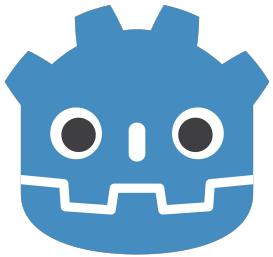
Introduzione a Godot



Presentazione - Elio Musacchio

UniBa

SWAP researchgroup

Cos'è Godot

Godot Engine is a feature-packed, cross-platform game engine to create 2D and 3D games from a unified interface. It provides a comprehensive set of common tools, so that users can focus on making games without having to reinvent the wheel. Games can be exported with one click to a number of platforms, including the major desktop platforms (Linux, macOS, Windows), mobile platforms (Android, iOS), as well as Web-based platforms and consoles.

Installare Godot

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

Oppure ...

https://editor.godotengine.org/releases/4.4.1.stable/godot.editor.html

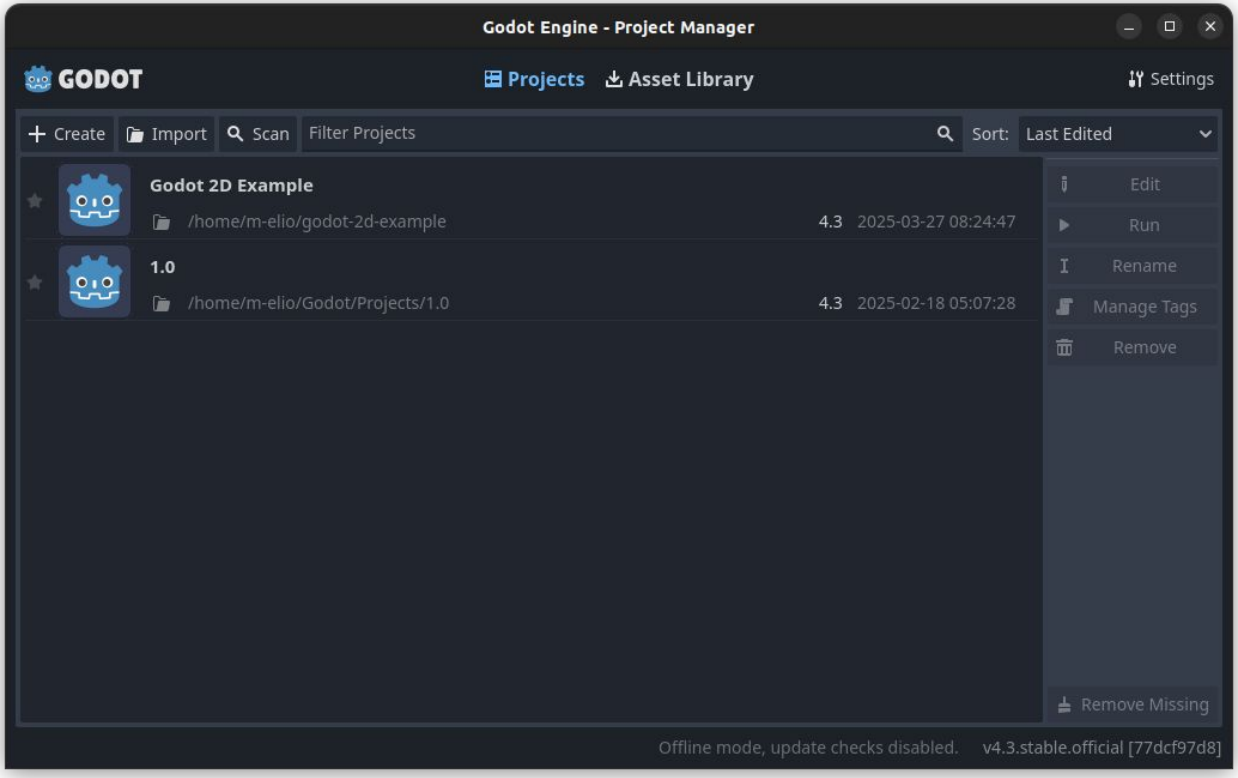
Altre risorse che useremo oggi

https://itch.io/game-assets

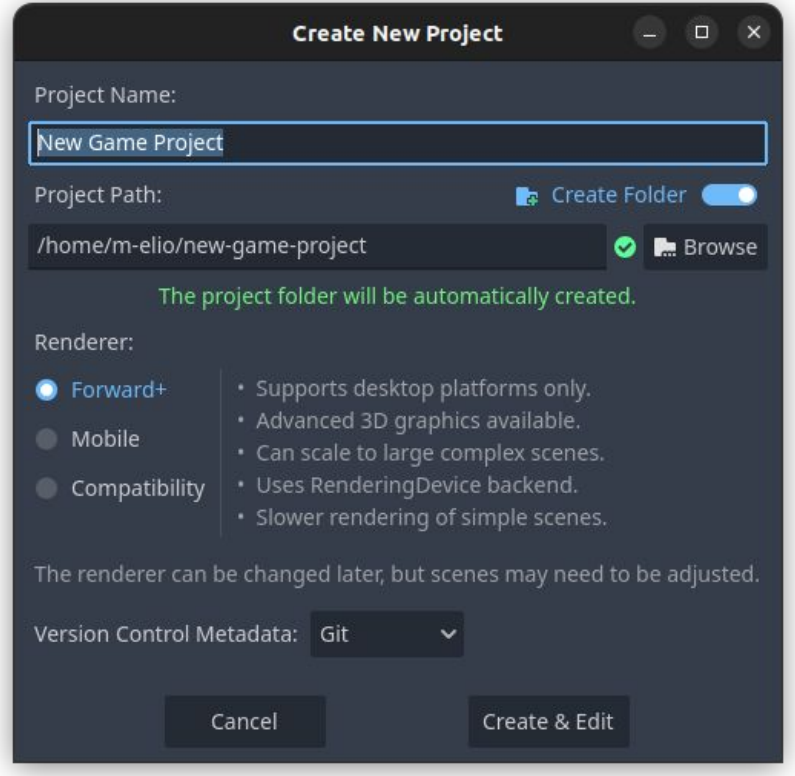
Oggi useremo nello specifico i seguenti asset:

