scena **2f\_nursery** e clicchiamo sul nodo padre **2FNursery**, in modo che la scena **Player** venga istanziata come suo nodo figlio.



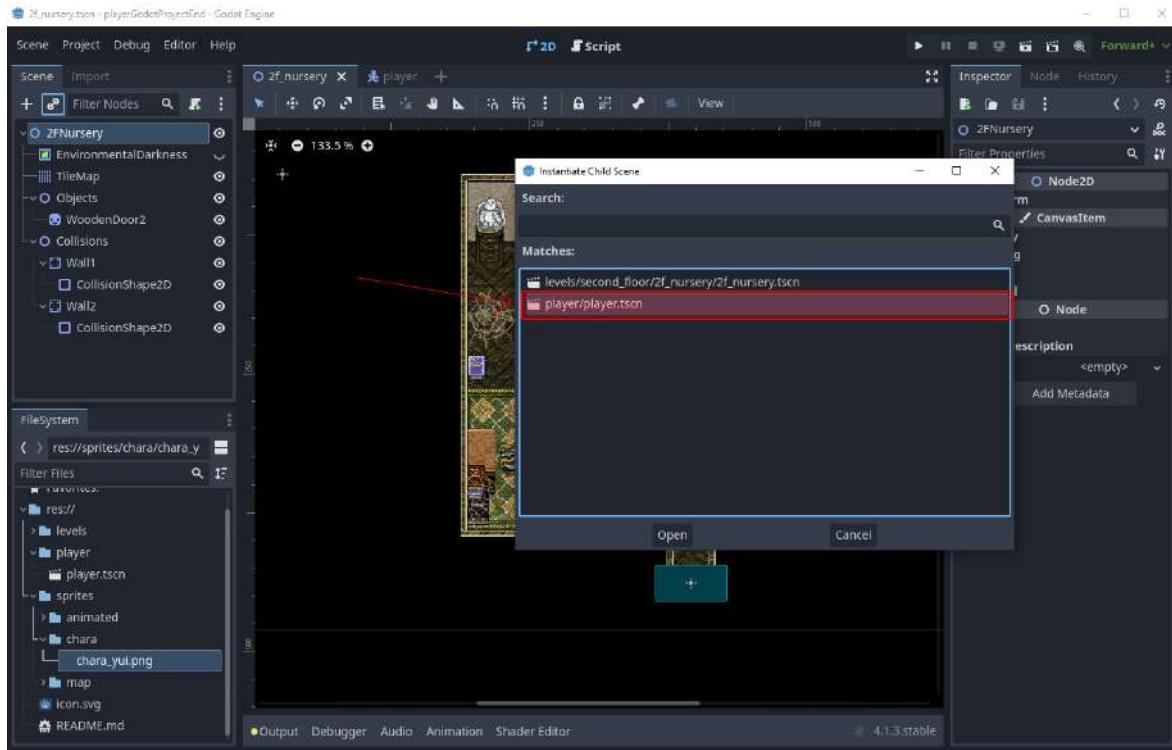
clicchiamo sull'icona della catena



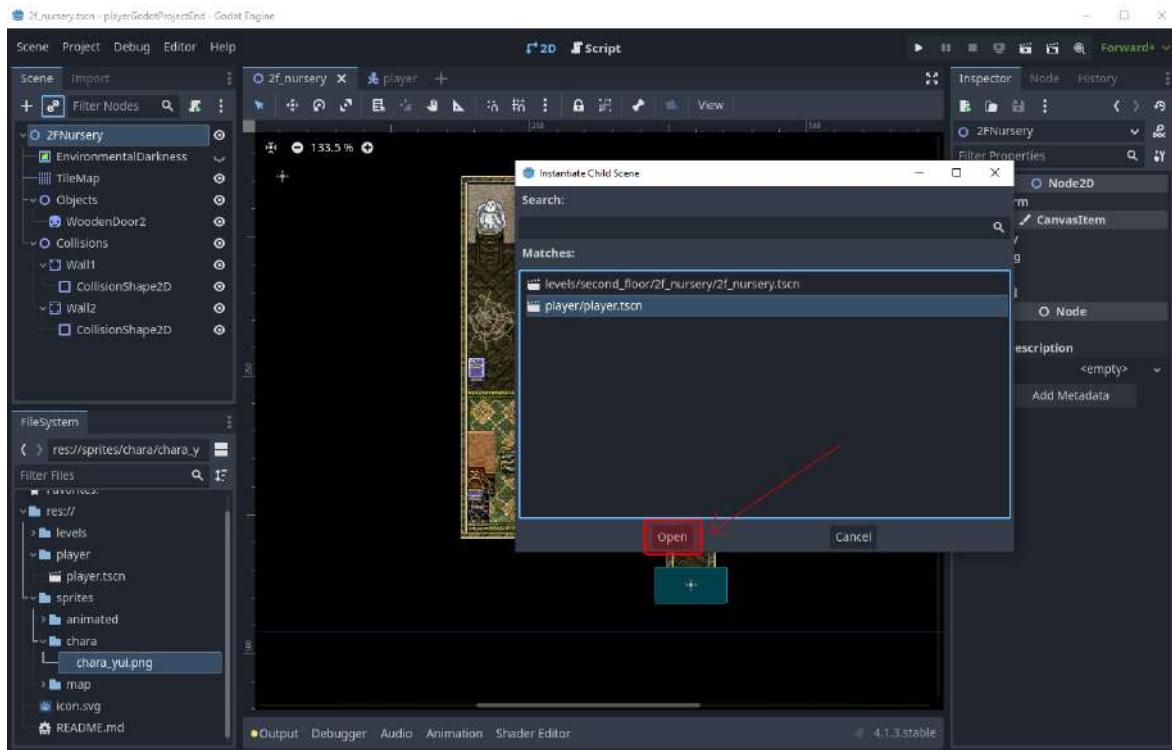
apparirà una finestra in cui sono elencate tutte le scene che abbiamo creato fino a questo momento.



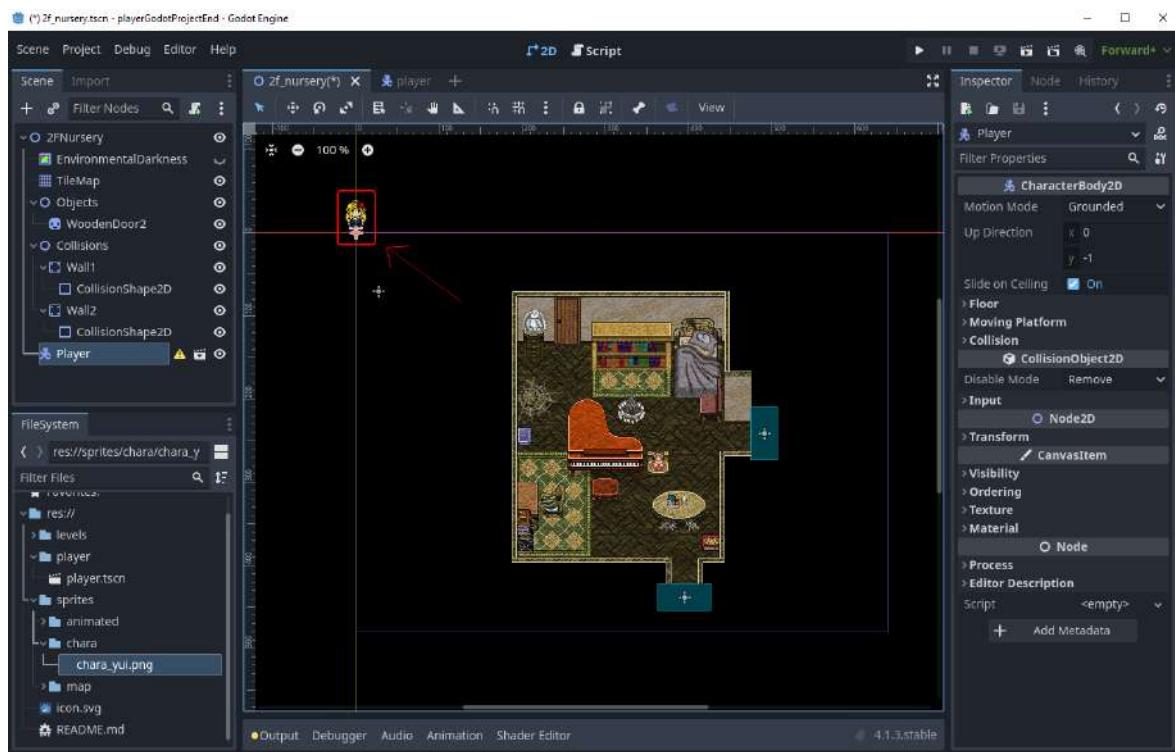
selezioniamo la scena `player/player.tscn`, lì dove `.tscn` è l'estensione che Godot utilizza per salvare le scene



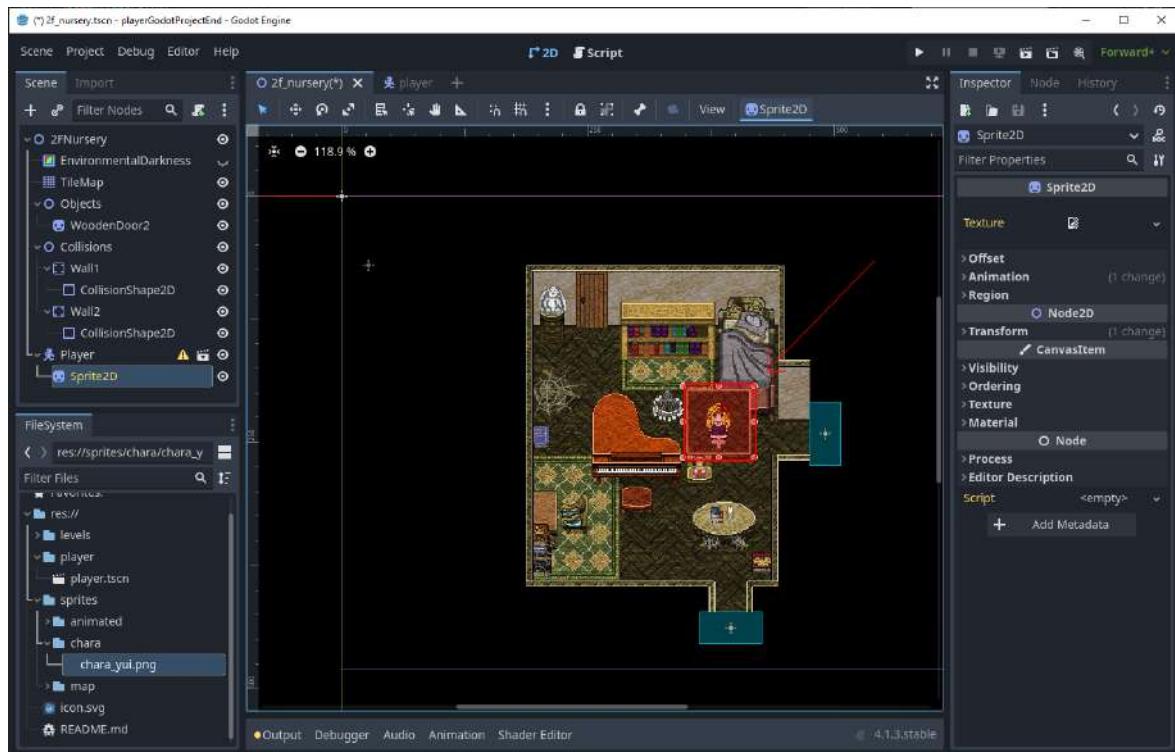
e clicchiamo su **Open**



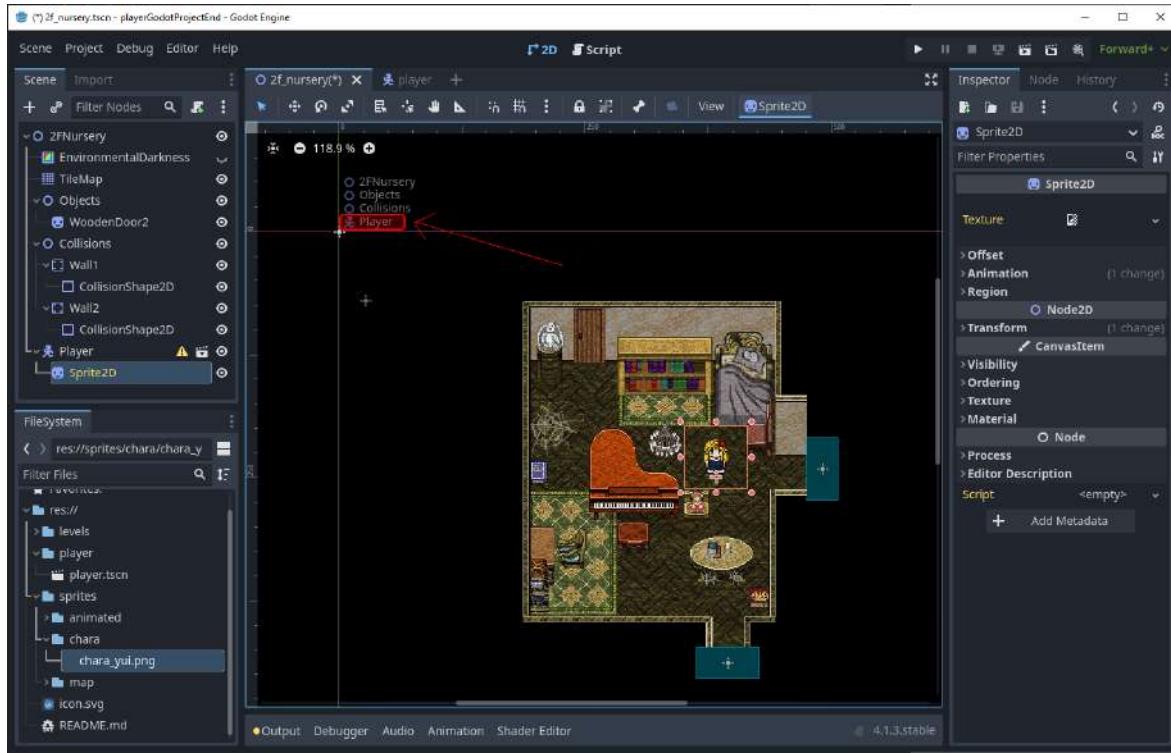
Il nostro player verrà così correttamente importato all'interno della scena [2f\\_nursery](#)



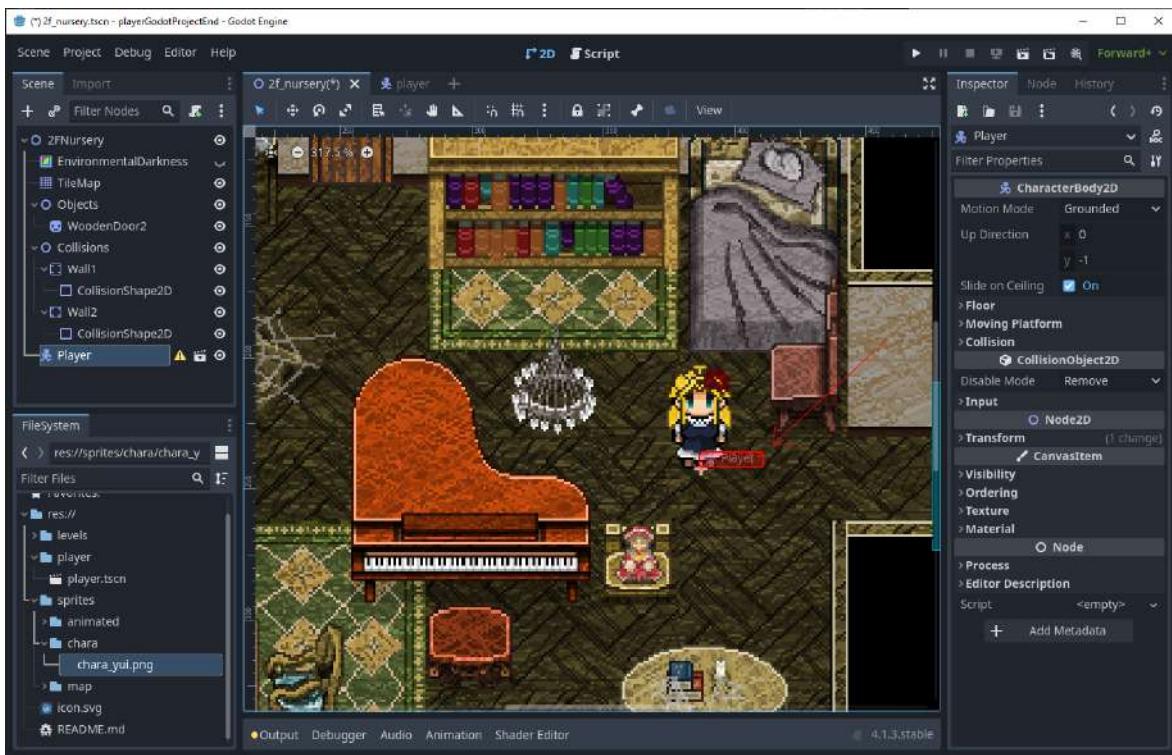
Se provassimo a trascinare con il mouse il nostro giocatore all'interno della stanza, così come faremmo con qualsiasi file sul nostro desktop



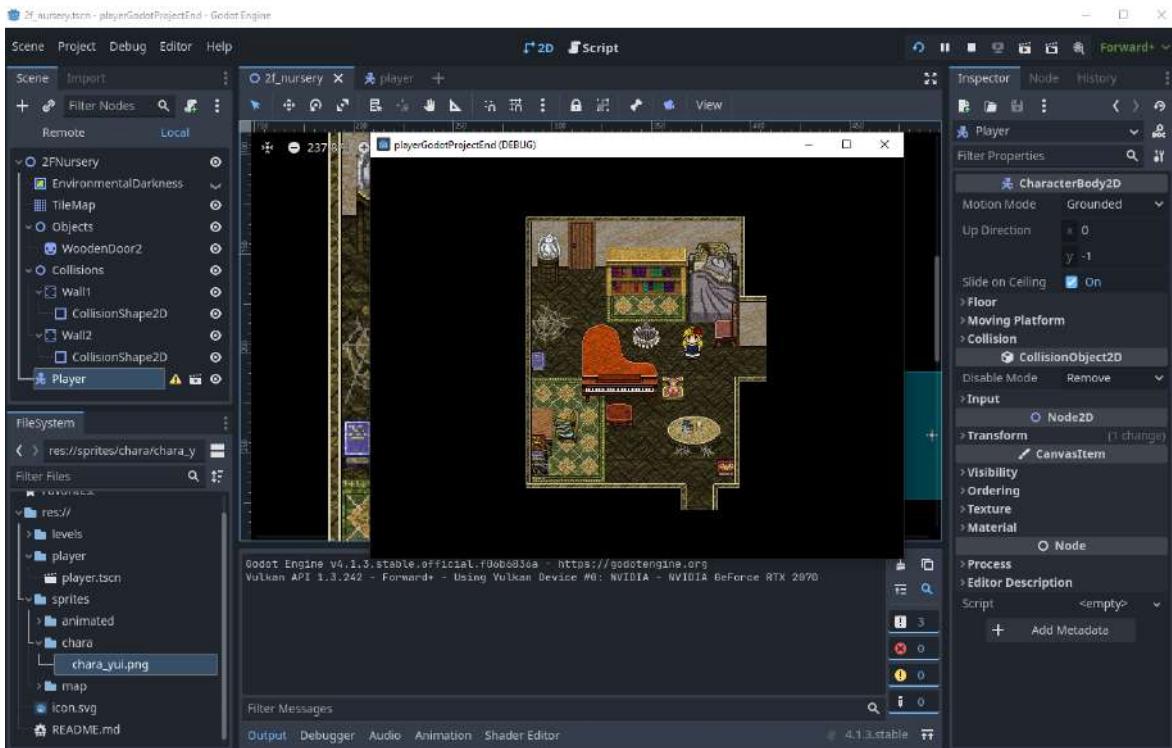
potrebbe succedere di spostare solamente parte della scena generale, come in questo caso è successo con la **Sprited2D** del nostro giocatore; mentre la scena padre vera e propria potrebbe essere ancora rimasta ancorata nell'origine



Per ovviare a questo problema, è consigliabile spostare le scene con il mouse mentre si sta tenendo premuto il tasto **ALT**. Solo in questo modo si è sicuri di star spostando l'intera scena padre, con tutti i suoi nodi figlio.



Adesso che il nostro player è all'interno della stanza, facendo partire il gioco ci rendiamo conto di un altro problema



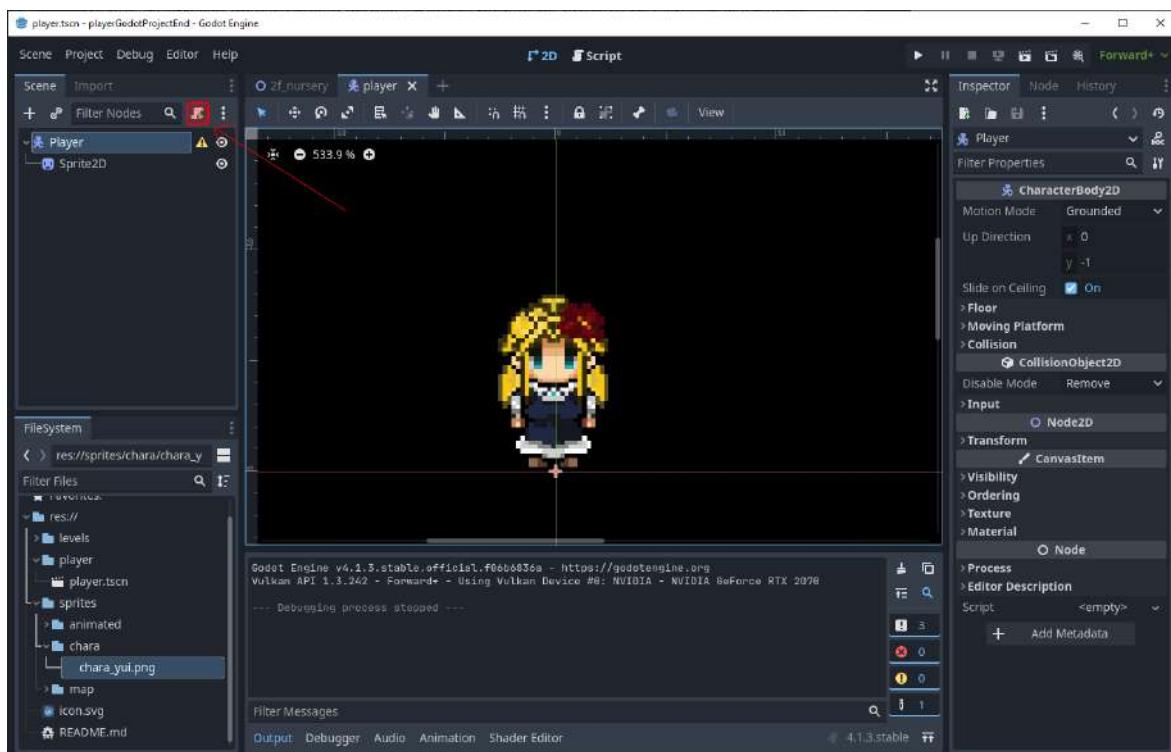
non abbiamo alcun modo di muovere il giocatore.

È qui che entra finalmente in gioco anche lo **scripting**: il coniglio dentro il cilindro che "fa muovere" tutti gli oggetti di un videogioco.

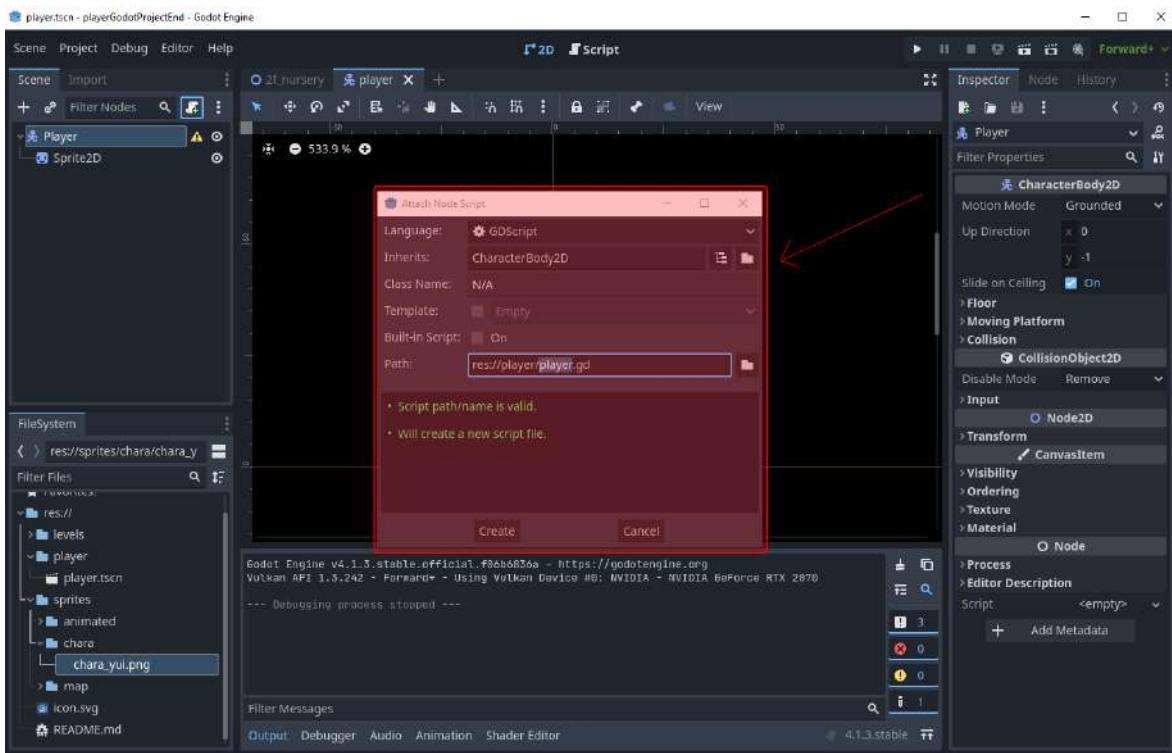
Per far muovere il nostro personaggio, per prima dobbiamo associare un file di script alla nostra scena **Player**. Per fare ciò, dopo essere ritornati nella tab della scena **Player** e aver cliccato sul nodo **Player**, clicchiamo sull'icona a forma di pergamena con un segno +

### ! Attenzione

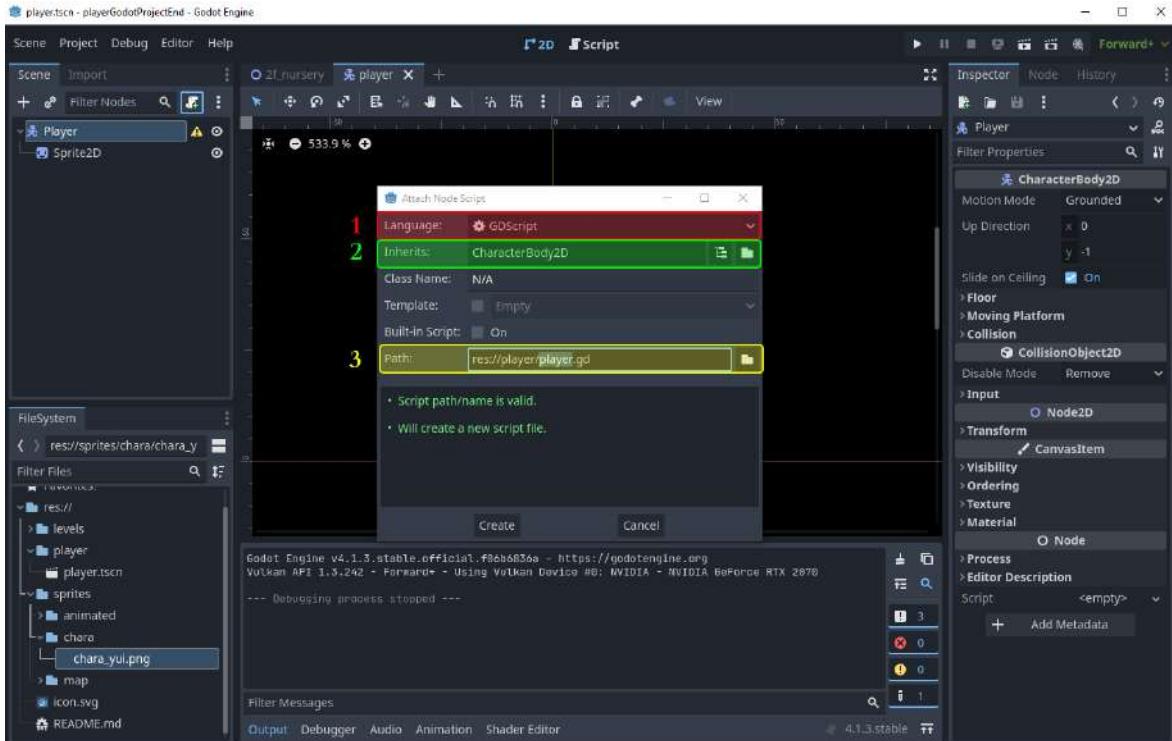
Quando si crea un nuovo script esso viene associato al nodo che si sta selezionando. È per questo motivo che subito dopo essere entrati nella tab della scena **Player** clicchiamo immediatamente sul nodo **Player**.



si aprirà la finestra di creazione di un nuovo script



in cui si possono modificare il linguaggio di scripting che si desidera utilizzare (1), la classe da cui il nuovo script eredita (2), la cartella in cui salvare il file di script e il nome del file stesso (3) e altri parametri



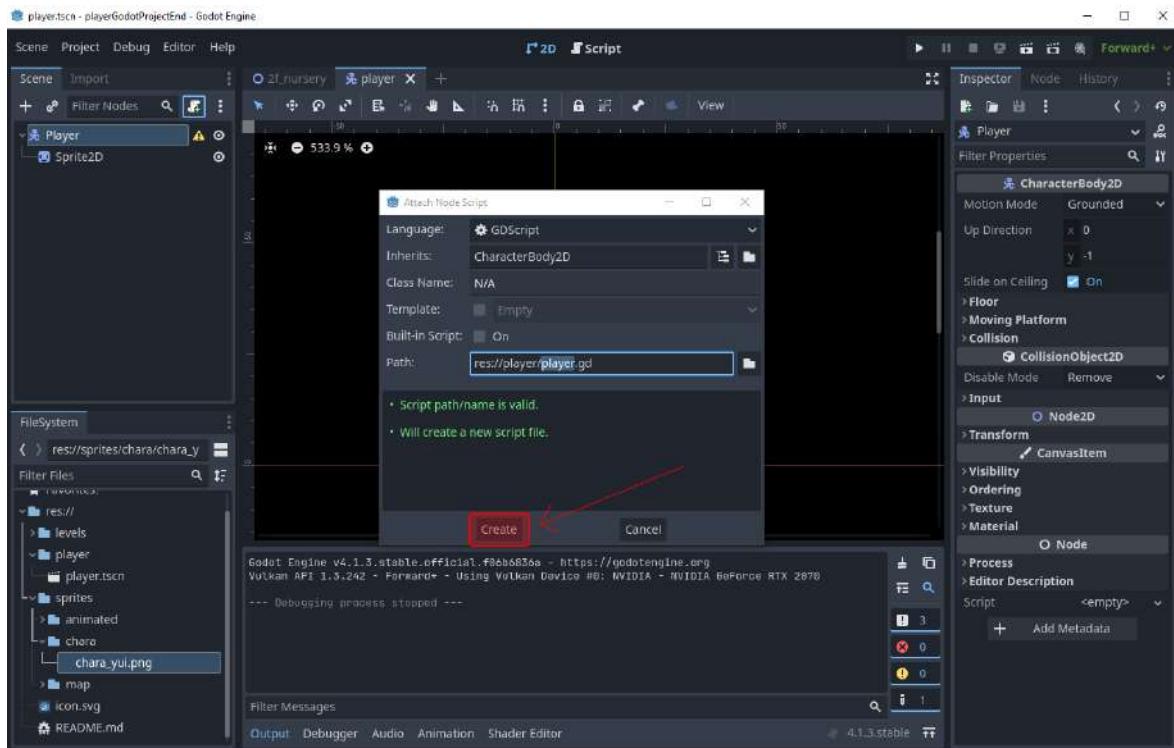
Nonostante Godot offre quattro linguaggi di programmazione (GDScript, C#, C e C++), che possono anche essere mischiati tra loro, noi utilizzeremo solamente

## GDScript.

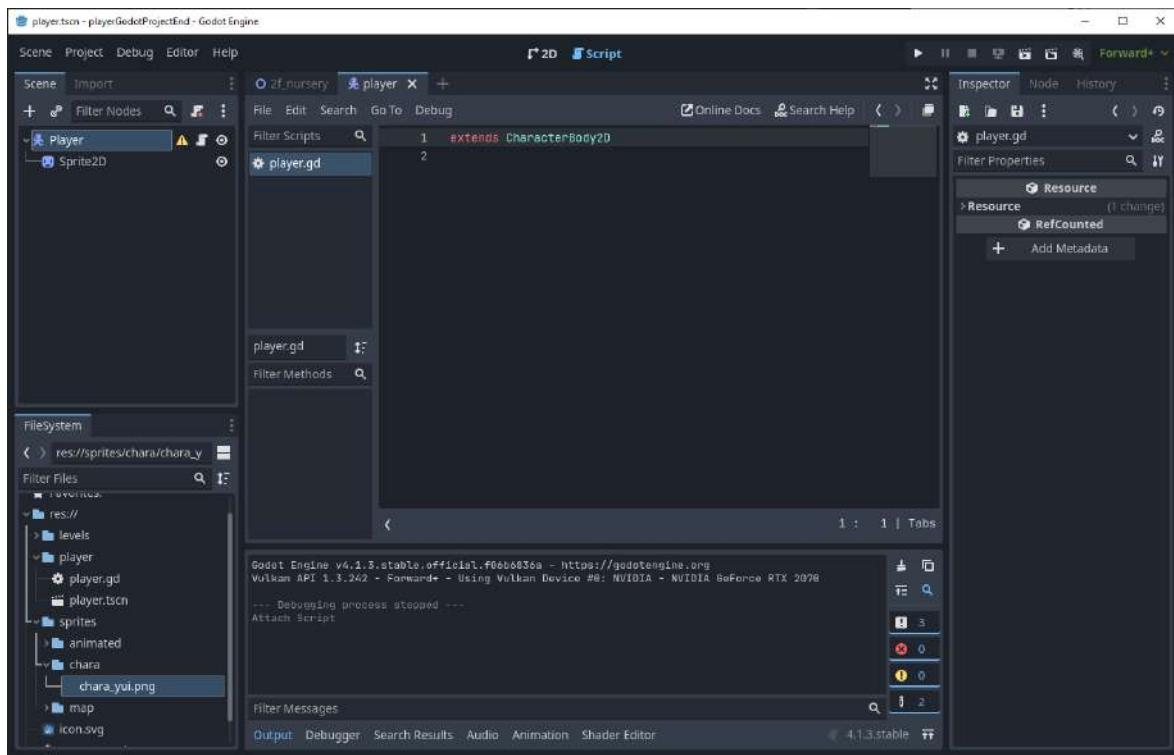
**GDScript** è perfetto per chi ha poche esperienze di programmazione: è stato pensato appositamente per Godot e per soddisfare i bisogni degli sviluppatori; ha una sintassi leggera e diretta e, inoltre, la vasta documentazione disponibile, online e non solo, può aiutare il lettore a esercitarsi anche in modo indipendente.

Per quanto riguarda il nome e la cartella di creazione, invece, come visto nella sezione "Organizzazione del progetto", e più precisamente in "**player**", utilizzeremo la cartella **player**.

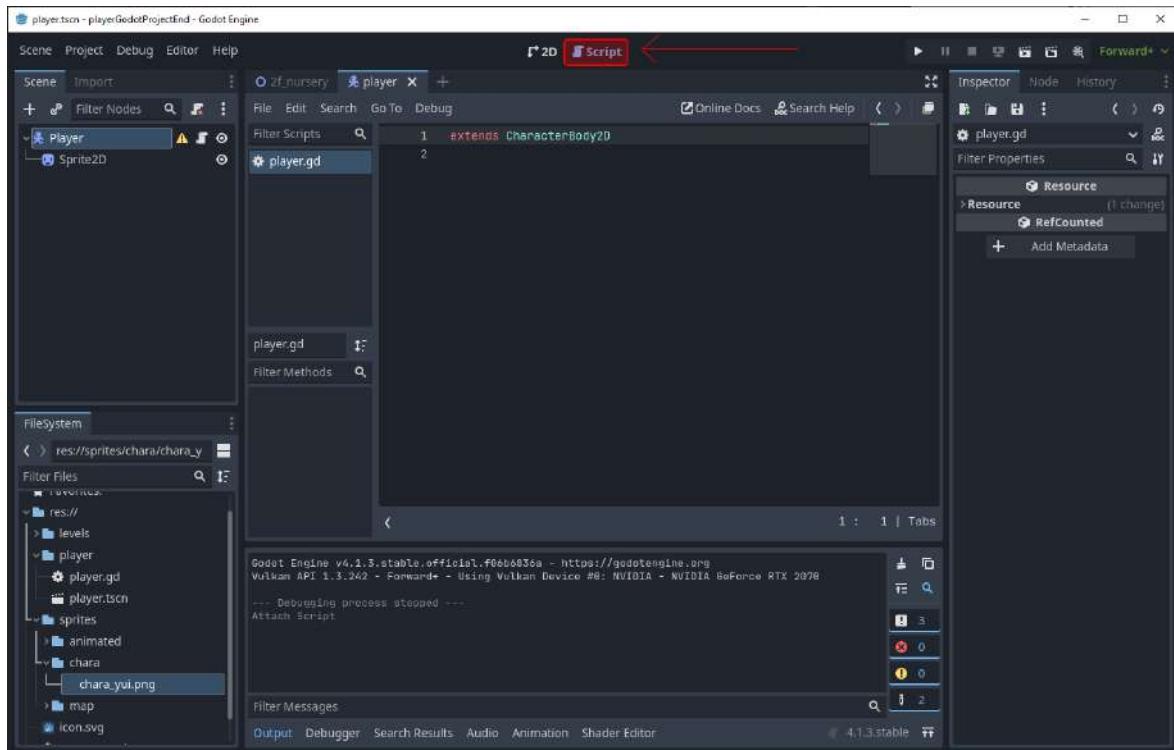
Alla luce di tutto questo, clicchiamo su **Create** per creare il nostro nuovo file di script



Si aprirà in automatico lo **Script Editor** di Godot con il nostro nuovo file di script appena creato.

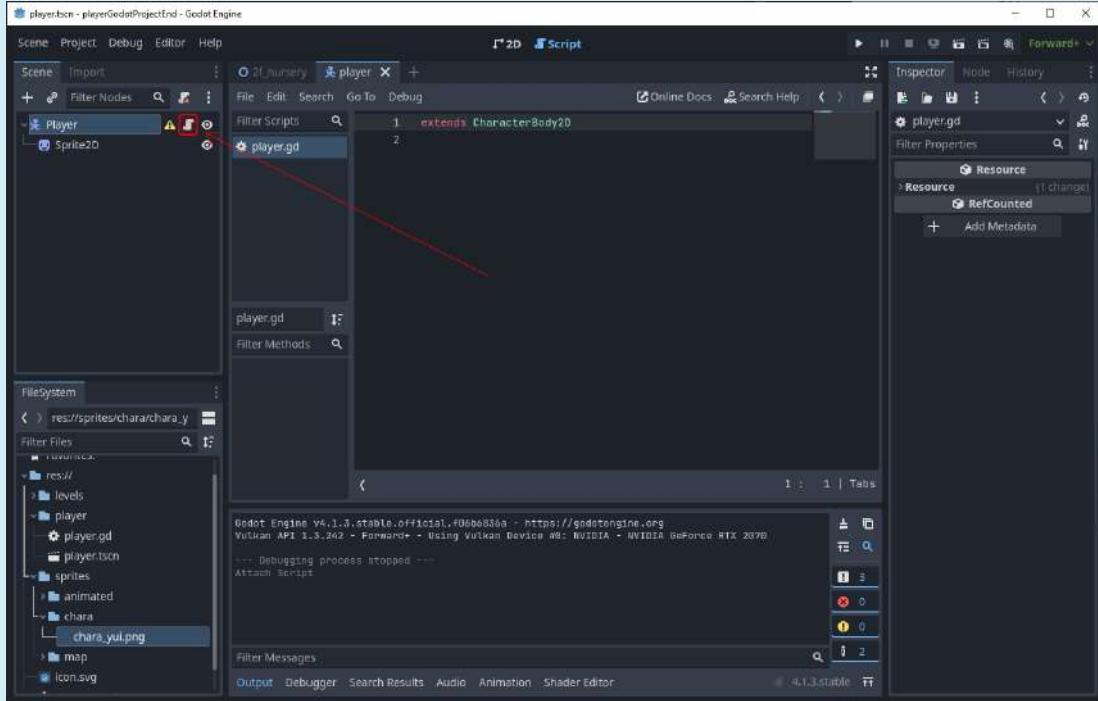


Ricordiamo che lo **Script Editor** può essere aperto tramite l'apposito pulsante in alto



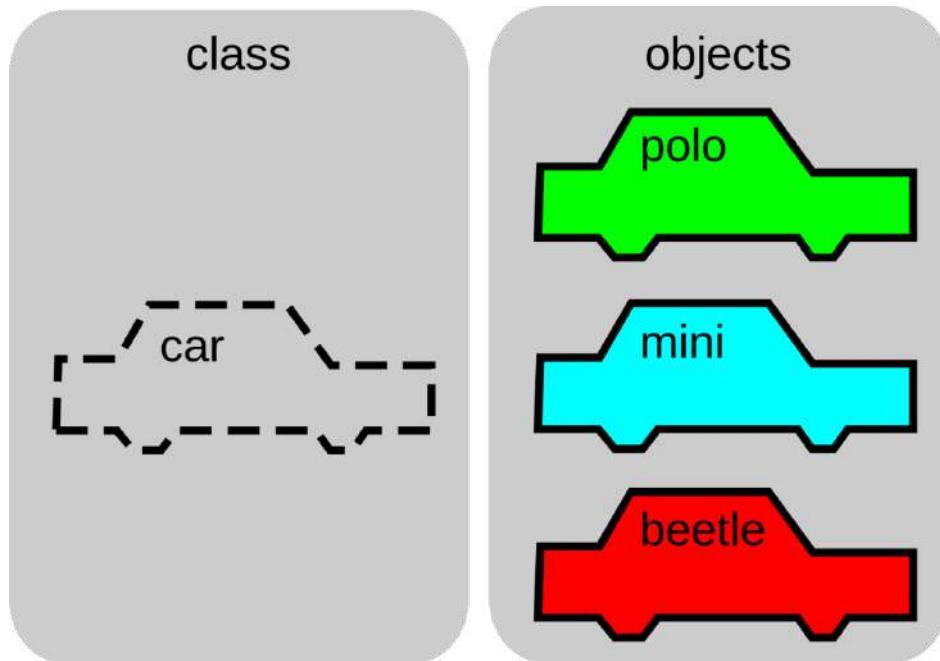
## ! Nota

L'icona della pergamena di fianco al nome di uno script indica che il nodo ha uno script attaccato ad esso



Giunti a questo punto, prima di proseguire con il codice che permette il movimento del giocatore, è opportuno ricordare e/o spiegare alcuni concetti legati alla programmazione.

In Godot ogni file di script è una **classe**. Nella OOP ("object-oriented programming"), una **classe** è progetto per creare **oggetti** (che in Godot sono rappresentati dai nodi), che fornisce anche valori iniziali per lo stato (variabili membro o attributi) e implementazioni di comportamento (funzioni membro o metodi). Una classe è dunque un progetto che definisce la natura di un oggetto futuro. Una **istanza** è un'oggetto specifico creato da una particolare classe. Le classi sono utilizzate per creare e gestire nuovi oggetti, oltre che per supportare l'**ereditarietà** (su Godot gestita mediante la keyword `extends`), ingrediente chiave della OOP e meccanismo di riuso di codice. Ad esempio, un classe **Car** può fungere da progetto per molte istanze diverse di **Car**, come ad esempio gli oggetti **polo**, **mini** e **beetle**. Tutti questi oggetti, seppur avendo caratteristiche diverse come il modello, il colore o il numero di passeggeri, hanno la stessa struttura di base: una carrozzeria, un motore, delle ruote, ecc...



38

Su Godot, si utilizza la keyword `class_name` per definire uno script come una classe globalmente accessibile mediante un nome specificato. Nel caso della nostra classe `Player` scriviamo dunque il seguente codice

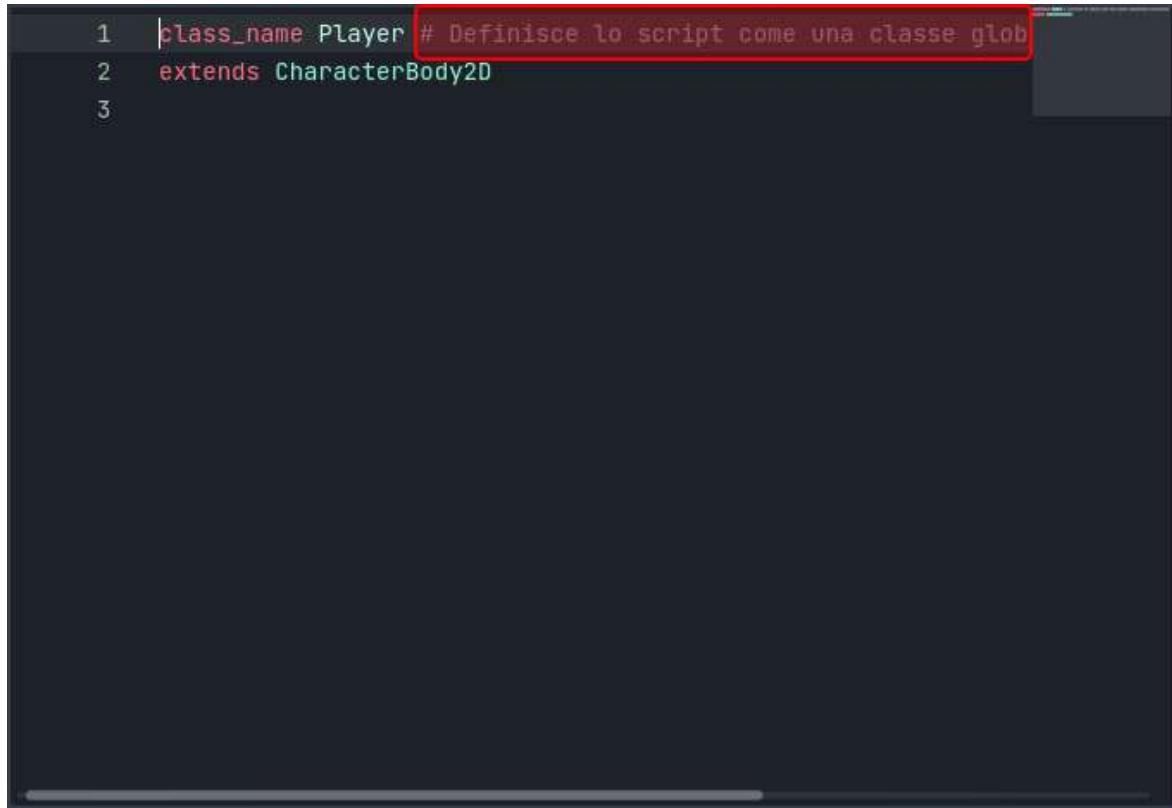
```

1  class_name Player # Definisce lo script come una classe globale
2  extends CharacterBody2D
3

```

<sup>38</sup>Fonte: <https://brilliant.org/wiki/classes-oop/>

Lì dove il testo in grigio preceduto da un `#` si chiama **commento** ed è semplicemente del testo che non viene valutato da Godot ed è quindi privo di significato. Nella programmazione, i commenti sono utili per spiegare cosa fa un determinato pezzo di codice.



```
1 |class_name Player # Definisce lo script come una classe globale
2 |extends CharacterBody2D
3 |
```

Grazie alla keyword `class_name` il nostro script è adesso una classe a pieno titolo. Come già accennato, l'altra parola chiave presente in questo codice, ovvero `extends`



```
1 class_name Player # Definisce lo script come una classe globale
2 extends CharacterBody2D
3
```

è utilizzata da Godot per gestire il concetto di **ereditarietà**.

In OOP, l'**ereditarietà** permette a una nuova classe di *ereditare* appunto tutte le proprietà e i metodi di una classe esistente. Questa feature promuove il riuso di codice e crea relazioni padre-figlio tra le classi.

Le **variabili** membro di una classe sono dette **proprietà**. Nella programmazione, una **variabile** è un valore che può cambiare, a seconda di condizioni o informazioni passati dal programma.

Un **metodo** è un'insieme di linee di codice che definiscono un'azione che un oggetto è in grado di compiere.

Nel nostro caso, ad esempio, con la keyword `extends CharacterBody2D`, GDScript ci sta dicendo che la nostra nuova classe `Player` avrà accesso a tutti i metodi e le variabili della classe esistente `CharacterBody2D`.

## Properties

```

bool floor_block_on_wall [default: true]
bool floor_constant_speed [default: false]
float floor_max_angle [default: 0.785398]
float floor_snap_length [default: 1.0]
bool floor_stop_on_slope [default: true]
int max_slides [default: 4]
MotionMode motion_mode [default: 0]
int platform_floor_layers [default: 0xFFFFFFFF]
PlatformOnLeave platform_on_leave [default: 0]
int platform_wall_layers [default: 0]
float safe_margin [default: 0.08]
bool slide_on_ceiling [default: true]
Vector2 up_direction [default: Vector2(0, -1)]
Vector2 velocity [default: Vector2(0, 0)]
float wall_min_slide_angle [default: 0.261799]

```

## Methods

```

void apply_floor_snap()
float get_floor_angle(up_direction: Vector2 = Vector2(0, -1)) const
Vector2 get_floor_normal() const
Vector2 get_last_motion() const
KinematicCollision2D get_last_slide_collision()
Vector2 get_platform_velocity() const
Vector2 get_position_delta() const
Vector2 get_real_velocity() const
KinematicCollision2D get_slide_collision(slide_idx: int)
int get_slide_collision_count() const

```

Ma tornando al nostro personaggio, per farlo muovere utilizzeremo il metodo `_physics_process(delta)`, ereditato dalla classe `Node`, la quale è da qualche parte nell'albero dell'ereditarietà della classe `CharacterBody2D`.

Il metodo `_physics_process(delta)` viene chiamato durante la fase di elaborazione della fisica del loop principale. Elaborare la fisica significa che il framerate e la fisica stessa sono sincronizzati, e cioè che la variabile `delta` dovrebbe essere costante (`delta` è in secondi)

In Godot, entrambe le funzioni `_process(delta)` e `_physics_process(delta)` sono funzioni di callback che vengono chiamate a ogni frame ma, tuttavia, nonostante questa similarità, hanno scopi completamente diversi e sono sincronizzate con parti differenti del loop cycle di Godot.

La funzione `_process(delta)` viene utilizzata quando si ha bisogno di aggiornare qualcosa a ogni frame, indipendentemente dalla fisica. Dentro questa funzione finisce ogni genere di codice che non richiede una sincronizzazione precisa con il physics engine, come ad esempio:

- animare sprite o elementi della UI
- gestire la logica del videogioco che non riguarda la fisica
- aggiornare stati del gioco non legati alla fisica

Il parametro `delta` rappresenta il tempo trascorso (in secondi) dall'ultimo frame e

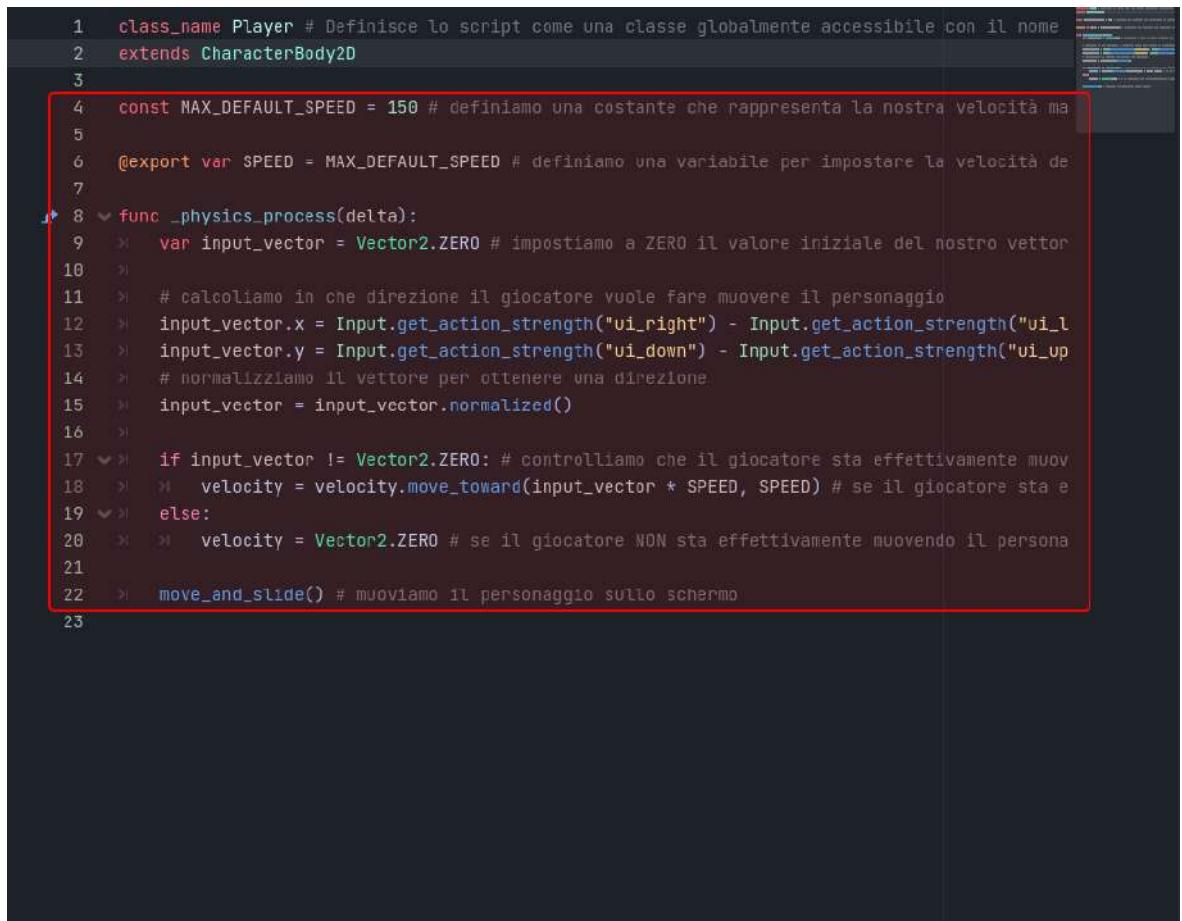
aiuta ad assicurarsi che gli aggiornamenti si verifichino con velocità costante, indipendentemente dalle variazioni del framerate.

D'altro canto, la funzione `_physics_process(delta)` è specificatamente progettata per essere eseguita in sincronia con il physics engine. Questo significa che viene chiamata a intervalli regolari, assicurando una simulazione della fisica coerente. Questa funzione viene dunque utilizzata per:

- muovere oggetti (`RigidBody`, `KinematicBody`, `CharacterBody2D`, ecc...) utilizzando il physics engine
- controllare interazioni fisiche come le collisioni
- applicare forze o impulsi a oggetti fisici

Similmente a `_process(delta)`, il parametro `delta` indica l'intervallo di tempo fisso che c'è tra due frame fisici e assicura un comportamento coerente della fisica.

Poiché dunque il nostro obiettivo è quello di *muovere* un oggetto, ci rivolgiamo a `_physics_process(delta)` e scriviamo le seguenti righe di codice



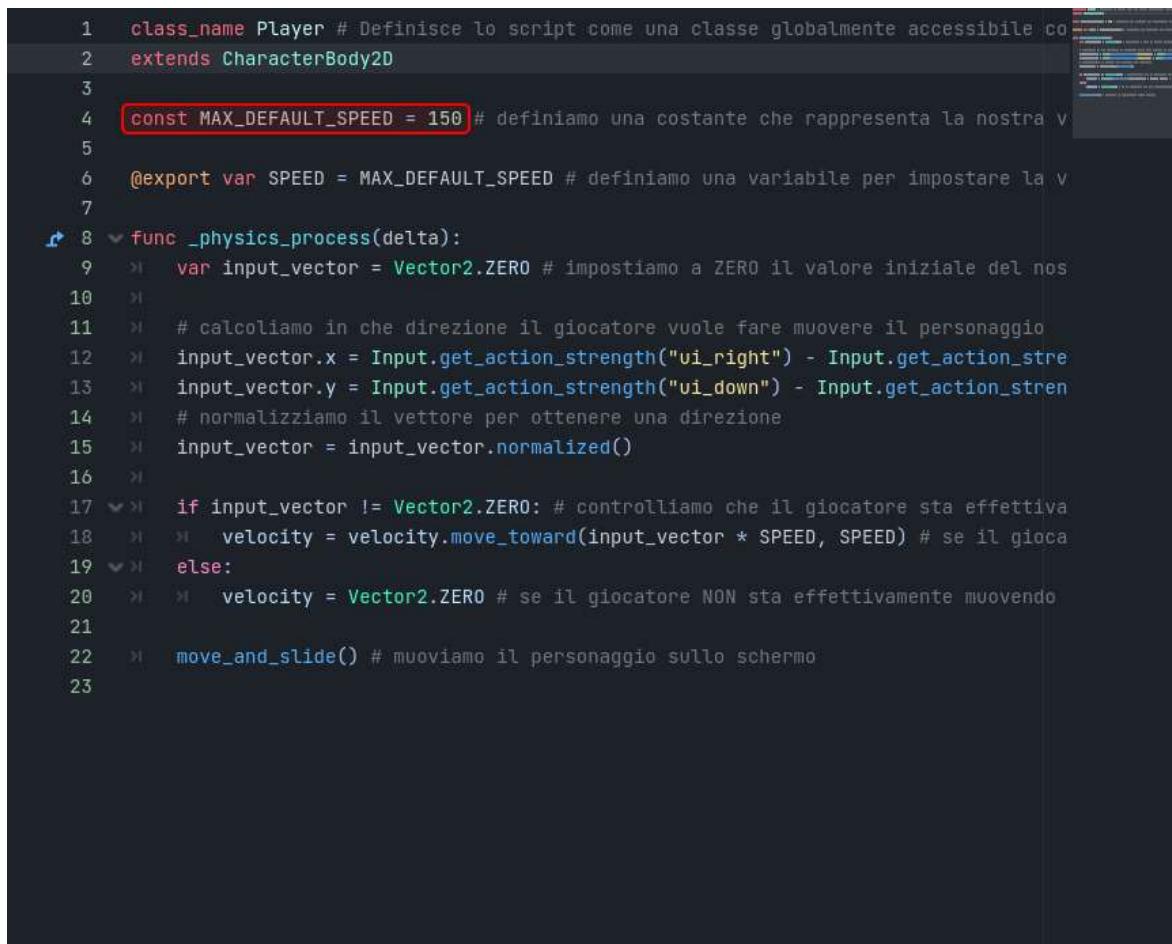
```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità massima
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo il personaggio
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio
22
23     move_and_slide() # muoviamo il personaggio sullo schermo

```

di cui illustriamo ora il funzionamento.

Iniziamo con la riga numero 4



```

1  class_name Player # Definisce lo script come una classe globalmente accessibile co
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra v
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la v
7
8  func _physics_process(delta):
9    var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nos
10   #
11   # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12   input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_stre
13   input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_stren
14   # normalizziamo il vettore per ottenere una direzione
15   input_vector = input_vector.normalized()
16   #
17   if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettiva
18     velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il gioca
19   else:
20     velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo
21
22   move_and_slide() # muoviamo il personaggio sullo schermo
23

```

**const** è la keyword che Godot riserva per la definizione di **costanti**.

In programmazione, una **costante**, come suggerito dal nome, è un valore che non può *mai* essere modificato da un programma durante l'esecuzione dello stesso.

Nella nostra riga 4 stiamo dunque definendo una costante di nome **MAX\_DEFAULT\_SPEED** che rappresenterà la velocità massima che potrà avere il nostro personaggio.

Le costanti, ma così come le variabili, inoltre, possono essere **dichiarate** o **inizializzate**. Con **dichiarazione** si indica l'introduzione di una nuova entità nel programma, ovvero, il *dichiarare* la creazione di una variabile o costante.

```
24  var color
```

lì dove **var** è la keyword che Godot riserva per la definizione di **variabili**.

Con **inizializzazione**, invece, si indica l'assegnamento di un valore a una variabile o costante.

```
25  color = "red"
```

Come possiamo osservare in questo esempio, l'inizializzazione avviene in un secondo

momento rispetto alla dichiarazione. Difatti, se provassimo solamente a dichiarare una costante su Godot, senza contemporaneamente inizializzarla come abbiamo fatto a riga

4

A screenshot of the Godot Script Editor. The code editor window shows line 25 with the text "const HEIGHT". Below the editor, a status bar displays the message "< Error at (25, 12): Expected initializer after constant name." followed by a red error icon, the number "1", and the text "25 : 13 | Tabs".

lo **Script Editor** ci segnalerebbe un errore, dicendoci che, poiché stiamo dichiarando una costante, ovvero un valore che non può essere cambiato nel tempo, dobbiamo contemporaneamente anche inizializzarla. Non possiamo solo dichiararla. Cosa potremmo farci altrimenti di un valore vuoto che non può essere cambiato?

25 const HEIGHT = 14|

Come possiamo intuire, le variabili, invece, a differenza delle costanti, possono essere sia dichiarate e inizializzate contemporaneamente che in un secondo momento.

24 var month = "July"  
25  
26 var year  
27 year = "2024"|

Un altro aspetto fondamentale di variabili e costanti è lo **scope**. Per **scope** si intende la regione di un programma in cui un **name binding** è valido. Lì dove **name binding** è solo un termine raffinato per dire "variabile". Esistono diversi tipi di scope:

- Global Scope
- Class/File/Module Scope
- Function Scope
- Code Block Scope

Una classe, variabile, costante o funzione si dice avere **Global Scope** quando può essere utilizzata ovunque all'interno di un'applicazione Godot. Un esempio sarebbe la classe **Node**, che possiamo sempre estendere da qualunque posto in cui ci troviamo

24 extends Node|

Una variabile, costante o funzione si dice avere **Class Scope** quando può essere utilizzata ovunque all'interno del file di una classe, come ad esempio è il caso per la nostra costante `MAX_DEFAULT_SPEED`

Una variabile o costante si dice avere **Function Scope** quando può essere utilizzata solamente all'interno di una funzione. Un esempio è la variabile `input_vector` del nostro codice



```

1  class_name Player # Definisce lo script come una classe globalmente accessibile co
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra v
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la v
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nos
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_stre
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_stre
14     # normalizziamo il vettore per ottenere una direzione
15     input_vector = input_vector.normalized()
16
17     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettiva
18         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il gioca
19     else:
20         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo
21
22     move_and_slide() # muoviamo il personaggio sullo schermo
23

```

Infine, una variabile o costante si dice avere **Code Block Scope** quando può essere utilizzata solamente all'interno di un **blocco di codice**. Esempi di **blocchi di codice** sono l'*if statement*, il *match statement*, i *for loops*, ecc..., che vedremo più avanti



```

24  if(true):
25      var message = "Ciao!"

```

Un'ultimo aspetto importante di variabili è costanti è il loro **tipo**. Il **tipo** o **data type** descrive appunto il *tipo* di valore che una variabile/costante possiede e i tipi di operazione matematiche, relazionali o logiche che possono essere eseguite su di esso senza sfociare in errore (non ha senso sommare due stringhe, ovvero due sequenze di caratteri, ma due numeri sì). Ad esempio, la costante `MAX_DEFAULT_SPEED` è di tipo `int`, di tipo *intero*, poiché l'inizializzazione al valore `150`, è difatti ciò che conferisce

il tipo alla costante. Tuttavia, una variabile può essere anche dichiarata fin da subito con un tipo specifico

```
24  var text: String
```

e la stessa cosa vale anche per le costanti, anche se, in questo caso, può risultare ridondante per via dell'inizializzazione che conferisce automaticamente appunto il tipo alla costante stessa

```
24  const VECTOR: Vector2 = Vector2.DOWN
```

### ! Nota

In programmazione, lo **static typing** ("tipizzazione statica") è un sistema dove i tipi di dati delle variabili vengono controllati durante la compilazione, prima che il programma venga eseguito. Il programmatore dichiara esplicitamente il tipo di ogni variabile, e il compilatore verifica che le operazioni siano compatibili con quei tipi. Questo approccio aiuta a trovare errori di tipo in anticipo, migliorando la sicurezza e potenzialmente le prestazioni del codice, ma può rendere il codice più verboso e meno flessibile. Linguaggi come Java, C++ e C# utilizzano la tipizzazione statica.

In Godot, GDScript è tradizionalmente un linguaggio a tipizzazione dinamica, il che significa che i tipi di dati sono determinati durante l'esecuzione. Tuttavia, Godot ha introdotto la tipizzazione statica opzionale. Ciò consente ai programmatori di dichiarare esplicitamente i tipi di dati, beneficiando del rilevamento precoce degli errori, di potenziali miglioramenti delle prestazioni e di una migliore leggibilità del codice. La tipizzazione statica in GDScript è uno strumento potente per progetti di grandi dimensioni, che offre maggiore controllo e sicurezza senza sacrificare completamente la flessibilità del linguaggio dinamico.

Il tipo `String` definisce sequenze di caratteri, mentre `Vector2` un vettore bidimensionale (come può ad esempio essere (1,3)). Godot possiede molti altri data type, che però vedremo a mano a mano, quando dovremmo utilizzarli.

Riprendendo in mano il nostro codice, capiamo dunque ora che in riga 4 stiamo dichiarando e inizializzando contemporaneamente una costante di tipo `int` (intero) chiamata `MAX_DEFAULT_SPEED`, di valore 150 e scope Class Scope.

Andando avanti, riga 6 contiene invece una nuova keyword, `@export`

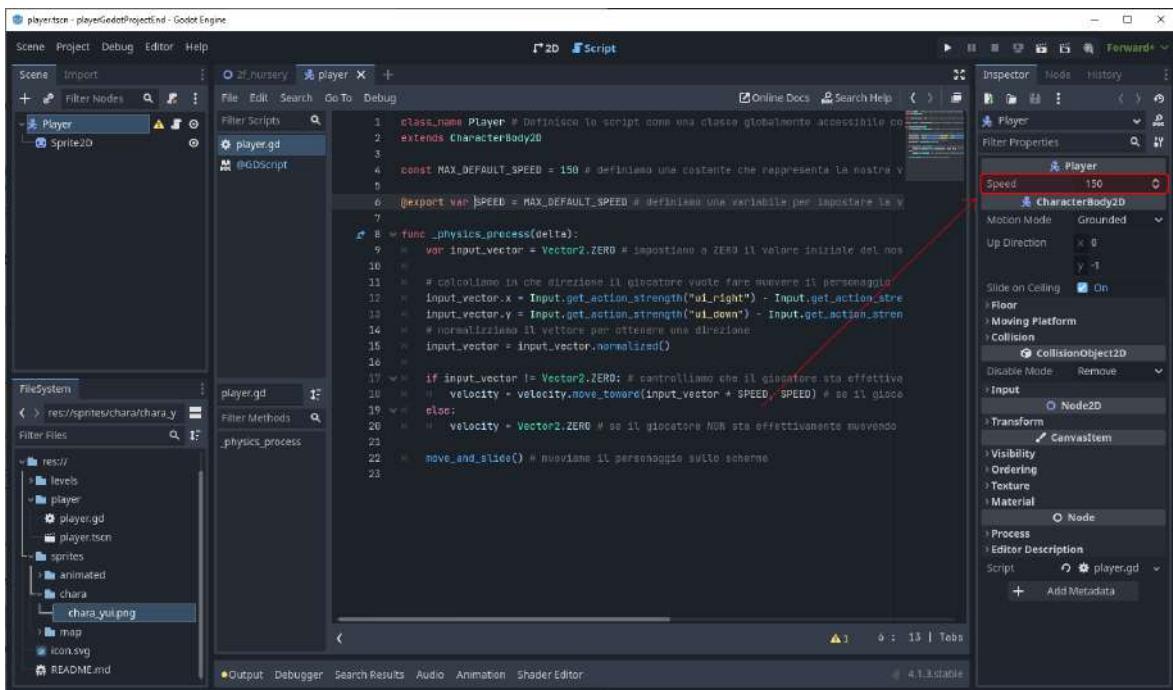


```

1  class_name Player # Definisce lo script come una classe globalmente accessibile co
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra v
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la v
7
8  func _physics_process(delta):
9    var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nos
10   #
11   # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12   input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_stre
13   input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_stre
14   # normalizziamo il vettore per ottenere una direzione
15   input_vector = input_vector.normalized()
16   #
17   if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettiva
18     velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il gioca
19   else:
20     velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo
21
22   move_and_slide() # muoviamo il personaggio sullo schermo
23

```

In Godot, i membri di una classe possono essere esportati. Questo significa che il loro valore viene salvato insieme alla risorsa (come ad esempio la scena) a cui sono collegati. Questi potranno inoltre essere modificati dall'Inspector dock, come difatti è possibile fare per la variabile esportata `SPEED` del nostro codice



Una variabile esportata deve essere inizializzata a un'espressione costante, come è il caso della nostra variabile **SPEED**, oppure avere il proprio tipo specificato

24    `@export var number: int`

Uno dei maggiori benefici di esportare una variabile membro è proprio quello di essere visibile e modificabile dall'editor. In questo modo, durante il testing, si può cambiarne il valore al volo per vedere come ciò influisca sul gioco.

A riga 6 del nostro codice stiamo dunque dichiarando una variabile esportata **SPEED**, che corrisponde alla velocità a cui il personaggio si sta muovendo in un determinato istante, e la stiamo inizializzando con il valore della costante **MAX\_DEFAULT\_SPEED**, ovvero **150**.

A riga 8 eseguiamo invece l'override della nostra cara funzione **\_physics\_process(delta)**. Parliamo di override perché **\_physics\_process(delta)** non è una funzione che stiamo creando noi in questo istante, ma è bensì una funzione (o metodo) della classe **Node**. Tutto quello che stiamo facendo noi è effettuarne l'override, ovvero, semplicemente cambiarne il contenuto, l'azione che compirà quando verrà chiamata (invocata).

Come già detto in precedenza, una **funzione** è un'unità modulare di codice progettato per eseguire task specifiche. In altre parole, è un'insieme di righe di codice che serve a fare un qualcosa di specifico, come ad esempio la somma tra due numeri o il prodotto cartesiano di due vettori.

**func** è la keyword che Godot riserva per definire una funzione. Per spiegare meglio l'anatomia di una funzione, consideriamo la funzione esempio **sum(number\_1, number\_2)** che esegue la somma tra due numeri frazionari

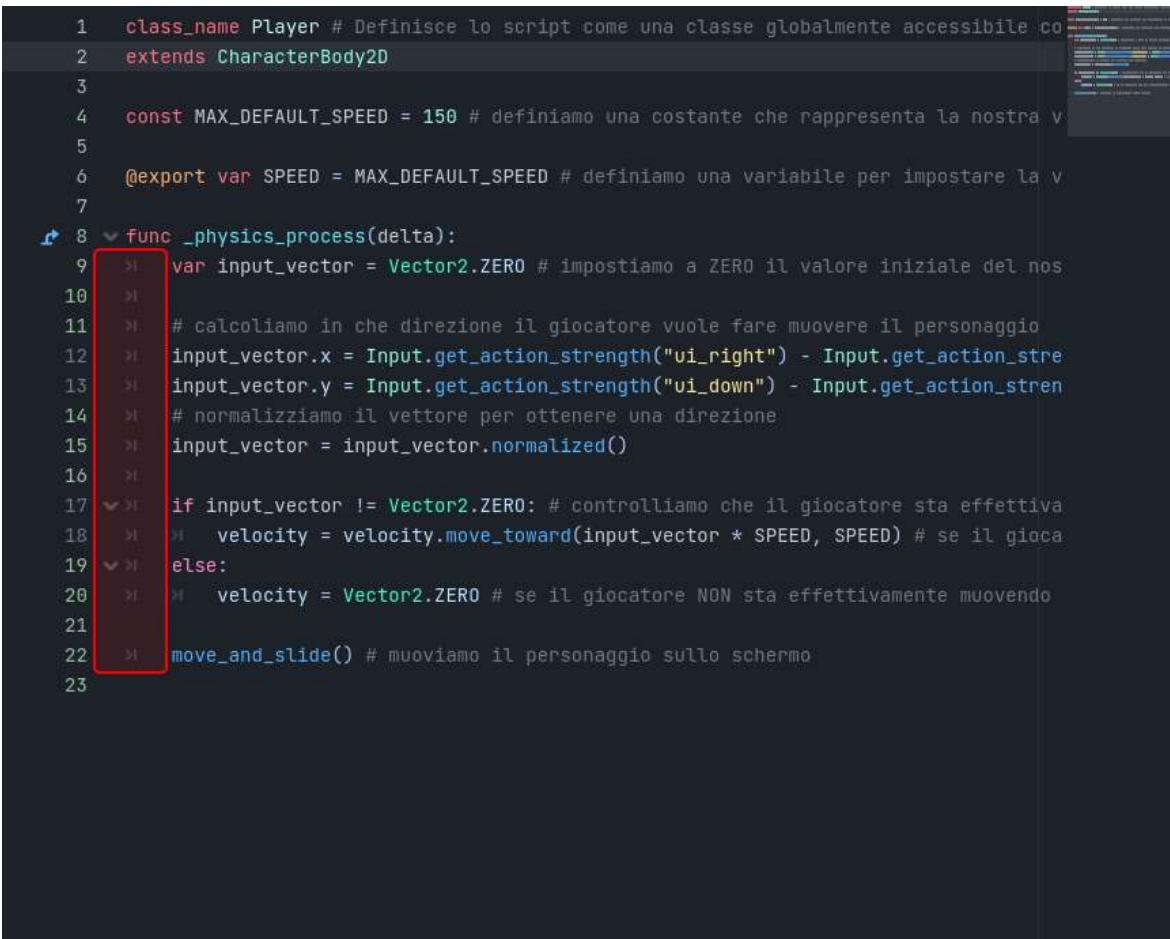
```
24 func sum(number_1: float, number_2: float) -> float:
25     return number_1 + number_2;
```

Come possiamo notare dall'immagine, una funzione inizia con una keyword (`func` nel caso di Godot) (1), a cui segue il nome della funzione (2), gli argomenti o parametri, separati da una virgola, passati alla funzione (3) con i loro rispettivi tipi (4) e il tipo (5) del valore di ritorno della funzione (6), specificato dalla keyword `return`. Il tutto chiuso da un `:`.

```
23  1  2  3  4  3  4  5
24 func sum(number_1: float, number_2: float) -> float:
25     return number_1 + number_2; 6
```

La riga 24, ovvero la prima riga della funzione (o metodo) è detta **firma** o **signature** del metodo. La firma del metodo può cambiare a seconda del linguaggio di programmazione che si sta utilizzando.

Si noti inoltre che, come meglio si evince maggiormente dal codice della nostra funzione `_physics_process(delta)`, il **corpo**, ovvero tutto il codice che appartiene a una funzione va **indentato** (si lascia uno spazio bianco a sinistra).



The screenshot shows a Godot script editor window. The code is as follows:

```
1 class_name Player # Definisce lo script come una classe globalmente accessibile co
2 extends CharacterBody2D
3
4 const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra v
5
6 @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la v
7
8 func _physics_process(delta):
9     var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nos
10    # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
11    input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_stre
12    input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_stren
13    # normalizziamo il vettore per ottenere una direzione
14    input_vector = input_vector.normalized()
15
16    if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettiva
17        velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il gioco
18    else:
19        velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo
20
21    move_and_slide() # muoviamo il personaggio sullo schermo
22
23
```

A red box highlights the entire body of the `_physics_process(delta)` function, from the opening brace at line 8 to the closing brace at line 22. The code uses standard Python-style indentation with four spaces per level.

### ! Nota

In programmazione, l'**override** ("sovrascrittura") di una funzione è un concetto fondamentale legato alla programmazione orientata agli oggetti (OOP). L'override si verifica quando una sottoclasse (o classe derivata) fornisce una specifica implementazione per una funzione che è già definita nella sua superclasse (o classe base). In pratica, la sottoclasse "sovrascrive" il comportamento della funzione ereditata dalla superclasse. Quando si crea una sottoclasse, essa eredita tutte le funzioni e le proprietà della sua superclasse. Se la sottoclasse definisce una funzione con lo stesso nome, gli stessi parametri e lo stesso tipo di ritorno di una funzione presente nella superclasse, allora la funzione della sottoclasse "sovrascrive" quella della superclasse. Quando viene chiamata la funzione su un oggetto della sottoclasse, verrà eseguita l'implementazione della sottoclasse, non quella della superclasse. Lo scopo dell'override è consentire alle sottoclassi di adattare o modificare il comportamento delle funzioni ereditate per soddisfare le proprie esigenze specifiche. È un meccanismo chiave per l'implementazione del polimorfismo, uno dei principi fondamentali dell'OOP.

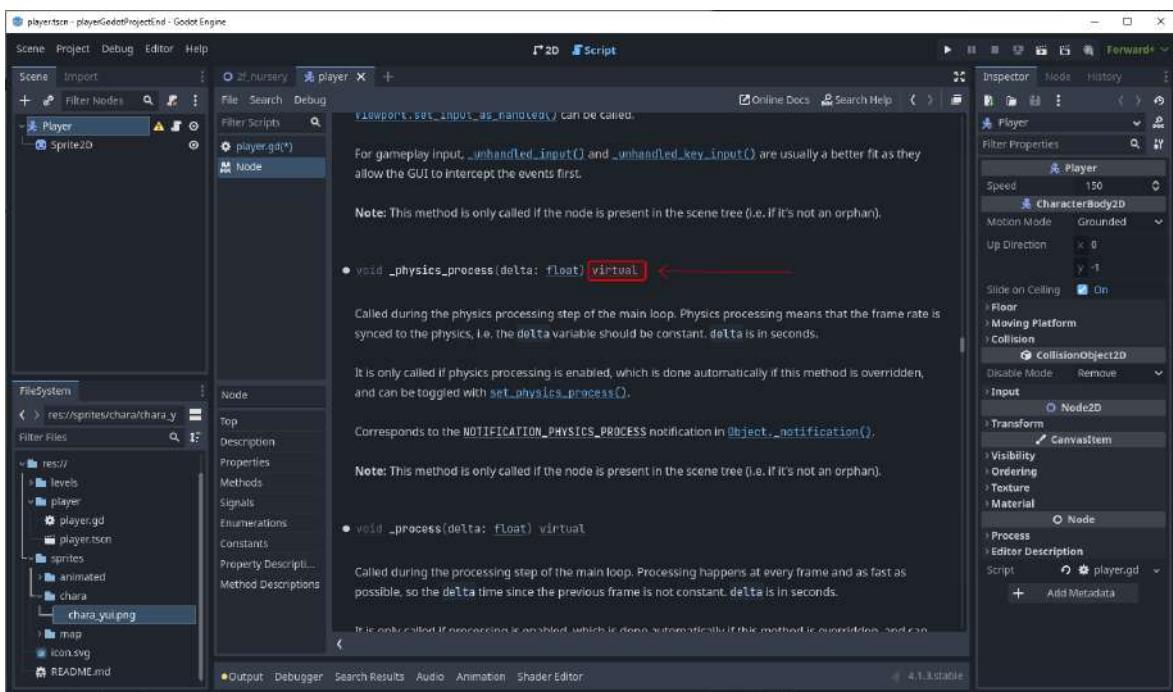
Un esempio pratico è immaginare di avere una superclasse **Animal** con una funzione **make\_sound()**. Le sottoclassi **Dog** e **Cat** possono sovrascrivere la funzione **make\_sound()** per emettere il verso specifico di ciascun animale (**Bau!** per il cane e **Miao!** per il gatto).

L'override è essenziale per creare gerarchie di classi flessibili e riutilizzabili. Consente di specializzare il comportamento delle classi senza dover riscrivere interamente il codice.

In sintesi, l'override di una funzione permette a una sottoclasse di fornire una propria versione di una funzione ereditata dalla superclasse, consentendo di personalizzare e adattare il comportamento delle classi in modo efficace.

Un'ultima cosa prima di vedere il codice vero e proprio della nostra funzione **\_physics\_process(delta)** è proprio il **-** con cui inizia il nome di quest'ultima. Il **-** all'inizio del nome di una funzione può significare diverse cose, ma è sempre solo e soltanto una convenzione in tutte le circostanze. Non c'è alcun "obbligo" da parte del linguaggio di programmazione (GDScript) o dalle regole. Piuttosto, è semplicemente un modo per trasmettere informazioni a chi sta leggendo il codice sorgente per fornire loro un'idea migliore del significato che si nasconde dietro il metodo. Tra i vari utilizzi di **-** troviamo:

1. il metodo è "*privato*": può essere utilizzato solamente all'interno della classe dalla classe stessa. Le altre classi al di fuori dello scope della classe non dovrebbero provare ad accederlo. Molti altri linguaggi di programmazione statically-typed (C++, C#, Java, ecc...), in realtà, forzano questo comportamento tramite la keyword **private**. In GDScript, tuttavia, tutto è pubblico su un livello tecnico (la funzione **\_shoot()** può ad esempio essere chiamata anche al di fuori dell'oggetto e anche se inizia con un **-**)
2. il metodo è "*virtuale*" ("*virtual*"): è pensato per essere sovrascritto tramite override da una classe derivata, come ad esempio è il caso per **\_physics\_process(delta)**.



Questo può essere anche il caso di uno script che ha una funzione personalizzata con un **pass**, ovvero una keyword riservata di Godot che evita che l'editor dia errore quando si lascia un metodo vuoto, ovvero senza corpo

```
24 func _attack():
25     pass
```

3. in Godot c'è anche un terzo caso: il metodo è una "notification" ("notifica"), ovvero, una funzione di comodo che consente agli utenti di accedere facilmente a certe sottosezioni del core dell'iteration loop di godot. Ad esempio, in **\_notification(what)**, il **what** è un **enum**, un tipo enumerativo, che contiene diversi valori interi definiti nello scope globale con il prefisso **NOTIFICATION\_\***. Difatti, la funzione **\_ready()** è una shortcut per il codice

```
24 <pre><code>func _notification(p_what):
25     match p_what:
26         NOTIFICATION_READY:
27             # _ready() content goes here
```

Un tipo **enum** è un data type che consiste in un insieme di named value chiamati *elementi, membri, numerali o enumerators* del tipo. I nomi degli elementi sono spesso identificatori che si comportano allo stesso modo delle costanti del linguaggio.

```

24 enum Difficulty {
25     LOW,
26     MEDIUM,
27     HIGH,
28     HARDCORE
29 }

```

Per accedere a una costante `enum` si utilizza la **dot syntax** ("sintassi a punto") o **dot notation** ("notazione col punto").

```

31 var current_difficulty = Difficulty.HARDCORE;

```

Proseguendo con le linee di codice, in riga 9, all'interno del corpo della nostra funzione `_physics_process(delta)`, definiamo una variabile Function Scope di nome `input_vector` a cui assegniamo la costante `ZERO` della classe `Vector2`, a cui si accede mediante dot notation.

```

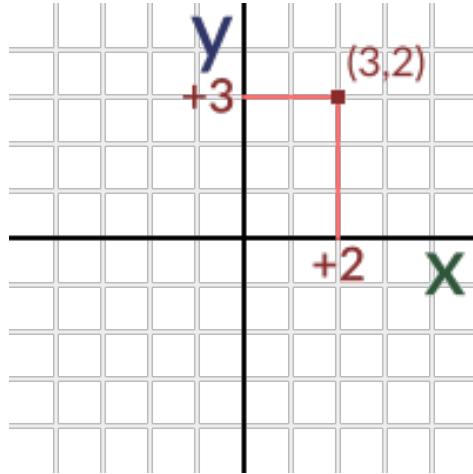
1  class_name Player # Definisce lo script come una classe globalmente accessibile co
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra v
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la v
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nos
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_stre
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_stren
14     # normalizziamo il vettore per ottenere una direzione
15     input_vector = input_vector.normalized()
16
17     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettiva
18         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il gioca
19     else:
20         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo
21
22     move_and_slide() # muoviamo il personaggio sullo schermo
23

```

La classe `Vector2` di Godot è utilizzata per rappresentare **vettori** bidimensionali, ovvero con coordinate `x` e `y`.

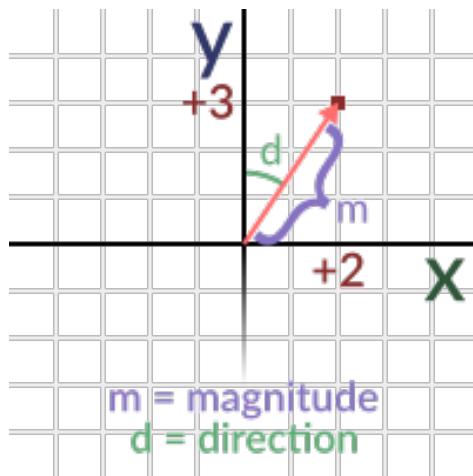
Normalmente, una coordinata è definita come una coppia (x,y), lì dove la x rapp-

resenta l'asse orizzontale, mentre la y quello verticale. Il che è sensato dato che uno schermo non è nient'altro che un rettangolo in due dimensioni. Un'esempio di posizione in uno spazio 2D è il seguente.

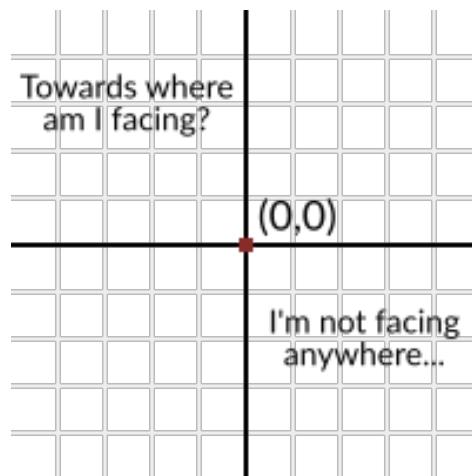


Una posizione può trovarsi in qualsiasi punto dello spazio. La posizione (0,0) è chiamata **origine**.

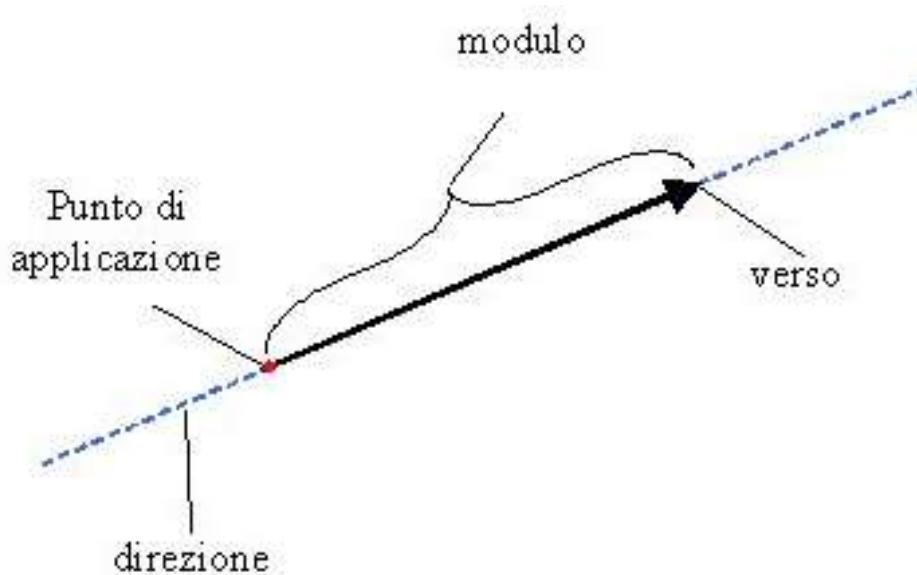
Nella teoria dei vettori, le coordinate hanno due utilizzi diversi, entrambi ugualmente importanti: esse sono utilizzate per rappresentare una *posizione* ma anche un *vettore*. La stessa posizione dell'immagine precedente, se pensata come un vettore, assume un significato diverso



Quando si immagina un vettore, si possono dedurre due proprietà: la direzione (direction) e l'intensità (magnitude). Ogni posizione nello spazio può essere un vettore, fatta eccezione dell'origine; questo perché le coordinate (0,0) non possono rappresentare una direzione (l'intensità è 0)



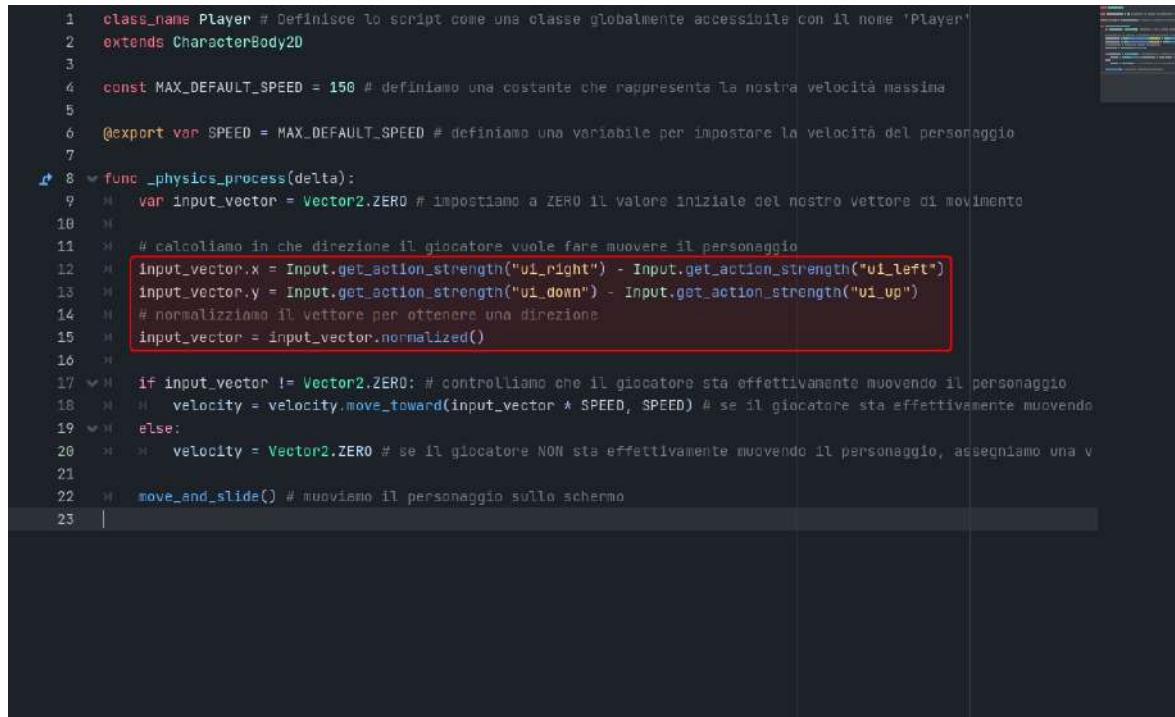
In uno spazio a due dimensioni (piano x,y) un **vettore** può essere dunque immaginato come un segmento orientato al quale è assegnato un modulo (intensità, norma o lunghezza), una direzione e un verso. Il punto da cui "ha origine" un vettore è detto punto di applicazione.



Godot utilizza la classe `Vector2` sia come posizione che come direzione (vettore). La costante `ZERO` della classe `Vector2` equivale a un vettore con tutte le componenti poste a 0, ovvero (0,0). Dunque, un modo equivalentemente per scrivere la nostra riga 9 è

```
25 var input_vector = Vector2(0,0)
```

Per comprendere al meglio le righe che vanno dalla 12 alla 15

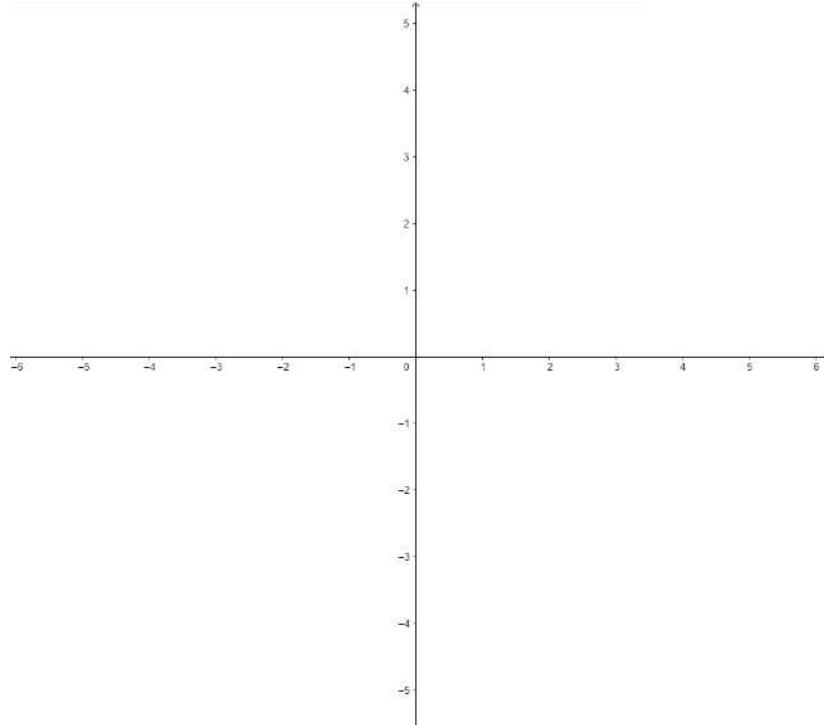


```

1 class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2 extends CharacterBody2D
3
4 const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6 @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8 func _physics_process(delta):
9     var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14     # normalizziamo il vettore per ottenere una direzione
15     input_vector = input_vector.normalized()
16
17 if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
18     velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
19 else:
20     velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
21
22 move_and_slide() # muoviamo il personaggio sullo schermo
23

```

ci spostiamo invece su un piano cartesiano.



e immaginiamo di posizionare su questo piano un controller, fancedo coincidere il centro di una delle sue due levette analogiche con l'origine degli assi.



Il massimo che possiamo muovere questa levetta analogica verso destra è 1 unità



Come possiamo vedere, le nuove coordinate della levetta analogica sul piano cartesiano sono (1,0).

Allo stesso modo, spostando la levetta analogica in alto avremo coordinate (0,1)



a sinistra (-1,0)



e, infine, in basso (0,-1).

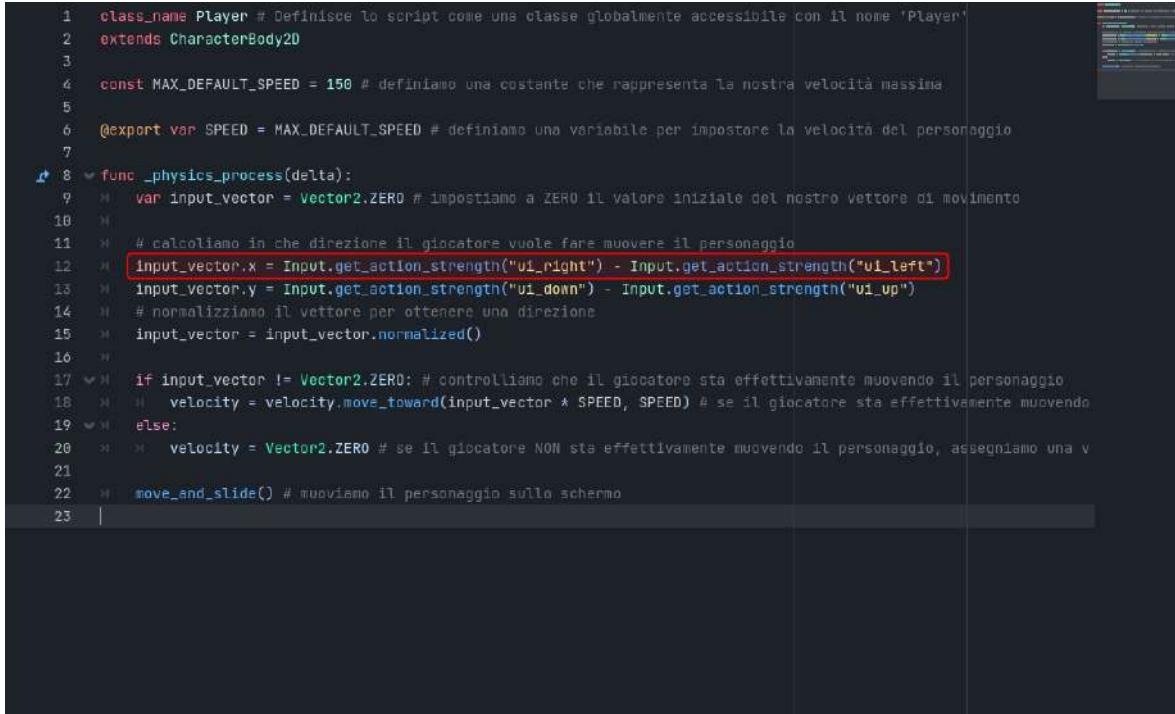


Poiché però nei videogiochi l'asse y viene invertito, le vere coordinate della nostra levetta analogica nello spazio saranno:

- (1,0) a destra
- (0,-1) in alto
- (-1,0) a sinistra
- (0,1) in basso

Quello che abbiamo appena descritto è il nostro `input_vector`.

Trasportando questo concetto dal controller alle frecce direzionali della tastiera, se il giocatore sta ad esempio premendo la freccia direzionale destra, senza premere contemporaneamente quella sinistra, allora, il valore di `Input.get_action_strength("ui_right")` sarà `1` mentre quello di `Input.get_action_strength("ui_left")` sarà `0`. Dunque, poiché  $1-0=1$ , la coordinata `x` del nostro `input_vector` sarà `1`.



```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
22
23     move_and_slide() # muoviamo il personaggio sullo schermo
24

```

lì dove il singleton `Input` è incaricato di gestire l'input utente, ovvero, di riconoscere quando un utente ha premuto un tasto della tastiera, del mouse o di un gamepad; mentre il suo metodo `get_action_strength(action, exact_match)` ritorna un valore compreso tra 0 e 1 che rappresenta l'intensità (il modulo) di una data azione.

### ! Nota

Per chiamare una funzione o accedere a una proprietà di un oggetto si utilizza la sintassi `object_name.function_name()`, in caso di una funzione e `object_name.property_name`, in caso di una proprietà. Ad esempio, `Input.is_action_pressed("ui_accept")` chiama la funzione `is_action_pressed()` del singleton `Input`, e `Input.get_mouse_position()` chiama la funzione `get_mouse_position()` del singleton `Input`.

Discorso analogo vale per le direzioni su e giù della riga `13`. Notiamo che, nella sottrazione, su e giù sono invertiti proprio perché l'intero asse y è invertito.

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
22
23     move_and_slide() # muoviamo il personaggio sullo schermo

```

In poche parole, il nostro `input_vector` non è nient'altro che il vettore che rappresenta la direzione in cui il giocatore vuole muoversi.

Se adesso immaginiamo invece di star premendo in basso a destra



avranno coordinate (1,1). E questo è gestito perfettamente dalle righe di codice che abbiamo scritto. Tuttavia, poiché per il teorema di Pitagora, la lunghezza di c è maggiore di quella di a o di b, lungo le diagonali la velocità del giocatore risulterebbe

sempre maggiore di quella sugli assi.



L'istruzione di riga 15 ci viene in aiuto proprio per risolvere questo problema.

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14     # normalizziamo il vettore per ottenere una direzione
15     input_vector = input_vector.normalized()
16
17     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
18         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
19     else:
20         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
21
22     move_and_slide() # muoviamo il personaggio sullo schermo
23

```

Normalizzare un vettore vuol dire "accorciare" quest'ultimo tramite un cerchio di raggio 1.

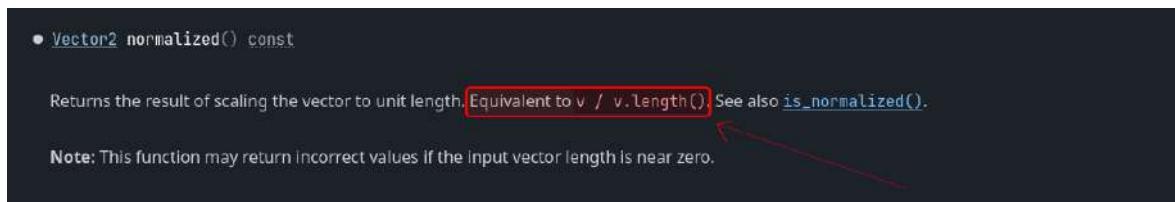


Così facendo, il nostro nuovo vettore c risultante da questo taglio è



che ci consente di mantenere la stessa velocità degli assi anche sulle diagonali.

Come indicatoci anche dalla reference del metodo `normalized()`, per normalizzare un vettore basta dividere il vettore per la propria lunghezza.



La riga dalla 17 alla 20 vede invece la presenza di un *if statement*.

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
17     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
18         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
19     else:
20         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
21
22     move_and_slide() # muoviamo il personaggio sullo schermo
23

```

Un **if statement** è un’istruzione condizionale che esegue o salta una sequenza di istruzioni a seconda del valore dell’espressione booleana.

Nel nostro codice, `if` è la keyword riservata da Godot per definire un’if statement, mentre `input_vector != Vector2.ZERO` è l’espressione booleana da valutare. Se l’espressione booleana è vera, ovvero, se `input_vector` ha un valore diverso (`!=`) da `Vector2.ZERO`, che ricordiamo essere (0,0), allora l’if statement eseguirà l’istruzione a riga 18. Se invece l’espressione booleana è falsa, verrà eseguita l’istruzione a riga 20. La keyword `else` funge da separatore tra il ramo vero e quello falso di un if statement. Come possiamo notare, inoltre, entrambi i rami dell’if statement vanno indentati.

Il controllo a riga 17 ci serve solamente per capire se il giocatore sta effettivamente muovendo il player (caso in cui eseguiamo la riga 18) oppure no (caso in cui eseguiamo la riga 20).

Iniziamo con l’analizzare la riga 18.

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
22
23     move_and_slide() # muoviamo il personaggio sullo schermo
24

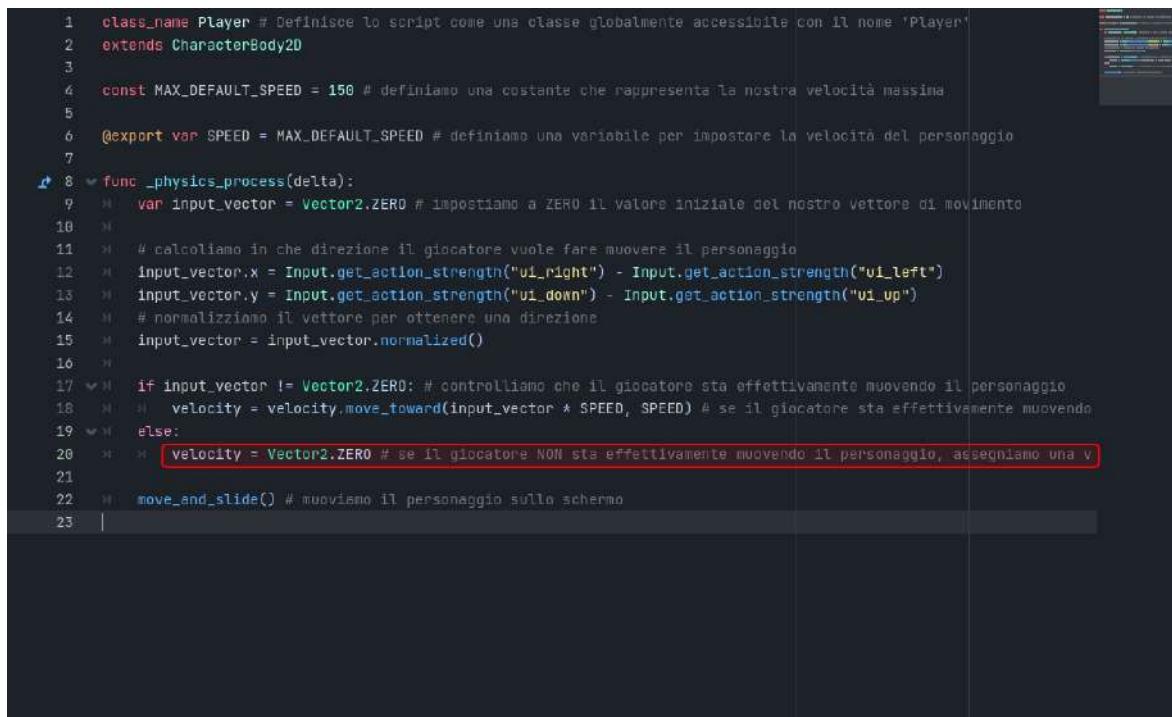
```

La variabile `velocity` è una variabile della classe `CharacterBody2D` che rappresenta il vettore della velocità corrente in pixel per secondo ed è utilizzata e modificata all'interno della funzione `move_and_slide()`, che vedremo a breve.

`move_toward(to, delta)` è una funzione della classe `Vector2` (ricordiamo che `velocity` è di tipo `Vector2`), che ritorna un vettore (nel nostro caso il vettore `velocity`, poiché stiamo chiamando `move_toward(to, delta)` proprio su questo vettore tramite dot notation) spostato verso la posizione `to` della quantità fissata `delta`. È inoltre importante sapere che `move_toward(to, delta)` non andrà mai oltre il valore finale.

Quello che facciamo a riga 18 è dunque semplicemente spostare il vettore dalla posizione in cui si trova attualmente a quella definita da `input_vector * SPEED` di una quantità `SPEED` a ogni frame. Poiché la funzione `_physics_process(delta)` viene chiamata a intervalli fissi, indipendentemente dal frame rate (per impostazione predefinita, questo intervallo è di 60 volte al secondo, ma può essere modificato nelle impostazioni del progetto), se `input_vector * SPEED` è uguale, ad esempio, a `(3, 0) * SPEED`, ciò significa che ci vorranno 3 frame per raggiungere la posizione `input_vector * SPEED`. Questo perché con ogni frame ci muoviamo di una quantità `SPEED`.

Alla riga 20, invece, poiché il player non deve essere mosso, impostiamo una velocità nulla

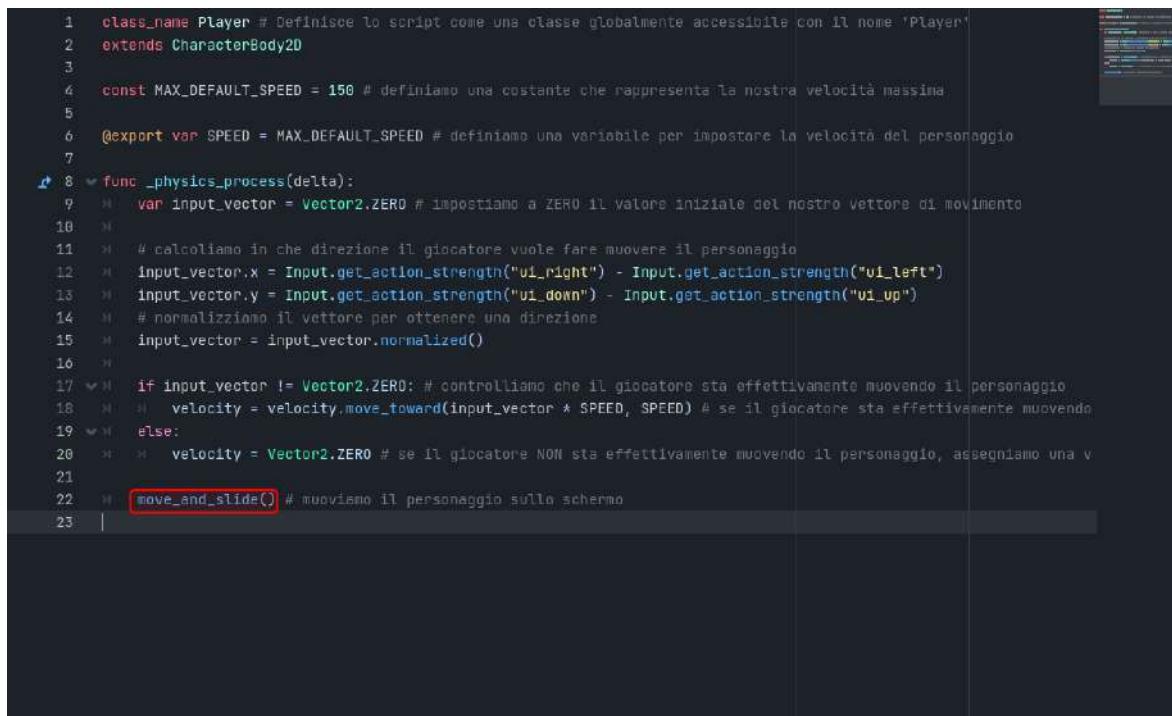


```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
22
23     move_and_slide() # muoviamo il personaggio sullo schermo

```

Una volta usciti dall'if statement, infine, a riga 22



```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamo una v
22
23     move_and_slide() # muoviamo il personaggio sullo schermo

```

invociamo il metodo `move_and_slide()` della classe `CharacterBody2D`, il quale muoverà il player in base alla variabile `velocity`. Grazie al metodo `move_and_slide()`, se il corpo (in questo caso il player) entra in collisione con un altro corpo, scivolerà lungo l'altro corpo (per default solo sul pavimento) piuttosto che fermarsi immediatamente.

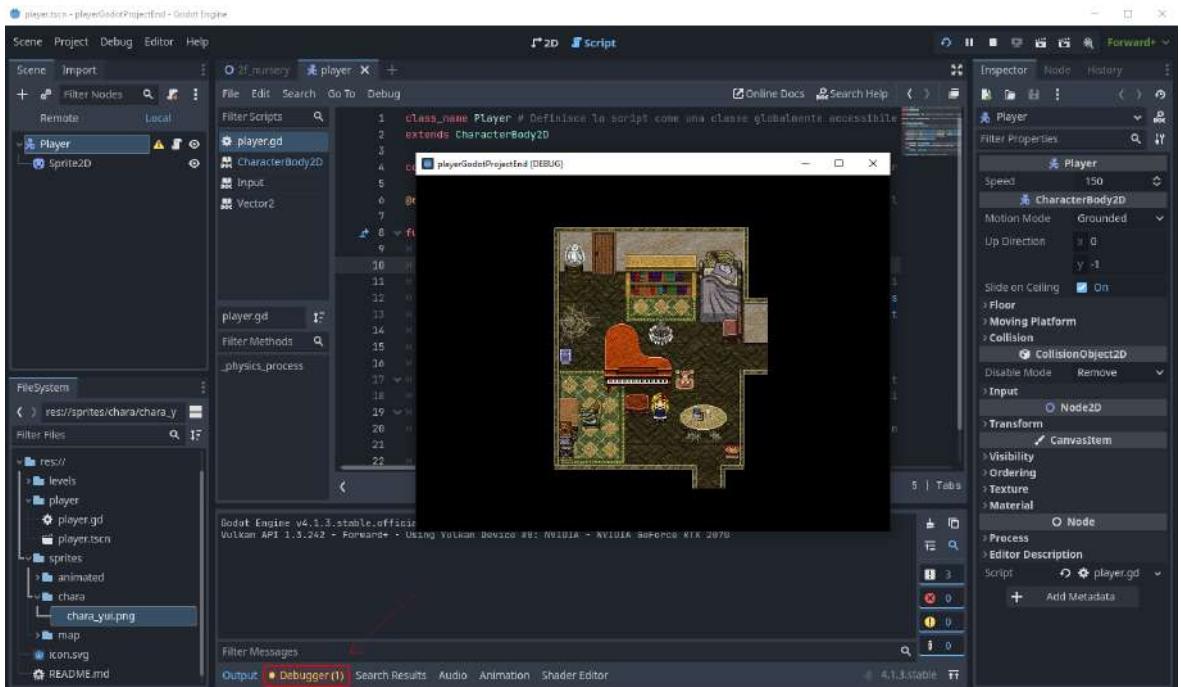
A questo punto, se eseguiamo il nostro videogioco



riusciremo tranquillamente a muovere il player tramite le frecce direzionali.

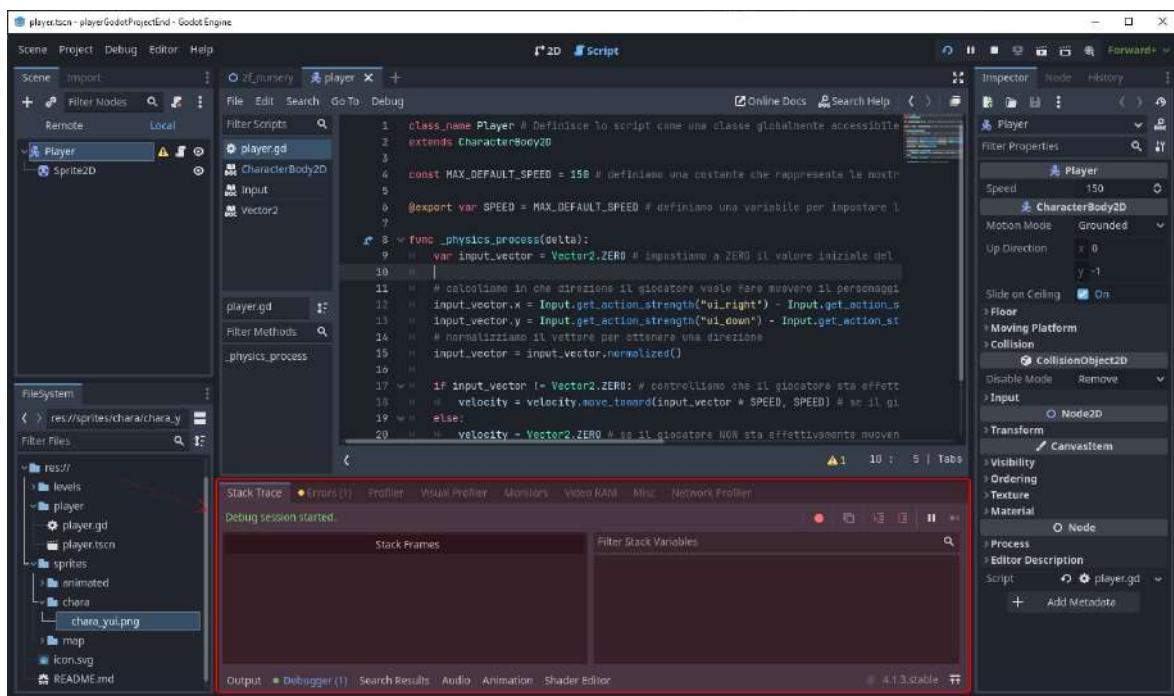


Se prestiamo però attenzione all'editor, noteremo che la scritta **Debugger** si è colorata di giallo e presenta un **(1)** di fianco

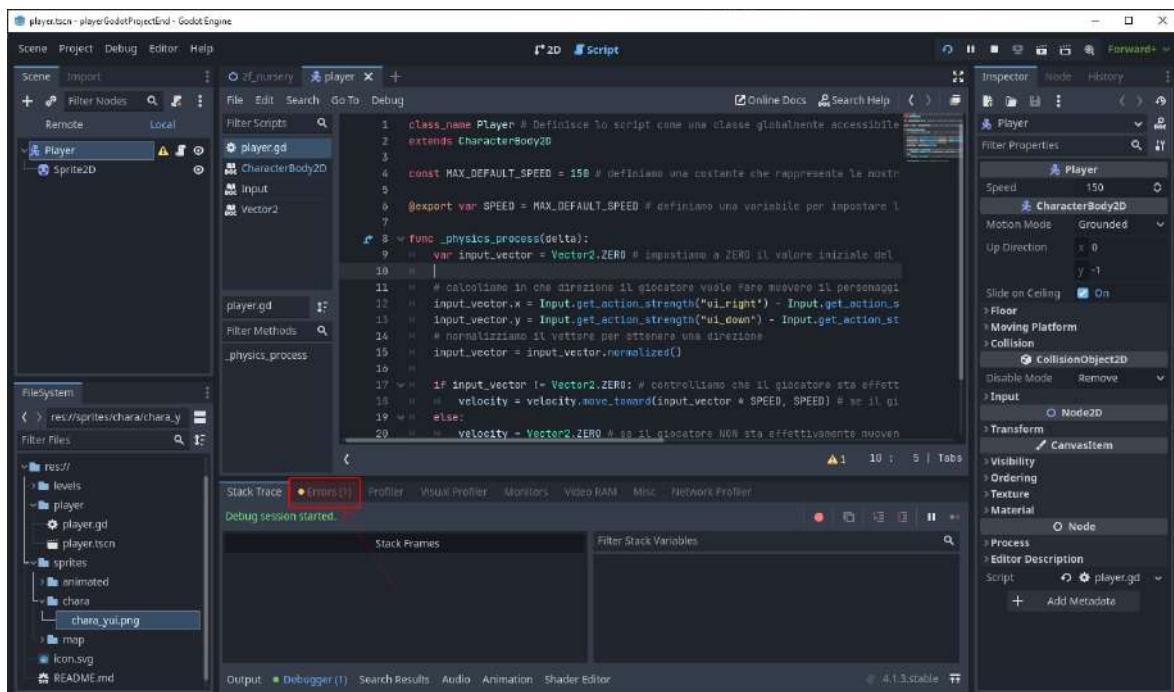


cliccandoci sopra, si aprirà la finestra del debugger

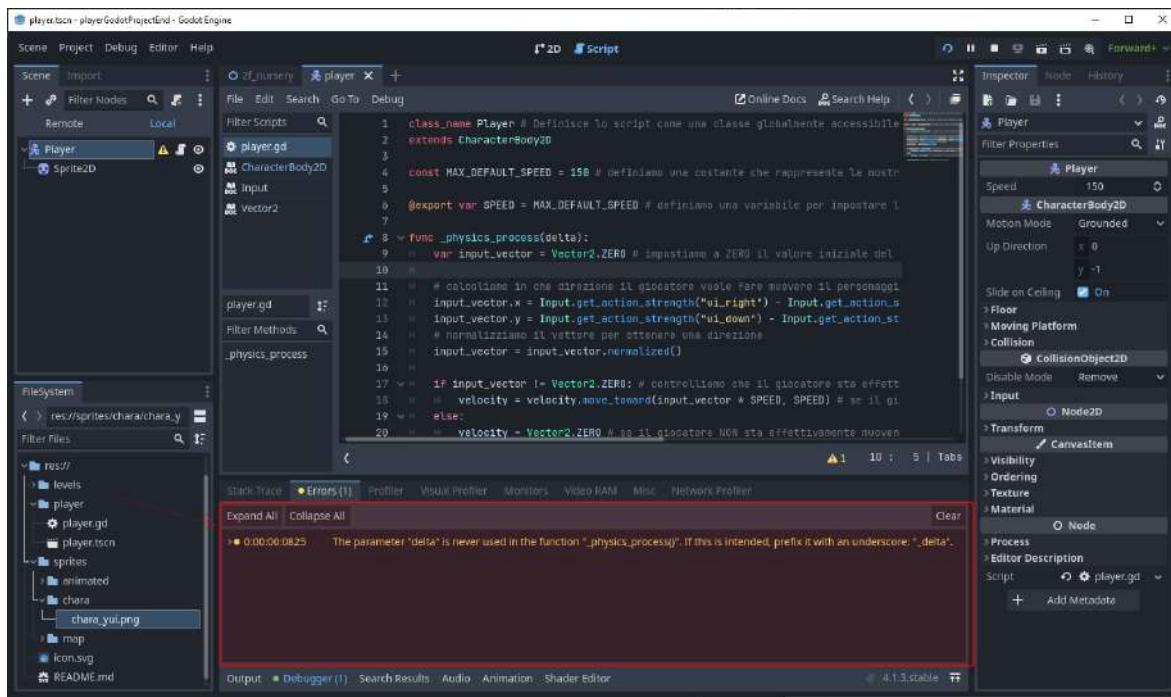
## CHAPTER 2. GODOT



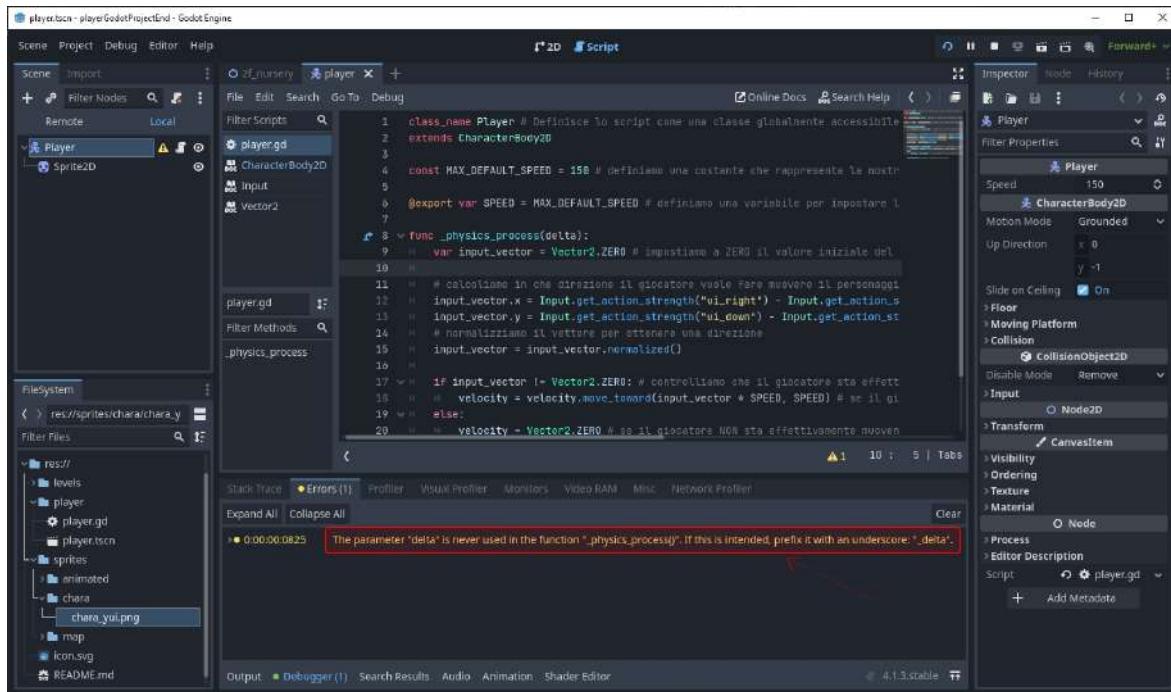
facciamo dunque click su **Errors**



per aprire la tab degli errori



Qui troviamo una (da qui il (1) di fianco a **Debugger**). Se fossero state più di una, tra parentesi avremmo invece avuto il numero preciso di scritte) scritta di colore giallo.



Le scritte che l'editor colora di giallo sono **Warning** e possono essere ignorate poiché si riferiscono a del codice che non è fonte di errore. Quelle invece colorate di rosso sono **Error**, ovvero veri e propri errori che non possono essere ignorati ma devono essere assolutamente risolti per far partire il gioco (se vi è anche un solo errore, infatti, il gioco termina la sua esecuzione).

Nel nostro caso, come scritto nella frase stessa, quello che l'editor ci sta comuni-

cando è che non abbiamo utilizzato il parametro `delta` all'interno della funzione `_physics_process(delta)`. Poiché questo comportamento è voluto, per risolvere, proprio come suggeritoci da Godot, basta aggiungere un `-` davanti al parametro.

```
1 class_name Player # Definisce lo script come una classe globalmente accessibile
2 extends CharacterBody2D
3
4 const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6 @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la nostra velocità
7
8 func _physics_process(delta):
9     var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del vettore di input
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il gioco non ha raggiunto la velocità massima, la aumenta
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, la velocità è zero
22
23     move_and_slide() # muoviamo il personaggio sullo schermo
```

Con `—` stiamo semplicemente dicendo a Godot che siamo consapevoli di non star utilizzando il parametro della funzione all'interno del corpo della funzione stessa.

## ⚠ Attenzione

Se provassimo a risolvere il problema togliendo `delta` dalla funzione `_physics_process(delta)`



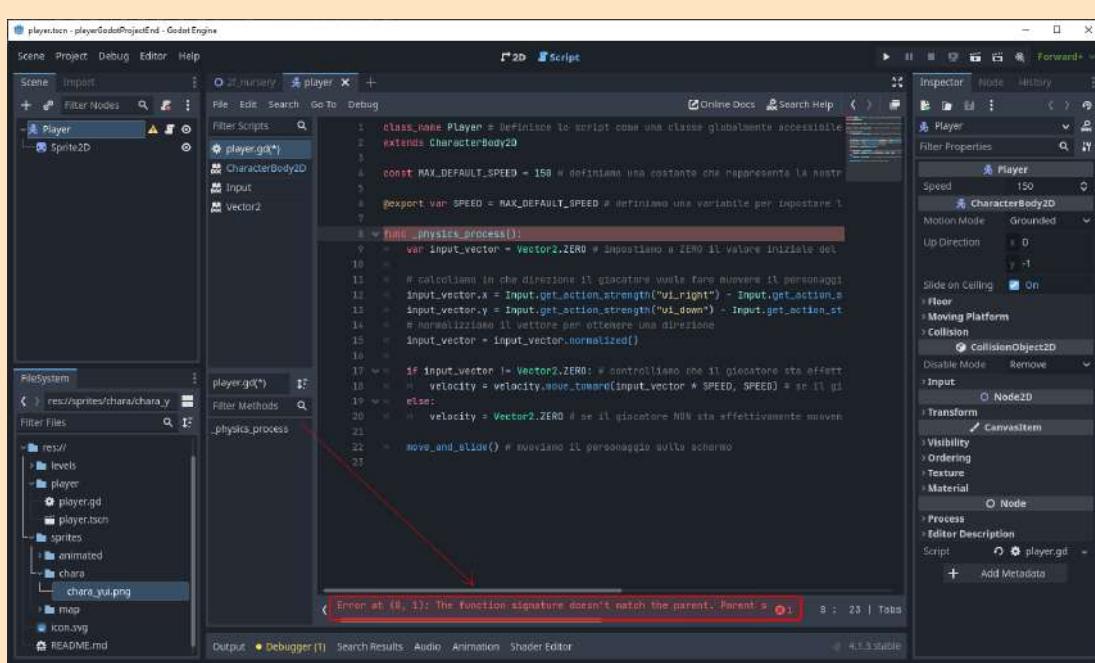
```

1  class_name Player # Definisce lo script come una classe globalmente accessibile
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la nostra velocità
7
8  func _physics_process(delta):
9      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del vettore di input
10
11     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
12     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
13     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
14
15     # normalizziamo il vettore per ottenere una direzione
16     input_vector = input_vector.normalized()
17
18     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
19         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta muovendo il personaggio
20     else:
21         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio
22
23     move_and_slide() # muoviamo il personaggio sullo schermo

```

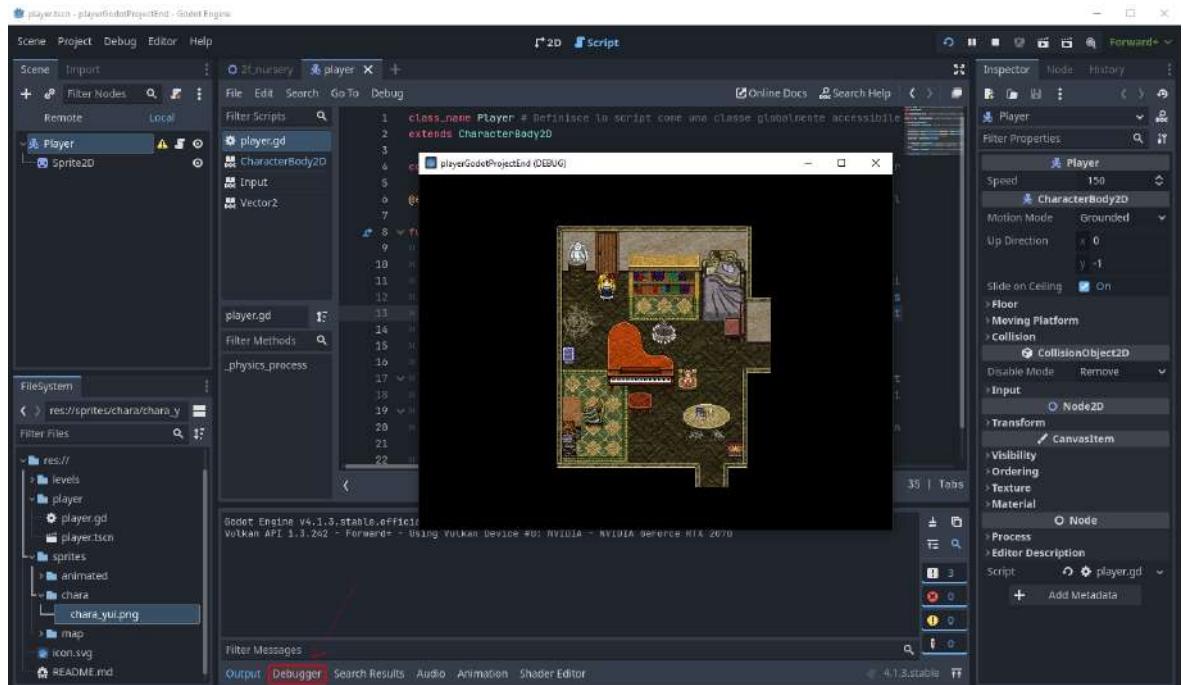
A red arrow points to the word `delta` in the `_physics_process(delta):` function signature.

l'editor di Godot ci darebbe immediatamente errore



Questo perché, come ci informa Godot stesso nel messaggio di errore, la firma del metodo "`_physics_process`" non è `_physics_process()` ma bensì `_physics_process(delta)`

Rieseguendo il gioco, il messaggio di `Warning` è scomparso



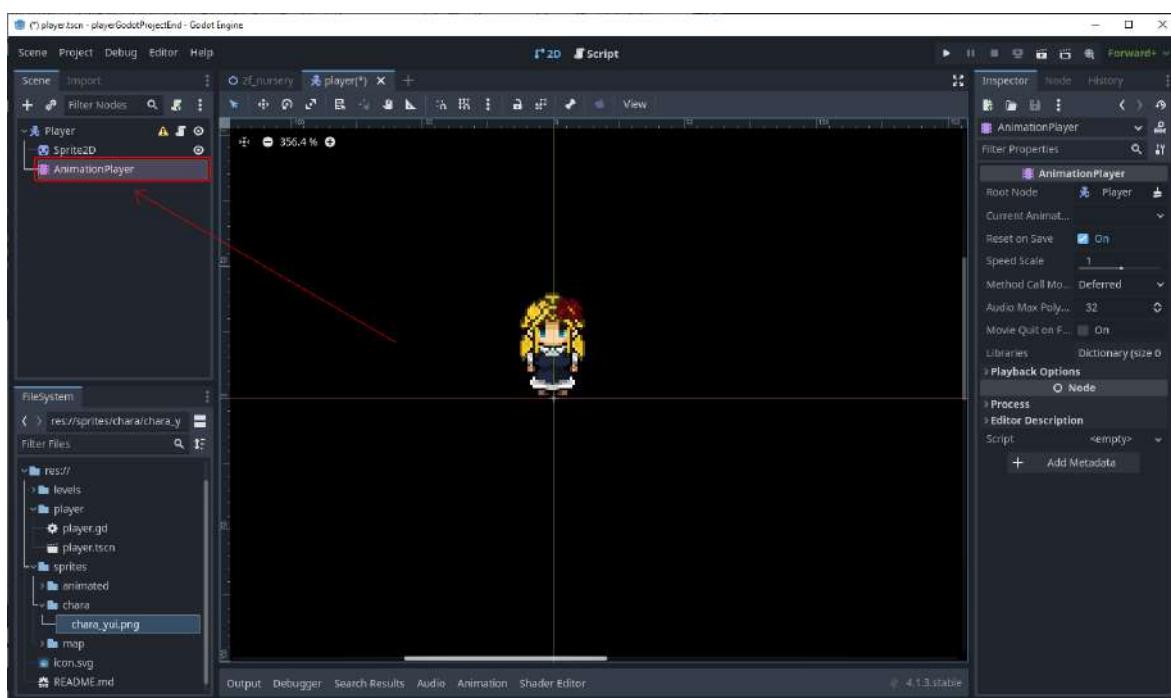
#### 2.4.4 Animare il Player

Ora che possiamo muovere a nostro piacimento il player nella stanza, sarebbe carino se quest'ultimo "camminasse" e non rimanesse fermo come una pietra.

Prestiamo attenzione al fatto che il nostro personaggio non camminerà sempre, ma bensì solamente quando il giocatore premerà le frecce direzionali. Il comportamento del player può essere dunque diviso in due:

1. *idle*: il player non sta camminando
2. *walk*: il player sta camminando

Per raggiungere il nostro obiettivo di mostrare l'animazione della camminata del player, per prima cosa aggiungiamo un nodo **AnimationPlayer** come figlio della nostra scena **Player**

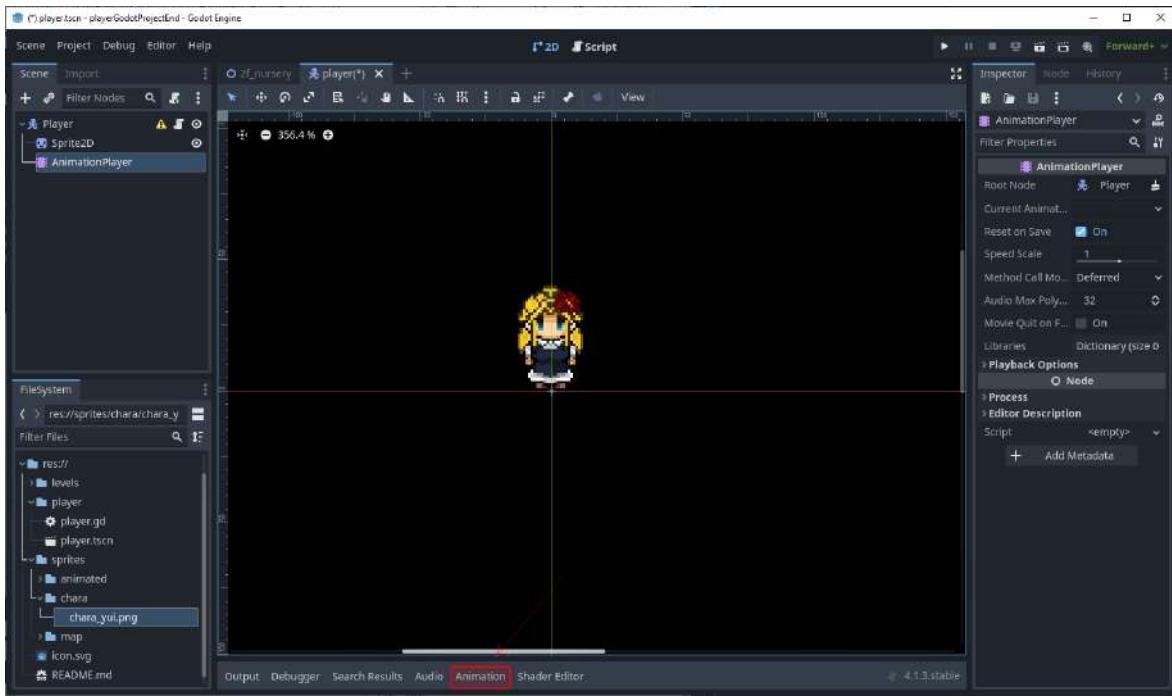


Un nodo **AnimationPlayer** è utilizzato per il playback general-purpose di animazioni; noi, infatti, lo utilizzeremo per comporre le animazioni di *walk* (camminata) e di *idle* a partire dalle sprite dello spritesheet del nostro personaggio.

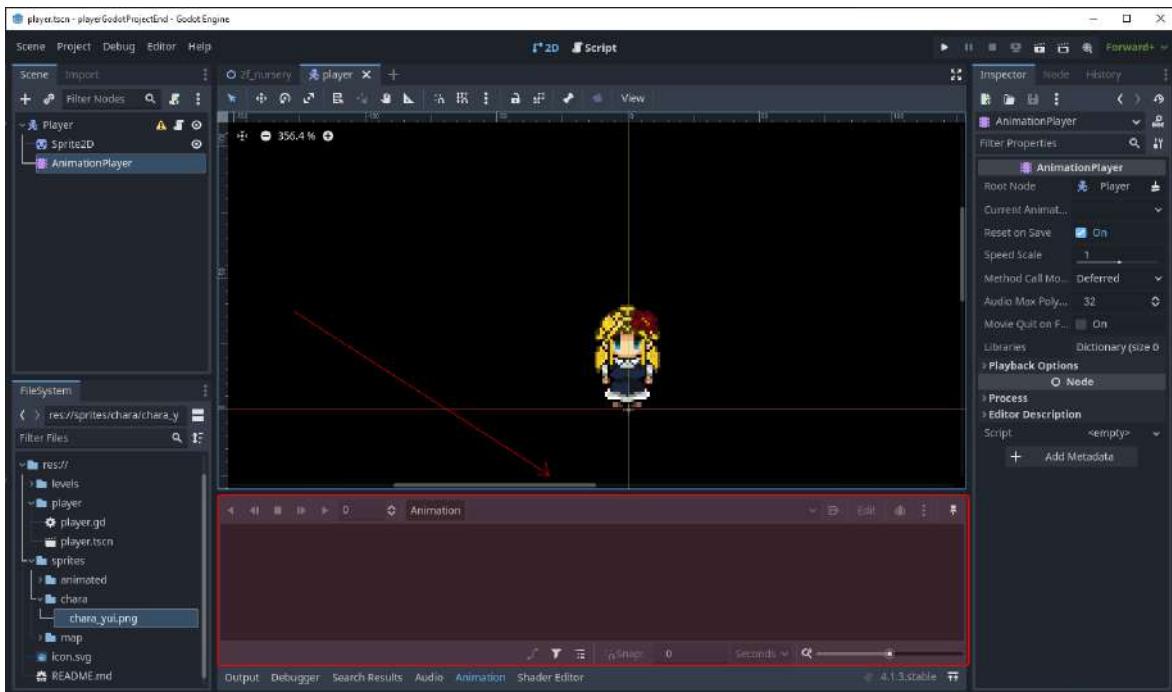
Partiamo con l'animazione di *walk*.

## CHAPTER 2. GODOT

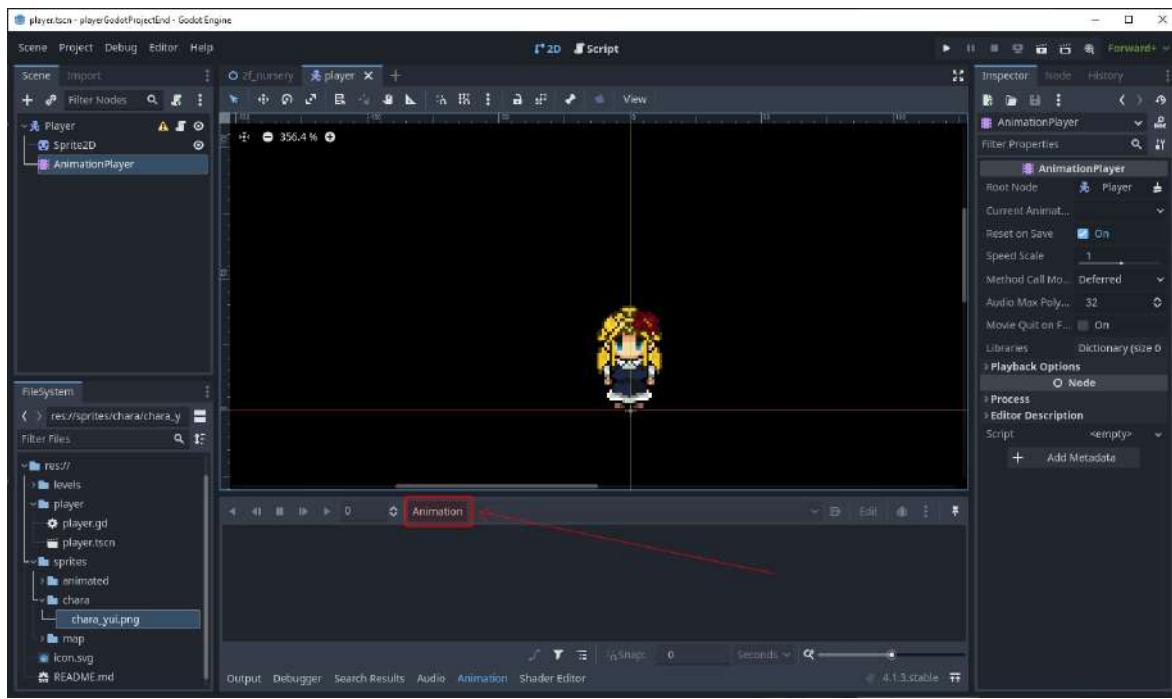
Clicchiamo sulla voce **Animation**



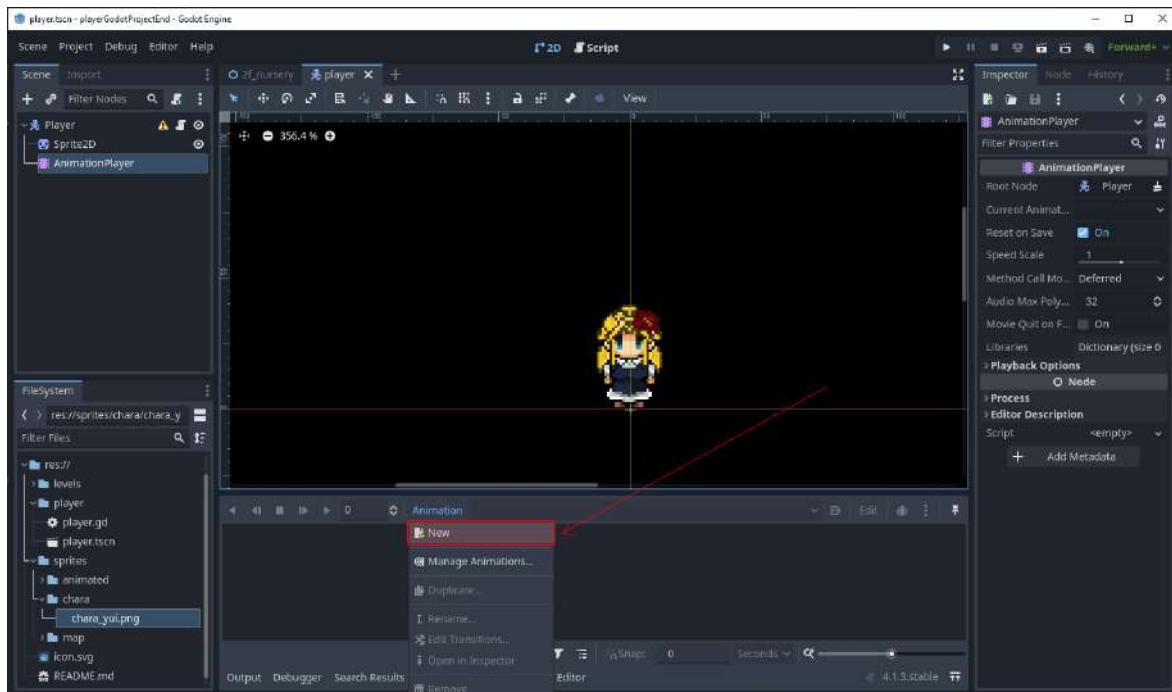
si aprirà la schermata di gestione delle animazioni



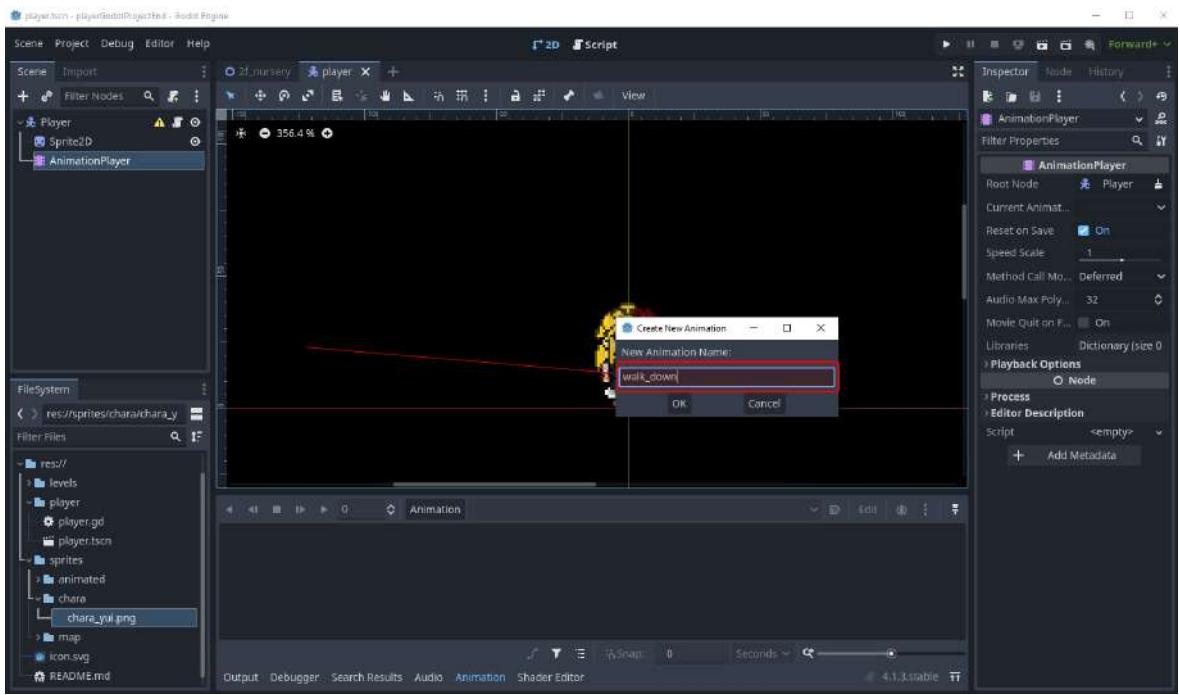
clicchiamo sul pulsante **Animation**



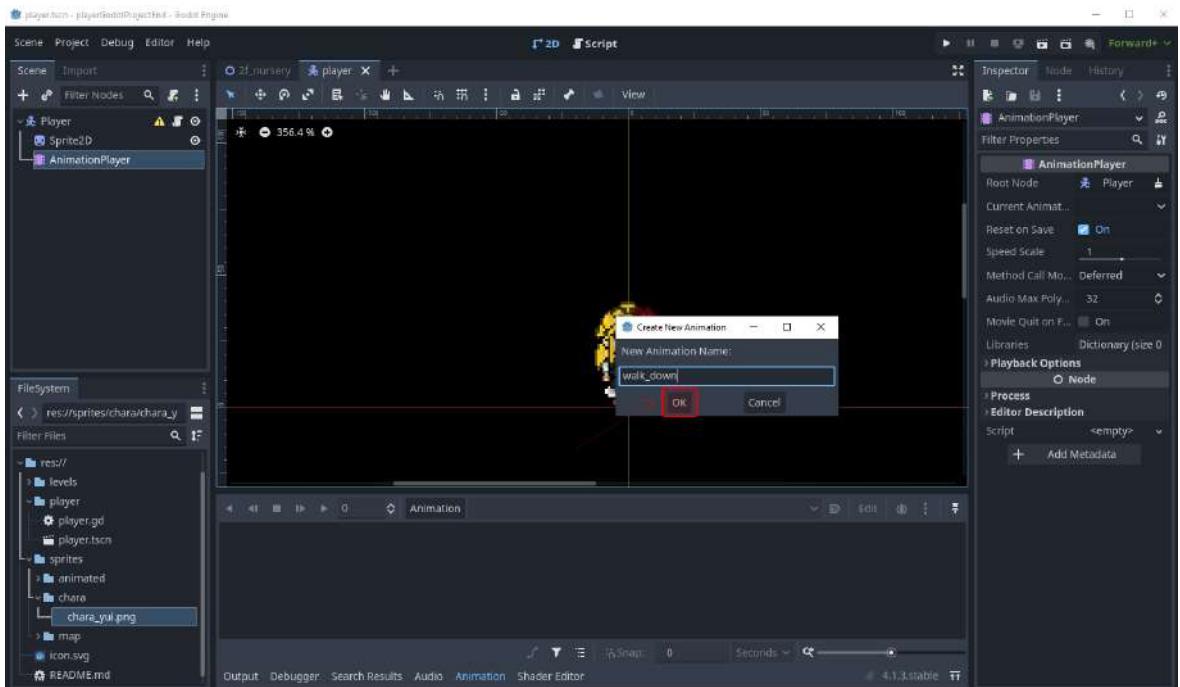
clicchiamo su **New**



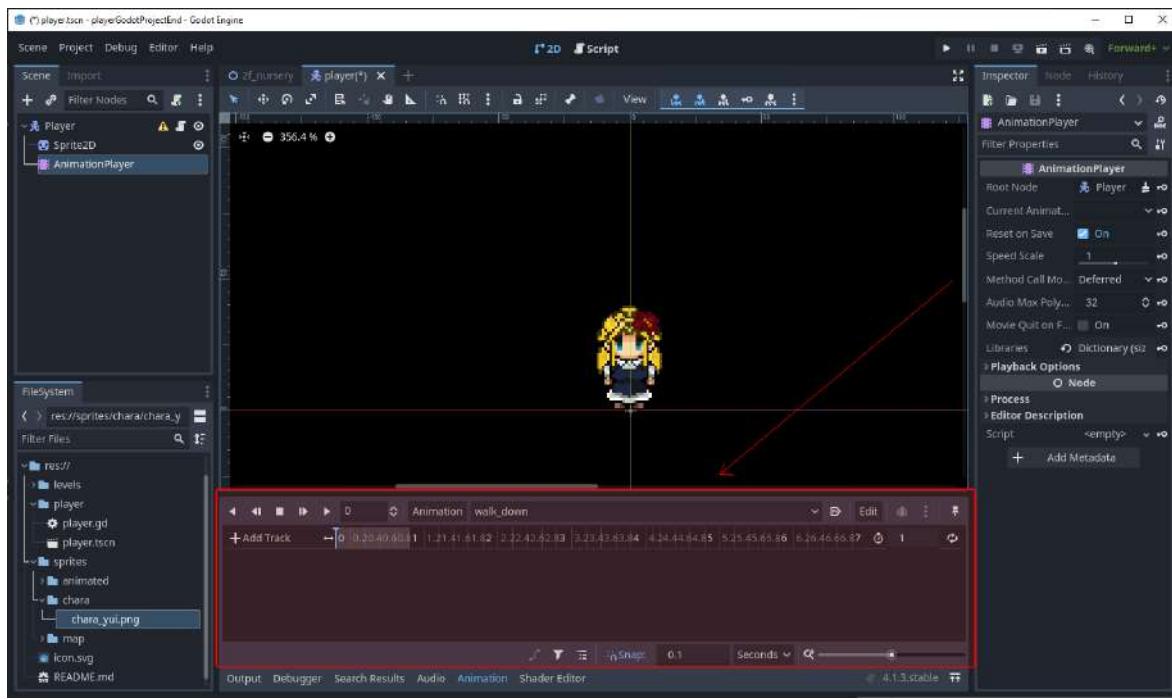
rinomiamo **walk\_down** la nuova animazione



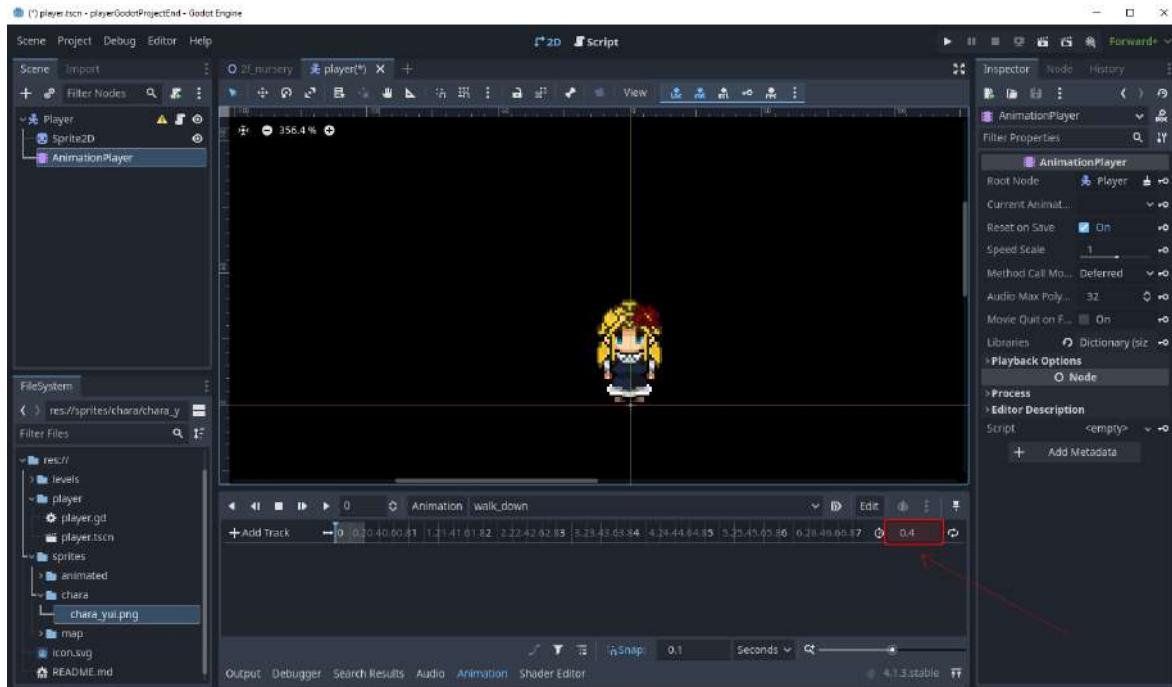
chicchiamo su **Ok**



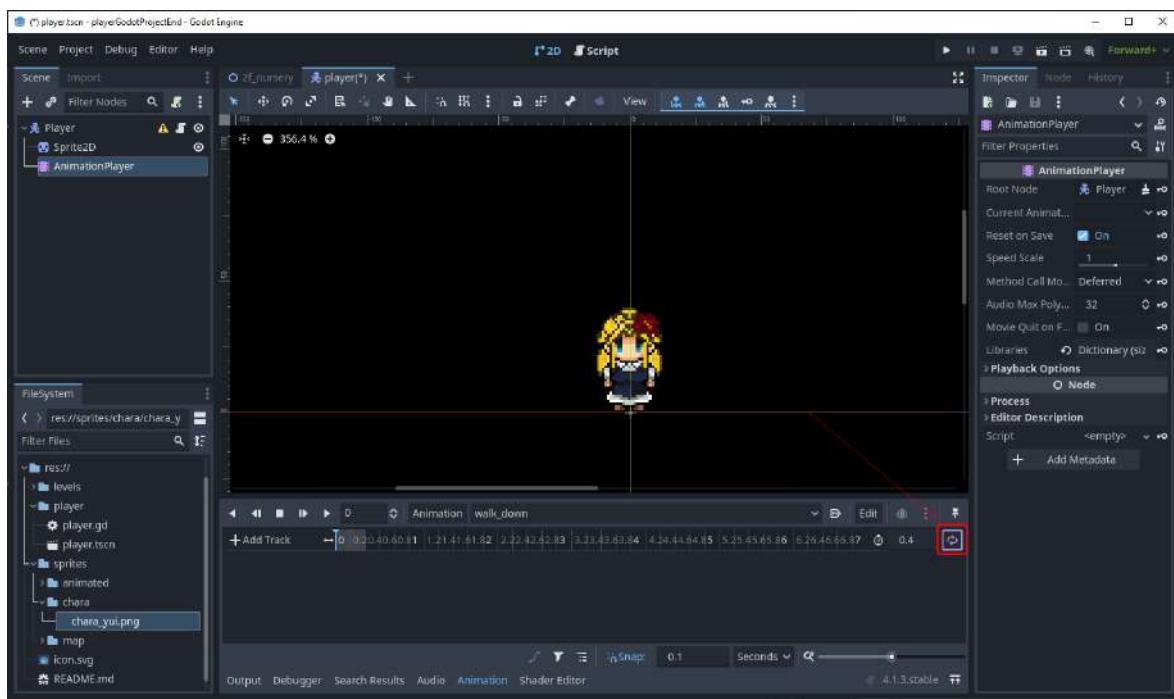
verrà aperta in automatico la schermata con la nostra nuova animazione



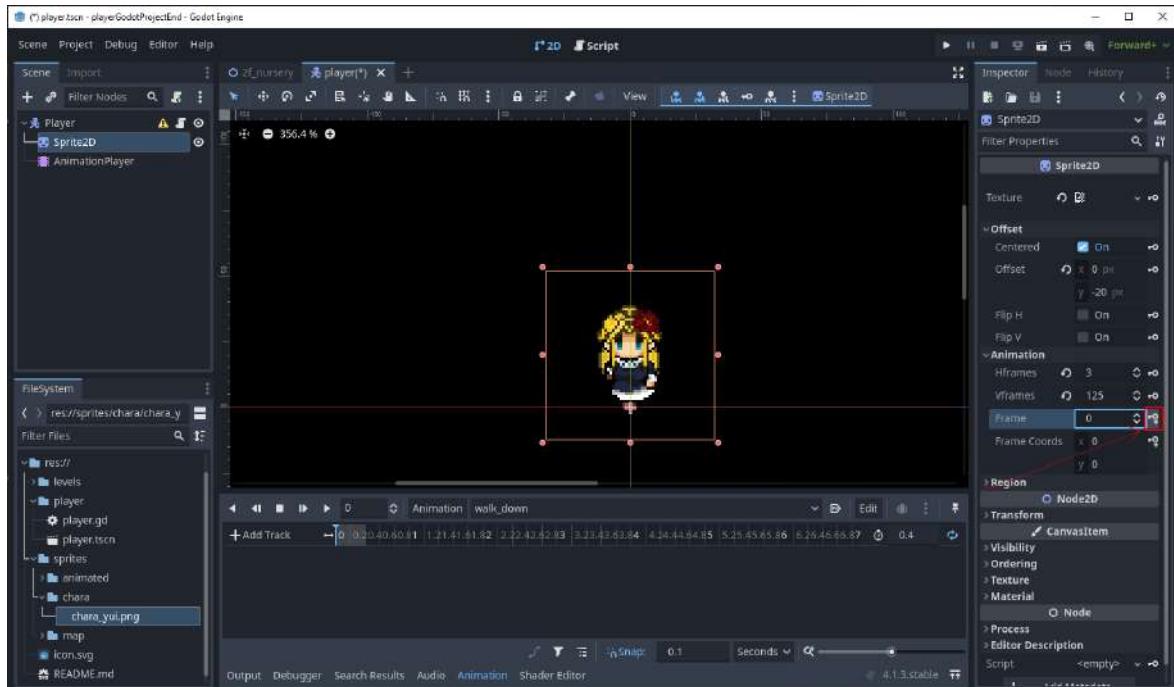
impostiamo la lunghezza dell'animazione a **0.4** secondi



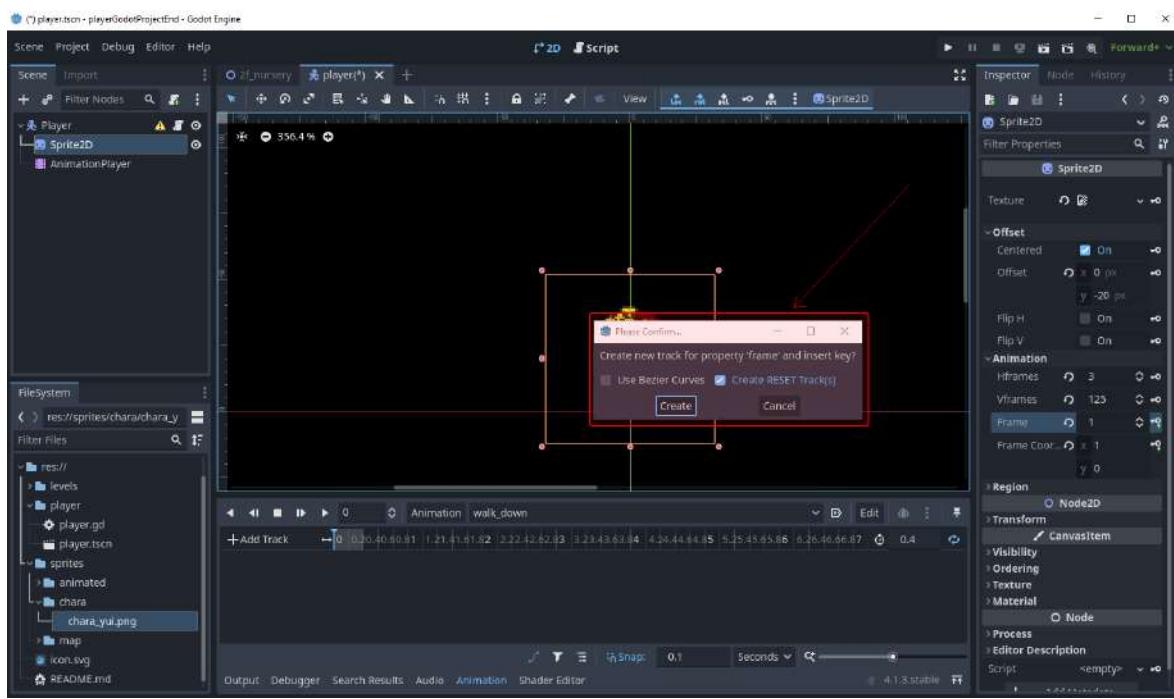
abilitiamo il loop dell'animazione



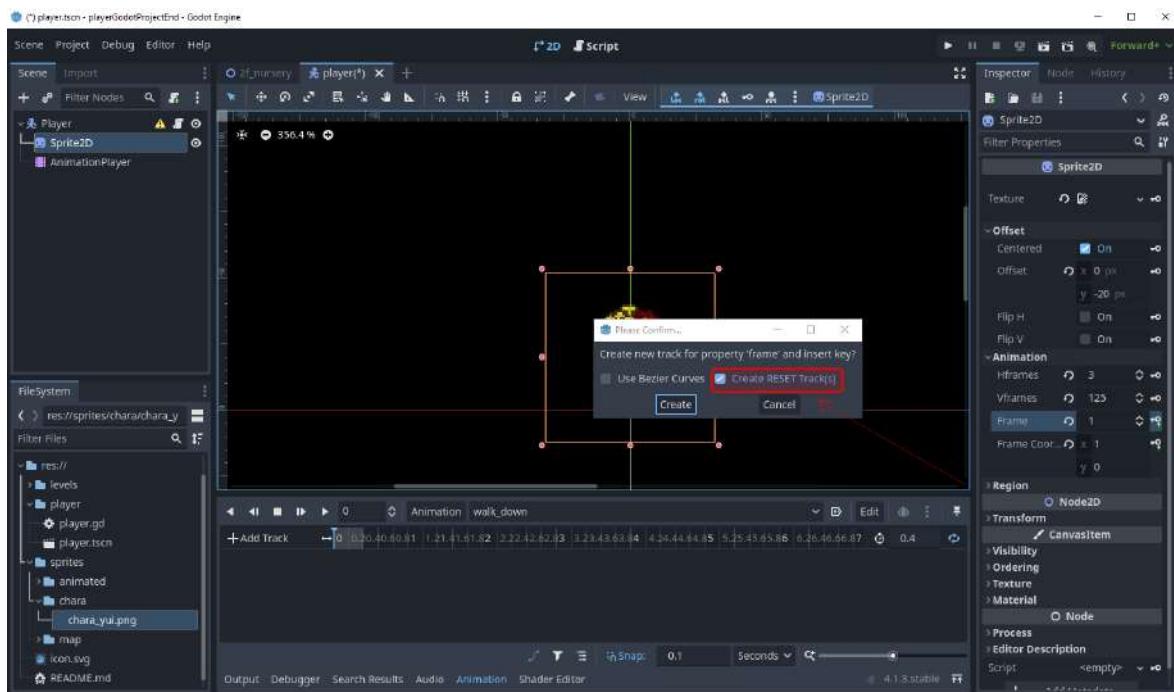
dopo aver cliccato sul nodo **Sprite2D** impostiamo l'opzione **Frame** a **0** e clicchiamo sull'icona a forma di chiave



si aprirà la finestra di conferma

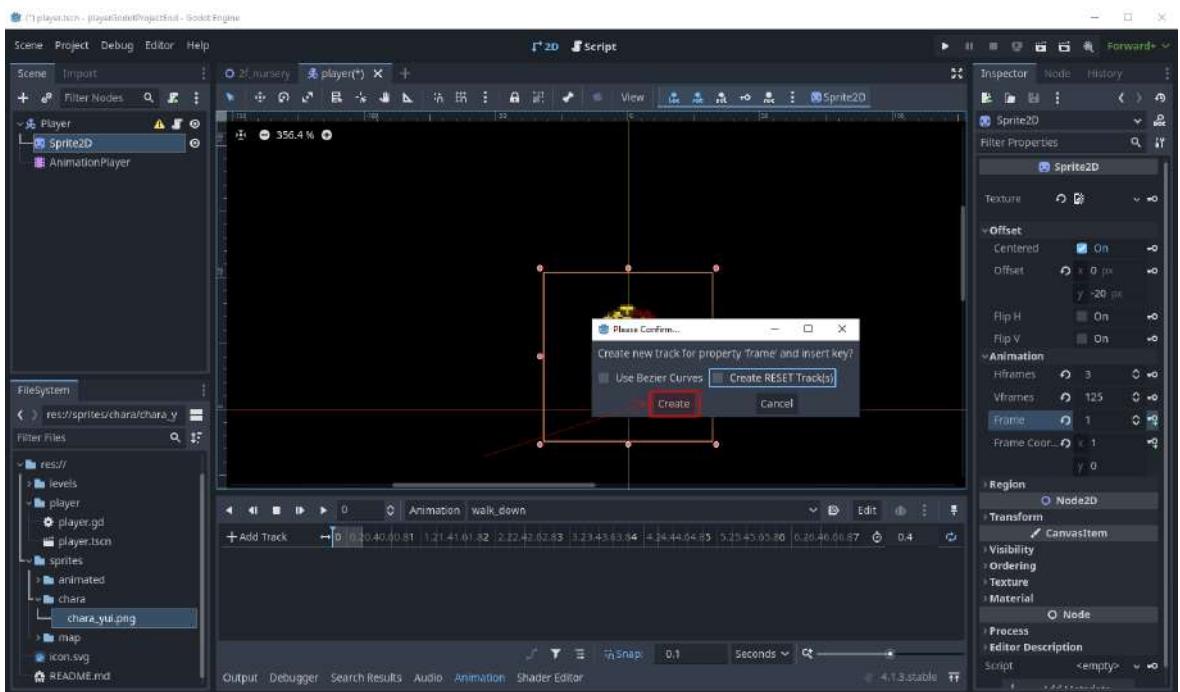


togliamo la spunta a **Create RESET Track(s)**

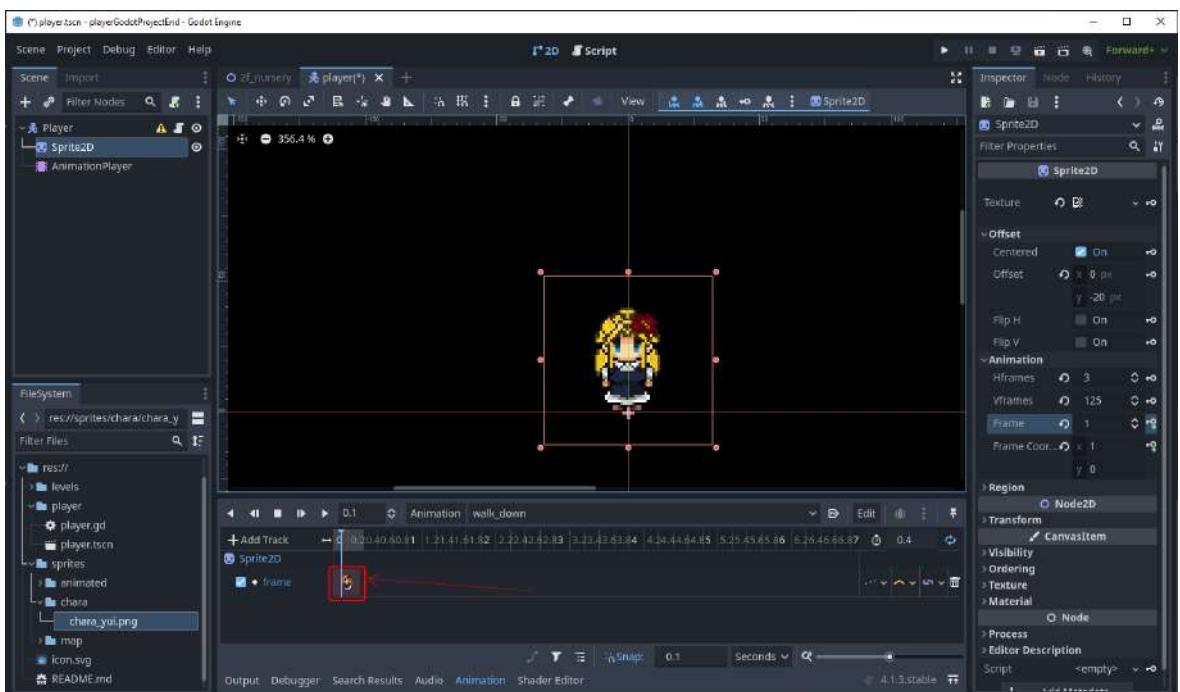


clicchiamo su **Create**

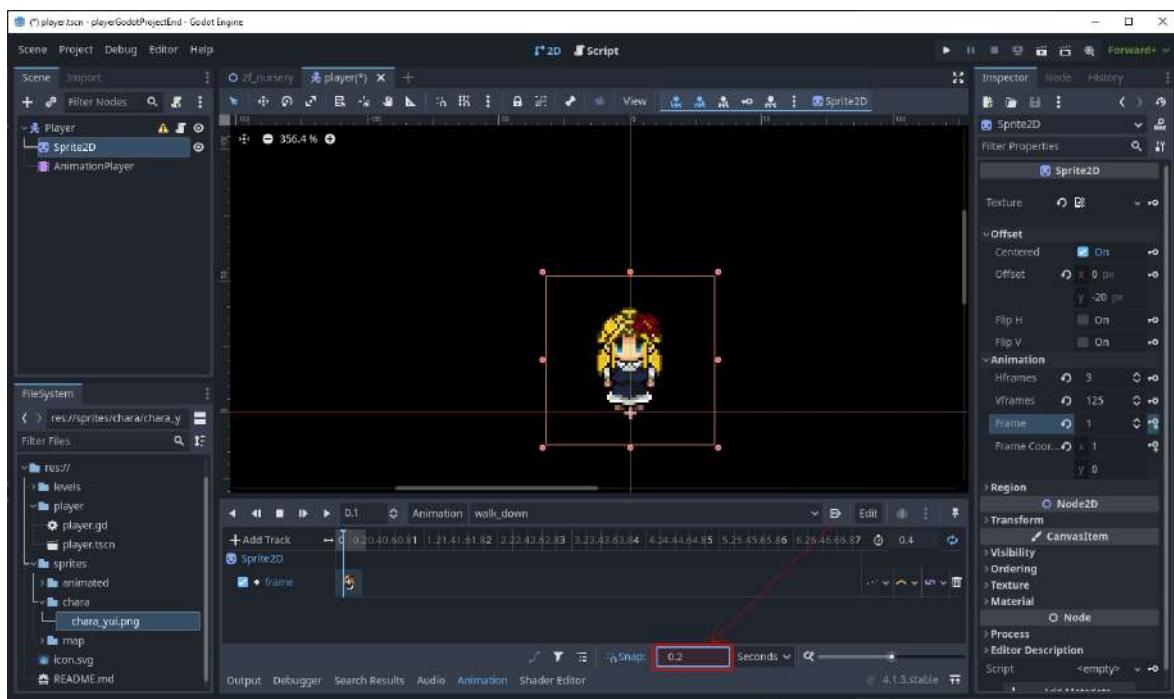
## CHAPTER 2. GODOT



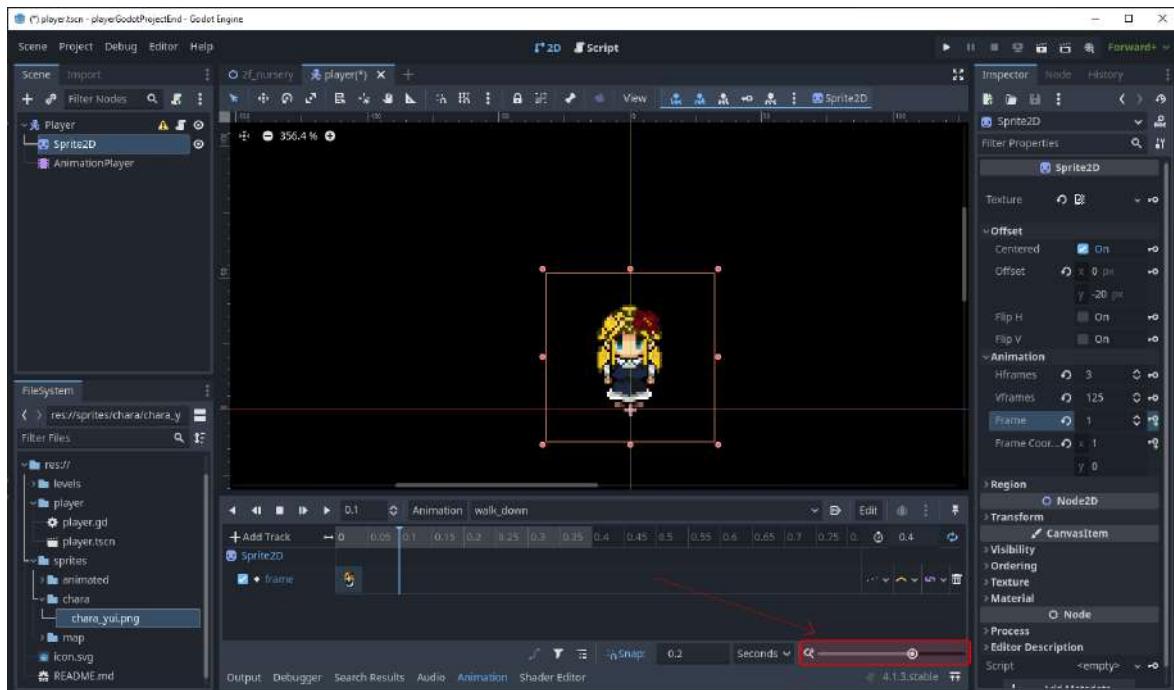
questo creerà un nuovo keyframe nella nostra animazione



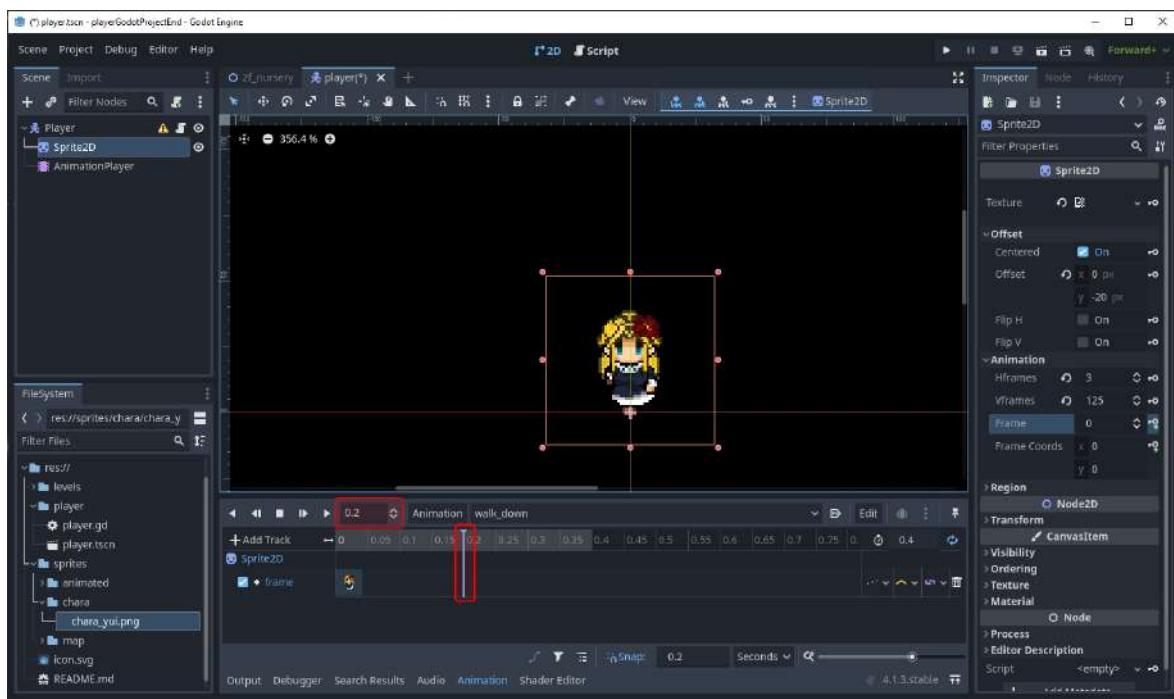
impostiamo lo snap a **0.2** secondi



aggiustiamo lo zoom



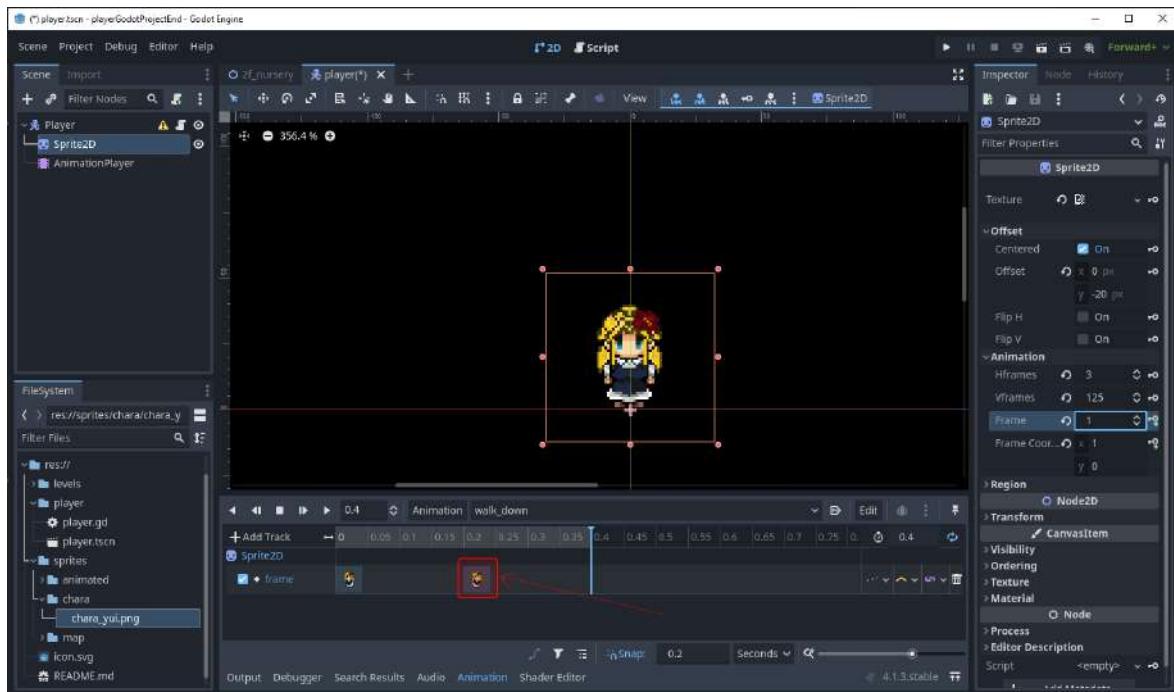
spostiamo la barra del tempo su **0.2** secondi.



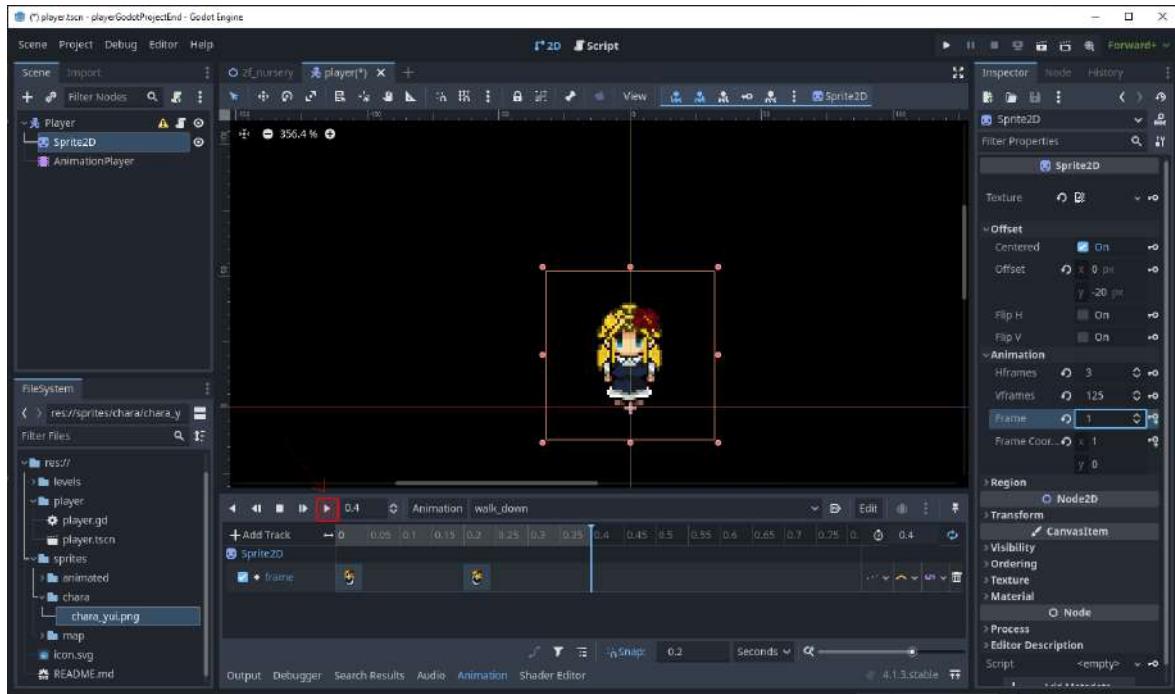
### ! Nota

Avendo impostato lo snap a **0.2** secondi, la barra del tempo andrà avanti di **0.2** secondi alla volta in ogni movimento.

impostiamo a **2** il valore dell'opzione **Frame** della **Sprite2D** e clicchiamo nuovamente sull'icona a forma di chiave per creare un nuovo frame



se adesso provassimo ad eseguire l'animazione tramite il tasto play

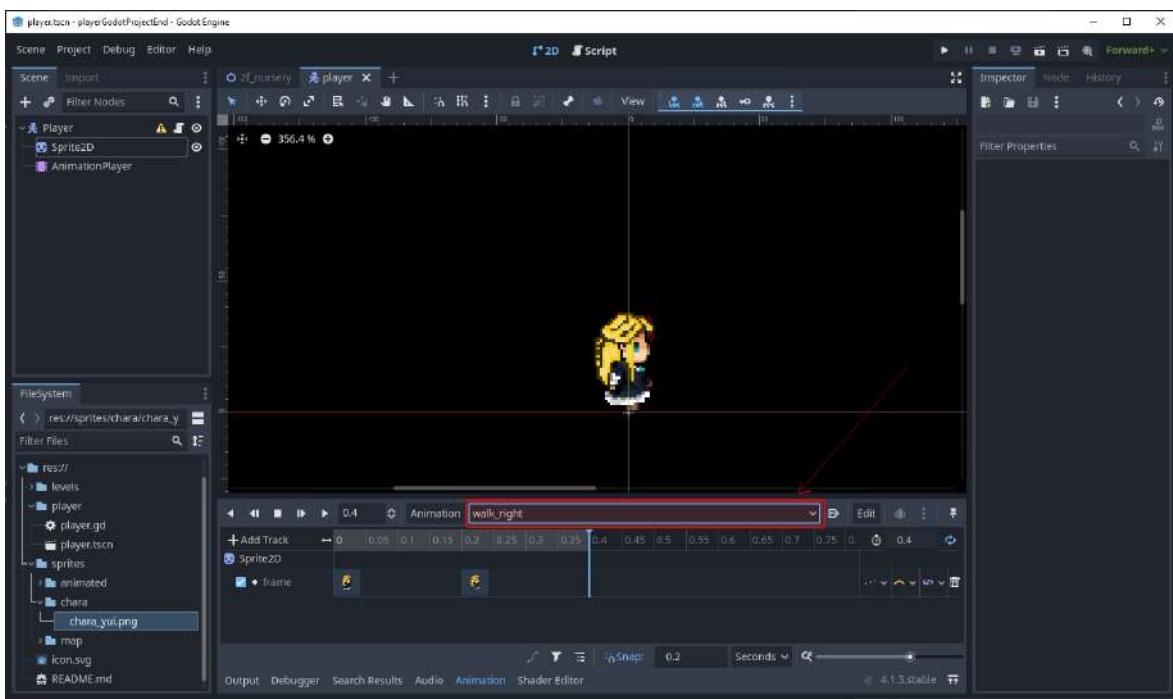


la successione delle due sprite farà sembrare che il player stia camminando verso il basso in loop.

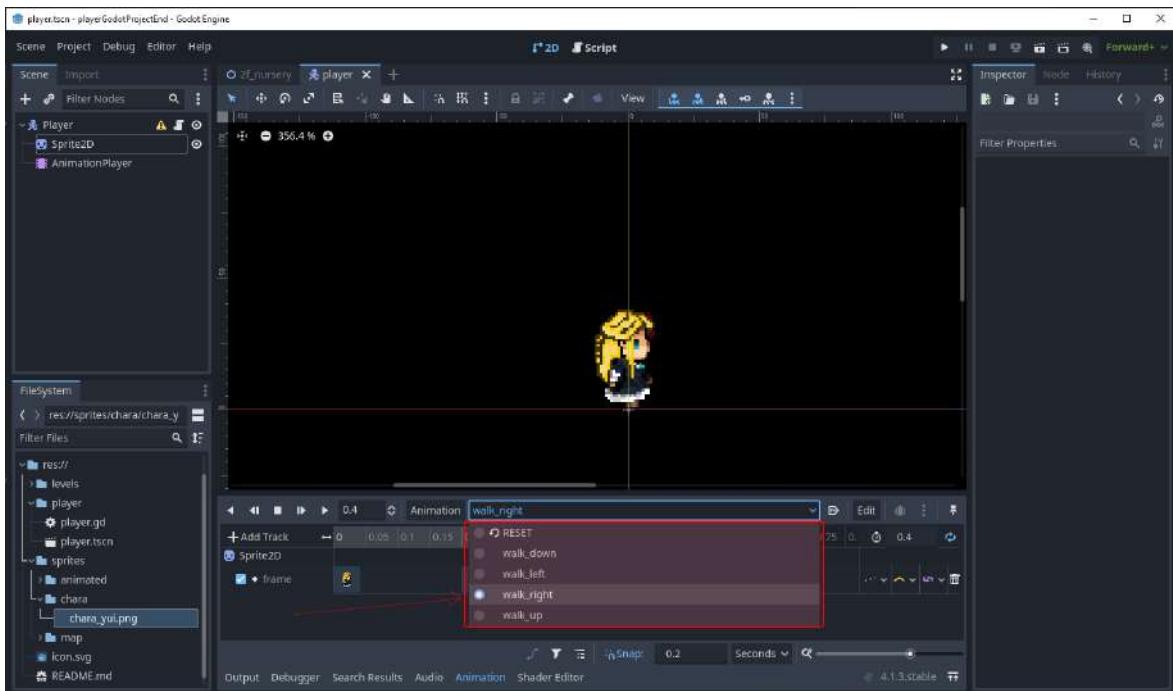
A questo punto, creiamo allo stesso modo un'animazione per ognuna delle direzioni rimanenti:

- `walk_left` con i frame 3 e 5
- `walk_right` con i frame 6 e 8
- `walk_up` con i frame 9 e 11

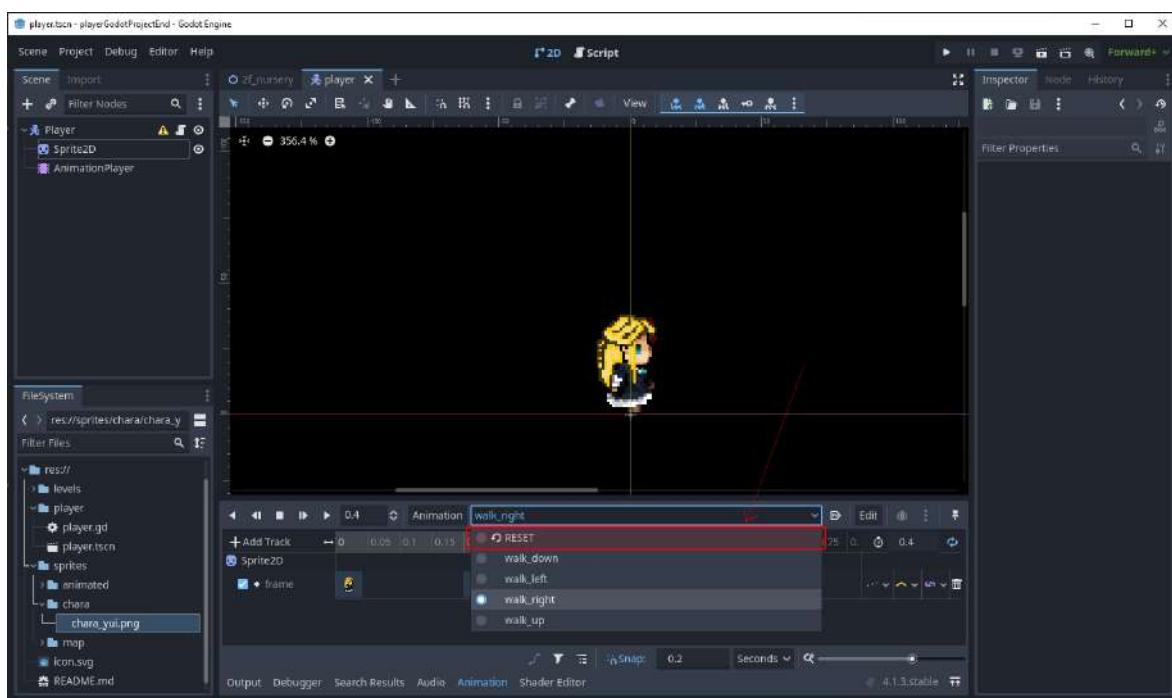
Per cambiare animazione selezionata, basta cliccare sul nome dell'animazione corrente



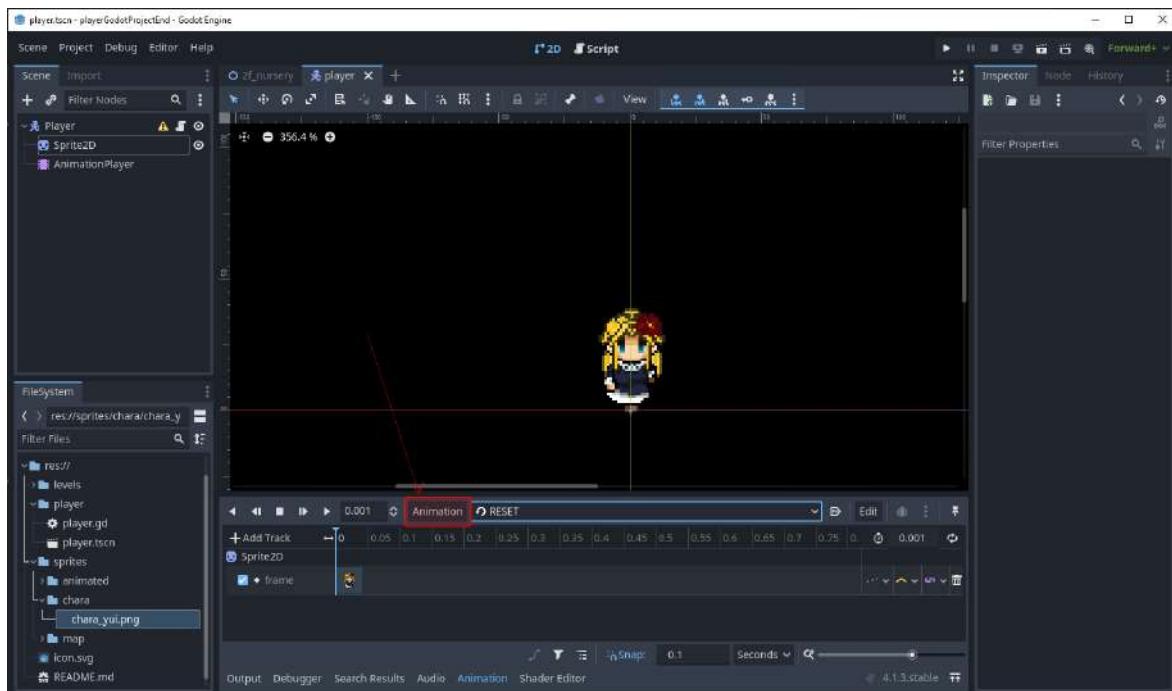
e selezionare l'animazione desiderata dal drop down menu



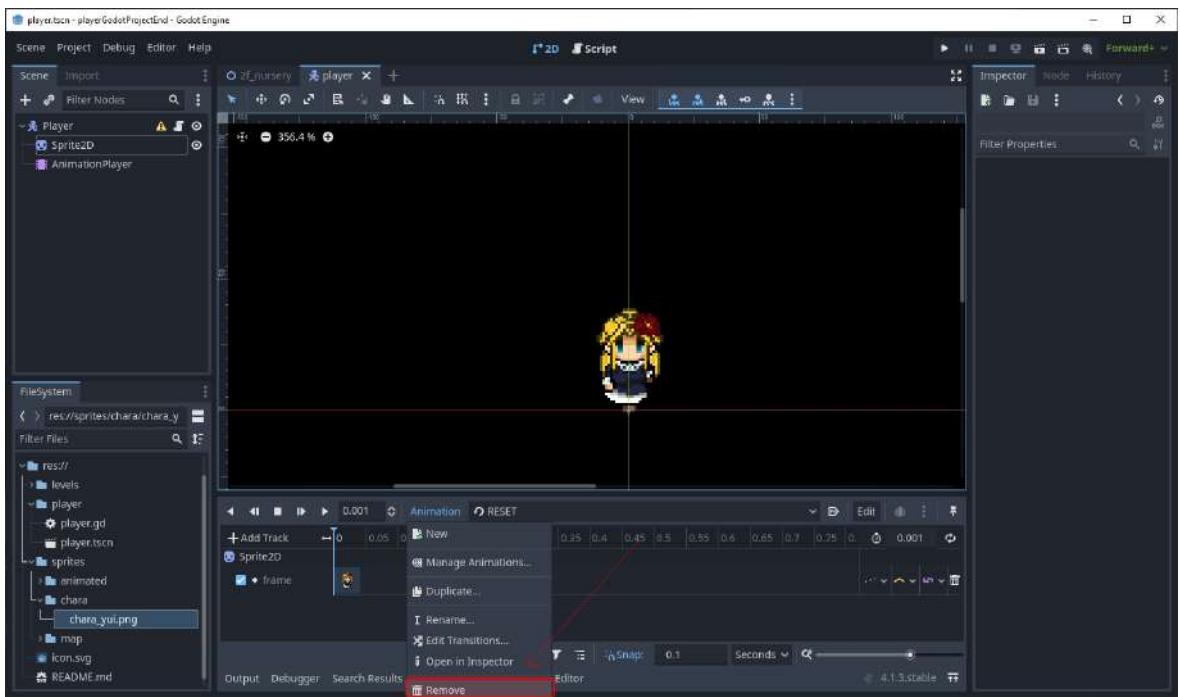
Inoltre, se, come in questo caso, Godot ha autogenerato un'animazione chiamata **RESET**



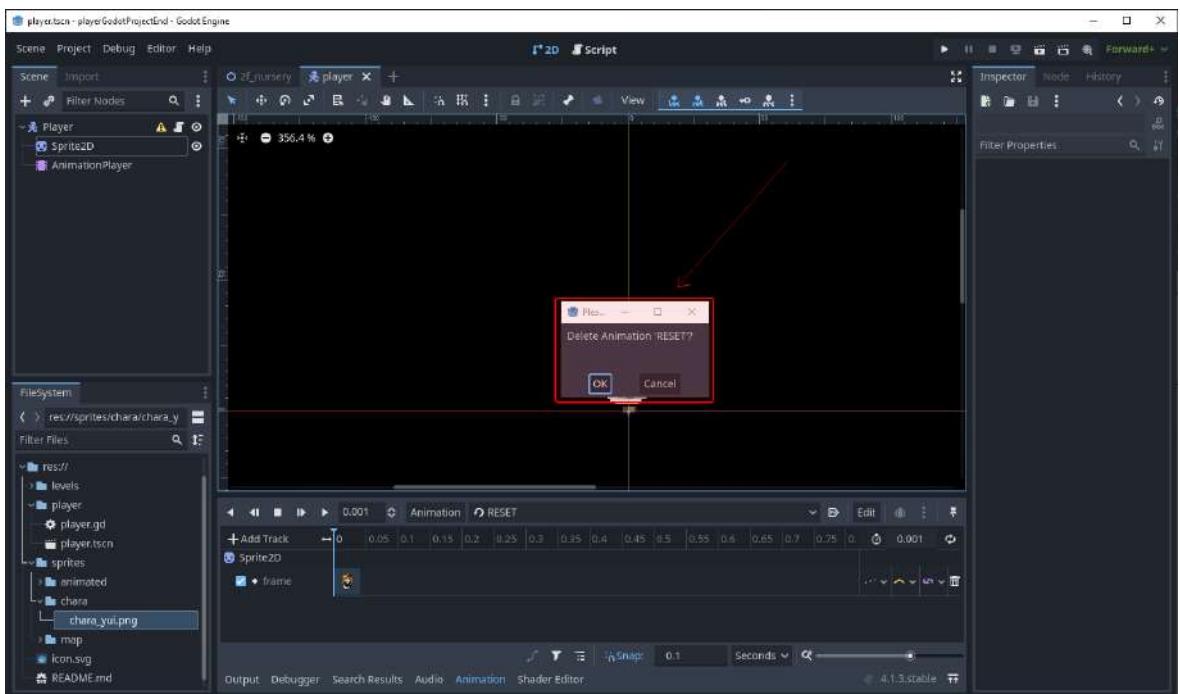
è consigliabile cancellarla. Per farlo la selezioniamo dal menu a tendina e poi clicchiamo il pulsante **Animation**



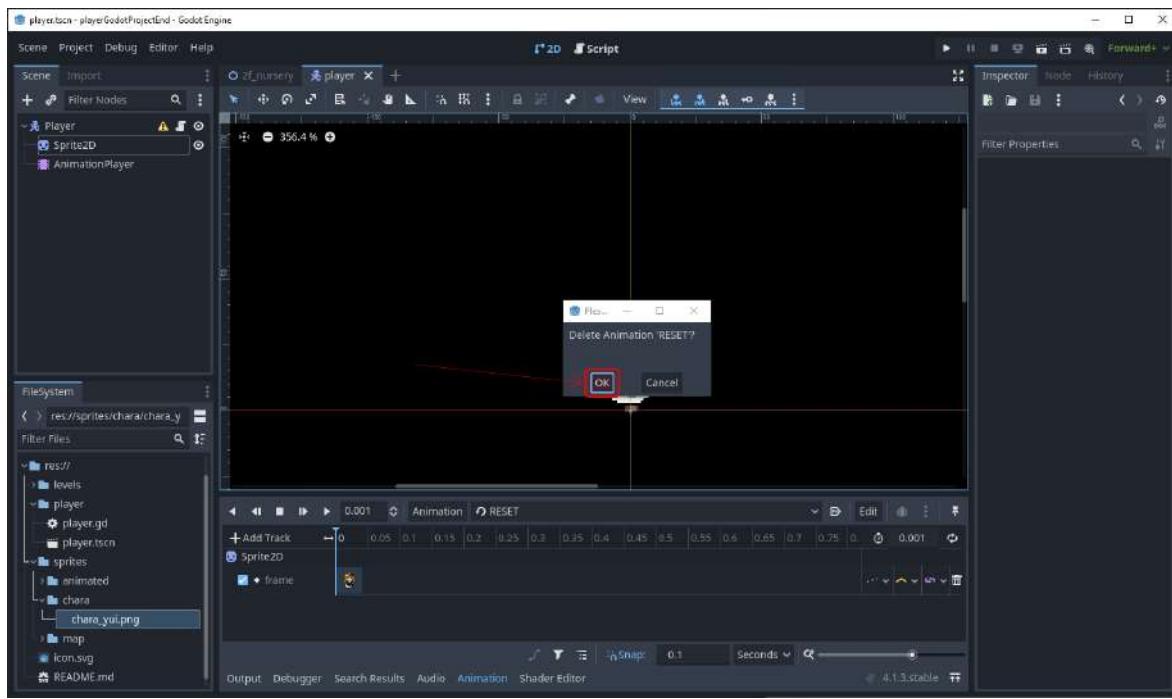
clicchiamo **Remove**



si aprirà la schermata di conferma



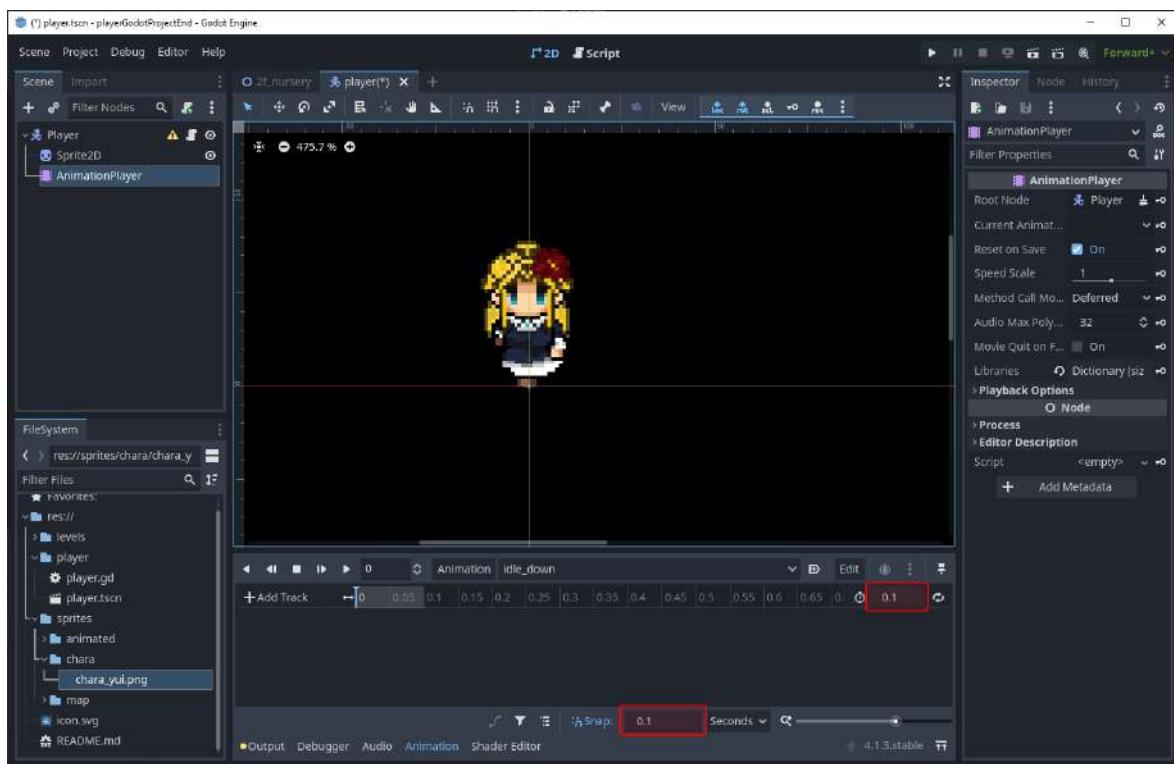
clicchiamo su **Ok** per eliminare definitivamente l'animazione **RESET**



L’animazione di **RESET**, come dice il nome stesso, è un’animazione che Godot utilizza per resettare una traccia all’avvio del gioco. Poiché tuttavia quest’animazione potrebbe creare problemi con delle funzioni che utilizzeremo durante le cutscene, noi preferiamo sempre farne a meno.