- ➔ https://brackeysgames.itch.io/brackeys-platformer-bundle
- ➔ https://bdragon1727.itch.io/fire-pixel-bullet-16x16
- ➔ https://loudeyes.itch.io/paper-ui-pack-for-games

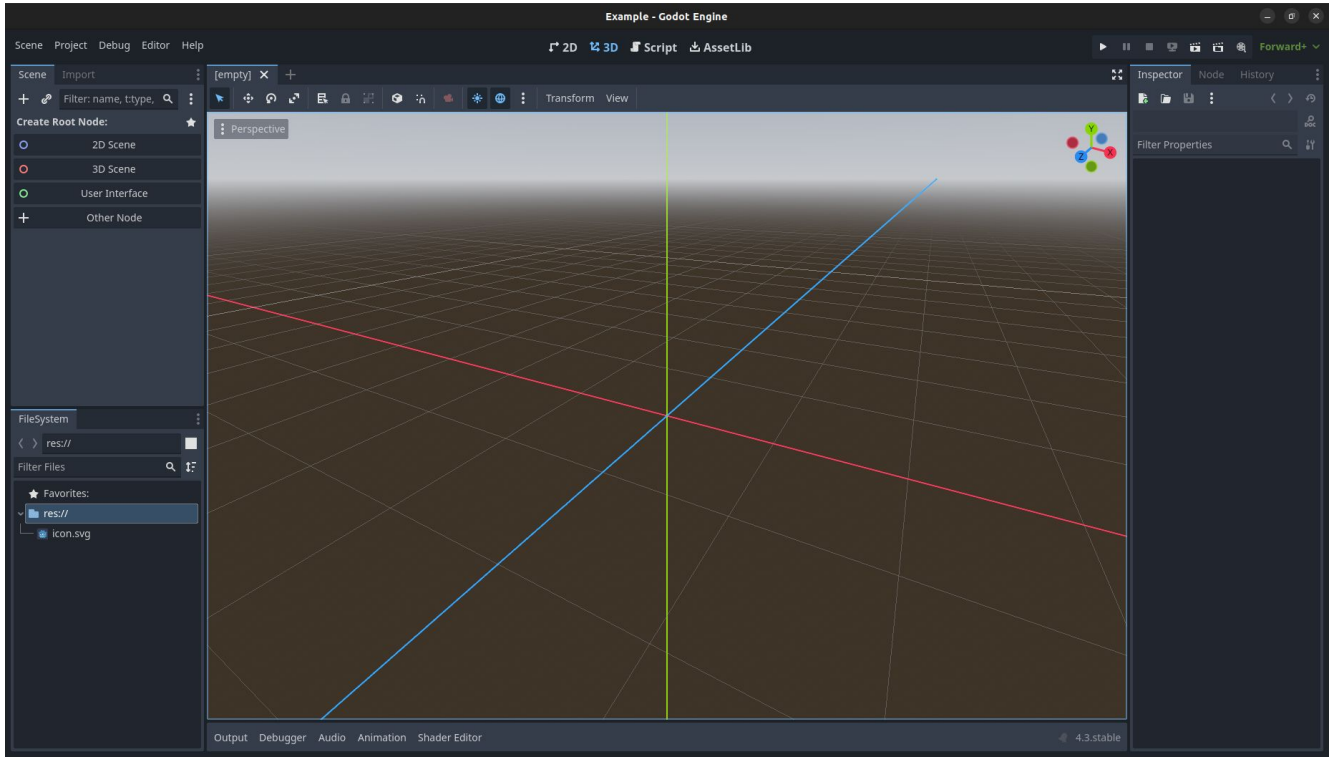
Godot: Project Manager



Godot: Project Manager - Nuovo Progetto



Godot: Interfaccia



Godot: Fondamenti

Ad alto livello Godot segue i principi della programmazione ad oggetti

Per esempio...

Object: Classe base da cui ereditano tutte le classi dell'engine

Tutte le istanze di Object possono avere una istanza di **Script** a loro associata (vedremo dopo)

Godot: Fondamenti

Node: Building Block dell'Engine. Si specializzano poi ulteriormente in:

- ➔ **Node2D**: Oggetto di Gioco 2D
- ➔ **Node3D**: Oggetto di Gioco 3D
- ➔ **Control**: Oggetto per GUI

Scena: Albero di nodi

Quindi, alla fine della fiera, un gioco è un **Albero di Scene**

Godot: Fondamenti

Tutti i nodi hanno le seguenti proprietà:

- Un nome
- Proprietà modificabili
- Aggiornamento ad ogni frame
- Possono essere estesi con nuove proprietà e funzioni
- Possono essere aggiunti ad altri nodi come loro figli

Nota: Ogni nodo eredita il transform del padre (posizione, rotazione, ecc.). Quando viene modificato, viene modificato anche per i figli