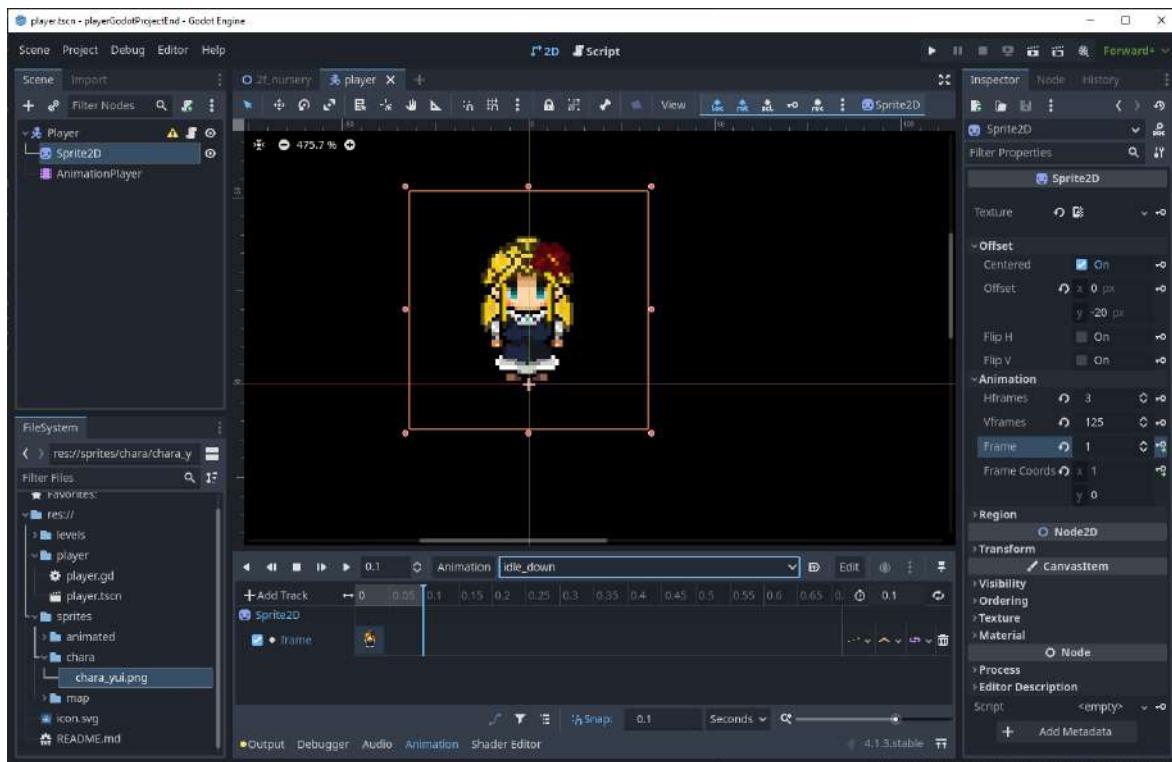
Occupiamoci adesso dell’animazione di *idle*. Il concetto è lo stesso che abbiamo utilizzato nell’animazione di *walk*, con le uniche differenze che ogni frame dura 0.1 secondi e non serve il loop perché ogni singola animazione è formata da un unico frame.

Per ogni animazione di *idle* impostiamo dunque a **0.1** secondi il valore di **Snap** e quello della lunghezza dell’animazione.



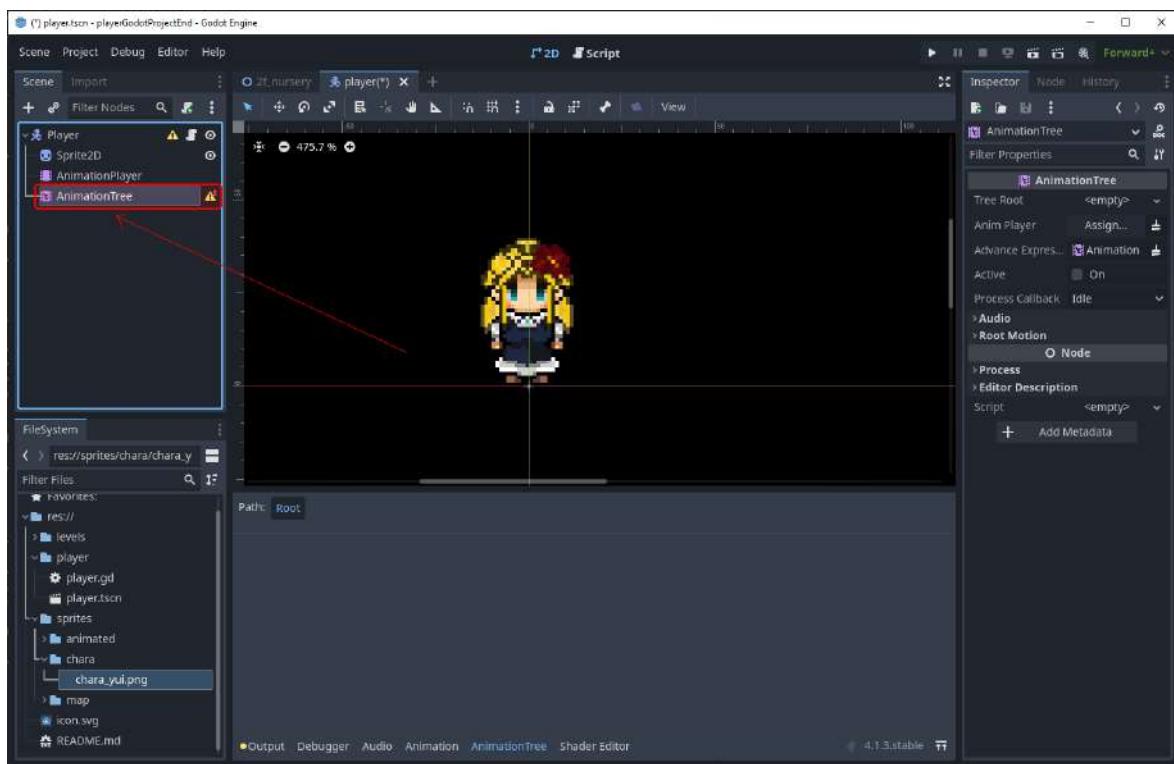
Le animazioni che necessitiamo per l'animazione di *idle* sono dunque:

- `idle_down` con il frame 1
- `idle_left` con il frame 4
- `idle_right` con il frame 7
- `idle_up` con il frame 10

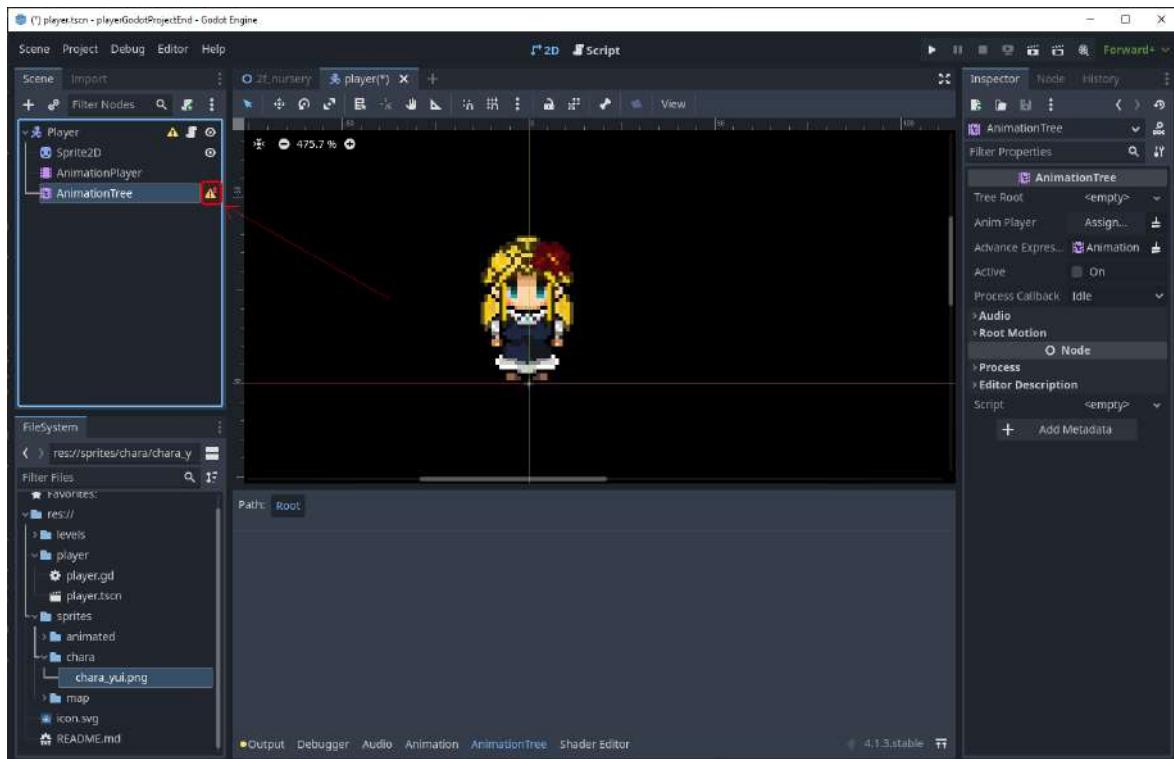


Nonostante adesso disponiamo di tutte le animazioni necessarie, se avviassimo il gioco, il nostro giocatore rimarrebbe ancora fermo mentre sfreccia da un lato all’altro della stanza.

Per collegare correttamente le animazioni di *idle* e di *walk* con lo spostamento effettivo del player nella stanza, necessitiamo innanzitutto di un nodo **AnimationTree**, il quale è utilizzato per effettuare una transizione tra due animazioni di un nodo **AnimationPlayer**

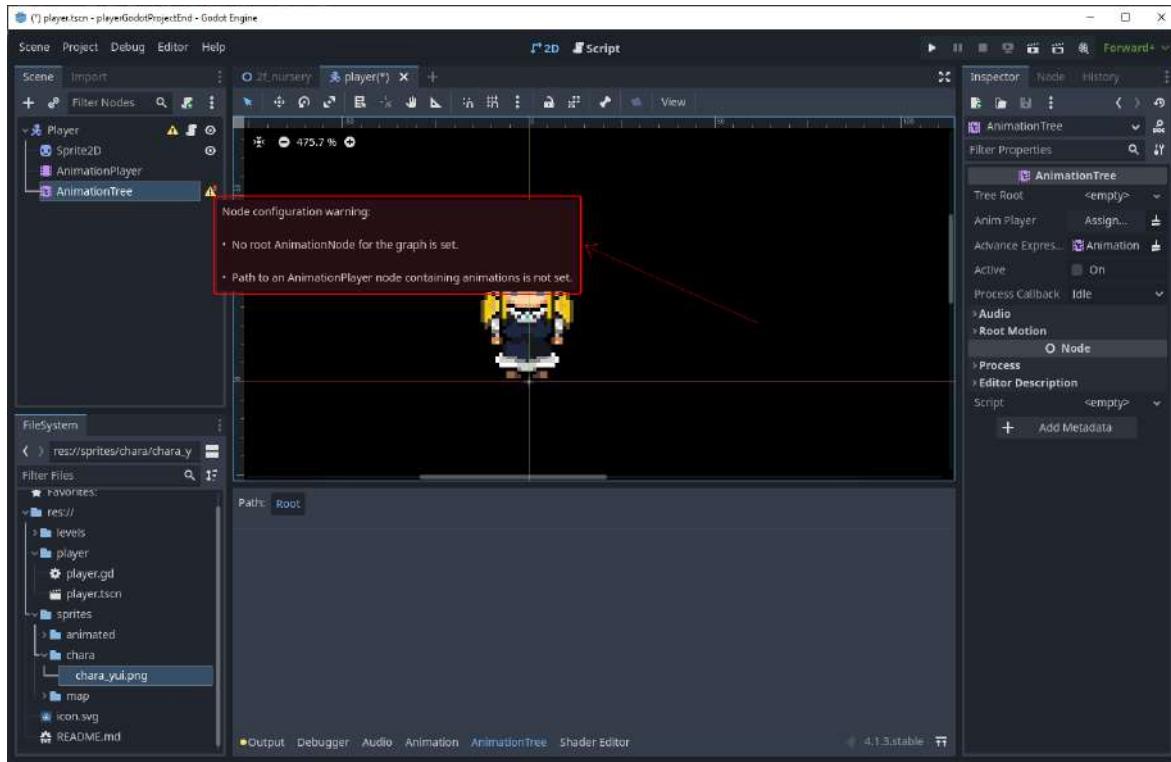


Andando con il mouse sopra l'icona di **Warning** che si trova di fianco a questo nuovo nodo

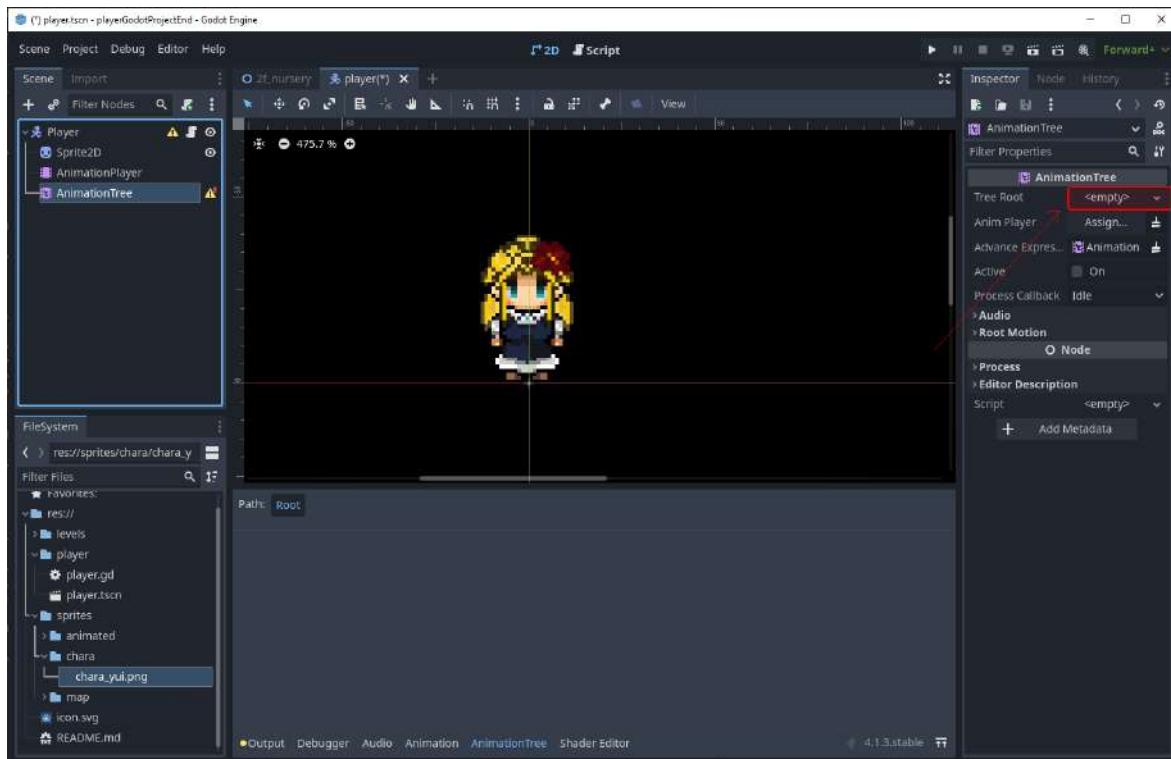


una finestra pop up di Godot ci avverte di non aver ancora configurato il nodo **AnimationTree** nel modo corretto. Questa icona di **Warning** è utilizzata da Godot

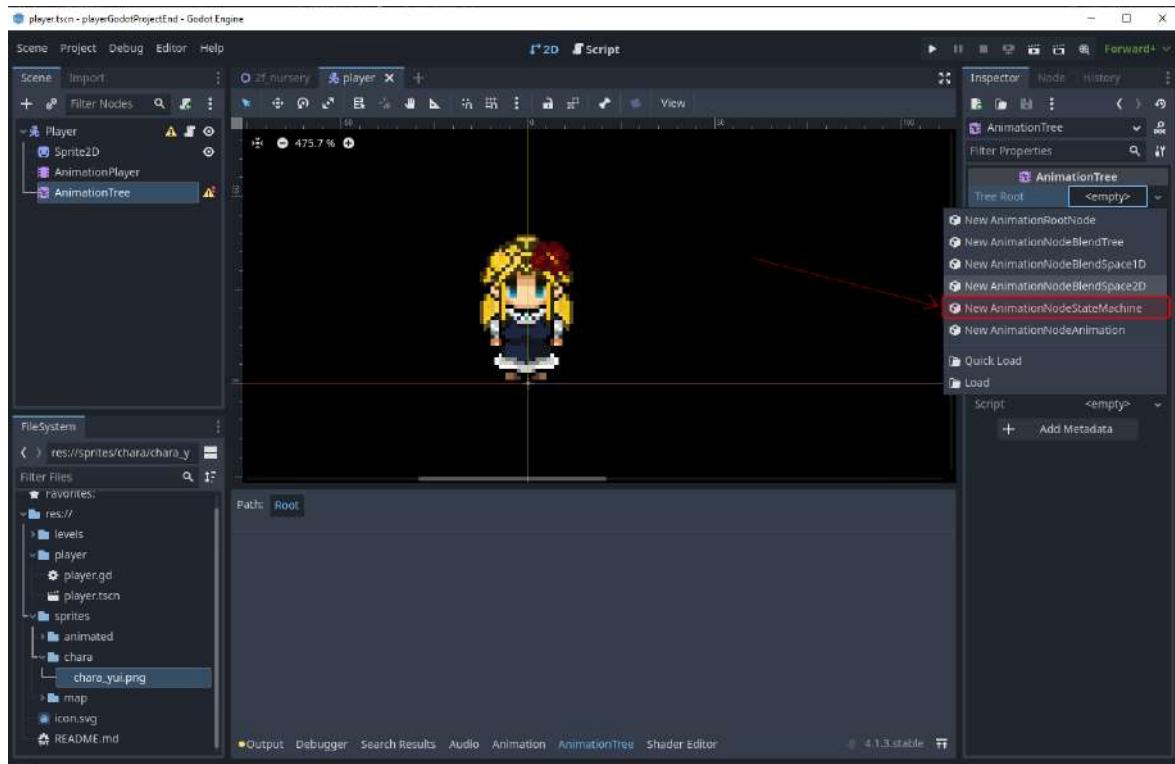
proprio per avvisare lo sviluppatore di dover ancora configurare nodi particolari come ad esempio il nodo **StaticBody2D**



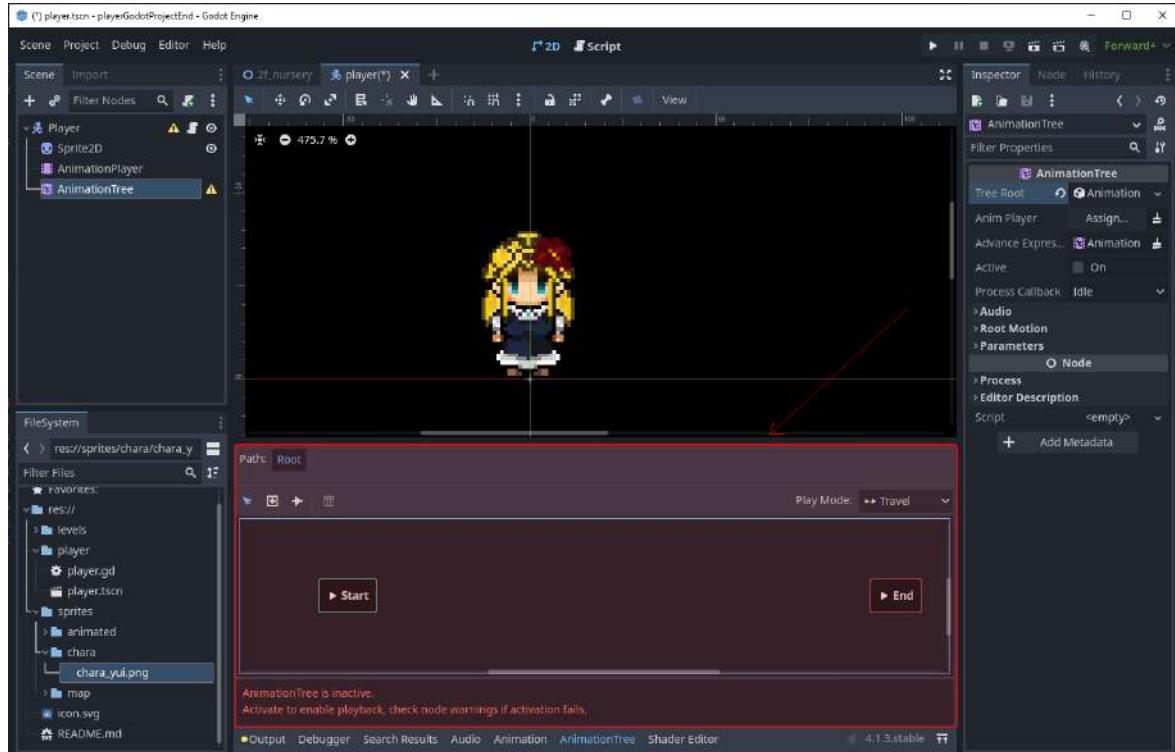
Per configurare correttamente il nodo **AnimationTree**, dopo averlo selezionato, clicchiamo su **<empty>** nella voce **Tree Root**



selezioniamo l'opzione **New AnimationNodeStateMachine**



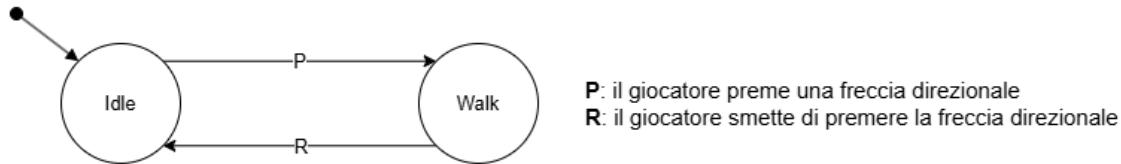
si aprirà una nuova schermata



Una **state machine**, in italiano **macchina a stati**, è un modello di comportamento

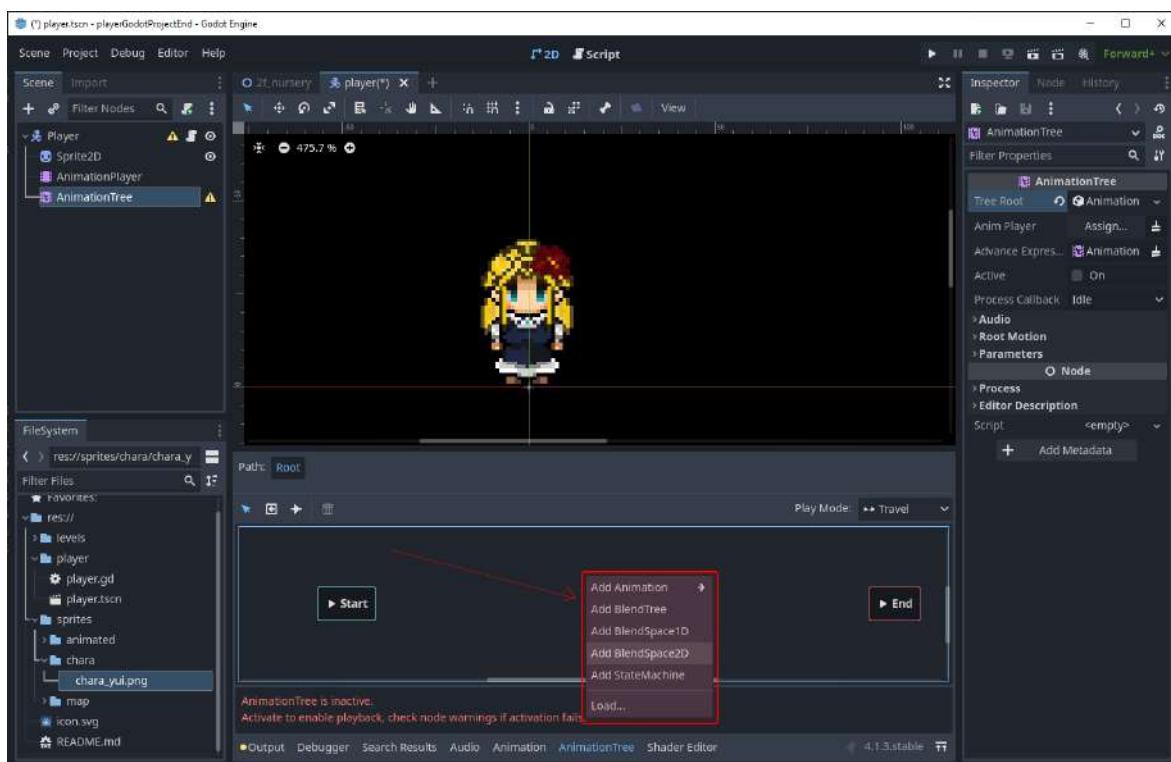
che, poiché consiste di un numero finito di stati, è anche chiamata **finite-state machine (FSM)** o **finite-state automaton (FSA)**, in italiano **macchina a stati finiti** o **automa a stati finiti**. Una macchina a stati esegue transizioni di stato e produce output in base allo stato corrente e a un dato input.

Ripensando ai due comportamenti del nostro player, *idle* e *walk*, ci accorgiamo che il player stesso può in realtà essere visto come una macchina a stati finiti. Il player, all'inizio in stato *idle*, transisce nello stato *walk* nel momento in cui il giocatore preme almeno una freccia direzionale, per poi tornare nuovamente nello stato *idle* quando quest'ultimo smette di premerle tutte.

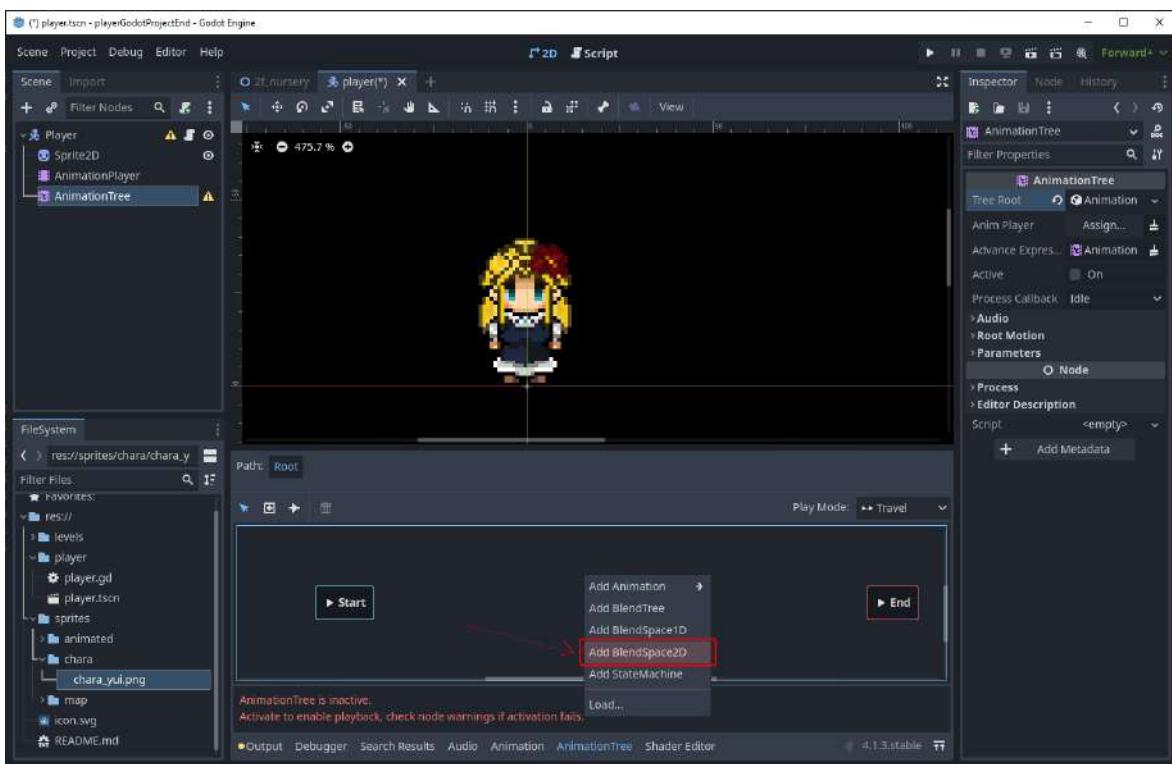


È proprio per questa similarità che utilizziamo un **AnimationNodeStateMachine** per il nodo **AnimationTree**.

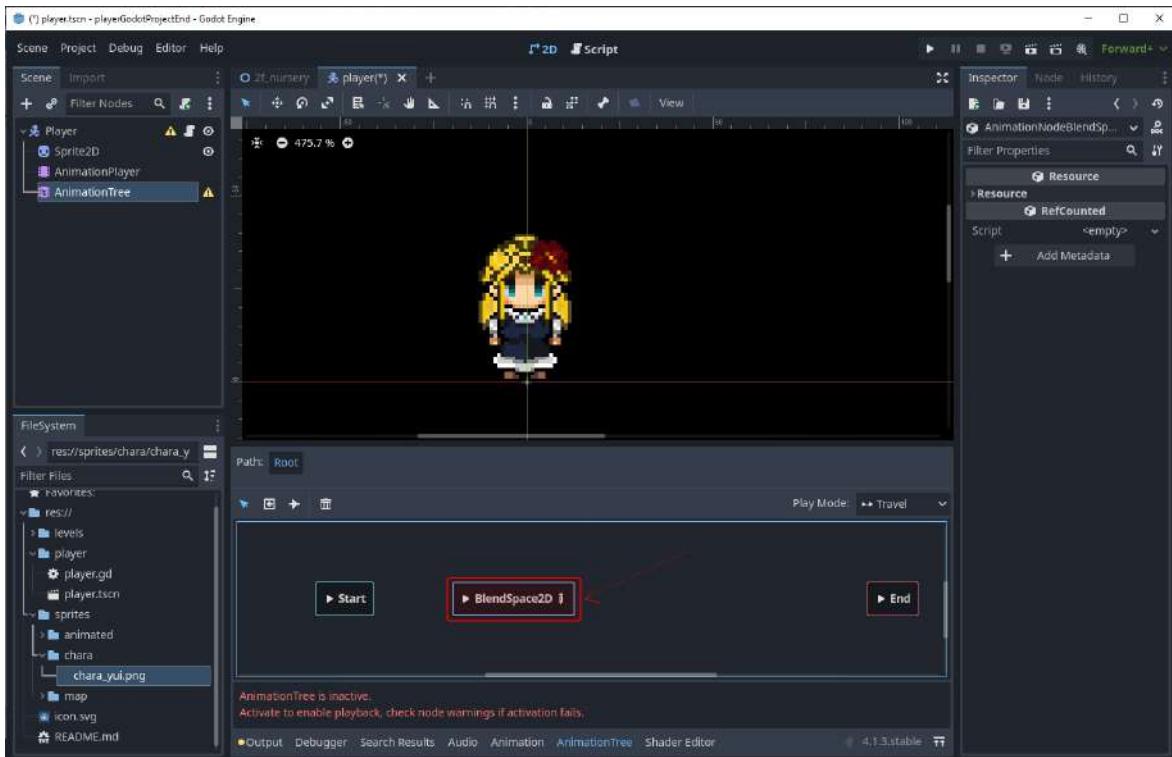
Cliccando con il destro sullo spazio vuoto si aprirà una finestra



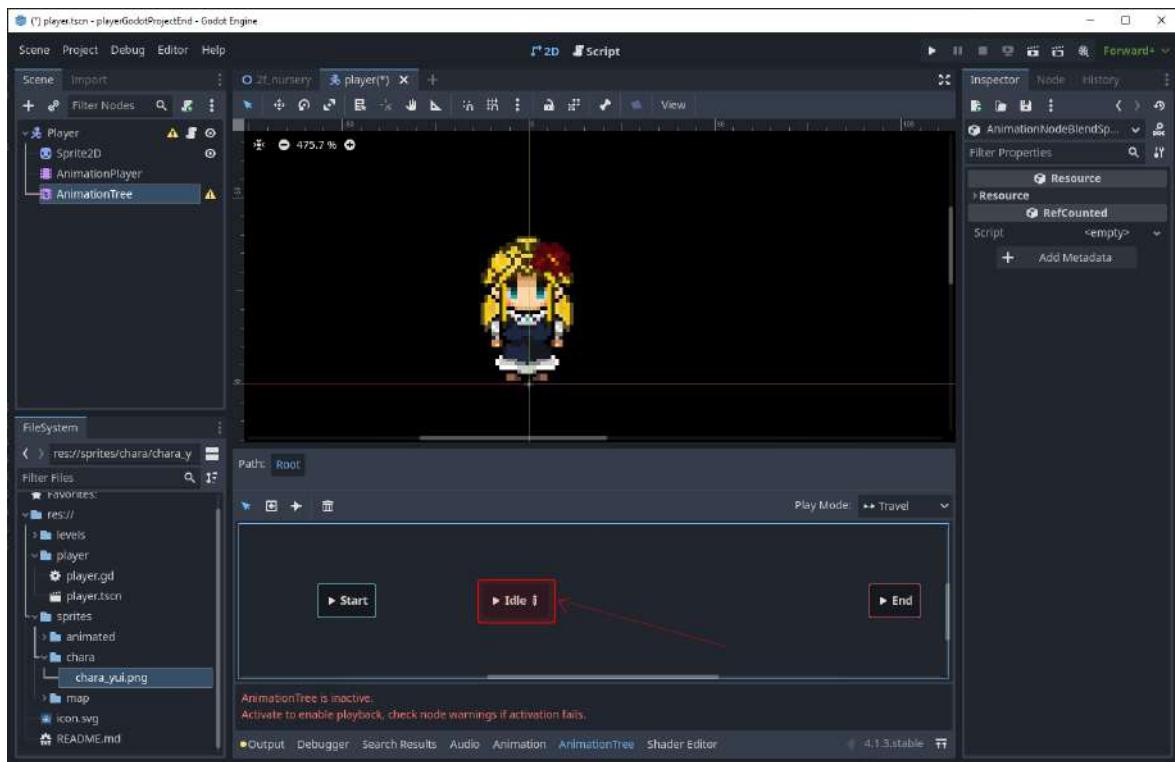
clicchiamo su **Add BlendSpace2D**



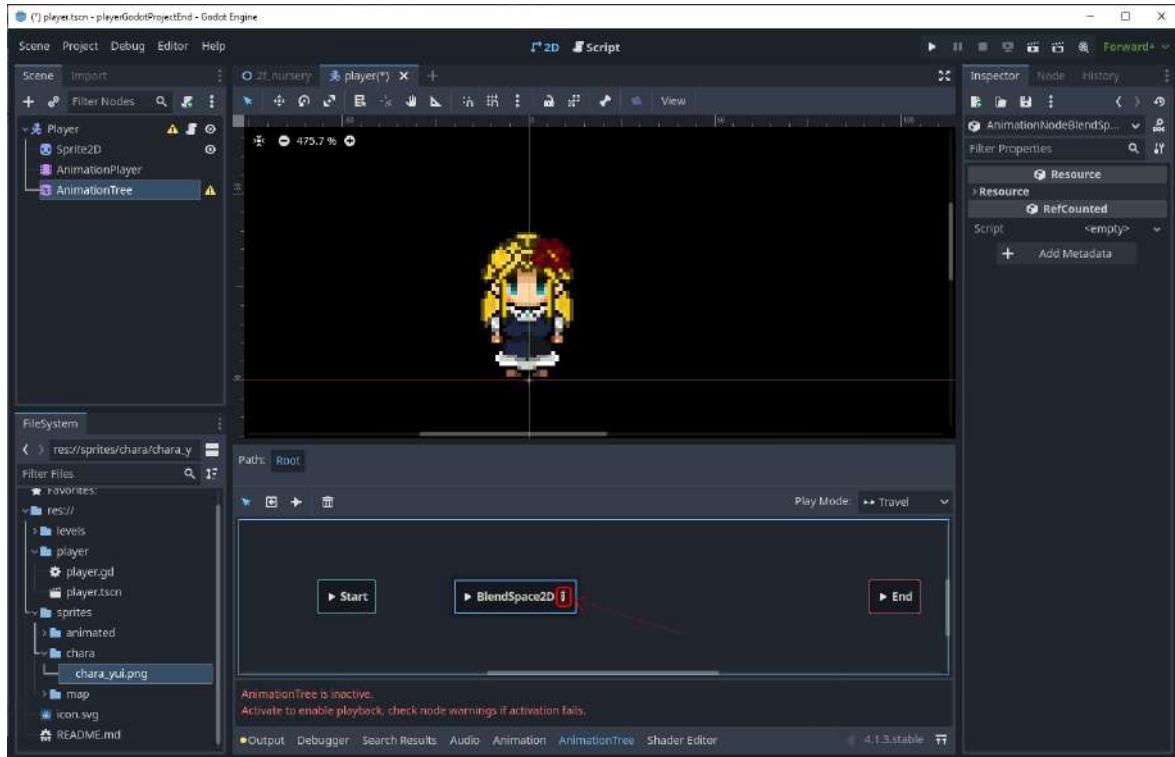
comparirà un nuovo "mattoncino"



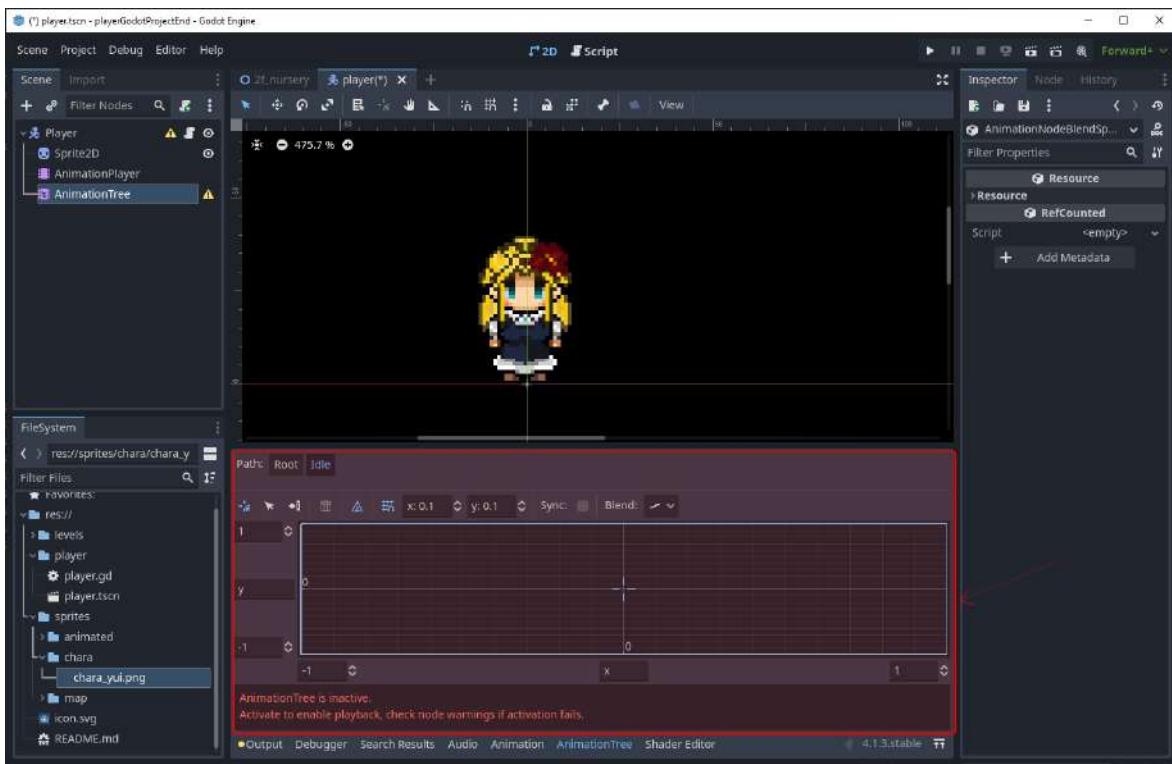
che rinominiamo **Idle**



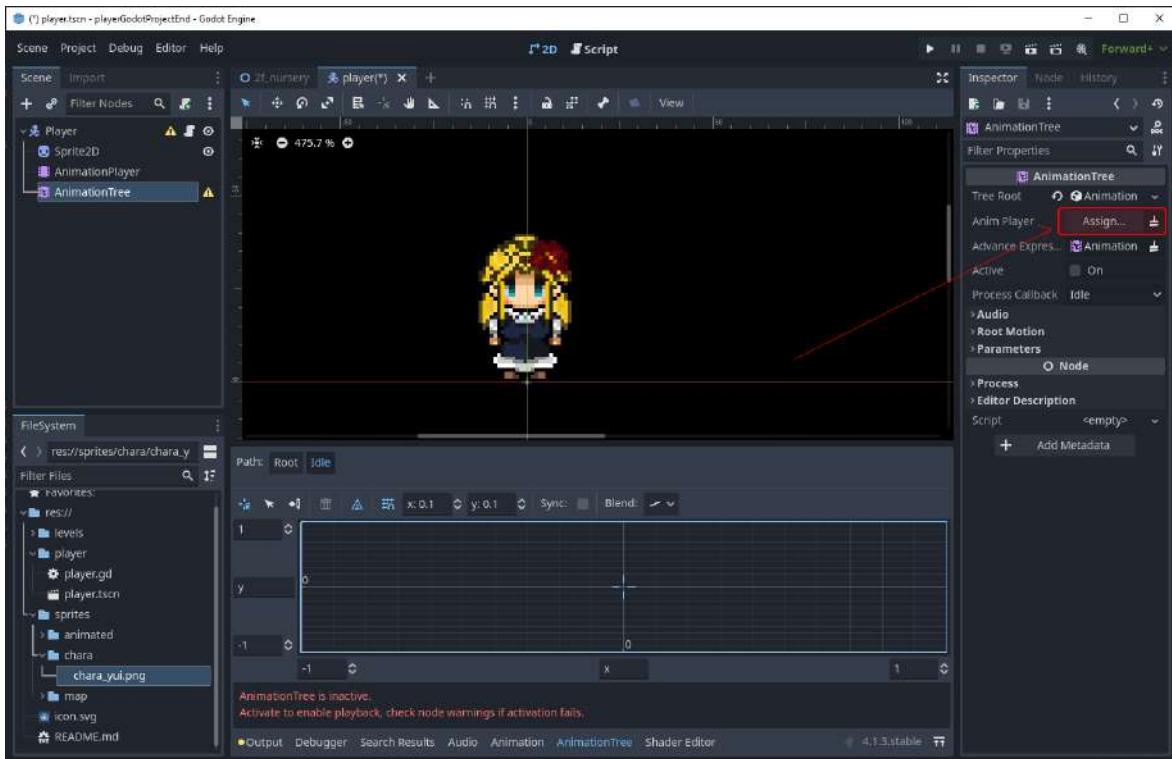
cliccando sull'icona della matita



si aprirà una nuova finestra di modifica

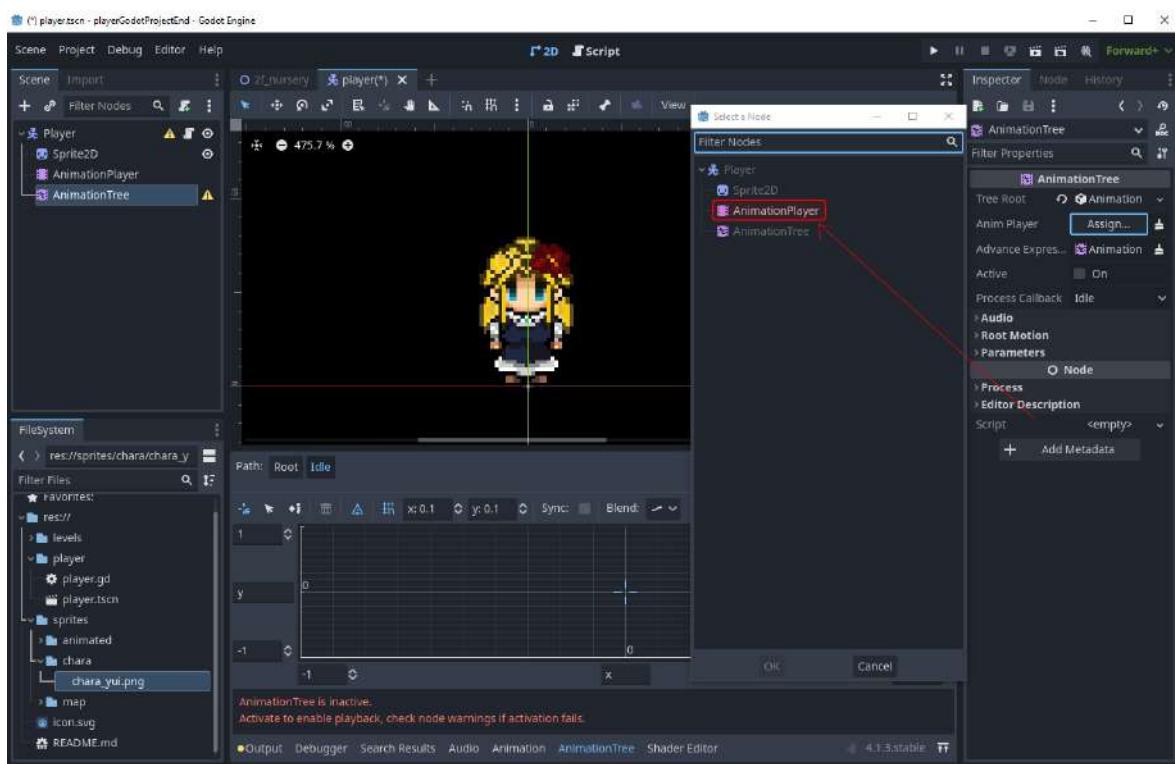


Per poter aggiungere qualcosa all'interno di questo spazio dobbiamo prima finire di configurare il nodo **AnimationTree**. Per farlo, dopo aver cliccato nuovamente il nodo **AnimationTree** per selezionarlo, clicchiamo su **Assign...** della voce **Anim Player**

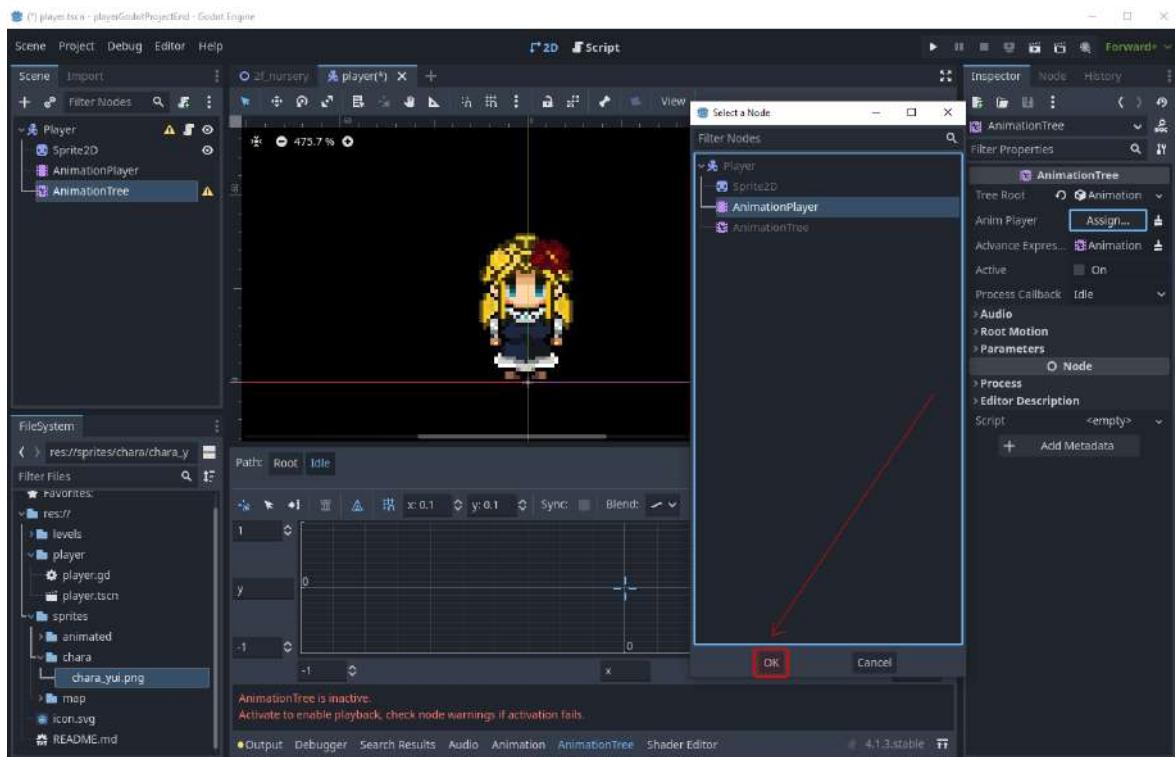


si aprirà una nuova finestra in cui selezionare il nodo **AnimationPlayer**

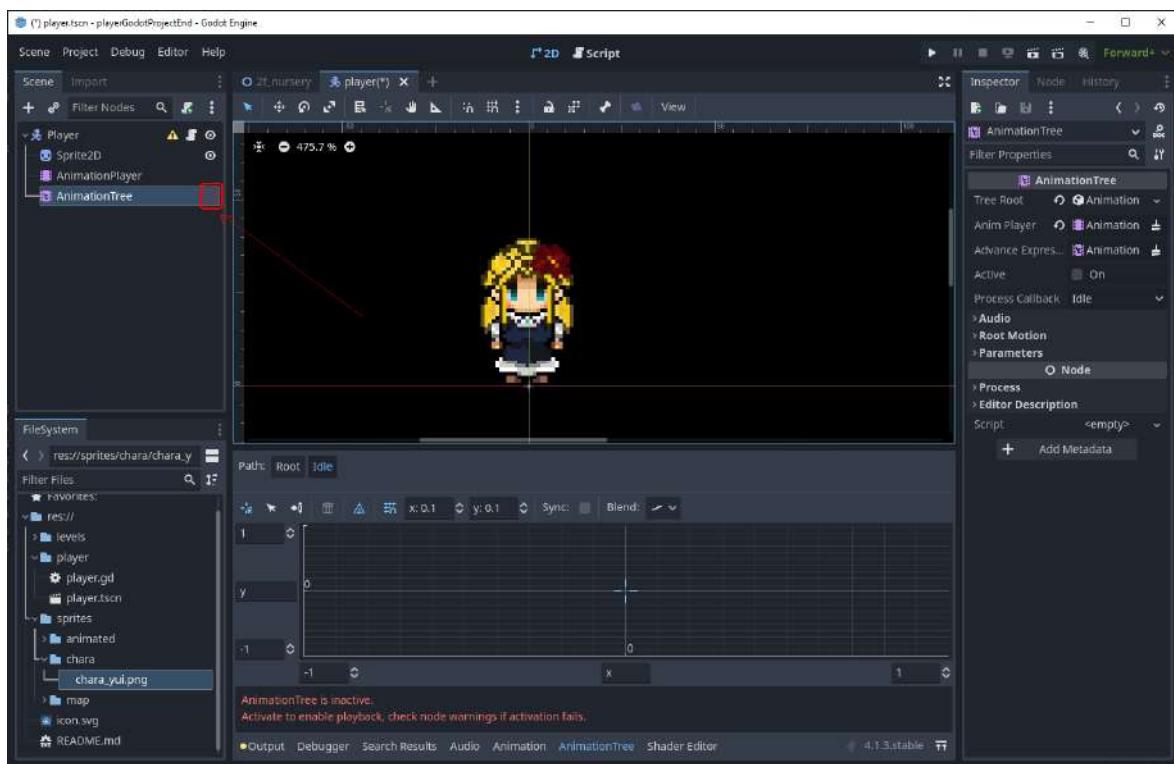
## 2.4. Player



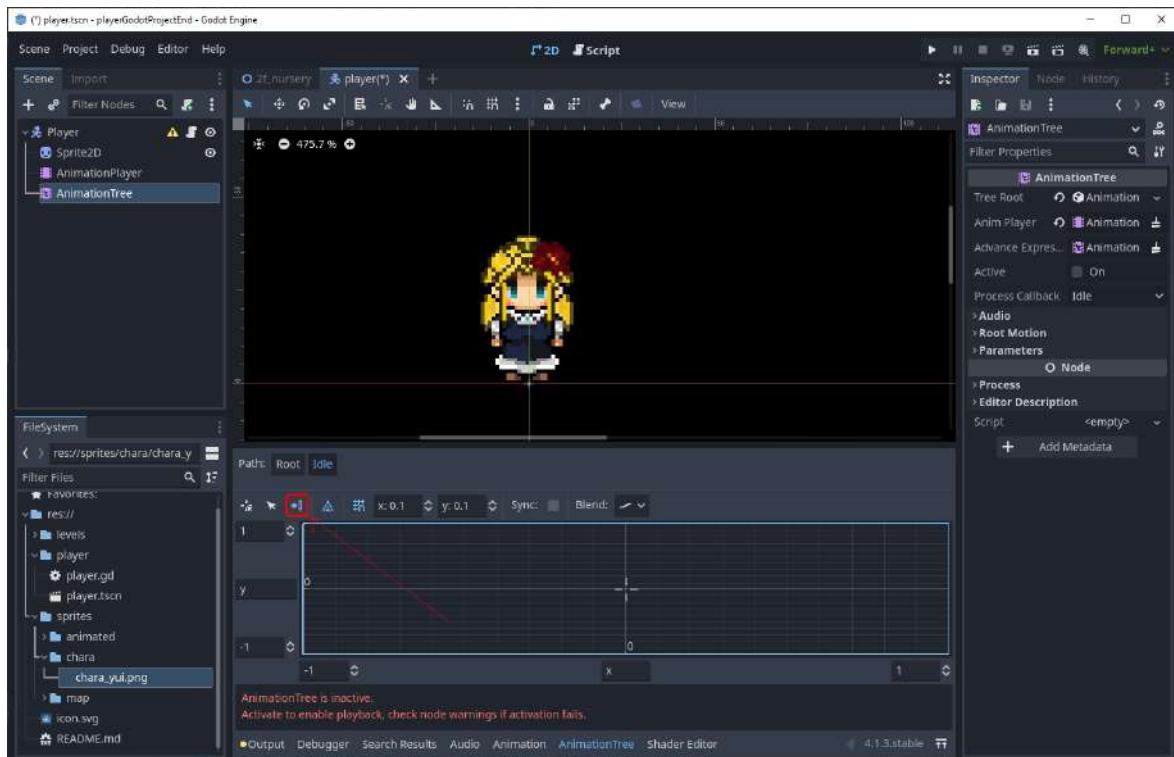
dopo aver cliccato su **Ok**



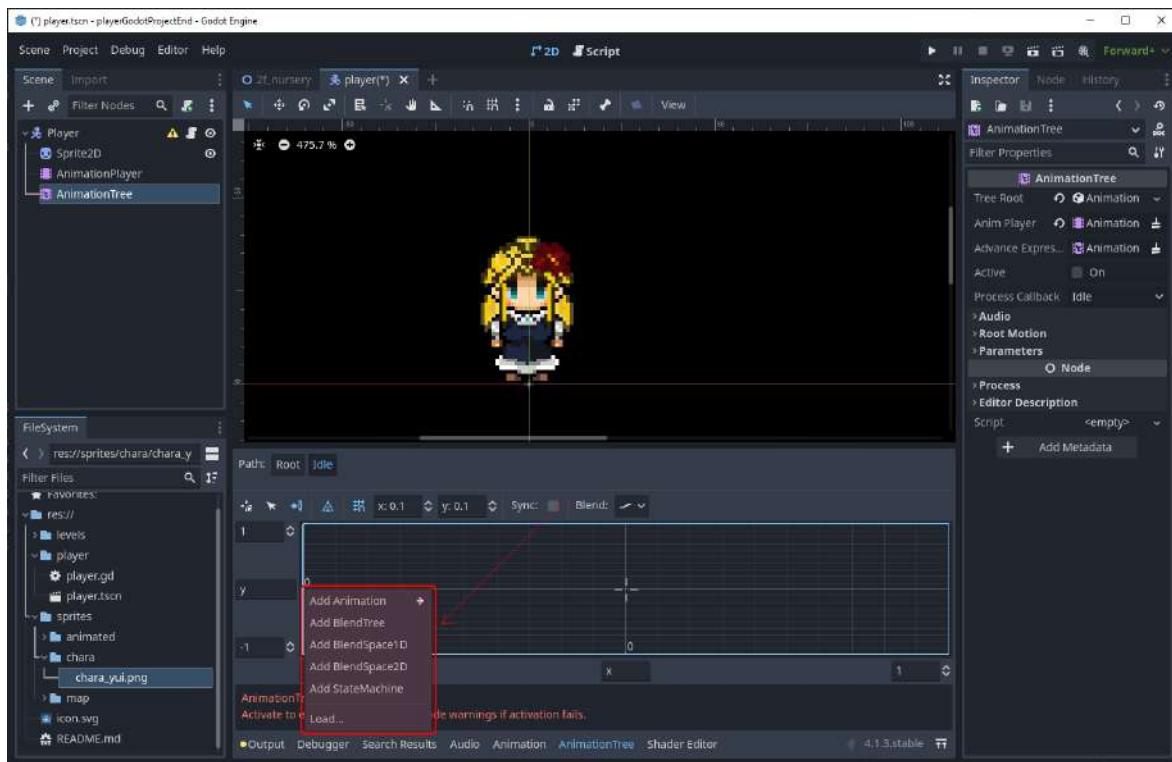
noteremo che il simbolo di **Warning** è finalmente scomparso



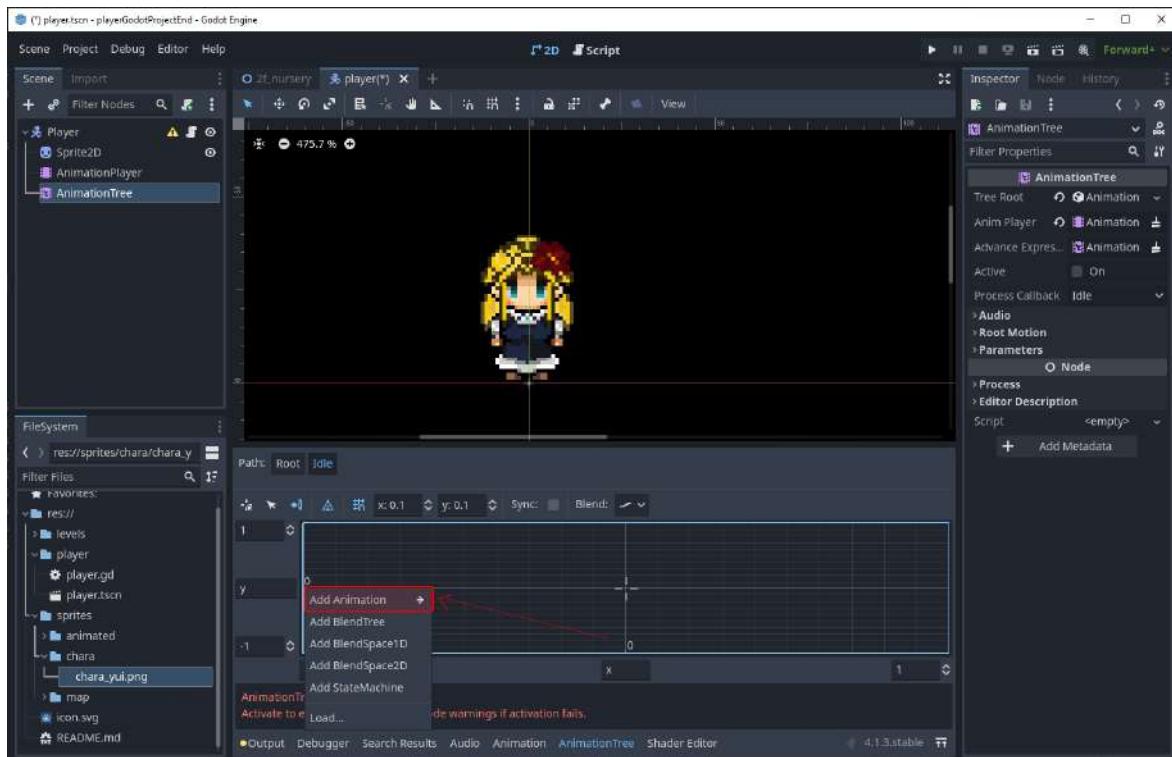
Tornando adesso al nostro spazio, clicchiamo sull'icona della matita con il rombo



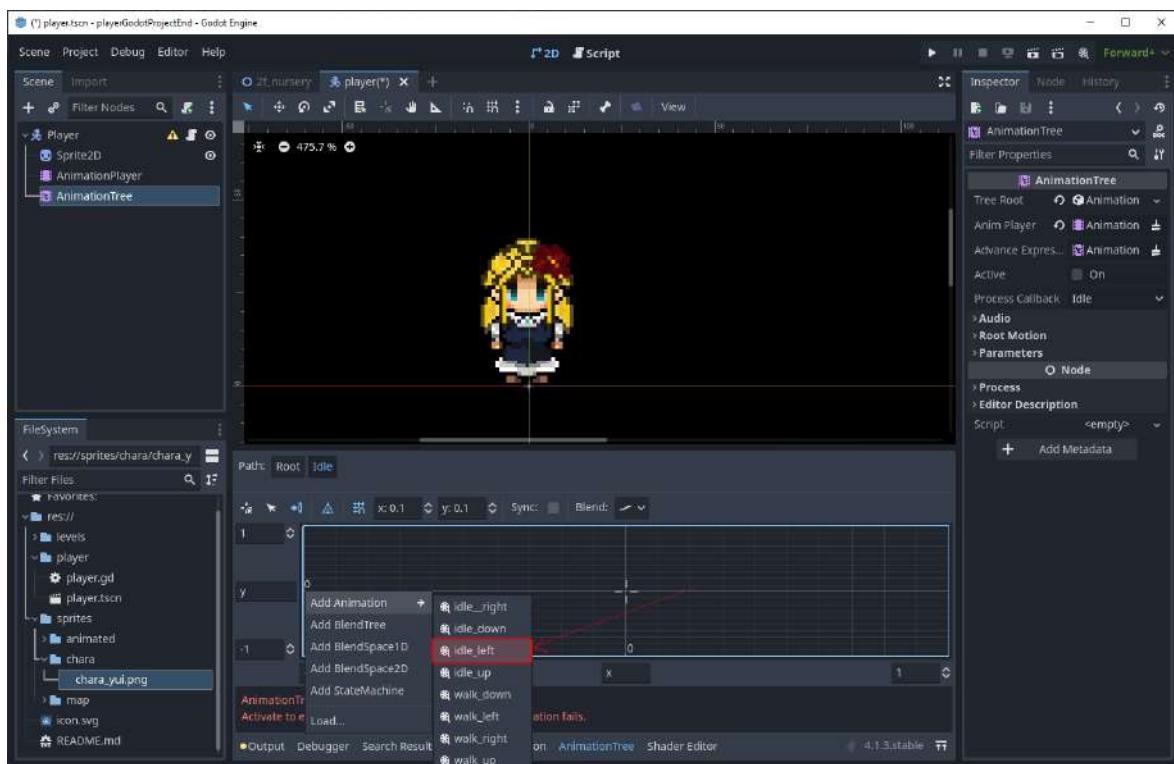
posizionando poi il mouse su una delle estremità degli assi, prendiamo ad esempio l'estremità sinistra, clicchiamo con il tasto sinistro per aprire una nuova finestra



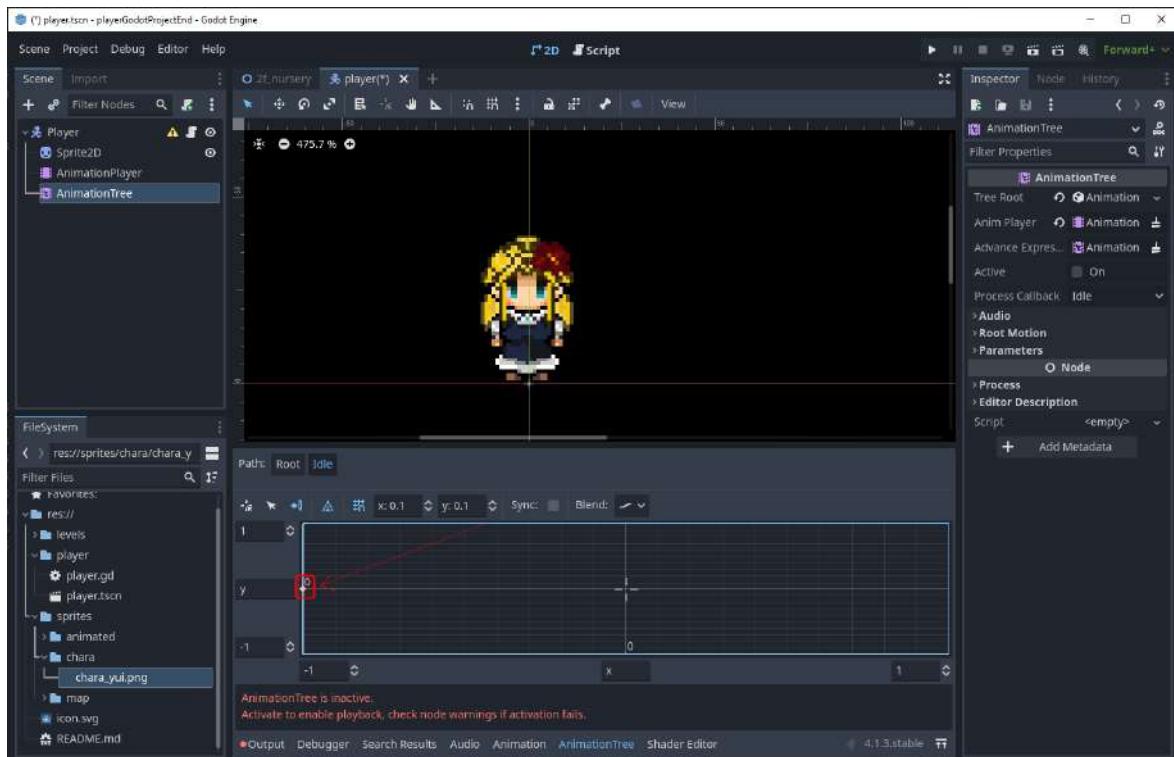
clicchiamo su **Add Animation**



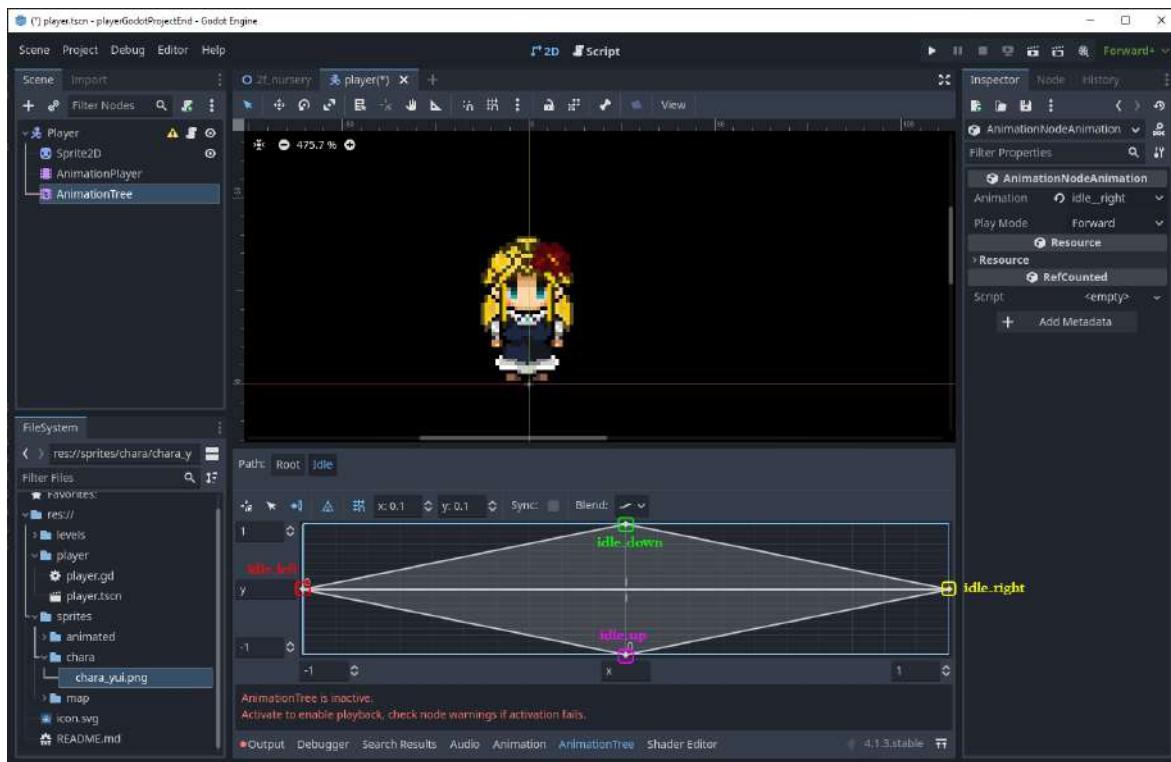
e selezioniamo l'animazione **idle\_left**



verrà aggiunto un nuovo frame

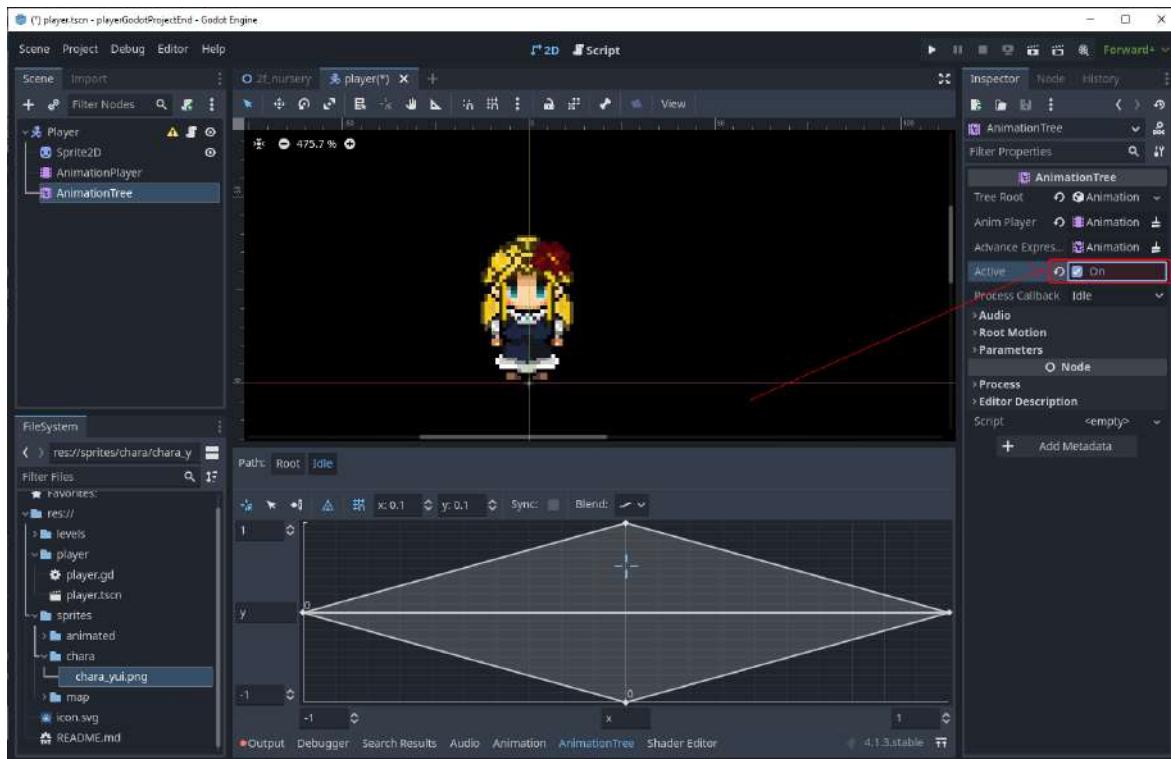


facciamo la stessa cosa con le altre tre estremità, ricordandoci però che, poiché l'asse y è invertito, **idle\_up** va sotto e **idle\_down** va sopra

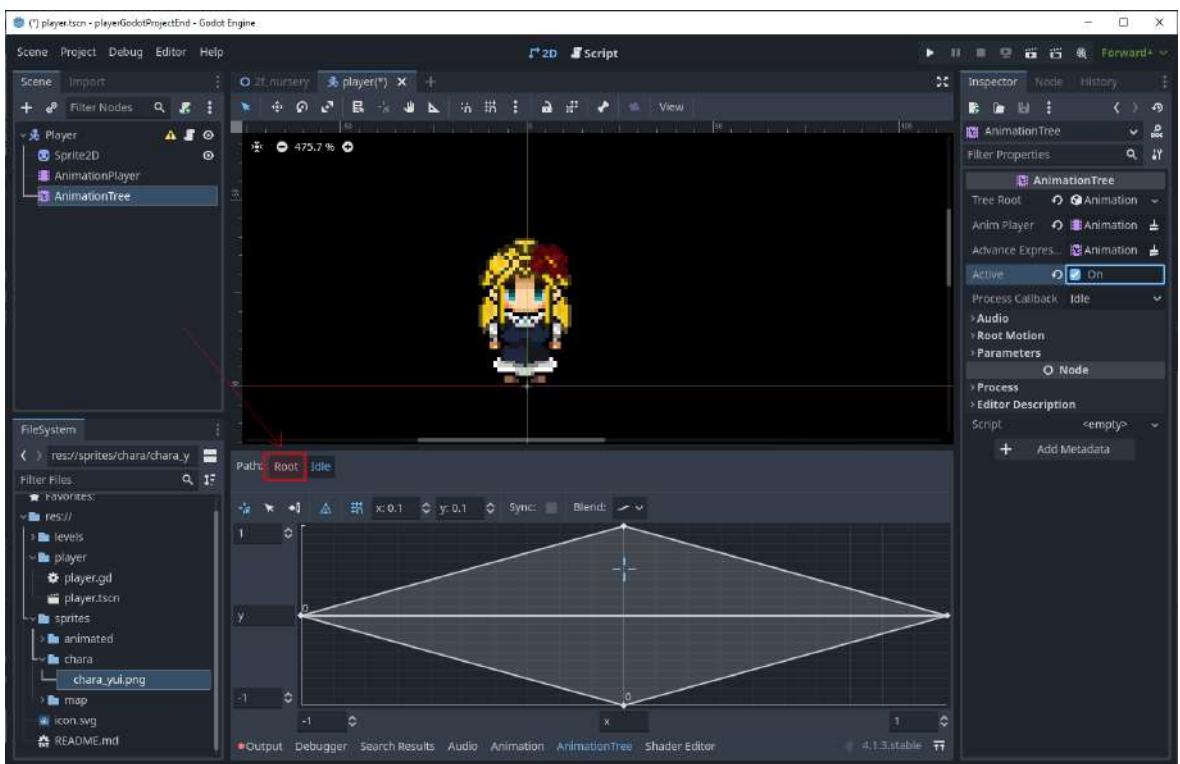


Come notiamo dall’immagine, già a partire dal terzo punto che inseriamo, si formerà un rombo che ricorda proprio il disegno del nostro `input_vector`.

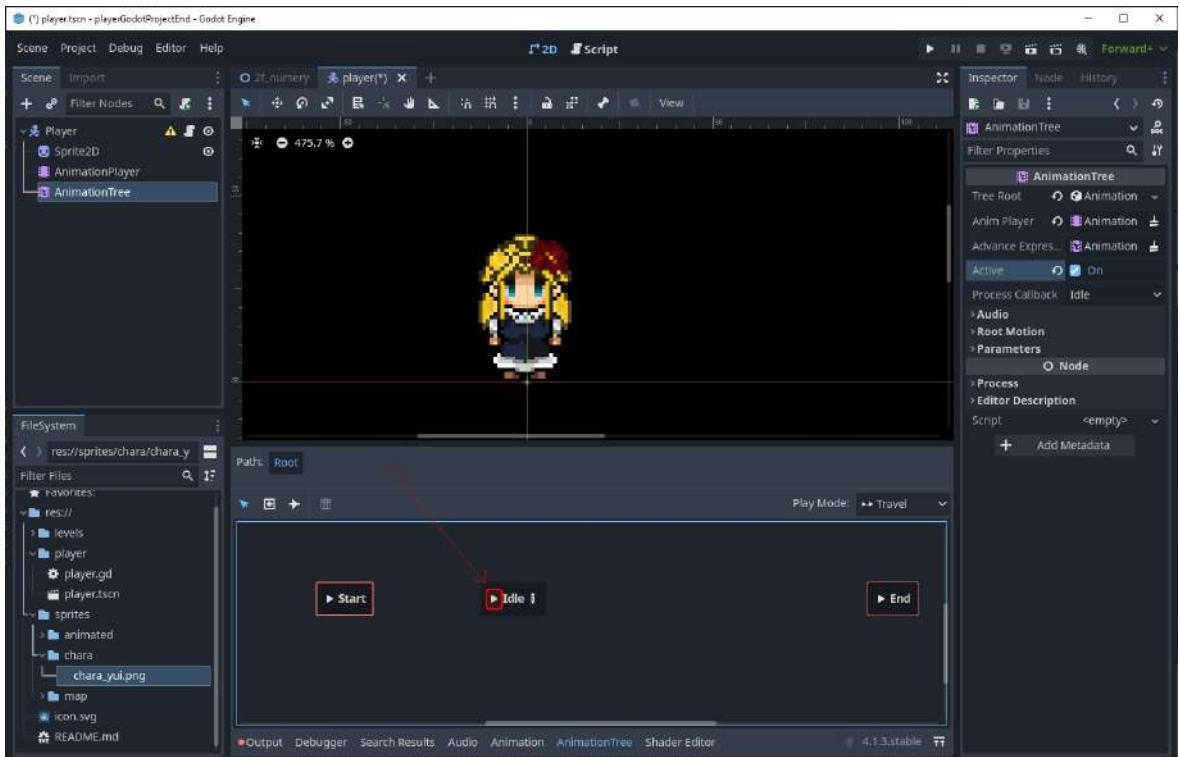
Se adesso abilitiamo l’opzione `Active` del nodo `AnimationTree`



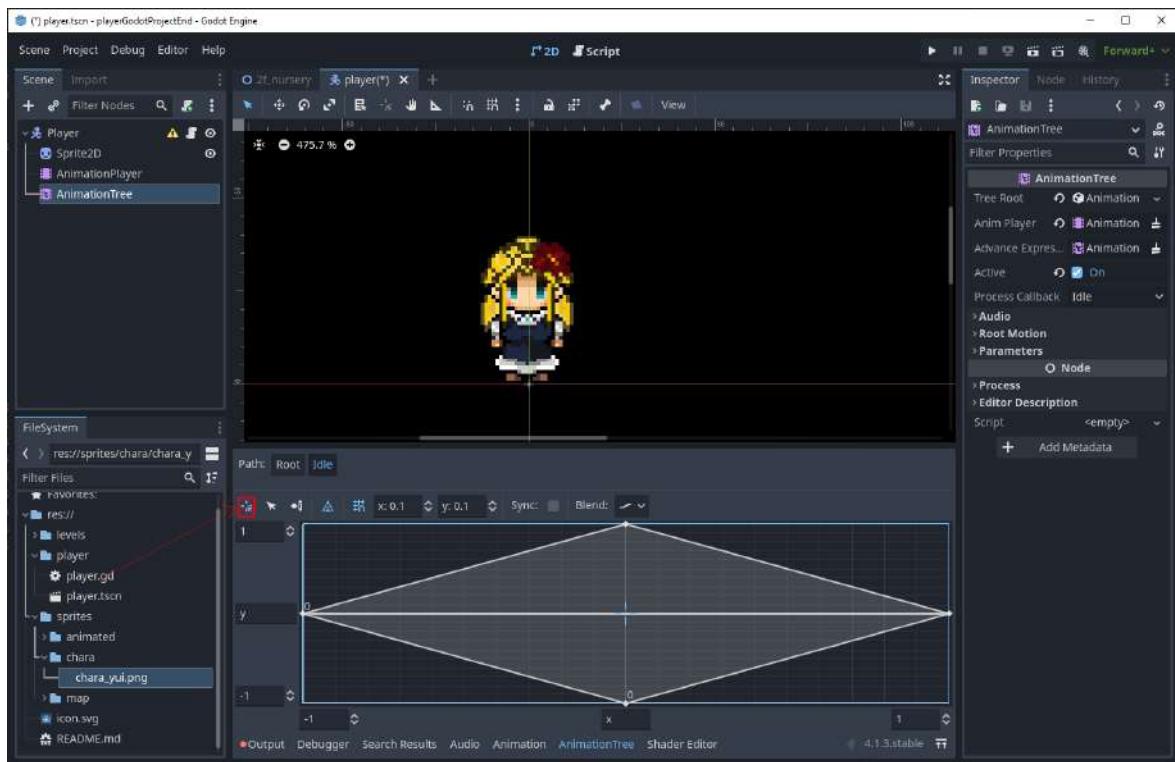
clicchiamo su `Root` per tornare indietro



clicchiamo sul simbolo di play



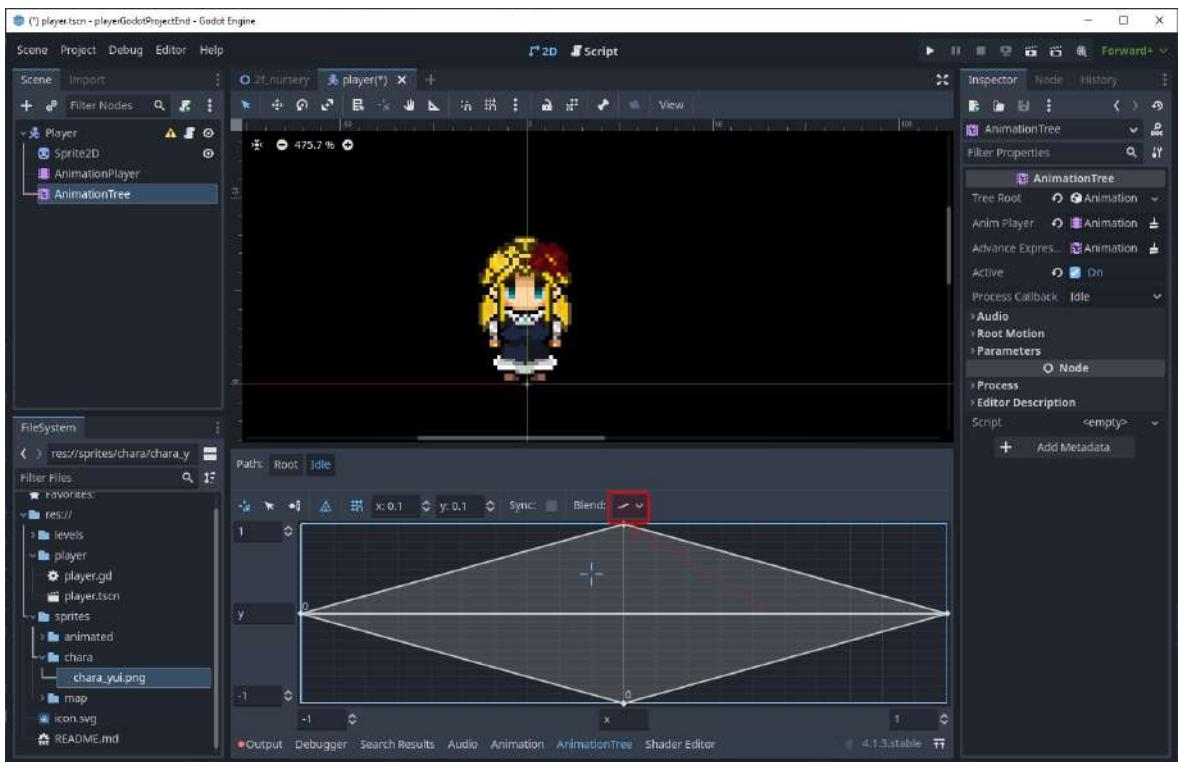
ritorniamo nella schermata con il piano di **idle** e clicchiamo sull'icona del crosshair



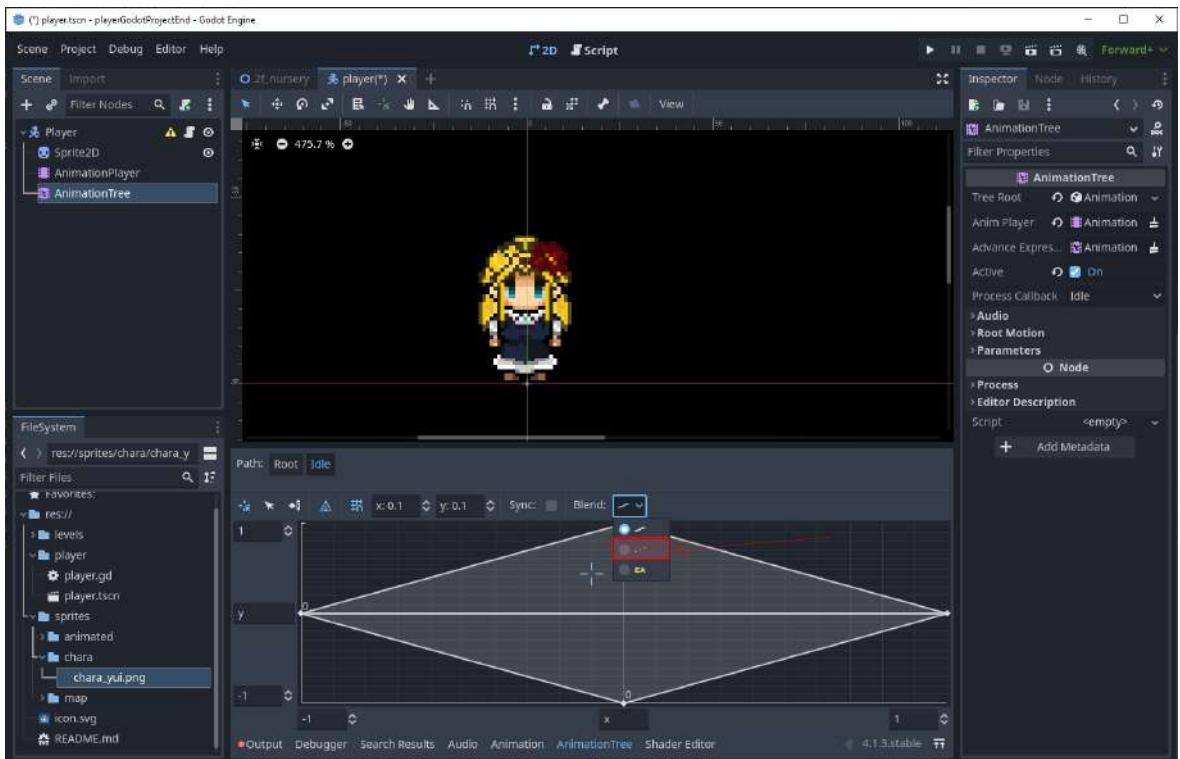
provando a trascinare il crosshair vero e proprio sul piano



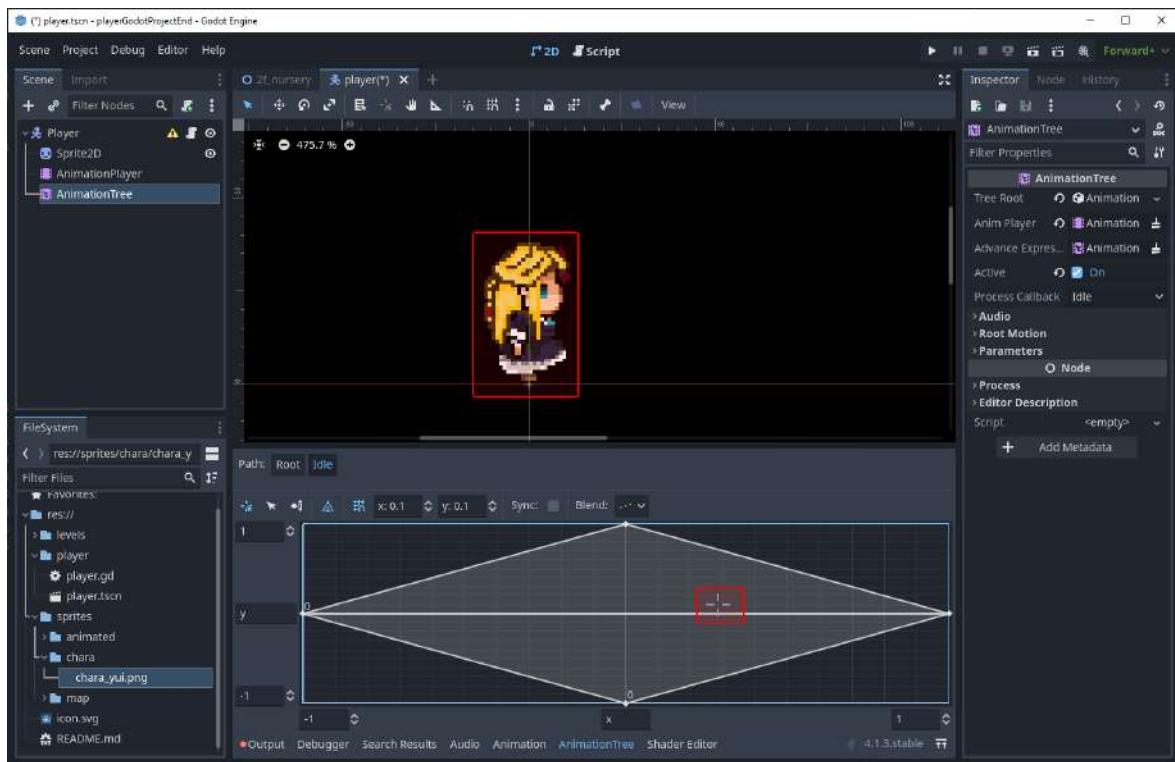
notiamo che in realtà non succede niente. Questo perché la blend mode è impostata su **Continuous**, la quale è utilizzata per animazioni fisiche, e non tramite sprite come la blend mode **Discrete**.



Impostando la blend mode su **discrete**

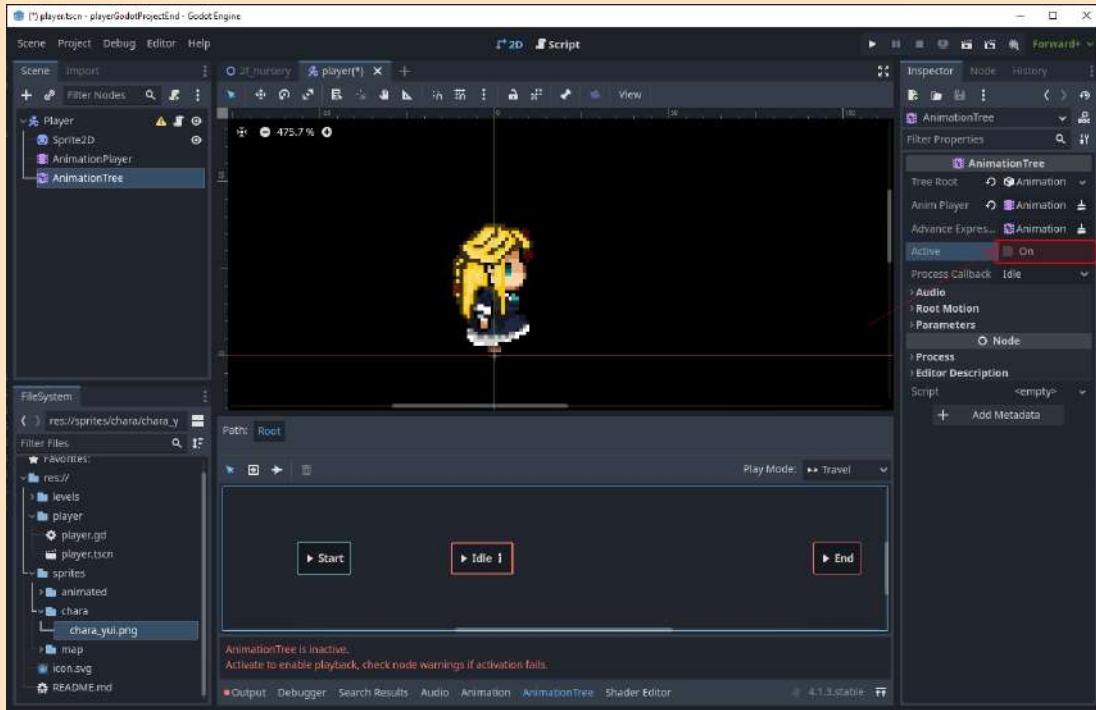


e muovendo il crooshair sullo schermo, notiamo infatti che il player cambia finalmente di sprite a seconda della posizione del crosshair stesso (il quale rappresenta la levetta analogica del controller)



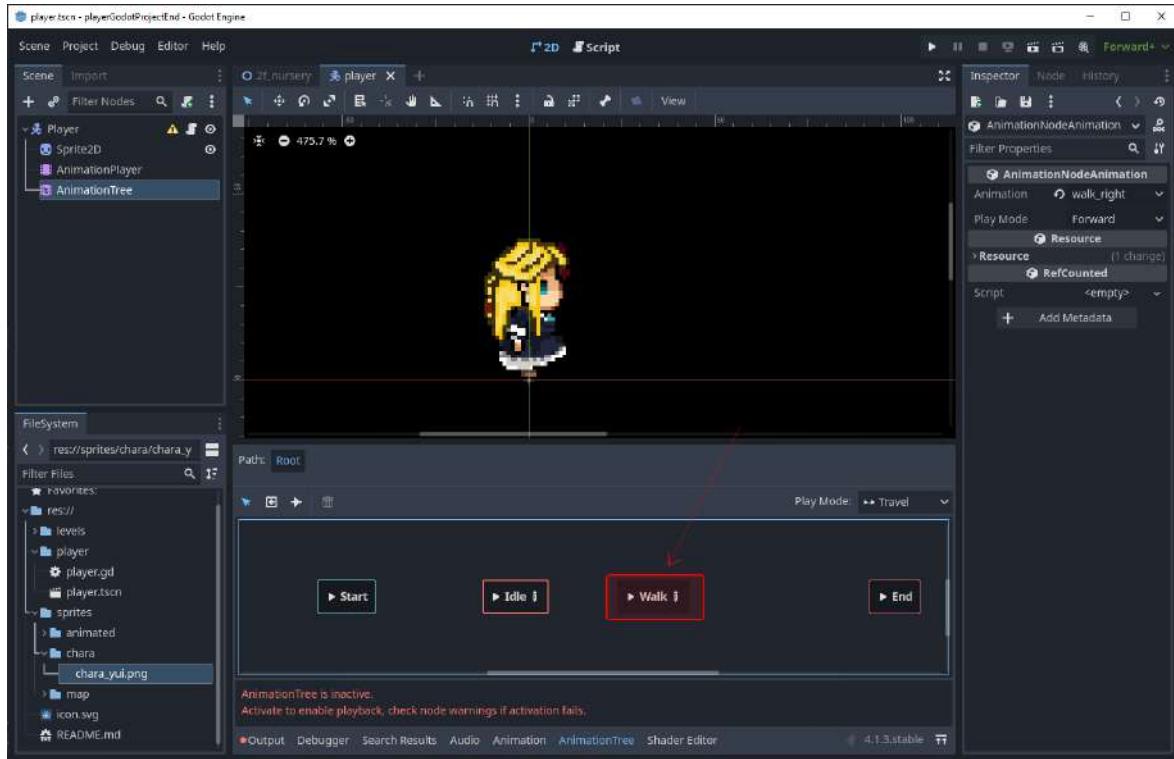
## ! Attenzione

Dopo aver terminato con il testing, ricordiamoci sempre di disabilitare l'opzione **Active** del nodo **AnimationTree**

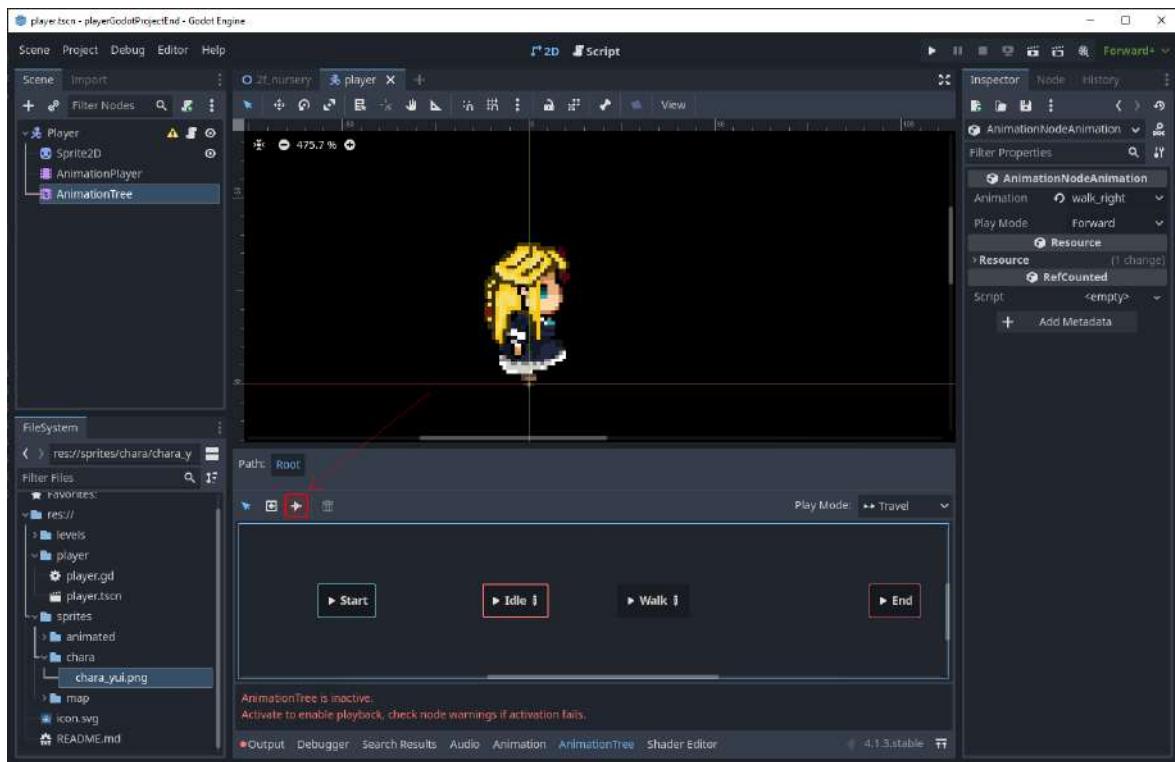


Questo perché, come vedremo tra poco, abiliteremo e disattiveremo l'`AnimationTree` tramite script.

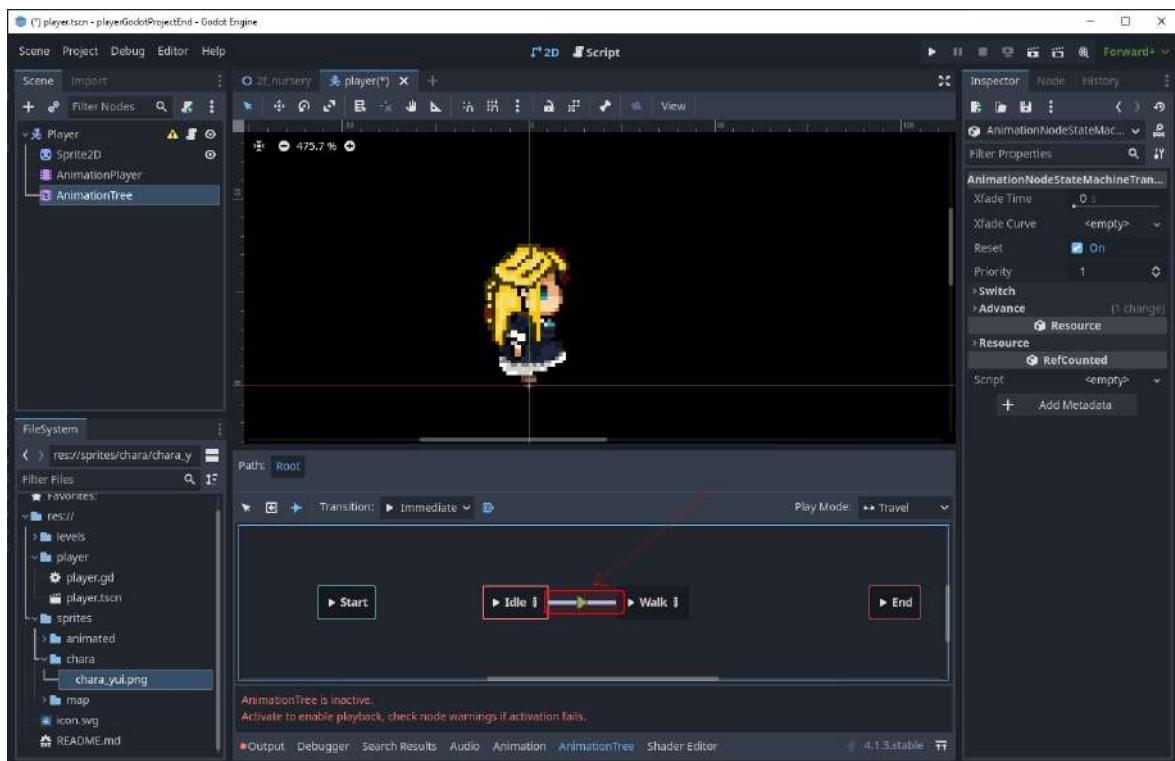
Facciamo ora la stessa cosa che abbiamo fatto per le animazioni di `idle`, con le animazioni di `walk`.



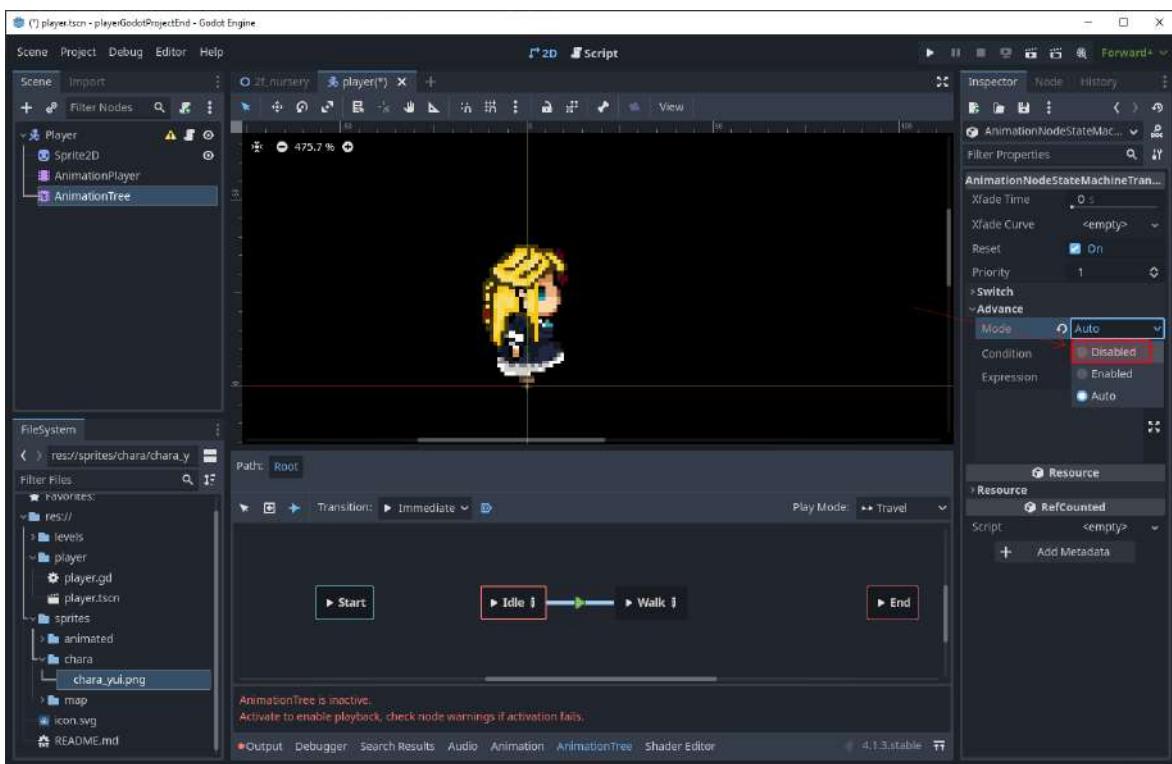
Prima di passare allo scripting, non ci resta che collegare gli stati `Idle` e `Walk` con le transizioni, proprio come nella figura della macchina a stati del player. Per farlo, clicchiamo sull'icona della freccia.



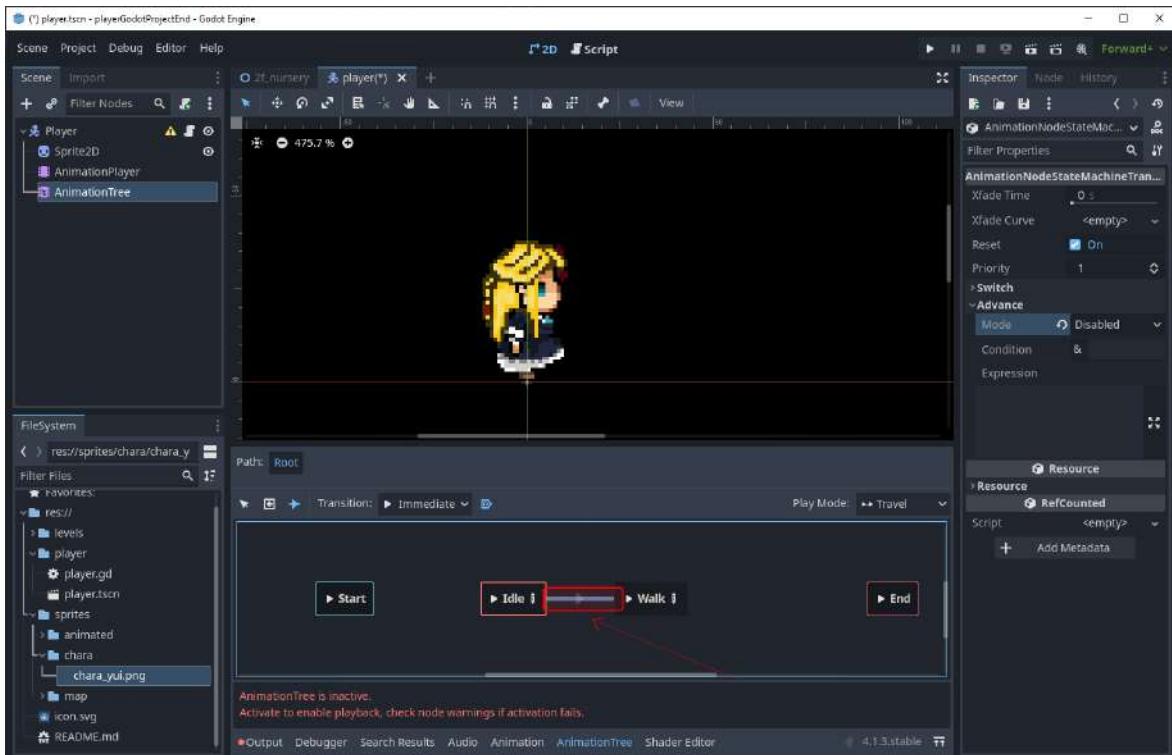
clicchiamo e teniamo premuto su **Idle** per poi rilasciare sopra **Walk**. Si creerà una freccia tra i due stati.



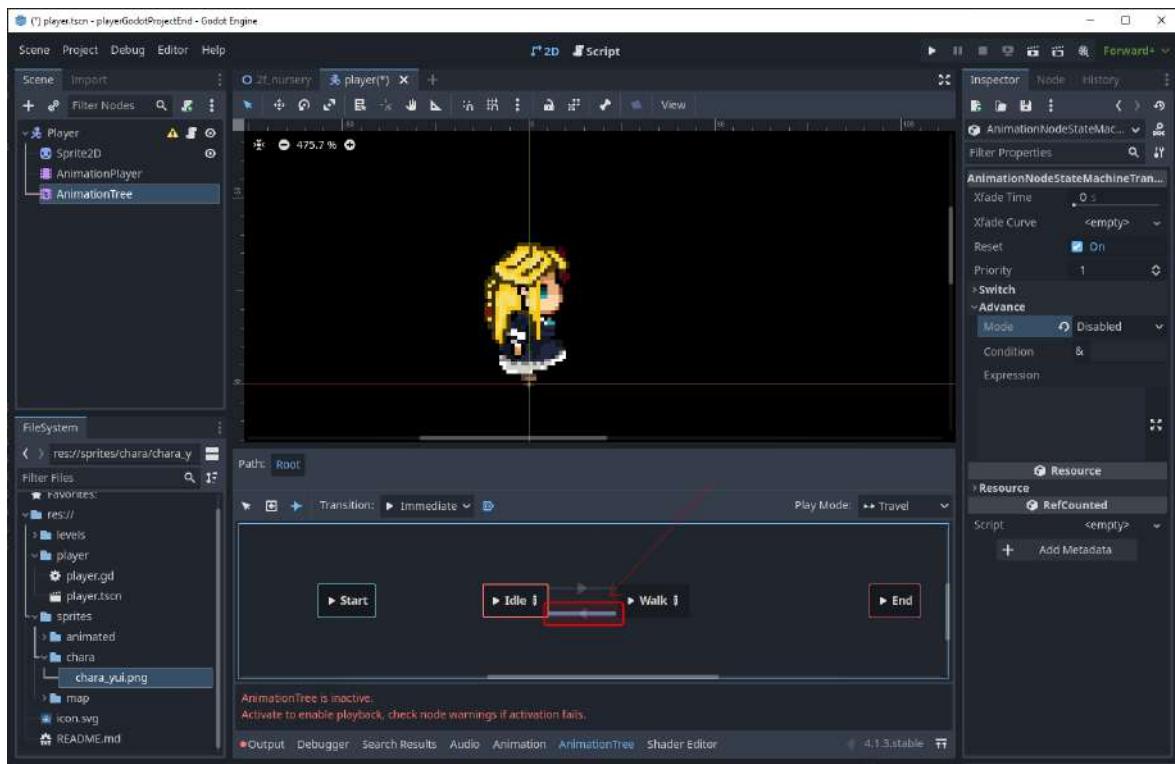
clicchiamo dunque sulla freccia per selezionarla, e impostiamo a **Disabled** l'opzione **Mode**, sotto la voce **Advance**.



la freccia assumerà un colore grigiastro.



Nello stesso modo, creiamo adesso un'altra freccia che va da **Walk** a **Idle**.



Abbiamo così terminato il setup della scena e possiamo finalmente passare allo scripting.

Il codice di scripting che ci permette di animare il nostro player è il seguente

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  @onready var animation_player = $AnimationPlayer # reference al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # reference al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # reference allo stato corrente
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'AnimationTree
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'AnimationTree
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'AnimationTree
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
17 const ANIMATION_TREE_PARAMETERS = [
18   "parameters/Idleblend_position",
19   "parameters/Walkblend_position"
20 ]
21
22 func _ready():
23   animation_tree.active = true # abilitiamo il nodo AnimationTree
24
25 func _physics_process(_delta):
26   var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
27
28   var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale della direzione del giocatore
29
30   # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
31   input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32   input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33   # normalizziamo il vettore per ottenere una direzione
34   input_vector = input_vector.normalized()
35   current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si sta muovendo
36
37 if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
38   face_direction(current_player_direction) # facciamo guardare al player la direzione che sta muovendo
39   animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41   velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo
42 else: # se non sta muovendo
43   animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44
45   velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio,
46
47   move_and_slide() # muoviamo il personaggio sullo schermo
48
49 func face_direction(direction: Vector2) -> void:
50   for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'AnimationTree
51     animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato dell'AnimationTree
52

```

Le righe 8, 9 e 10

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velo
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velo
7
8  @onready var animation_player = $AnimationPlayer # reference al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # reference al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # reference
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spaz
17 const ANIMATION_TREE_PARAMETERS = [
18     "parameters/Idleblend_position",
19     "parameters/Walkblend_position"
20 ]
21
22 func _ready():
23     animation_tree.active = true # abilitiamo il nodo AnimationTree
24

```

vedono l'utilizzo di una nuova keyword: `@onready`.

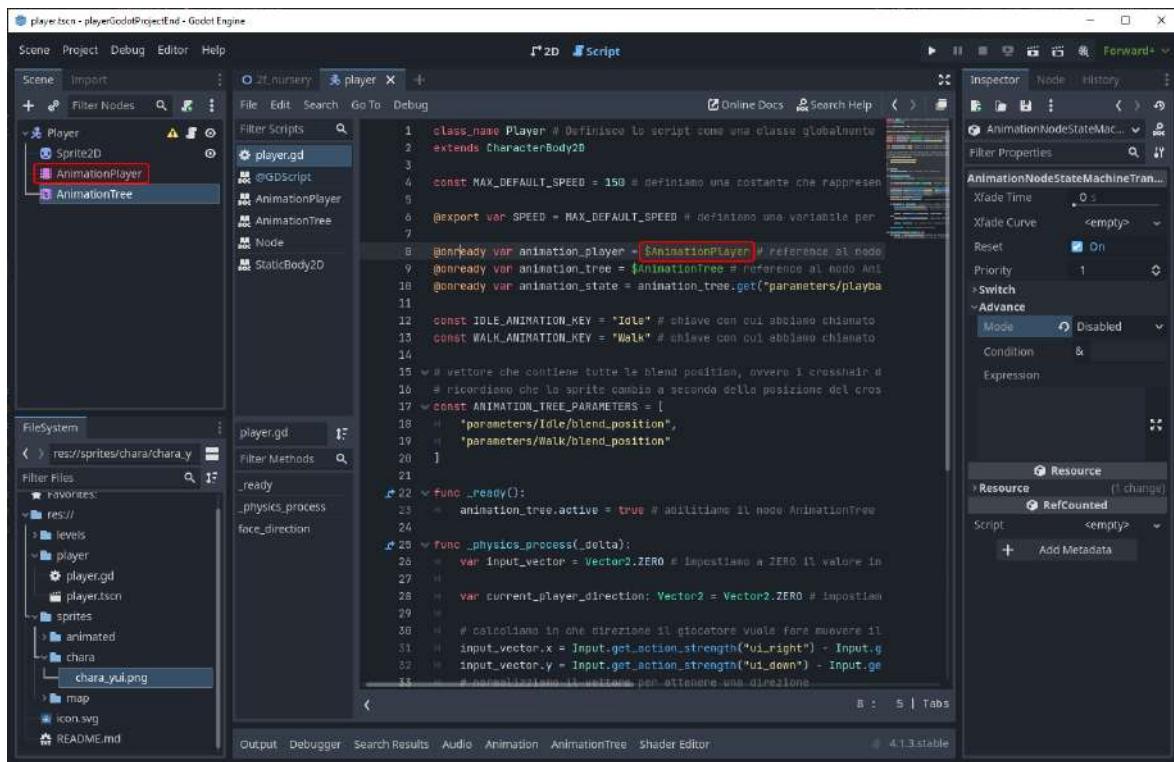
`@onready` è una keyword riservata di Godot che fa in modo di assegnare il valore alla variabile che la segue solamente quando il nodo è "ready" (pronto). I valori delle variabili precedute da `@onready` non vengono dunque assegnati immediatamente, quando il nodo viene inizializzato (ovvero quando viene invocato il metodo `_init()` di un oggetto), ma vengono bensì calcolati e memorizzati esattamente prima che venga chiamato il metodo `_ready()` di un nodo.

Il metodo `_init()` della classe `Object` è un metodo virtuale (dunque pensato per essere sovrascritto) che viene invocato quando lo script di un oggetto viene istanziato. La maggior parte delle volte ciò accade dopo che l'oggetto viene inizializzato in memoria attraverso `Object.new()`. Questo metodo è simile al metodo costrutture della stragrande maggioranza dei linguaggi di programmazione.

Il metodo `_ready()` è un metodo virtuale che viene chiamato quando il nodo è "ready" (pronto), cioè quando sia il nodo che i suoi nodi figlio sono entrati nello scene tree. Se il nodo in questione ha figli, le callback `_ready()` dei nodi figlio vengono eseguite prima, e il nodo padre riceverà la notifica di ready solamente una volta che tutte le callback `_ready()` dei nodi figlio hanno terminato l'esecuzione. Il metodo `_ready()` corrisponde alla notifica `NOTIFICATION_READY` in `Object.notification()`

Quello che si fa in queste righe, dunque, è semplicemente memorizzare i riferimenti ai nodi che compongono la scena `Player`, in modo da poterli accedere più tardi.

Nello specifico, a riga 8 si sta memorizzando nella variabile `animation_player` il riferimento al nodo `AnimationPlayer`



Il valore a destra dell'assegnamento, ovvero `$AnimationPlayer` indica il path (la strada che si deve percorrere, partendo dal nodo radice, per raggiungere il nodo in questione) del nodo all'interno della scena. Con `$` si intende infatti il nodo radice della scena, in questo caso `Player`. Dunque, poiché il nodo `AnimationPlayer` è figlio diretto (diretto nel senso che non ha altri padri nella scena oltre che il nodo `Player`) del nodo `Player`, allora il suo path è semplicemente `$AnimationPlayer`.

Per essere più precisi, `$AnimationPlayer` è una shorthand per il metodo

23    `self.get_node("AnimationPlayer")`

lì dove con `self` si intende il nodo a cui è attaccato lo script in cui è scritta la keyword, ovvero `Player` in questo caso.

Riga 9 è la stessa di riga 8, con l'unica differenza che si sta riferendo `AnimationTree` e non `AnimationPlayer`.

A riga 10, invece, stiamo utilizzando il metodo `get()` di `Object`, ereditato dalla classe `AnimationTree`, per ottenere un riferimento allo stato corrente di `AnimationTree`.

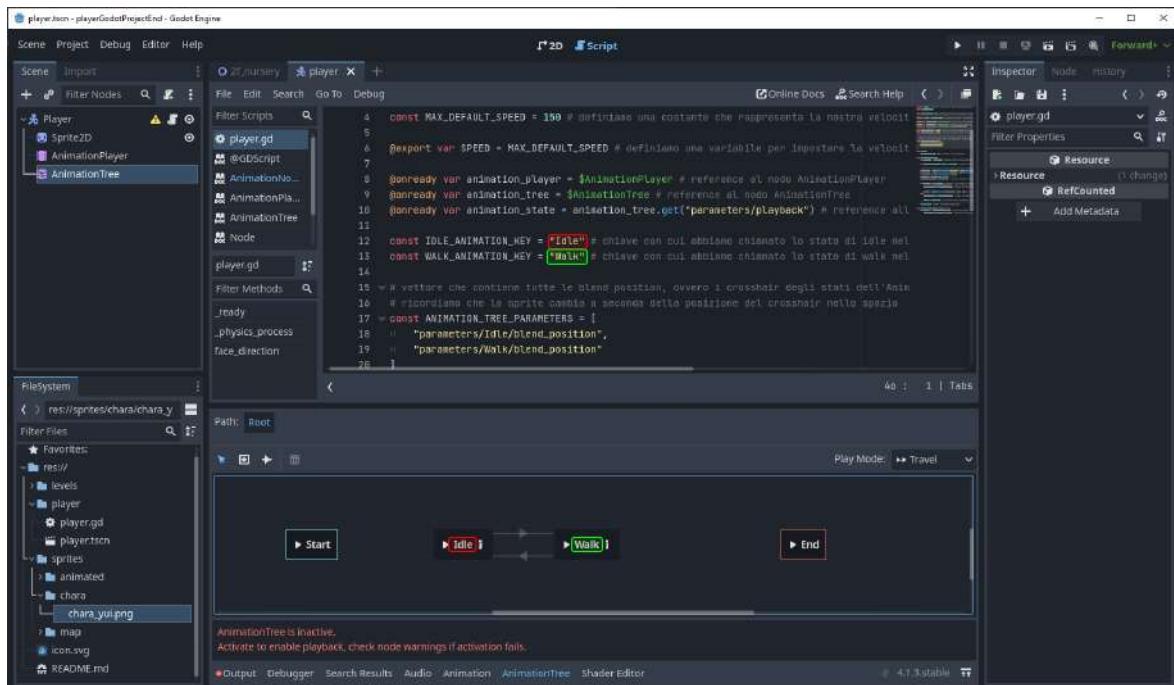
A riga 12 e 13

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra veloci
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la veloci
7
8  @onready var animation_player = $AnimationPlayer # reference al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # reference al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # reference all
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'animazion
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
17 const ANIMATION_TREE_PARAMETERS = [
18     "parameters/Idleblend_position",
19     "parameters/Walkblend_position"
20 ]
21
22 func _ready():
23     animation_tree.active = true # abilitiamo il nodo AnimationTree
24

```

memorizziamo in delle costanti il nome degli stati che abbiamo creato nel nodo **AnimationTree**



Utilizzando delle costanti per questi valori: evitiamo di doverli scrivere a mano ogni

volta; preveniamo possibili errori di battura; se modifichiamo il nome di uno stato, non dobbiamo cambiarlo in tutti i posti dove lo abbiamo utilizzato, ma basta bensì modificare il valore unico della costante. Di solito, quando ci sono valori come questi, conviene sempre utilizzare delle costanti per memorizzarli, evitando così di buttare qualche ora più avanti nello sviluppo a causa di errori vari.

Nelle righe che vanno dalla 17 alla 20

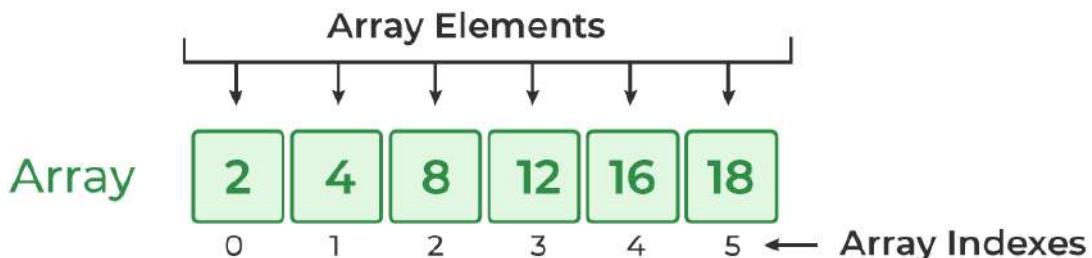
```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velo
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velo
7
8  @onready var animation_player = $AnimationPlayer # reference al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # reference al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # reference
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spaz
17 const ANIMATION_TREE_PARAMETERS = [
18     "parameters/Idleblend_position",
19     "parameters/Walkblend_position"
20 ]
21
22 func _ready():
23     animation_tree.active = true # abilitiamo il nodo AnimationTree
24

```

stiamo utilizzando un nuovo tipo di struttura dati: l'**Array**.

Un **Array** è una struttura dati lineare dove tutti gli elementi sono disposti in modo sequenziale. È una collezione di elementi dello stesso tipo memorizzati in posizioni di memoria contigue.



Per semplicità, possiamo pensare a un array come a una rampa di scale in cui su ogni gradino è posto un valore. In questo modo, per conoscere la posizione di un qualsiasi valore, basta semplicemente contare i gradini in cui si trova. Questo sistema rende molto più facile calcolare la posizione di ciascun elemento aggiungendo un valore di **offset** a un valore di base, cioè, la locazione di memoria del primo elemento dell'array (generalmente denotano con il nome dell'array). Il valore di base è l'indice (index) 0, mentre la differenza tra i due indici è l'**offset**.

Quello che facciamo in queste righe di codice è dunque creare un array di stringhe in cui andiamo a memorizzare le **blend\_position** (ovvero i **crosshair**) dei nostri due stati **Walk** e **Idle**

Nelle righe di codice 22 e 23

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra veloci
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la veloci
7
8  @onready var animation_player = $AnimationPlayer # reference al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # reference al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # reference
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'animazion
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
17 const ANIMATION_TREE_PARAMETERS = [
18   "parameters/Idleblend_position",
19   "parameters/Walkblend_position"
20 ]
21
22 func _ready():
23   animation_tree.active = true # abilitiamo il nodo AnimationTree
24

```

effettuiamo l'override del metodo **\_ready()**, il quale, come abbiamo visto in precedenza, viene chiamato al momento dell'inizializzazione dell'oggetto, ovvero di **Player**. Nel corpo della funzione, invece, ci occupiamo solamente di attivare il nodo **AnimationTree**.

A riga 28

---

<sup>39</sup>Fonte: GeeksforGeeks

```

22  func _ready():
23      animation_tree.active = true # abilitiamo il nodo AnimationTree
24
25  func _physics_process(_delta):
26      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di m
27
28      var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale de
29
30      # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
31      input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32      input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33      # normalizziamo il vettore per ottenere una direzione
34      input_vector = input_vector.normalized()
35      current_player_direction = input_vector # il player sta guardando la stessa direzione in cui s
36
37      if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo i
38          face_direction(current_player_direction) # facciamo guardare al player la direzione che de
39          animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41      velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effetti
42  else:
43      animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44

```

dichiariamo e inizializziamo a `(0,0)` la variabile `current_player_direction`, la quale non è strettamente necessaria per il funzionamento del nostro script, ma è comunque utile per farci capire in quale direzione il nostro giocatore deve guardare mentre si sta muovendo.

A riga 35, infatti, copiamo il valore di `input_vector` dentro `current_player_direction`. Quella che abbiamo appena memorizzato è la direzione effettiva che il nostro player deve guardare.

```

22  func _ready():
23      animation_tree.active = true # abilitiamo il nodo AnimationTree
24
25  func _physics_process(_delta):
26      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di m
27
28      var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale de
29
30      # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
31      input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32      input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33      # normalizziamo il vettore per ottenere una direzione
34      input_vector = input_vector.normalized()
35      current_player_direction = input_vector # il player sta guardando la stessa direzione in cui s
36
37      if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo i
38          face_direction(current_player_direction) # facciamo guardare al player la direzione che de
39          animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41      velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effetti
42  else:
43      animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44

```

A riga 38

```

22  func _ready():
23      animation_tree.active = true # abilitiamo il nodo AnimationTree
24
25  func _physics_process(_delta):
26      var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di m
27
28      var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale de
29
30      # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
31      input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32      input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33      # normalizziamo il vettore per ottenere una direzione
34      input_vector = input_vector.normalized()
35      current_player_direction = input_vector # il player sta guardando la stessa direzione in cui s
36
37      if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo i
38          face_direction(current_player_direction) # facciamo guardare al player la direzione che de
39          animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41      velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effetti
42  else:
43      animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44

```

invociamo la funzione `face_direction(direction)`, la quale è definita nelle righe che vanno dalla 49 alla 51. Invocare una funzione significa che, quando Godot esegue il codice dello script e arriva a riga 38, salta immediatamente a riga 49, le esegue tutte fino alla 51 e poi torna nuovamente alla 39. Per carpirci meglio, è come se il codice fosse scritto in questo modo

```

37  if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il pe
38      face_direction(current_player_direction) # facciamo guardare al player la direzione che deve g
39
40
41      for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'
42          animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
43
44      animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
45
46
47      velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente
48
49
50
51
52

```

li dove le righe 38 e 49 non vengono propriamente eseguite da Godot poiché sono, rispettivamente, l'invocazione della funzione `face_direction(direction)` (riga 38), alla quale viene passato il valore di `current_player_direction`, il che significa che la variabile `direction` definita nella firma del metodo `face_direction(direction)` avrà lo stesso valore contenuto dentro la variabile `current_player_direction`; e la

firma del metodo `face_direction(direction)` (riga 49).

Riga 50 e 51 presentano invece un nuovo costrutto: il **for loop**.

```
31     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33     # normalizziamo il vettore per ottenere una direzione
34     input_vector = input_vector.normalized()
35     current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si sta
36
37     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
38         face_direction(current_player_direction) # facciamo guardare al player la direzione che deve dare
39         animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo il personaggio
42     else:
43         animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44
45         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamone uno zero
46
47     move_and_slide() # muoviamo il personaggio sullo schermo
48
49     func face_direction(direction: Vector2) -> void:
50         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'AnimationTree
51             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato dell'AnimationTree
52 |
```

il *for loop* è un costrutto utilizzato quando si sa esattamente quante volte si vuole eseguire in loop il codice presente nel suo corpo. A riga 50 del nostro codice, ad esempio, stiamo dicendo di voler eseguire il codice del costrutto for (formato solamente dalla riga 51) per tante volte quante sono gli elementi dell'array `ANIMATION_TREE_PARAMETERS`, ovvero due. La variabile `parameter`, inoltre, in queste due esecuzioni assumerà prima il valore `parameters/Idle/blend_position` e poi `parameters/Walk/blend_position`.

A riga 51

```

31     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33     # normalizziamo il vettore per ottenere una direzione
34     input_vector = input_vector.normalized()
35     current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si sta
36
37     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
38         face_direction(current_player_direction) # facciamo guardare al player la direzione che deve guardare
39         animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo il personaggio
42     else:
43         animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44
45         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamogli una velocità nulla
46
47         move_and_slide() # muoviamo il personaggio sullo schermo
48
49     func face_direction(direction: Vector2) -> void:
50
51         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'AnimationTree
52             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato dell'AnimationTree nel nuovo direzione

```

impostiamo il valore del crosshair degli stati `Idle` e `Walk` nella posizione indicata da `direction`

A riga 39

```

31     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33     # normalizziamo il vettore per ottenere una direzione
34     input_vector = input_vector.normalized()
35     current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si sta
36
37     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
38         face_direction(current_player_direction) # facciamo guardare al player la direzione che deve guardare
39         animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo il personaggio
42     else:
43         animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44
45         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamogli una velocità nulla
46
47         move_and_slide() # muoviamo il personaggio sullo schermo
48
49     func face_direction(direction: Vector2) -> void:
50
51         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'AnimationTree
52             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato dell'AnimationTree nel nuovo direzione

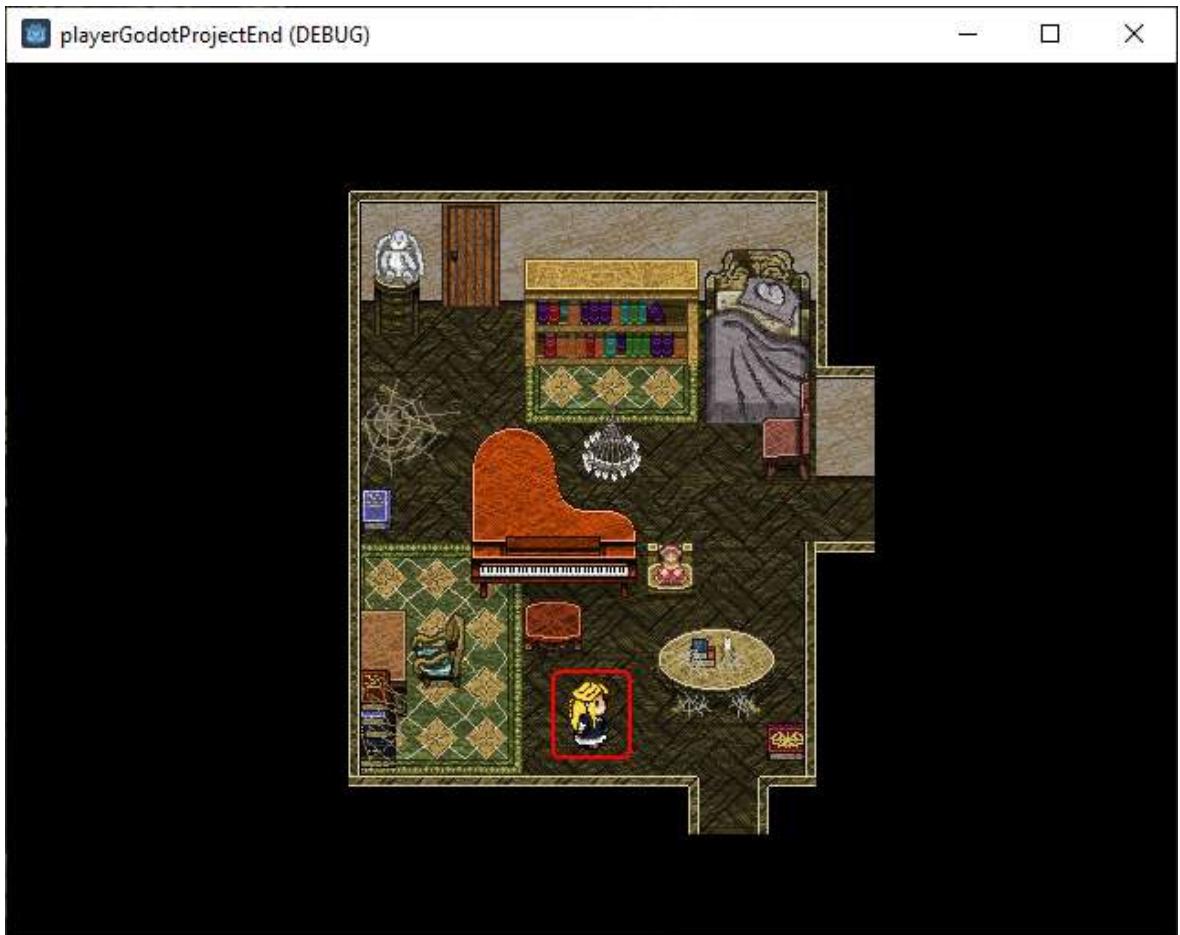
```

eseguiamo il passaggio di stato, ovvero, attraversiamo la freccia della macchina a stati del player, dallo stato `Idle` allo stato `Walk`

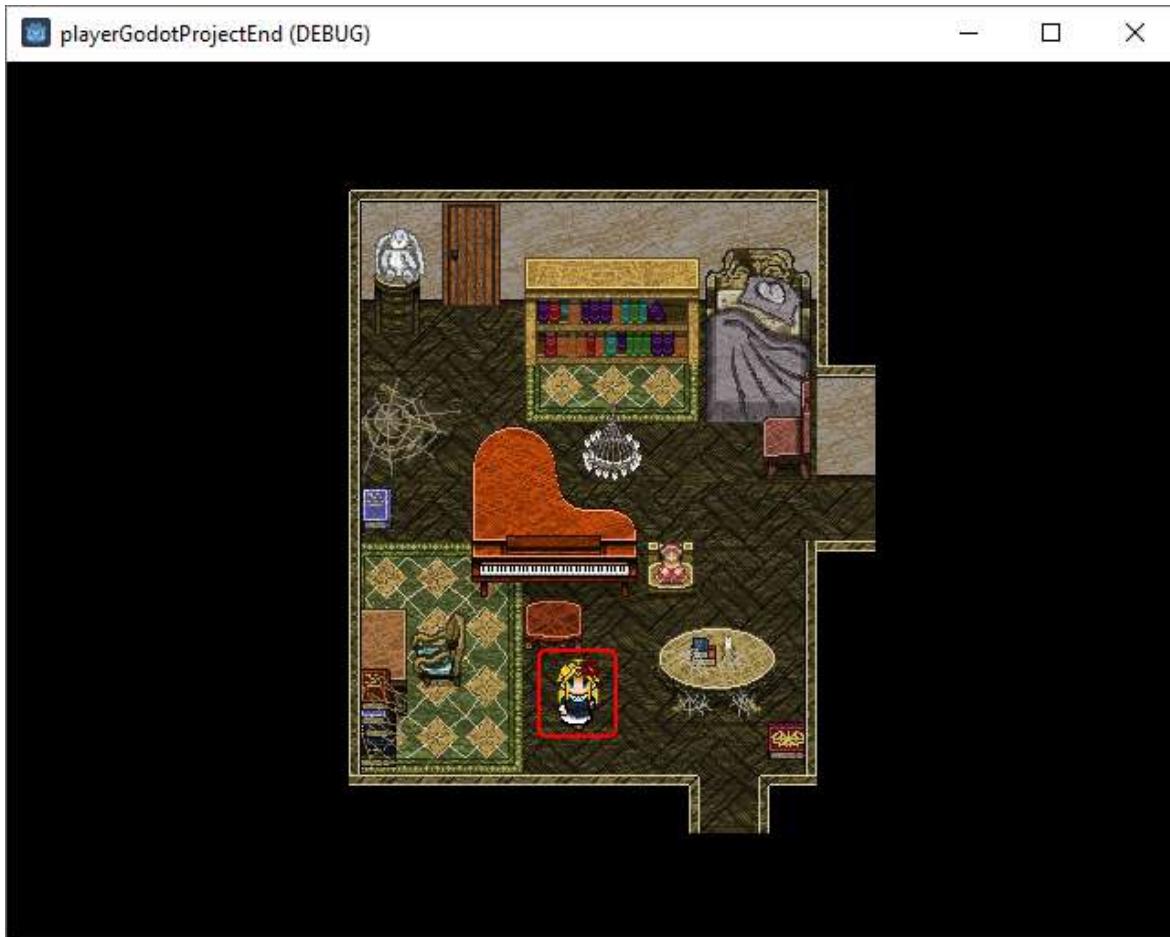
Riga 43 invece è l'esatto opposto, ovvero, transiamo dallo stato `Walk` allo stato `Idle`.

```
31     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
32     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
33     # normalizziamo il vettore per ottenere una direzione
34     input_vector = input_vector.normalized()
35     current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si sta
36
37     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
38         face_direction(current_player_direction) # facciamo guardare al player la direzione che deve guardare
39         animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
40
41         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo il personaggio
42     else:
43         animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
44
45         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, assegniamogli uno zero
46
47     move_and_slide() # muoviamo il personaggio sullo schermo
48
49     func face_direction(direction: Vector2) -> void:
50         for parameter in ANIMATION_TREE_PARAMETERS: # cicliamo ogni crosshair degli stati dell'AnimationTree
51             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato dell'AnimationTree
52 |
```

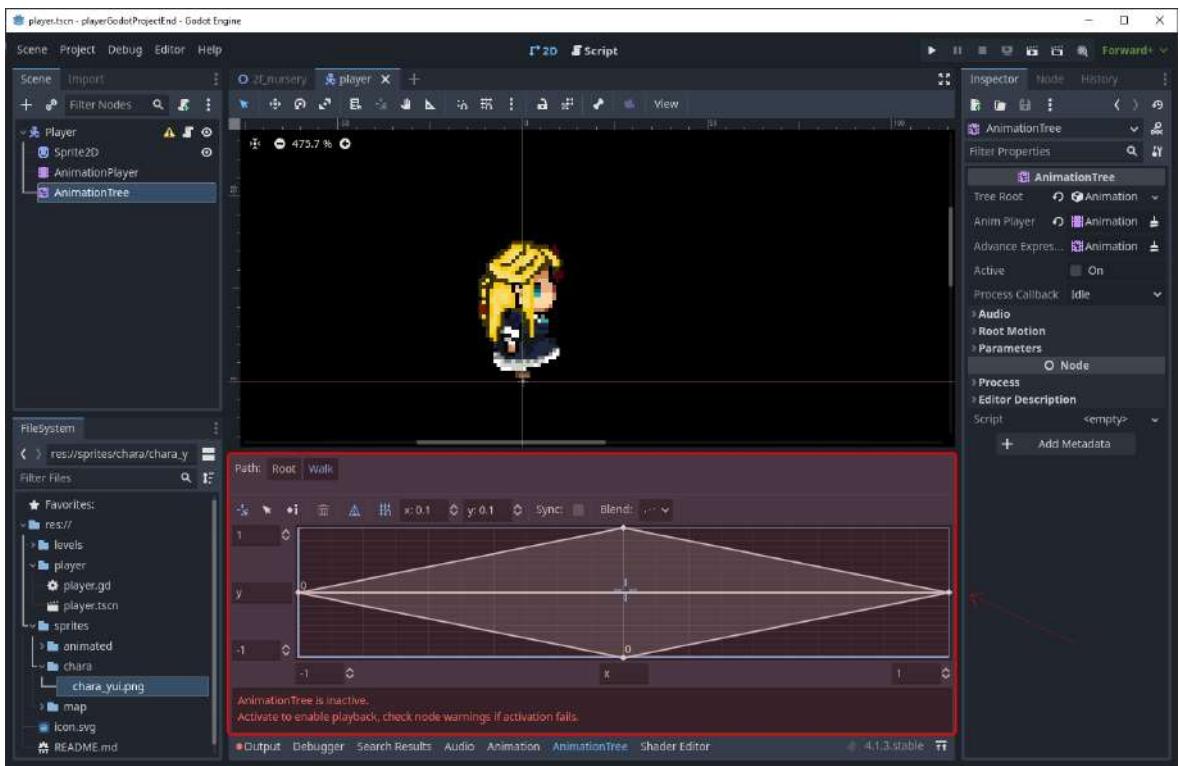
Con questo script, il nostro player verrà dunque animato mentre sta camminando



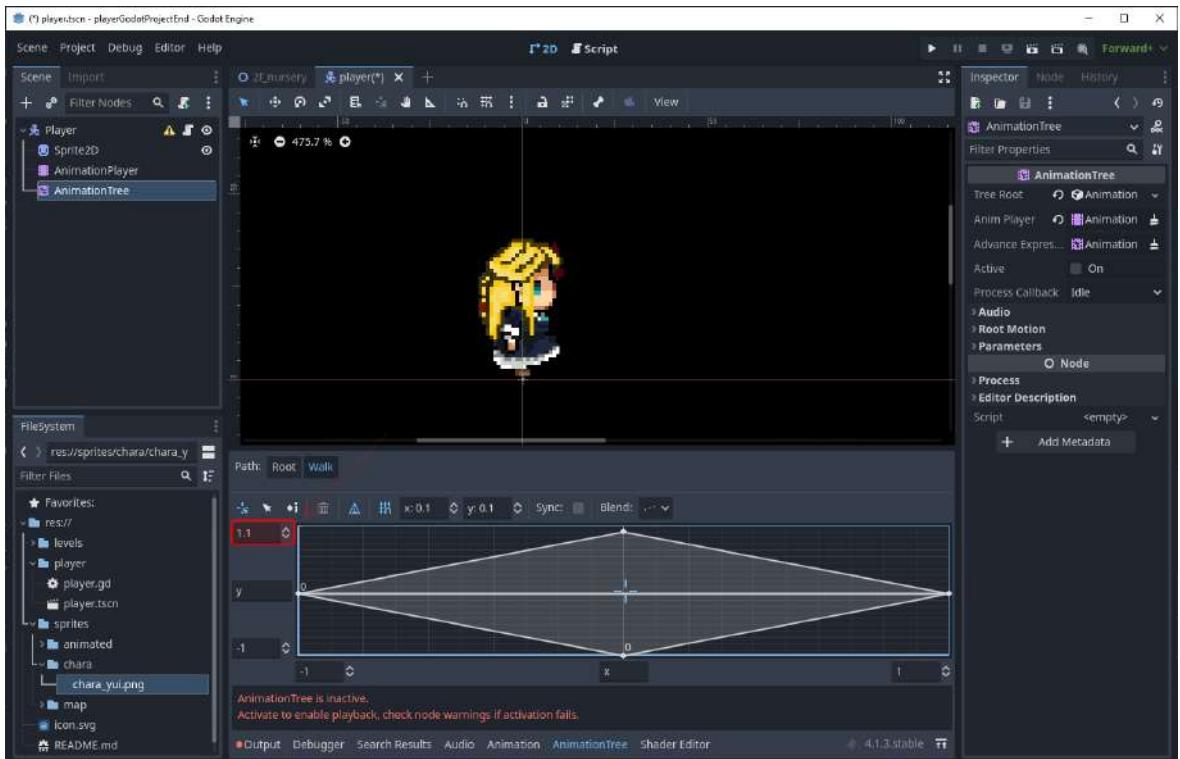
Notiamo però che quando camminiamo sulle diagonali (soprattutto in basso a destra), l'animazione riprodotta è quella sbagliata



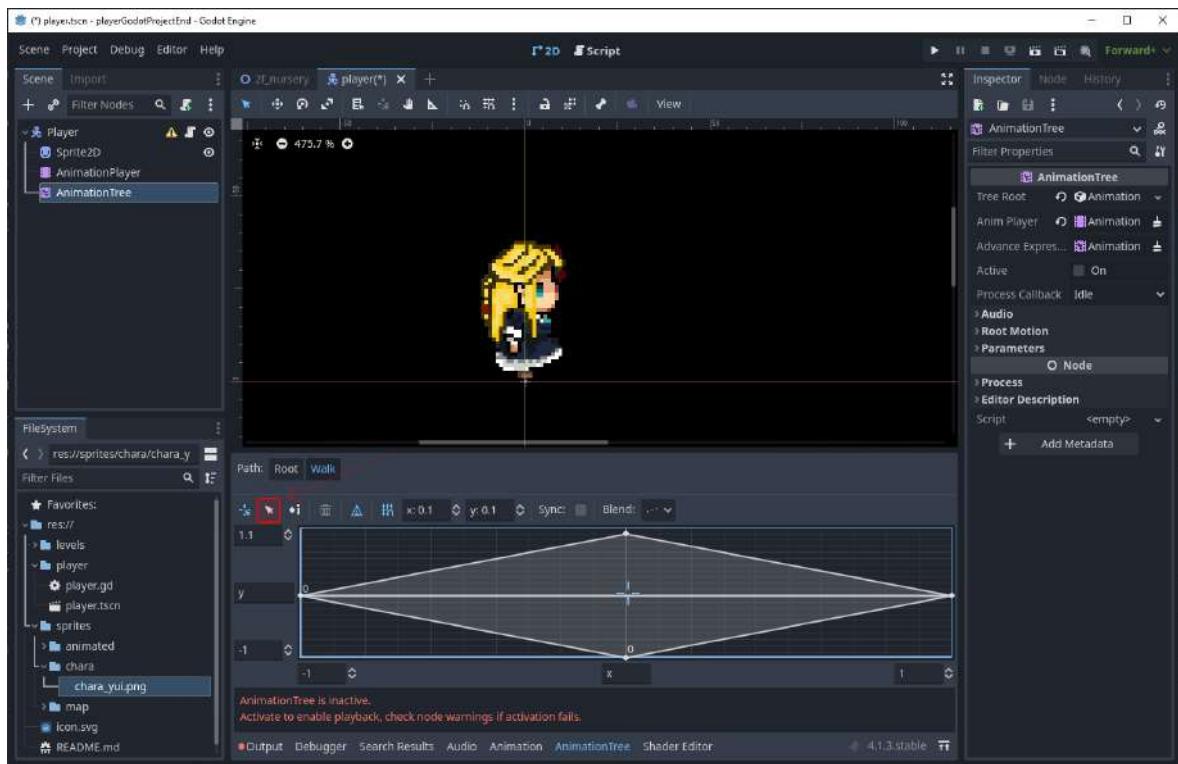
Questo significa che, lungo le diagonali, dobbiamo dare maggiore priorità all'asse x rispetto che all'asse y. Per farlo, nell'editor dello spazio dello stato [Walk](#)



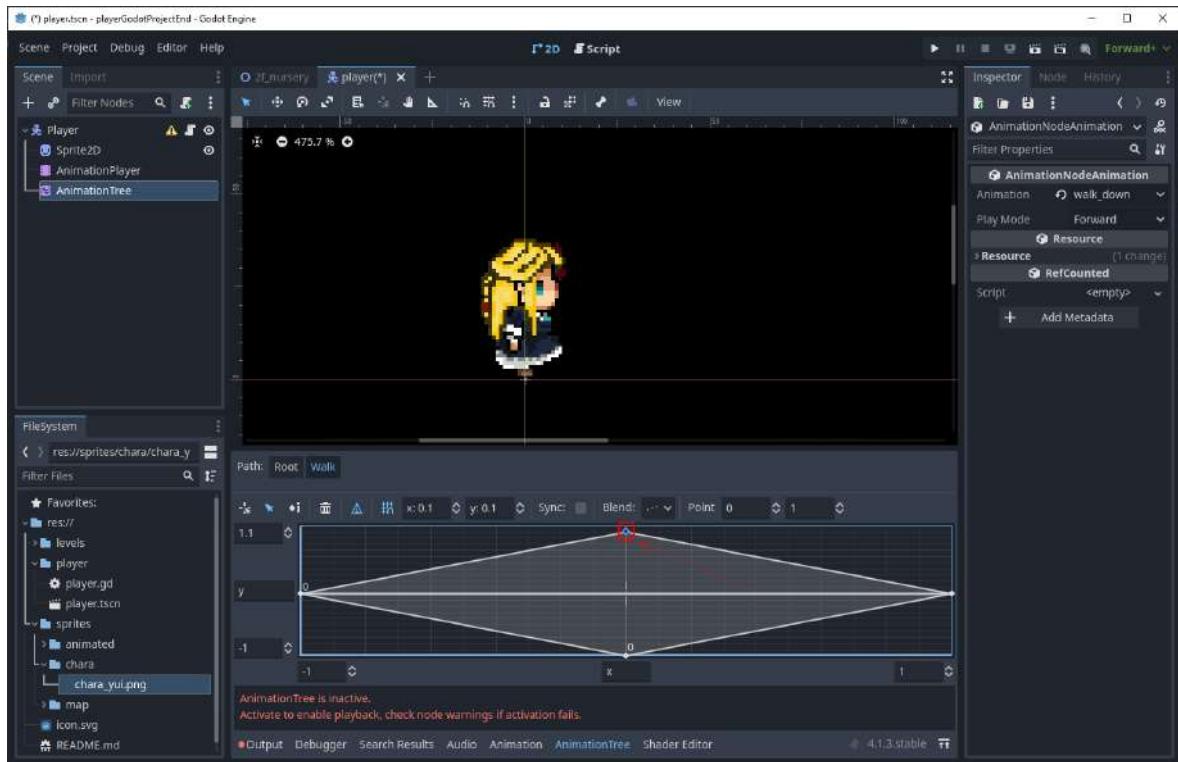
impostiamo il valore superiore dell'asse y a **1.1**



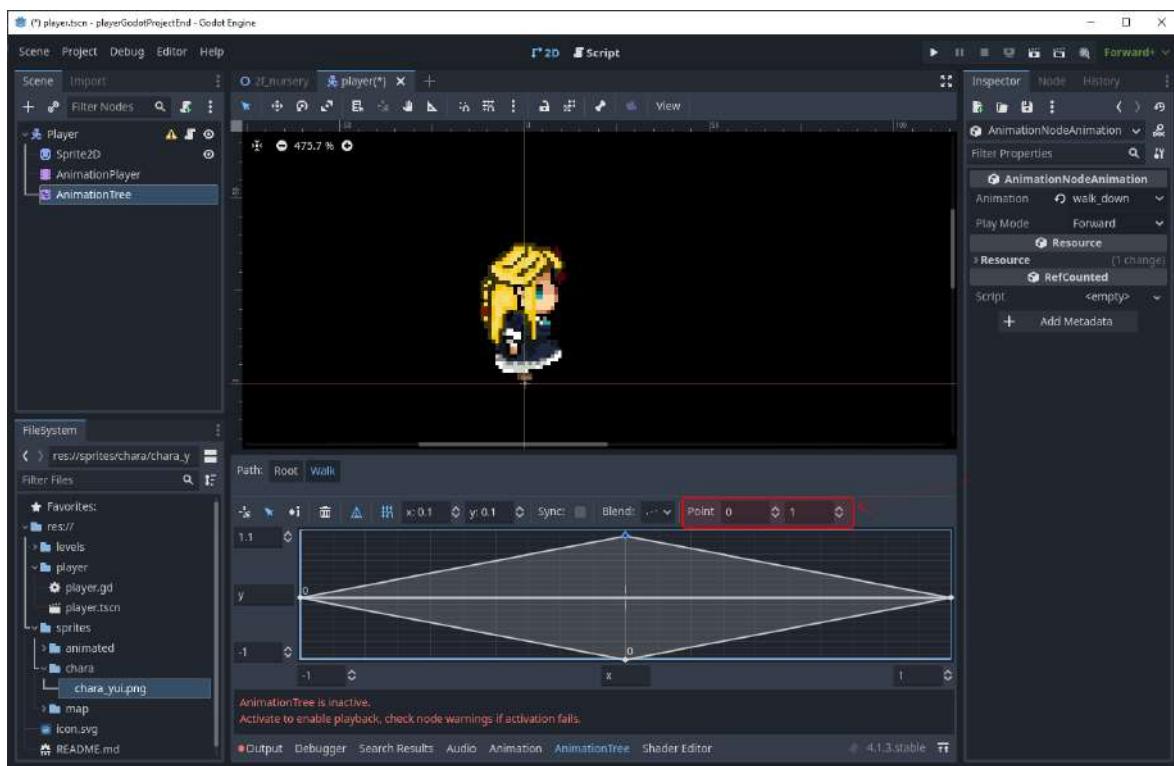
clicchiamo sull'icona della freccia



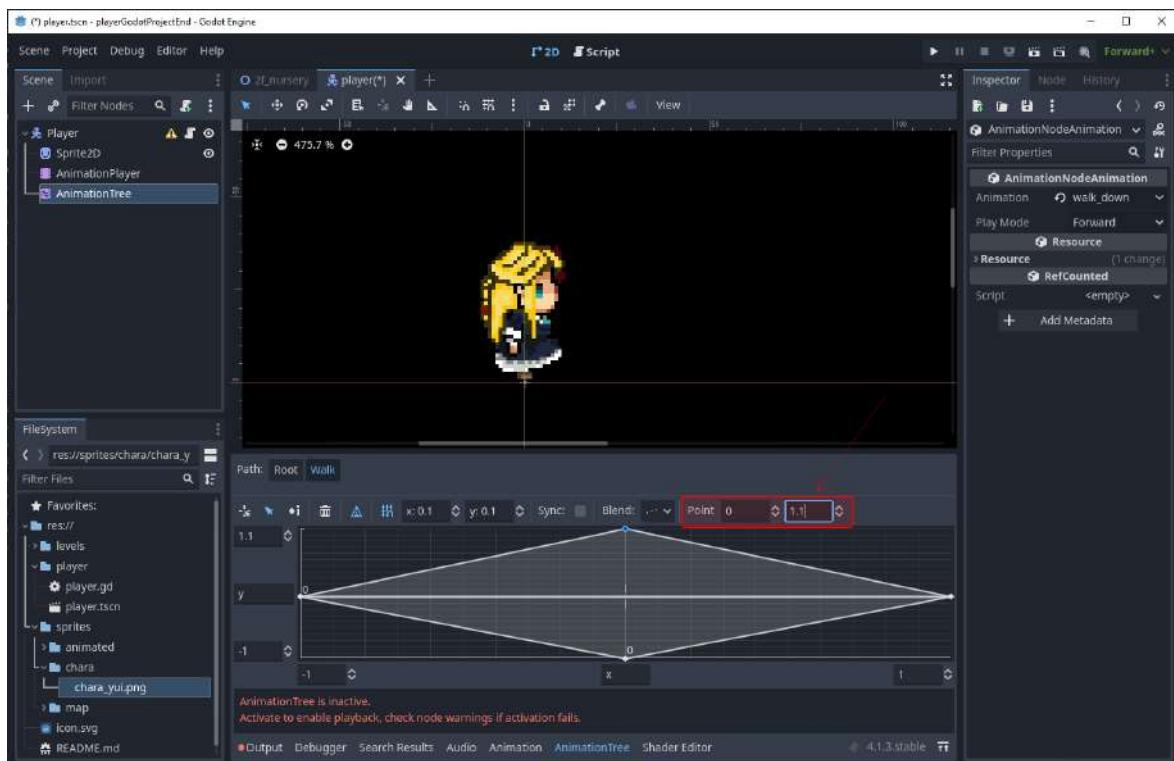
clicchiamo sul punto che rappresenta l'animazione `walk_down` (ricordiamo che l'asse y è invertito)



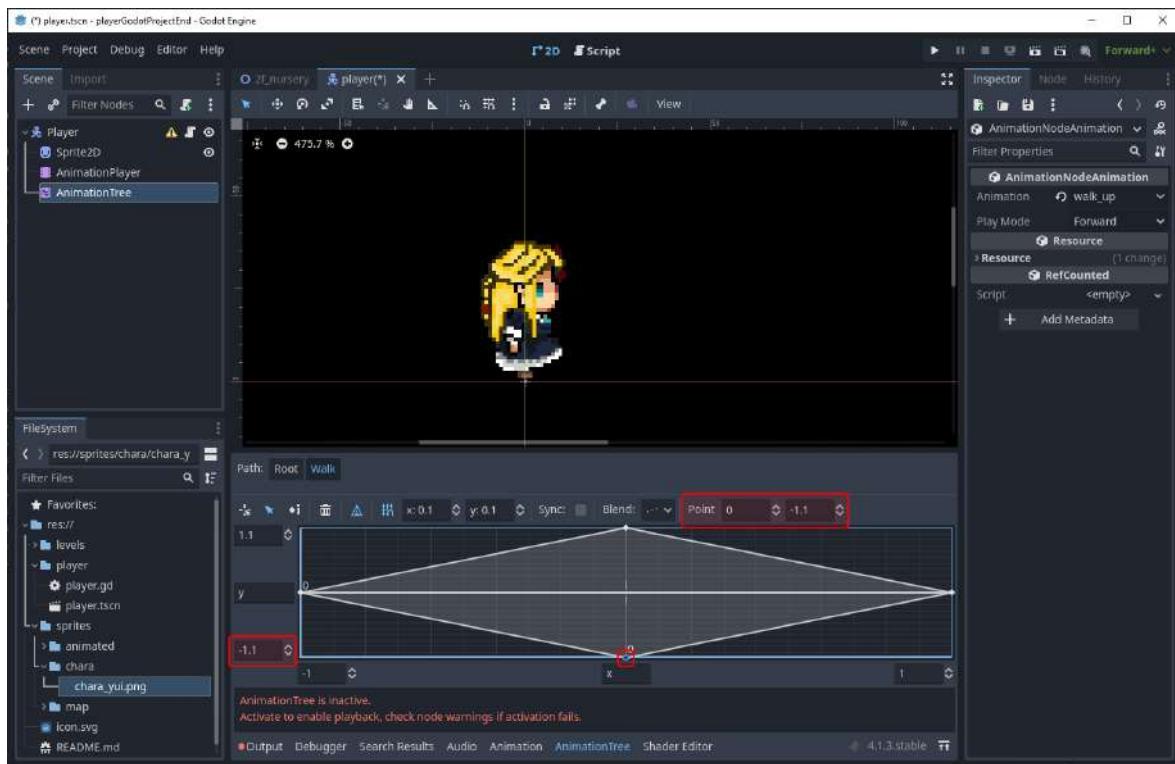
e ne cambiamo la posizione da `(0, 1)`



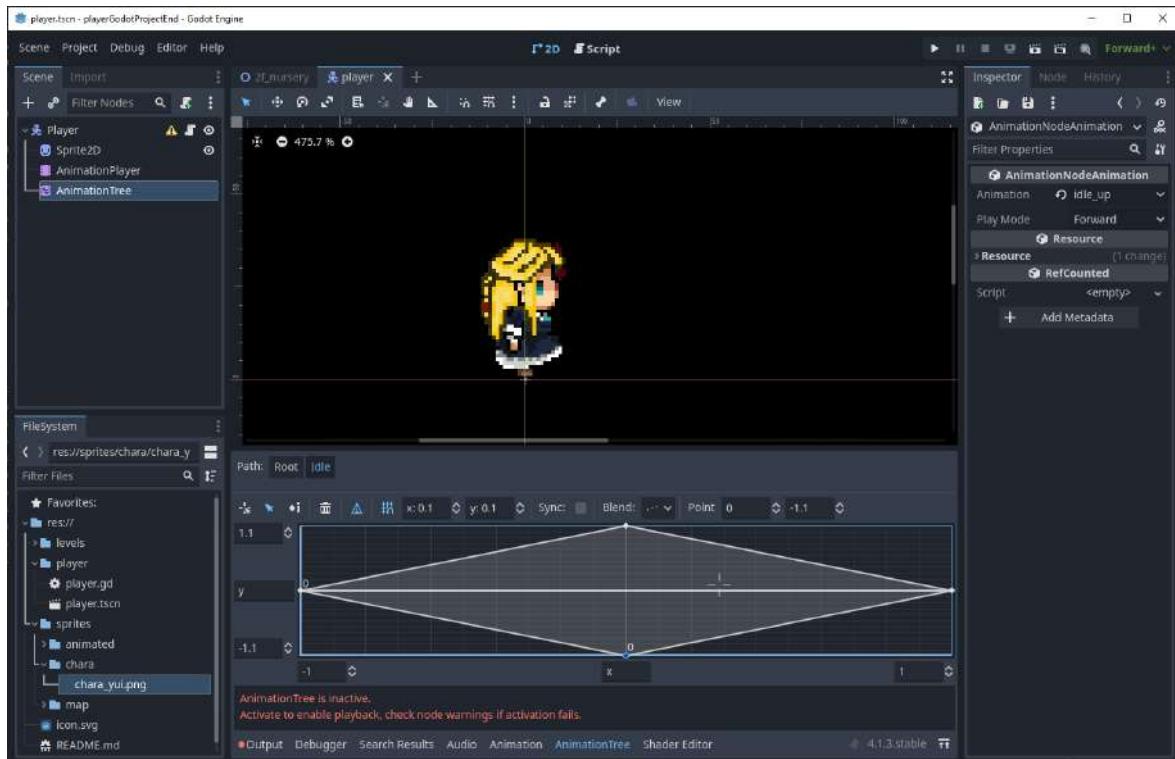
a (0, 1.1)



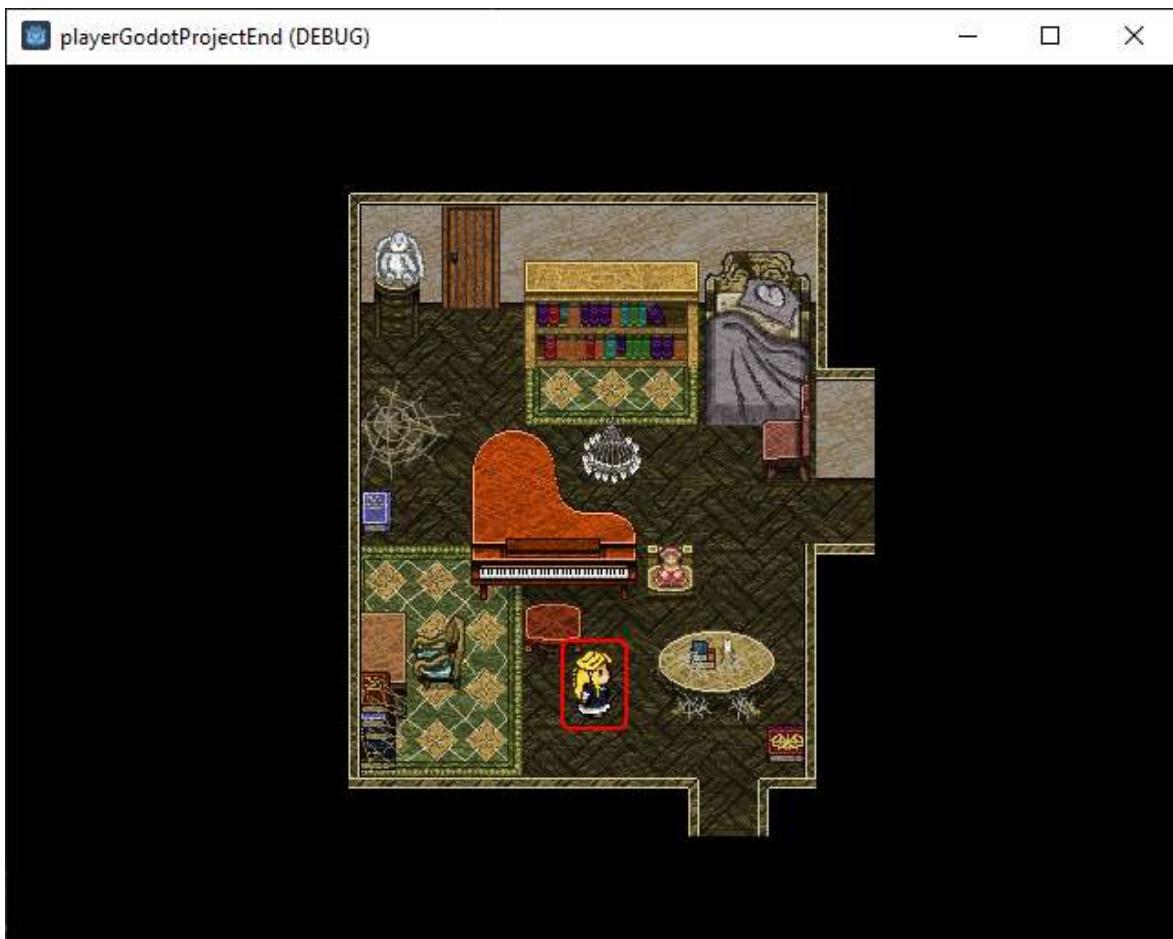
facciamo la stessa cosa con il punto dell'animazione `walk_up`, impostando il valore inferiore dell'asse y a -1.1



Facciamo la stessa cosa con lo stato **Idle**

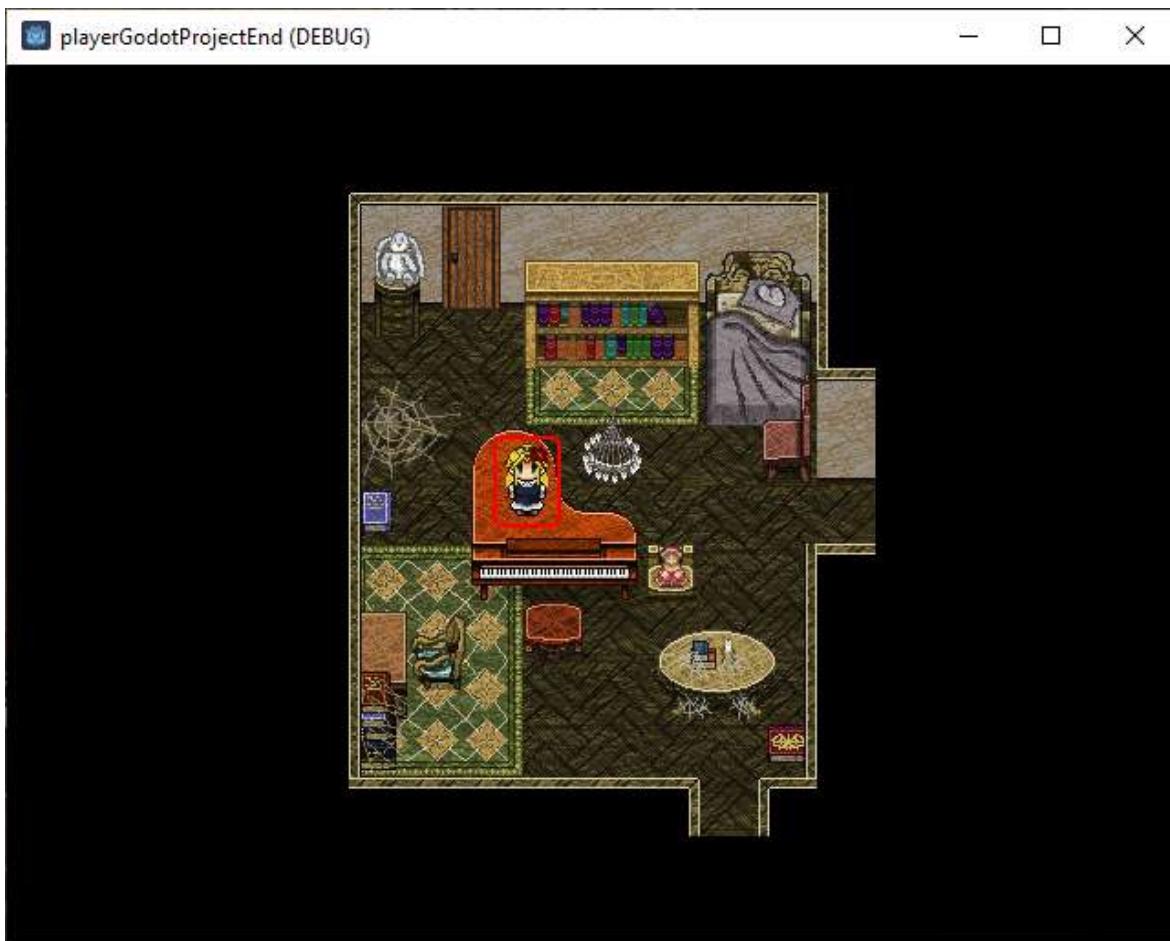


Adesso il nostro player camminerà correttamente lungo le diagonali



#### 2.4.5 Collisioni del player con il mondo

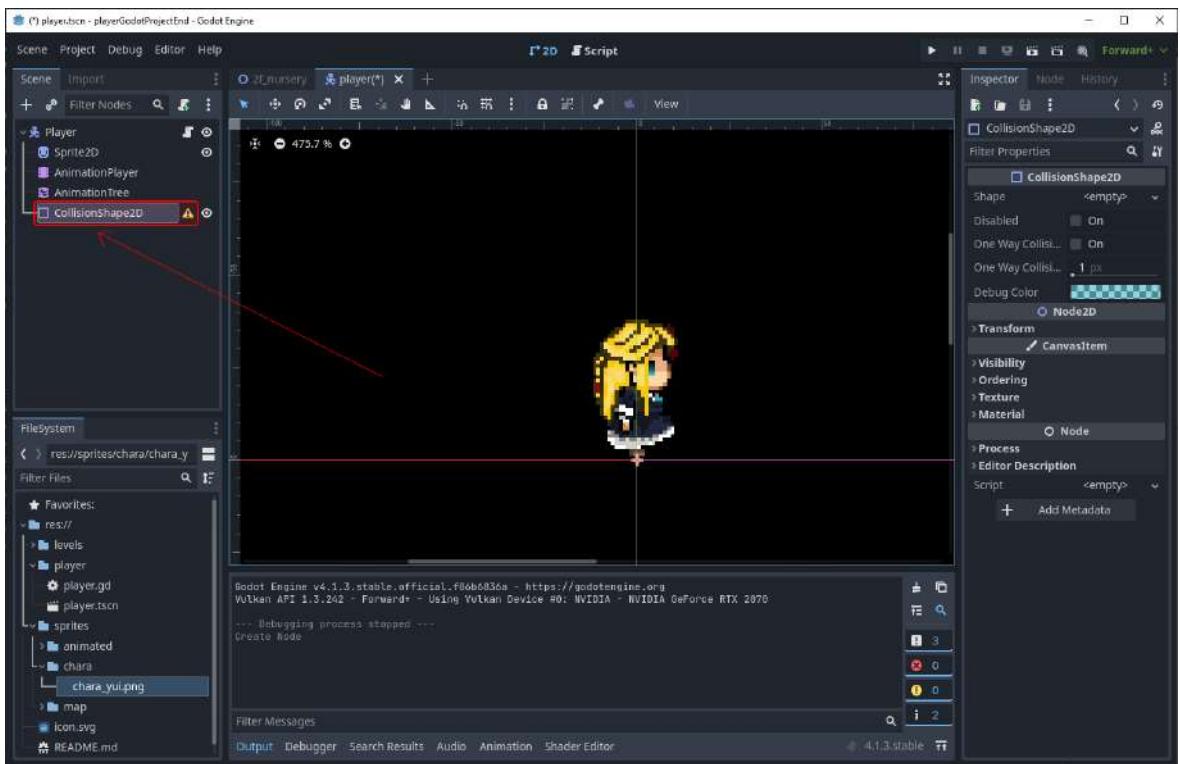
Ora che il nostro player è finalmente libero di correre animato all'interno della stanza, ci farebbe magari piacere se non salisse, ad esempio, con i piedi sul pianoforte



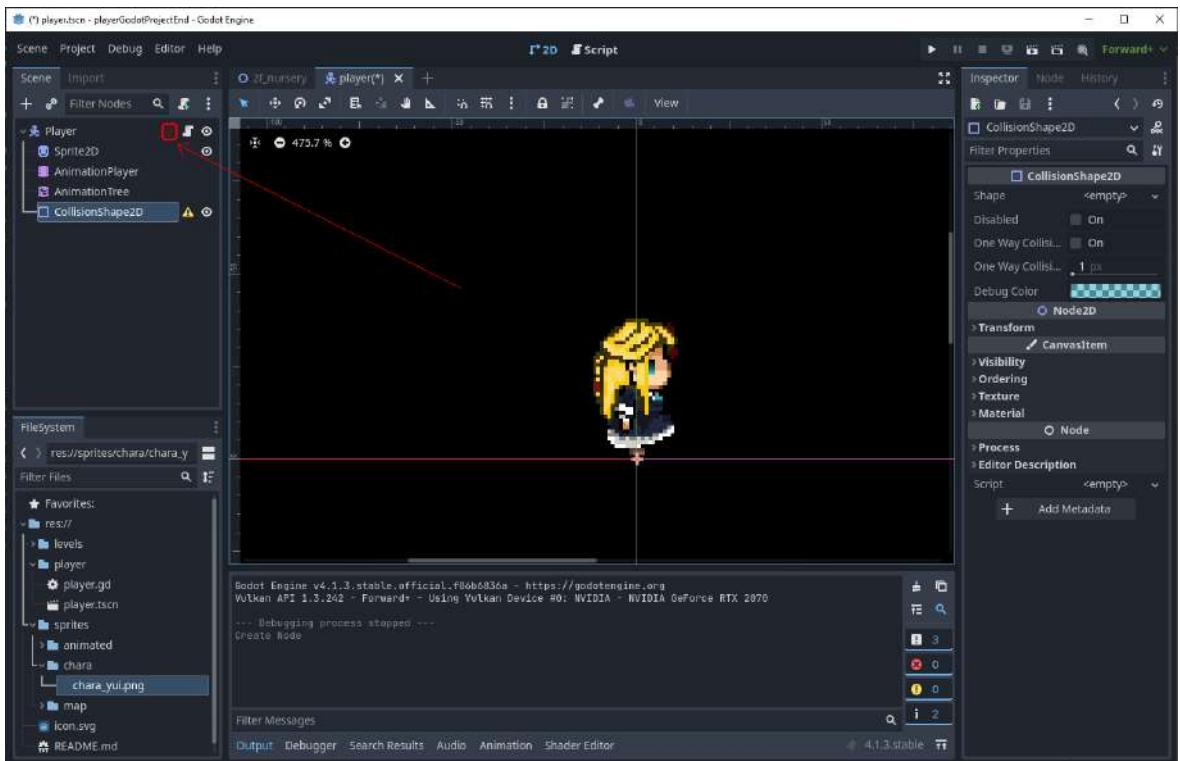
Per fare in modo che il player *collida* con gli oggetti della stanza, o comunque del mondo, non ci resta che implementare il concetto di **collision** (**collisione**).

Nel game development, si ha una **collision** (**collisione**) quando due oggetti nel gioco si intersecano o sono entrati in contatto tra di loro. La logica in grado di rilevare quando si verifica una collisione è chiamata **collision detection** (**rilevamento delle collisioni**). La risposta al rilevamento di una collisione è detta invece **collision response** (**risposta di collisione**).

Nel caso del nostro personaggio, come prima cosa aggiungiamo un nodo **CollisionShape2D**



`CollisionShape2D` è un nodo che fornisce una `Shape2D` a un suo nodo padre `CollisionObject2D` (da cui `CharacterBody2D` estende) e ne permette la modifica. È proprio grazie alla forma fornita dal nodo `CollisionShape2D` che, il nodo `Player`, ovvero un nodo `CharacterBody2D`, diventa un oggetto solido a tutti gli effetti. Questa "trasformazione" è testimoniata anche dalla scomparsa del simbolo di warning



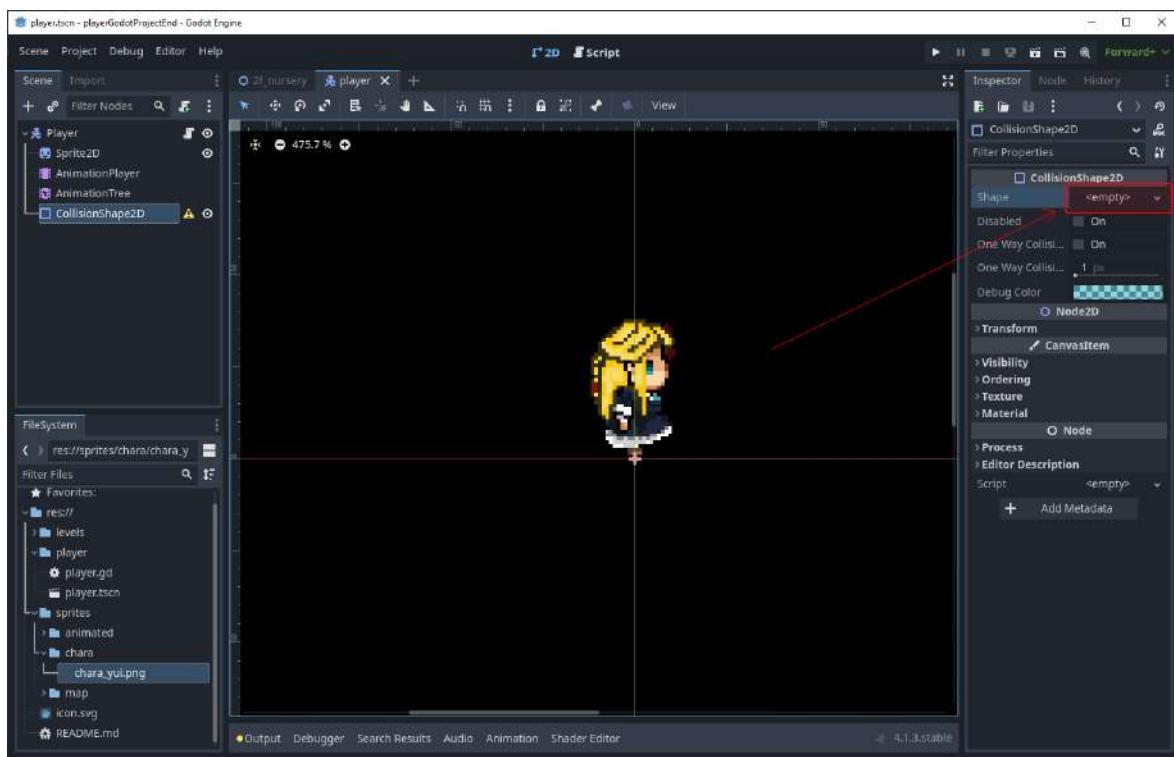
La classe `Shape2D` è una semplice classe **astratta** pensata per tutte le forme 2D.

Ma cos'è una classe **astratta**? Per **data abstraction (astrazione dei dati)** si intende quel particolare processo in cui si nascondono dei dettagli specifici, e si mostrano all'utente solamente le informazioni essenziali. L'astrazione può essere raggiunta o tramite le **classi astratte** o tramite le **interfacce**, lì dove un'**interfaccia** è un modo per raggruppare insieme metodi correlati con corpo vuoto.

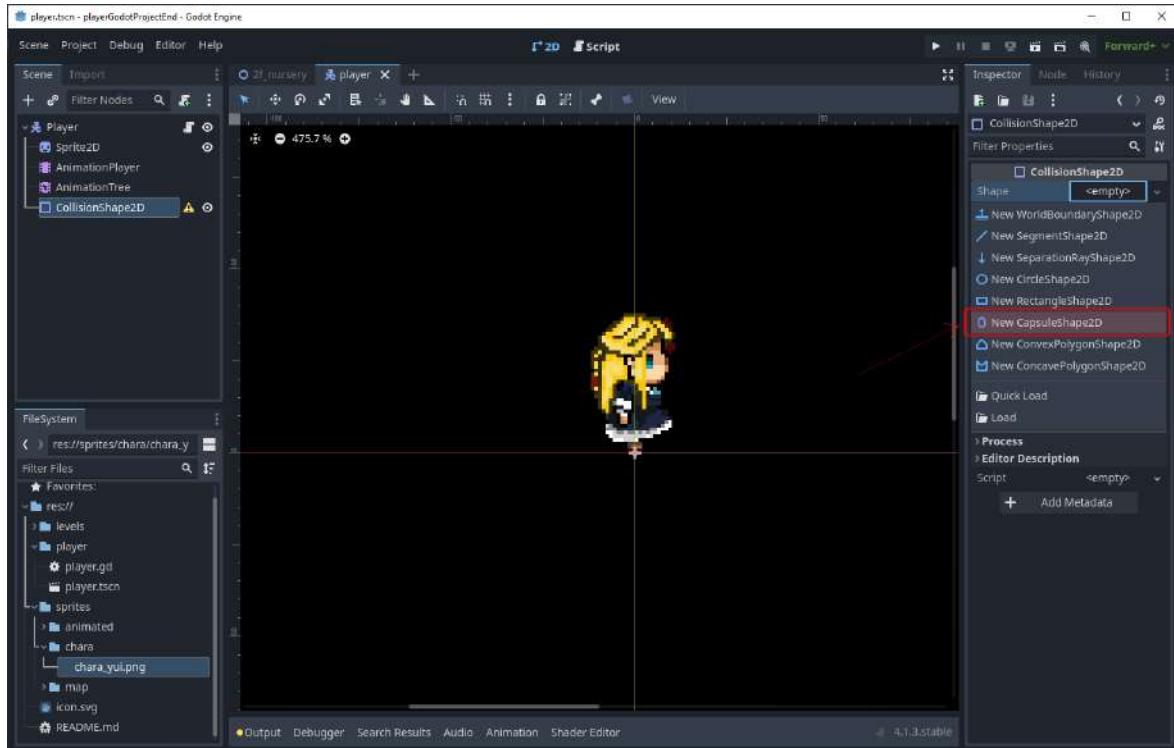
Per capire a cosa servono le classi astratte, facciamo un esempio strettamente legato al concetto di `Shape2D`. Come vedremo a breve, per rappresentare una collisione dovremmo disegnare cerchi, rettangoli, linee, ecc.... Questi oggetti possiedono tutti delle proprietà (come ad esempio: posizione, orientamento, colore, ecc...) e dei comportamenti (per esempio: `moveTo()`, `rotate()`, `resize()`) in comune. Alcune di queste proprietà o comportamenti sono gli stessi per tutti le forme geometriche (come ad esempio la posizione), mentre altri richiedono un'implementazione diversa (come la funzione per calcolare l'area o per ridimensionare l'oggetto). Tutti questi oggetti che rappresentano forme geometriche sono dunque in grado di disegnare e ridimensionare sé stessi; differendo solo nel modo in cui lo fanno. Questa è la situazione perfetta per una superclasse astratta: la classe `Shape2D`. Nella dichiarazione della classe `Shape2D`, infatti, si raccolgono e sfruttano tutte le similarità e si fa in modo che le classi che poi andranno a definire le funzioni geometriche, come ad esempio le classi `CircleShape2D` o `RectangleShape2D`, ereditino dalla classe `Shape2D`. Altro aspetto importante dell'astrazione è che una classe astratta non può essere istanziata, ovvero, non si possono creare oggetti a partire da una classe astratta. Questo principalmente perché una classe astratta è, per forza di cose, *incompleta* (i metodi non sono implementati) e dunque non utilizzabile.

Anche la classe `CollisionObject2D` è a sua volta una classe astratta che modella oggetti fisici 2D.

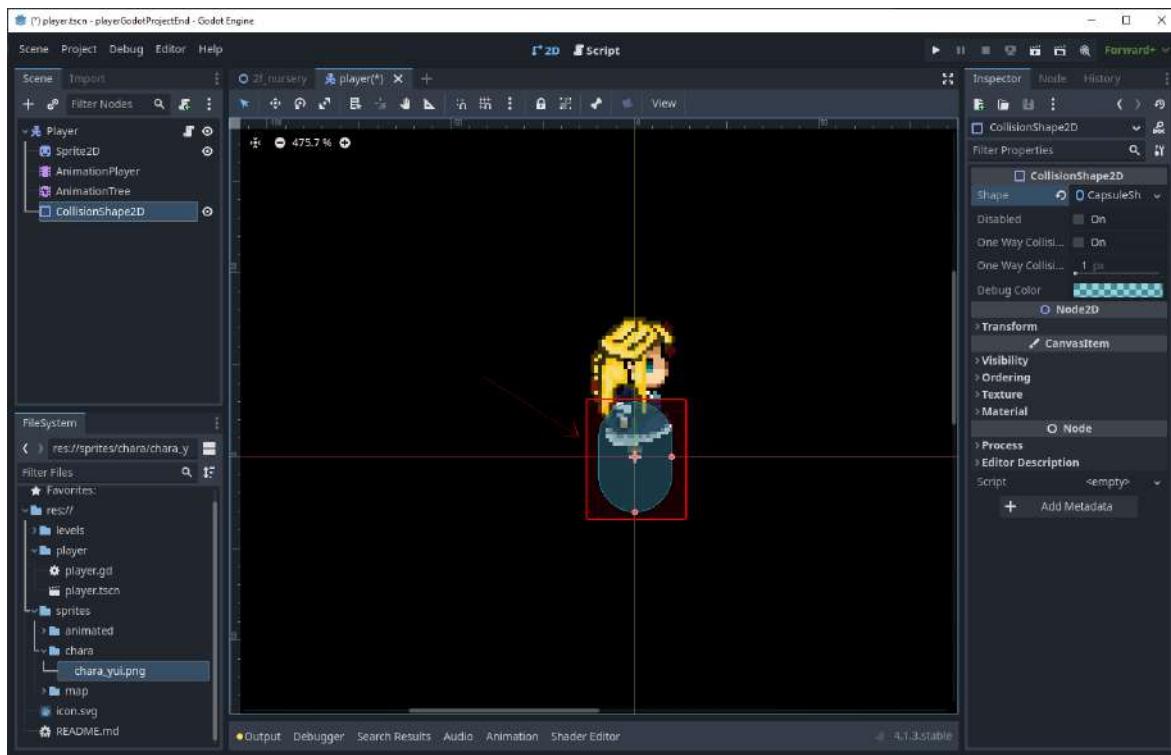
Tornando al nostro `CollisionShape2D`, dopo averlo prima selezionato, clicchiamo sulla voce `<empty>` dell'opzione `Shape`



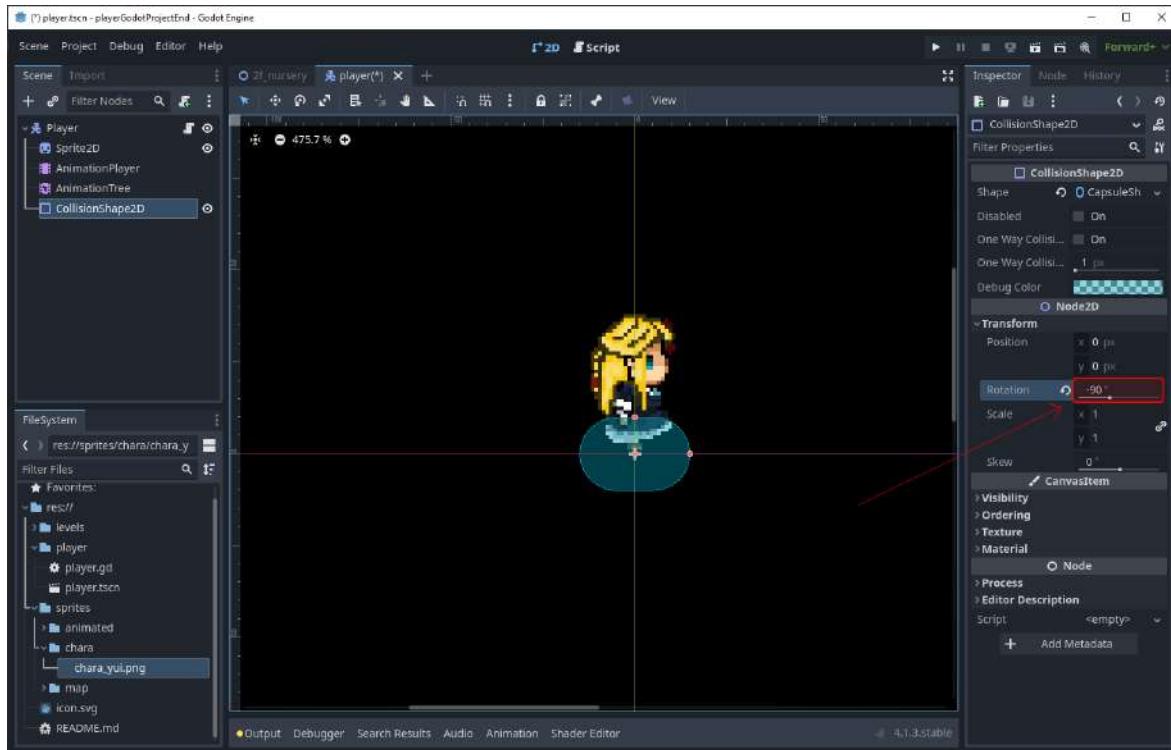
clicchiamo su **New CapsuleShape2D**. Una **CapsuleShape2D**, come dicevamo prima, quando abbiamo introdotto l'astrazione, è semplicemente una **Shape2D** a forma di ellisse (o capsula).



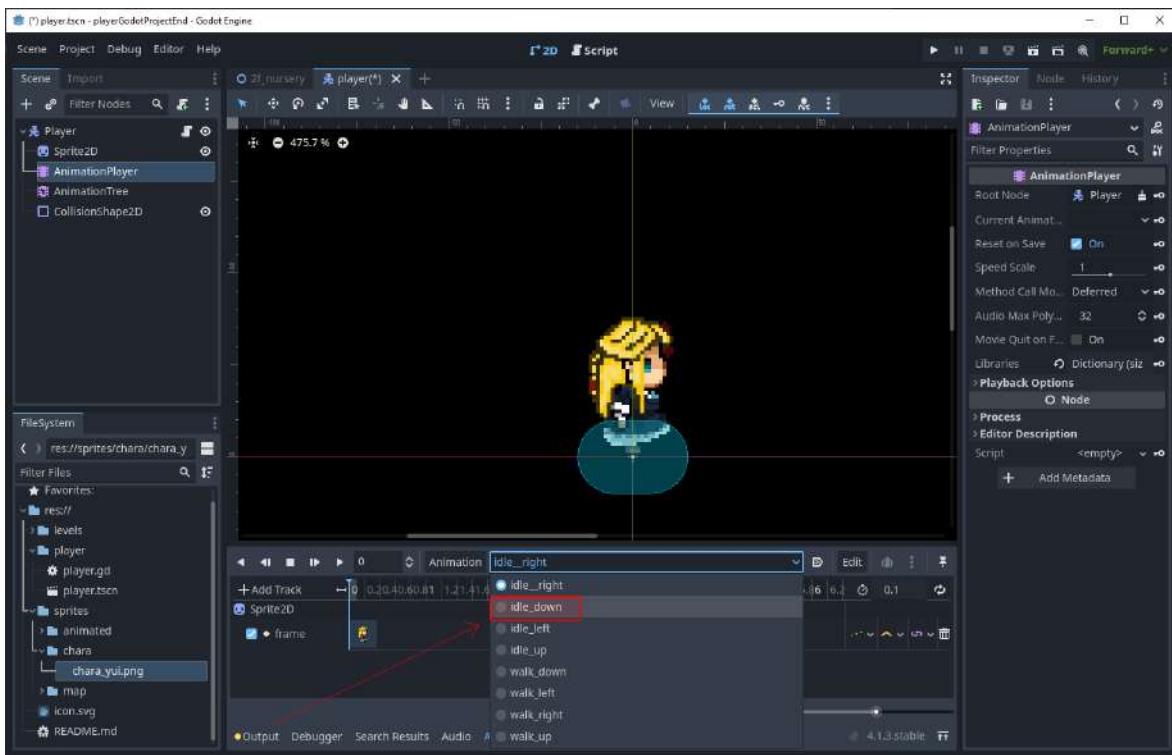
si creerà un nuovo ellisse.



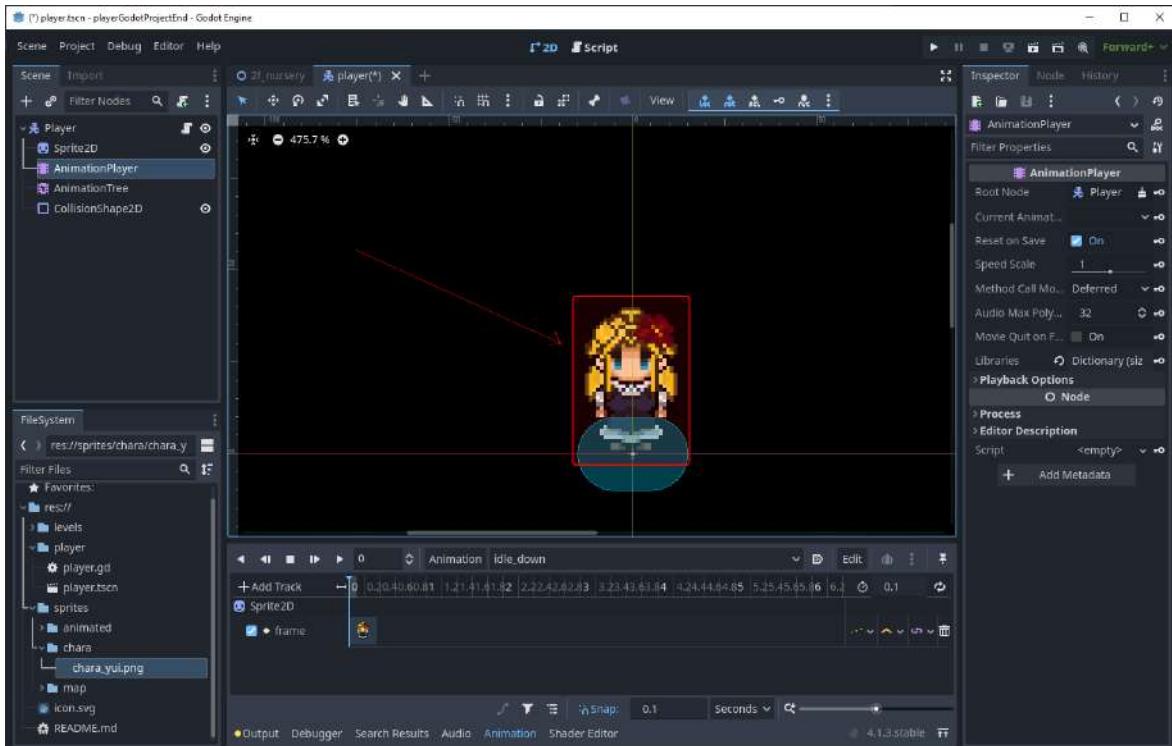
impostiamo a **-90°** il valore dell'opzione **Rotation**, sotto la voce **Transform**



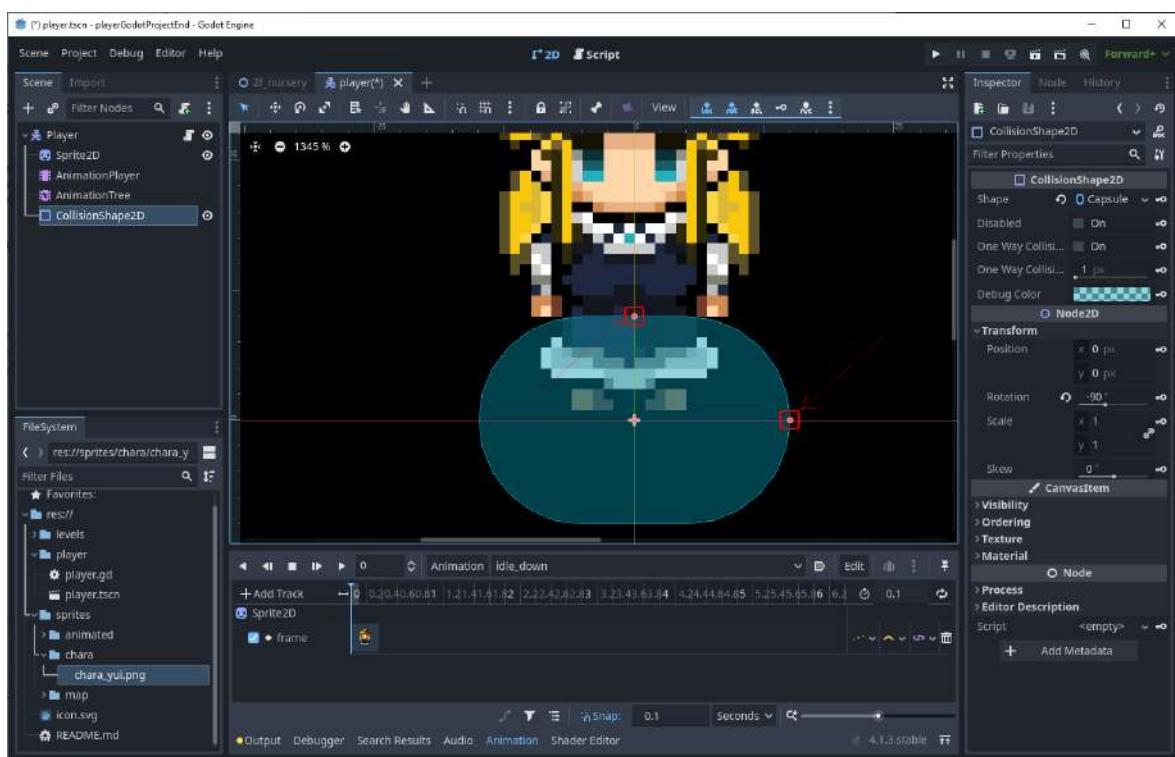
Per meglio posizionare la forma, selezioniamo l'animazione **idle\_down** dal nodo **AnimationPlayer**.



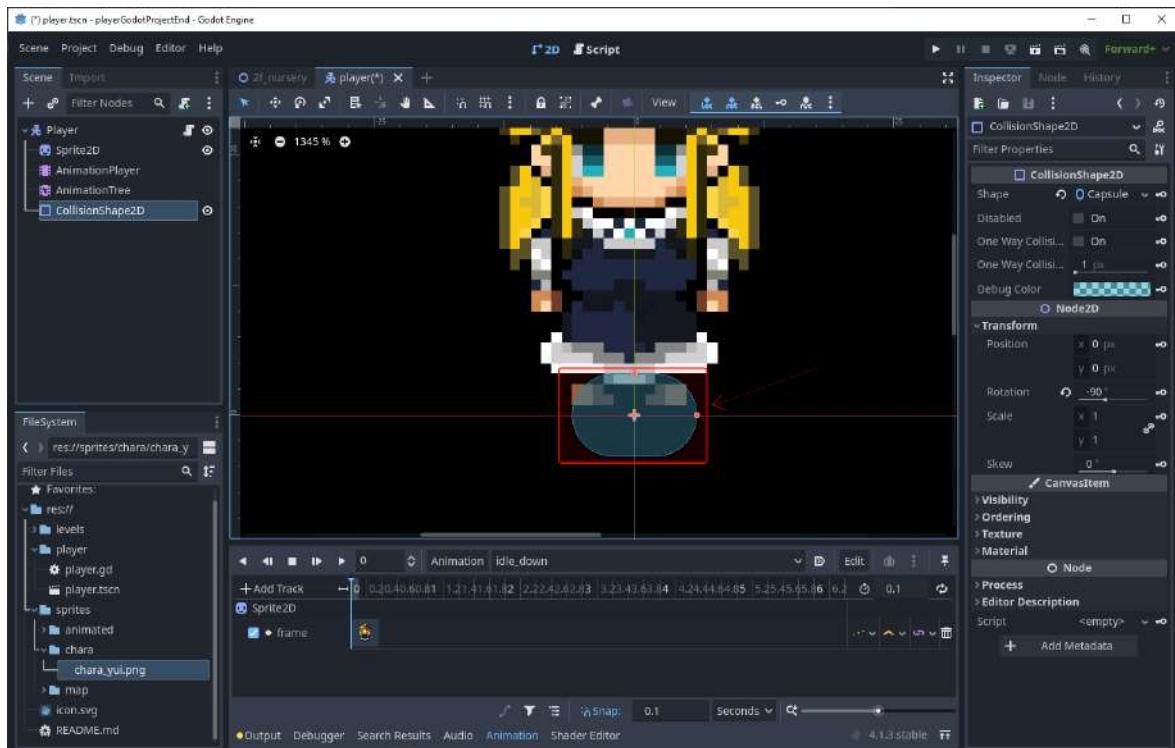
Come possiamo vedere questo farà girare il personaggio. Quando si crea una forma per una collisione, è sempre buona norma, infatti, lavorare su un'animazione ferma e che "sia diretta" verso il basso.