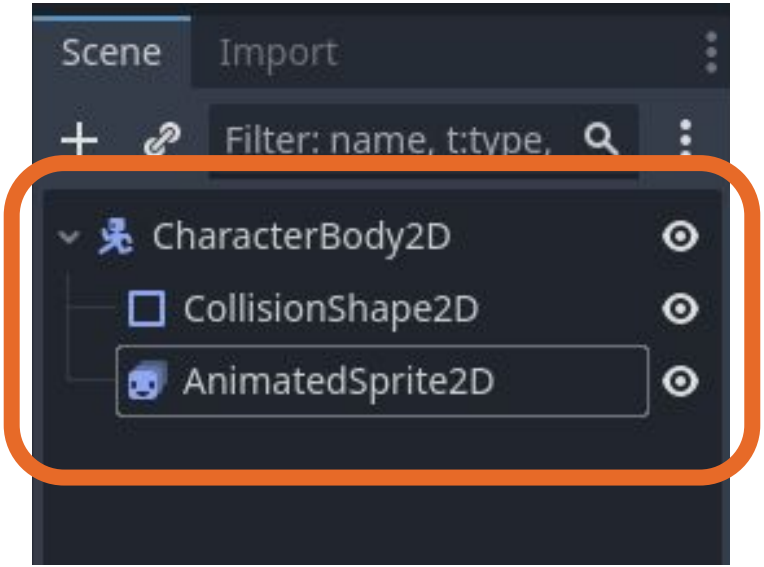
Godot: Fondamenti

Tutte le scene hanno le seguenti proprietà:

- Vi è un nodo radice
- Possono essere salvate e riutilizzate in altre scene
- Possono essere istanziate

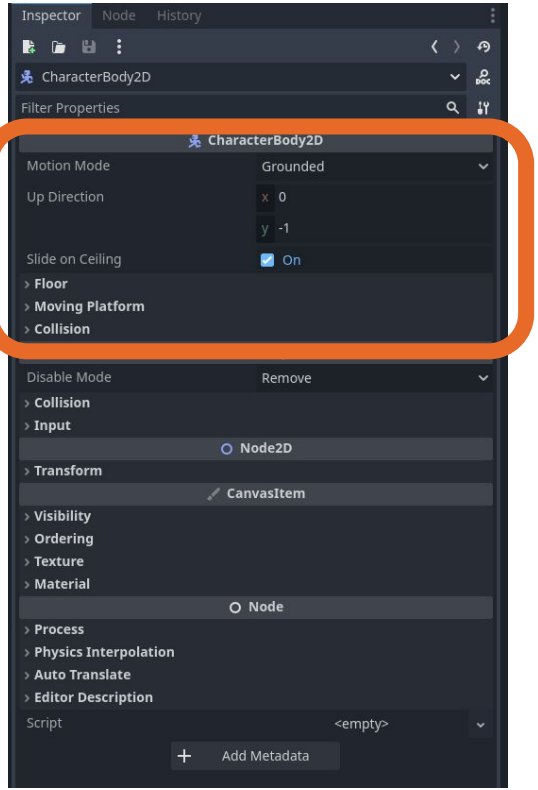
Godot: Esempio di Scena

Albero di nodi



Godot: Proprietà di un Nodo

Attributi della classe CharacterBody2D



Godot: Fondamenti

Vi è infine un'ultima distinzione da fare, che è quella tra scene e **risorse**.

Mentre una scena è un elemento attivo ("fa cose" all'interno del gioco), una risorsa è un elemento passivo (mantiene informazioni sul gioco). Forniscono informazioni che vengono poi usate dalle scene (e.g., immagini, file audio, ...).

Godot: Fondamenti

Una risorsa è implementata con la classe **Resource** che eredita da **RefCounted** (che eredita a sua volta da Object)

RefCounted: mantengono un contatore interno e vengono automaticamente rimosse quando non utilizzate da alcun nodo

Godot: Esempio di Risorsa

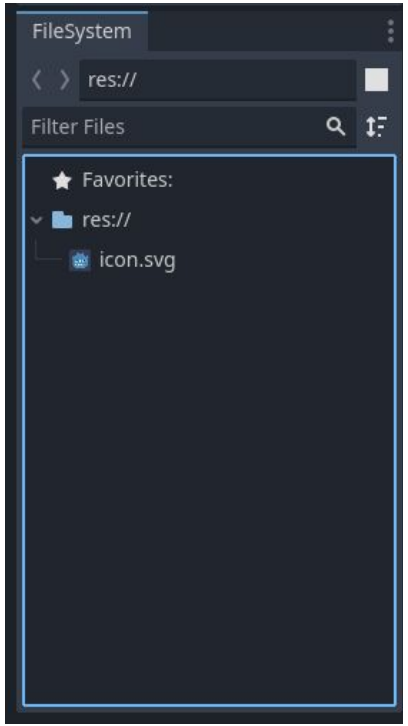
Immagine eredita da risorsa

Godot: Iniziamo!

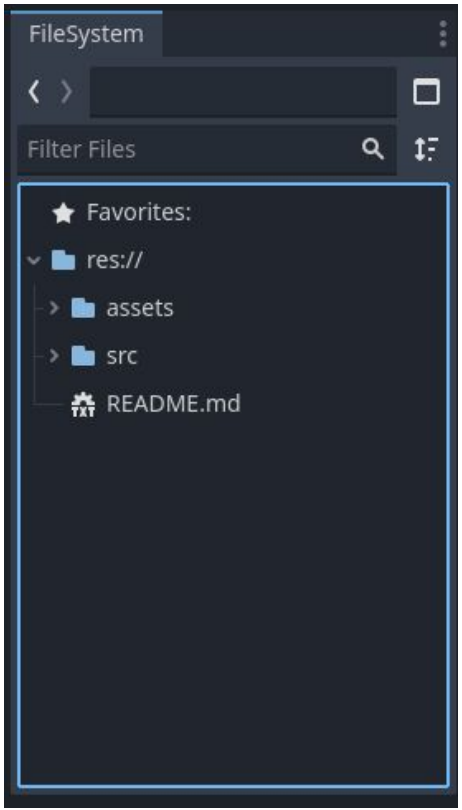
Scaricati gli asset che sono stati forniti precedentemente

Inizializzate il progetto creando delle apposite cartelle dove mantenere le varie risorse che andremo ad utilizzare (e.g. assets, scene, ...)

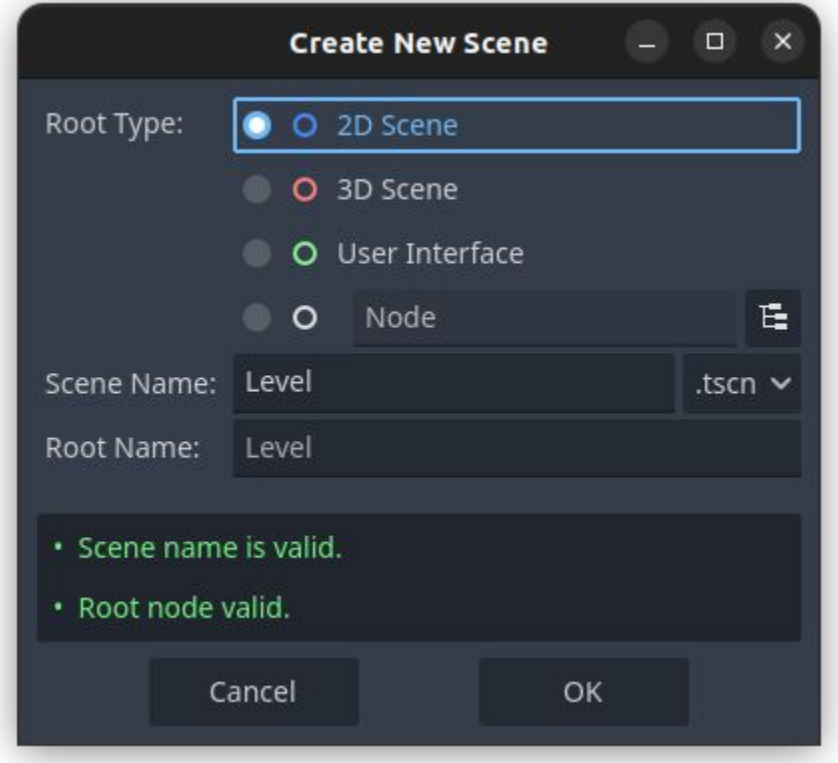
Godot: Organizzare la working directory



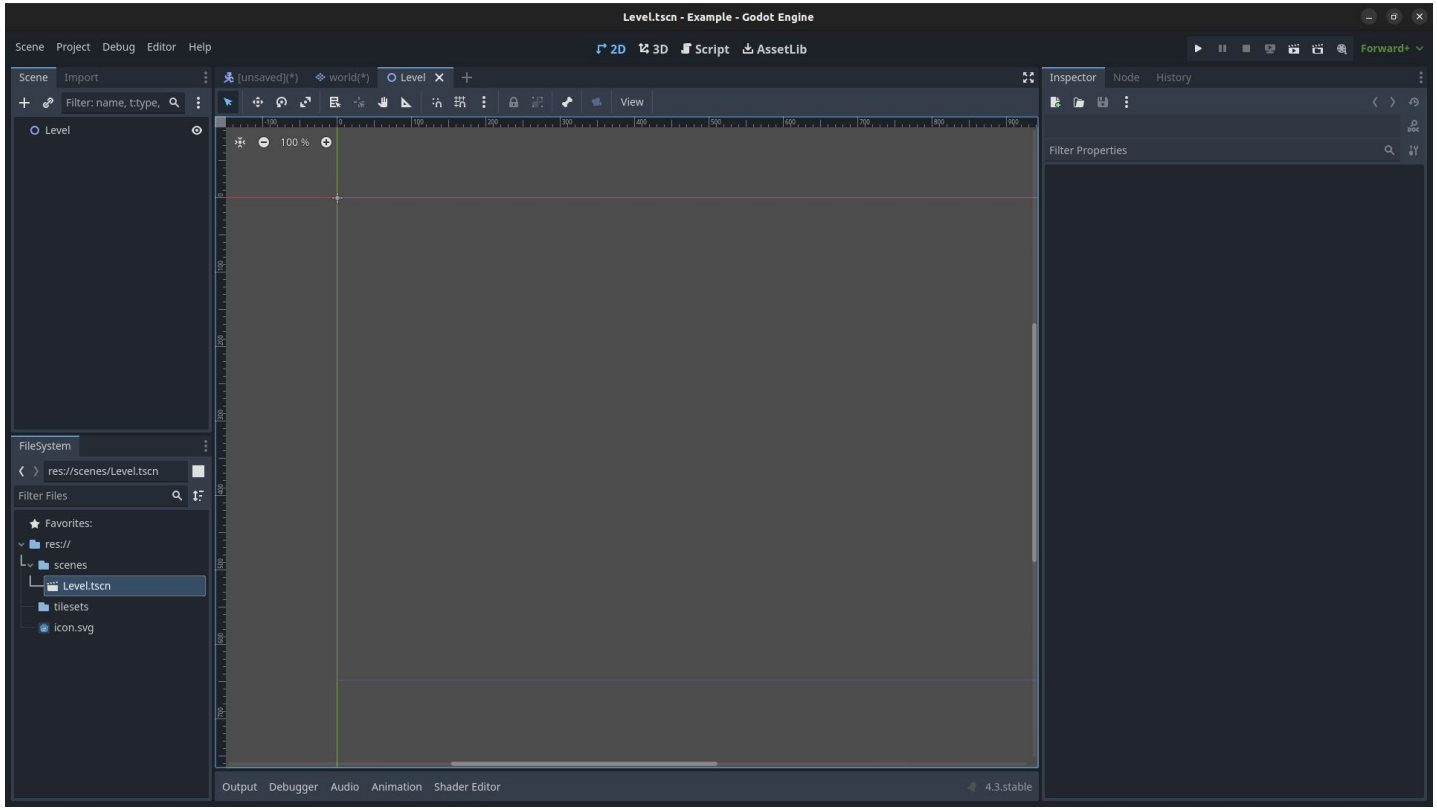
Godot: Organizzare la working directory



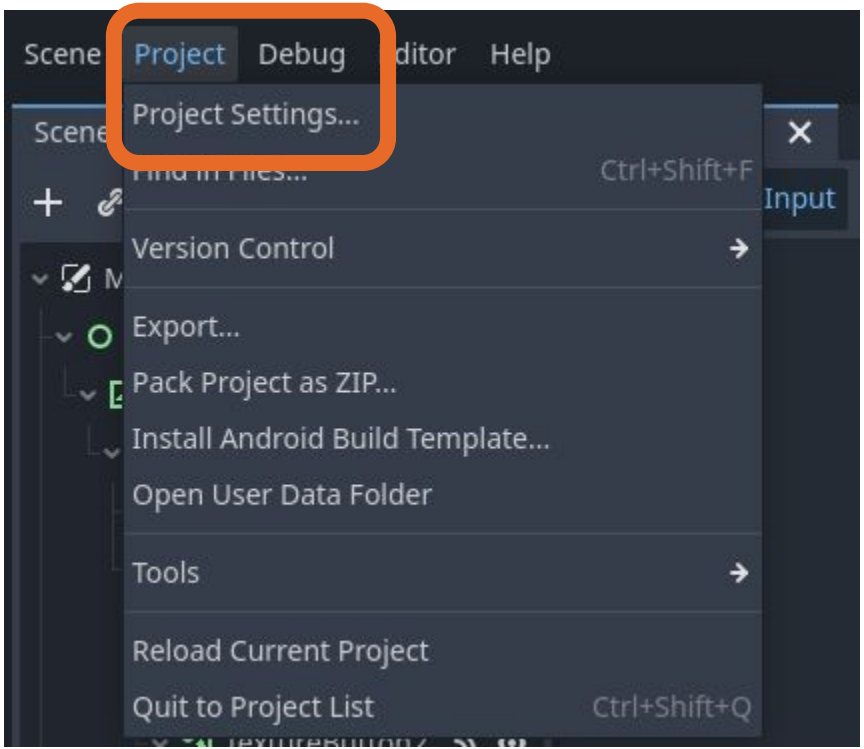