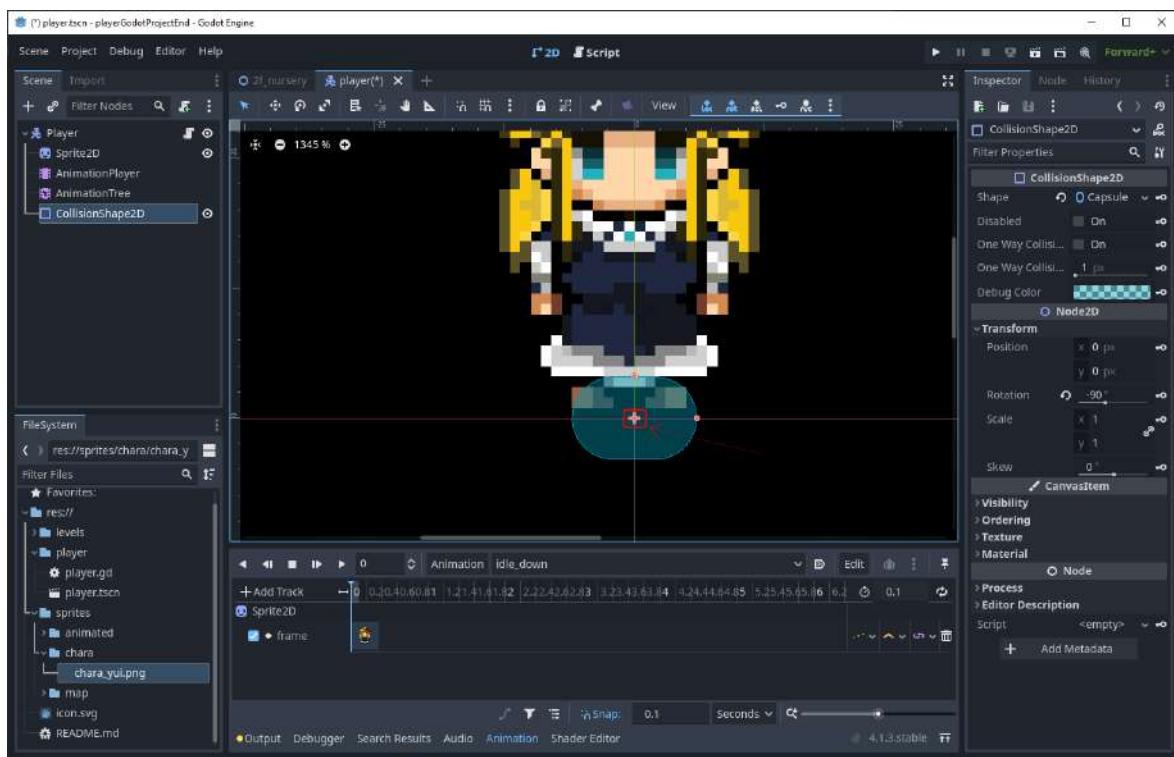
Riselezionando nuovamente il nodo **CollisionShape2D**, usiamo i punti in rosso per modellare la forma geometrica



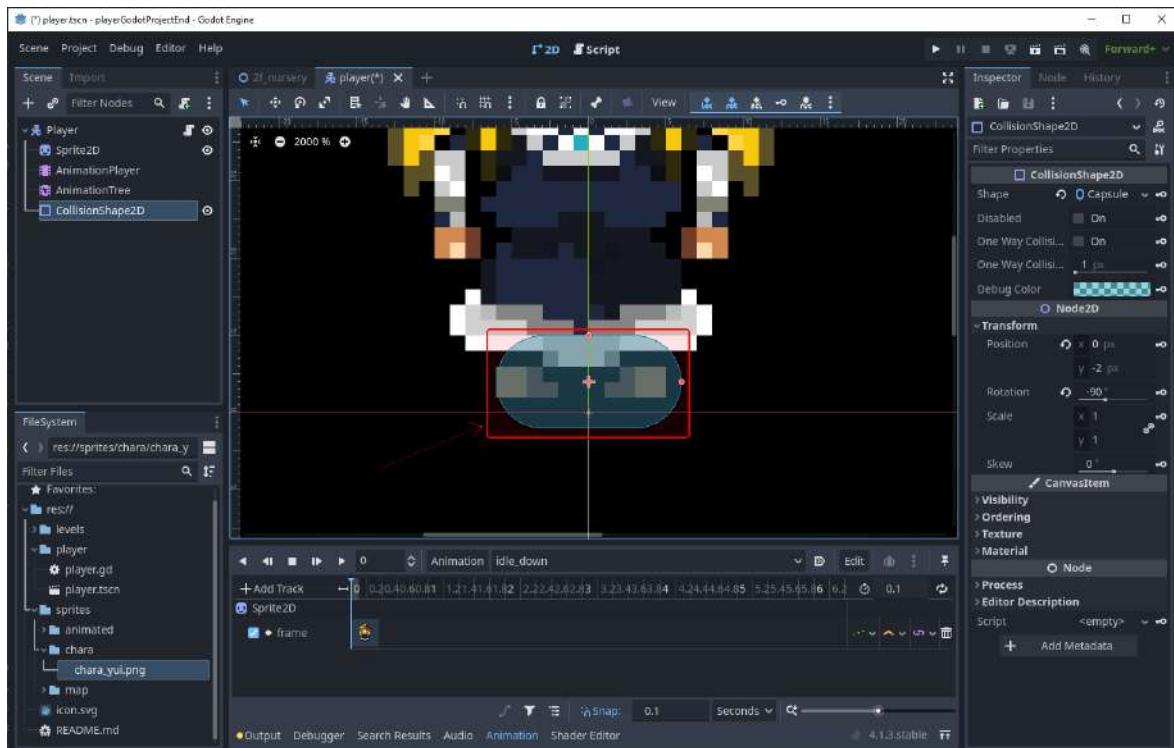
in modo simile a questo



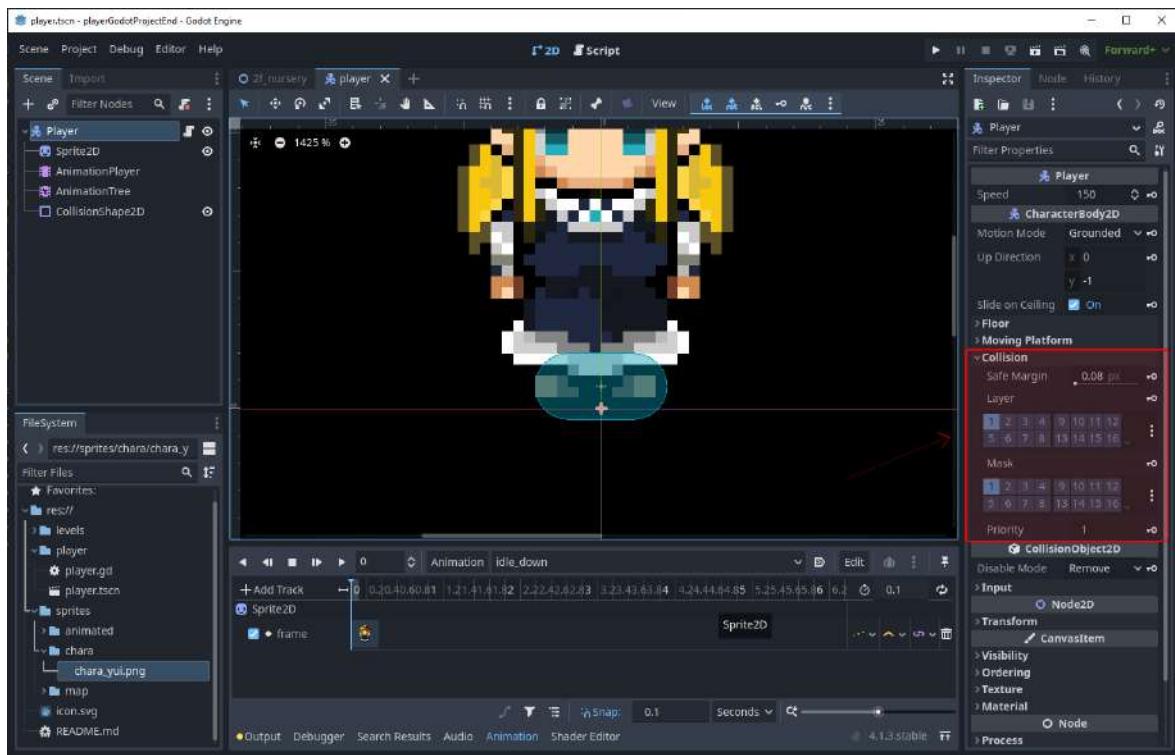
tenendo premuto **ALT**, spostiamo poi il crosshair in rosso



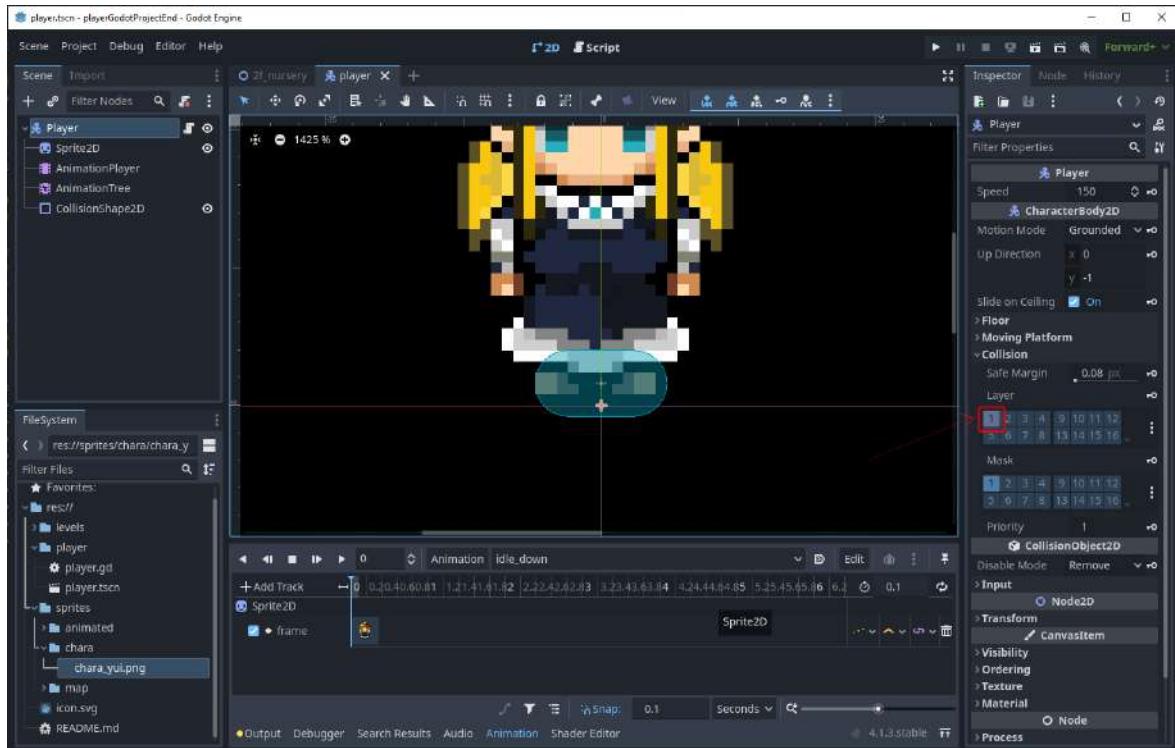
per posizionare la forma in una posizione che va quasi a "coprire i piedi" del player



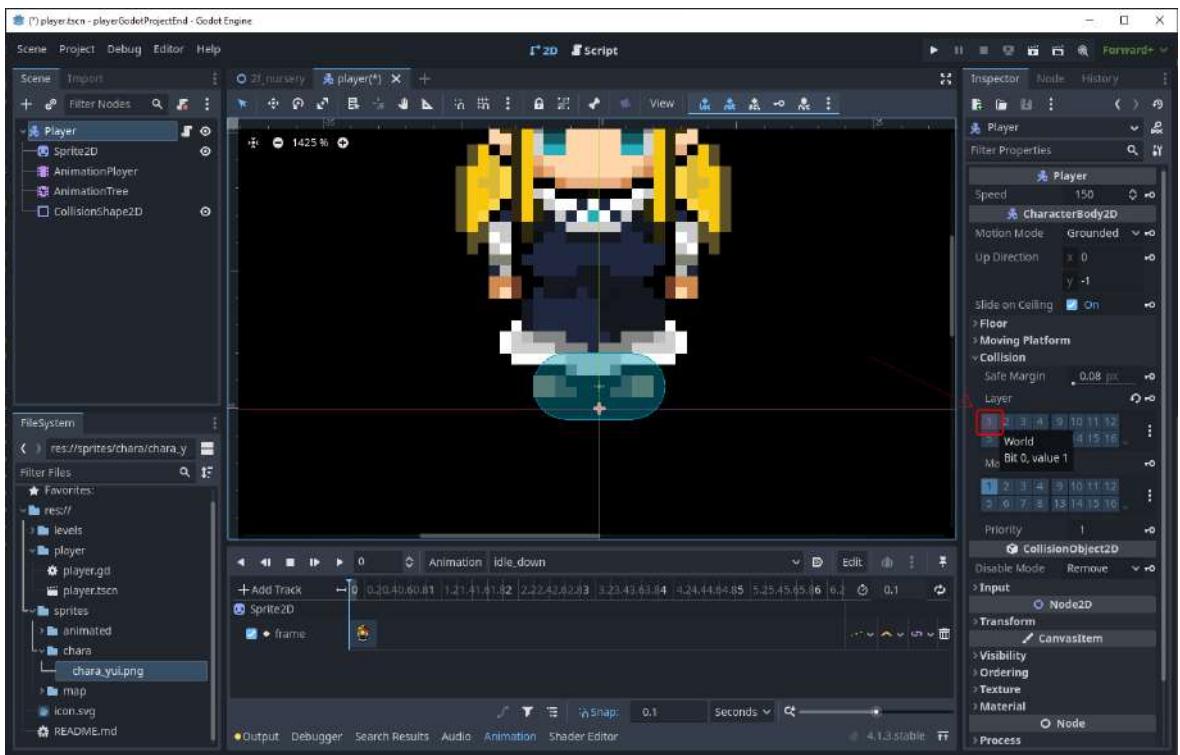
Se adesso clicchiamo sul nodo **Player**, noteremo una voce **Collision**



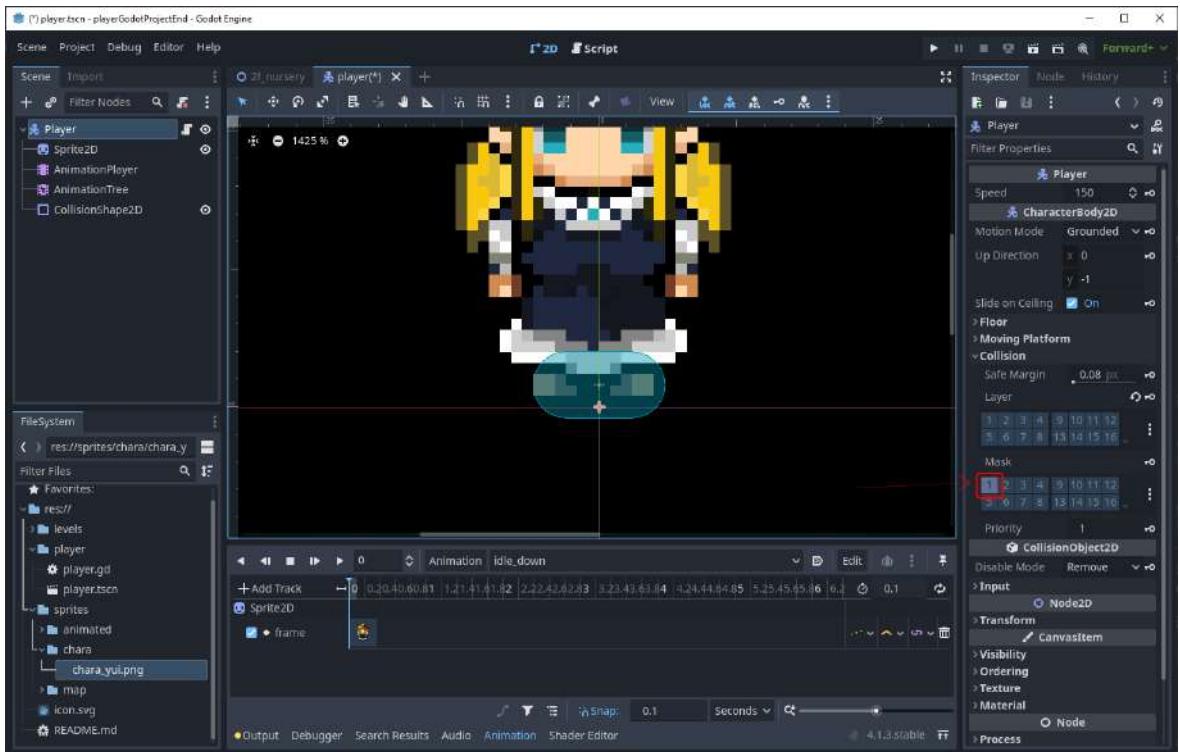
nella quale disabilitiamo l'**1** sotto la voce **Layer**



semplicemente cliccandoci sopra

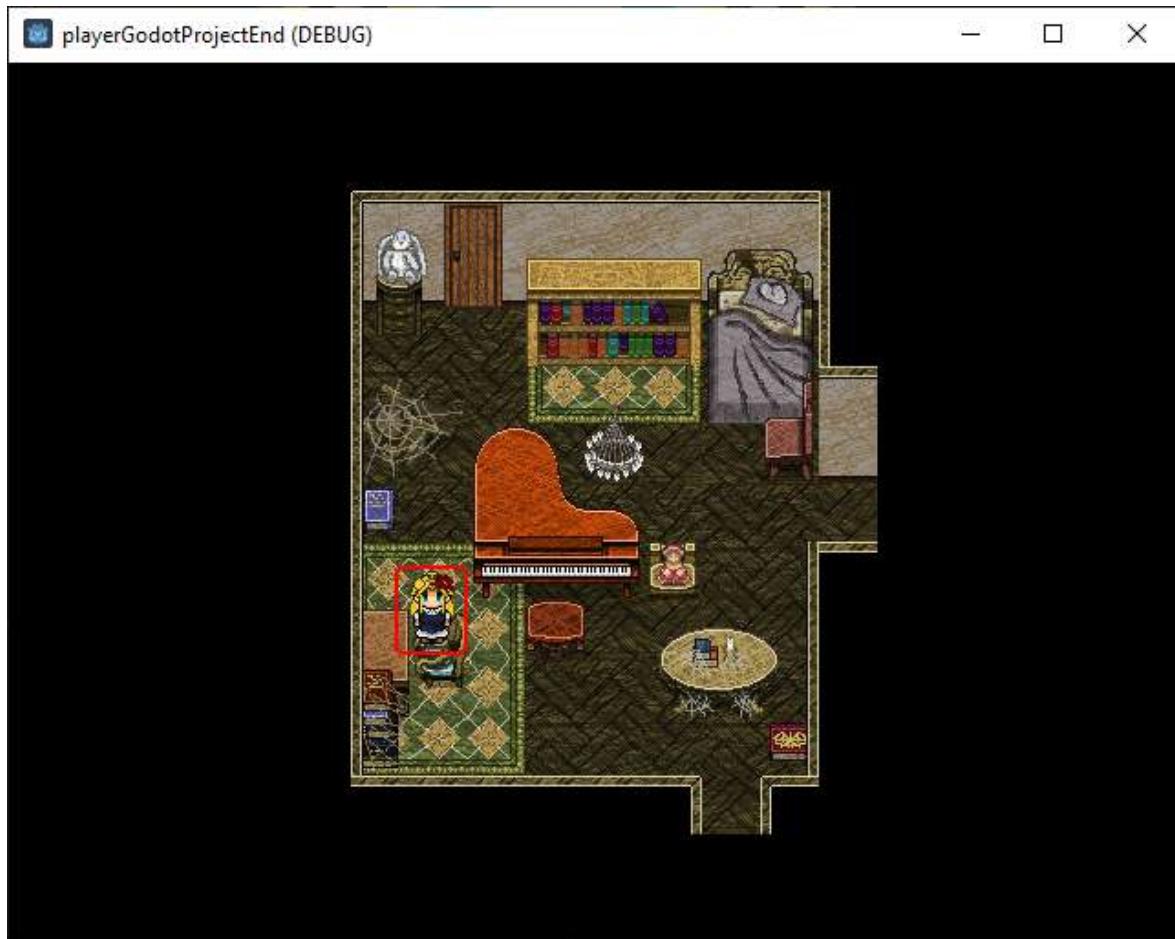


L'unico valore che ci interessa al momento, e che permette la collisione del player con gli oggetti del mondo, è, infatti, l'**1** sotto la voce **Mask**



La logica che si nasconde dietro questo **1**, la vedremo tra qualche pagina, quando parleremo dei "Terreni" nella sezione "Terreni". Per adesso dunque, non ce ne curiamo.

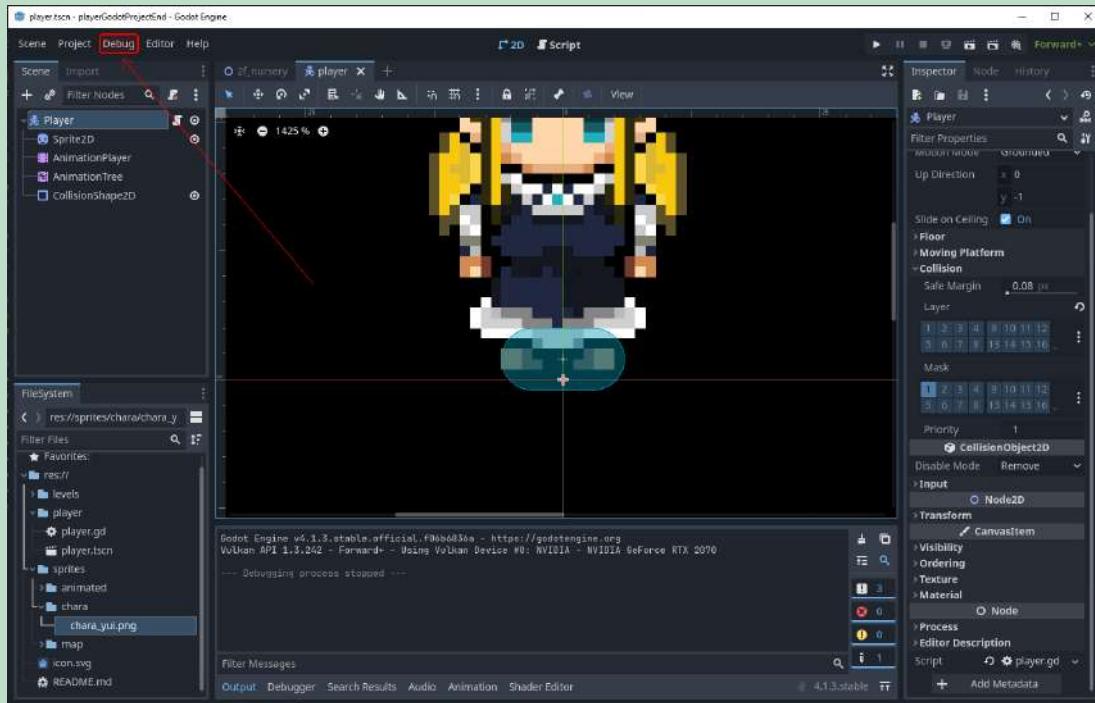
Il nostro player è ora in grado di collidere con gli oggetti del mondo



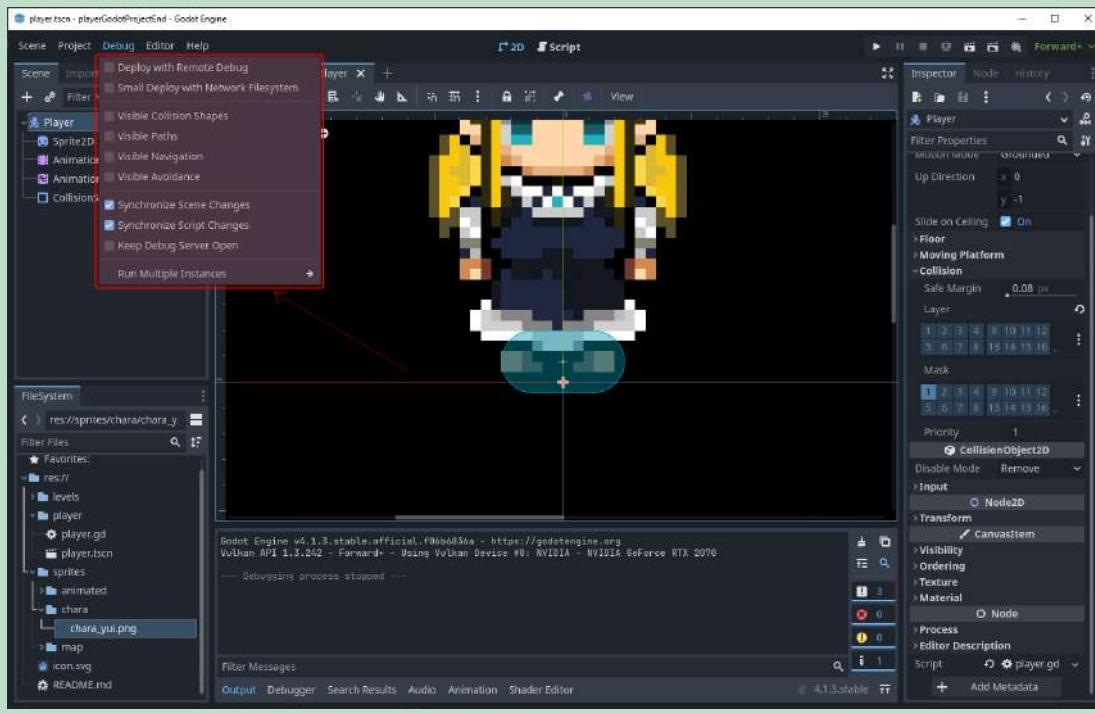
Sempre nella sezione "Terreni", vedremo inoltre come far sì che il player venga disegnato dietro alcuni oggetti, come ad esempio questa sedia della stanza

## ! Consiglio

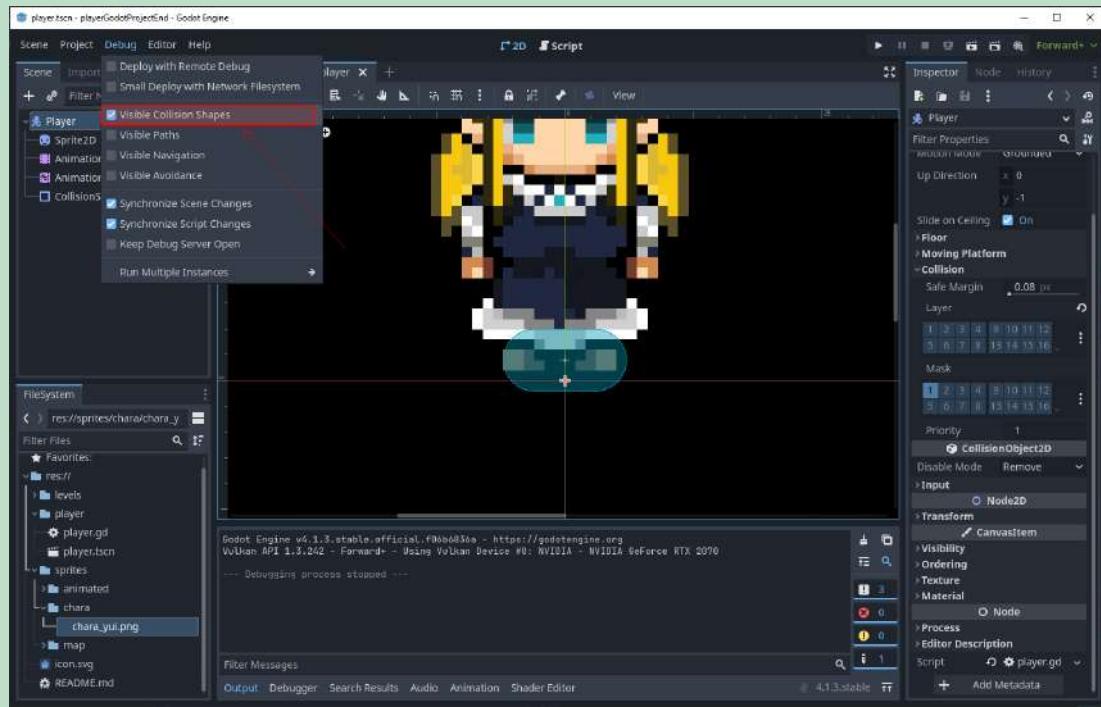
Durante il testing è molto utile visualizzare a schermo tutte le collision shape presenti in una scena, per farlo, clicchiamo sulla voce **Debug** in alto a sinistra



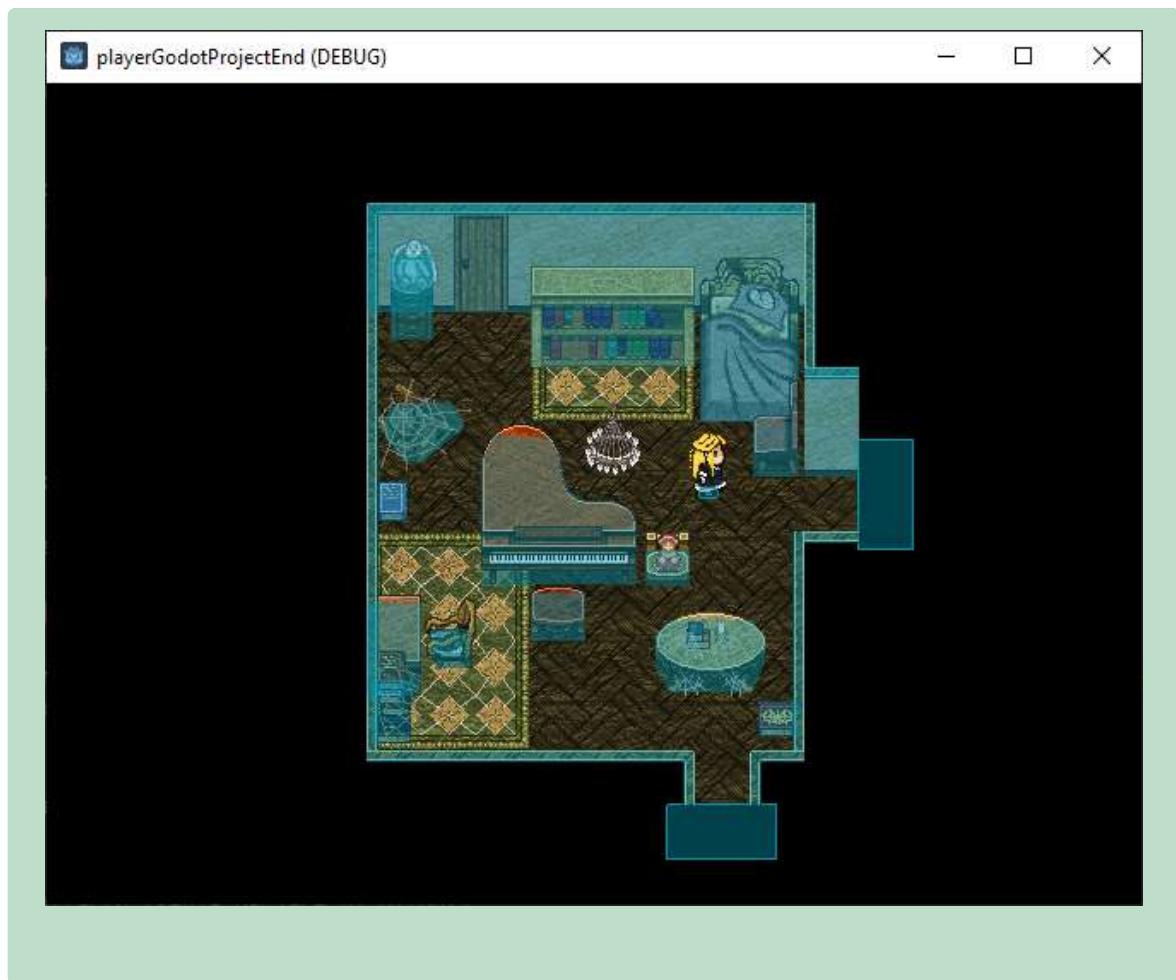
che farà comparire un apposito menu



in cui abilitiamo l'opzione **Visible Collision Shapes**

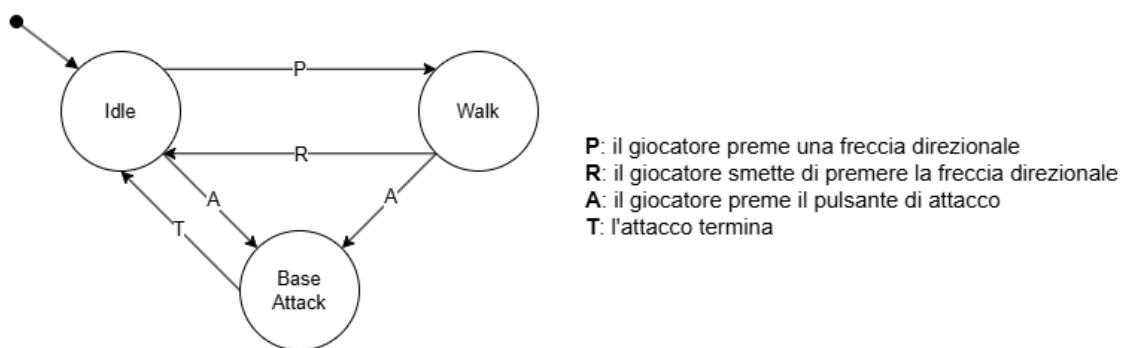


Con questa opzione abilitata, tutte le collision shape presenti nella scena vengono evidenziate in blu, anche quella del nostro nodo **Player**



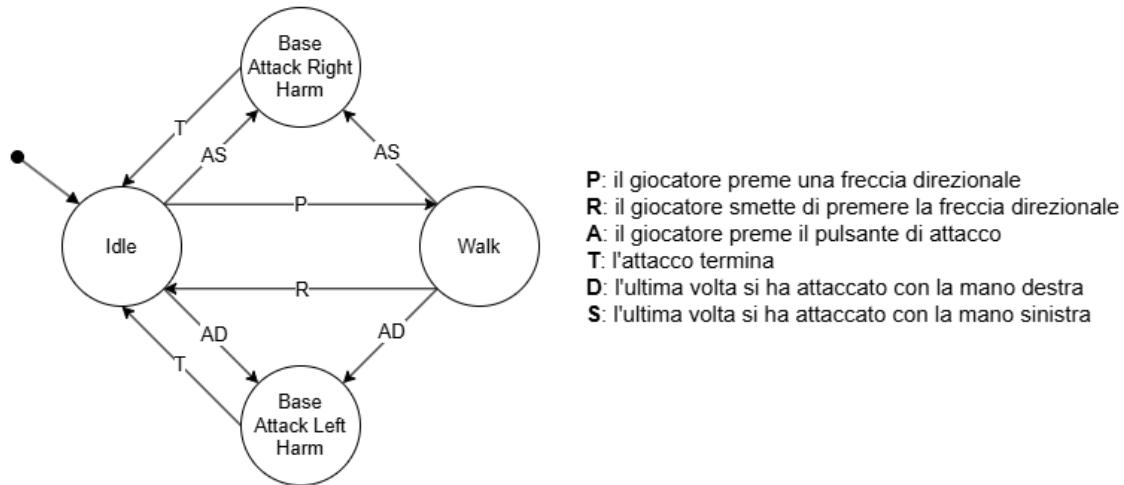
#### 2.4.6 Animazione d'attacco

Vogliamo adesso implementare un'animazione d'attacco per il nostro personaggio. Tornando al concetto di player come macchina a stati, allora l'animazione d'attacco non sarebbe nient'altro che un nuovo stato di questo automa a stati finiti



Lì dove lo stato **attack** non può transire a quello **walk** poiché vogliamo che, dopo aver attaccato, per poter andare eventualmente nello stato **walk**, il player deve prima tornare al nostro stato di partenza, ovvero, lo stato **idle**. Questo comportamento evita molti problemi legati alle animazioni, mentre si è in transizione tra i vari stati.

Nel caso del nostro personaggio però, l'animazione d'attacco che vogliamo implementare consiste nell'attaccare utilizzando o la mano sinistra o quella destra. Il nostro stato **attack** si traduce in ben due stati diversi



Applichiamo dunque questi cambiamenti al nostro codice di script. Per prima cosa, dobbiamo fare **code refactoring** (**refactoring di codice**), ovvero, migliorare la struttura del nostro codice senza modificarne il comportamento.

Il nuovo codice di script del **Player** è il seguente

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  @onready var animation_player = $AnimationPlayer # riferimento al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # riferimento al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # riferimento allo stato corrente
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'AnimationTree
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'AnimationTree
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'AnimationTree
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
17 const ANIMATION_TREE_PARAMETERS = [
18     "parameters/Idleblend_position",
19     "parameters/Walkblend_position"
20 ]
21
22 # definiamo un tipo enumerativo per gli stati del nostro player
23 enum States {
24     WALK,
25     BASE_ATTACK
26 }
27
28 # definiamo una variabile che tiene traccia dello stato corrente in cui si trova il nostro player
29 var state: States = States.WALK
30
31 func _ready():
32     animation_tree.active = true # abilitiamo il nodo AnimationTree
33
34 func _physics_process(_delta):
35     # eseguiamo un'azione specifica in base allo stato in cui si trova il nostro player
36     match state:
37         States.WALK: # il player sta camminando o è fermo
38             walk_state()
39         States.BASE_ATTACK: # il player sta attaccando con l'attacco base
40             pass
41
42 # funzione che gestisce le operazioni dello stato WALK del nostro player
43 func walk_state():
44     var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
45
46     var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale della direzione del giocatore
47
48     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
49     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
50     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
51
52     # normalizziamo il vettore per ottenere una direzione
53     input_vector = input_vector.normalized()
54     current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si muove
55
56     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
57         face_direction(current_player_direction) # facciamo guardare al player la direzione che deve avere
58         animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
59
60     else:
61         animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
62
63     velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, a
64
65     move_and_slide() # muoviamo il personaggio sullo schermo
66
67 func face_direction(direction: Vector2) -> void:
68     for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'AnimationTree
69         animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato dell'AnimationTree
70

```

Le righe che vanno dalla 23 alla 26 riguardano la definizione di un tipo `enum` (già descritto in precedenza) che rappresenta tutti gli stati della nostra macchina a stati finiti del player.

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  @onready var animation_player = $AnimationPlayer # reference al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # reference al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # reference allo stato corrente
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'AnimationTree
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'AnimationTree
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'AnimationTree
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
17 const ANIMATION_TREE_PARAMETERS = [
18   "parameters/Idleblend_position",
19   "parameters/Walkblend_position"
20 ]
21
22 # definiamo un tipo enumerativo per gli stati del nostro player
23 enum States {
24   WALK,
25   BASE_ATTACK
26 }
27
28 # definiamo una variabile che tiene traccia dello stato corrente in cui si trova il nostro player
29 var state: States = States.WALK
30

```

A riga 29 troviamo invece la dichiarazione di una variabile `State`, la quale è di tipo enum `States`, e la sua inizializzazione a `States.WALK`. Questo perché, nel nostro codice, gli stati `idle` (stato di partenza) e `walk` della macchina a stati finiti del player coincidono: "stare fermi" può infatti essere visto come "muoversi con velocità 0". In questa riga di codice stiamo dunque definendo lo stato iniziale (o stato di partenza) del nostro player.

```

1  class_name Player # Definisce lo script come una classe globalmente accessibile con il nome 'Player'
2  extends CharacterBody2D
3
4  const MAX_DEFAULT_SPEED = 150 # definiamo una costante che rappresenta la nostra velocità massima
5
6  @export var SPEED = MAX_DEFAULT_SPEED # definiamo una variabile per impostare la velocità del personaggio
7
8  @onready var animation_player = $AnimationPlayer # reference al nodo AnimationPlayer
9  @onready var animation_tree = $AnimationTree # reference al nodo AnimationTree
10 @onready var animation_state = animation_tree.get("parameters/playback") # reference allo stato corrente
11
12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'AnimationTree
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'AnimationTree
14
15 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'AnimationTree
16 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
17 const ANIMATION_TREE_PARAMETERS = [
18   "parameters/Idleblend_position",
19   "parameters/Walkblend_position"
20 ]
21
22 # definiamo un tipo enumerativo per gli stati del nostro player
23 enum States {
24   WALK,
25   BASE_ATTACK
26 }
27
28 # definiamo una variabile che tiene traccia dello stato corrente in cui si trova il nostro player
29 var state: States = States.WALK
30

```

Il vero refactoring è la funzione `_physics_process(delta)`, la quale ora presenta al suo interno uno **switch statement** da riga 36 a riga 40

```

28 # definiamo una variabile che tiene traccia dello stato corrente in cui si trova il nostro player
29 var state: States = States.WALK
30
31 func _ready():
32   animation_tree.active = true # abilitiamo il nodo AnimationTree
33
34 func _physics_process(_delta):
35   # eseguiamo un'azione specifica in base allo stato in cui si trova il nostro player
36   match state:
37     States.WALK: # il player sta camminando o è fermo
38       walk_state()
39     States.BASE_ATTACK: # il player sta attaccando con l'attacco base
40       pass
41
42 # funzione che gestisce le operazioni dello stato WALK del nostro player
43 func walk_state():
44   var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
45
46   var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale della direzione
47
48   # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
49   input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
50   input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
51   # normalizziamo il vettore per ottenere una direzione
52   input_vector = input_vector.normalized()
53   current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si muove
54

```

Uno **switch statement** controlla se un'espressione (nel nostro caso, il valore della variabile `state`) coincide con uno dei pattern forniti (nell'esempio, `States.WALK` o `States.BASE_ATTACK`), eseguendo solo e soltanto il blocco di codice del primo pattern che fa match.

In poche parole, prendendo come esempio il nostro codice, se il valore della variabile `state`:

- è uguale a quello di `States.WALK` (ovvero `WALK`), viene eseguita riga [38](#)
- è uguale a quello di `States.BASE_ATTACK` (ovvero `WALK`), viene eseguita riga [40](#)
- non è uguale né a quello di `States.WALK` né a quello di `States.BASE_ATTACK`, si continua l'esecuzione a partire da riga [41](#). Si noti che questo caso è impossibile poiché la variabile `state` viene inizializzata a riga [29](#) e, dunque, nel caso peggiore, il valore contenuto nella variabile `state` è sicuramente almeno uguale a quello di `States.WALK`

Infine, la funzione `walk_state()`, invocata a riga [38](#) e definita da riga [43](#) a riga [65](#), contiene semplicemente tutto il codice che abbiamo visto fino a questo momento.

```

43 func walk_state():
44     var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore di movimento
45
46     var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale della direzione del personaggio
47
48     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
49     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
50     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
51     # normalizziamo il vettore per ottenere una direzione
52     input_vector = input_vector.normalized()
53     current_player_direction = input_vector # il player sta guardando la stessa direzione in cui si muove
54
55     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo il personaggio
56         face_direction(current_player_direction) # facciamo guardare al player la direzione che deve guardare
57         animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
58
59         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effettivamente muovendo il personaggio
60     else:
61         animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"
62
63         velocity = Vector2.ZERO # se il giocatore NON sta effettivamente muovendo il personaggio, a velocità zero
64
65     move_and_slide() # muoviamo il personaggio sullo schermo

```

Adesso che abbiamo fatto refactoring del codice per "fargli assumere le sembianze" di una vera macchina a stati finiti, possiamo concentrarci sull'implementare l'animazione d'attacco. Creiamo dunque, nel modo che abbiamo visto in precedenza, le seguenti animazioni:

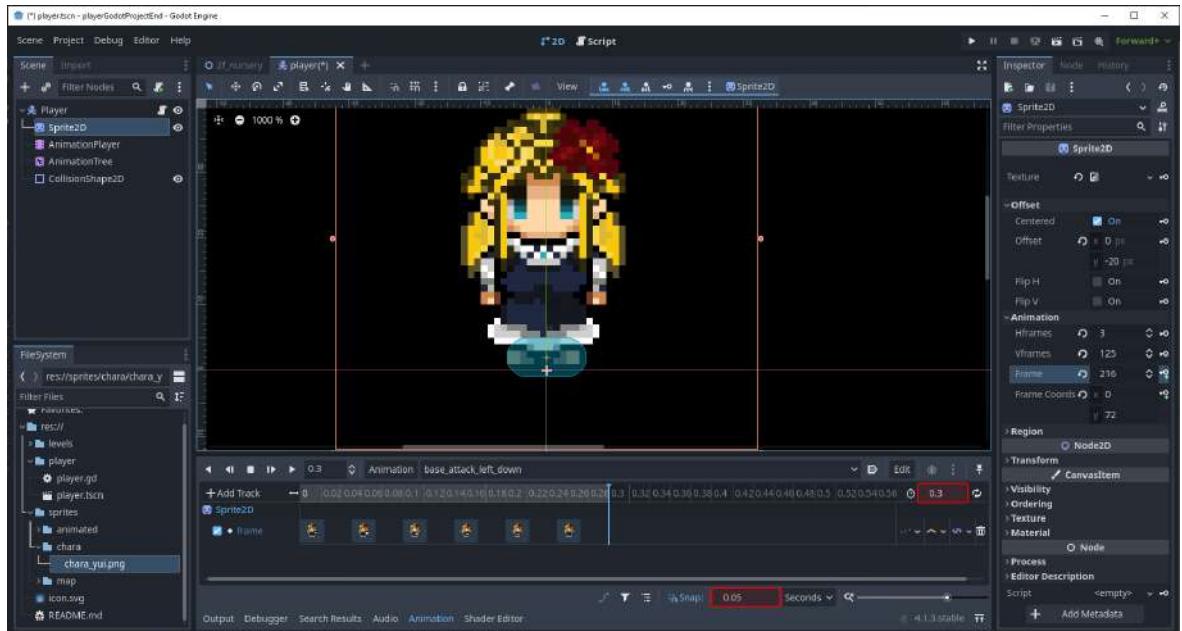
- `base_attack_left_down` con i frame che vanno da [211](#) a [216](#) (estremi inclusi)
- `base_attack_left_left` con i frame che vanno da [223](#) a [228](#) (estremi inclusi)
- `base_attack_left_right` con i frame che vanno da [241](#) a [246](#) (estremi inclusi)

- `base_attack_left_up` con i frame che vanno da 253 a 258 (estremi inclusi)
- `base_attack_right_down` con i frame che vanno da 217 a 222 (estremi inclusi)
- `base_attack_right_left` con i frame che vanno da 229 a 234 (estremi inclusi)
- `base_attack_right_right` con i frame che vanno da 235 a 240 (estremi inclusi)
- `base_attack_right_up` con i frame che vanno da 247 a 252 (estremi inclusi)

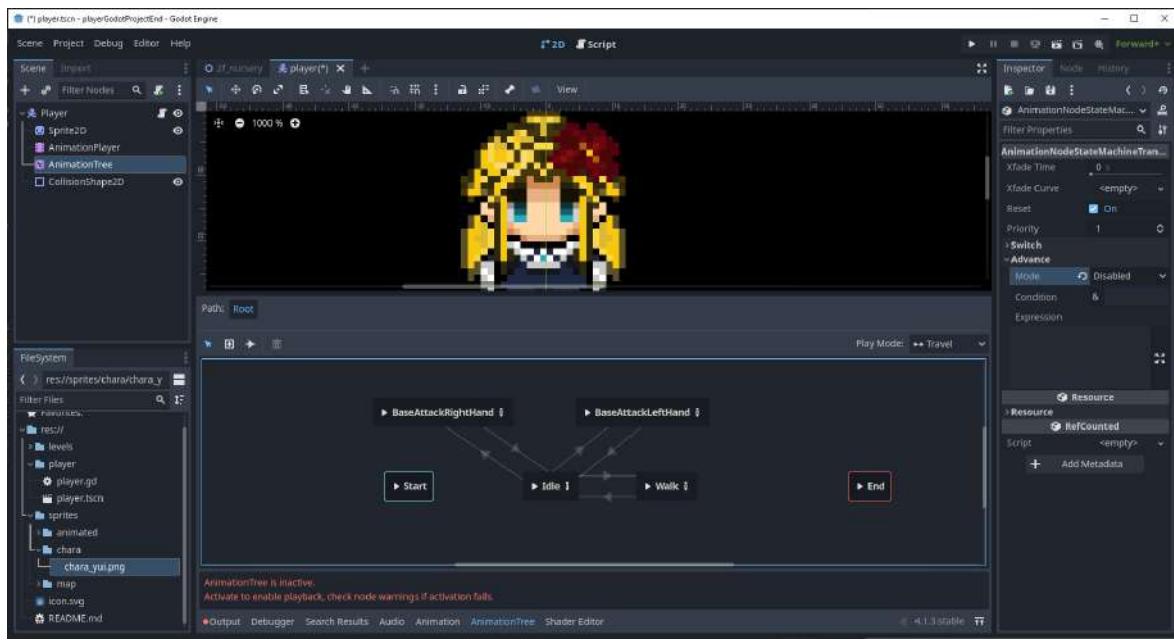
tutte senza loop, una lunghezza di animazione di 0.3 secondi e snap di 0.05 secondi.

### ! Consiglio

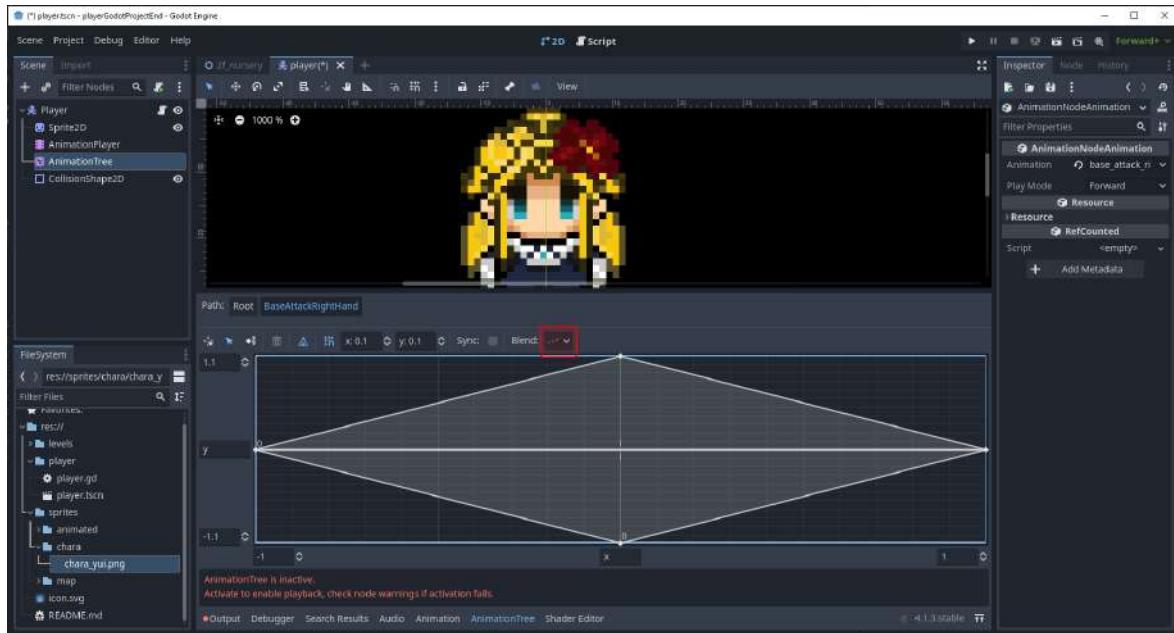
Ricordiamoci di cancellare sempre e immediatamente l'animazione `RESET`, auto-generata da Godot ogni volta stiamo aggiungendo un nuovo frame all'interno di un'animazione vuota.



creiamo dunque gli stati `BaseAttackRightHand` e `BaseAttackLeftHand` e li collegiamo con lo stato `Idle` in entrambe le direzioni. Scegliamo di non collegarli con lo stato `Walk` per fare in modo che, anche mentre si sta camminando, l'animazione di attacco parta nel momento in cui la velocità del player è pari a 0.

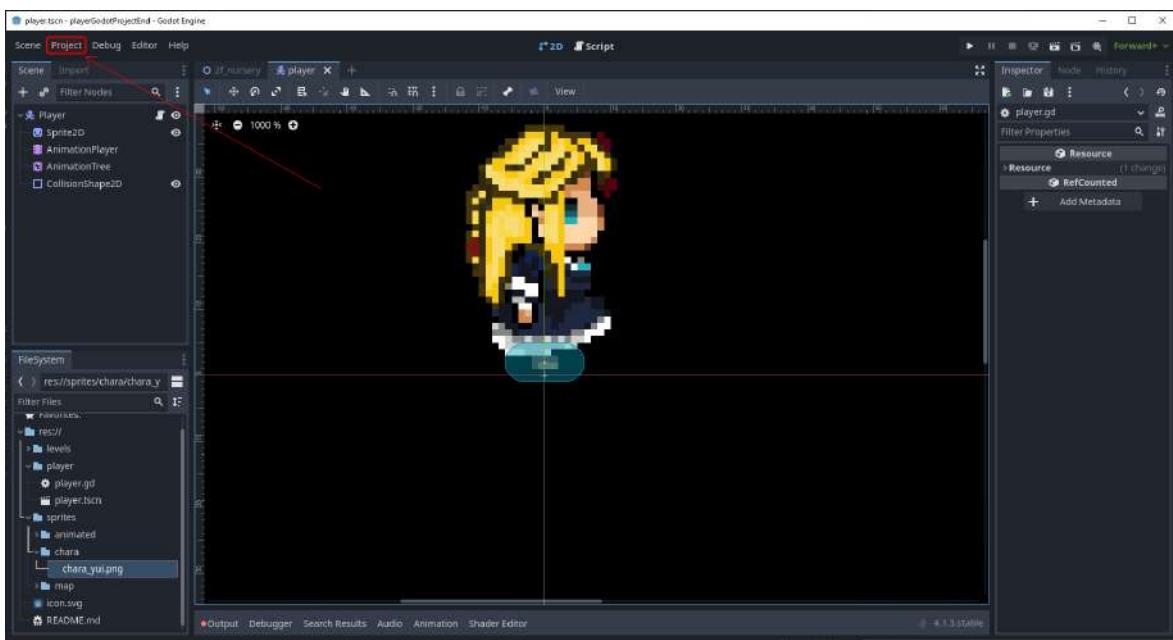


Ricordiamoci, inoltre, di impostare la blend mode su **Discrete**

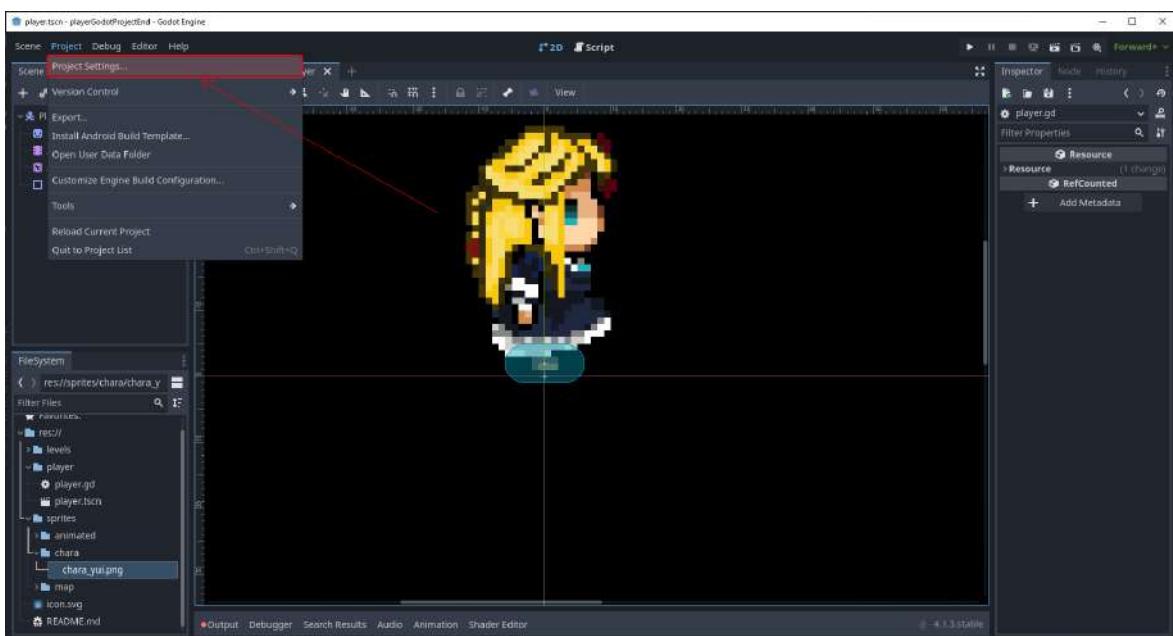


A questo punto, scegliamo un tasto per attaccare, nel nostro caso **c**. Per impostarlo, clicchiamo sulla voce **Project**

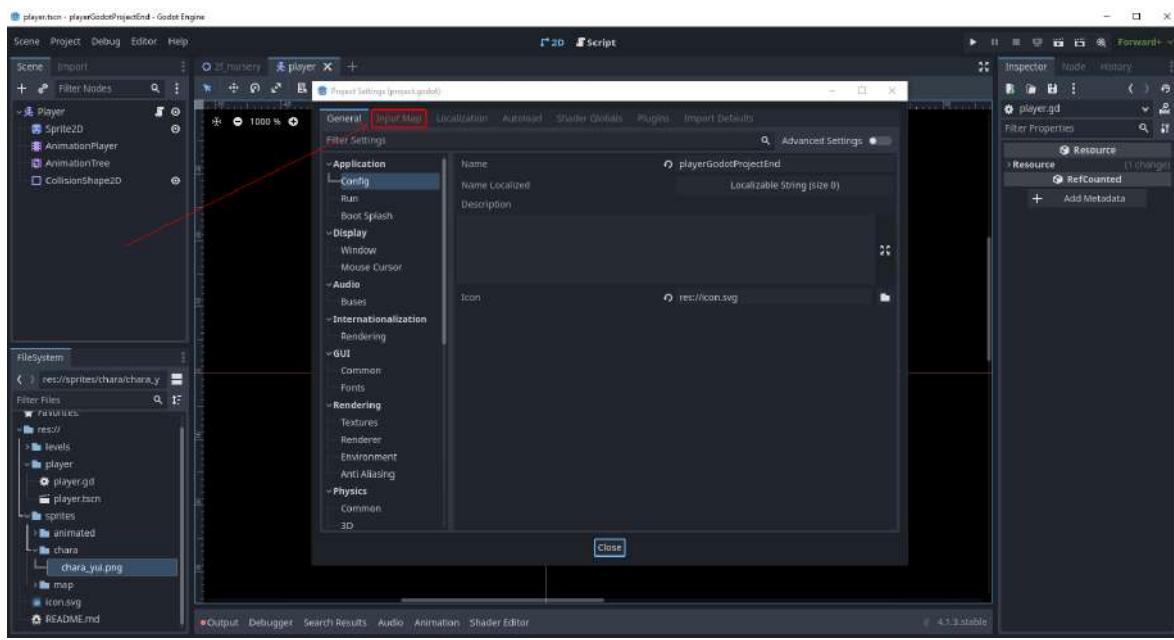
## CHAPTER 2. GODOT



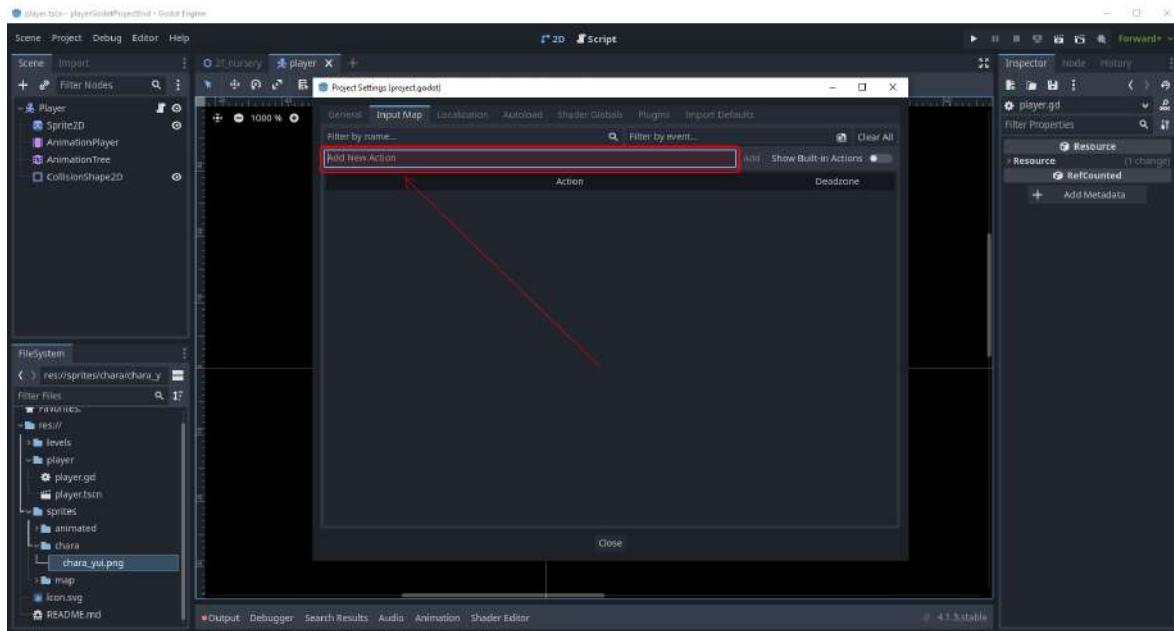
clicchiamo su **Project Settings...**



clicchiamo su **Input Map** nella nuova finestra che si aprirà

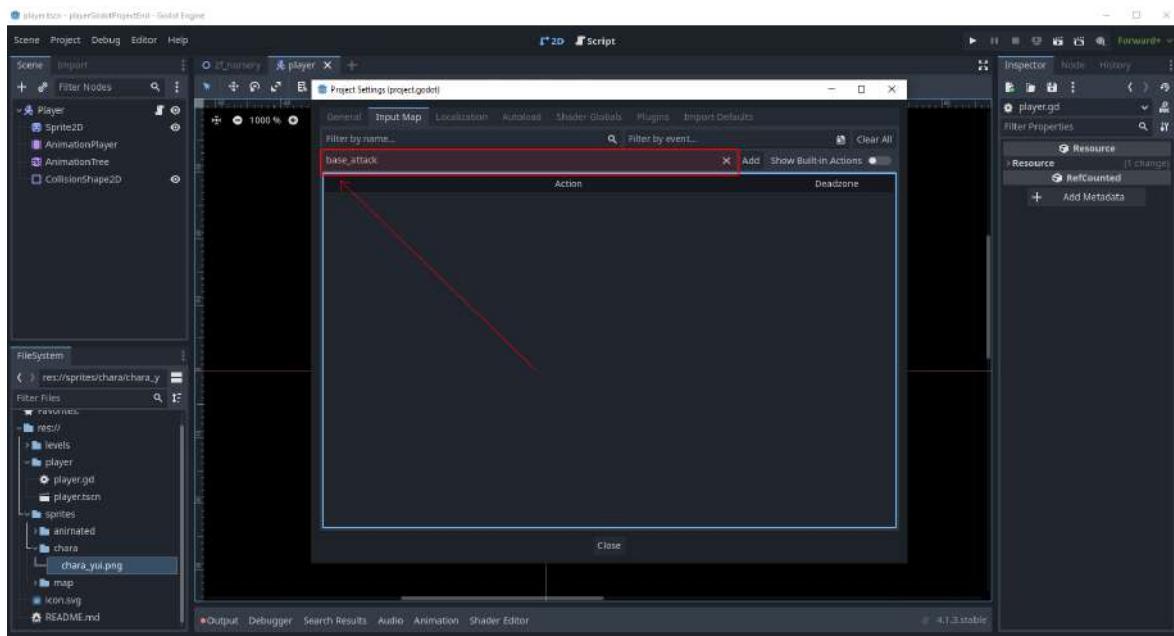


clicchiamo su **Add New Action**

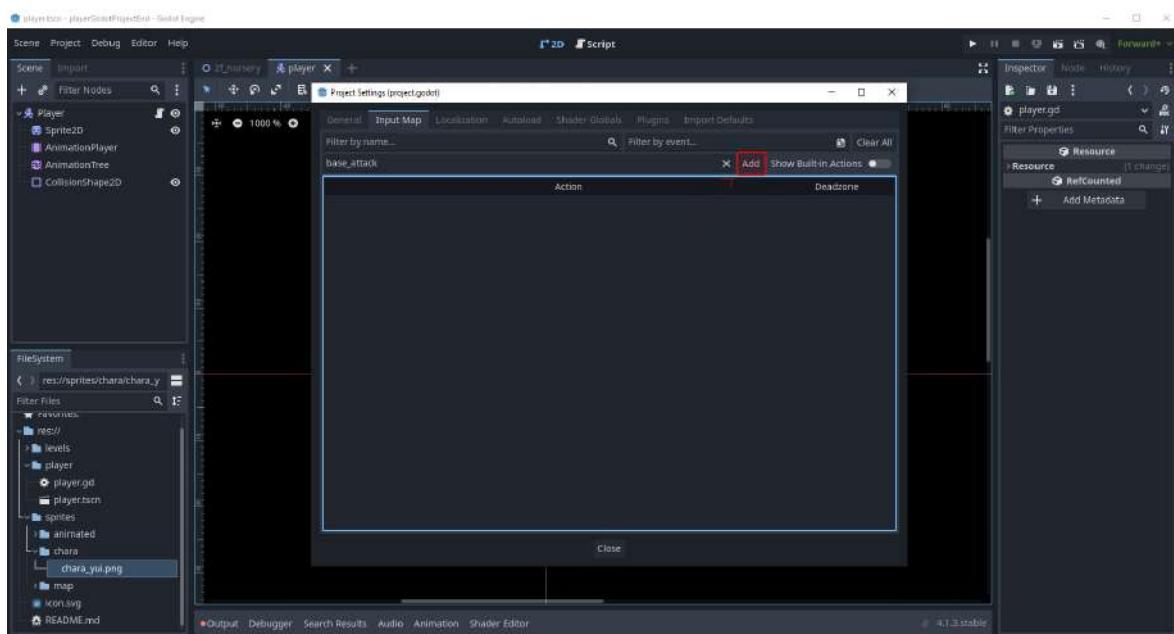


inseriamo il nome **base\_attack**

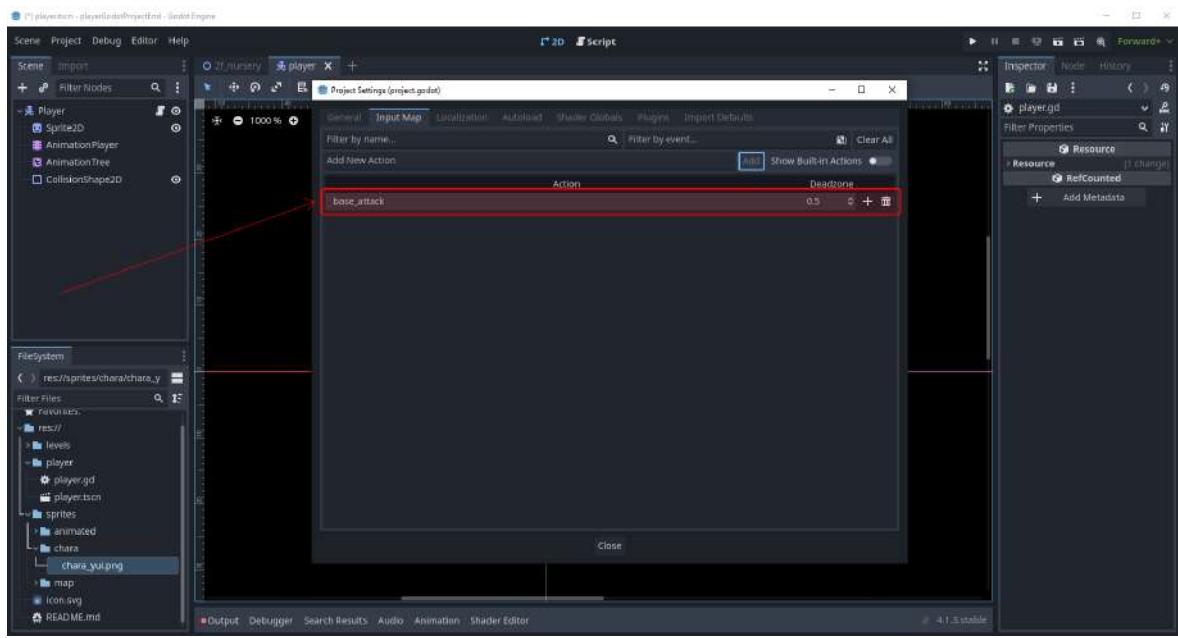
## CHAPTER 2. GODOT



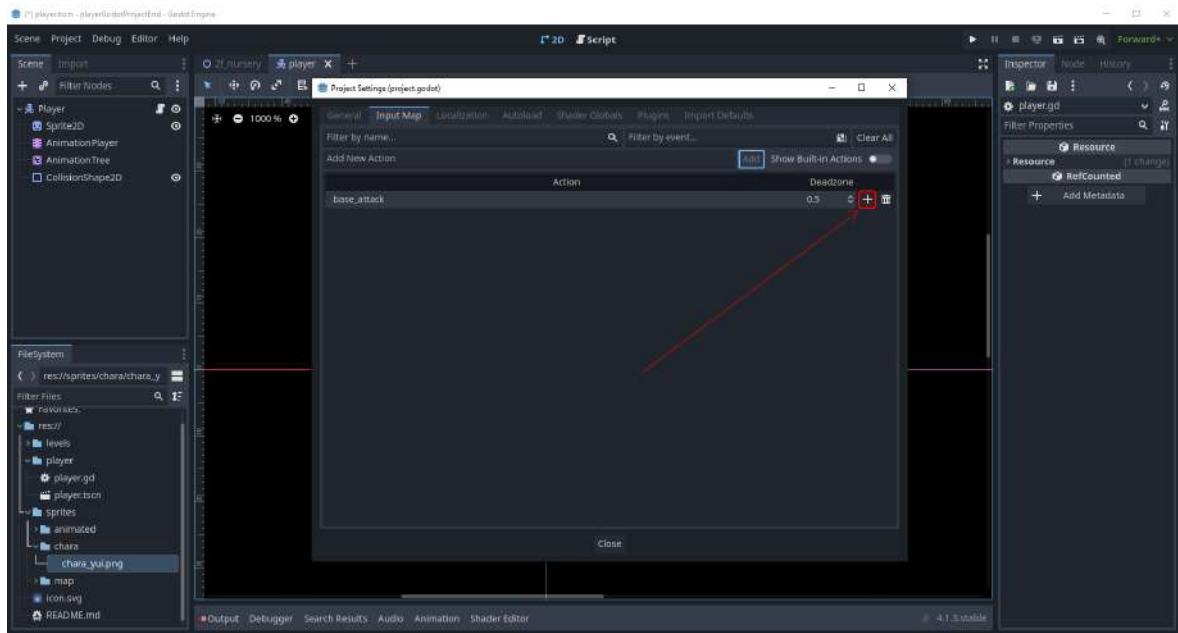
clicchiamo su **Add**



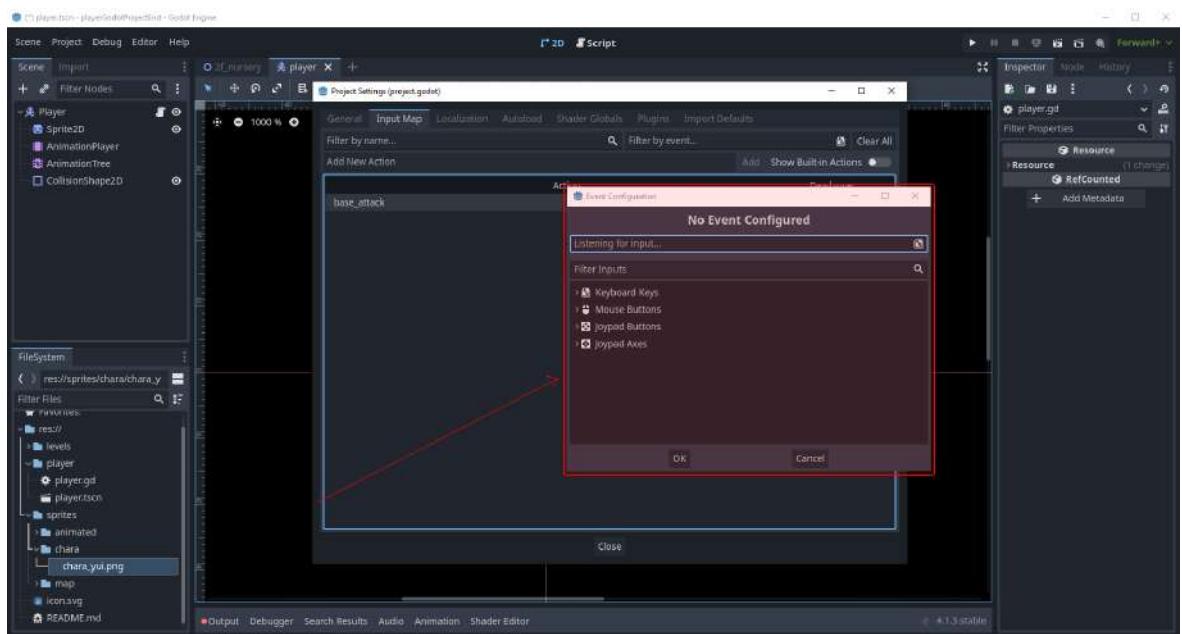
si creerà un nuovo evento di nome **base\_attack**



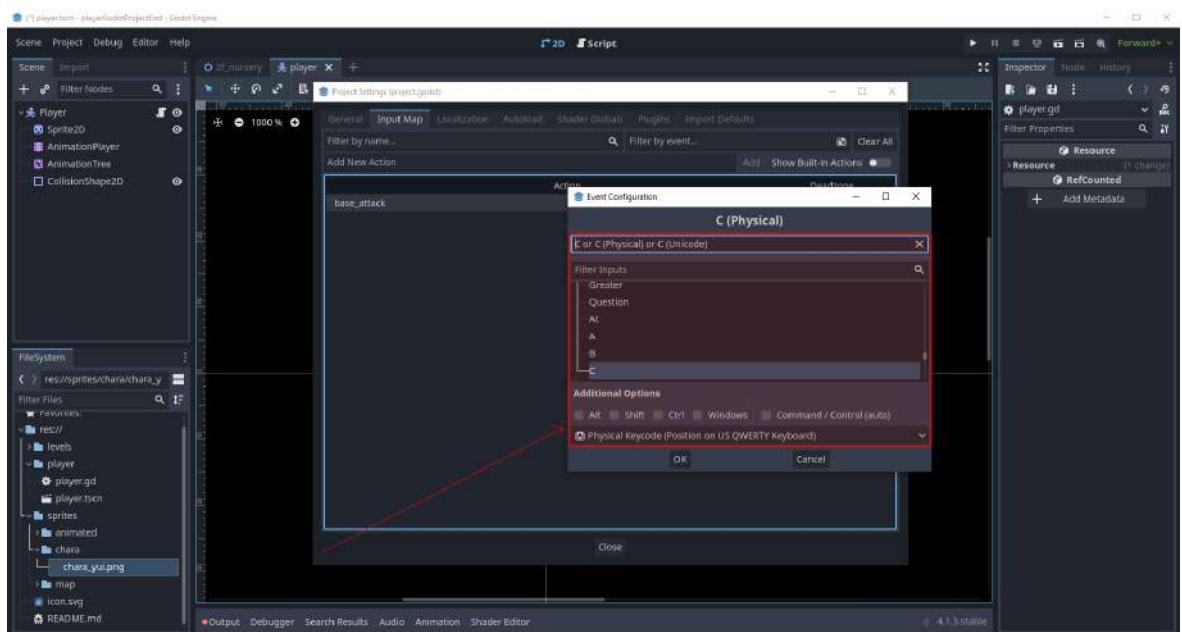
clicchiamo sul



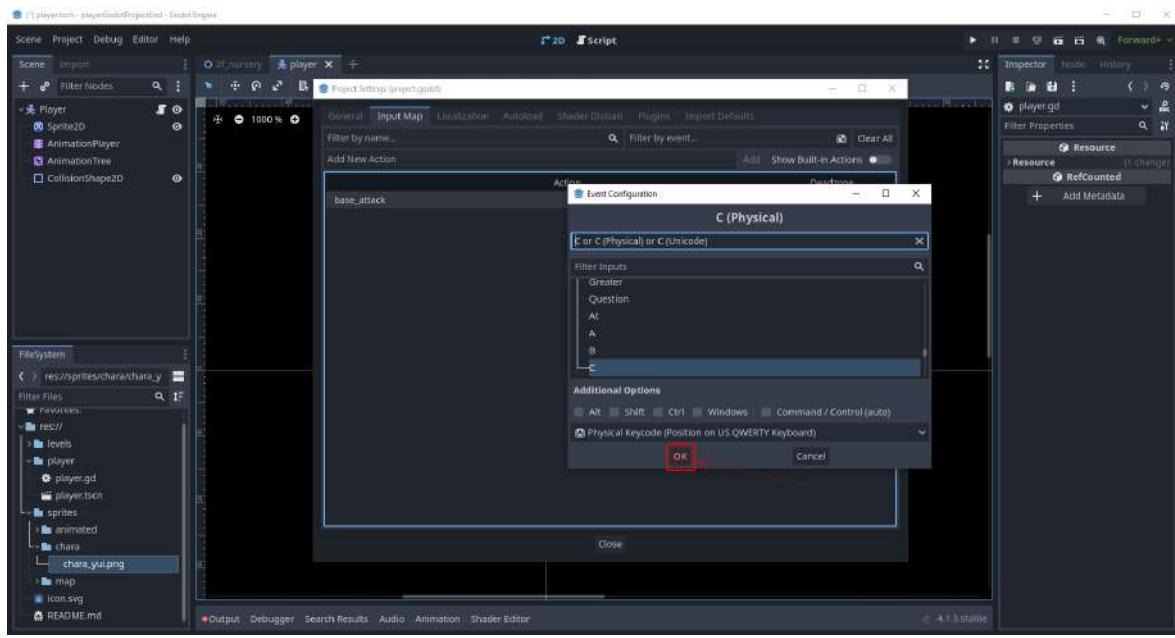
si aprirà una nuova finestra



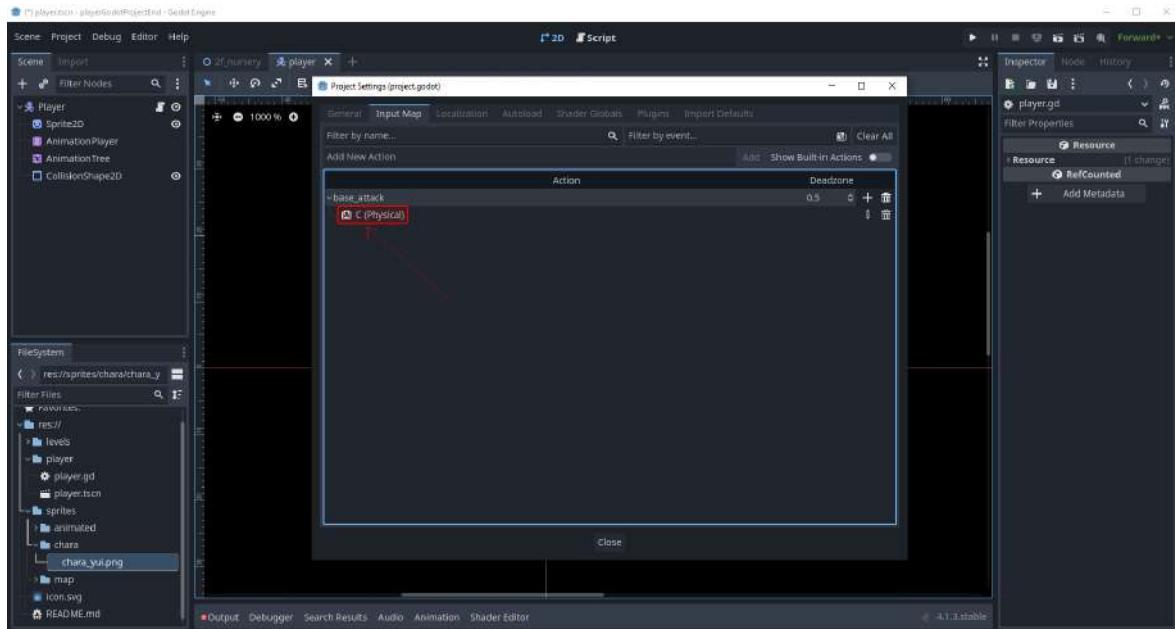
premiamo fisicamente il tasto **C** sulla tastiera per far apparire la seguente schermata



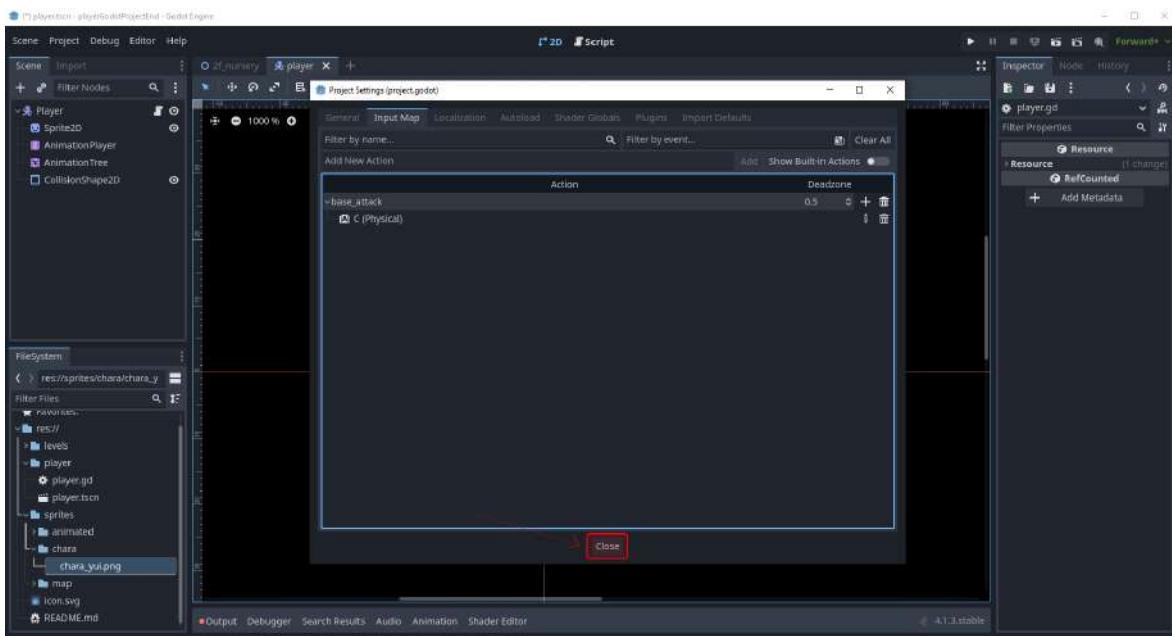
premiamo **Ok**



comparirà il tasto **C** sotto la voce **base\_attack**



premiamo dunque **Close** per chiudere la finestra



Dal punto di vista del codice, lo script che necessitiamo per eseguire l'animazione di attacco è il seguente

```

12 const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'AnimationTree
13 const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'AnimationTree
14 const BASE_ATTACK_LEFT_HAND_ANIMATION_KEY = "BaseAttackLeftHand" # chiave con cui abbiamo chiamato lo stato di base attack con la mano sinistra
15 const BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY = "BaseAttackRightHand" # chiave con cui abbiamo chiamato lo stato di base attack con la mano destra
16
17 var is_was_last_base_attack_done_with_righthand = false # memorizziamo se l'ultimo attacco è stato fatto con la mano destra
18
19 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'AnimationTree
20 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
21 const ANIMATION_TREE_PARAMETERS = [
22     "parameters/Idleblend_position",
23     "parameters/Walkblend_position",
24     "parameters/BaseAttackLeftHandblend_position",
25     "parameters/BaseAttackRightHandblend_position"
26 ]
27
28 # definiamo un tipo enumerativo per gli stati del nostro player
29 enum States {
30     WALK,
31     BASE_ATTACK
32 }
33 move_and_slide() # muoviamo il personaggio sullo schermo
34
35 if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato il tasto per l'attacco
36     state = States.BASE_ATTACK # cambiamo stato
37     base_attack()
38
39 func face_direction(direction: Vector2) -> void:
40     for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'AnimationTree
41         animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato dell'AnimationTree nel nuovo direzione
42
43 # gestisce il comportamento del player per l'attacco base
44 func base_attack():
45     velocity = Vector2.ZERO # fermiamo il player
46
47     # controlliamo con che mano ha attaccato l'ultima volta
48     if is_was_last_base_attack_done_with_righthand:
49         animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animazione di attacco con la mano sinistra
50         is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima volta abbiam attaccato con la mano sinistra
51     else:
52         animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animazione di attacco con la mano destra
53         is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima volta abbiam attaccato con la mano destra
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92

```

lì dove a riga 14 e 15 dichiariamo le costanti per le chiavi degli stati dell'animazione di attacco, così come abbiamo fatto per gli stati `Idle` e `Walk` in precedenza

```
11
12  const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nel
13  const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nel
14  const BASE_ATTACK_LEFT_HAND_ANIMATION_KEY = "BaseAttackLeftHand" # chiave con cui abbiamo
15  const BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY = "BaseAttackRightHand" # chiave con cui abbiamo
16
17  var is_was_last_base_attack_done_with_righthand = false # memorizziamo se l'ultimo attac
18
19  # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'Animat
20  # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
21  const ANIMATION_TREE_PARAMETERS = [
22    "parameters/Idleblend_position",
23    "parameters/Walkblend_position",
24    "parameters/BaseAttackLeftHandblend_position",
25    "parameters/BaseAttackRightHandblend_position"
26  ]
27
28  # definiamo un tipo enumerativo per gli stati del nostro player
29  enum States {
30    WALK,
31    BASE_ATTACK
32 }
33
```

a riga 17 dichiariamo la variabile `is_was_last_base_attack_done_with_righthand` che ci serve per capire se, per lanciare l'ultimo attacco, il player ha utilizzato la mano destra; e la inizializziamo a `false`. Così facendo, il primo attacco del player sarà sempre eseguito con la mano destra (poiché è come se l'attacco "zero" fosse stato eseguito con la mano sinistra)

```

11
12  const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nel
13  const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nel
14  const BASE_ATTACK_LEFT_HAND_ANIMATION_KEY = "BaseAttackLeftHand" # chiave con cui abbiamo
15  const BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY = "BaseAttackRightHand" # chiave con cui abbiamo
16
17  var is_was_last_base_attack_done_with_rigth_hand = false # memorizziamo se l'ultimo attacco
18
19  # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'Animazione
20  # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
21  const ANIMATION_TREE_PARAMETERS = [
22    "parameters/Idleblend_position",
23    "parameters/Walkblend_position",
24    "parameters/BaseAttackLeftHandblend_position",
25    "parameters/BaseAttackRightHandblend_position"
26  ]
27
28  # definiamo un tipo enumerativo per gli stati del nostro player
29  enum States {
30    WALK,
31    BASE_ATTACK
32  }
33

```

a riga 24 e 25 aggiungiamo i crosshair degli stati di attacco all'array delle blend position **ANIMATION\_TREE\_PARAMETERS**

```

11
12  const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nel
13  const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nel
14  const BASE_ATTACK_LEFT_HAND_ANIMATION_KEY = "BaseAttackLeftHand" # chiave con cui abbiamo
15  const BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY = "BaseAttackRightHand" # chiave con cui abbiamo
16
17  var is_was_last_base_attack_done_with_rigth_hand = false # memorizziamo se l'ultimo attacco
18
19  # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'Animazione
20  # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
21  const ANIMATION_TREE_PARAMETERS = [
22    "parameters/Idleblend_position",
23    "parameters/Walkblend_position",
24    "parameters/BaseAttackLeftHandblend_position",
25    "parameters/BaseAttackRightHandblend_position"
26  ]
27
28  # definiamo un tipo enumerativo per gli stati del nostro player
29  enum States {
30    WALK,
31    BASE_ATTACK
32  }
33

```

A riga 73 facciamo uso del fatto di aver legato il tasto **C** con il valore **base\_attack**, controllando dunque se il player ha premuto tale tasto oppure no. Il tutto dentro la funzione **walk\_state()**

```
71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_righthand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima vo
92     |
```

la funzione **is\_action\_just\_pressed(action, exact\_match)** del singleton **Input**, ritorna **true** se l'utente ha *iniziato* a premere il tasto collegato (nel nostro caso il tasto **C**) all'**action event** fornita (nel nostro caso **base\_attack**). Utilizziamo questa funzione per fare in modo che il codice venga eseguito solamente una volta quando il tasto viene premuto, invece che ogni frame mentre il tasto continua ad essere premuto.

A riga 74 transiamo dallo stato **WALK** allo stato **BASE\_ATTACK**

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # cicliamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_righthand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima v
92

```

A riga 75 invochiamo la funzione `base_attack()`

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # cicliamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_righthand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima v
92

```

definita nelle righe che vanno dalla 82 alla 91

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_rigth_hand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_rigth_hand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima vo
92

```

All'interno del corpo della funzione `base_attack()`, a riga 83, impostiamo a (0,0) la variabile `velocity` del player in modo da farlo fermare

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_rigth_hand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_rigth_hand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima vo
92

```

con l'if statement dalla riga 86 alla 91 controlliamo con quale mano il player abbia attaccato l'ultima volta

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # cicliamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_righthand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima vo
92

```

Se la mano è la destra, eseguiamo le righe 87 e 88, lì dove, rispettivamente, a riga 87 transiamo dallo stato **Idle** allo stato **BaseAttackLeftHand** dell'**AnimationTree** per riprodurre l'animazione di attacco con la mano sinistra; mentre a riga 88 impostiamo a **false** la variabile **is\_was\_last\_base\_attack\_done\_with\_righthand**, in modo da ricordarci che, l'ultima volta, il personaggio ha attaccato con la mano sinistra.

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_rigth_hand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_rigth_hand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima vo
92

```

Il discorso inverso vale per le righe 90 e 91.