Godot: La prima scena



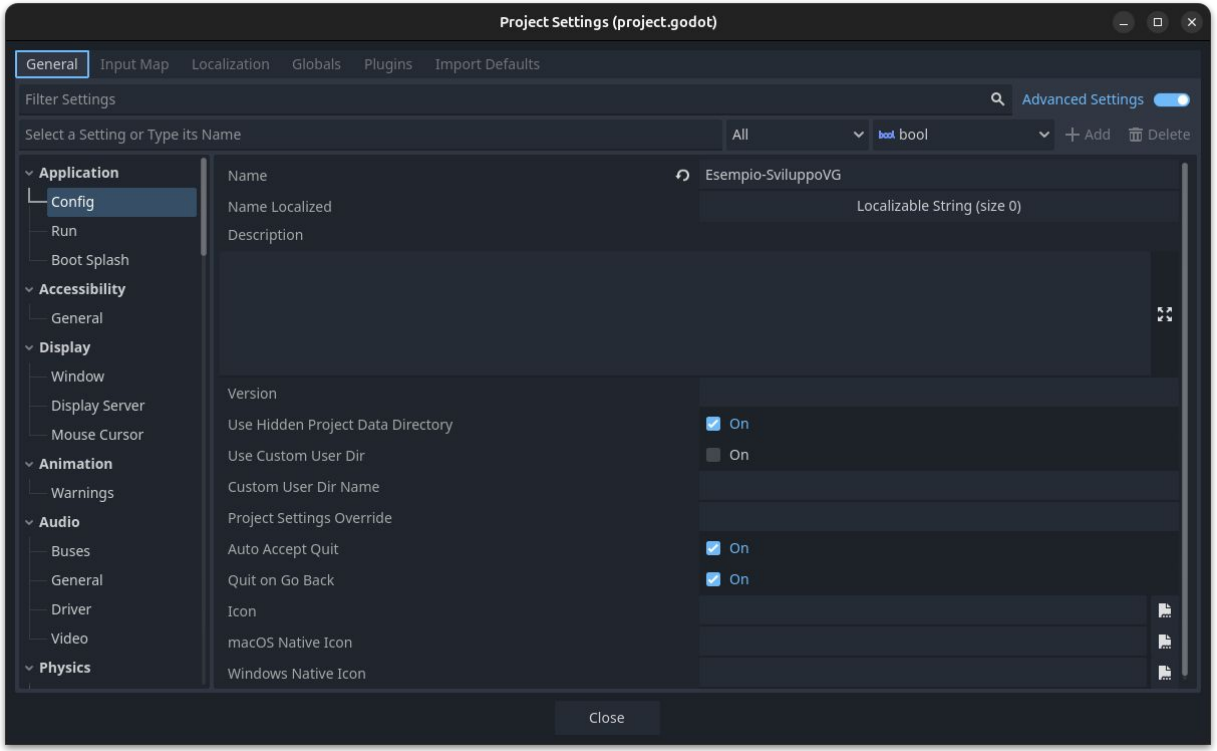
Godot: La prima scena



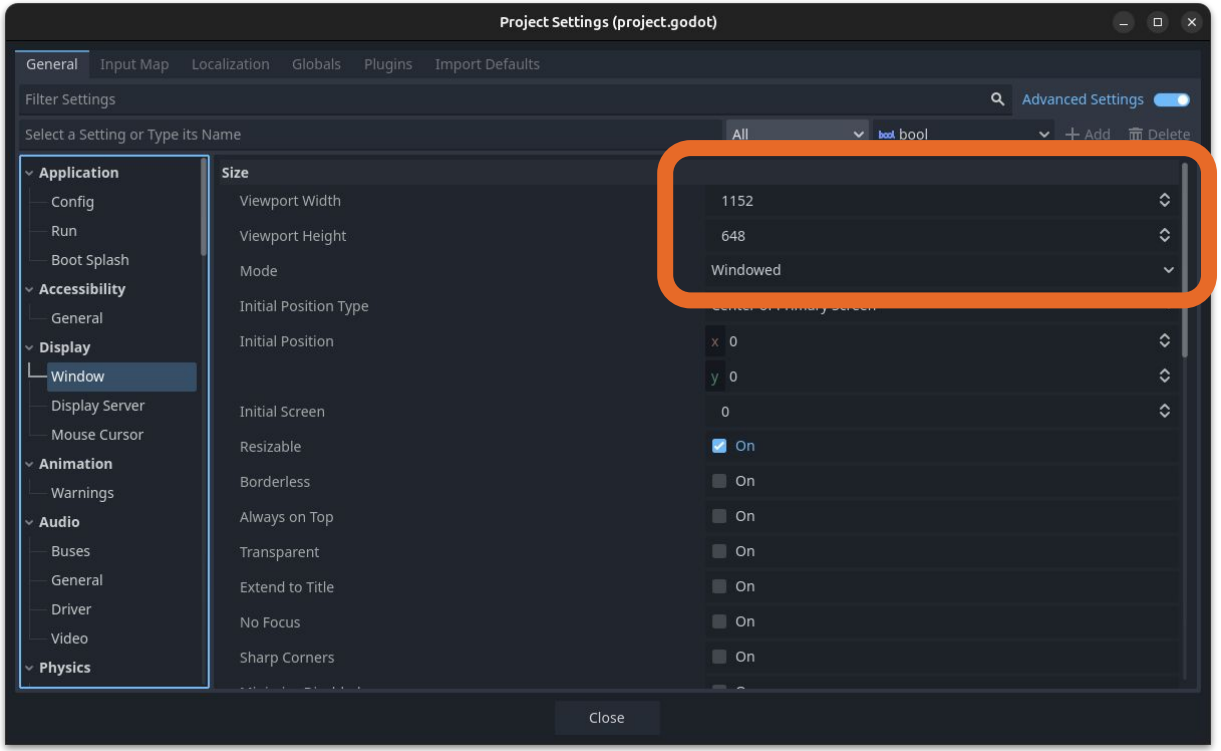
Godot: Project Settings



Godot: Project Settings



Godot: Project Settings



TileSets e TileMaps

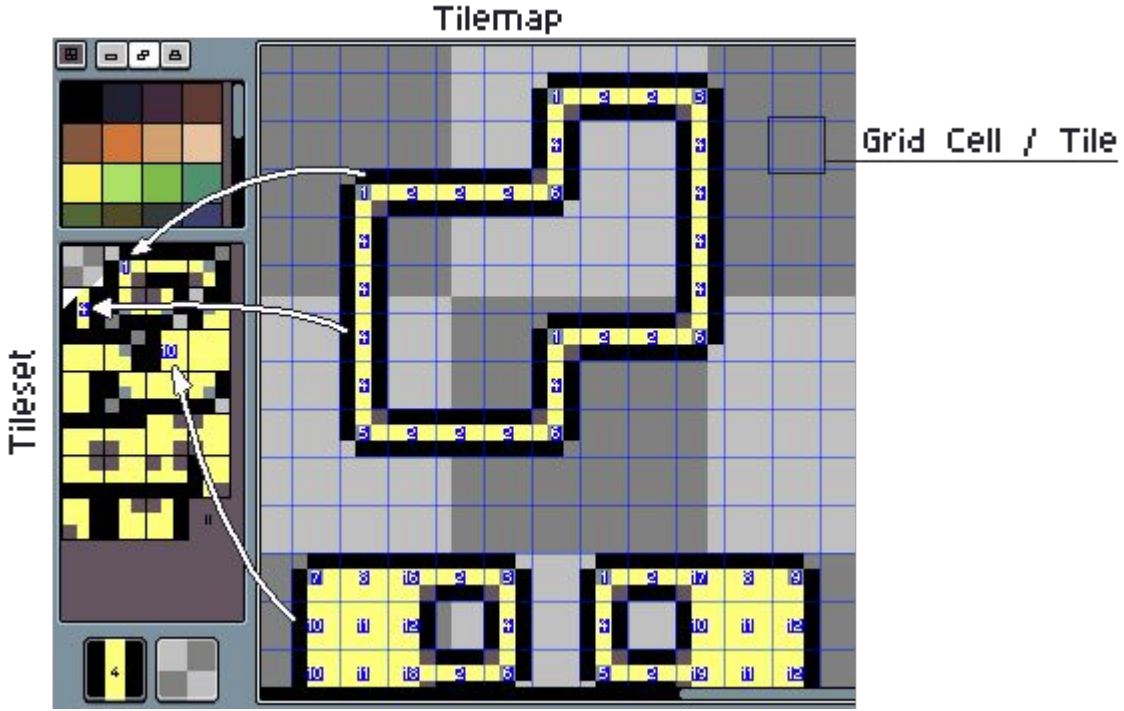
Un **Tile** è una piccola immagine che può essere riutilizzata in diversi punti all'interno del gioco