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_rigth_hand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_rigth_hand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima vo
92

```

Avviando il gioco e premeto **C**, il player attacca correttamente nella direzione che

sta guardando



Notiamo però che c'è un problema: dopo aver premuto **C** una volta, il player non cammina né attacca più. Questo perché per attaccare transiamo allo stato **BASE\_ATTACK**

```

71     move_and_slide() # muoviamo il personaggio sullo schermo
72
73     if Input.is_action_just_pressed("base_attack"): # controlliamo se l'utente ha cliccato
74         state = States.BASE_ATTACK # cambiamo stato
75         base_attack()
76
77     func face_direction(direction: Vector2) -> void:
78         for parameter in ANIMATION_TREE_PARAMETERS: # ciclichiamo ogni crosshair degli stati dell'
79             animation_tree.set(parameter, direction) # posizioniamo il crosshair di ogni stato
80
81     # gestisce il comportamento del player per l'attacco base
82     func base_attack():
83         velocity = Vector2.ZERO # fermiamo il player
84
85         # controlliamo con che mano ha attaccato l'ultima volta
86         if is_was_last_base_attack_done_with_rigth_hand:
87             animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
88             is_was_last_base_attack_done_with_rigth_hand = false # ci ricordiamo che l'ultima v
89         else:
90             animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animaz
91             is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima vo
92

```

e dunque la funzione `_physics_process(delta)` esegue, ogni volta che viene chiamata, solo e soltanto riga 46, ovvero nulla

```

36
37     func _ready():
38         animation_tree.active = true # abilitiamo il nodo AnimationTree
39
40     func _physics_process(_delta):
41         # eseguiamo un'azione specifica in base allo stato in cui si trova il nostro player
42         match state:
43             States.WALK: # il player sta camminando o è fermo
44                 walk_state()
45             States.BASE_ATTACK: # il player sta attaccando con l'attacco base
46                 pass
47
48         # funzione che gestisce le operazioni dello stato WALK del nostro player
49     func walk_state():
50         var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro ve
51
52         var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore i
53
54         # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
55         input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
56         input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
57
58         # normalizziamo il vettore per ottenere una direzione
59         input_vector = input_vector.normalized()

```

Una volta aver terminato l'animazione di attacco dobbiamo tornare allo stato **Idle**, così come vediamo anche dal diagramma della macchina a stati finiti del player.

Per fare questo, prima di tutto definiamo la funzione da chiamare al termine dell'animazione di attacco: **base\_attack\_animation\_finished()**

```

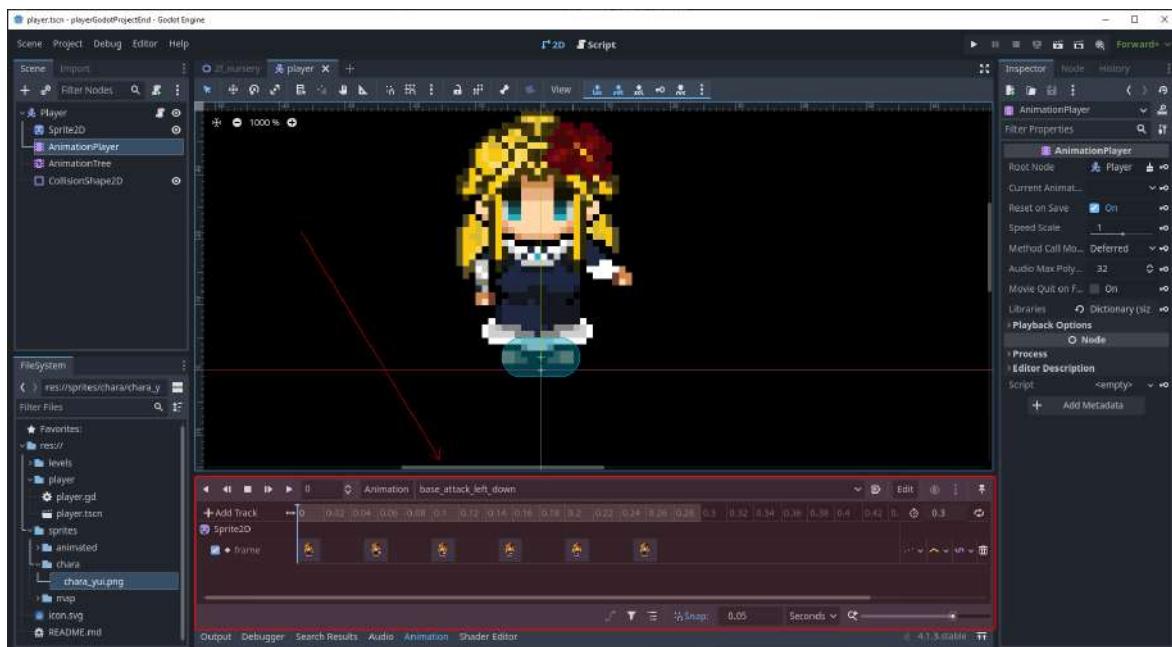
81  # gestisce il comportamento del player per l'attacco base
82  func base_attack():
83      >> velocity = Vector2.ZERO # fermiamo il player
84
85      >> # controlliamo con che mano ha attaccato l'ultima volta
86      >> if is_was_last_base_attack_done_with_righthand:
87          >> animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animazione di at-
88          >> is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima volta abbiam
89      >> else:
90          >> animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animazione di a-
91          >> is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima volta abbiam
92
93  func base_attack_animation_finished():
94      state = States.WALK
95

```

Come possiamo vedere, tutto quello che fa **base\_attack\_animation\_finished()** è transire dallo stato **BASE ATTACK** allo stato **WALK**

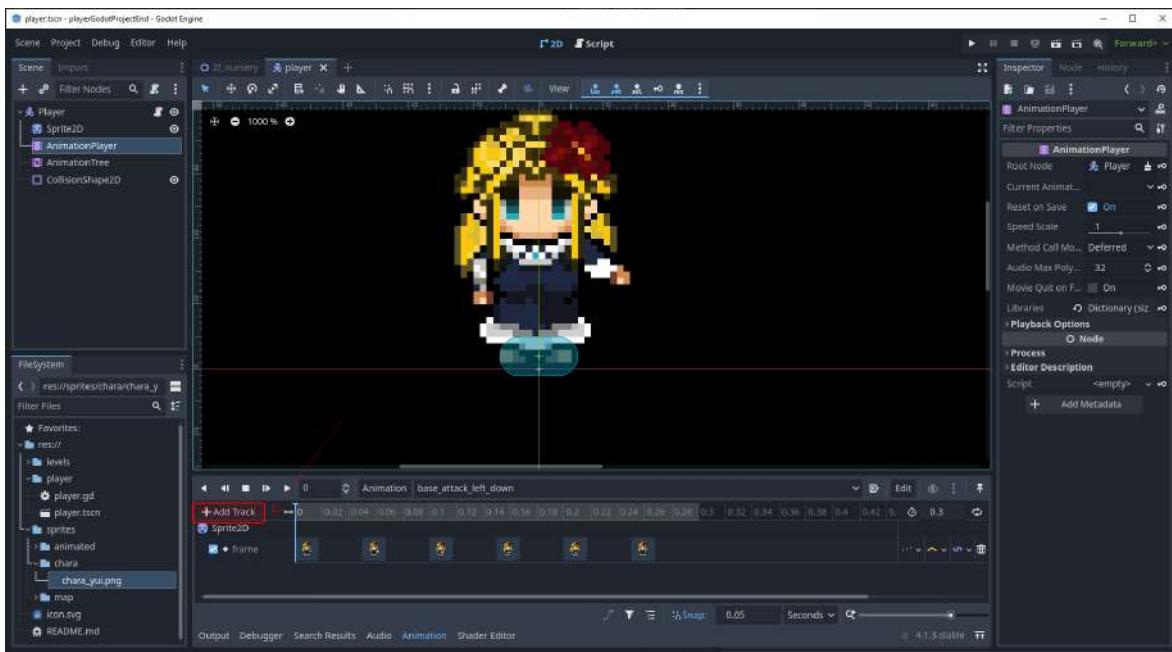
Ma come chiamare questa funzione proprio quando l'animazione di attacco è terminata? Vi sono più modi per farlo. Quello che abbiamo scelto in questa occasione riguarda il nodo **AnimationPlayer**.

Apriamo dunque una qualsiasi animazione di attacco nel panel **Animation**

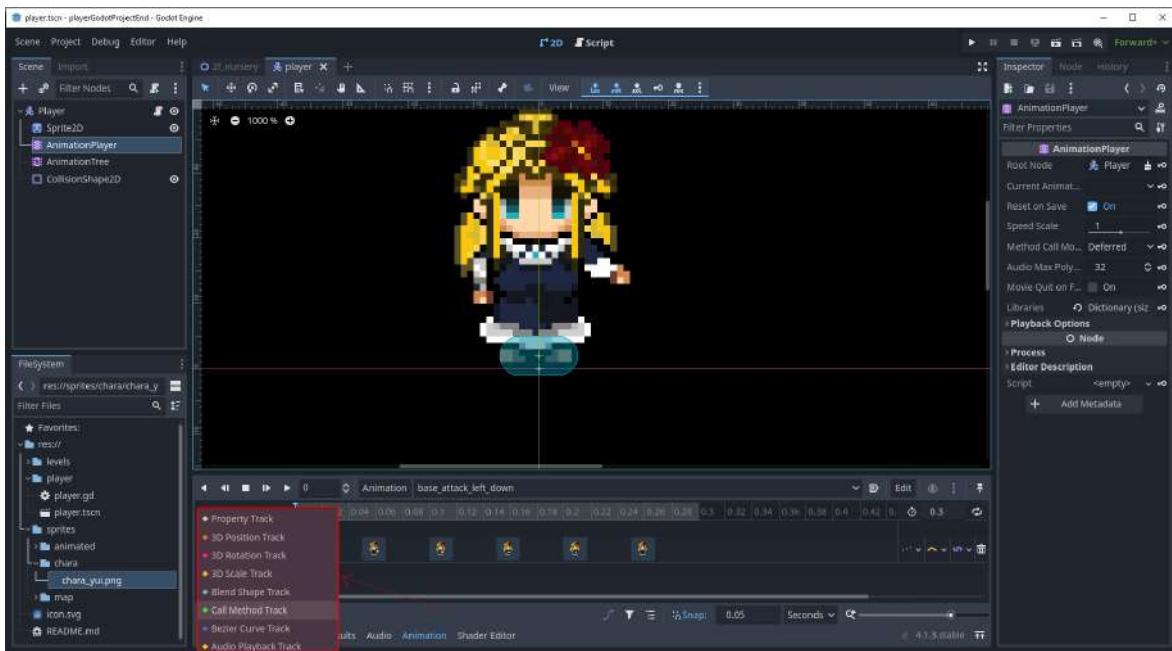


clicchiamo su **Add Track**

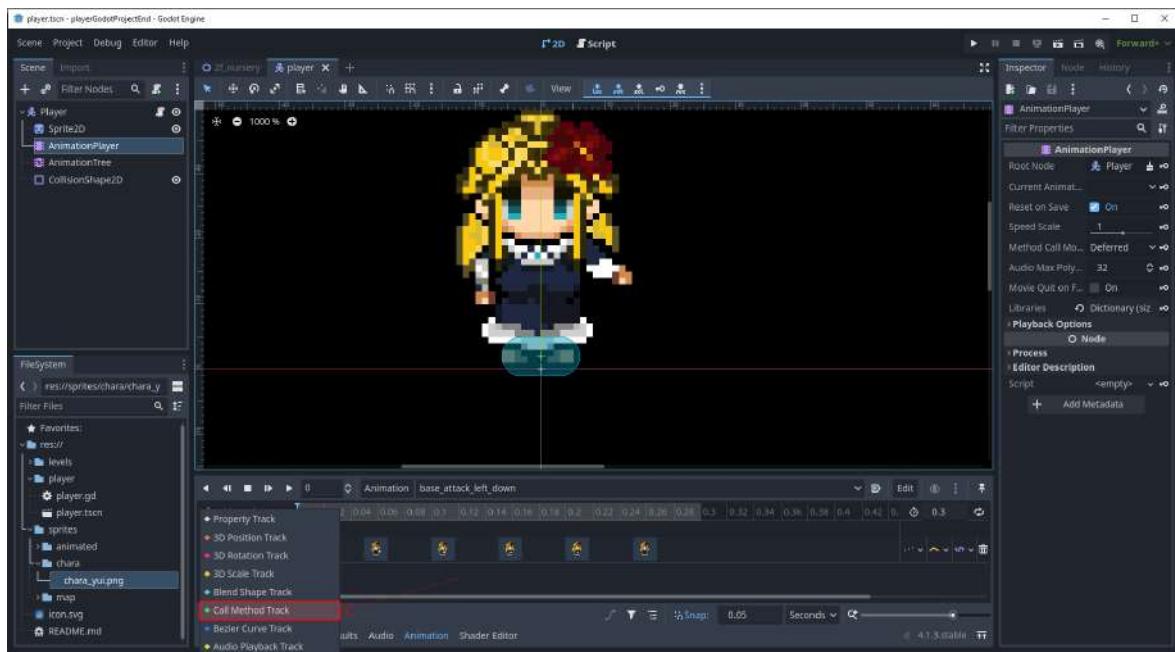
## CHAPTER 2. GODOT



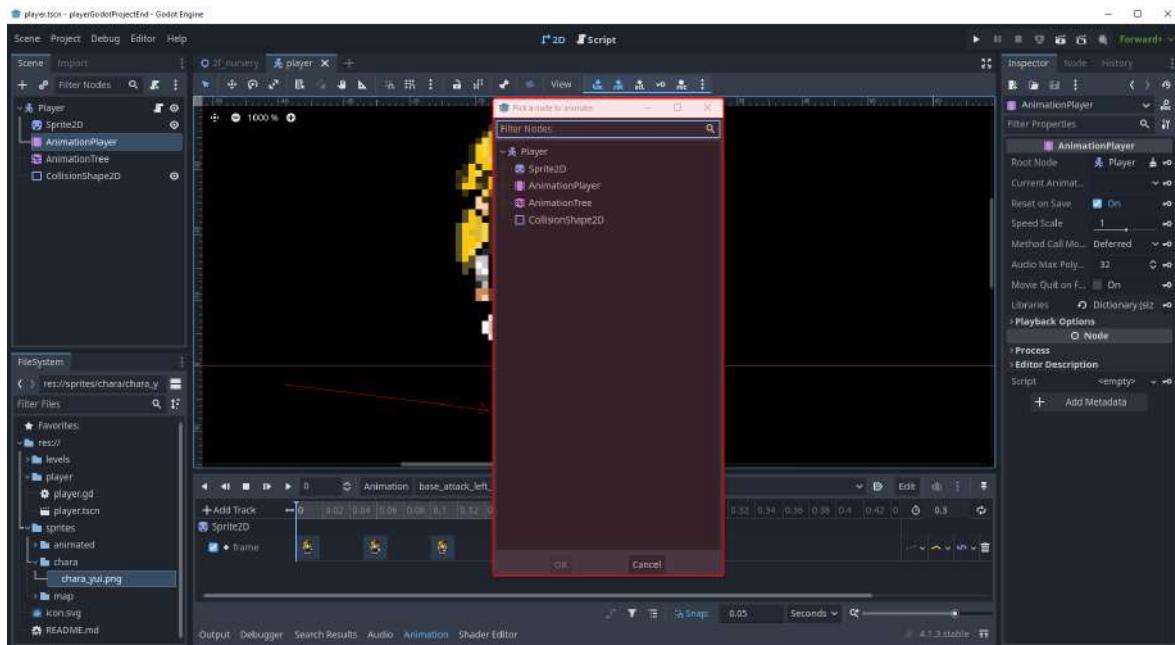
si aprirà un menu a tendina



clicchiamo su **Call Method Track**

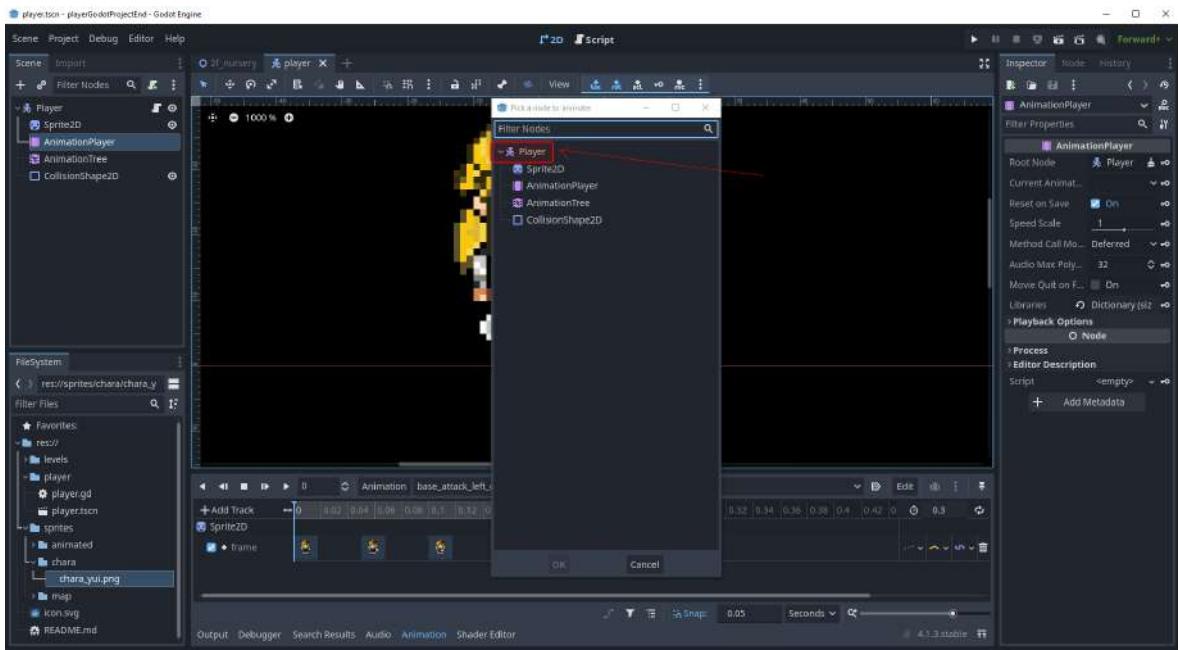


si aprirà una nuova finestra con l'elenco di tutti i nodi della scena

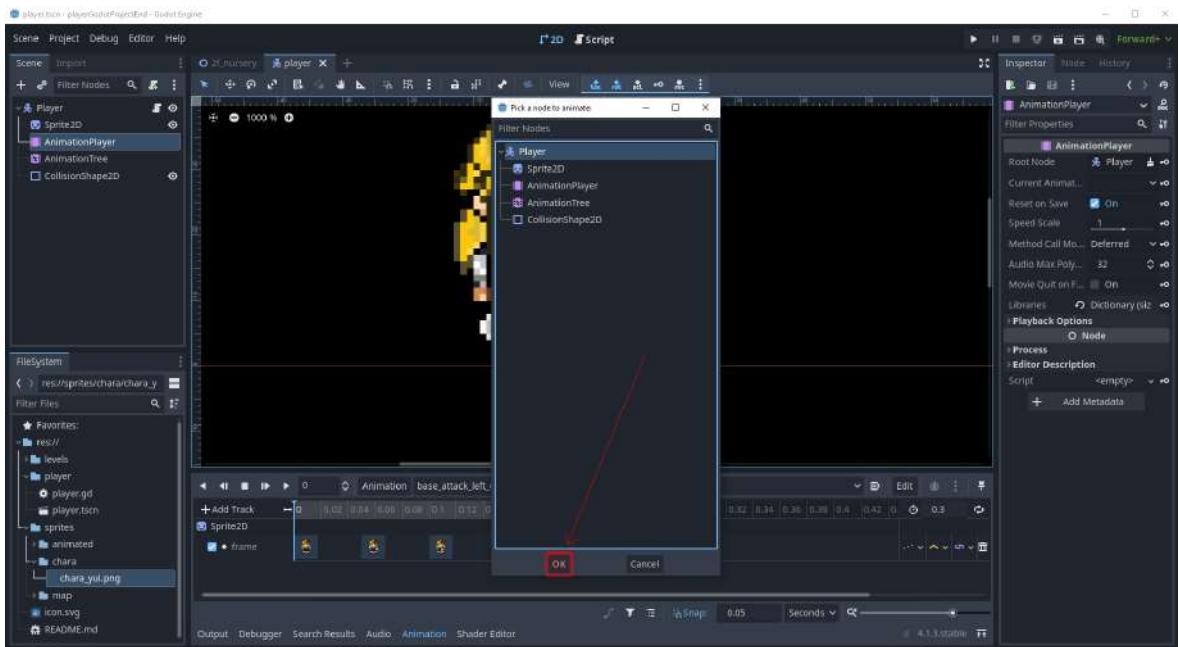


selezioniamo il nodo **Player**

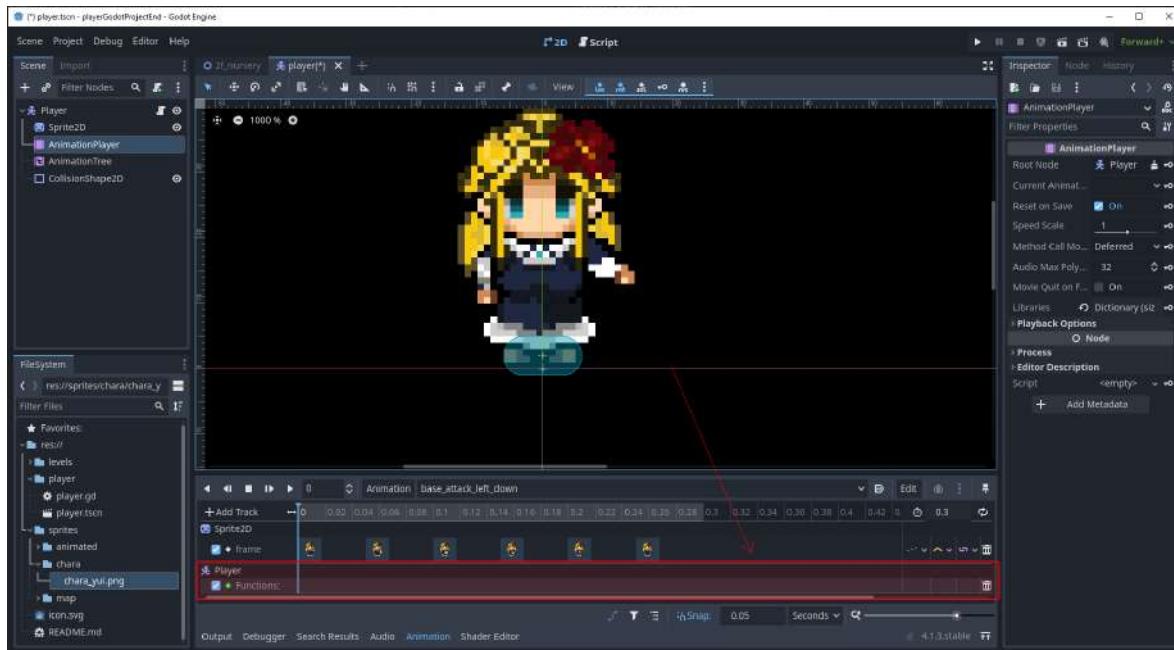
## CHAPTER 2. GODOT



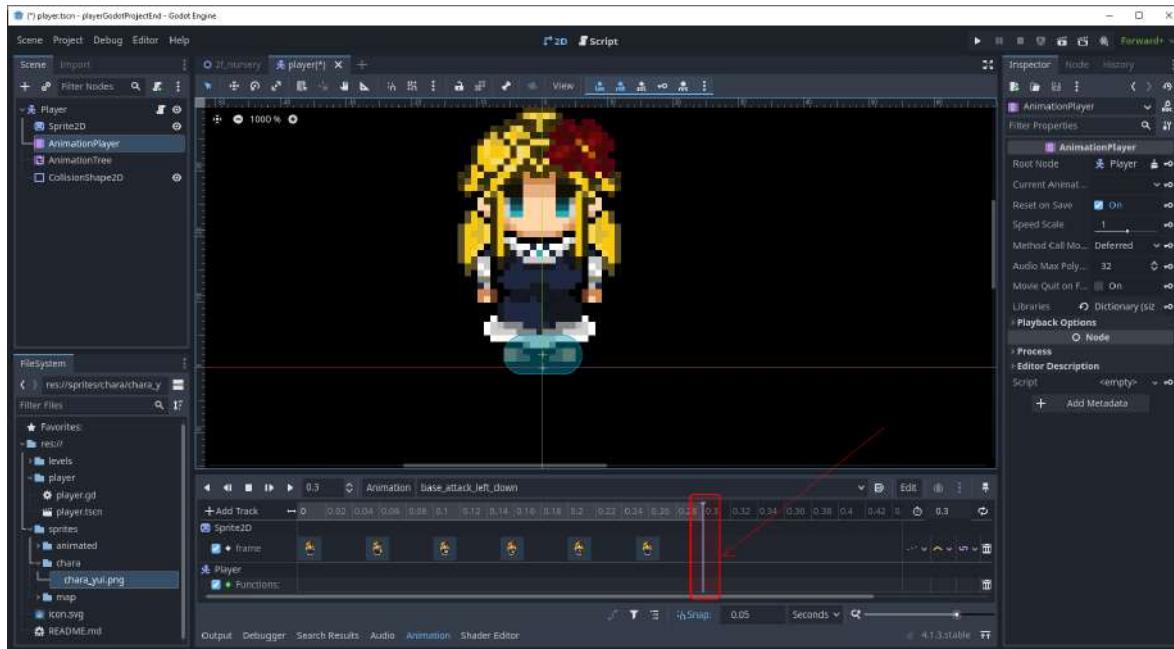
clicchiamo su **Ok**



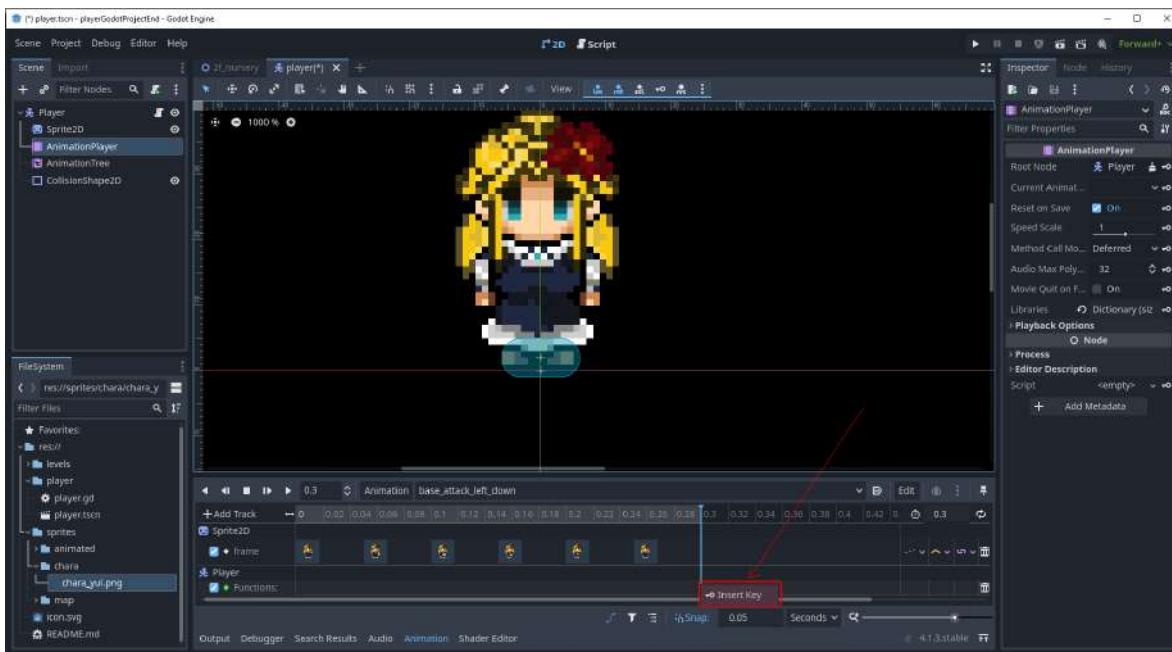
una track del **Player** apparirà nella barra dell'animazione



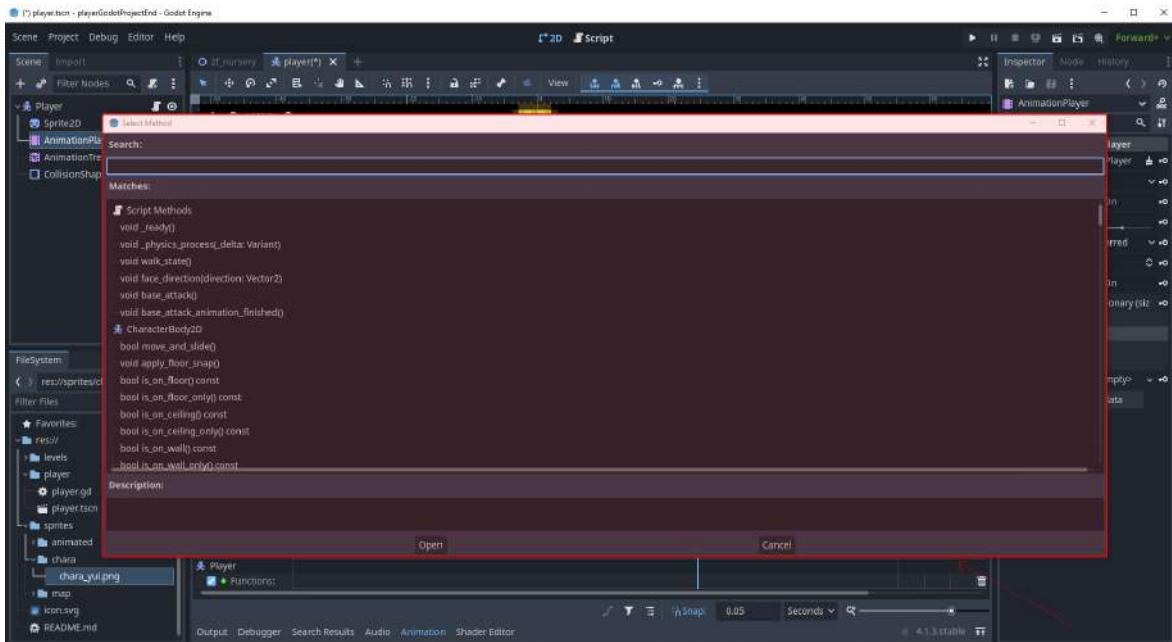
posizioniamo l'indicatore del tempo a fine animazione



clicchiamo con il tasto destro sulla track del **Player**, nel punto dove si trova l'indicatore del tempo, per far apparire una finestra con scritto **Insert Key**

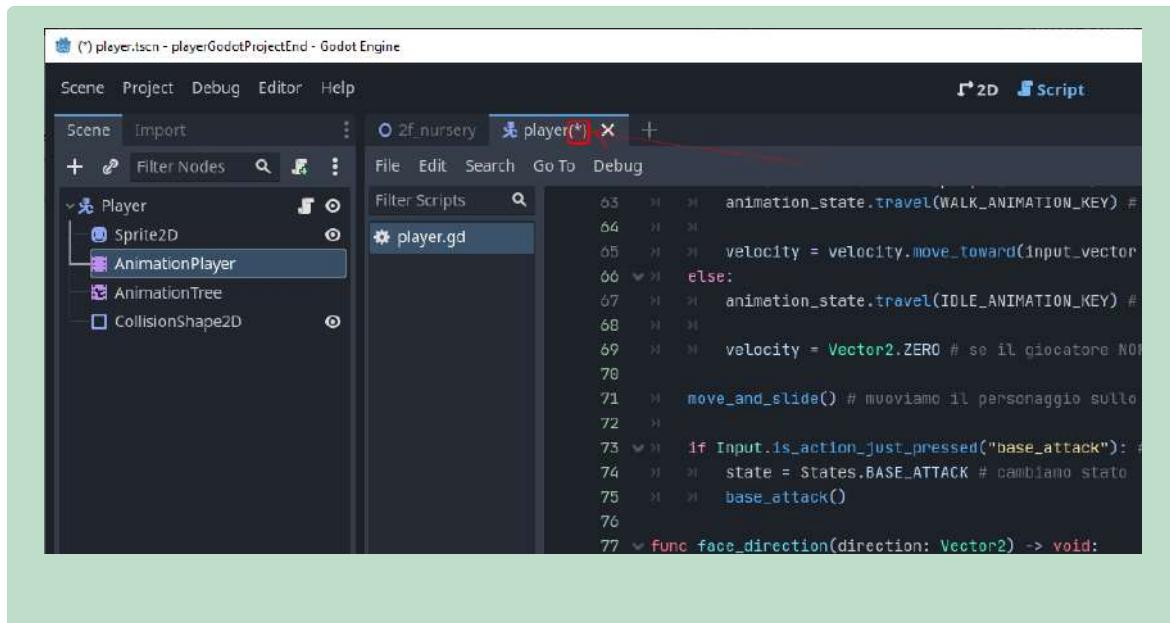


cliccandoci su, apparirà una lista con tutti i metodi della classe **Player**

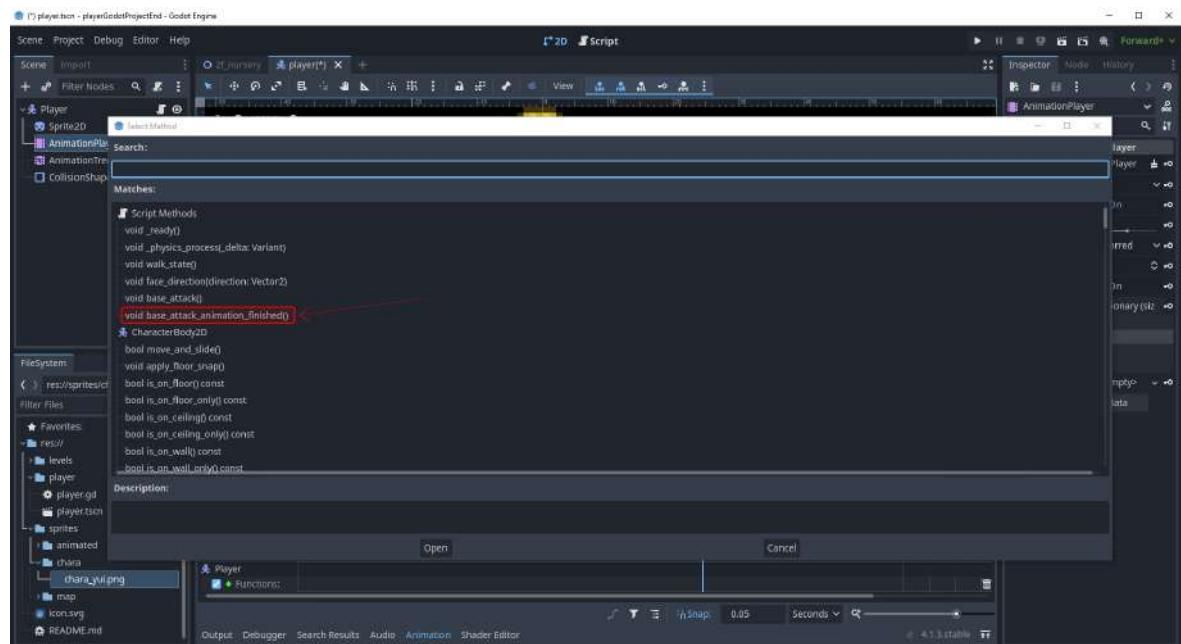


### ⚠️ Consiglio

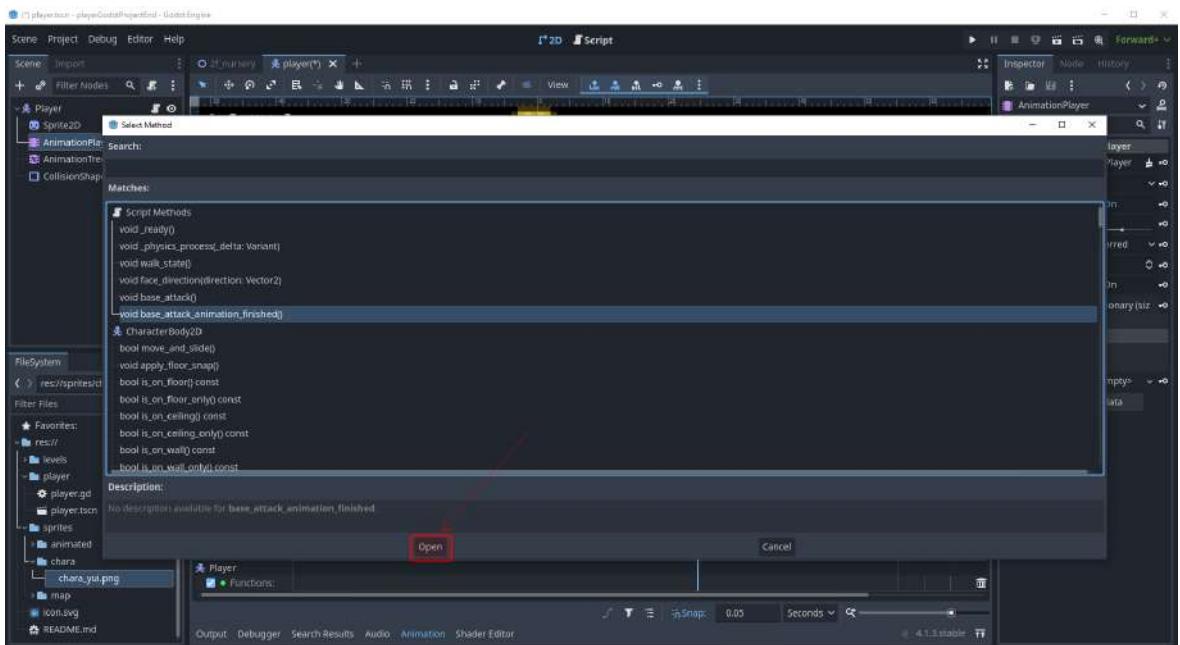
Se un metodo non appare nella lista, molto probabilmente è perché non si ha ancora salvato la scena (e quindi il file di script) (ad esempio tramite **CTRL + S**). Con il simbolo **(\*)**, Godot ci indica che la scena aperta è stata modificata, ma non ancora salvata



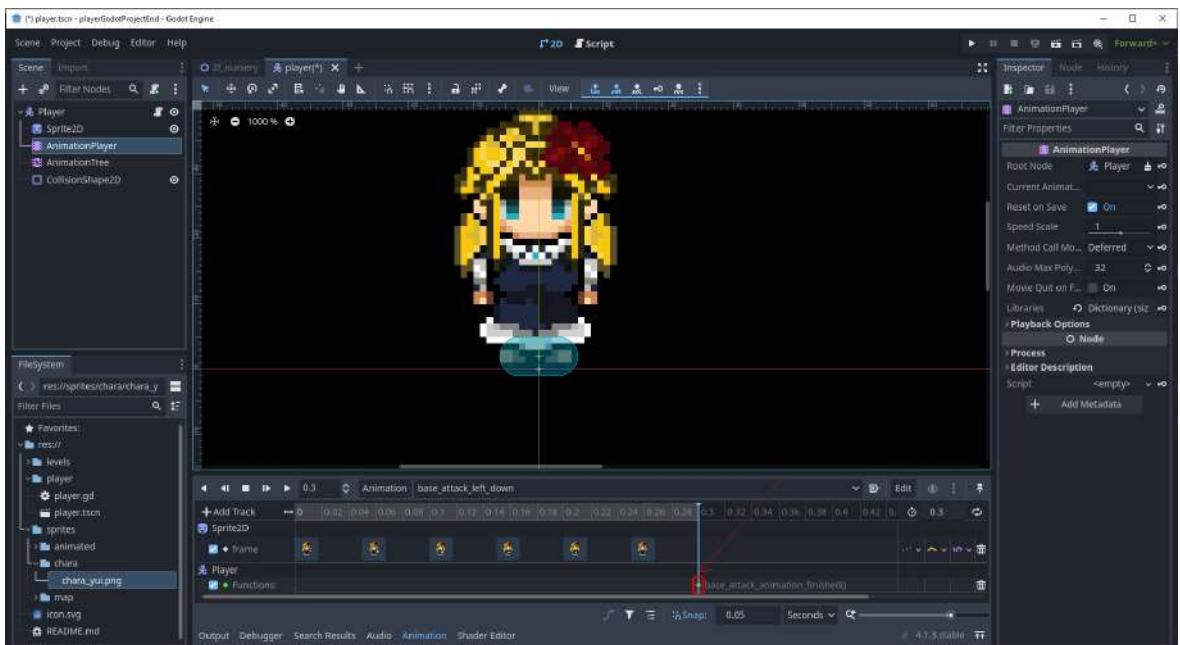
clicchiamo su **base\_attack\_animation\_finished()**



clicchiamo su **Open**



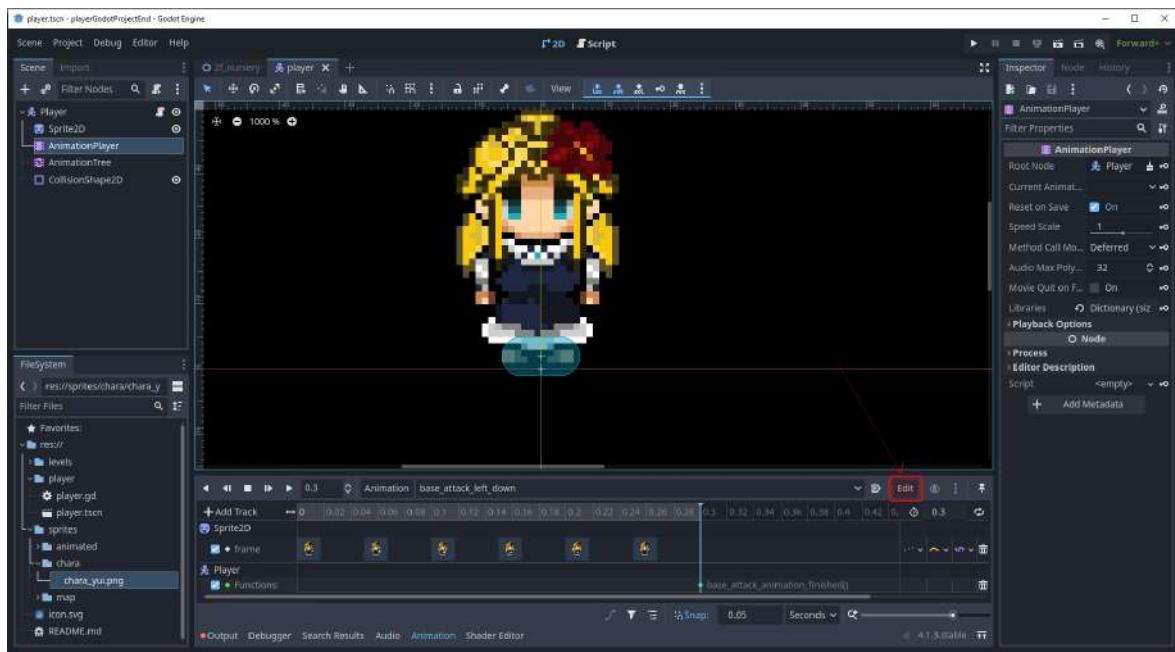
verrà aggiunto un nuovo keyframe nella timeline dell'animazione



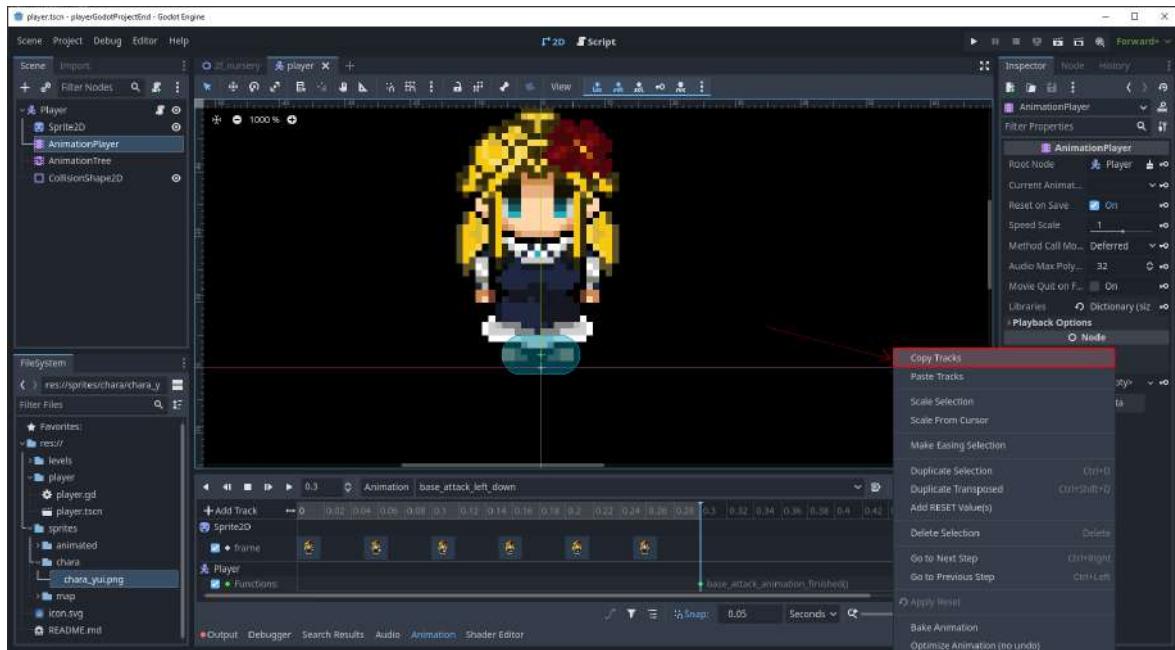
Così facendo, Godot si farà cura di chiamare la funzione `base_attack_animation_finished()` al termine dell'animazione di attacco.

Questo passaggio, tuttavia, va fatto per *tutte* le animazioni di attacco.

C'è però un modo per copiare e incollare le track di un'animazione velocemente. Clicchiamo su `Edit`

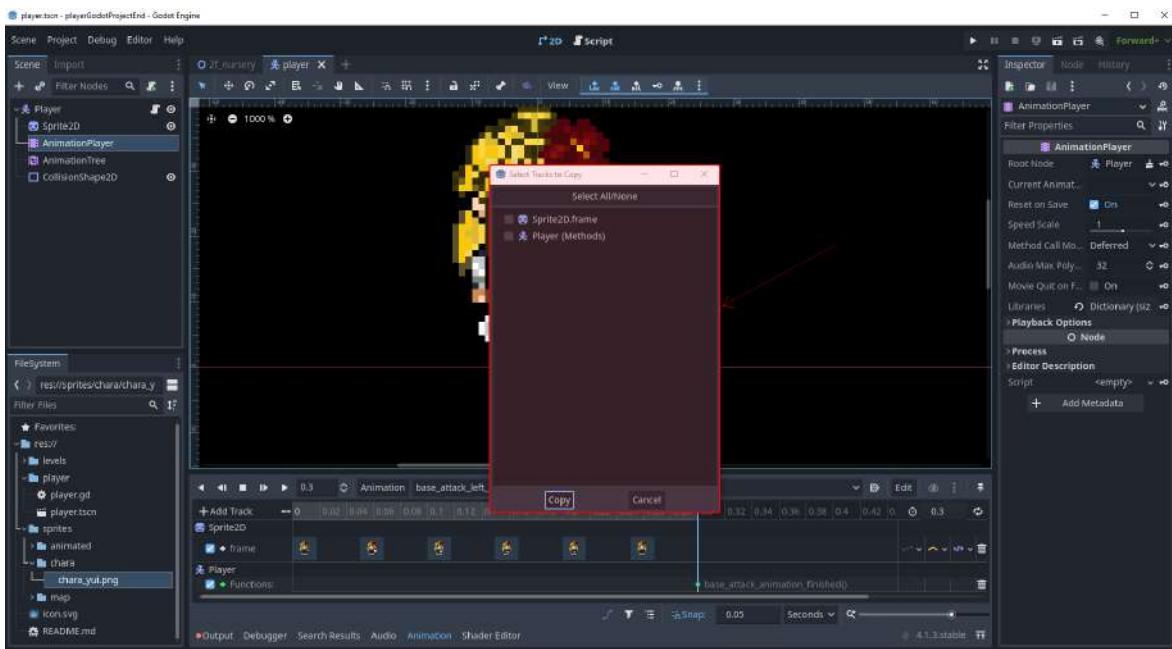


clicchiamo su **Copy Tracks**

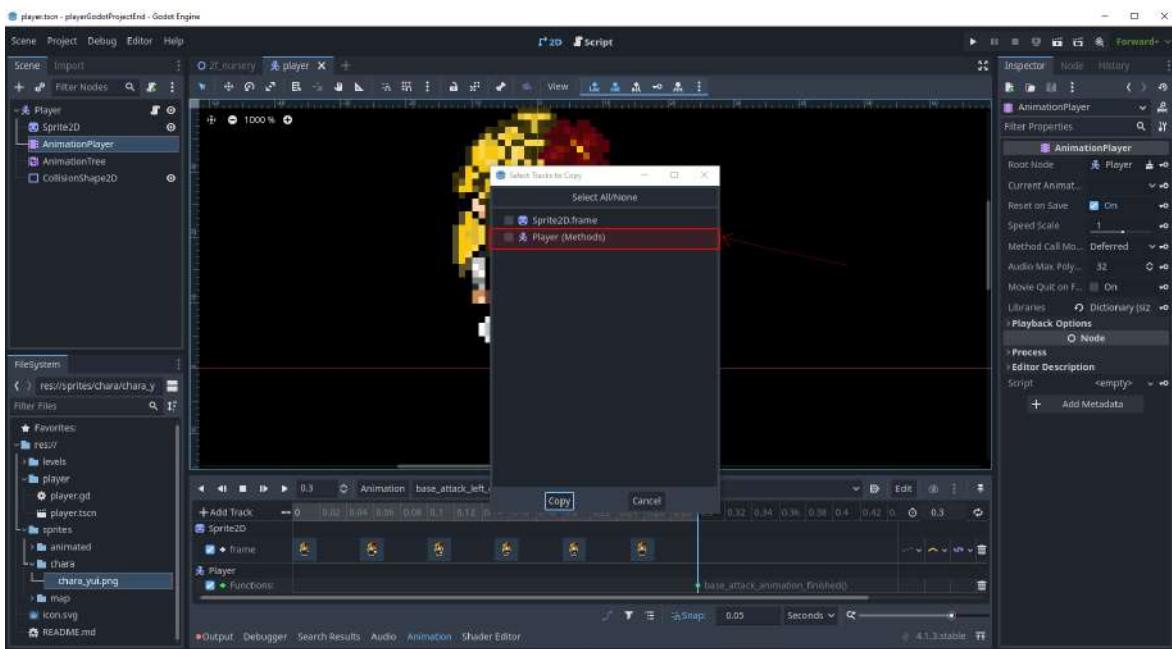


apparirà una lista di tutte le tracce dell'animazione

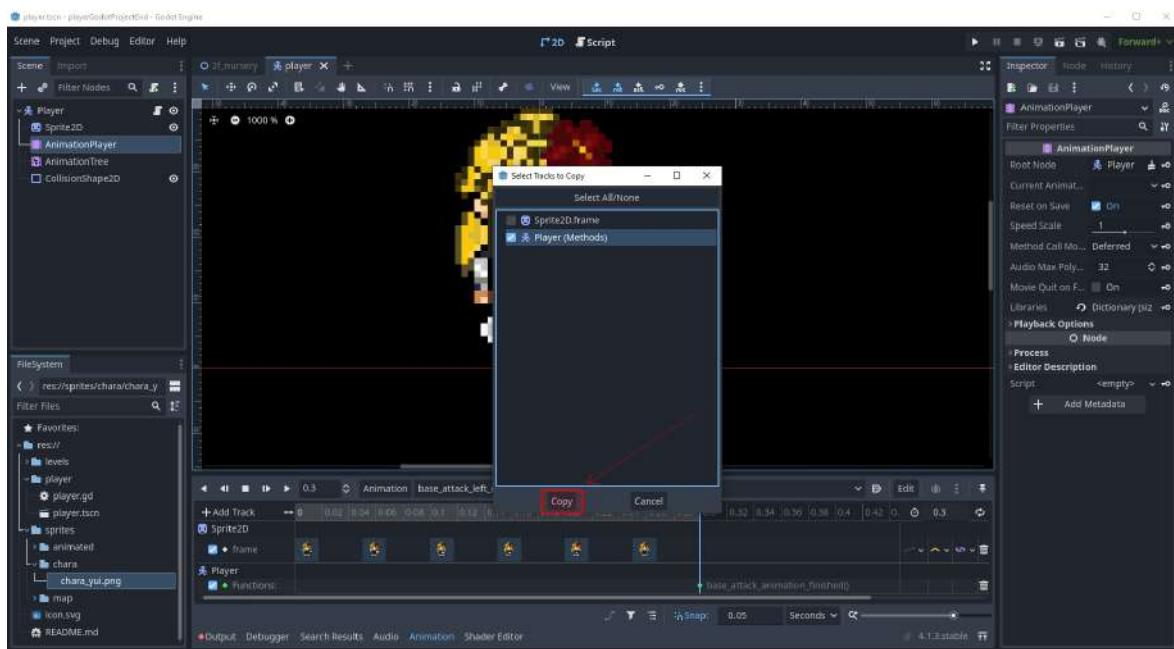
## CHAPTER 2. GODOT



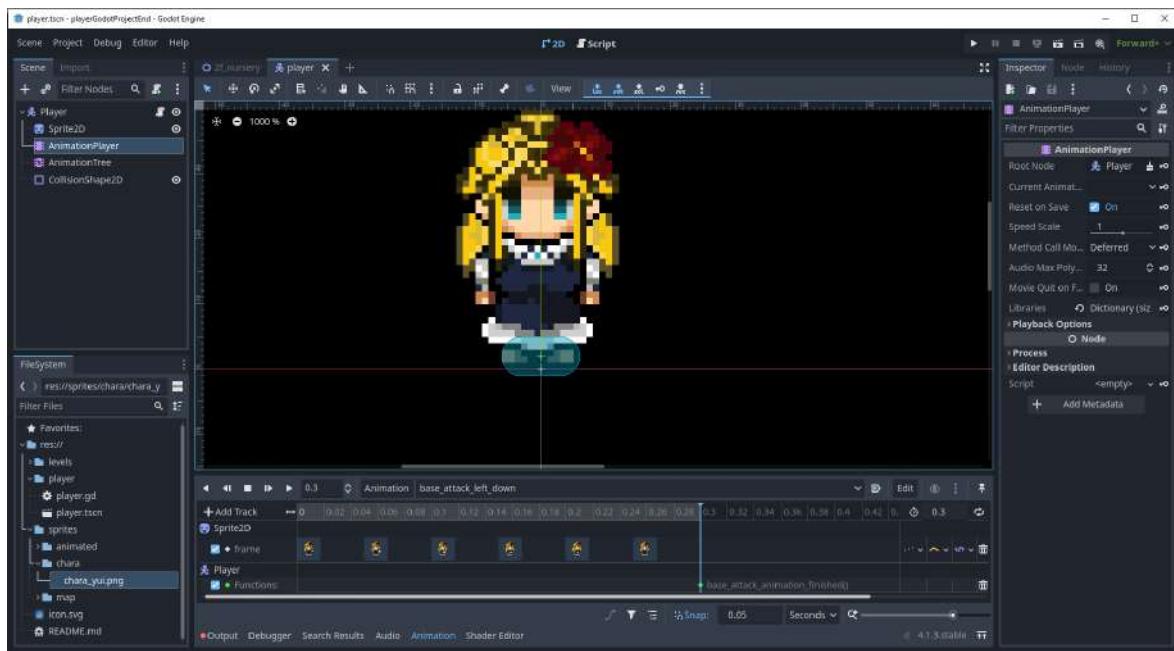
clicchiamo sulla voce **Player (Methods)**



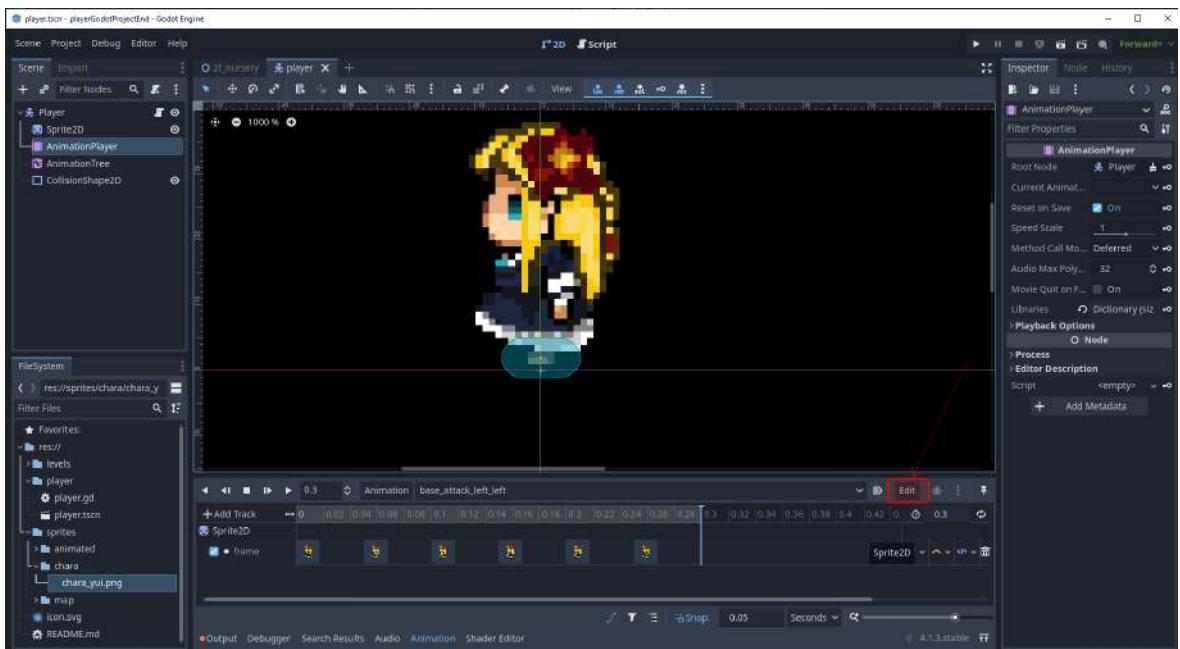
clicchiamo su **Copy**



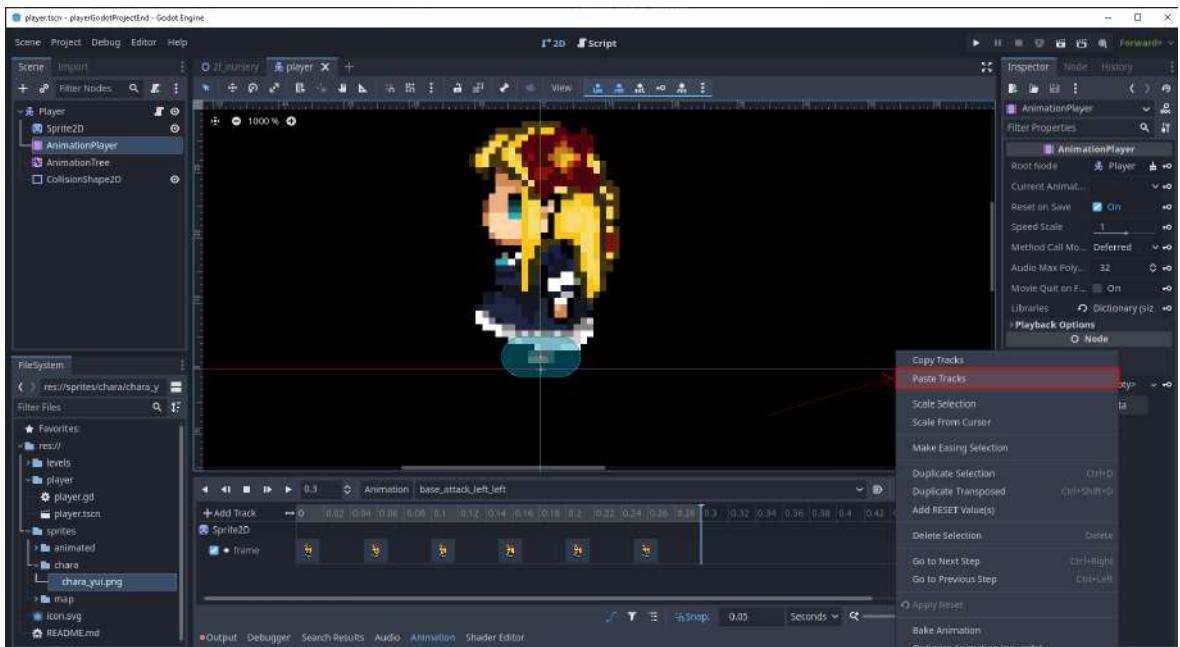
la finestra si chiuderà.



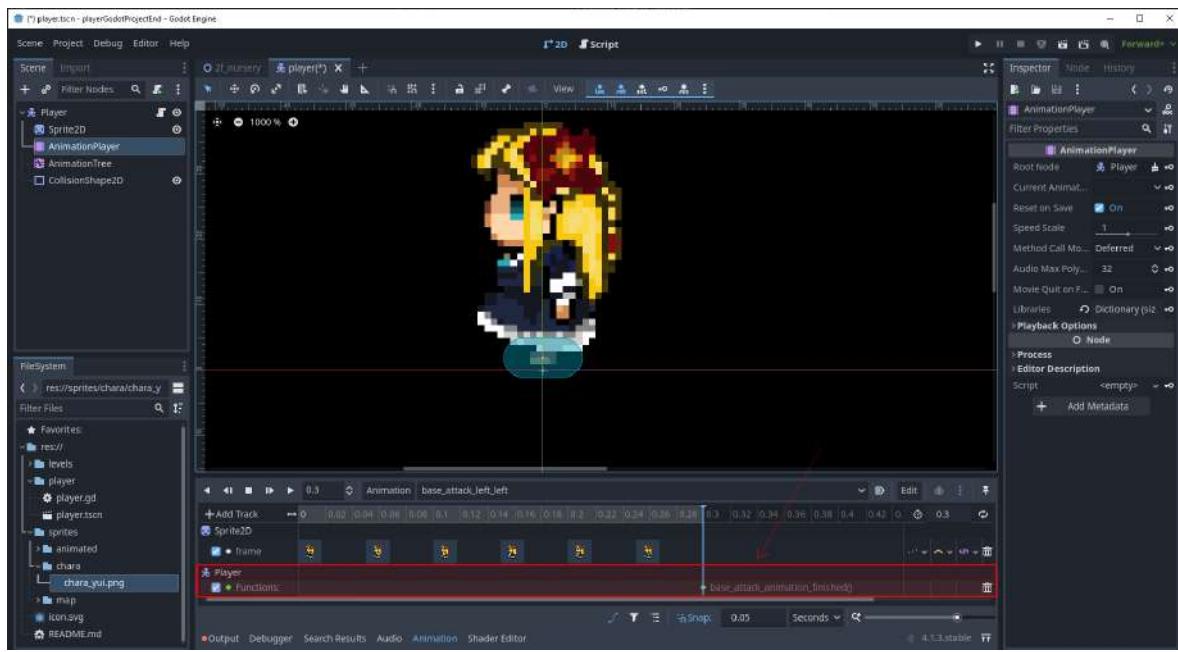
Dopo aver aperto l'animazione in cui incollare la traccia appena copiata, clicchiamo su **Edit**



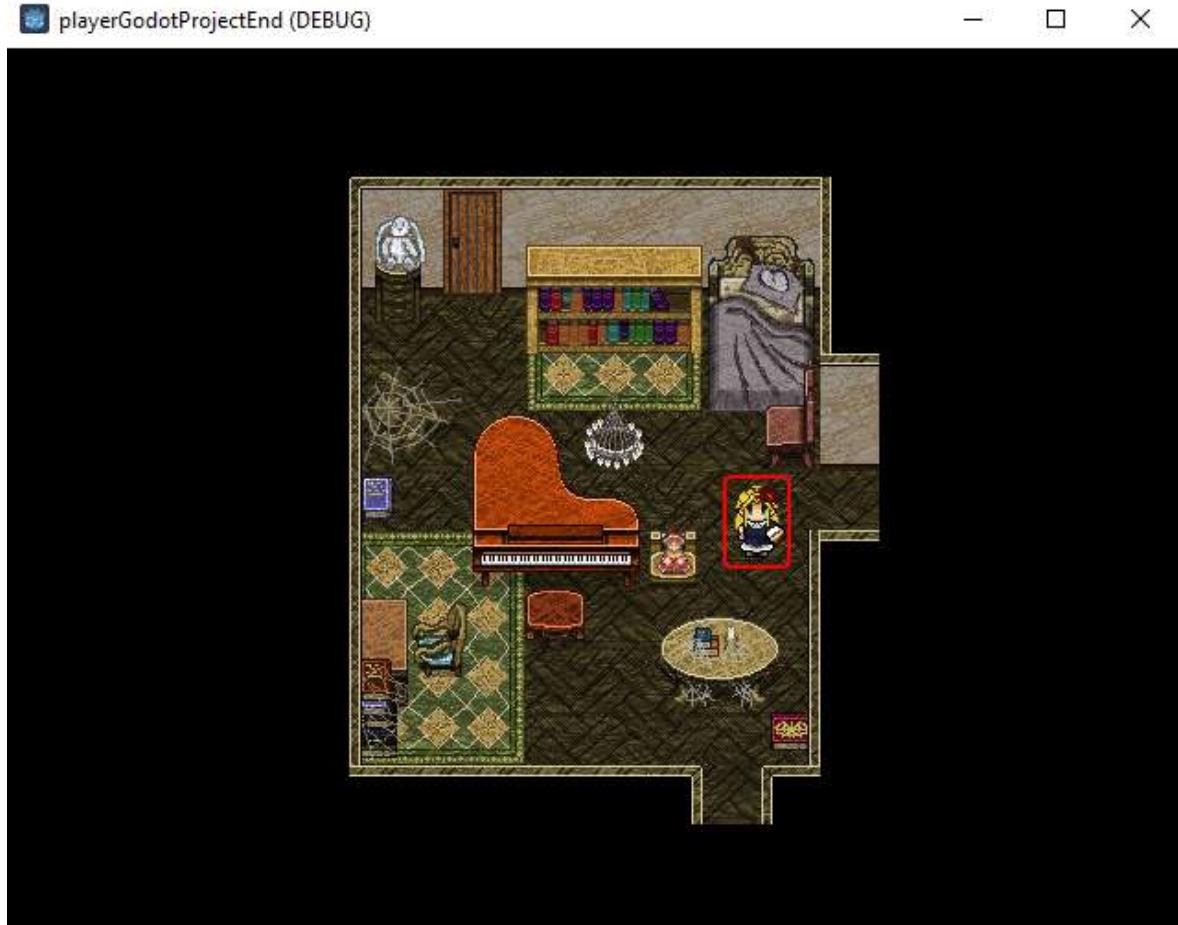
clicchiamo **Paste Tracks**



la traccia verrà incollata con successo nell'animazione



Dopo aver fatto questo procedimento con tutte le animazioni di attacco, avviando il gioco, il player non si bloccherà più dopo il primo attacco ma potremmo bensì attaccare e camminare quanto vogliamo



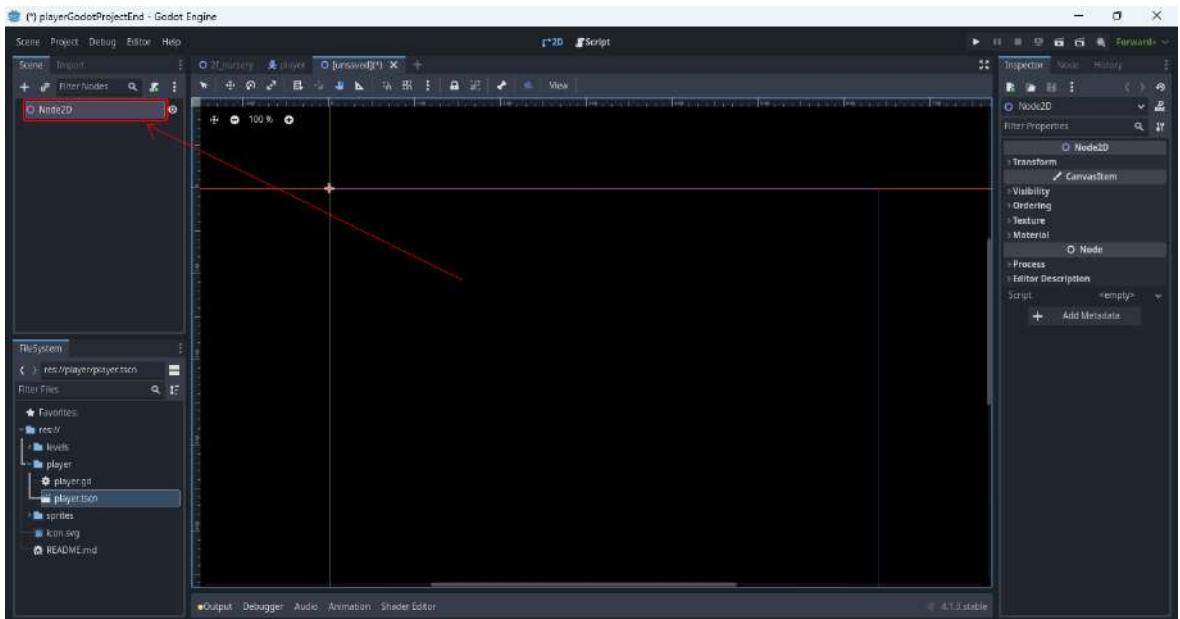
### 2.4.6.1 Projectiles

Il nostro personaggio è finalmente in grado di attaccare. Anche se, ad essere sinceri, al momento sembra solamente star tirando schiaffi all'aria. Per rendere il tutto più emozionante, aggiungiamo dunque un **projectile** all'attacco.

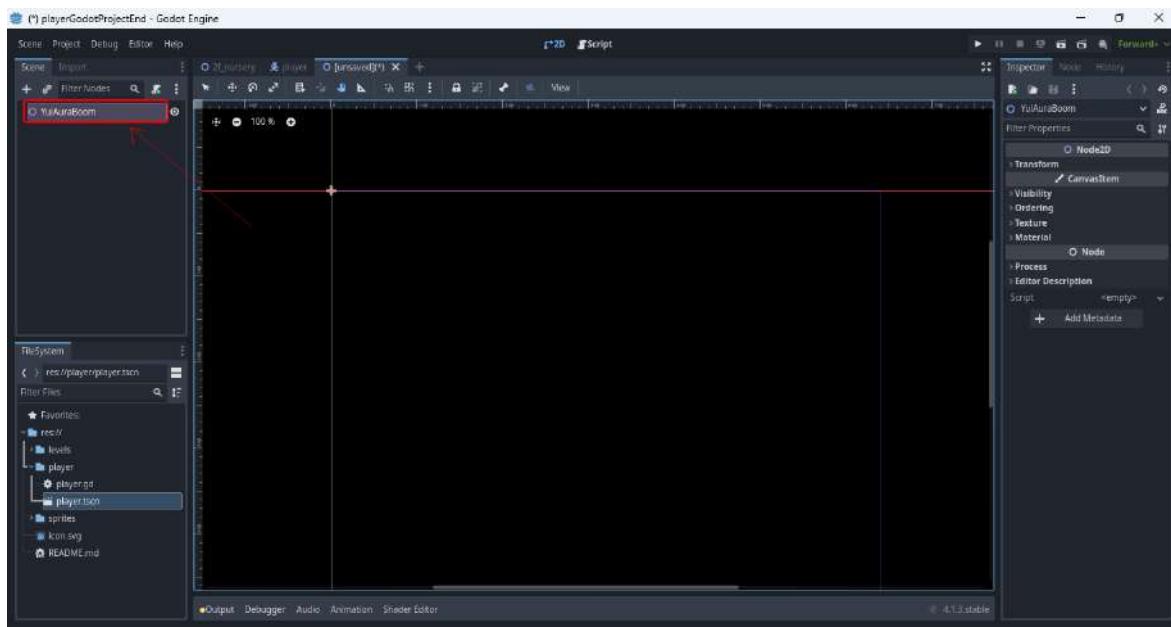
Nell'ambito dei videogiochi, per **projectile** (**proiettile**) si intendono tutti quegli oggetti il cui movimento è basato sul moto dei proiettili. Rientrano in questa categoria oggetti come: proiettili di pistola, frecce di un'arco, raggi laser, ma anche sassi o sedie lanciate. In poche parole, nello sviluppo dei videogiochi, tutto ciò che può essere lanciato (e che dunque segue il moto del proiettile), è un **projectile**.

Nel nostro caso, il proiettile che vogliamo far sparare al player è una sonic boom di energia.

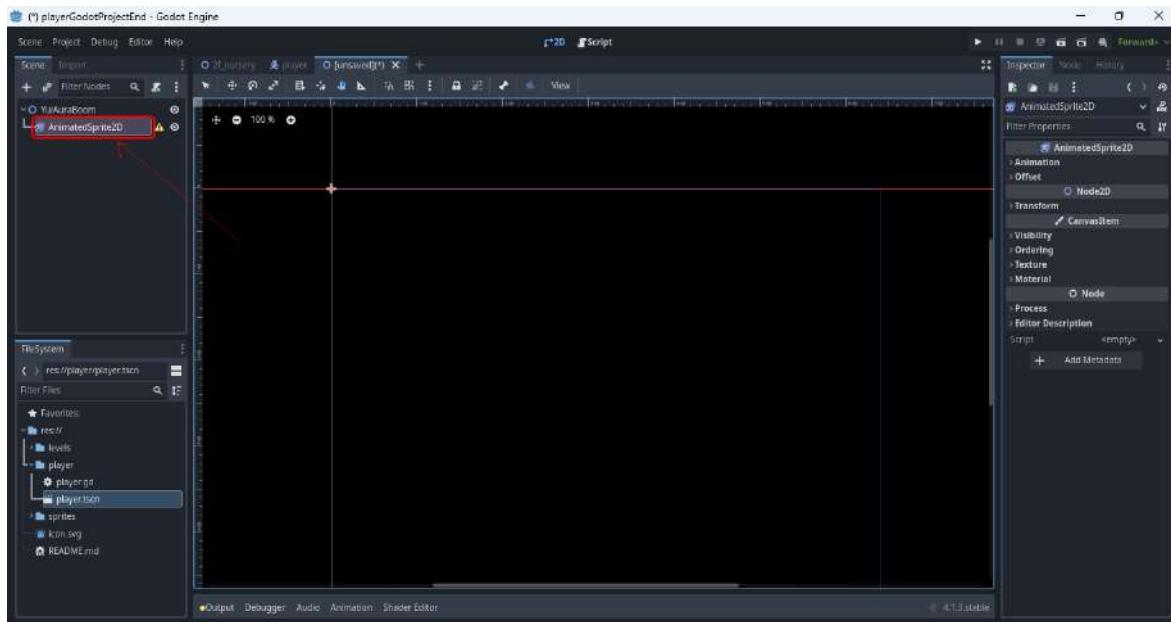
Creiamo dunque una nuova scena di tipo **Node2D**



rinominiamola **YuiAuraBoom**



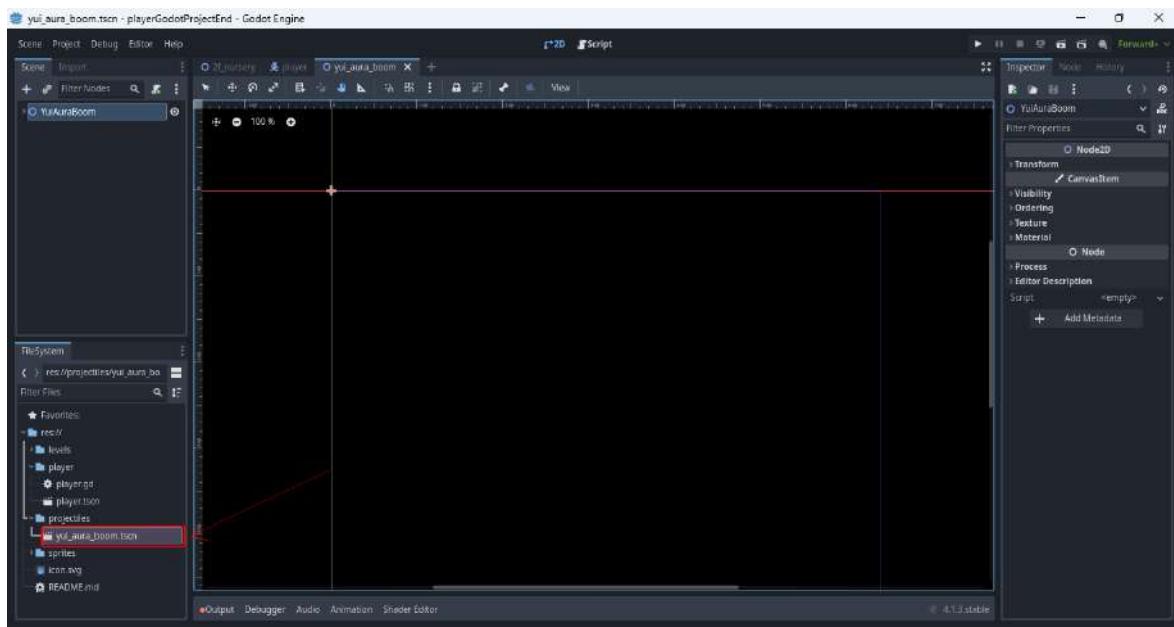
aggiungiamo un nodo **AnimatedSprite2D**



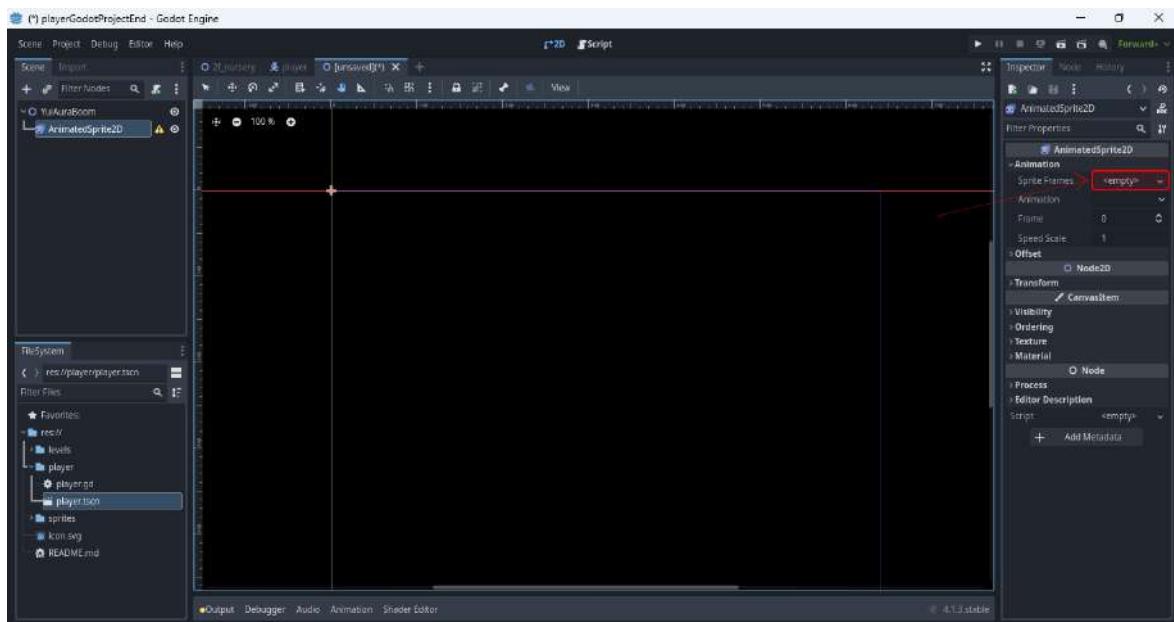
Il nodo **AnimatedSprite2D** è simile al nodo **Sprite2D**, con la differenza che contiene più texture come animation frame. Le animazioni vengono create utilizzando una risorsa **SpriteFrames** (che vedremo a breve), che consente di importare file immagine (o una cartella contenente tali file) per fornire animation frame per lo sprite. La risorsa **SpriteFrames** può essere configurata nell'editor tramite il pannello inferiore **SpriteFrames**.

Salviamo dunque la nuova scena all'interno della cartella **res://projectiles**

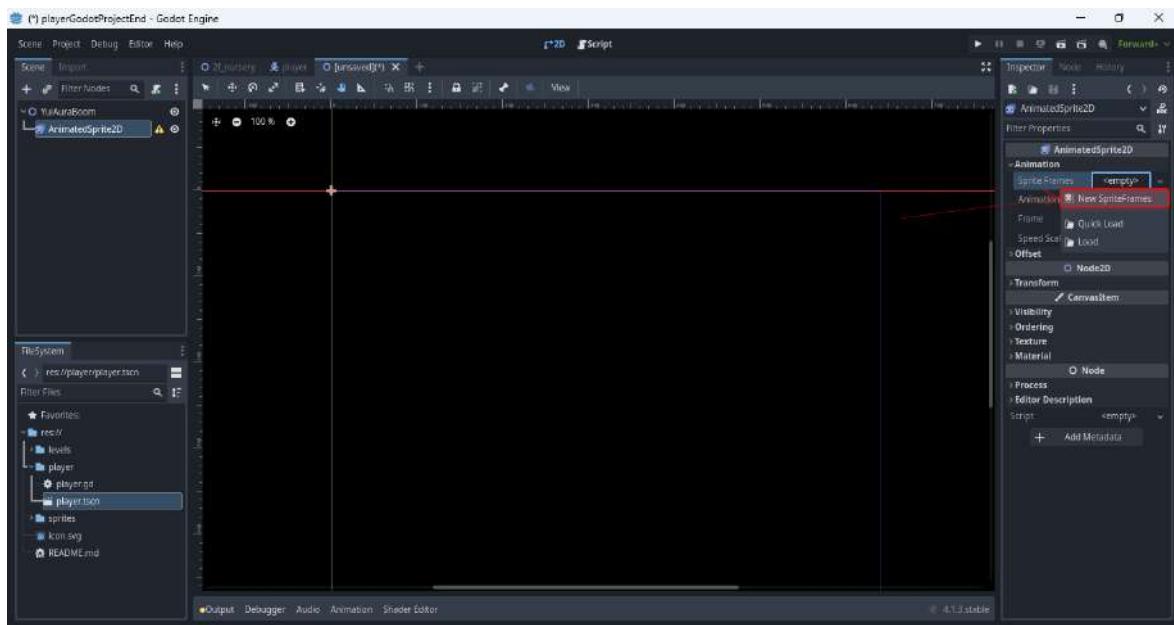
## CHAPTER 2. GODOT



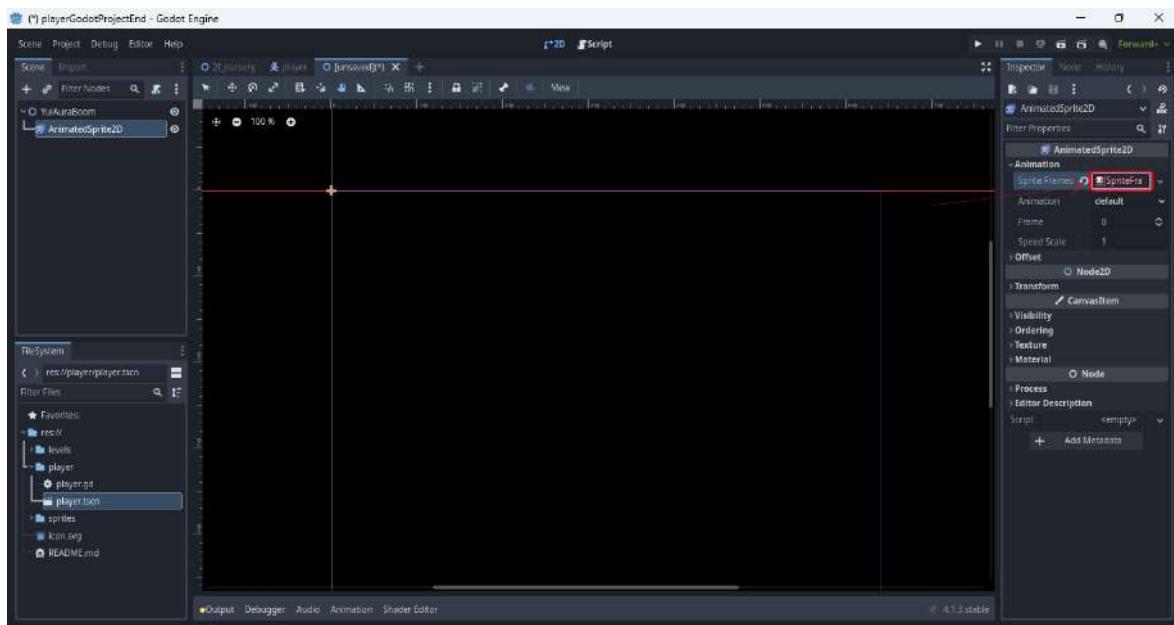
dopo averlo selezionato, clicchiamo su `<empty>` sotto la voce `Sprite Frames`



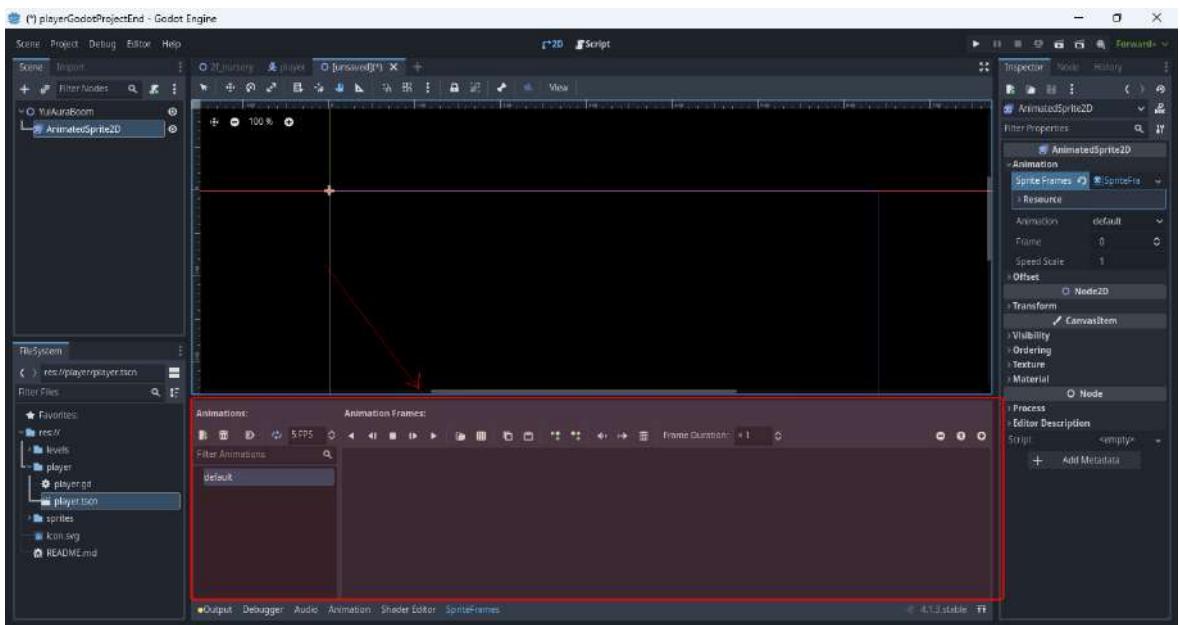
clicchiamo `New SpriteFrames`



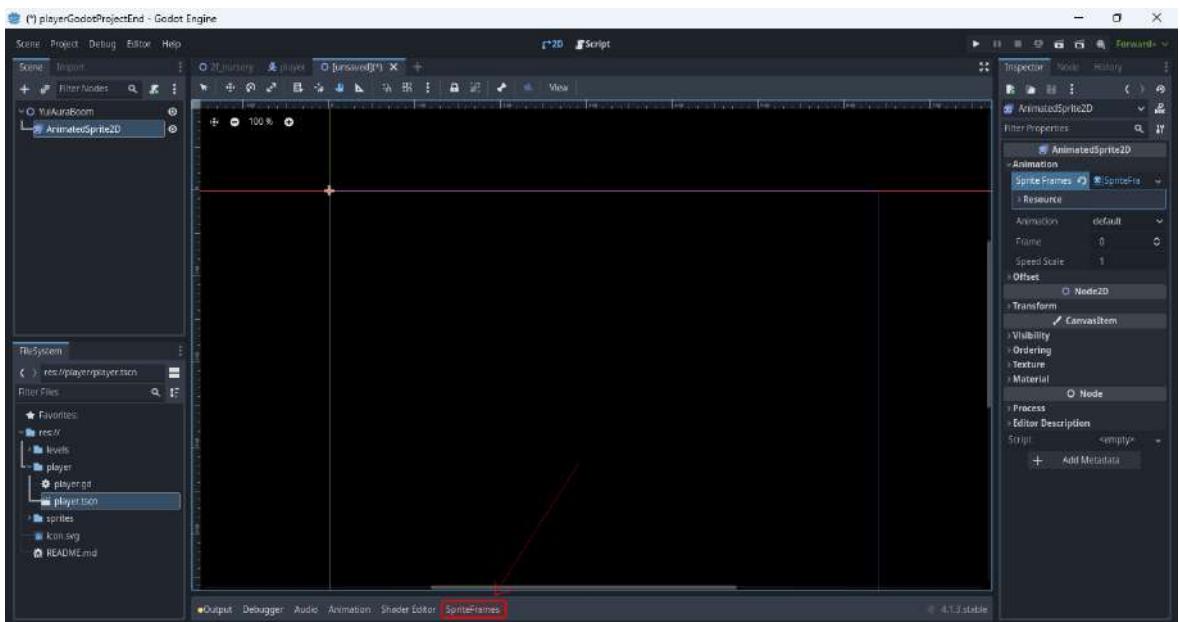
clicchiamo sullo **SpriteFrame** appena creato



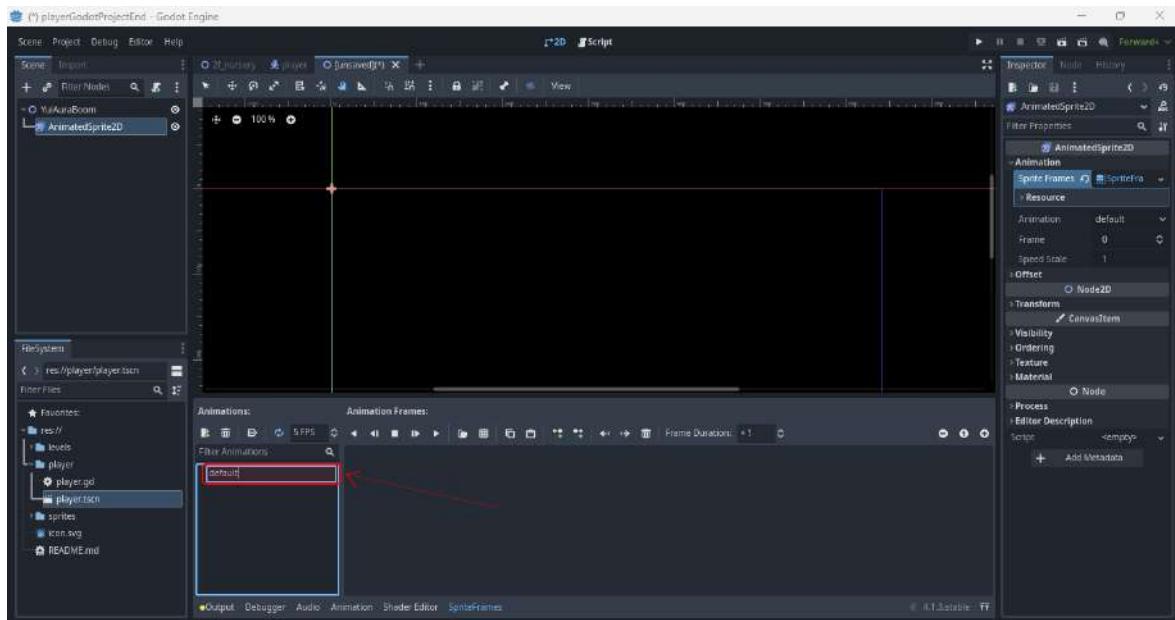
si aprirà una nuova finestra



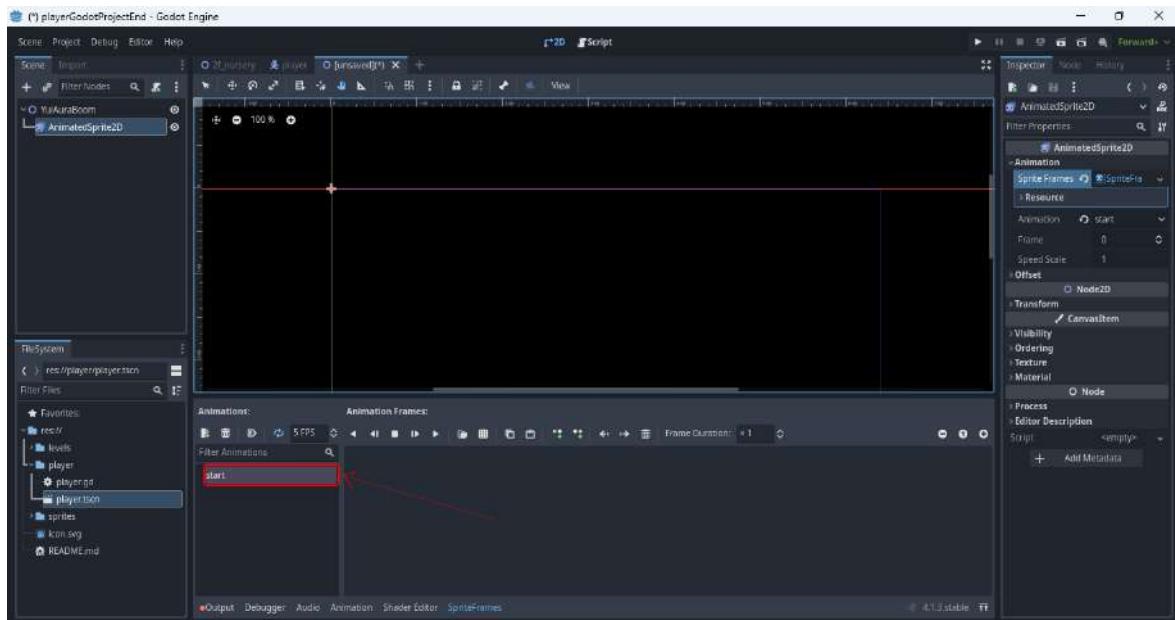
se ciò non dovesse accadere, basta cliccare sulla voce **SpriteFrames**



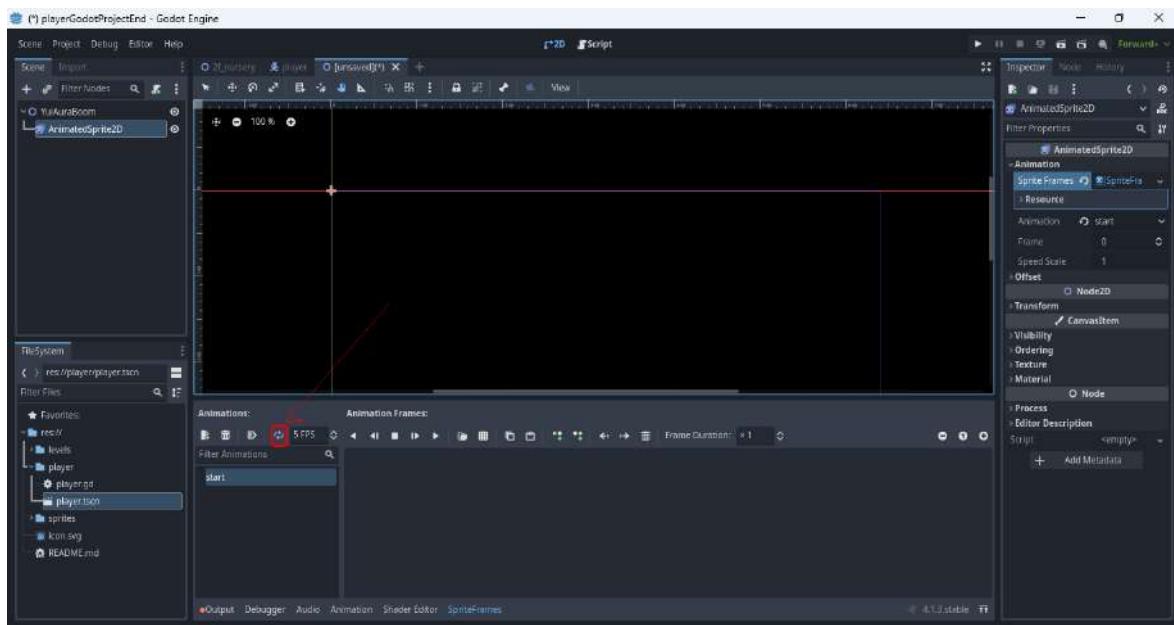
facendo doppio click sull'animazione **default**



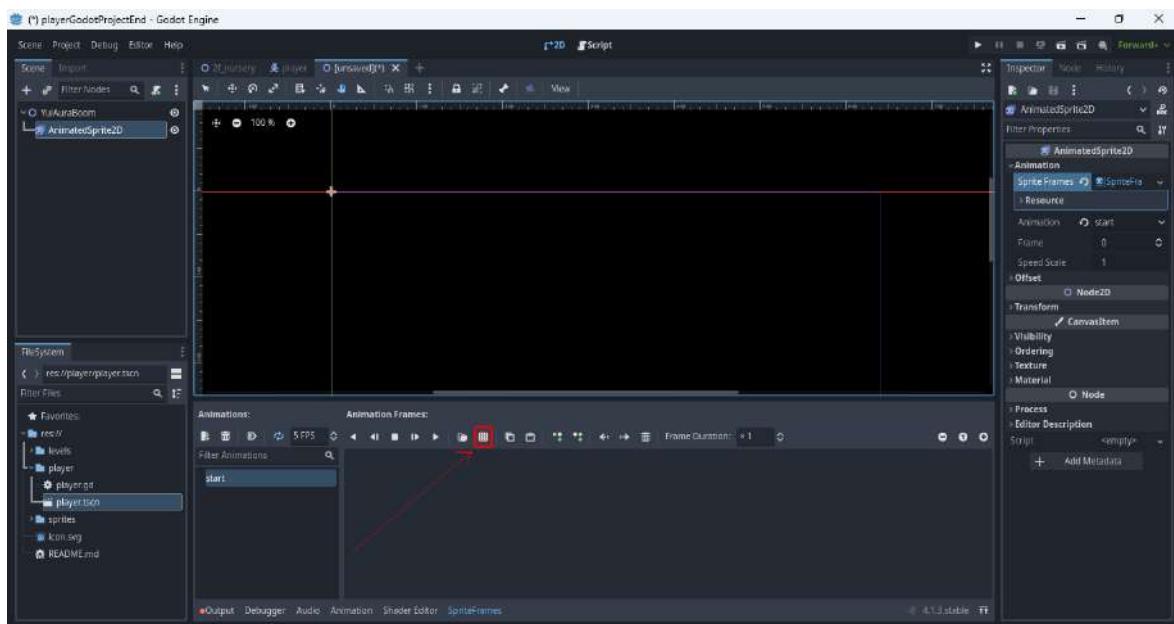
la rinominiamo in **start**



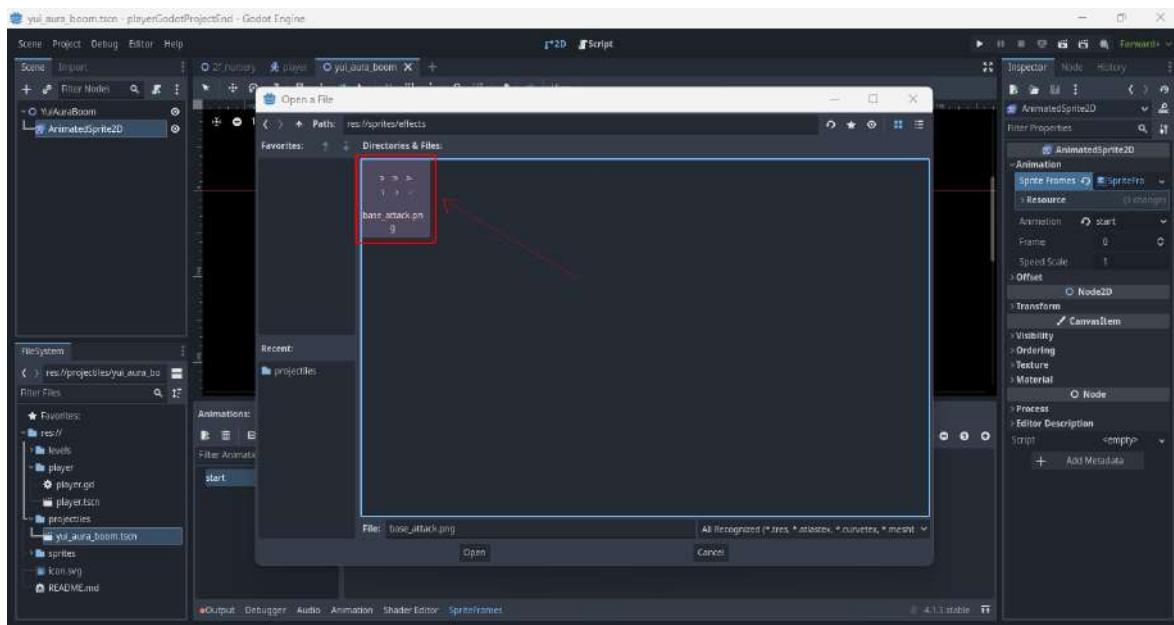
Questa sarà l'animazione che verrà riprodotta in loop mentre il nostro projectile è in movimento



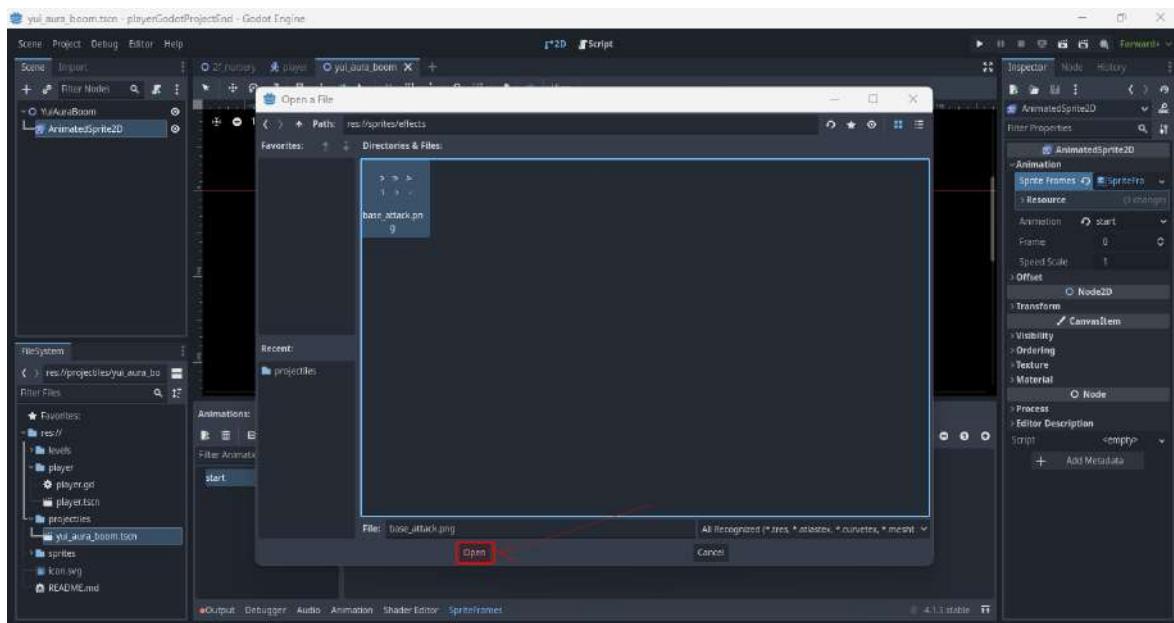
clicchiamo sull'icona della griglia per aggiungere frames a partire da uno spritesheet



selezioniamo lo spritesheet `base_attack.png` dentro la cartella `res://sprites/effects`

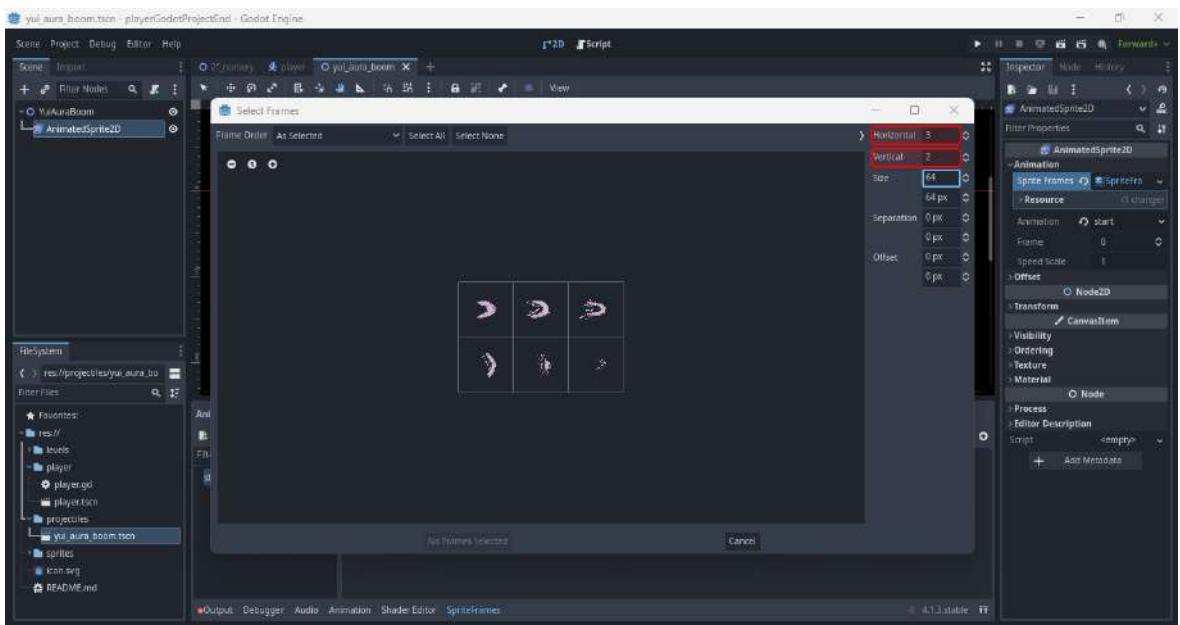


clicchiamo su **Open**

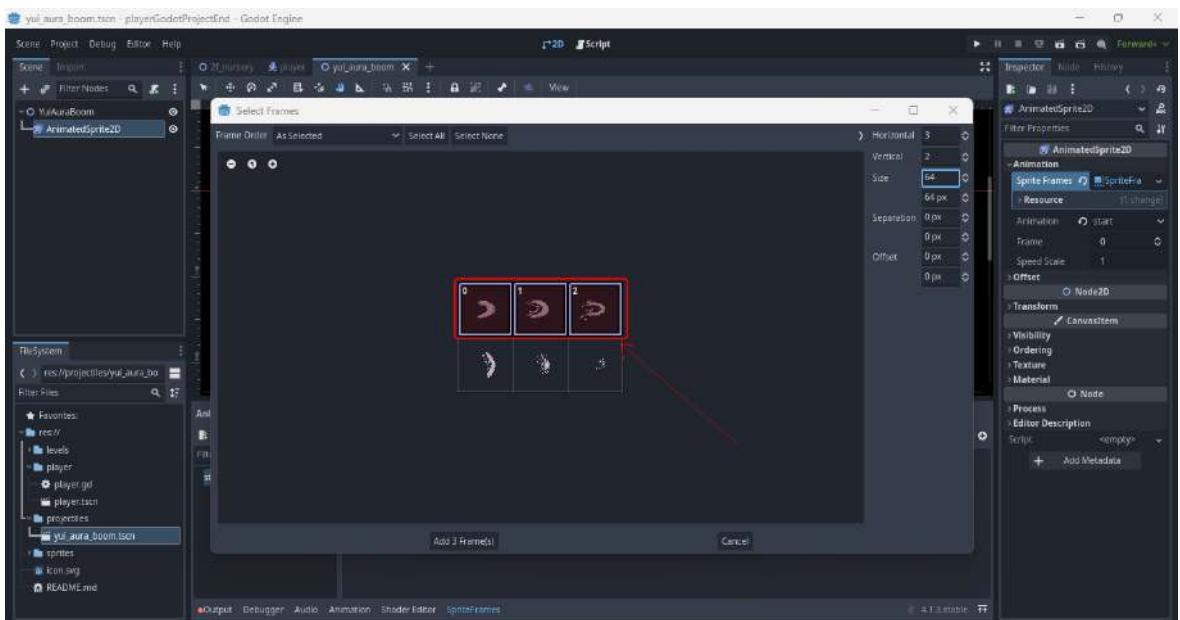


2 nella nuova finestra impostiamo le dimensioni dello spritesheet, ovvero **3** colonne e **2** righe

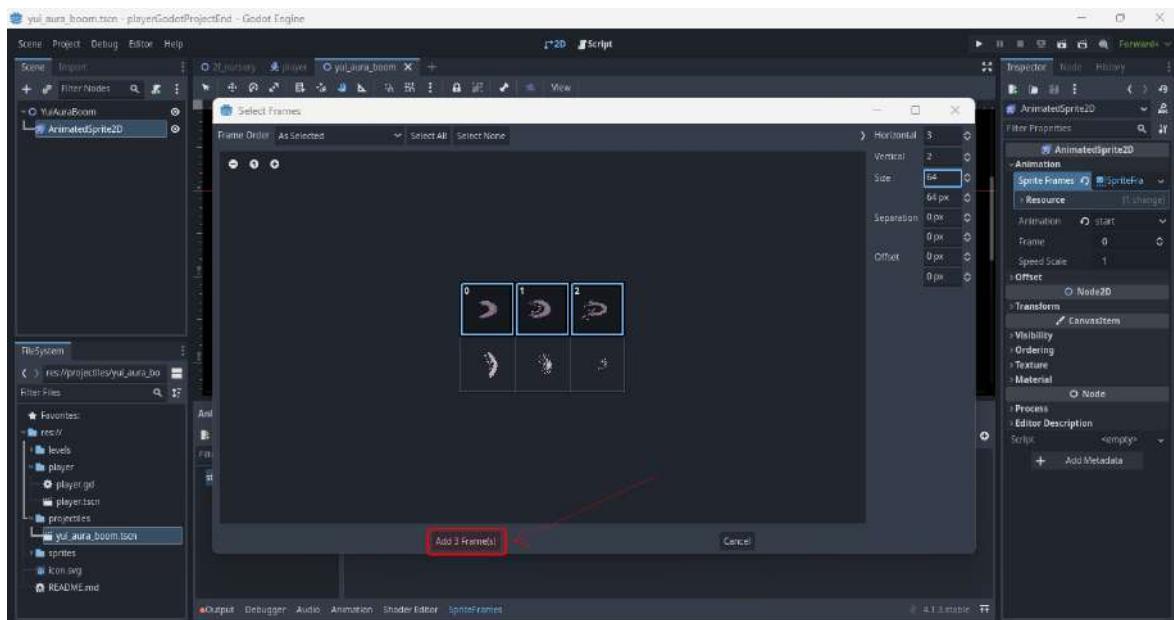
## CHAPTER 2. GODOT



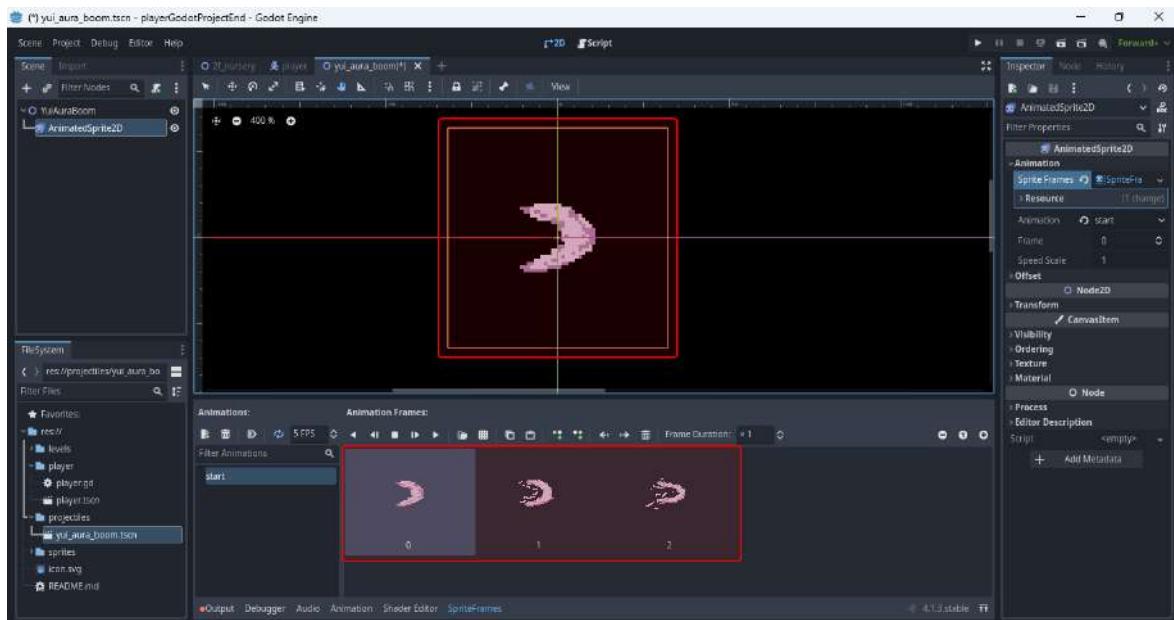
selezioniamo gli sprite 0, 1 e 2



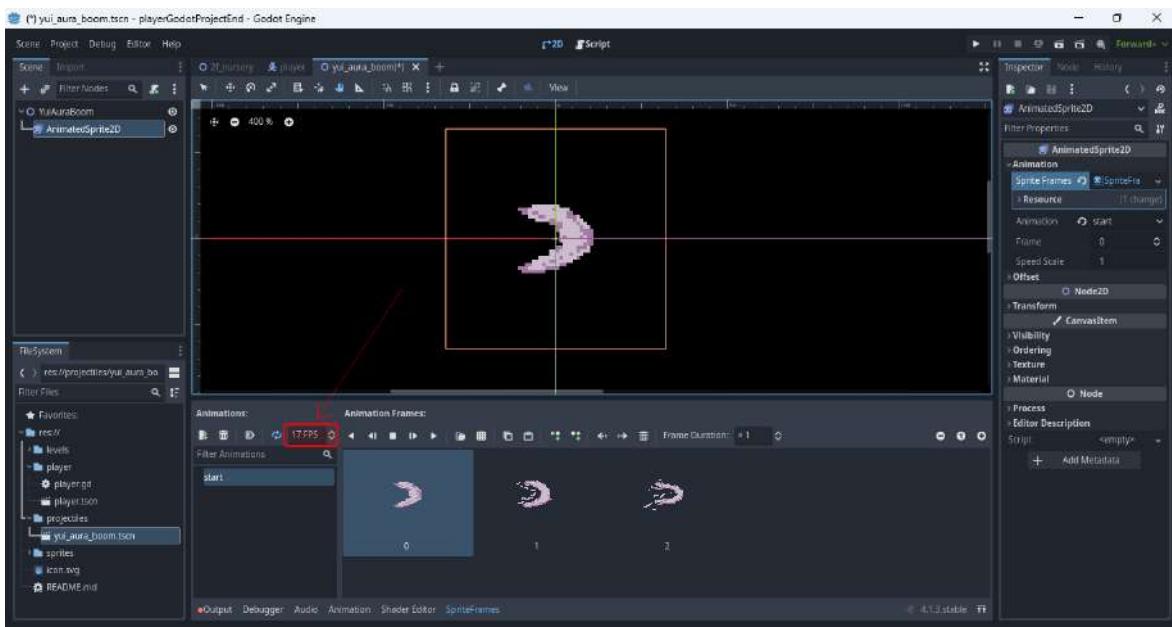
e clicchiamo su **Add 3 Frame(s)**



i frame verranno correttamente aggiunti all'animazione e l'immagine del nostro "aura boom" comparirà nella scena



impostiamo ora la velocità di animazione a **17 FPS** ("frame per second", "frame al secondo"). Così facendo, ogni secondo avrà 17 frame al suo interno. Utilizzando un po' di matematica, poiché 1 secondo è formato da 1000 millisecondi, allora  $1000/17=58.8$  millisecondi circa; questo significa che i frame 0, 1 e 2 del nostro aura boom si alterneranno, nel giro di 1 secondo, ogni 58.8 millisecondi per 17 volte nel seguente modo 0 1 2 0 1 2 0 1 2 0 1.

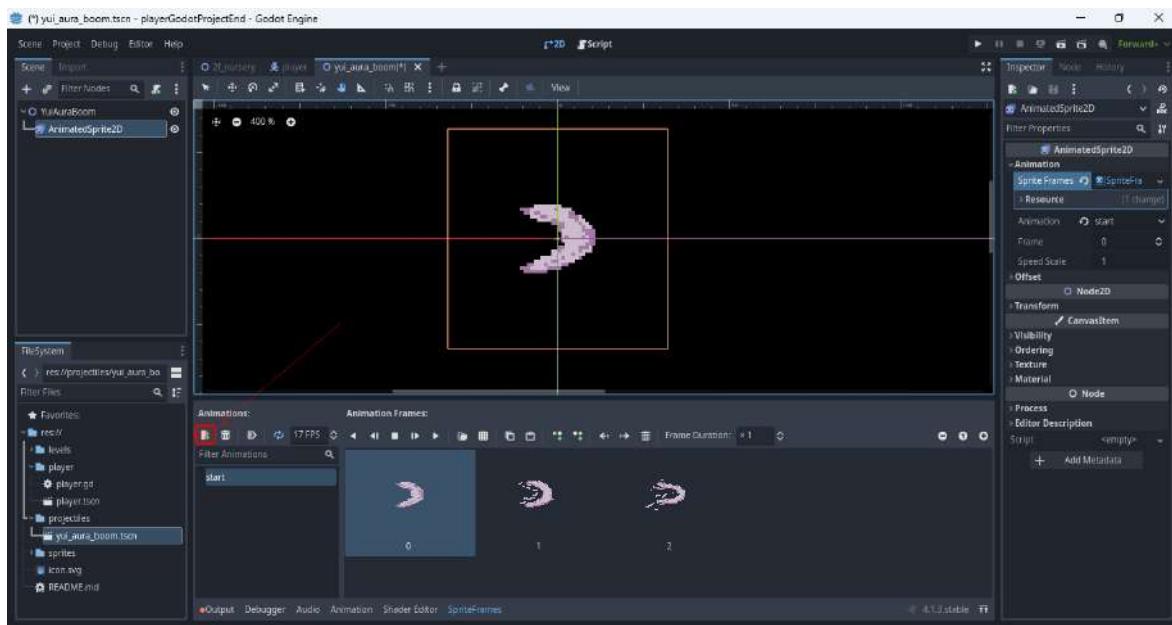


### ! Nota

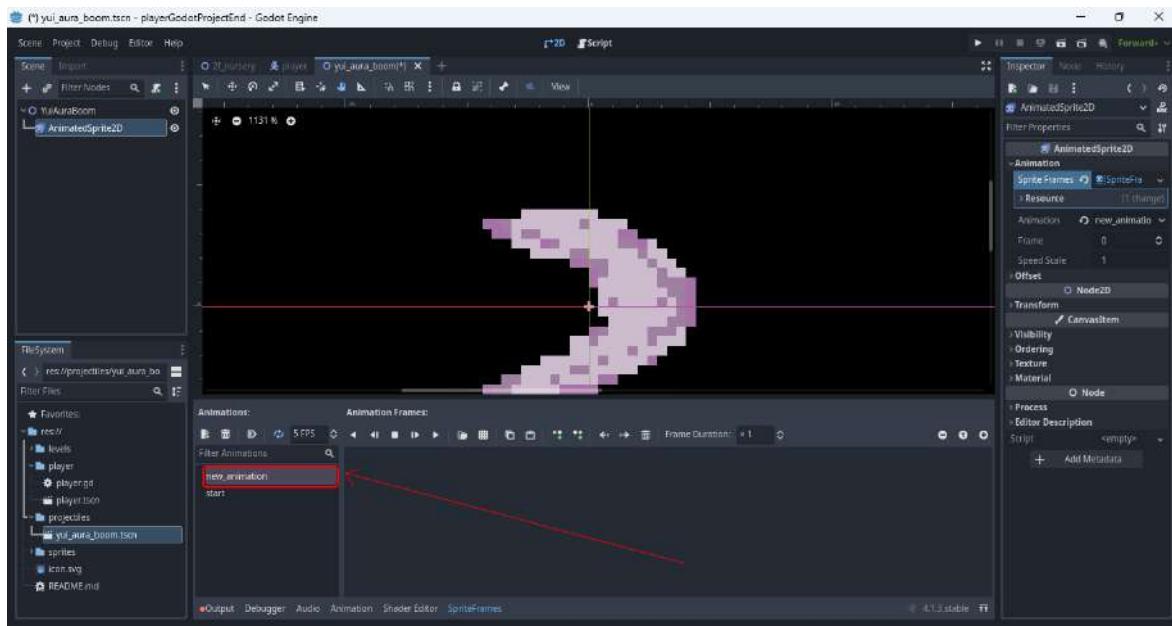
L'**FPS**, o "frame per second" ("fotogrammi al secondo"), è una misura della frequenza con cui un dispositivo (come un computer, una console di gioco o una fotocamera) è in grado di visualizzare immagini consecutive, chiamate "frame" ("fotogrammi"), in un secondo. In pratica, indica quante immagini distinte vengono mostrate sullo schermo in un singolo secondo. Le immagini in movimento che vediamo nei video o nei videogiochi sono in realtà una rapida successione di immagini statiche. Maggiore è il numero di FPS, più fluido e realistico apparirà il movimento. Un basso numero di FPS può causare un'esperienza visiva "scattosa" o "a scatti". L'FPS è particolarmente importante nei videogiochi, dove un'alta frequenza di fotogrammi può fare la differenza tra un'esperienza di gioco fluida e reattiva e una frustrante. Anche nei video, un FPS più elevato può migliorare la qualità visiva, specialmente nelle scene d'azione o nei movimenti rapidi. Nei film, lo standard è di 24 FPS, questo da un effetto cinematografico. 30 FPS è considerato il minimo per un'esperienza di gioco accettabile, 60 FPS offre un'esperienza di gioco molto fluida e piacevole, 120 FPS o superiori sono utilizzati principalmente in videogiochi competitivi o per schermi ad alta frequenza di aggiornamento. In sintesi, l'FPS è un indicatore chiave della fluidità e della qualità visiva delle immagini in movimento.

Anche se non la useremo in questa sezione, creiamo inoltre l'animazione **end**, che verrà riprodotta quando la nostra aura boom collierà con qualcosa.

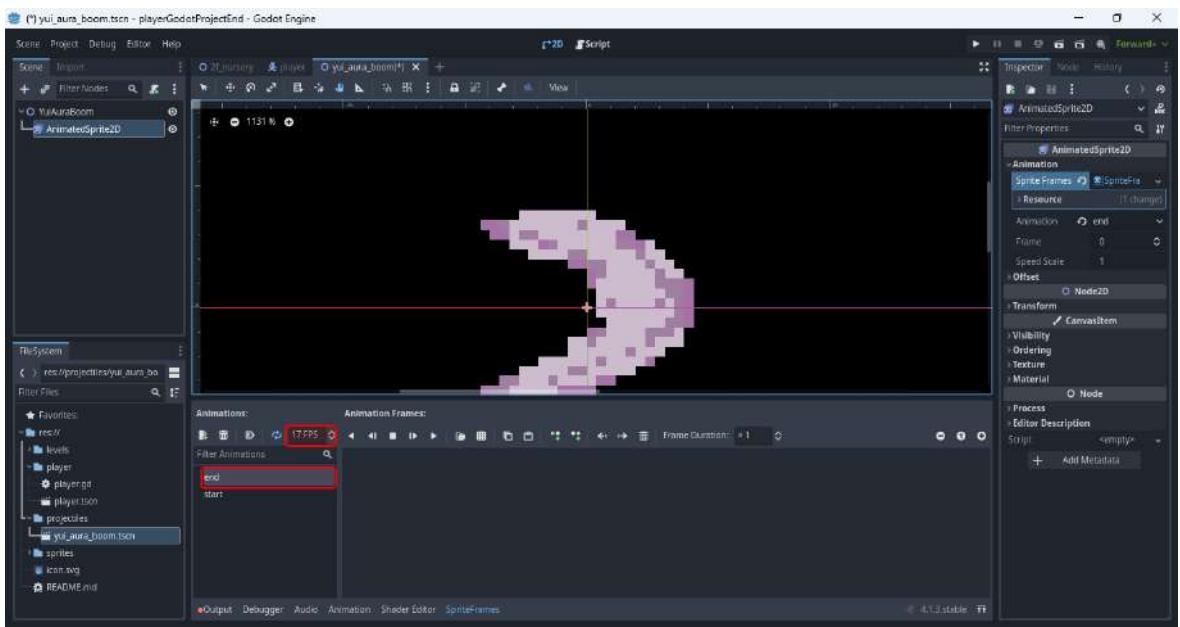
Clicchiamo sull'icona **Add Animation**



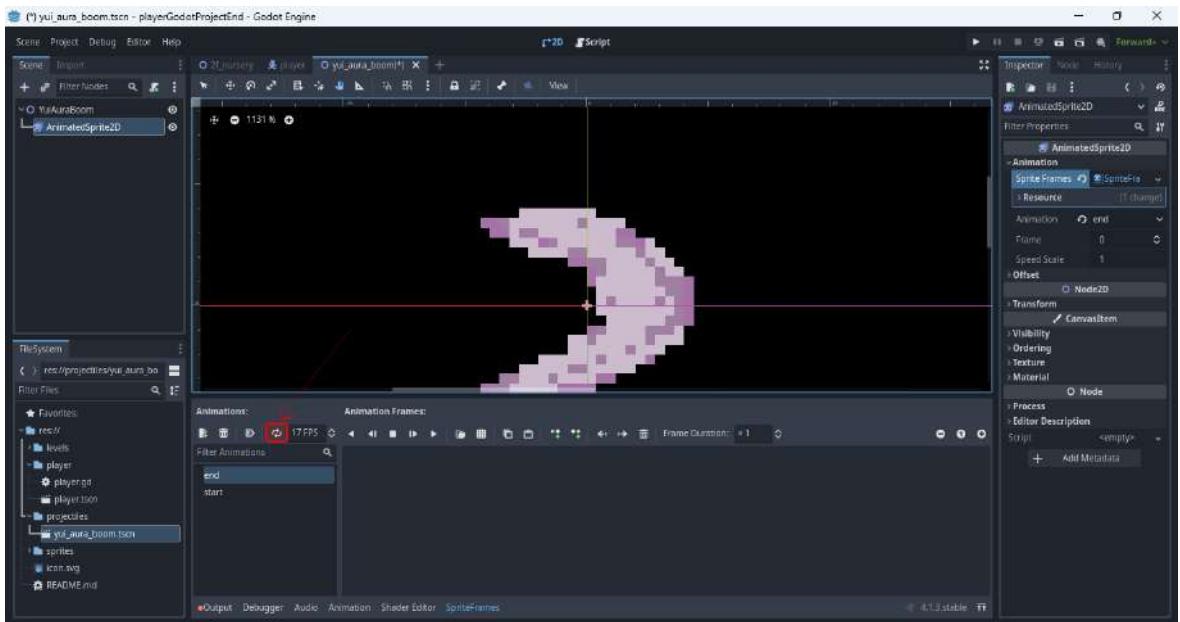
verrà creata una nuova animazione



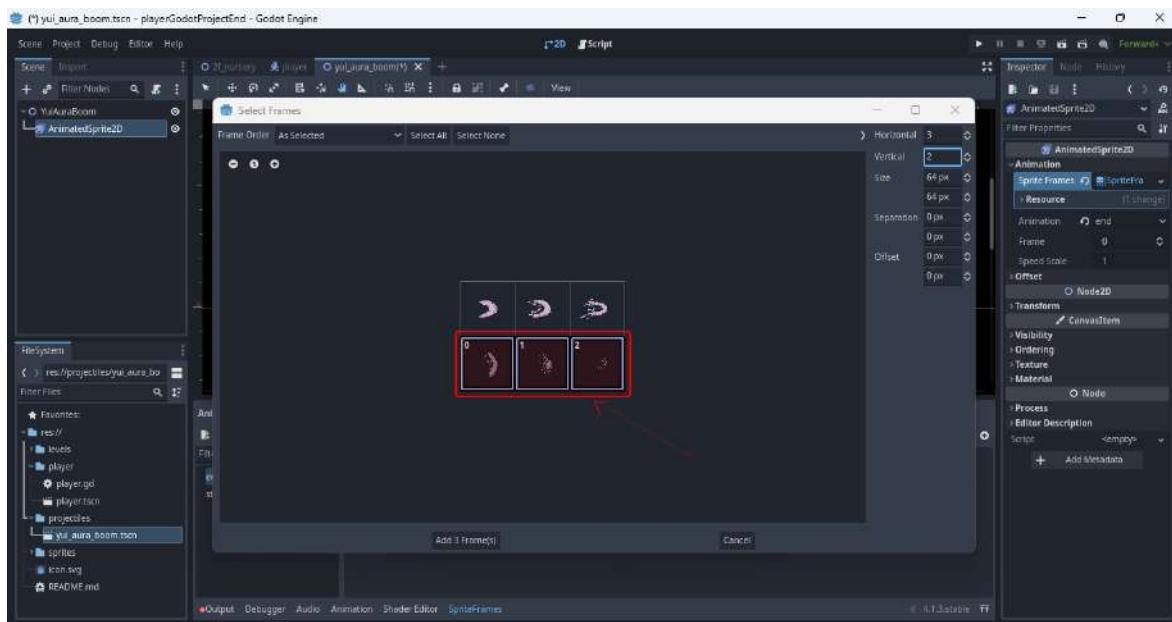
che rinominiamo **end**, impostiamo sempre a **17 FPS**



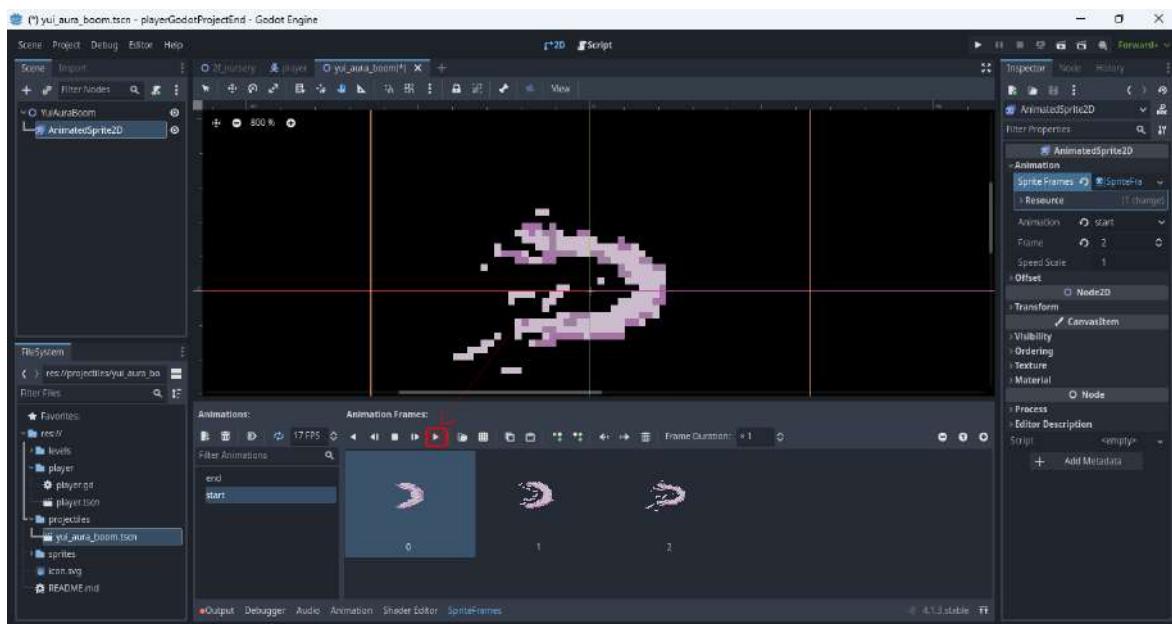
e da cui rimuoviamo il **loop**.



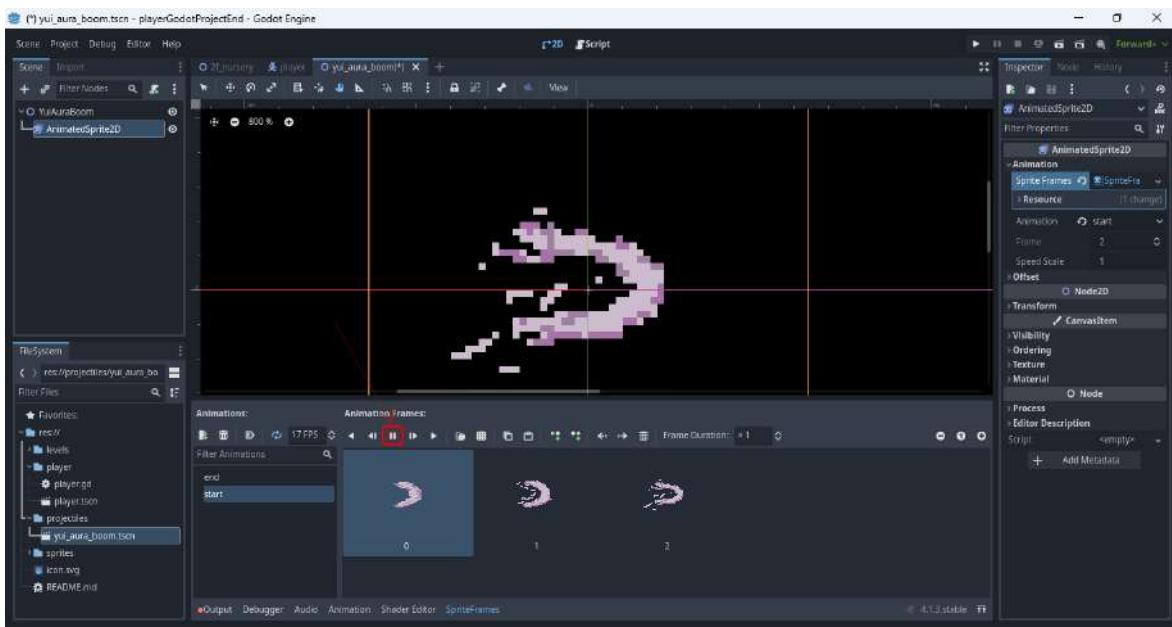
Creiamo dunque l'animazione aggiungendo i frame **3**, **4** e **5** dello spritesheet **base\_attack.png** (che Godot segna come 0, 1 e 2 perché sta contando i frame presenti nell'animazione e non dello spritesheet)



Per riprodurre una delle due animazioni basta semplicemente cliccare sul tasto **play**

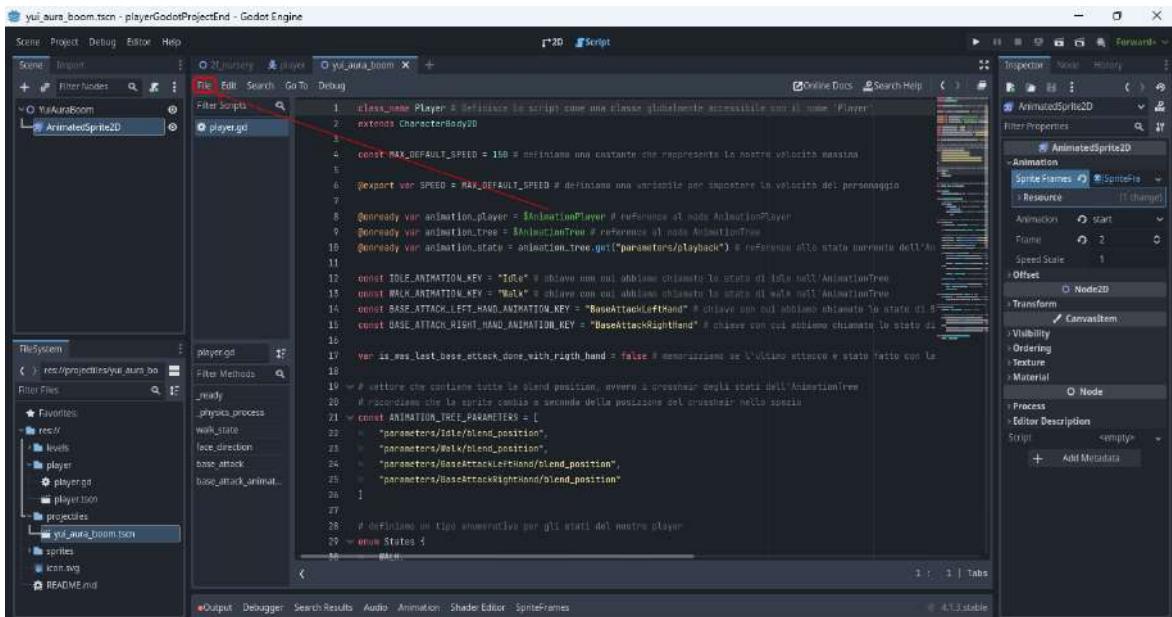


mentre per stopparla quello di **pause**

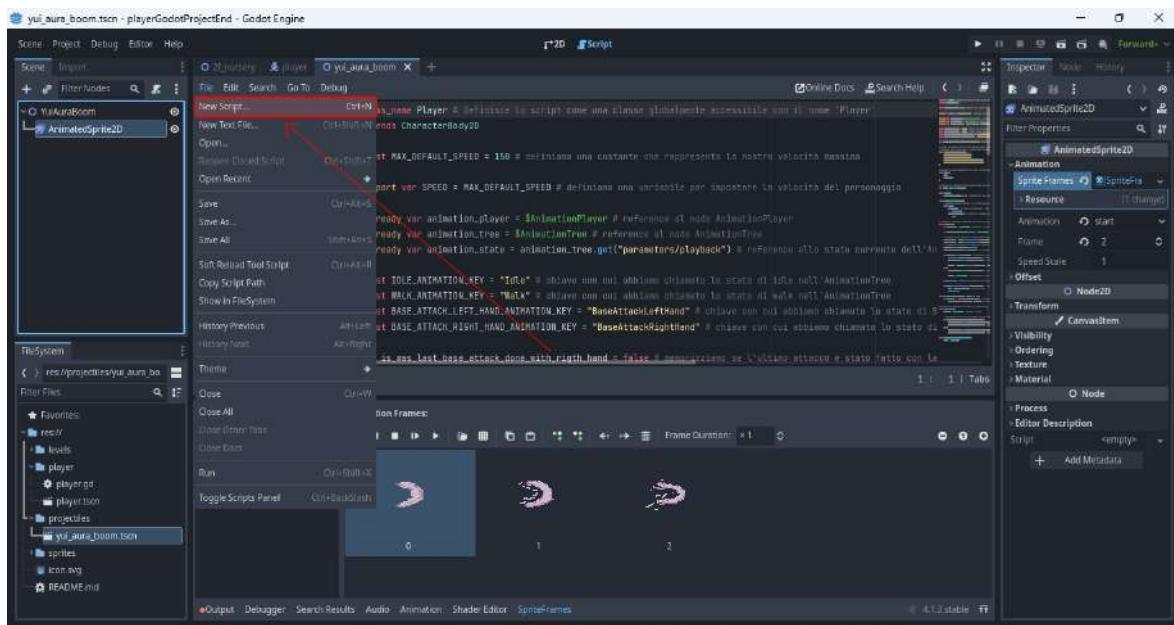


Poiché, come detto in precedenza, tutti gli oggetti che possono essere lanciati si comportano allo stesso modo, e per questo vengono raggruppati sotto il nome di **projectiles**; allora possiamo creare una classe base da far estendere a tutti gli oggetti projectile del nostro progetto.

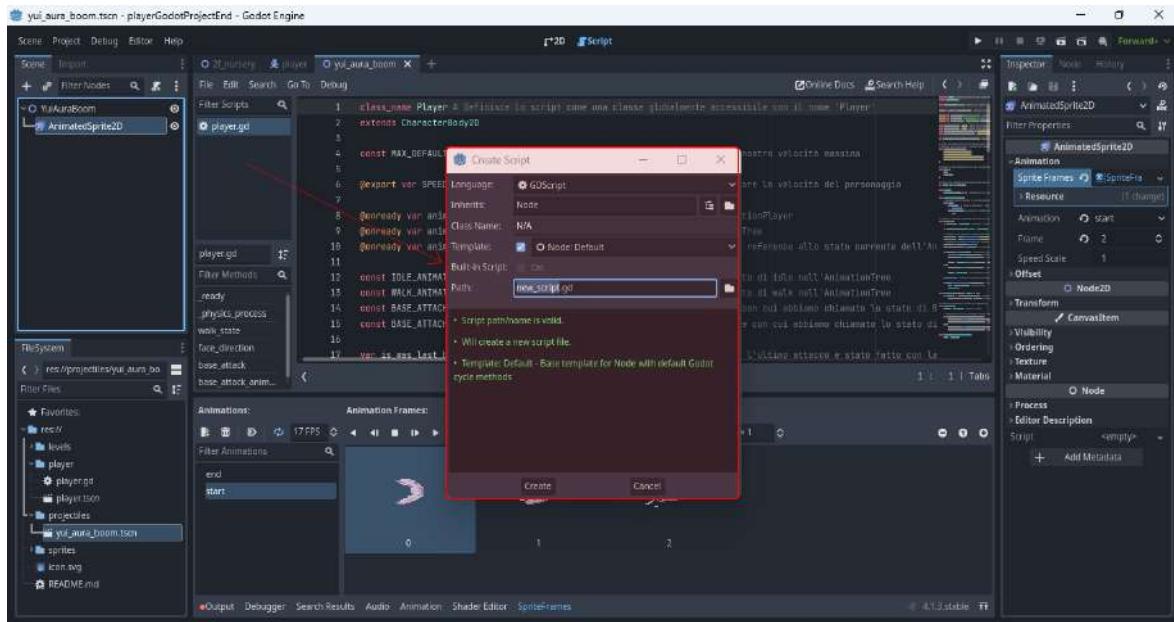
Apriamo dunque lo script editor e clicchiamo sulla voce **File**



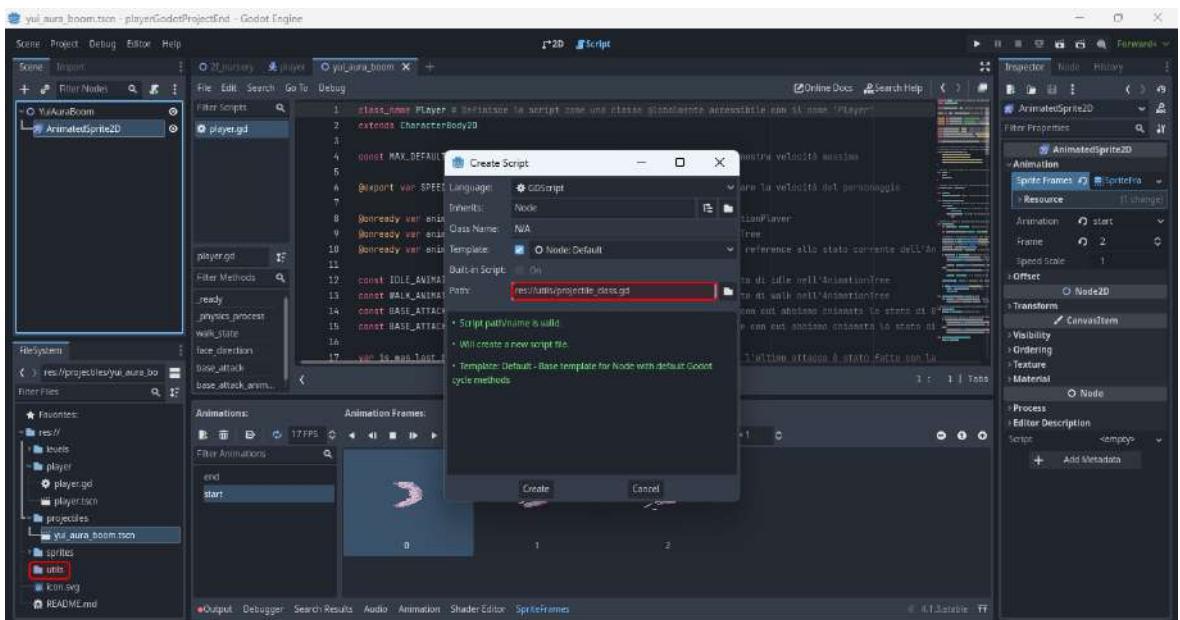
clicchiamo su **New Script...**



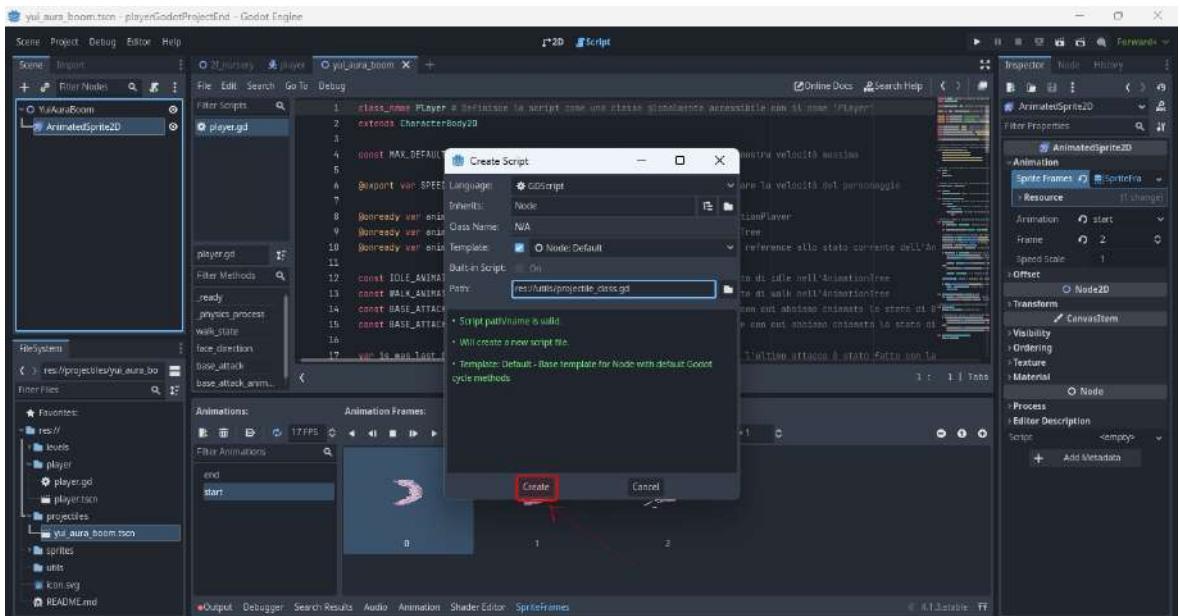
si aprirà la finestra di creazione di un nuovo script



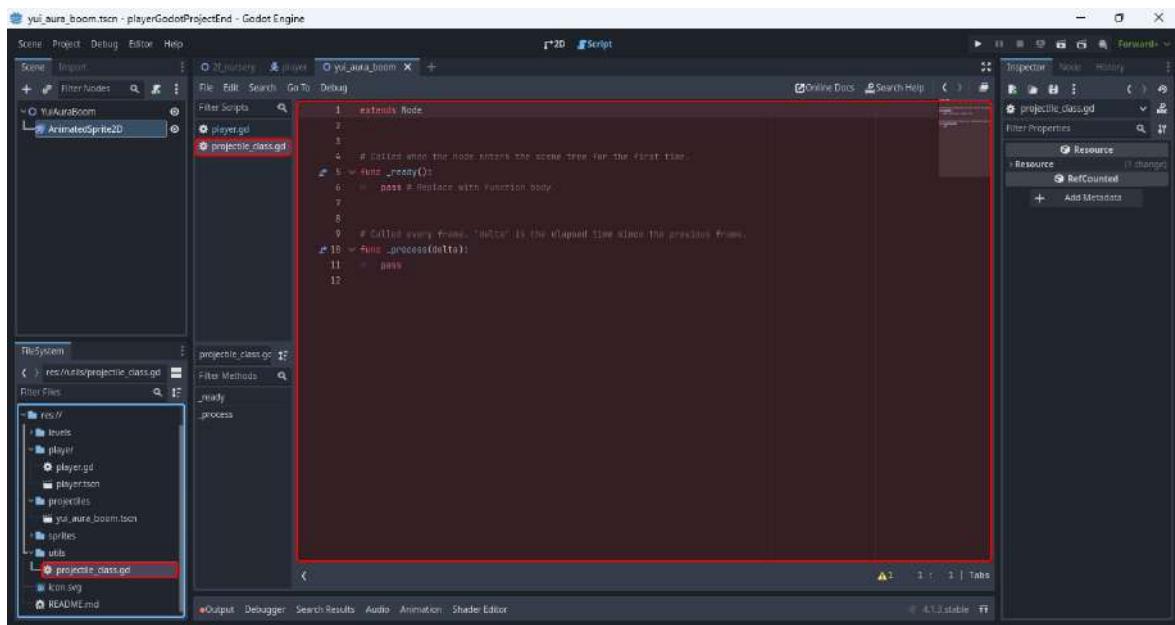
creiamo dunque un nuovo script di nome `projectile_class` all'interno della cartella `res://utils` (anch'essa da creare)



clicchiamo su **Create** per confermare la creazione



il nostro nuovo script **projectile\_class** verrà creato e aperto correttamente nello script editor



cancelliamo tutto e scriviamo dunque il codice della nostra classe **Projectile**