Un **TileSet** è una collezione di Tile della stessa dimensione

Una **TileMap** è una griglia di riferimenti a Tile presenti nel TileSet

https://www.aseprite.org/docs/tilemap/

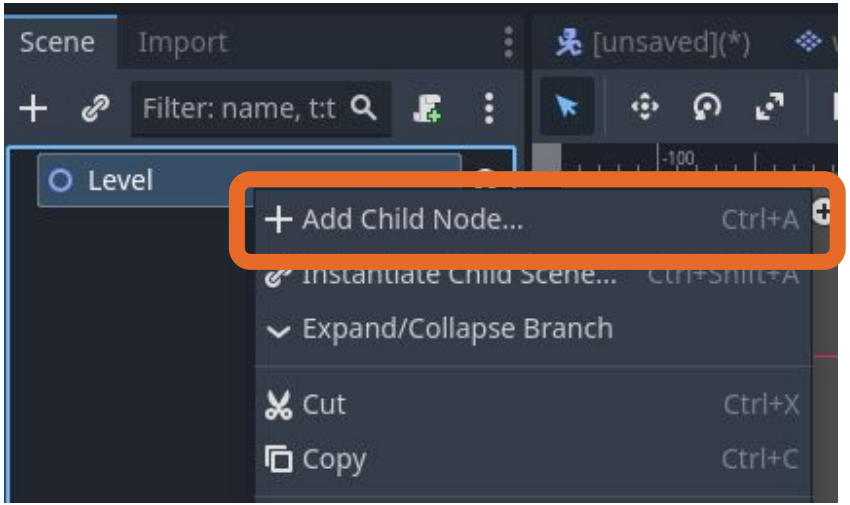
TileSets e TileMaps



https://www.aseprite.org/docs/tilemap/

Godot: TileSets e TileMaps

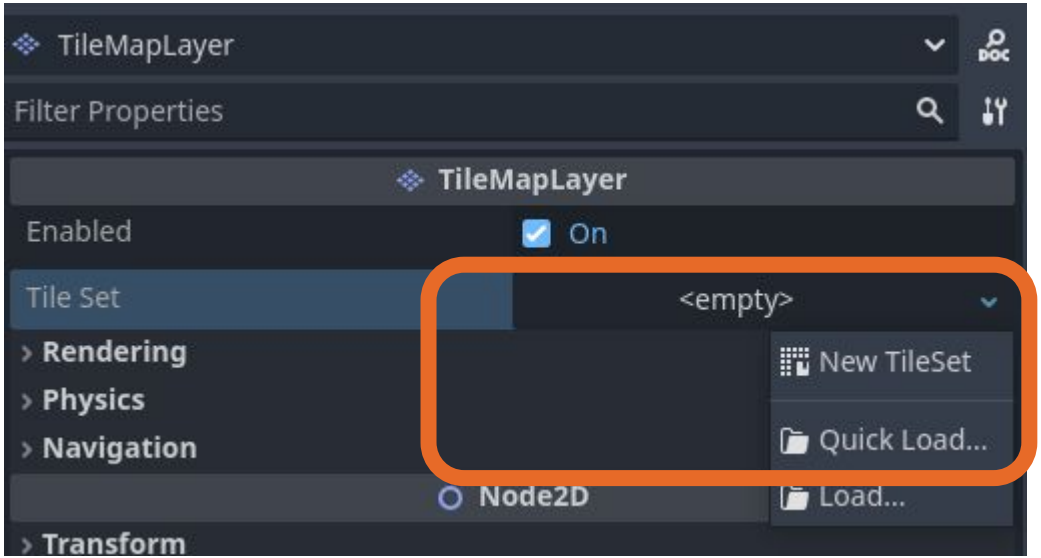
Godot supporta queste funzionalità attraverso i nodi **TileMapLayer** e **TileSet**



Nella scena è presente un Root Node (Level) a cui possiamo aggiungere dei nodi figli. Aggiungiamo quindi un nodo TileMapLayer!