```

1 class_name Projectile
2 extends Node2D
3
4 @export var animated_sprite_2D: AnimatedSprite2D
5
6 @export var SPEED: int = 200
7 const START_ANIMATION_KEY = "start"
8
9 func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY)
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation)
14     global_position += SPEED * direction * delta
15

```

lì dove a riga 1 definiamo il nome della classe come **Projectile**, in modo da poter essere vista globalmente

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del pro
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

a riga 2, poiché abbiamo deciso che tutti i proiettili del nostro gioco sono in realtà dei nodi `Node2D`, come ad esempio il nodo `YuiAuraBoom`, allora facciamo estendere allo script la classe `Node2D`

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del pro
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

a riga 4 definiamo la variabile di export che conterrà il riferimento al nodo `AnimatedSprite2D`, che dunque ogni oggetto che estende da questa classe dovrà avere

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del pro
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

a riga 6 dichiariamo invece la variabile **SPEED**, che rappresenterà la velocità che avrà il nostro proiettile, e la inizializziamo a **200**

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del pro
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

a riga 7 definiamo la costante **START\_ANIMATION\_KEY** che contiene la chiave, ovvero il nome dell'animazione **start**, che abbiamo creato in precedenza nel nodo **AnimatedSprite2D**

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del proie
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

a righe 9 e 10 definiamo la funzione `_ready()` in cui facciamo semplicemente partire l'animazione `start` del nostro nodo `AnimatedSprite2D`

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del proie
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

dalla riga 12 alla 14 siamo invece dentro la funzione `_physics_process(delta)`

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del pro
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

più precisamente, a riga 13 calcoliamo la direzione che dovrà avere il nostro proiettile, ruotando il vettore `(1,0)`, ovvero `Vector2.RIGHT`, dell'angolo dato dalla rotazione del nodo stesso, contenuta dentro la variabile `rotation` della classe `Node2D`. Come vedremo tra poco, la variabile `rotation` del nostro proiettile verrà impostata dallo script del player al momento del lancio.

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del pro
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

mentre a riga 14 sommiamo alla posizione attuale del nostro proiettile un'offset dato dalla moltiplicazione tra le variabili `SPEED`, `direction` e `delta`.

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del proie
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position += SPEED * direction * delta # muoviamo il proiettile
15

```

Aggiungendo un offset alla posizione attuale del nostro proiettile ad ogni frame, esso si muoverà nella direzione fornita.

La variabile `global_position` della classe `Node2D` indica infatti la posizione globale posseduta da un nodo all'interno di una scena in ogni istante.

Il `+ =` invece indica che si sta sommando alla variabile a sinistra dell'`=`, la quantità a destra dell'`=`; dunque, riga `14` può essere riscritta nel seguente modo

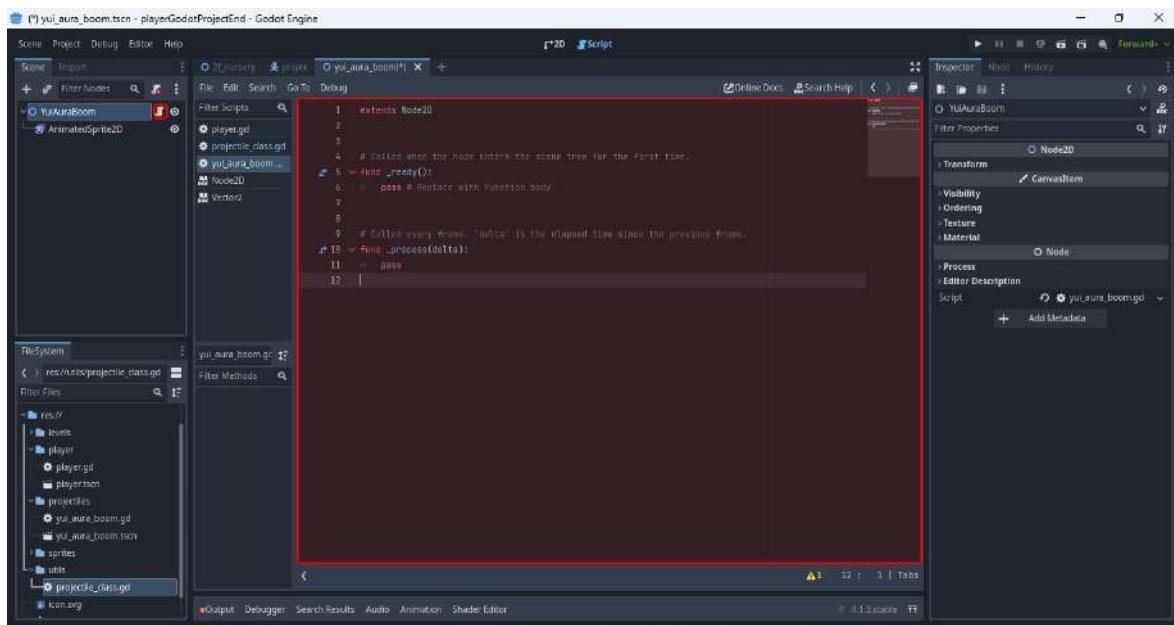
```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per
5
6  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del proie
7  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazion
8
9  func _ready():
10     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start'
11
12 func _physics_process(delta):
13     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve av
14     global_position = global_position + SPEED * direction * delta # muoviamo il proiettile
15

```

lì dove ricordiamo che, per le regole di precedenza degli operatori, vengono eseguite prima le moltiplicazioni e infine l'addizione.

A questo punto, attacchiamo un nuovo script al nodo `YuiAuraBoom`



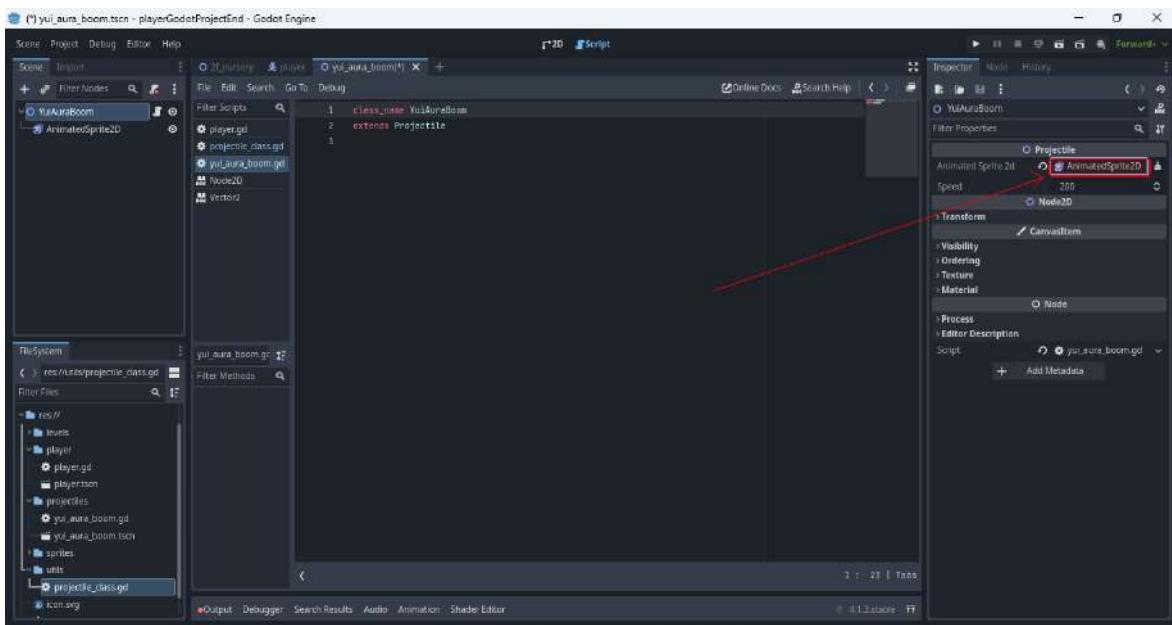
e scriviamo il seguente codice



Avendo creato la classe **Projectile**, infatti, ogni volta che creiamo un nuovo proiettile, tutto quello che dobbiamo fare è estendere la classe **Projectile** e dare un nome alla classe del nuovo proiettile. Proprio come abbiamo appena fatto per il proiettile **YuiAuraBoom**.

Poiché abbiamo dichiarato una variabile export per il nodo **AnimatedSprite2D** all'interno della classe **Projectile**, non dimentichiamo inoltre di specificarlo nell'Inspector dock del nodo **YuiAuraBoom**.

CHAPTER 2. GODOT



Terminata questa parte di preparazione del proiettile, occupiamo ora di modificare lo script del player in modo che possa gestirlo. Il codice da aggiungere è il seguente

```

16
17 const AURA_BOOM_DISTANCE_FROM_PLAYER = 20 # definiamo una costante che rappresenta la distanza ch
18
19 var player_direction: Vector2 = Vector2.DOWN # definiamo una variabile che tiene in memoria la di
20 var is_was_last_base_attack_done_with_rigth_hand = false # memorizziamo se l'ultimo attacco è sta
21
22 const YuiAuraBoom = preload("res://projectiles/yui_aura_boom.ts") # precarichiamo il nodo YuiAu
23
24
25 if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo
26     player_direction = current_player_direction
27
28     face_direction(current_player_direction) # facciamo guardare al player la direzione che d
29     animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
30
31     velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore sta effett
32     animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animazione
33     is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima volta c
34
35
36 if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi è stato
37     var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della classe Y
38     get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile YuiAuraBo
39     yui_aura_boom.global_position = self.global_position # impostiamo la posizione del proie
40     var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea per memo
41
42     match player_direction: # controlliamo in che direzione sta attualmente guardando il pl
43         case Vector2.UP:
44             yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER) # do
45             yui_aura_boom_rotation = deg_to_rad(270) # impostiamo la rotazione del proiettil
46
47         case Vector2.DOWN:
48             yui_aura_boom.global_position += Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER)
49             yui_aura_boom_rotation = deg_to_rad(90)
50
51         case Vector2.LEFT:
52             yui_aura_boom.global_position -= Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
53             yui_aura_boom_rotation = deg_to_rad(180)
54
55         case Vector2.RIGHT:
56             yui_aura_boom.global_position += Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
57             yui_aura_boom_rotation = deg_to_rad(360)
58
59     yui_aura_boom.rotation = yui_aura_boom_rotation # ruotiamo il proiettile
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118

```

lì dove a riga 17 definiamo la costante `AURA_BOOM_DISTANCE_FROM_PLAYER`, la quale rappresenta la distanza, sia essa sull'asse x o y a seconda della direzione del giocatore, che il proiettile deve avere dal player quando viene creato

```

11
12  const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'
13  const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'
14  const BASE_ATTACK_LEFT_HAND_ANIMATION_KEY = "BaseAttackLeftHand" # chiave con cui abbiamo
15  const BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY = "BaseAttackRightHand" # chiave con cui abbiam
16
17  const AURA_BOOM_DISTANCE_FROM_PLAYER = 20 # definiamo una costante che rappresenta la dist
18
19  var player_direction: Vector2 = Vector2.DOWN # definiamo una variabile che tiene in memori
20  var is_was_last_base_attack_done_with_rigth_hand = false # memorizziamo se l'ultimo attac
21
22  const YuiAuraBoom = preload("res://projectiles/yui_aura_boom.tscn") # precarichiamo il nod
23
24 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'Animat
25 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
26 const ANIMATION_TREE_PARAMETERS = [
27   "parameters/Idleblend_position",
28   "parameters/Walkblend_position",
29   "parameters/BaseAttackLeftHandblend_position",
30   "parameters/BaseAttackRightHandblend_position"
31 ]
32

```

a riga 19 definiamo la variabile `player_direction` che indicherà in ogni momento la posizione guardata dal player; e la inizializziamo a `Vector2.DOWN`

```

11
12  const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'
13  const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'
14  const BASE_ATTACK_LEFT_HAND_ANIMATION_KEY = "BaseAttackLeftHand" # chiave con cui abbiamo
15  const BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY = "BaseAttackRightHand" # chiave con cui abbiam
16
17  const AURA_BOOM_DISTANCE_FROM_PLAYER = 20 # definiamo una costante che rappresenta la dist
18
19  var player_direction: Vector2 = Vector2.DOWN # definiamo una variabile che tiene in memori
20  var is_was_last_base_attack_done_with_rigth_hand = false # memorizziamo se l'ultimo attac
21
22  const YuiAuraBoom = preload("res://projectiles/yui_aura_boom.tscn") # precarichiamo il nod
23
24 # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'Animat
25 # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
26 const ANIMATION_TREE_PARAMETERS = [
27   "parameters/Idleblend_position",
28   "parameters/Walkblend_position",
29   "parameters/BaseAttackLeftHandblend_position",
30   "parameters/BaseAttackRightHandblend_position"
31 ]
32

```

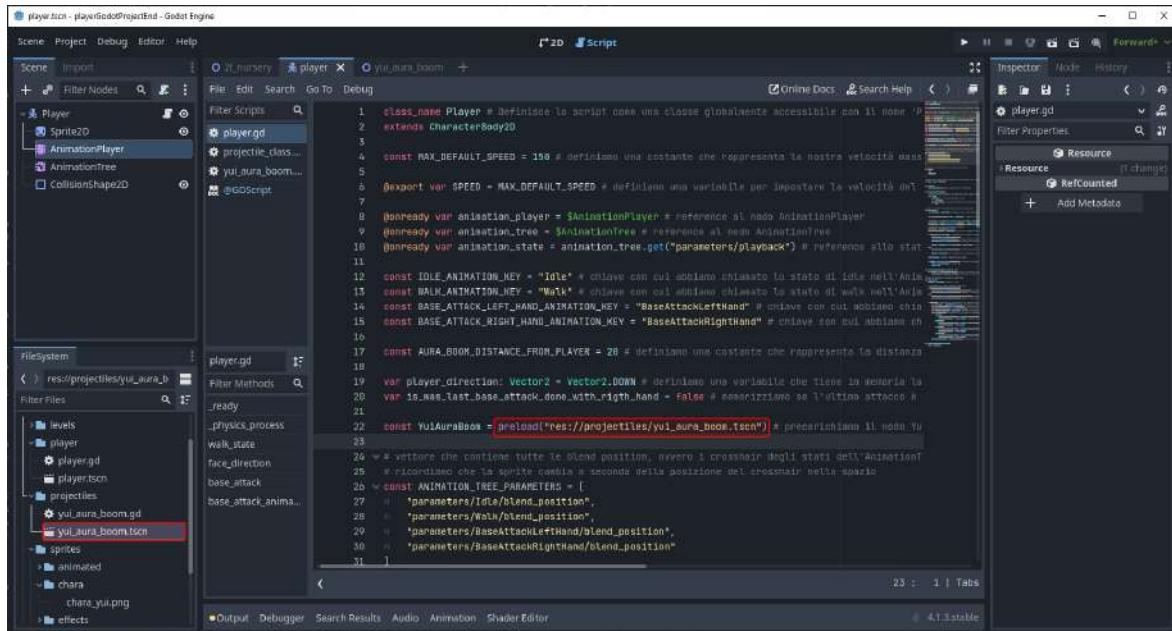
nella riga 22 precarichiamo invece la scena `YuiAuraBoom`

```

11
12  const IDLE_ANIMATION_KEY = "Idle" # chiave con cui abbiamo chiamato lo stato di idle nell'
13  const WALK_ANIMATION_KEY = "Walk" # chiave con cui abbiamo chiamato lo stato di walk nell'
14  const BASE_ATTACK_LEFT_HAND_ANIMATION_KEY = "BaseAttackLeftHand" # chiave con cui abbiamo
15  const BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY = "BaseAttackRightHand" # chiave con cui abbiam
16
17  const AURA_BOOM_DISTANCE_FROM_PLAYER = 20 # definiamo una costante che rappresenta la dist
18
19  var player_direction: Vector2 = Vector2.DOWN # definiamo una variabile che tiene in memori
20  var is_was_last_base_attack_done_with_rigth_hand = false # memorizziamo se l'ultimo attac
21
22  const YuiAuraBoom = preload("res://projectiles/yui_aura_boom.tscn") # precarichiamo il nod
23
24  # vettore che contiene tutte le blend position, ovvero i crosshair degli stati dell'Animat
25  # ricordiamo che la sprite cambia a seconda della posizione del crosshair nello spazio
26  const ANIMATION_TREE_PARAMETERS = [
27    "parameters/Idleblend_position",
28    "parameters/Walkblend_position",
29    "parameters/BaseAttackLeftHandblend_position",
30    "parameters/BaseAttackRightHandblend_position"
31  ]
32

```

Si può evitare di scrivere a mano la parte del `preload`, tracinando e rilasciando il file della scena `yui_aura_boom` dal file system dock allo script editor, il tutto tenendo sempre premuto `CTRL`



a riga 67 prendiamo la direzione attuale del player dalla variabile `current_player_direction` (e dunque da `input_vector`) e la copiamo dentro `player_direction` in modo da poterla accedere globalmente

```

52.
53. # funzione che gestisce le operazioni dello stato WALK del nostro player
54. func walk_state():
55.     var input_vector = Vector2.ZERO # impostiamo a ZERO il valore iniziale del nostro vettore
56. 
57.     var current_player_direction: Vector2 = Vector2.ZERO # impostiamo a ZERO il valore iniziale della direzione
58. 
59.     # calcoliamo in che direzione il giocatore vuole fare muovere il personaggio
60.     input_vector.x = Input.get_action_strength("ui_right") - Input.get_action_strength("ui_left")
61.     input_vector.y = Input.get_action_strength("ui_down") - Input.get_action_strength("ui_up")
62. 
63.     # normalizziamo il vettore per ottenere una direzione
64.     input_vector = input_vector.normalized()
65. 
66.     if input_vector != Vector2.ZERO: # controlliamo che il giocatore sta effettivamente muovendo
67.         player_direction = current_player_direction
68.         face_direction(current_player_direction) # facciamo guardare al player la direzione
69.         animation_state.travel(WALK_ANIMATION_KEY) # passiamo dallo stato "Idle" allo stato "Walk"
70. 
71.         velocity = velocity.move_toward(input_vector * SPEED, SPEED) # se il giocatore si muove
72.     else:
73.         animation_state.travel(IDLE_ANIMATION_KEY) # passiamo dallo stato "Walk" allo stato "Idle"

```

a riga 99 controlliamo prima di tutto che la scena **YuiAuraBoom** sia stata correttamente precaricata

```

86.
87. # gestisce il comportamento del player per l'attacco base
88. func base_attack():
89.     velocity = Vector2.ZERO # fermiamo il player
90. 
91.     # controlliamo con che mano ha attaccato l'ultima volta
92.     if is_was_last_base_attack_done_with_rigth_hand:
93.         animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animazione
94.         is_was_last_base_attack_done_with_rigth_hand = false # ci ricordiamo che l'ultima volta
95.     else:
96.         animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animazione
97.         is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima volta
98. 
99.     if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi lo istanziamo
100.        var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della classe YuiAuraBoom
101.        get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile Yui Aura Boom alla scena
102.        yui_aura_boom.global_position = self.global_position # impostiamo la posizione del proiettile
103.        var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea per la rotazione
104.        match player_direction: # controlliamo in che direzione sta attualmente guardando
105.            case Vector2.UP:
106.                yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
107.                yui_aura_boom_rotation = deg_to_rad(270) # impostiamo la rotazione del proiettile

```

una volta aver verificato ciò, a riga 100 istanziamo (ovvero creiamo un oggetto) la classe **YuiAuraBoom**

```

86
87  # gestisce il comportamento del player per l'attacco base
88  func base_attack():
89      velocity = Vector2.ZERO # fermiamo il player
90
91      # controlliamo con che mano ha attaccato l'ultima volta
92      if is_was_last_base_attack_done_with_righthand:
93          animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'anim
94          is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima
95      else:
96          animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'ani
97          is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima
98
99      if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi
100         var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della c
101         get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile Yui
102         yui_aura_boom.global_position = self.global_position # impostiamo la posizione de
103         var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea p
104         match player_direction: # controlliamo in che direzione sta attualmente guardando
105             Vector2.UP:
106                 yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYE
107                 yui_aura_boom.rotation = deg_to_rad(270) # impostiamo la rotazione del pi

```

a riga 101 aggiungiamo il proiettile alla scena corrente il cui si trova il player.

```

86
87  # gestisce il comportamento del player per l'attacco base
88  func base_attack():
89      velocity = Vector2.ZERO # fermiamo il player
90
91      # controlliamo con che mano ha attaccato l'ultima volta
92      if is_was_last_base_attack_done_with_righthand:
93          animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'anim
94          is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima
95      else:
96          animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'ani
97          is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima
98
99      if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi
100         var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della c
101         get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile Yui
102         yui_aura_boom.global_position = self.global_position # impostiamo la posizione de
103         var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea p
104         match player_direction: # controlliamo in che direzione sta attualmente guardando
105             Vector2.UP:
106                 yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYE
107                 yui_aura_boom.rotation = deg_to_rad(270) # impostiamo la rotazione del pi

```

La funzione `get_tree()` ritorna infatti lo SceneTree in cui è presente il nodo da cui è chiamato (in questo caso `Player`); `current_scene` è una variabile dell'oggetto `SceneTree` che contiene un oggetto `Node` che rappresenta la scena corrente, ovvero, la

scena che è attiva sullo schermo; infine, il metodo `add_child(node, force_readable_name, internal)` della classe `Node`, aggiunge un nodo figlio al nodo da cui è chiamato (in questo caso il nodo `current_scene`).

A riga 102 impostiamo momentaneamente la posizione del proiettile a quella del player. In questo istante, dunque, player e proiettile sono sovrapposti.

```
86
87  # gestisce il comportamento del player per l'attacco base
88  func base_attack():
89      velocity = Vector2.ZERO # fermiamo il player
90
91      # controlliamo con che mano ha attaccato l'ultima volta
92      if is_was_last_base_attack_done_with_rigth_hand:
93          animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animazione di attacco con la mano sinistra
94          is_was_last_base_attack_done_with_rigth_hand = false # ci ricordiamo che l'ultima volta abbiamo attaccato con la mano sinistra
95      else:
96          animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animazione di attacco con la mano destra
97          is_was_last_base_attack_done_with_rigth_hand = true # ci ricordiamo che l'ultima volta abbiamo attaccato con la mano destra
98
99      if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi possiamo usarlo
100         var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della classe YuiAuraBoom
101         get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile Yui Aura Boom alla scena attuale
102         yui_aura_boom.global_position = self.global_position # impostiamo la posizione del proiettile
103         var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea per la rotazione del proiettile
104         match player_direction: # controlliamo in che direzione sta attualmente guardando il player
105             Vector2.UP:
106                 yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER)
107                 yui_aura_boom.rotation = deg_to_rad(270) # impostiamo la rotazione del proiettile
```

riga 103 vede invece l'inizializzazione della variabile temporanea `yui_aura_boom_rotation`, la quale rappresenterà la rotazione che daremo al proiettile.

```

86
87  # gestisce il comportamento del player per l'attacco base
88  func base_attack():
89      velocity = Vector2.ZERO # fermiamo il player
90
91      # controlliamo con che mano ha attaccato l'ultima volta
92      if is_was_last_base_attack_done_with_righthand:
93          animation_state.travel(BASE_ATTACK_LEFT_HAND_ANIMATION_KEY) # riproduciamo l'animazione
94          is_was_last_base_attack_done_with_righthand = false # ci ricordiamo che l'ultima volta
95      else:
96          animation_state.travel(BASE_ATTACK_RIGHT_HAND_ANIMATION_KEY) # riproduciamo l'animazione
97          is_was_last_base_attack_done_with_righthand = true # ci ricordiamo che l'ultima volta
98
99      if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi è stato
100         var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della classe Yui
101         get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile Yui
102         yui_aura_boom.global_position = self.global_position # impostiamo la posizione del proiettile
103         var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea per memorizzare la rotazione
104         match player_direction: # controlliamo in che direzione sta attualmente guardando il player
105             Vector2.UP:
106                 yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER) # disegniamo il proiettile verso l'alto
107                 yui_aura_boom.rotation = deg_to_rad(270) # impostiamo la rotazione del proiettile
108             Vector2.DOWN:
109                 yui_aura_boom.global_position += Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER)
110                 yui_aura_boom.rotation = deg_to_rad(90)
111             Vector2.LEFT:
112                 yui_aura_boom.global_position -= Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
113                 yui_aura_boom.rotation = deg_to_rad(180)
114             Vector2.RIGHT:
115                 yui_aura_boom.global_position += Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
116                 yui_aura_boom.rotation = deg_to_rad(360)
117
118         yui_aura_boom.rotation = yui_aura_boom.rotation # ruotiamo il proiettile
119  func base_attack_animation_finished():

```

`deg_to_rad(deg)` è un metodo Global Scope che converte l'angolo dato in pasto alla funzione da gradi a radianti.

Il match statement che va dalla riga 104 alla 116, controlla la direzione del player e ruota e posiziona il proiettile in base ad essa.

```

98
99      if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi è stato
100         var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della classe Yui
101         get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile Yui
102         yui_aura_boom.global_position = self.global_position # impostiamo la posizione del proiettile
103         var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea per memorizzare la rotazione
104         match player_direction: # controlliamo in che direzione sta attualmente guardando il player
105             Vector2.UP:
106                 yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER) # disegniamo il proiettile verso l'alto
107                 yui_aura_boom.rotation = deg_to_rad(270) # impostiamo la rotazione del proiettile
108             Vector2.DOWN:
109                 yui_aura_boom.global_position += Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER)
110                 yui_aura_boom.rotation = deg_to_rad(90)
111             Vector2.LEFT:
112                 yui_aura_boom.global_position -= Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
113                 yui_aura_boom.rotation = deg_to_rad(180)
114             Vector2.RIGHT:
115                 yui_aura_boom.global_position += Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
116                 yui_aura_boom.rotation = deg_to_rad(360)
117
118         yui_aura_boom.rotation = yui_aura_boom.rotation # ruotiamo il proiettile
119  func base_attack_animation_finished():

```

Ad esempio, se il player sta guardando verso sinistra (righe dalla 111 alla 113),

spostiamo il proiettile di un offset `AURA_BOOM_DISTANCE_FROM_PLAYER` verso sinistra (riga 112) e impostiamo a `180°` la variabile `yui_aura_boom_rotation` (riga 113)

```

98
99  if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi è stato
100   var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della classe Yui
101   get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile YuiAuraBoo
102   yui_aura_boom.global_position = self.global_position # impostiamo la posizione del proie
103   var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea per memo
104   match player_direction: # controlliamo in che direzione sta attualmente guardando il pla
105     Vector2.UP:
106       yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER) # di
107       yui_aura_boom_rotation = deg_to_rad(270) # impostiamo la rotazione del proiettile
108     Vector2.DOWN:
109       yui_aura_boom.global_position += Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER)
110       yui_aura_boom_rotation = deg_to_rad(90)
111     Vector2.LEFT:
112       yui_aura_boom.global_position -= Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
113       yui_aura_boom_rotation = deg_to_rad(180)
114     Vector2.RIGHT:
115       yui_aura_boom.global_position += Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
116       yui_aura_boom_rotation = deg_to_rad(360)
117       yui_aura_boom.rotation = yui_aura_boom_rotation # ruotiamo il proiettile
118
119 func base_attack_animation_finished():

```

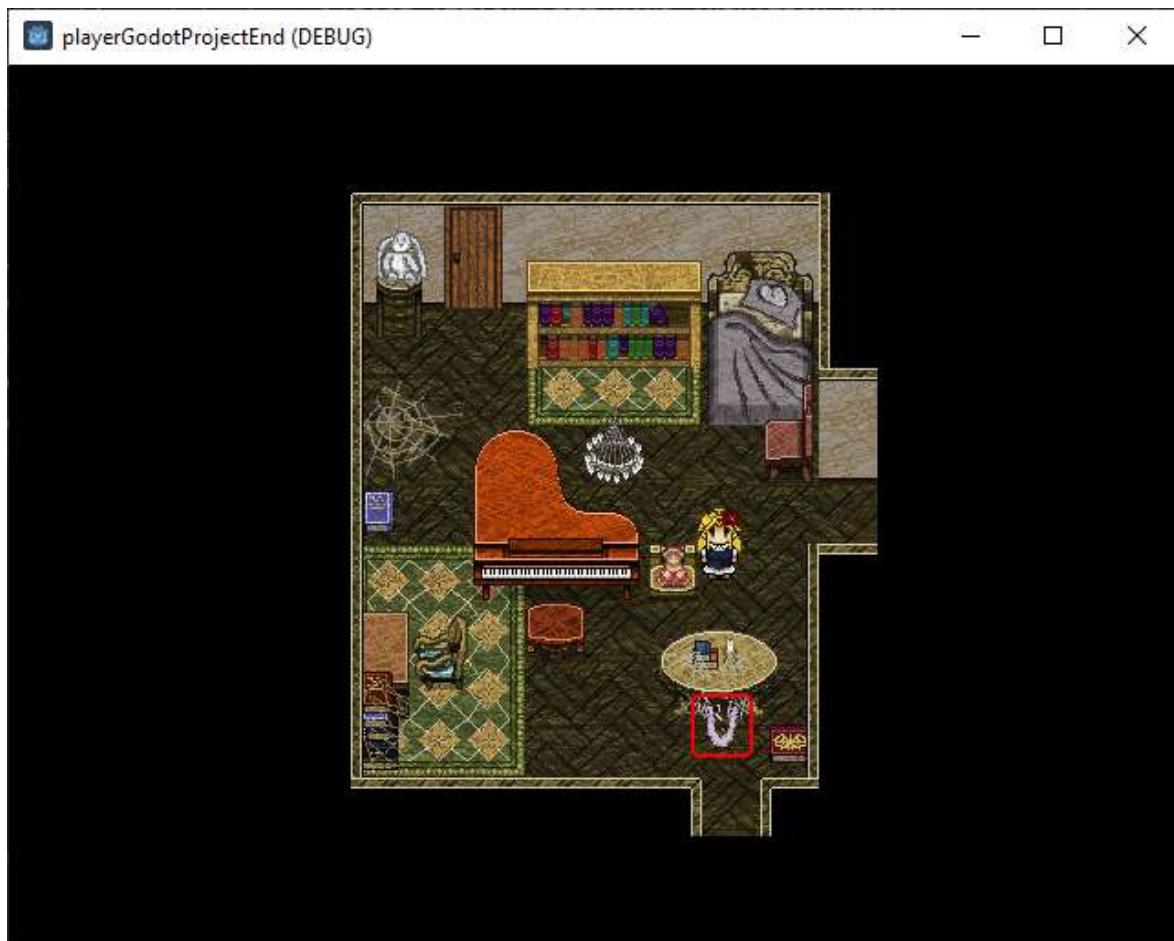
a riga 117, ruotiamo infine il proiettile dei *radiani* specificati in `yui_aura_boom_rotation`, in modo da farlo guardare nella giusta direzione

```

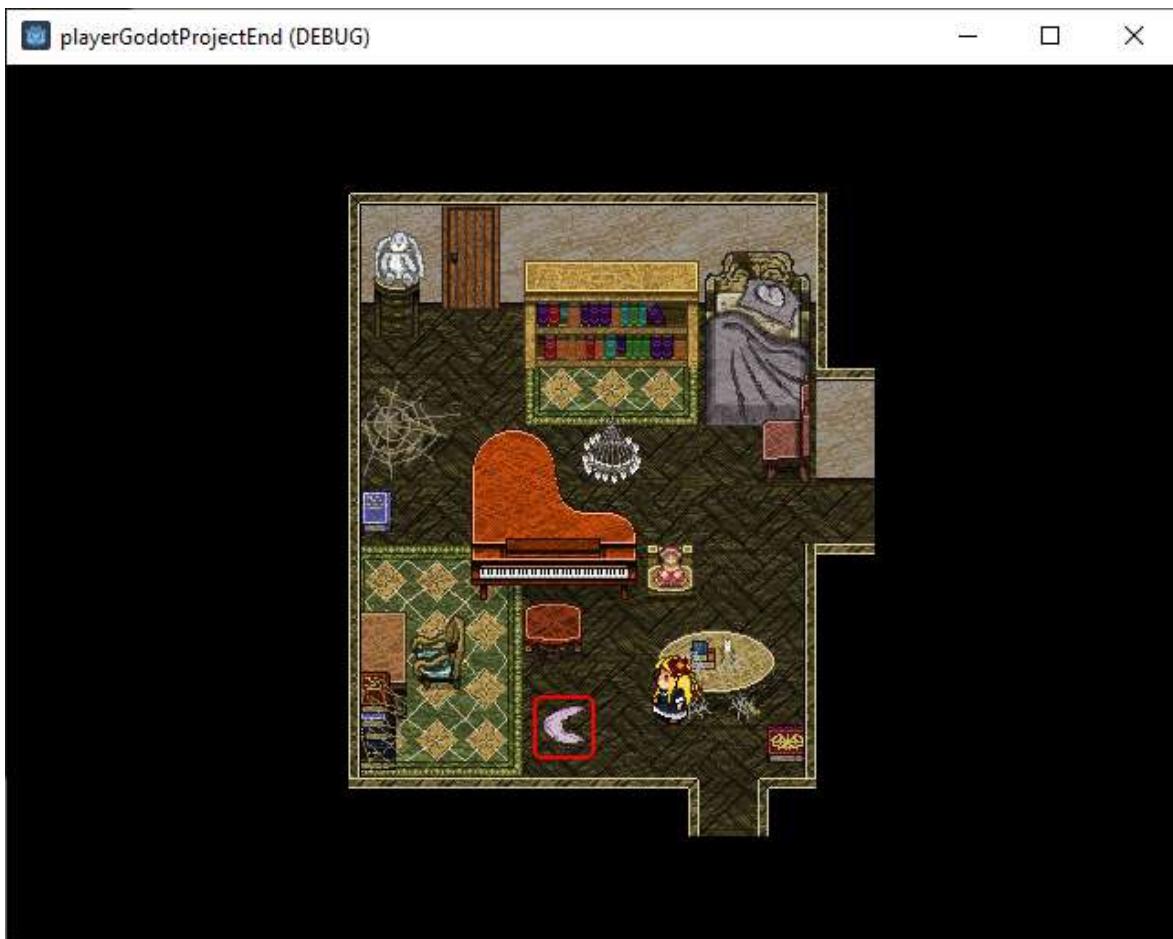
98
99  if YuiAuraBoom: # controlliamo se effettivamente il nodo YuiAuraBoom esiste e quindi è stato
100   var yui_aura_boom = YuiAuraBoom.instantiate() # istanziamo un nuovo nodo della classe Yui
101   get_tree().current_scene.add_child(yui_aura_boom) # aggiungiamo il proiettile YuiAuraBoo
102   yui_aura_boom.global_position = self.global_position # impostiamo la posizione del proie
103   var yui_aura_boom_rotation = deg_to_rad(0) # definiamo una variabile temporanea per memo
104   match player_direction: # controlliamo in che direzione sta attualmente guardando il pla
105     Vector2.UP:
106       yui_aura_boom.global_position -= Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER) # di
107       yui_aura_boom_rotation = deg_to_rad(270) # impostiamo la rotazione del proiettile
108     Vector2.DOWN:
109       yui_aura_boom.global_position += Vector2(0, AURA_BOOM_DISTANCE_FROM_PLAYER)
110       yui_aura_boom_rotation = deg_to_rad(90)
111     Vector2.LEFT:
112       yui_aura_boom.global_position -= Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
113       yui_aura_boom_rotation = deg_to_rad(180)
114     Vector2.RIGHT:
115       yui_aura_boom.global_position += Vector2(AURA_BOOM_DISTANCE_FROM_PLAYER, 0)
116       yui_aura_boom_rotation = deg_to_rad(360)
117       yui_aura_boom.rotation = yui_aura_boom_rotation # ruotiamo il proiettile
118
119 func base_attack_animation_finished():

```

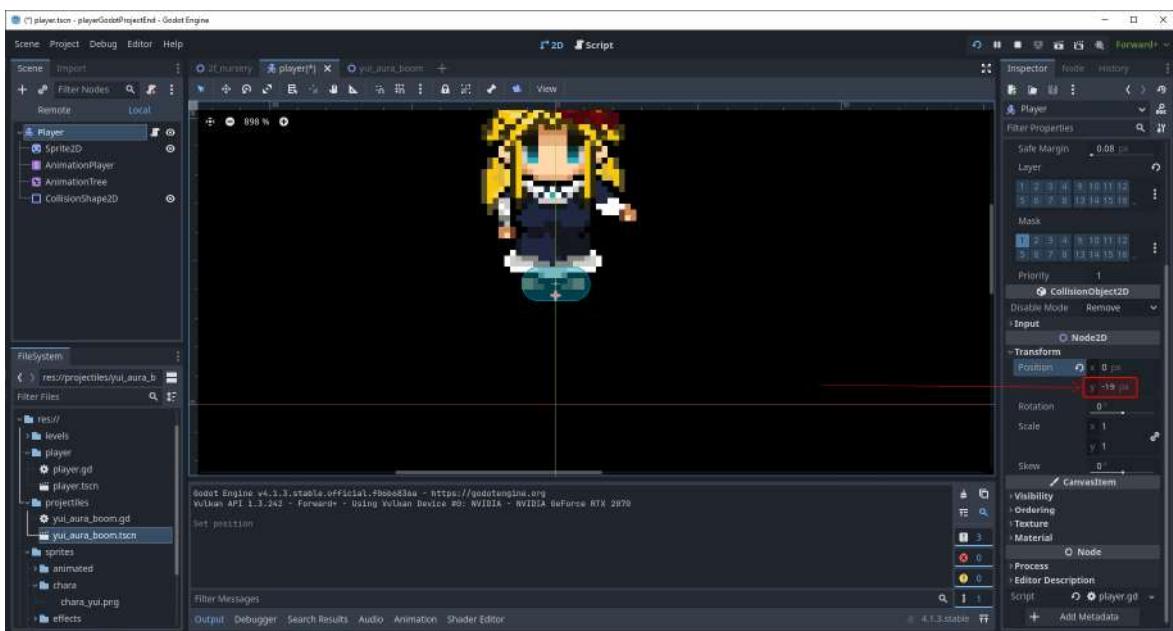
Provando il nostro gioco, il player sarà ora in grado di lanciare le "aura boom"



Se proviamo un attacco laterale, notiamo però che c'è un problema con la posizione del proiettile, il quale è leggermente più in basso rispetto al player

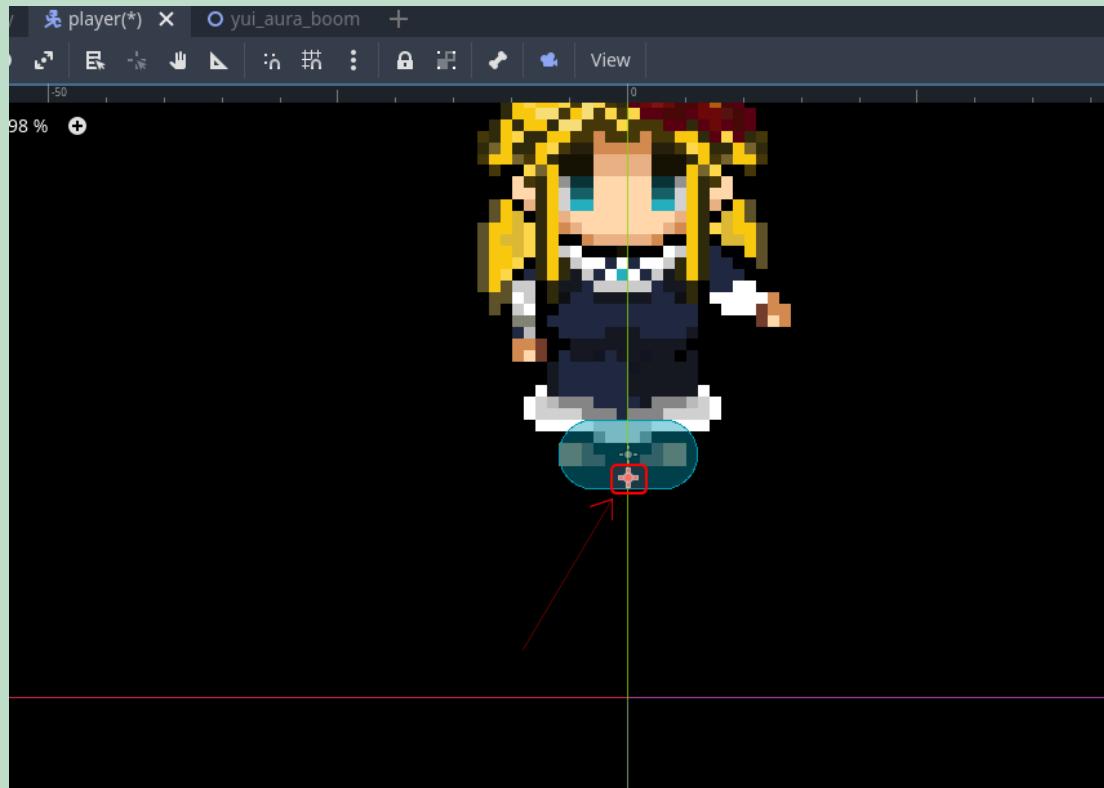


Per risolvere questo problema possiamo adottare moltissime strategie diverse. Quella che abbiamo scelto noi, anche in vista dei futuri argomenti che andremo a trattare, è spostare la posizione del nodo **Player**. Impostiamo dunque a **-19** la **y** della voce **Position** sotto **Transform**

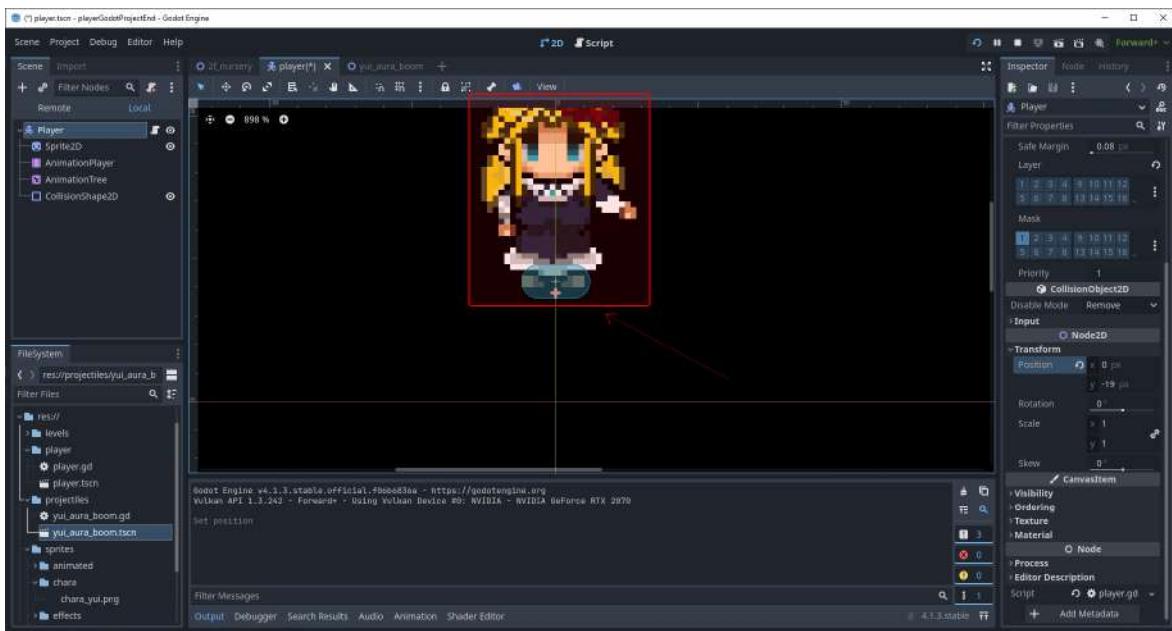


**! Consiglio**

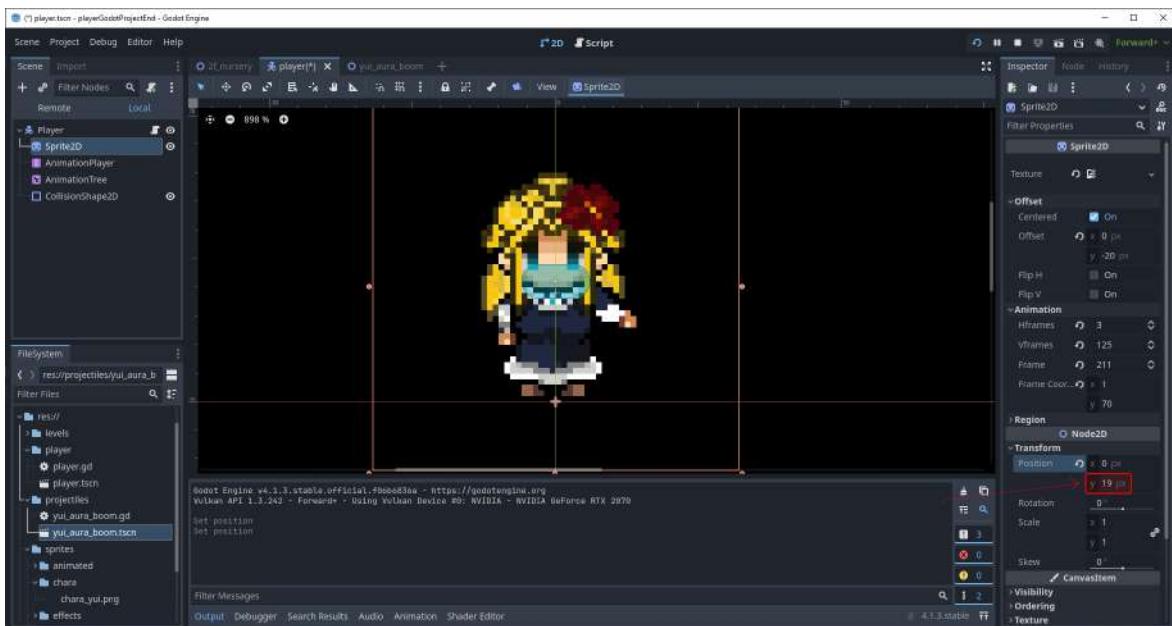
Ricordiamo che il nodo **Player** è il puntino rosso da cui vengono calcolate le posizioni di tutti i nodi figlio



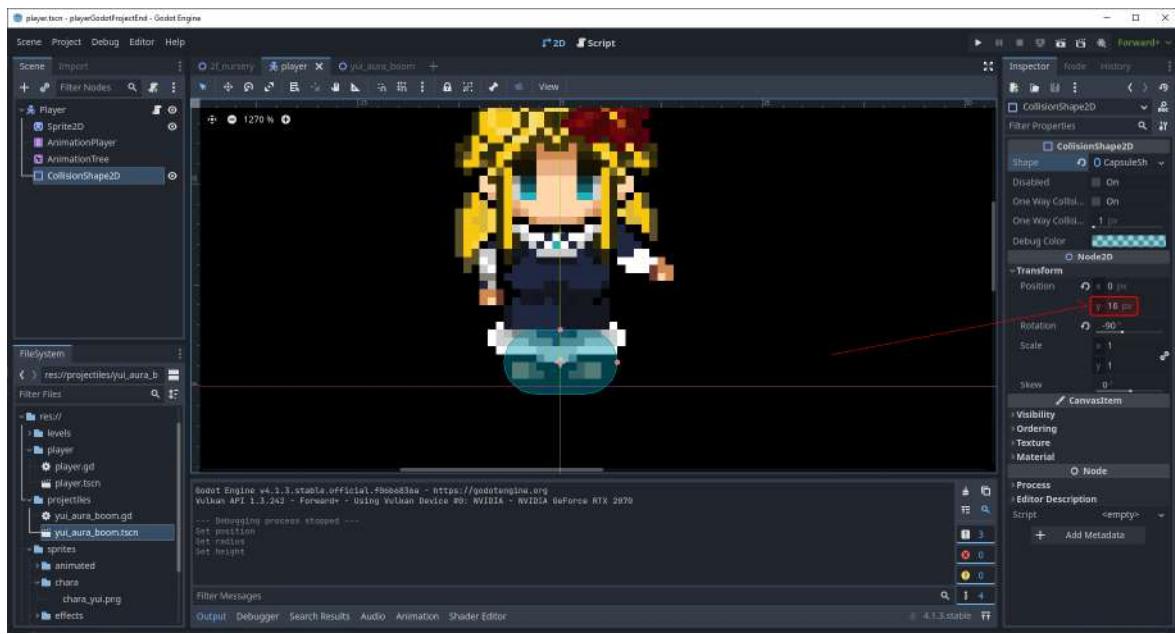
Così facendo però, come possiamo vedere, la nostra sprite verrà spostata a sua volta rispetto alla linea d'origine



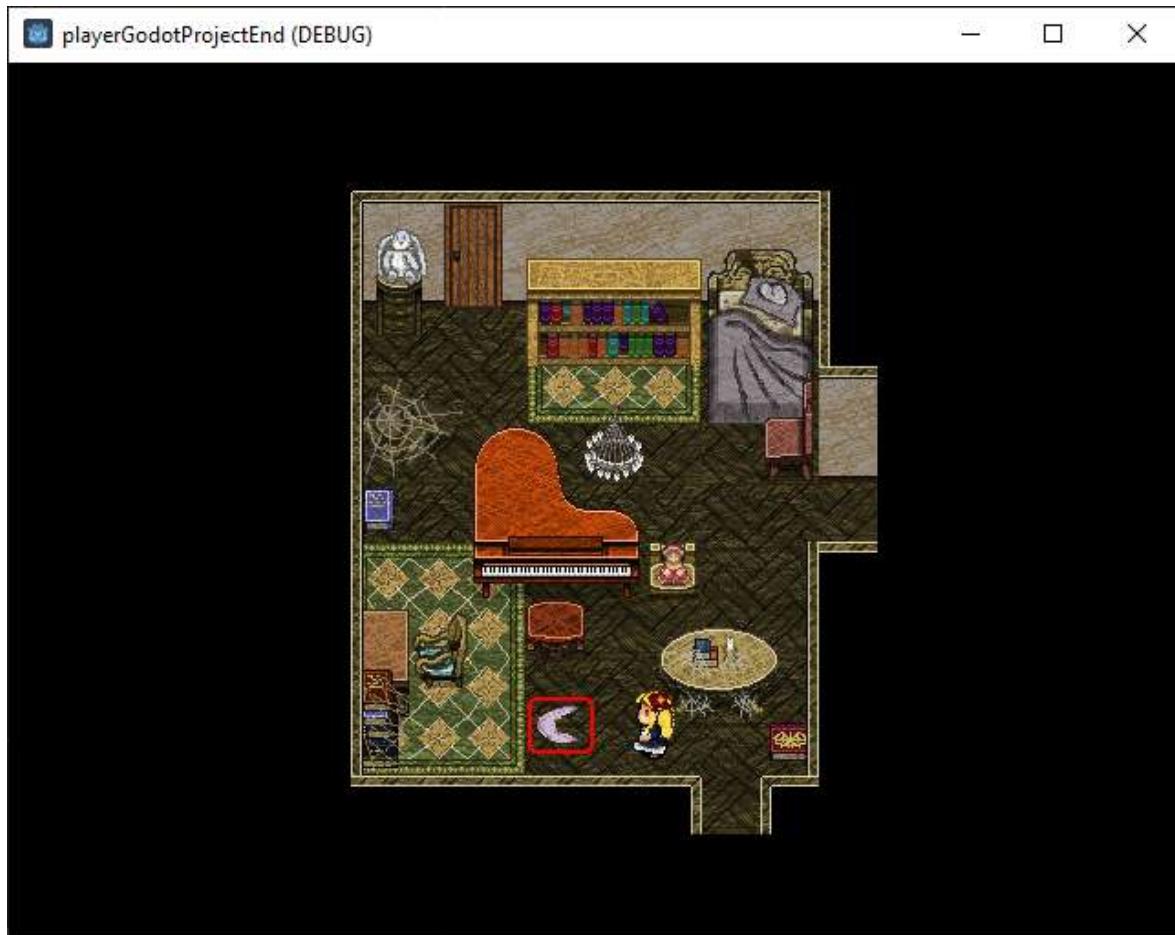
Poiché, anche spostando la posizione del nodo **Player**, la sprite del personaggio deve comunque essere posizionata sopra la linea di origine; impostiamo dunque a **19** la **y** della voce **Position** sotto **Transform** del nodo **Sprite2D**



Stesso discorso dicasi per il nodo **CollisionShape2D**. Impostiamo a **16** la sua **y** della voce **Position** sotto **Transform**

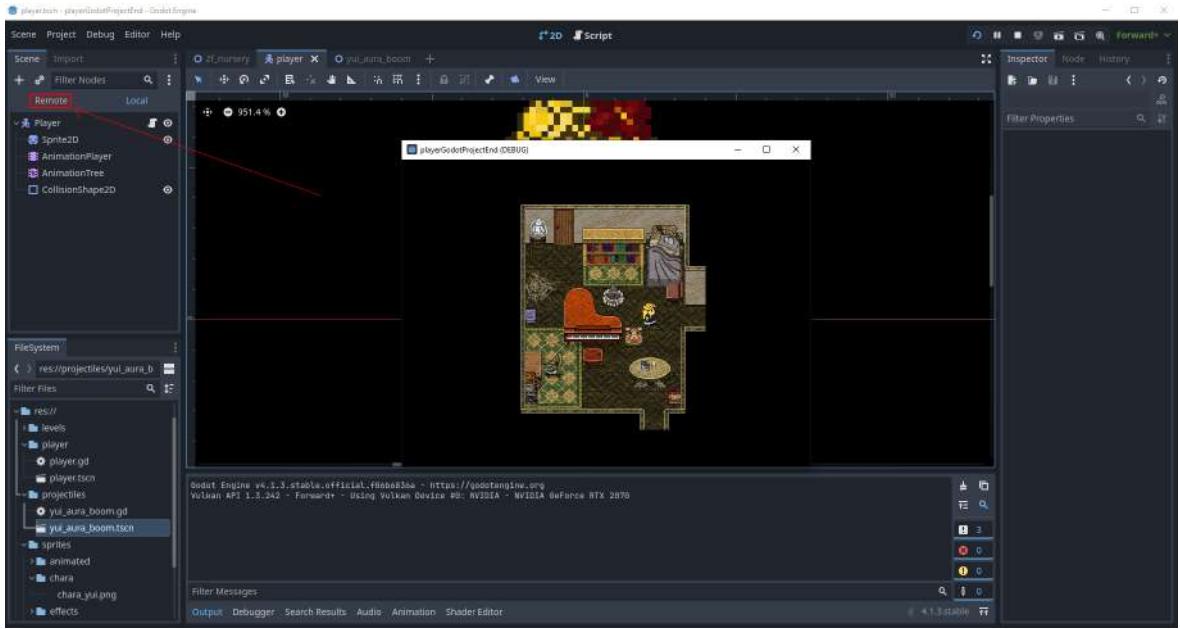


Il proiettile verrà ora creato nella giusta posizione

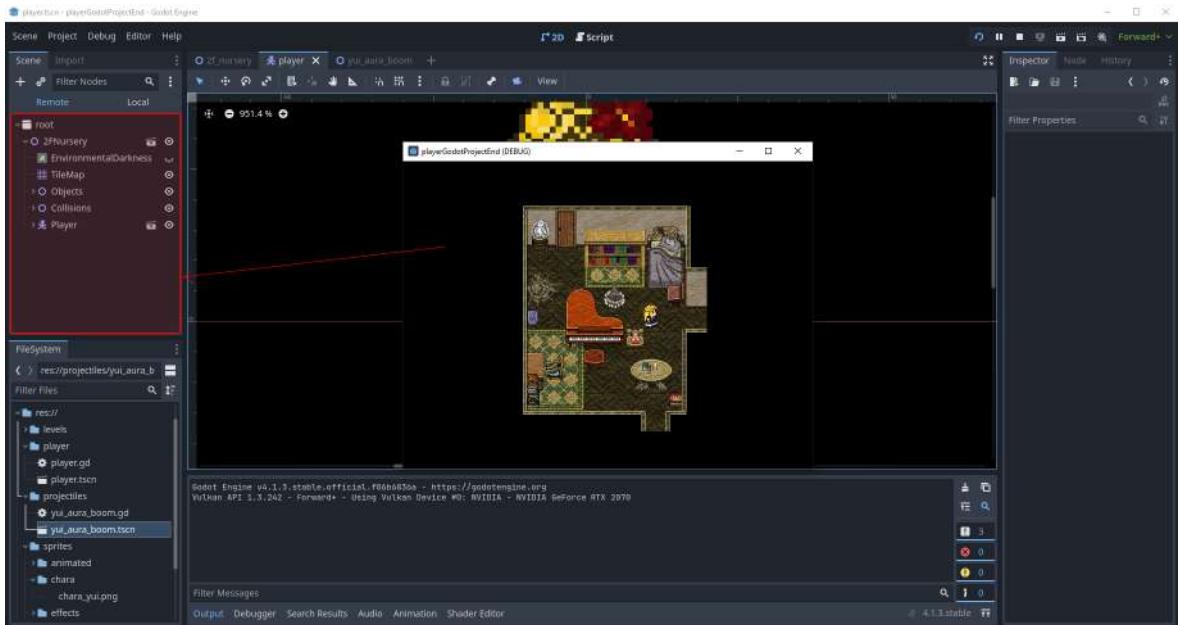


L'ultimo problema da considerare per i proiettili è un qualcosa di un po' subdolo. Quando noi creiamo un oggetto **YuiAuraBoom**, questo si muove fino ad uscire fuori dal

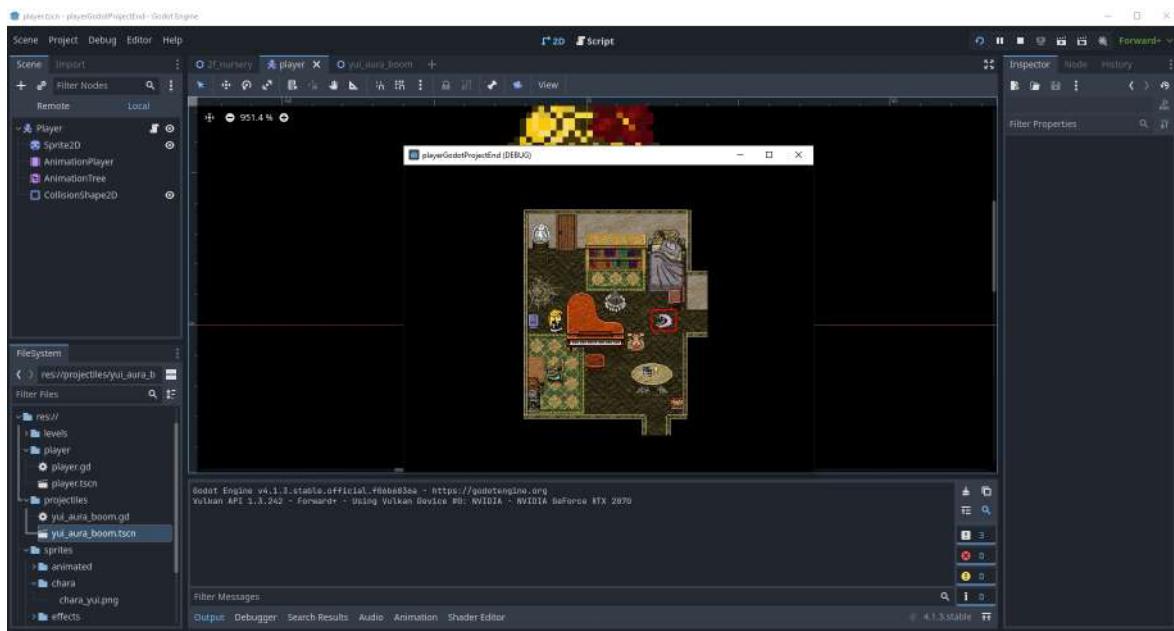
campo visivo del player... ma poi? Cosa fa? Che succede a questo nodo? Esso continua a muoversi nella parte di schermo che non vediamo. Per verificarlo infatti, con il gioco aperto, clicchiamo sulla voce **Remote**



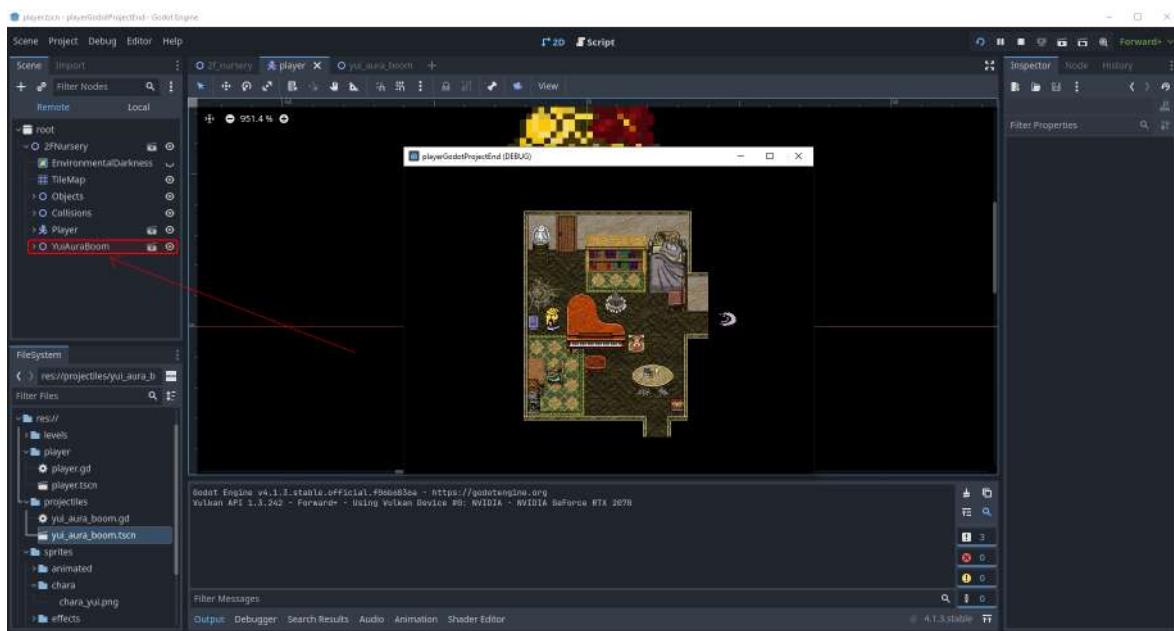
si aprirà una schermata con tutti i nodi attualmente attivi del videogioco, e dunque anche quelli globali non direttamente presenti (figli) nella scena che stiamo visualizzando



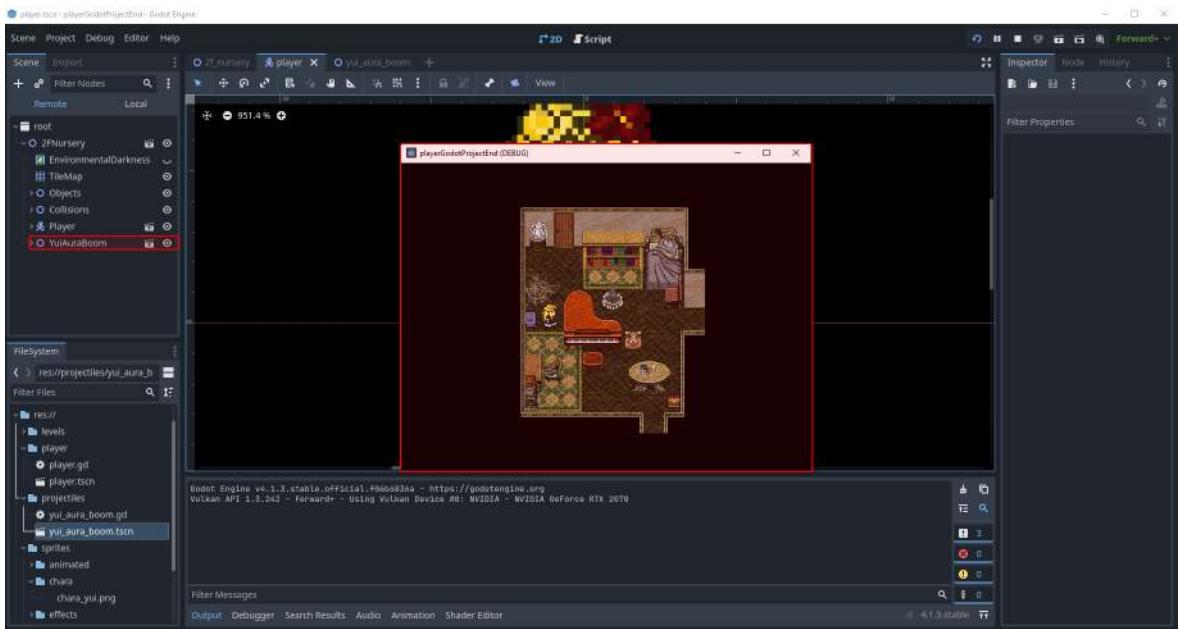
se adesso creassimo un proiettile



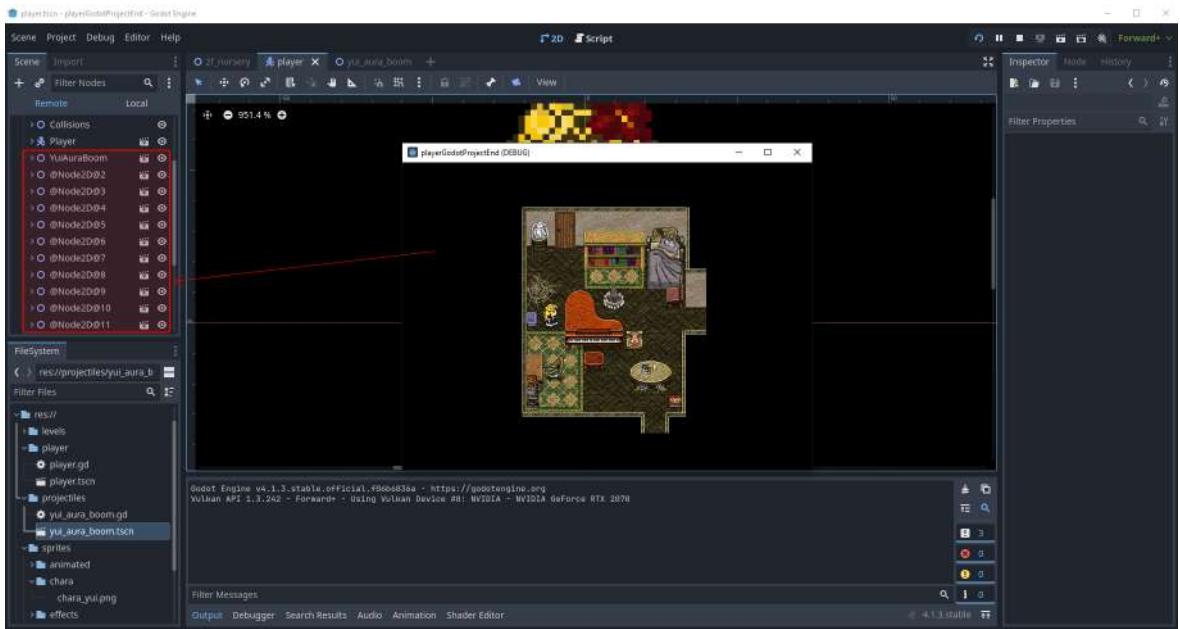
potremmo notare dalla finestra come sia stato creato il nodo corrispondente a questo proiettile all'interno della scena



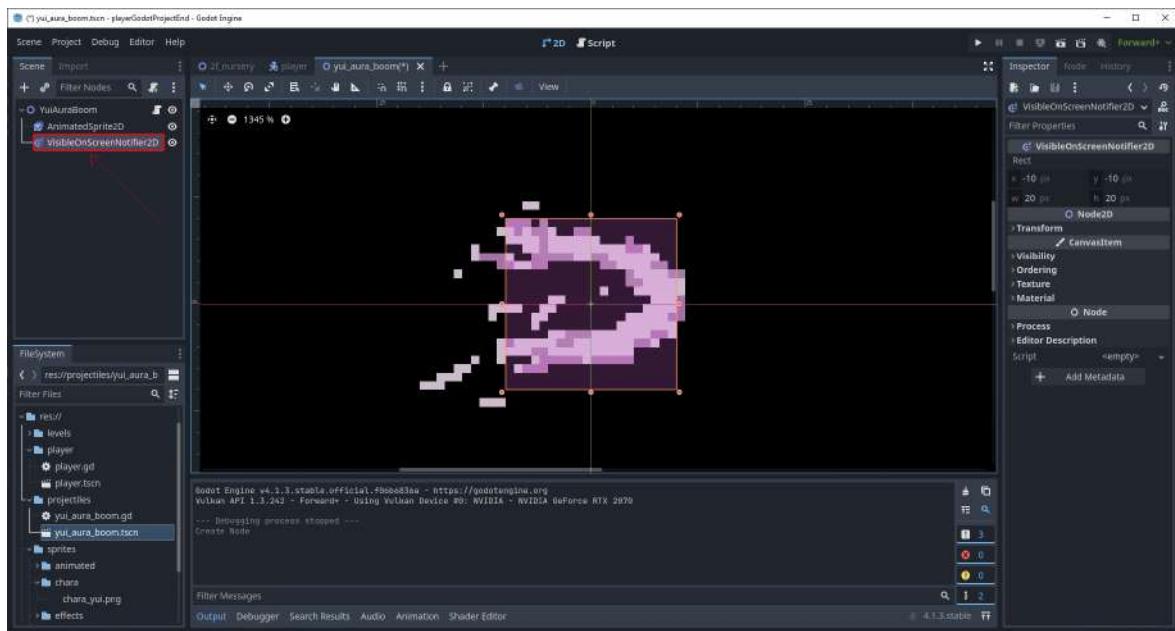
Questo nodo rimarrà nella scena anche dopo che il proiettile non sarà più visibile



Poiché, quando si parla di un numero esorbitante di proiettili, questo potrebbe sfociare in problemi di performance



per fare in modo che ogni proiettile venga distrutto non appena fuori dal campo visivo del giocatore, come prima cosa, aggiugiamo un nodo `VisibleOnScreenNotifier2D` al nostro nodo `YuiAuraBoom`



e modifichiamo la classe **Projectile**

```

1  class_name Projectile
2  extends Node2D
3
4  @export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export per il nodo Anim
5  @export var visible_on_screen_notitifier_2D: VisibleOnScreenNotifier2D # definiamo una variabile di e
6
7  @export var SPEED: int = 200 # definiamo una variabile di export per la velocità del proiettile
8  const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazione 'start' de
9
10 func _ready():
11     animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione 'start' del nostro p
12
13     visible_on_screen_notitifier_2D.screen_exited.connect(queue_free) # quando il proiettile esce dal
14
15 func _physics_process(delta):
16     var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che deve avere il proie
17     global_position += SPEED * direction * delta # muoviamo il proiettile
18

```

lì dove a riga 5 definiamo una variabile di export per avere sempre in memoria il riferimento del nuovo nodo **VisibleOnScreenNotifier2D**

```

1  |class_name Projectile
2  |extends Node2D
3
4  |@export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export
5  |@export var visible_on_screen_notifier_2D: VisibleOnScreenNotifier2D # definiamo una variabile di export
6
7  |@export var SPEED: int = 200 # definiamo una variabile di export per la velocità del proiettile
8  |const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazione
9
10 |func _ready():
11    |>    animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione
12    |
13    |>    visible_on_screen_notifier_2D.screen_exited.connect(queue_free) # quando il proiettile esce dallo schermo lo rimuoviamo
14
15 |func _physics_process(delta):
16    |>    var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che muove il proiettile
17    |>    global_position += SPEED * direction * delta # muoviamo il proiettile
18

```

mentre a riga 13 collegiamo il segnale `screen_exited` della classe `VisibleOnScreenNotifier2D` con la funzione `queue_free()`

```

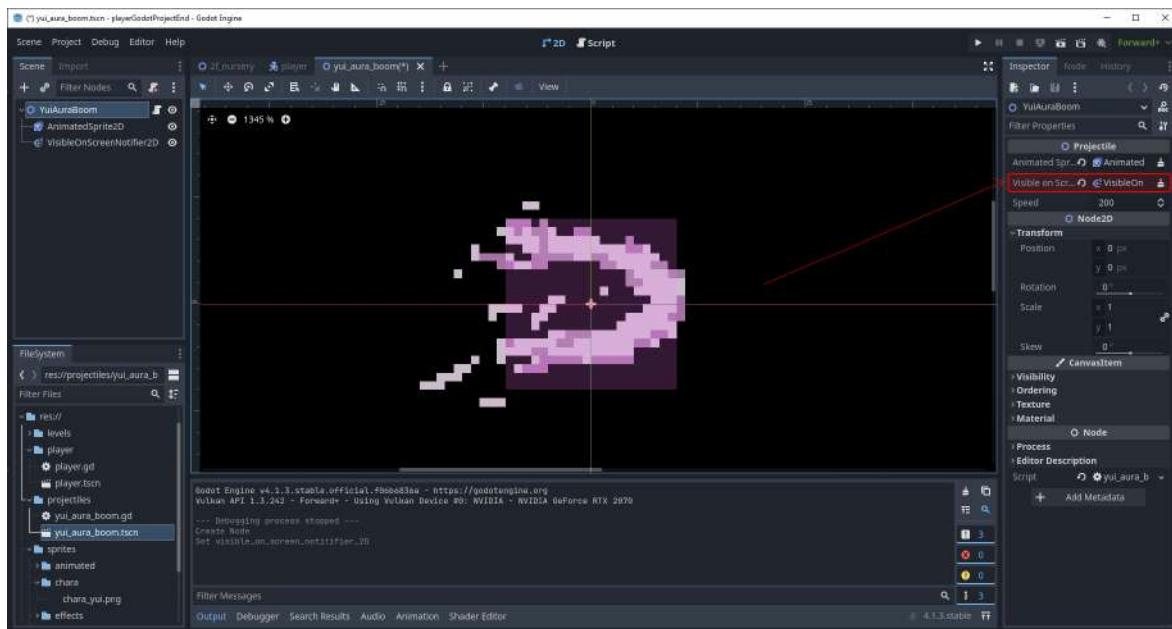
1  |class_name Projectile
2  |extends Node2D
3
4  |@export var animated_sprite_2D: AnimatedSprite2D # definiamo una variabile di export
5  |@export var visible_on_screen_notifier_2D: VisibleOnScreenNotifier2D # definiamo una variabile di export
6
7  |@export var SPEED: int = 200 # definiamo una variabile di export per la velocità del proiettile
8  |const START_ANIMATION_KEY = "start" # definiamo una costante per la chiave dell'animazione
9
10 |func _ready():
11    |>    animated_sprite_2D.play(START_ANIMATION_KEY) # facciamo partire l'animazione
12    |
13    |>    visible_on_screen_notifier_2D.screen_exited.connect(queue_free) # quando il proiettile esce dallo schermo lo rimuoviamo
14
15 |func _physics_process(delta):
16    |>    var direction = Vector2.RIGHT.rotated(rotation) # calcoliamo la direzione che muove il proiettile
17    |>    global_position += SPEED * direction * delta # muoviamo il proiettile
18

```

lì dove il signal (segnale) `screen_exited` viene emesso quando il nodo `VisibleOnScreenNotifier2D` "esce" dallo schermo; mentre la funzione `queue_free()` mette l'oggetto da cui viene invocata in coda per essere cancellato.

Poiché collegare un segnale a una funzione significa semplicemente fare in modo che la funzione venga evocata quando quel particolare segnale viene emesso; con questa riga di codice stiamo dunque dicendo che il proiettile deve essere rimosso dalla scena una volta uscito dallo schermo.

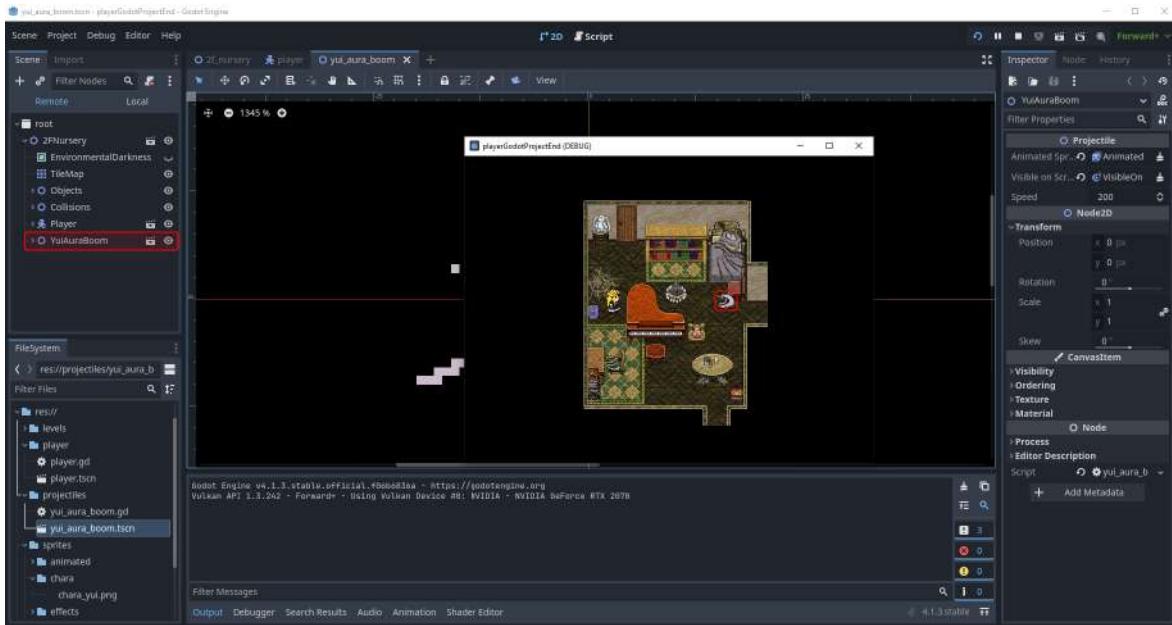
Ricordiamoci inoltre di assegnare il nodo `VisibleOnScreenNotifier2D` della scena `YuiAuraBoom` alla variabile di export `VisibleOnScreenNotifier2D`



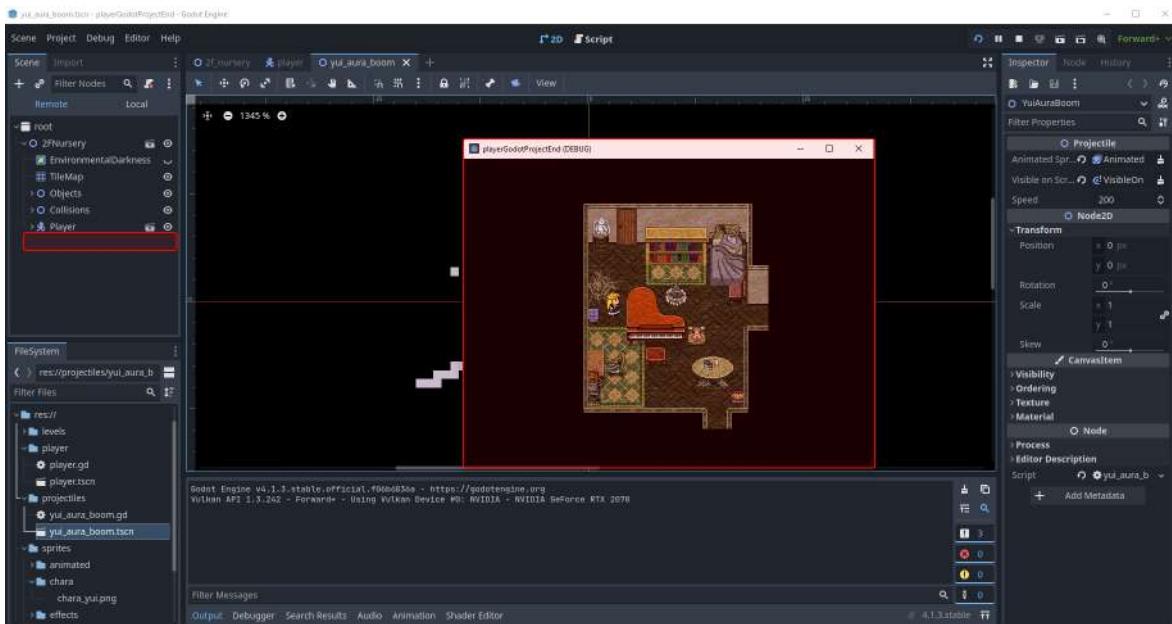
### ! Consiglio

Se una nuova variabile di export di una classe non viene visualizzata nell'editor di una sua classe derivata. Per forzare l'aggiornamento si può semplicemente chiudere e riaprire la scena.

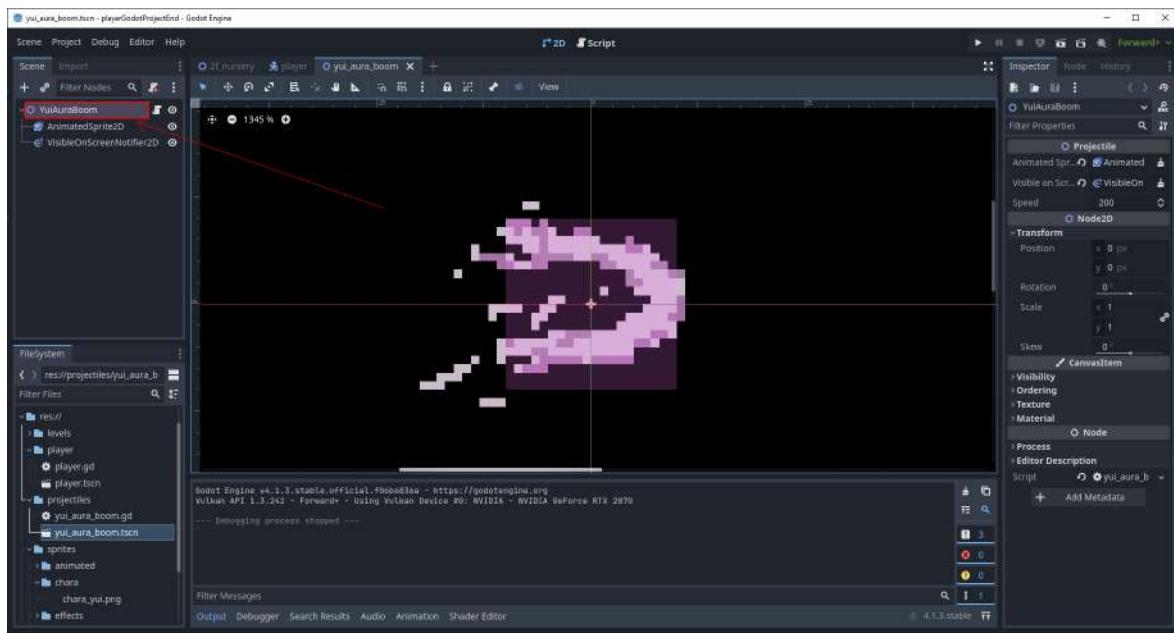
Così facendo, ogni nuovo proiettile



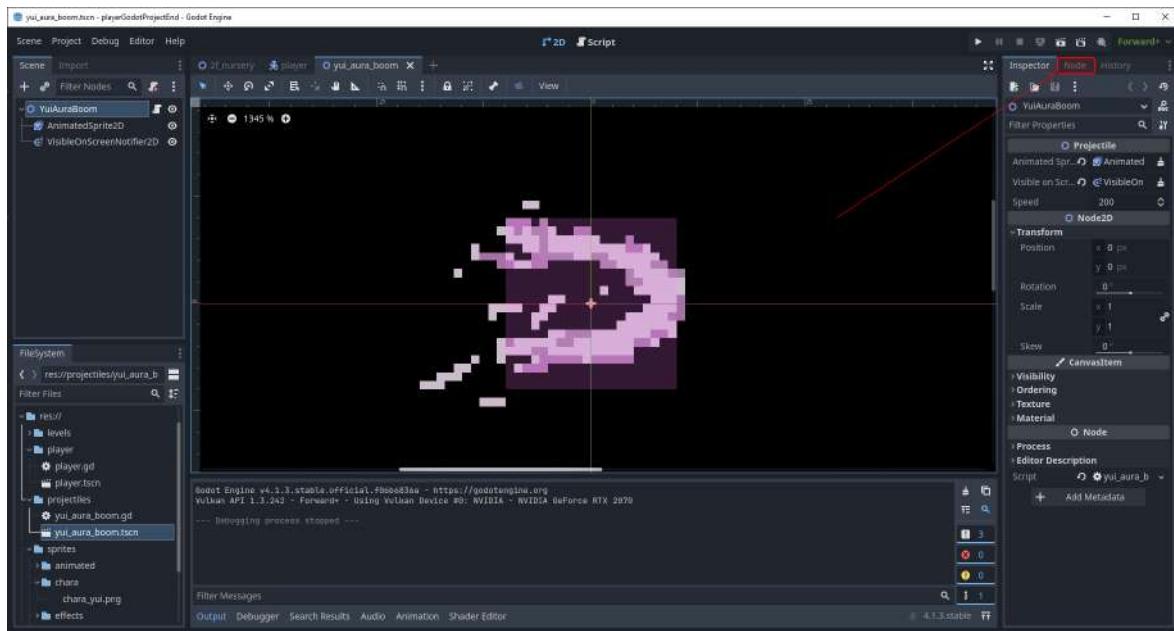
viene rimosso dalla scena una volta essere uscito dallo schermo



Come ciliegina sulla torta, assegnamo al nodo **YuiAuraBoom** il gruppo **Projectile**. Selezioniamo il nodo **YuiAuraBoom**

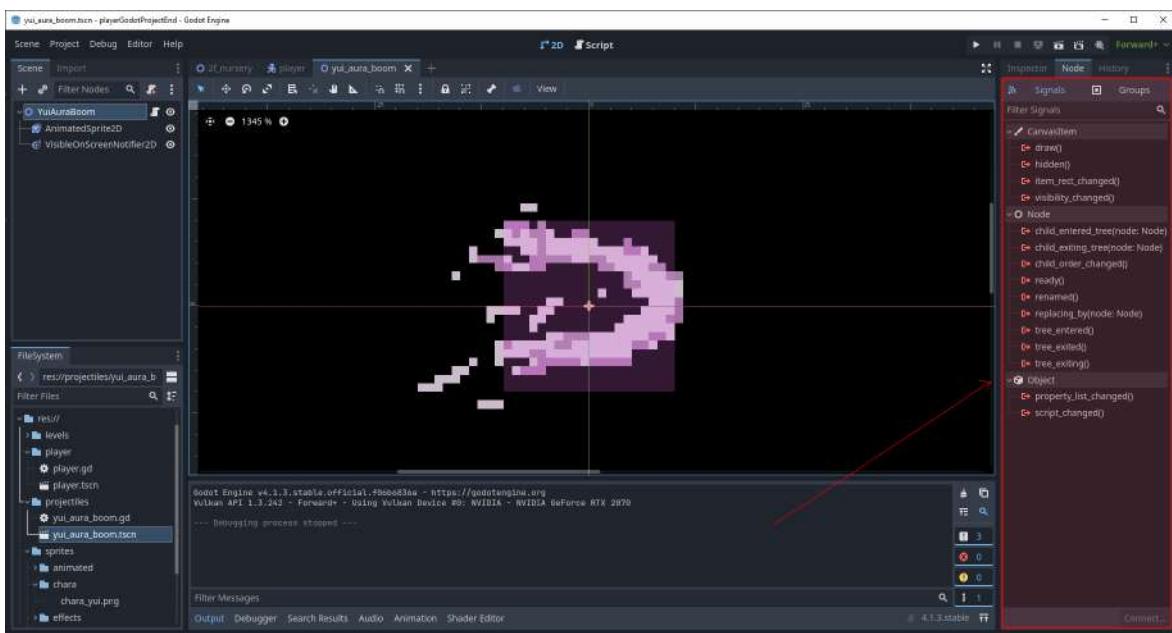


clicchiamo su **Node**

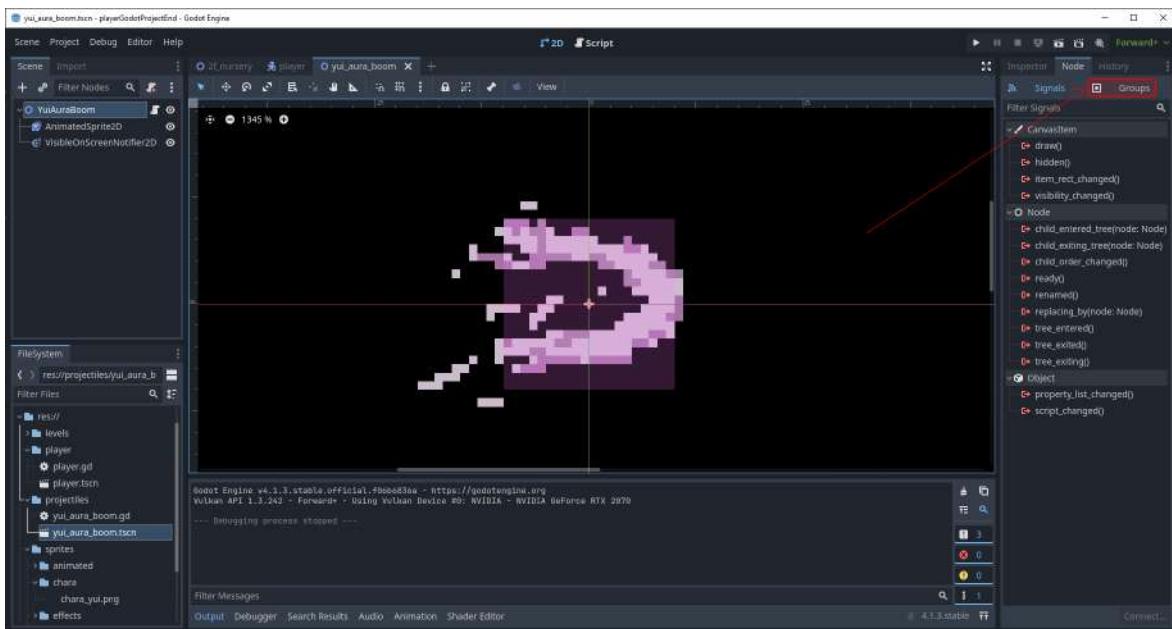


si aprirà un nuovo dock

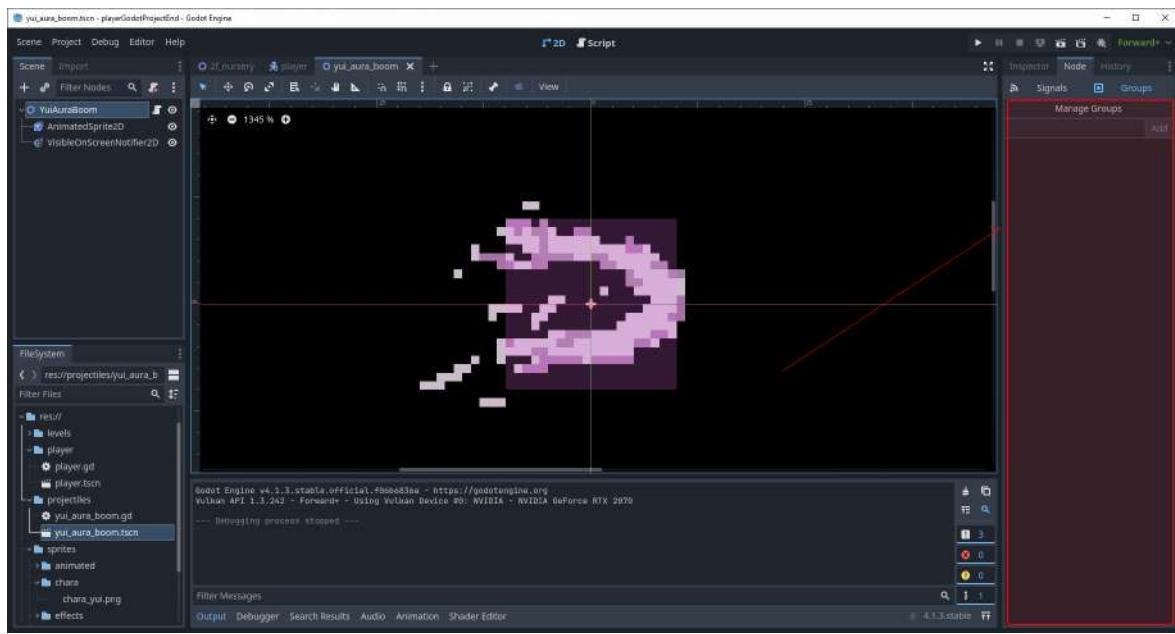
## CHAPTER 2. GODOT



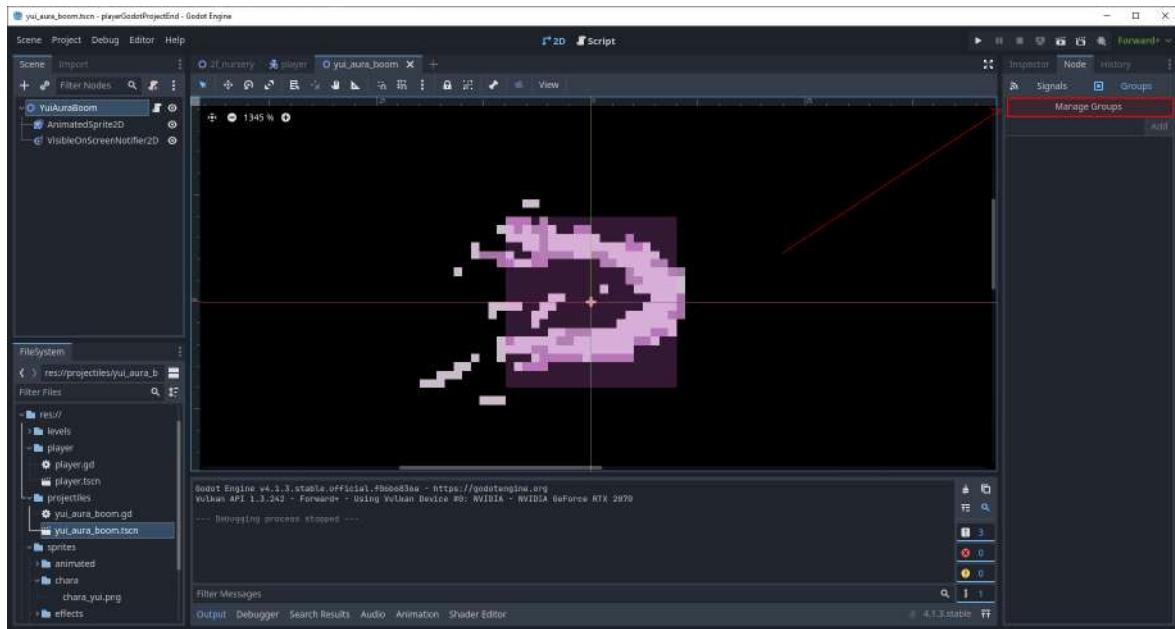
clicchiamo su **Groups**



si aprirà la sezione dedicata ai gruppi

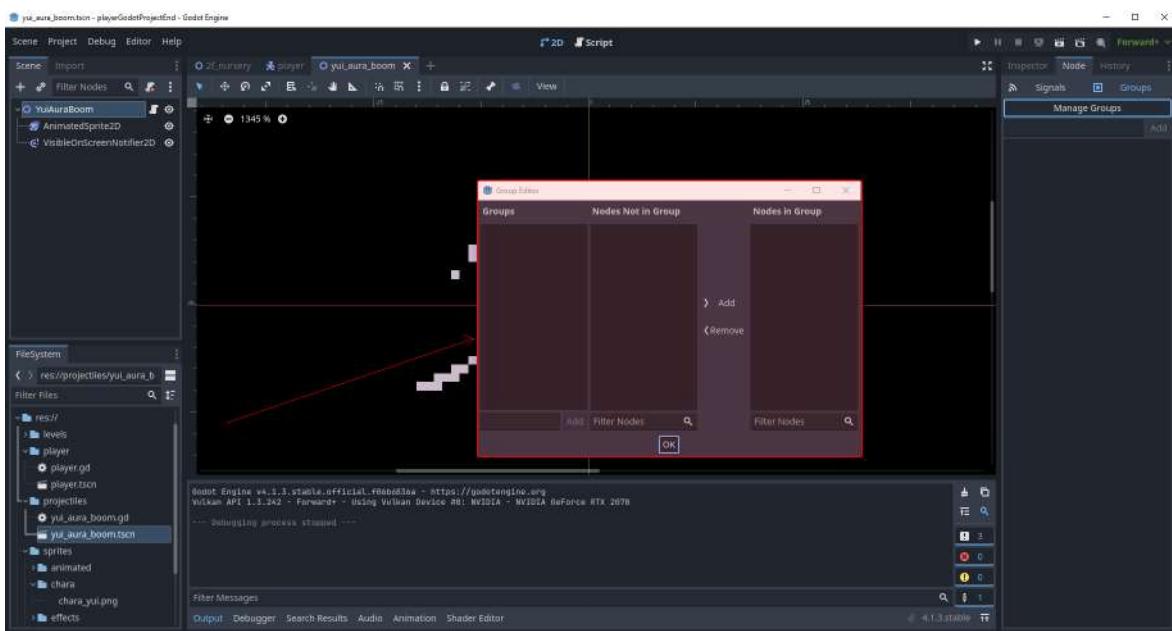


clicchiamo su **Manage Groups**

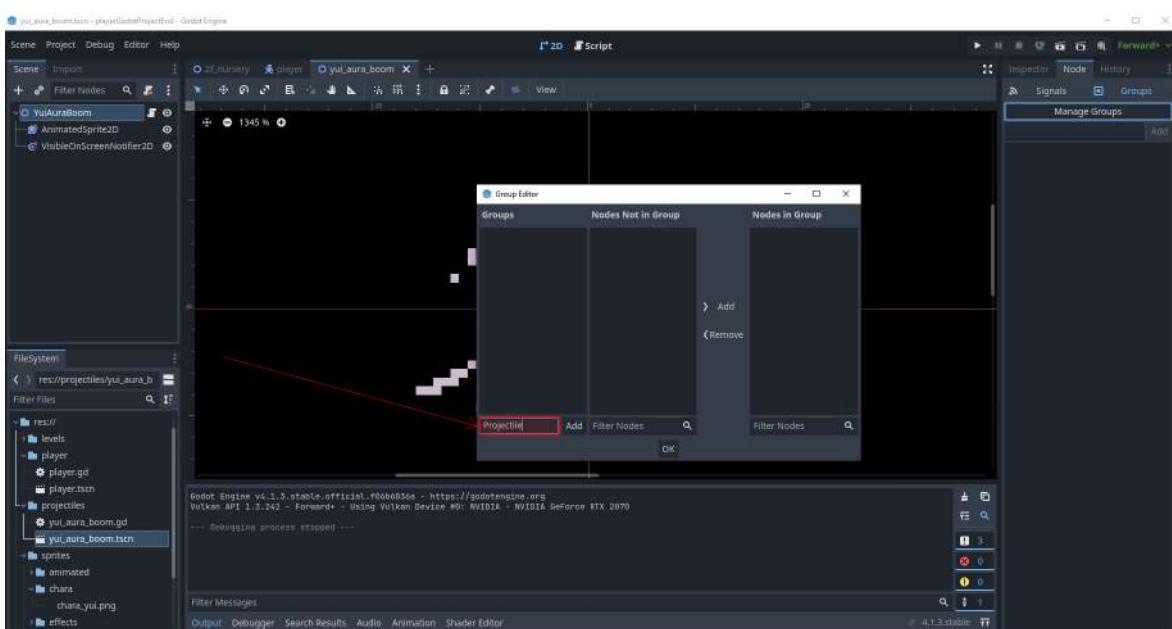


si aprirà una nuova finestra

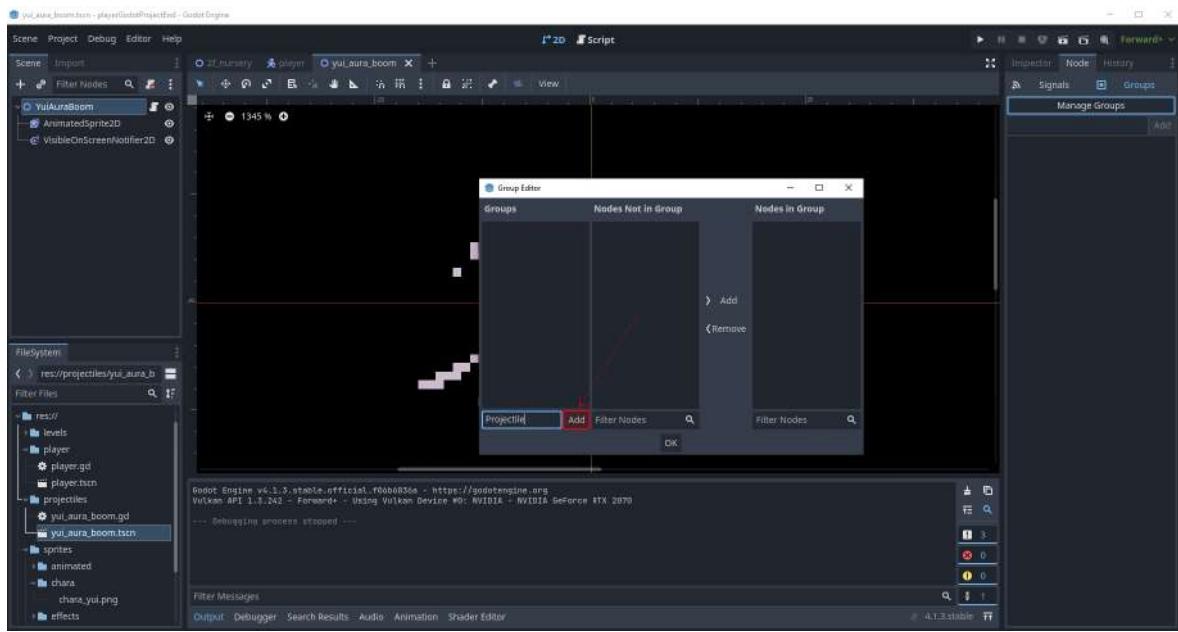
## CHAPTER 2. GODOT



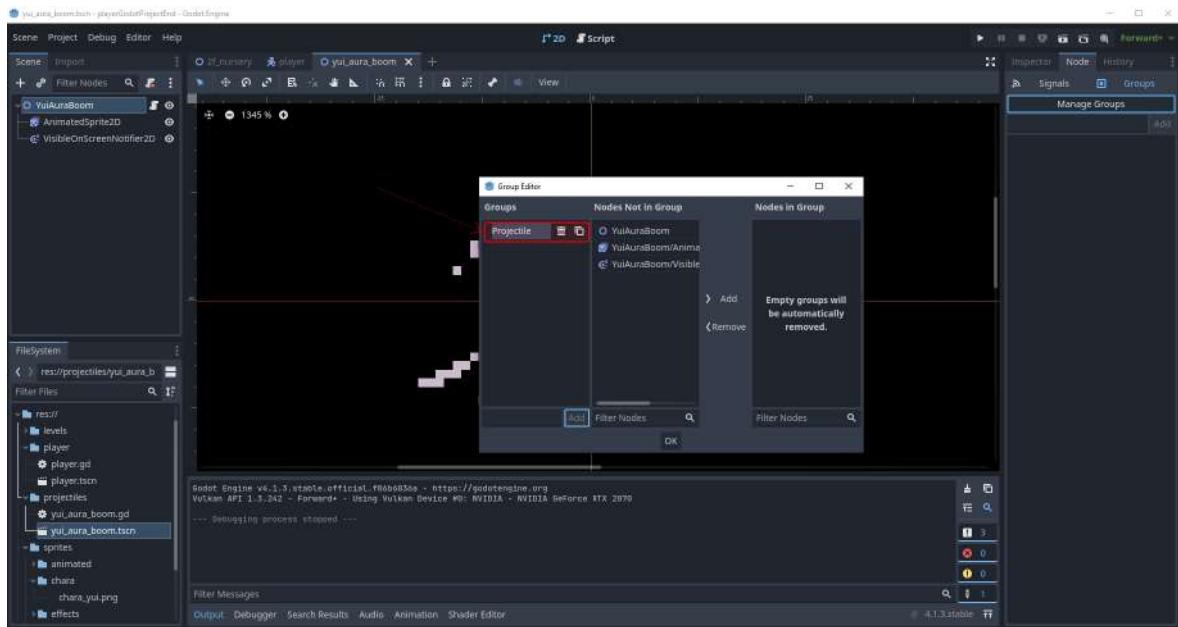
scriviamo **Projectile**



clicchiamo su **Add**

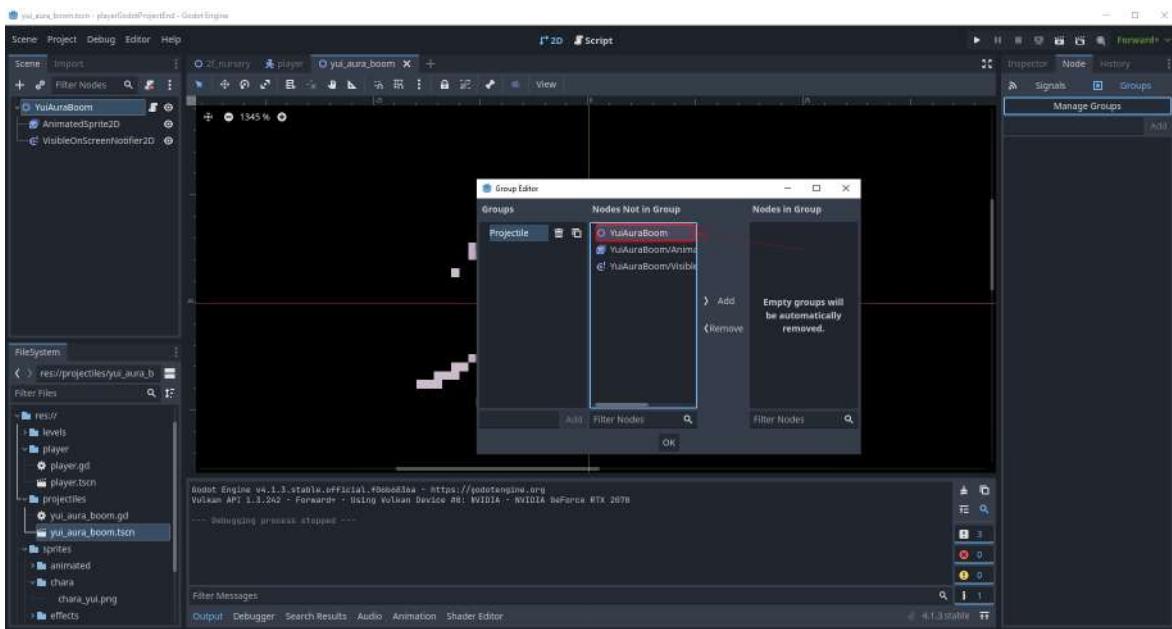


verrà creato un nuovo gruppo

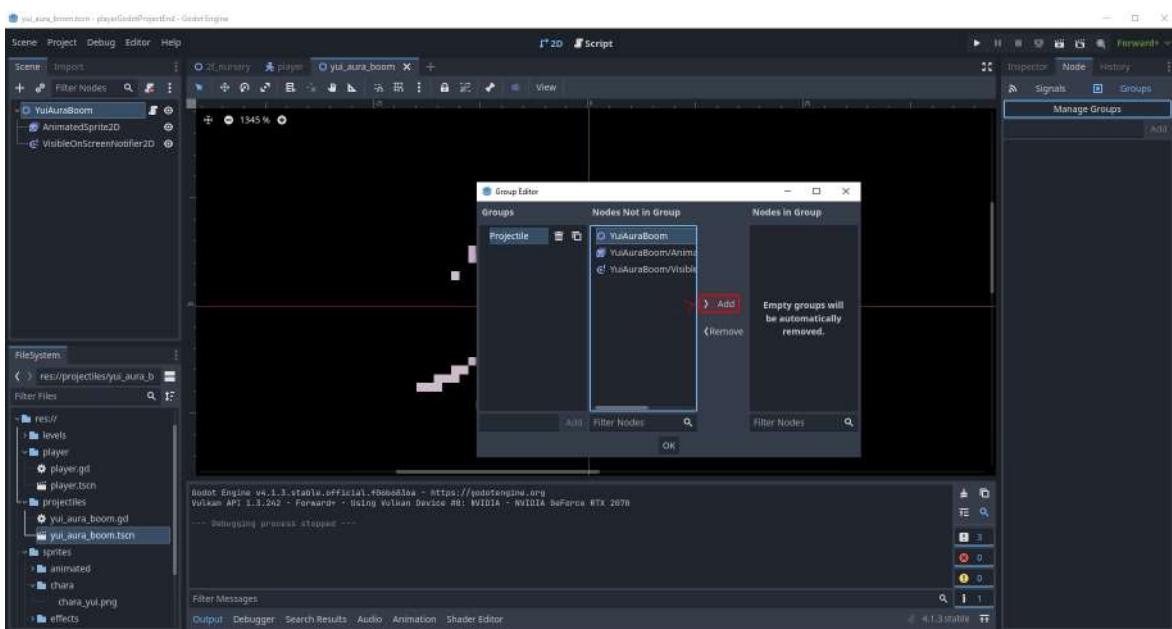


clicchiamo su **YuiAuraBoom**

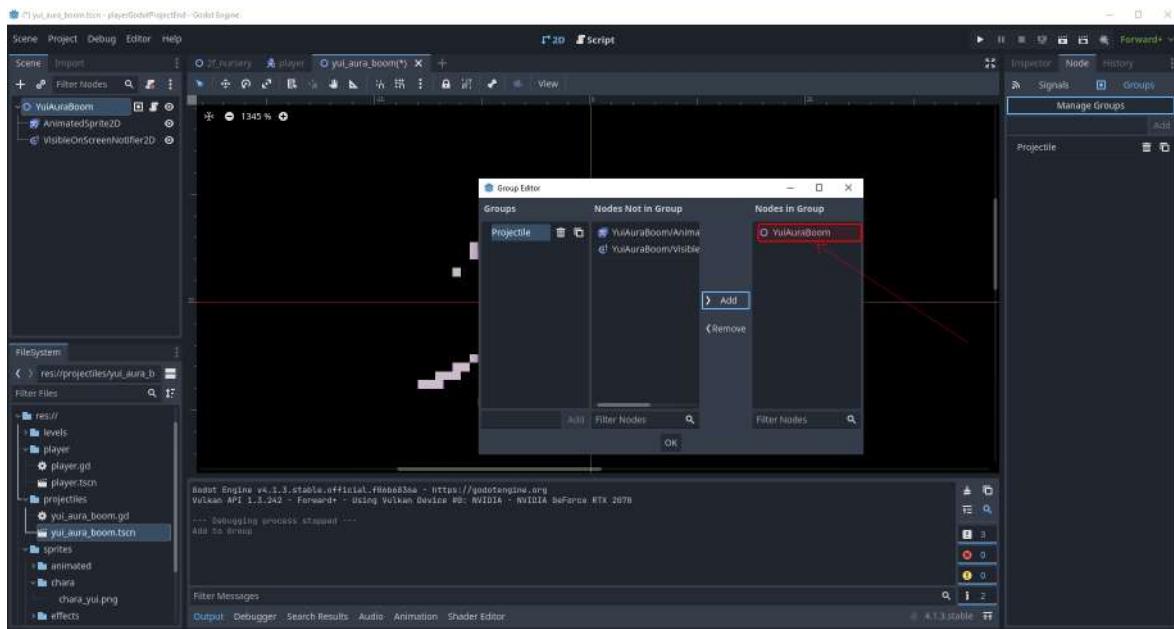
## CHAPTER 2. GODOT



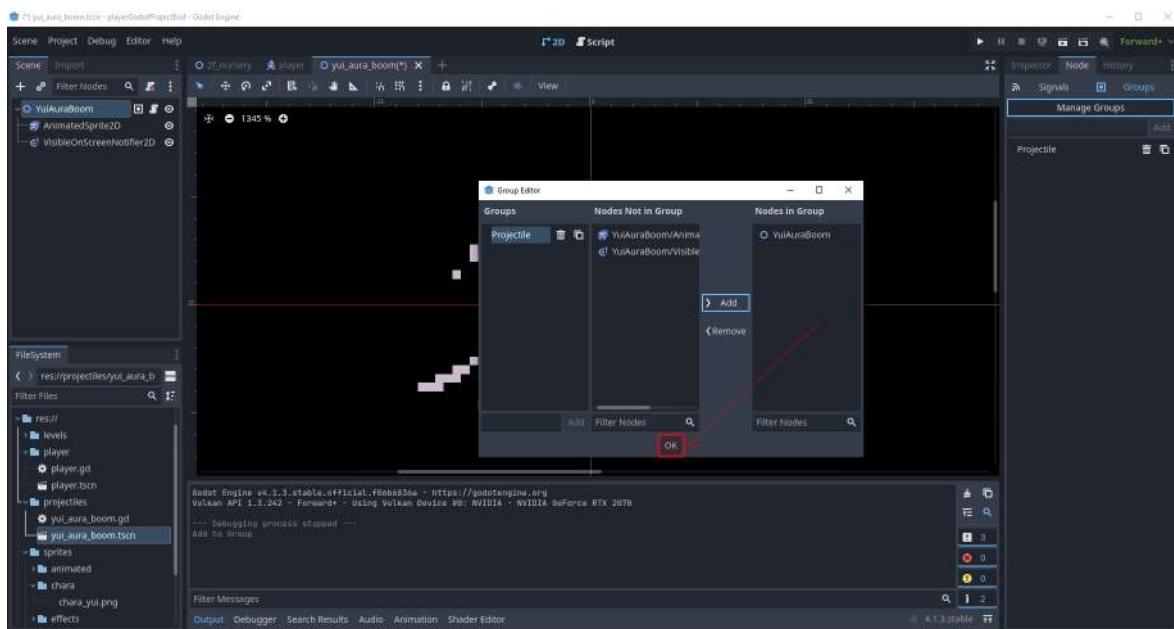
clicchiamo su **Add**



il nodo **YuiAuraBoom** verrà così inserito nel gruppo **Projectile**

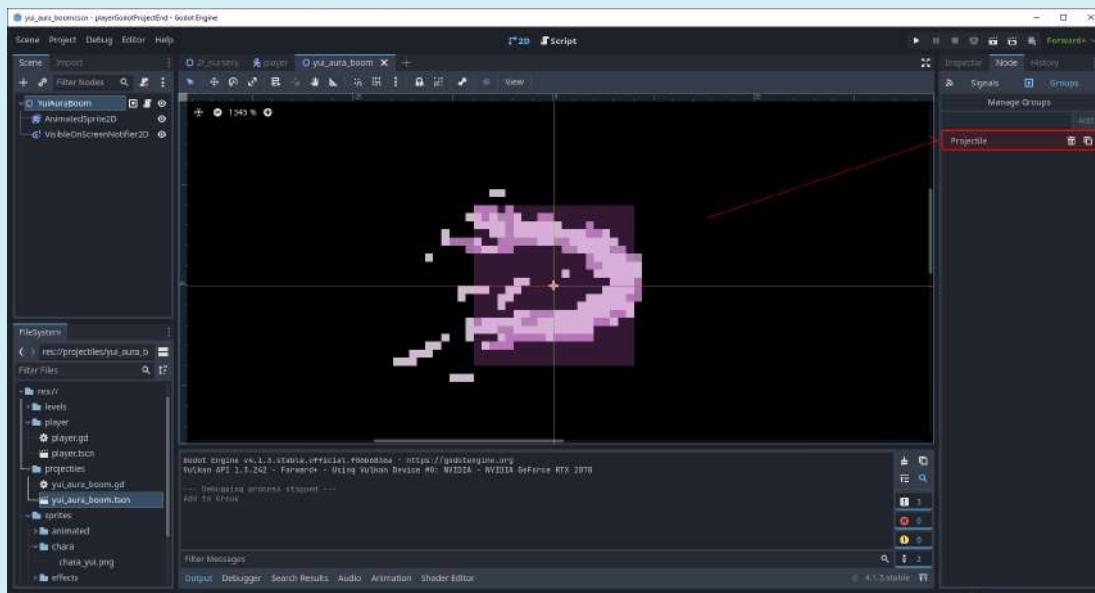


clicchiamo su **Ok** per chiudere la finestra

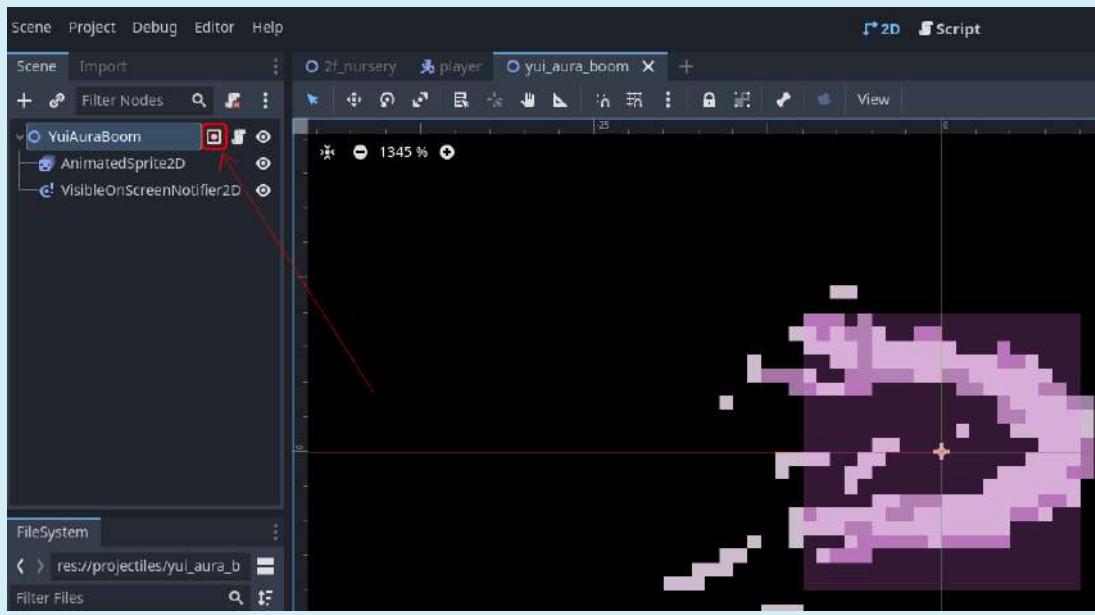


### ! Nota

Quando un nodo appartiene a un gruppo, il nome del gruppo appare nella lista del dock dei gruppi



Un ulteriore segno che il nodo appartiene a un gruppo è, inoltre, la seguente icona

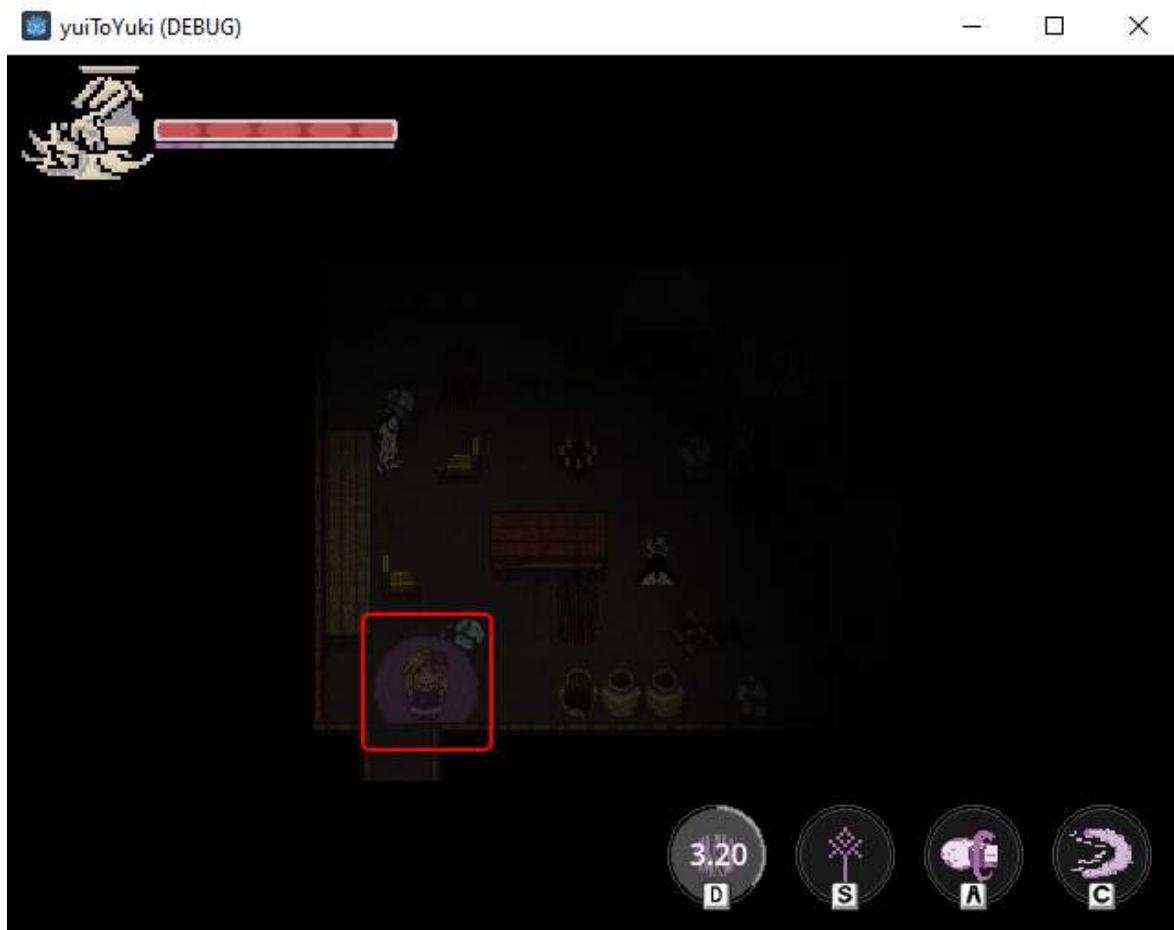


I **gruppi** in Godot funzionano come le tag negli altri software. Un nodo può essere aggiunto a tutti i gruppi che si vuole. Se un nodo appartiene a un gruppo, si può utilizzare lo SceneTree per:

- avere una lista di tutti i nodi di un gruppo
- invocare un metodo in tutti i nodi di un gruppo
- mandare una notifica a tutti i nodi di un gruppo

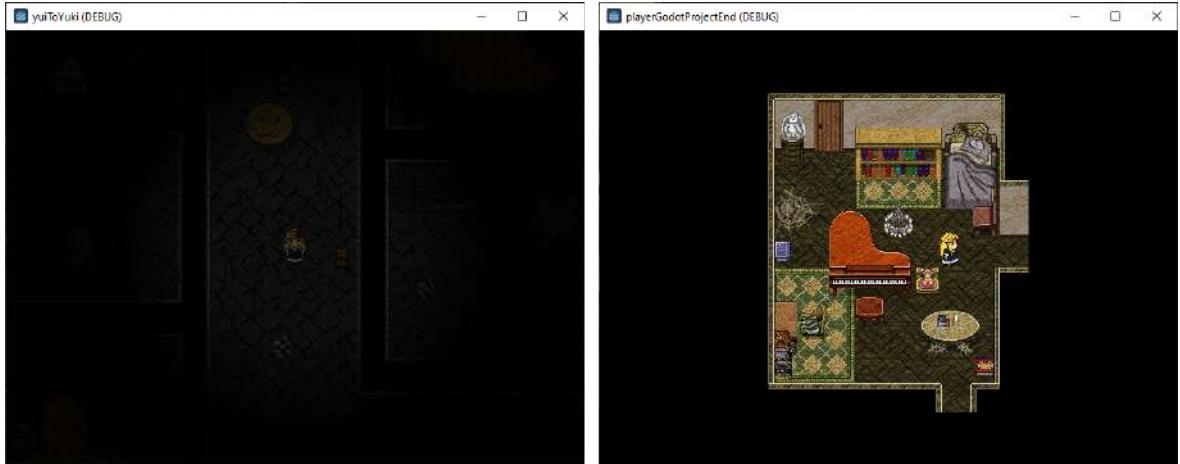
e molto altro ancora.

All'interno del nostro videogioco, i gruppi, più precisamente quello **Projectile**, vengono utilizzati per capire quali oggetti devono essere respinti dalla barriera del nostro player.



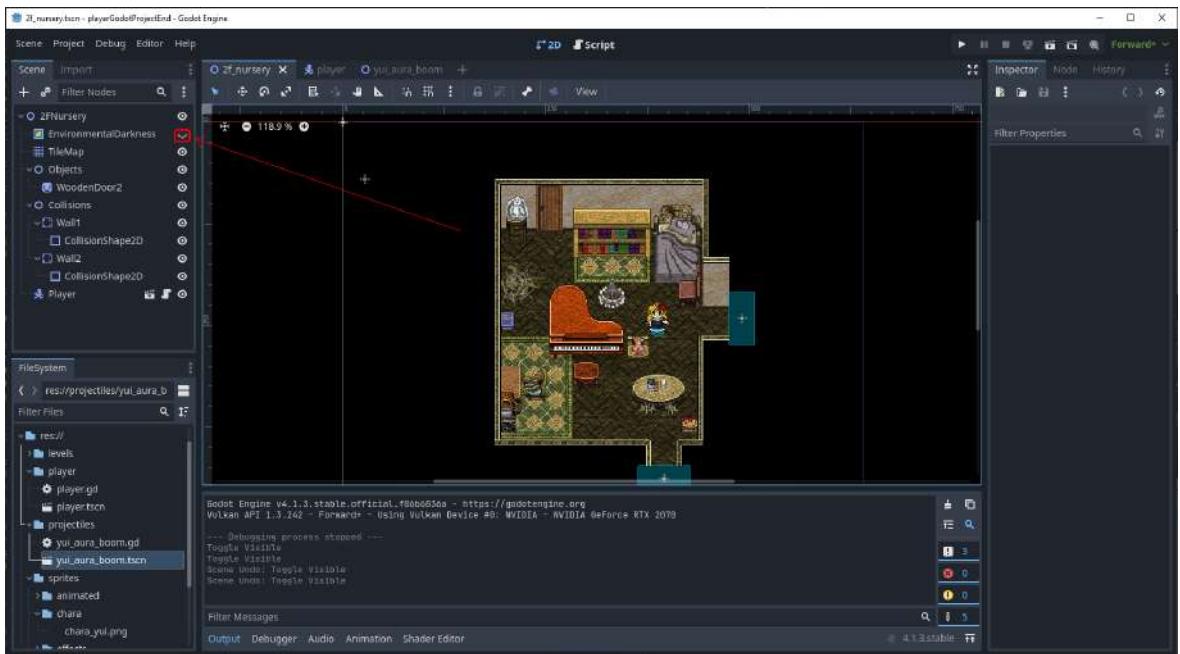
### 2.4.7 Player light

Nella versione finale del nostro gioco, notiamo però qualcosa di diverso rispetto a quella che stiamo progettando passo passo insieme

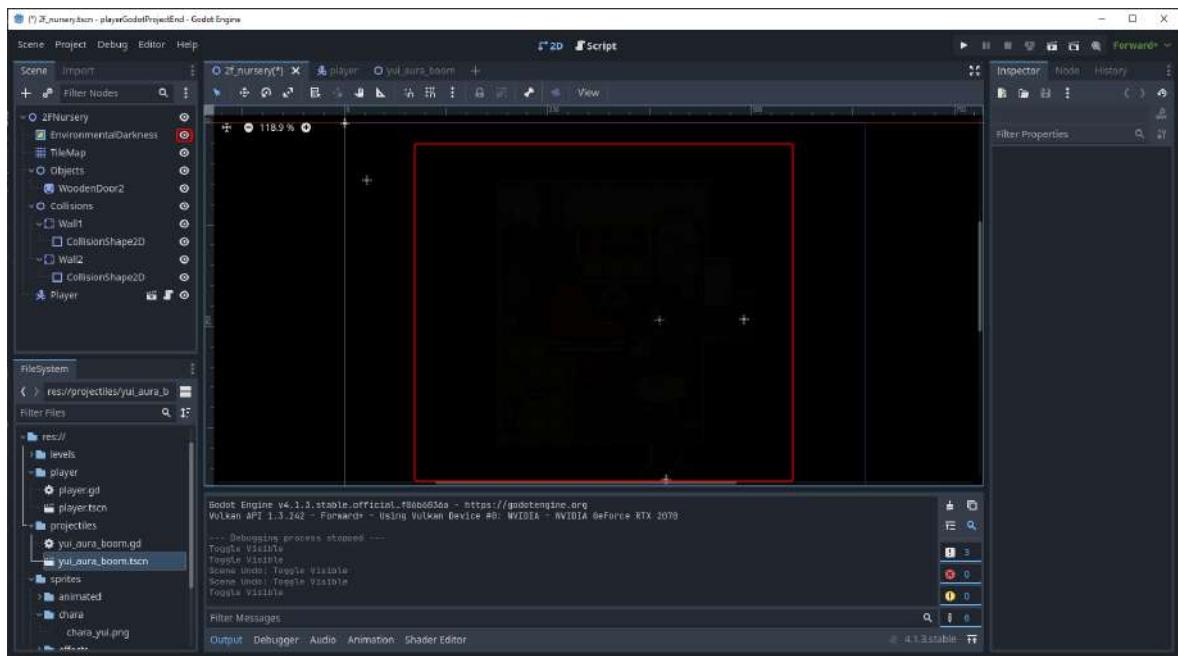


Messi l'uno di fianco all'altro è evidente: l'oscurità.

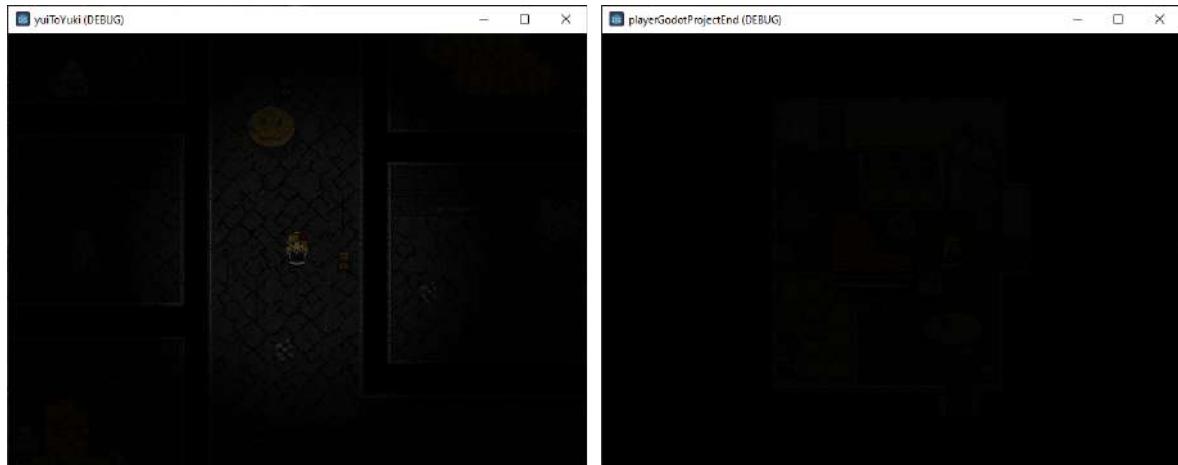
Per attivarla, basta semplicemente cliccare sull'icona dell'occhio chiuso del nodo **EnvironmentalDarkness** nella scena **Player**. Tutto quello che fa l'icona forma di occhio è rendere visibili o invisibile il rispettivo nodo.



Così facendo renderemo infatti visibile il nodo **EnvironmentalDarkness** all'interno della scena, il quale è responsabile dell'oscurità del gioco, e la stanza verrà effettivamente avvolta dalle tenebre.

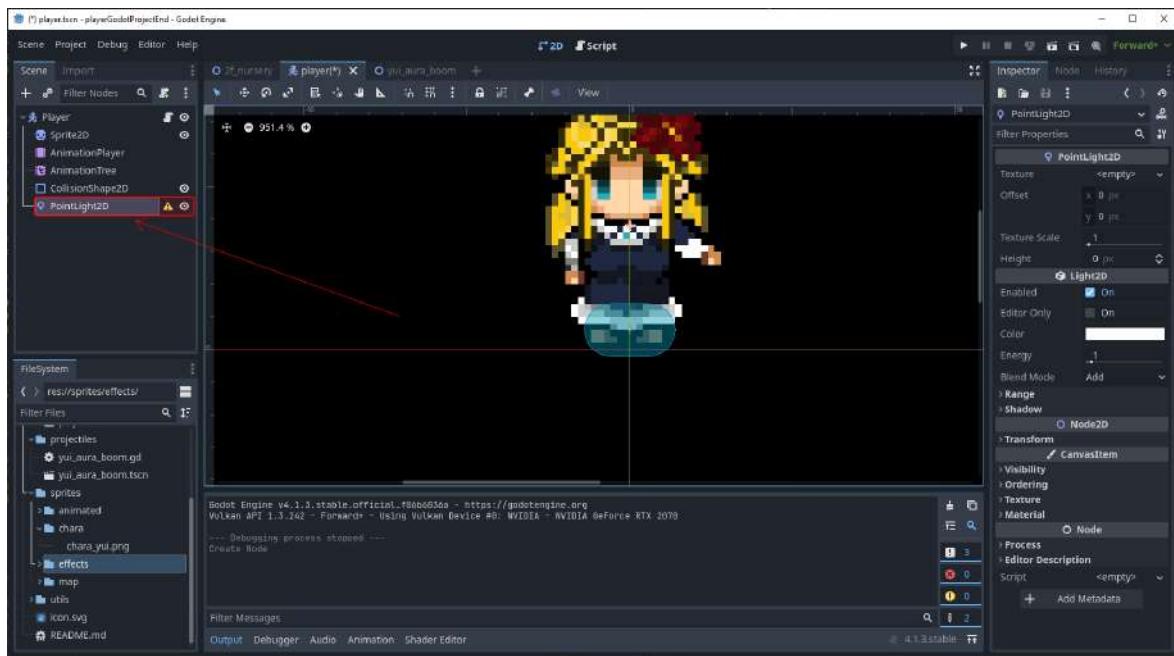


Avviando adesso il gioco ci rendiamo però conto che c'è ancora qualcosa che non va

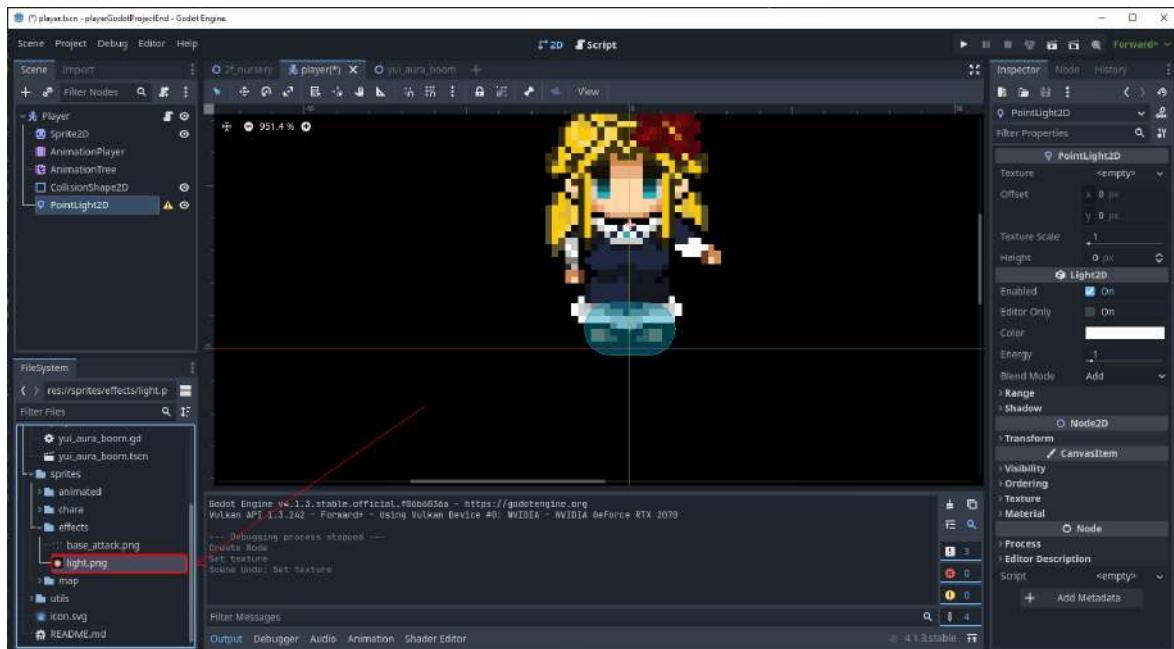


Nella versione finale è come se il giocatore emettesse una specie di luce che illumina il troppo buio della stanza, la quale, altrimenti, non si riuscirebbe a vedere per niente, come succede infatti nella nostra versione attuale del videogioco.

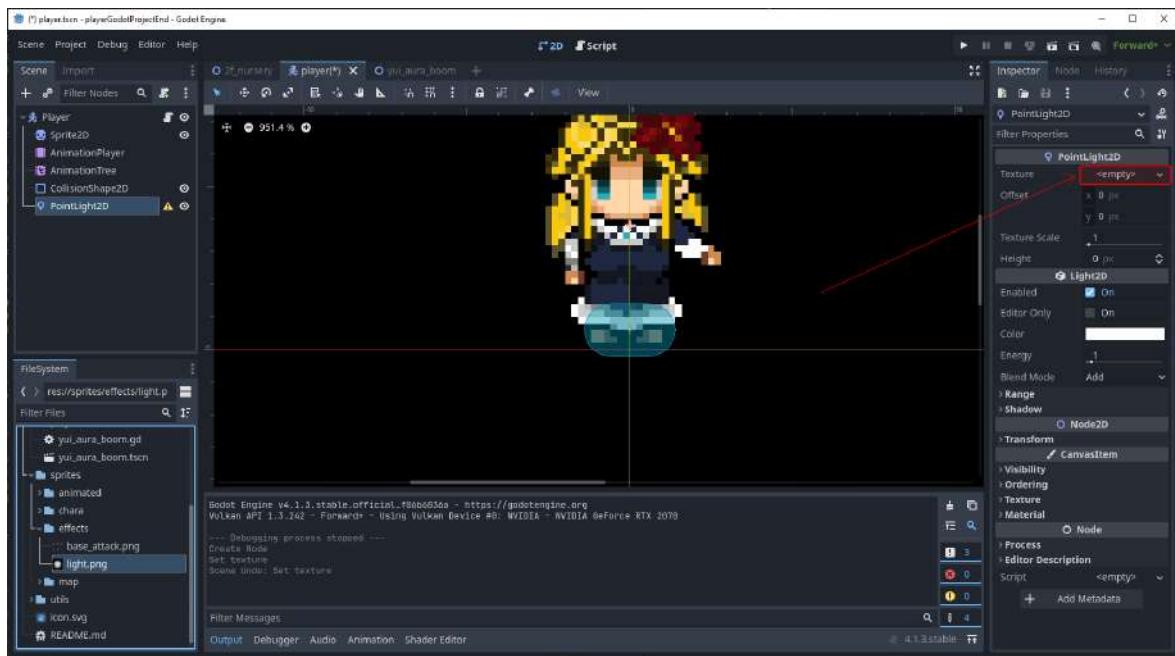
Per ottenere questo effetto, per prima cosa aggiungiamo un nodo **PointLight2D** al nostro nodo player.



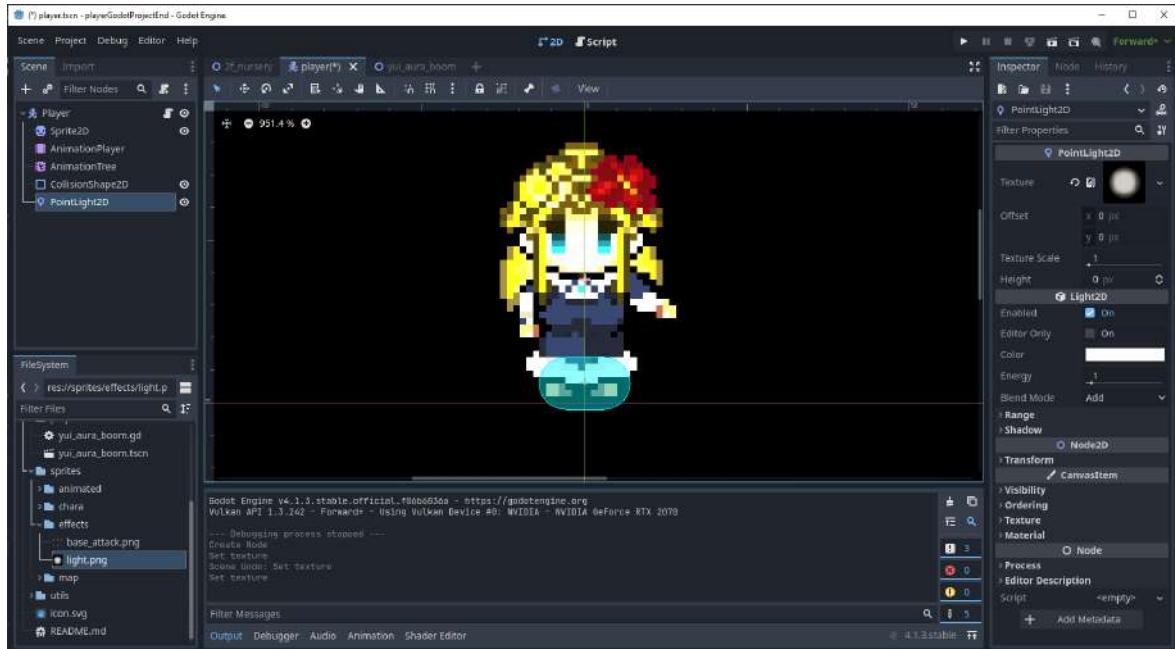
trasciniamo dunque la texture `light.png` della cartella `res://sprites/effects`.



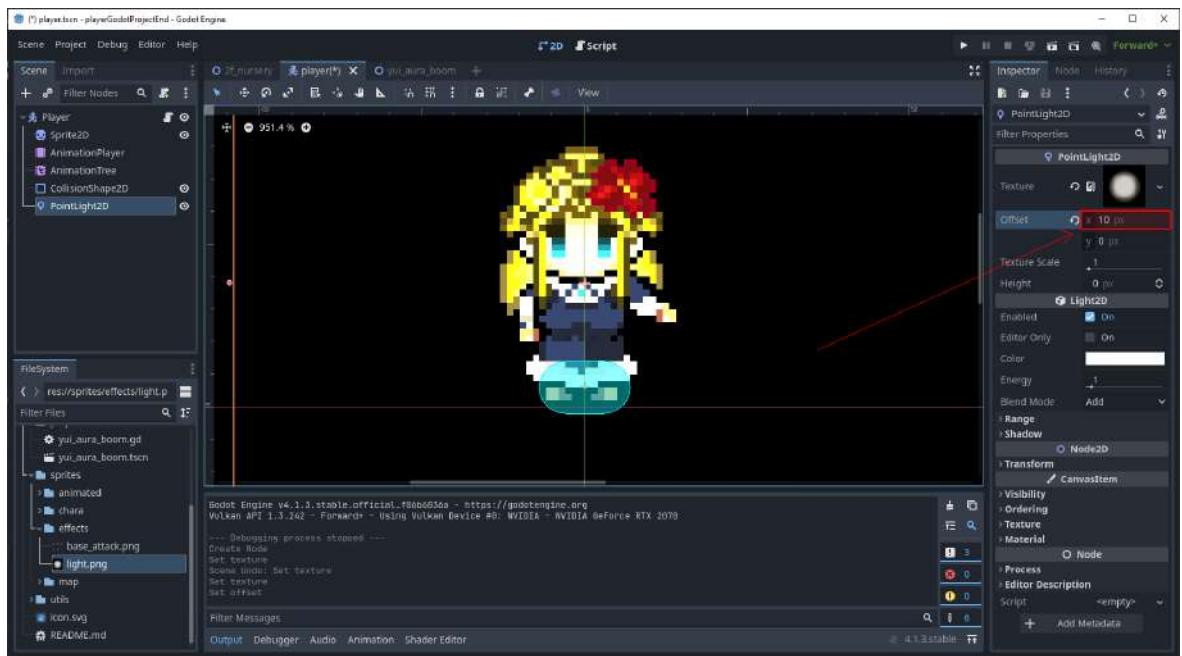
all'interno di `<empty>` della voce `Texture`.



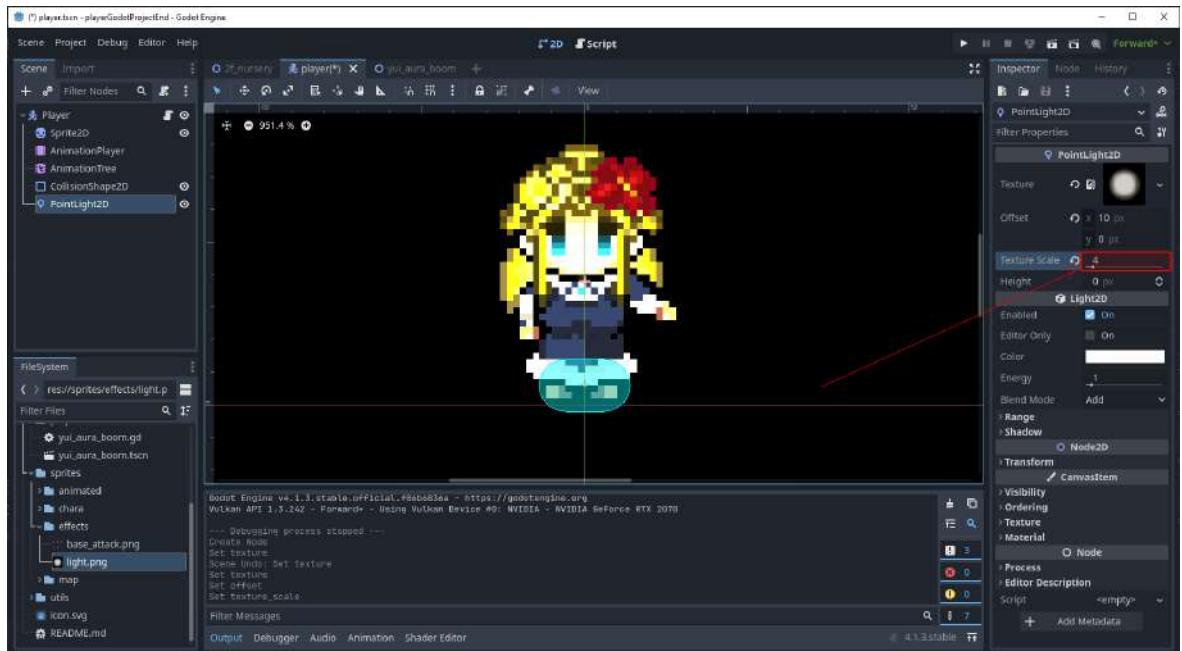
il player si illuminerà di bianco.



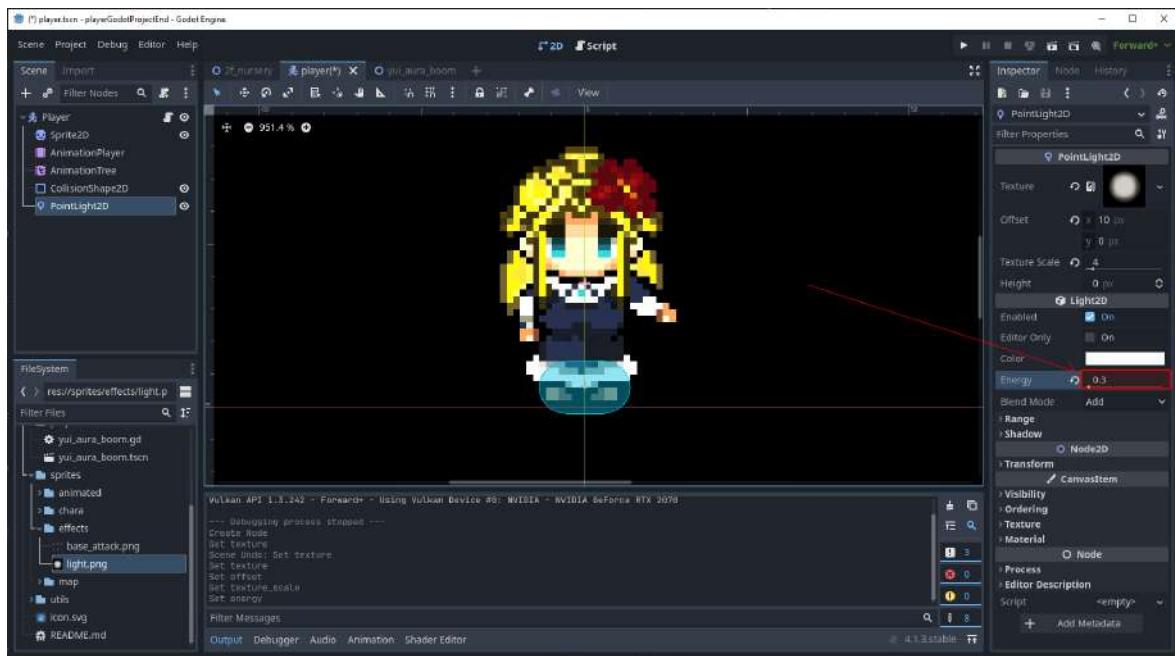
impostiamo ora a **10** il valore **x** della voce **Offset**.



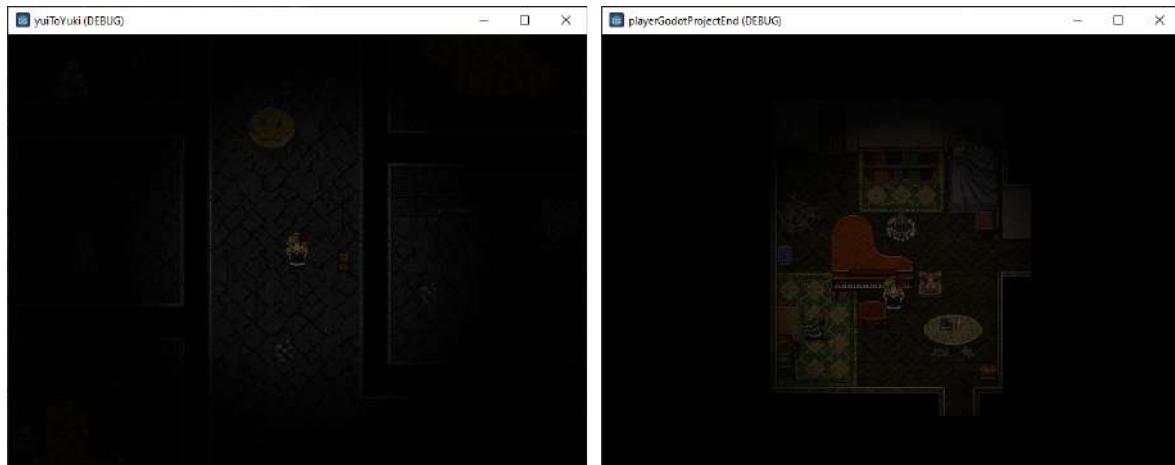
a 4 il valore di **TextureScale**.



e a 0.3 il valore di **Energy**, il quale renderà la luce meno accesa.



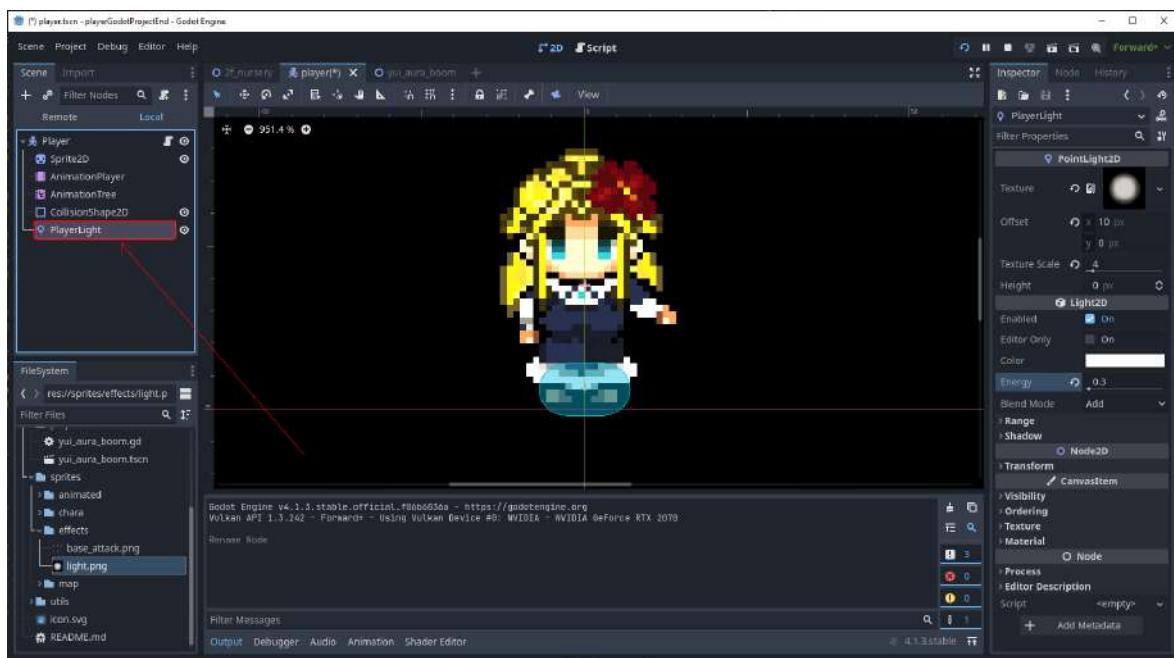
Avviando adesso il gioco, l'oscurità e la luce sono esattamente come nella versione finale



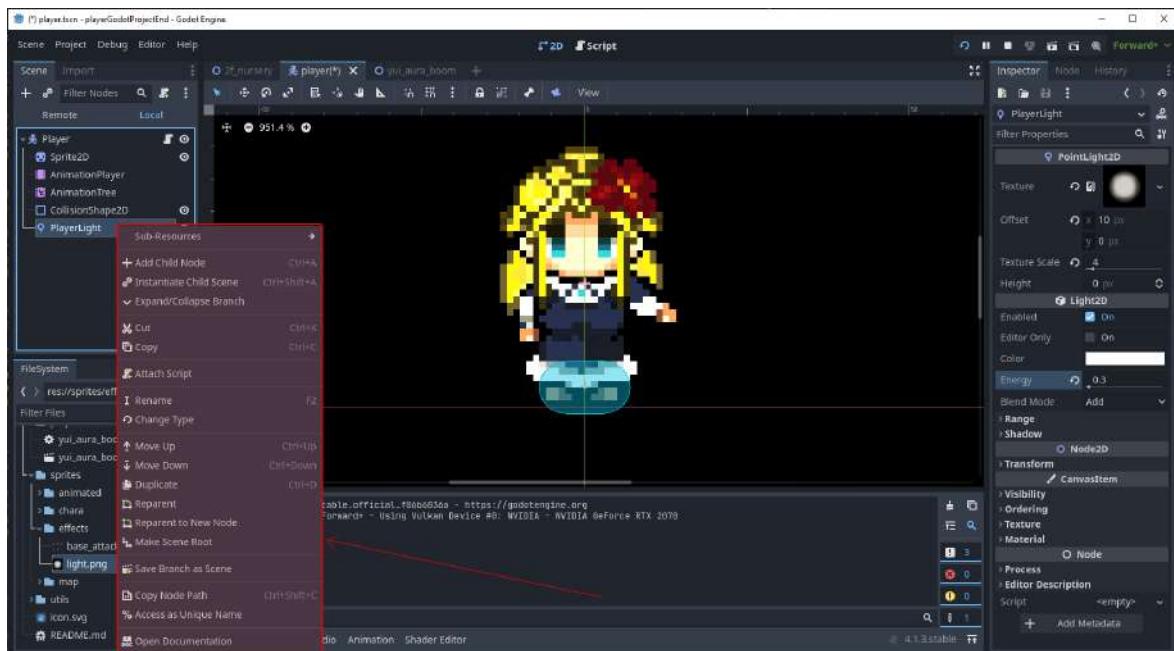
Salviamo ora il nodo **PointLight2D**, figlio del nodo **Player**, come una nuova scena indipendente, in modo da poterlo utilizzare anche in altre scene del nostro videogioco.

Rinominiamo il nodo **PointLight2D** in **PlayerLight**.

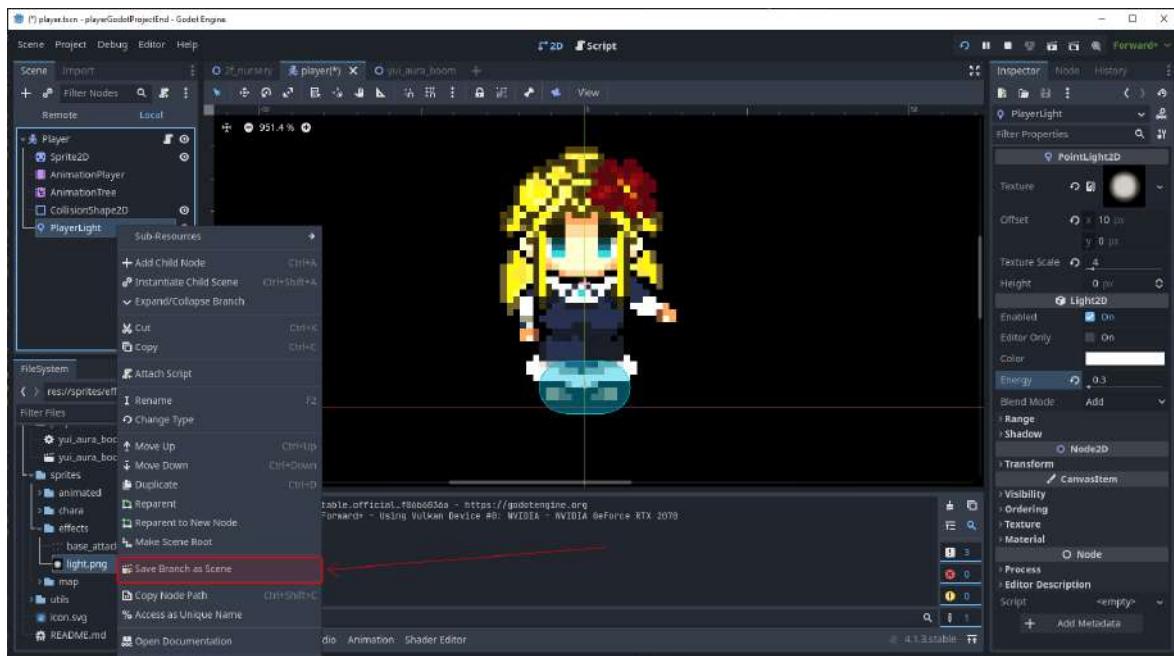
## CHAPTER 2. GODOT



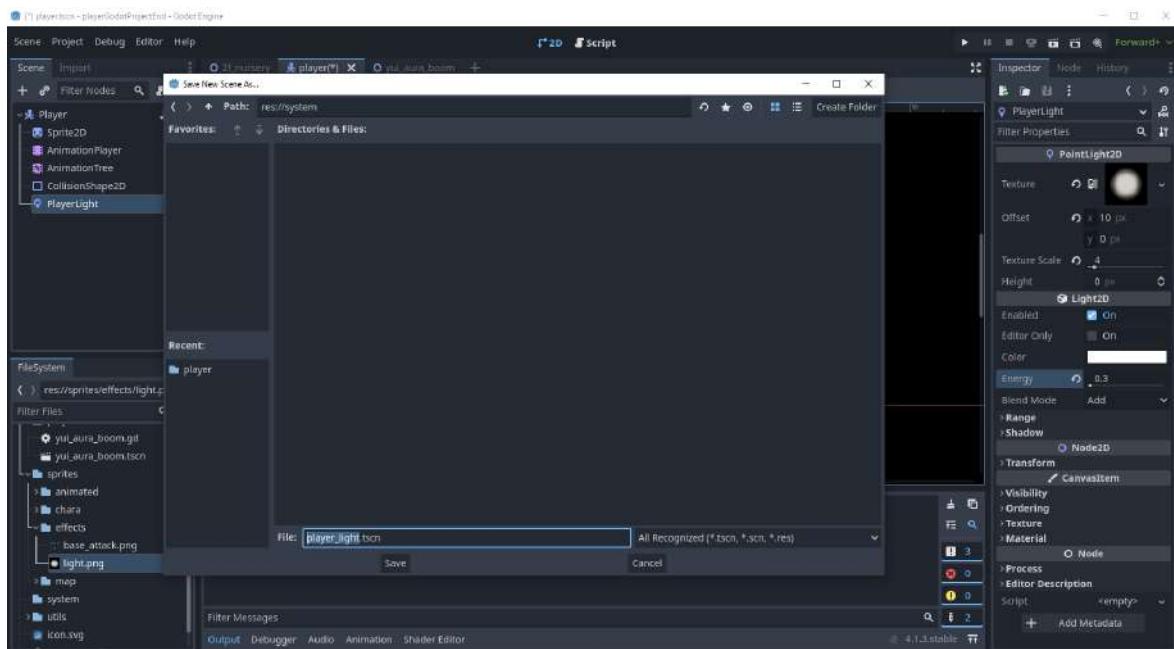
clicchiamo con il tasto destro sul nodo **PlayerLight** per aprire una nuova finestra di opzioni.



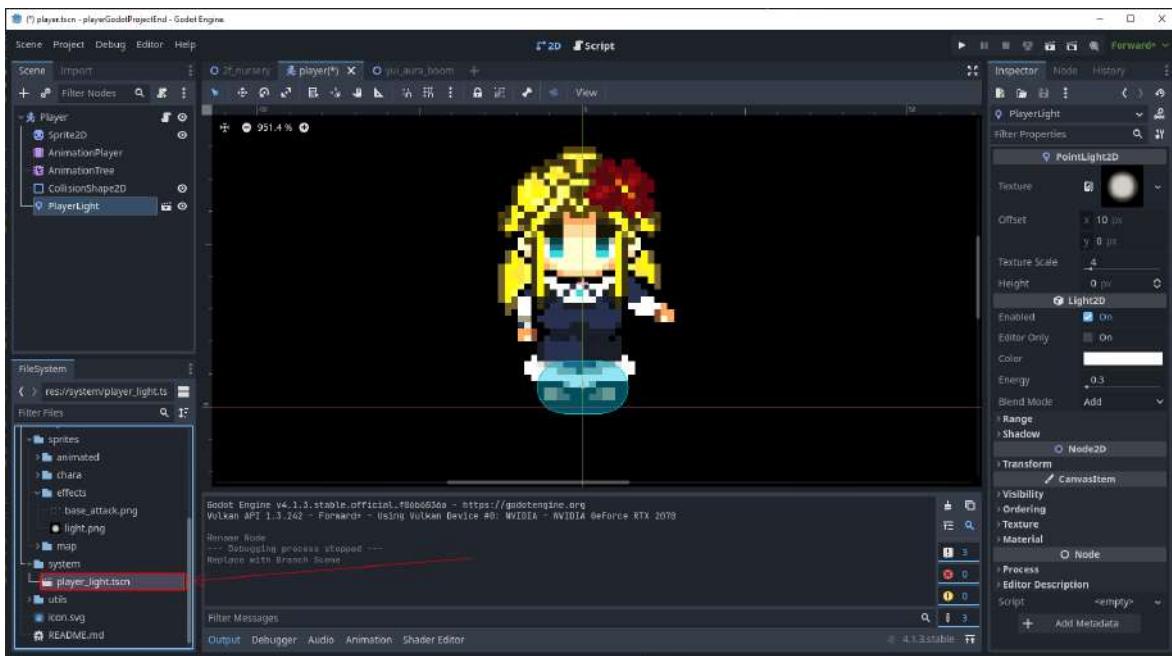
clicchiamo su **Save Branch as Scene**.



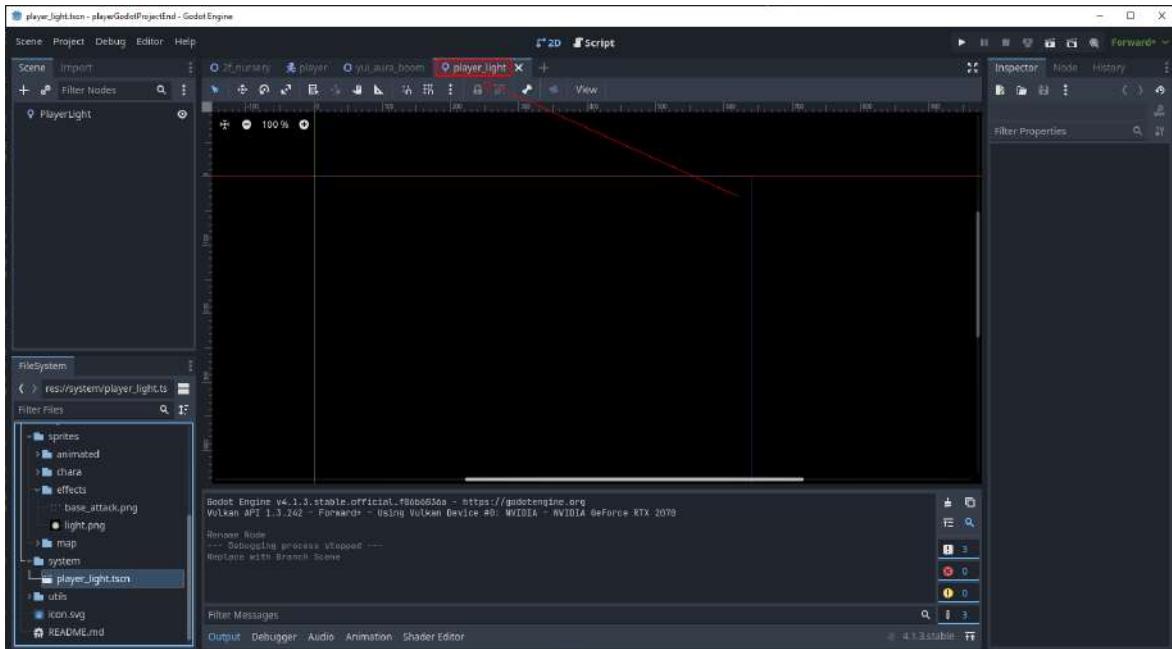
e salviamo la nuova scena.



in una nuova cartella `res://system`.

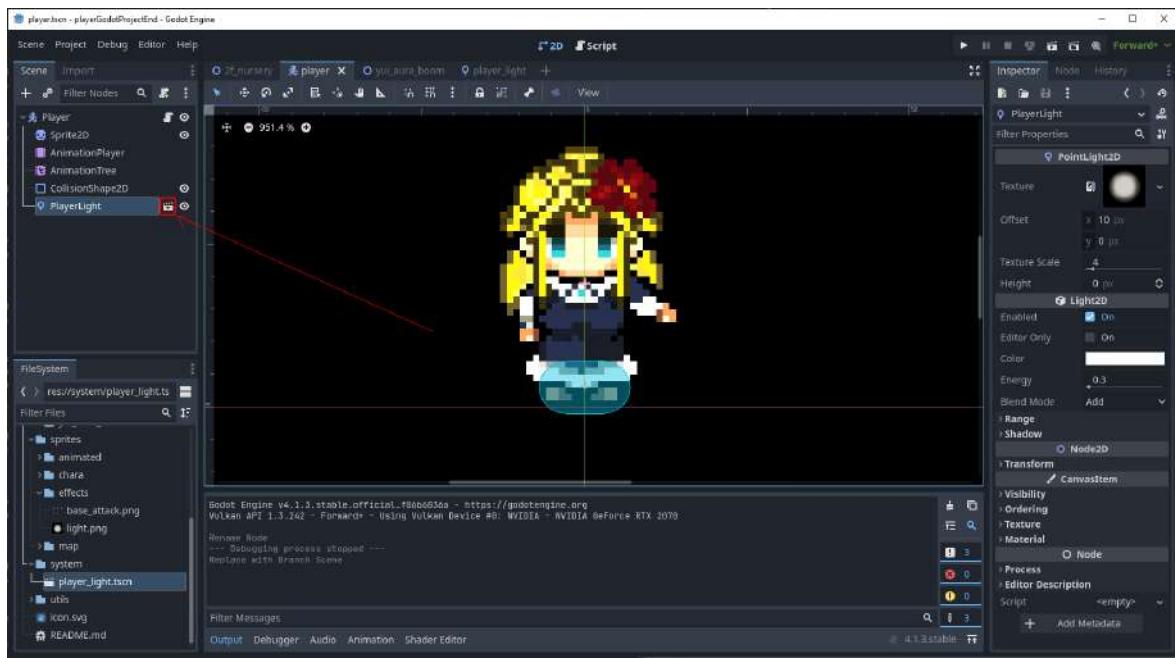


Facendo doppio click sul file `player_light.tscn`, si aprirà correttamente la finestra di modifica della scena `PlayerLight`.



Questa è la conferma che il nodo `PlayerLight` si è trasformato con successo nella scena `PlayerLight`.

Notiamo inoltre che, nella scena `Player`, il nodo `PlayerLight` presenta ora un'icona utilizzata da Godot per indicarci che il nodo considerato è una scena.



Un click su tale icona farà aprire la corrispondente scena nell'editor.