Godot: TileSets e TileMaps

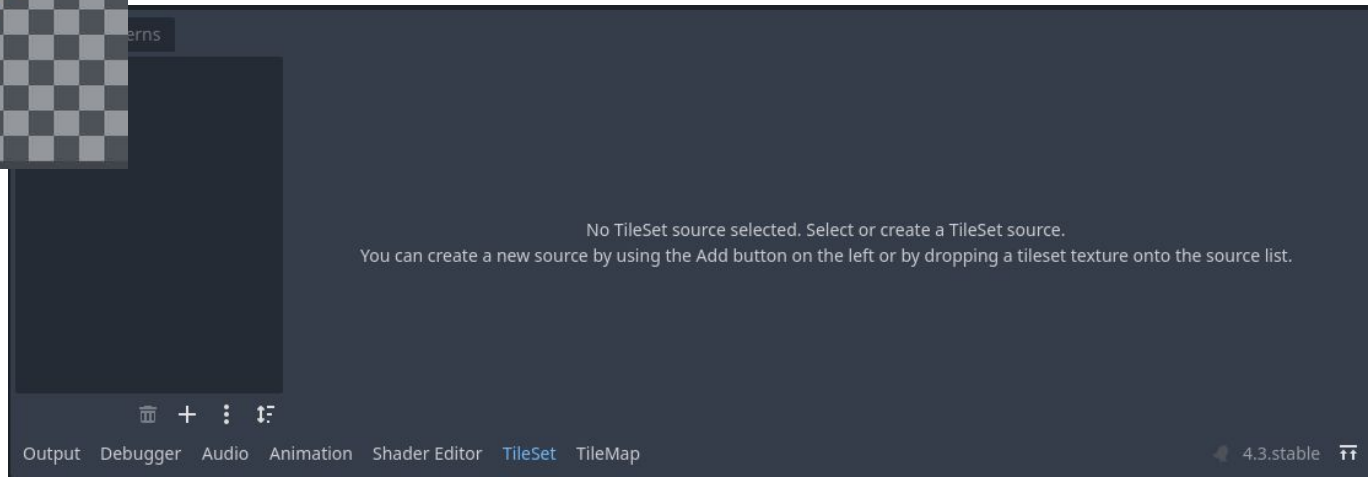
Godot supporta queste funzionalità attraverso i nodi **TileMapLayer** e **TileSet**



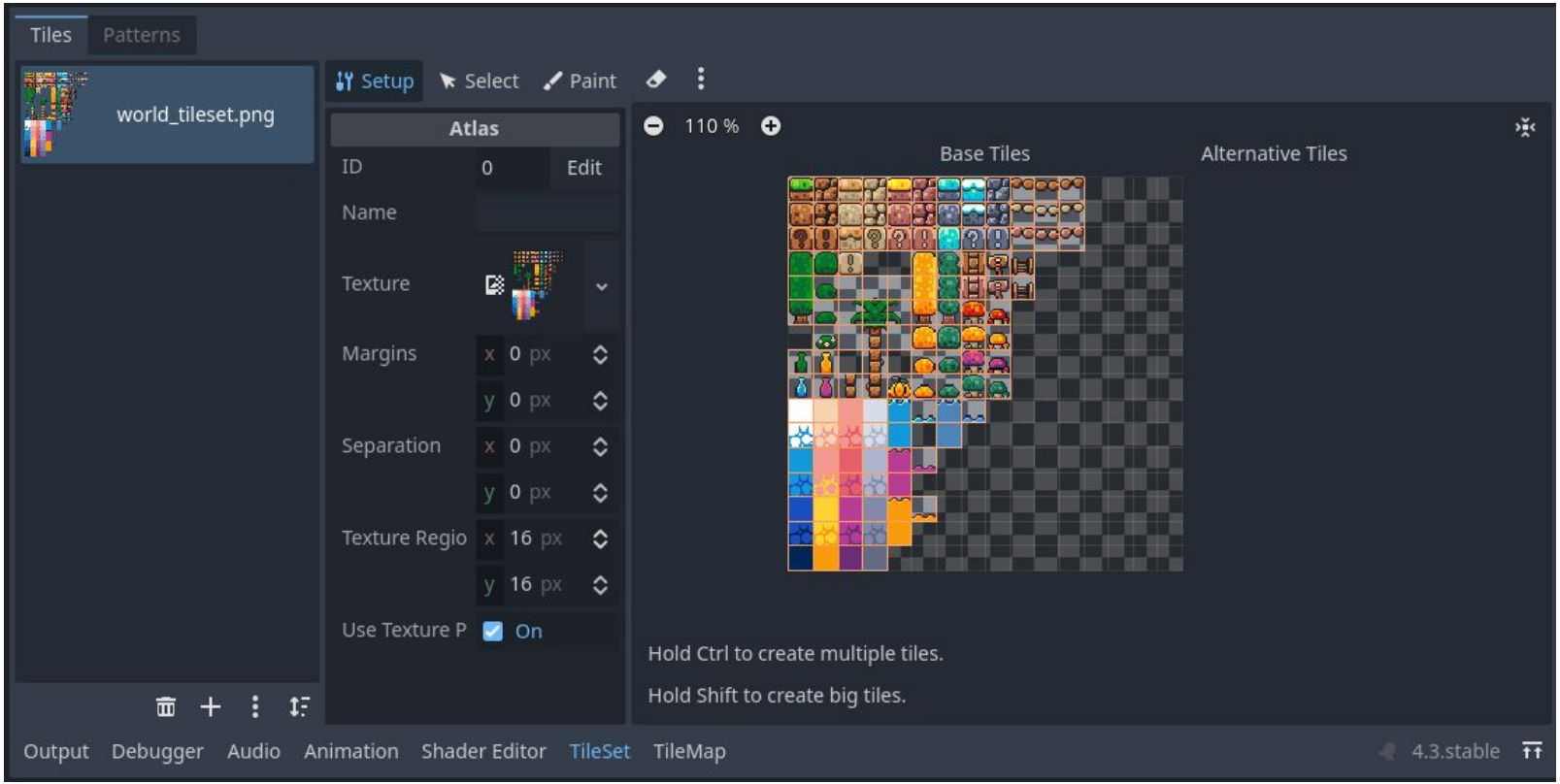
Creiamo un nuovo TileSet che utilizzeremo per questa TileMap

Godot: TileSets e TileMaps

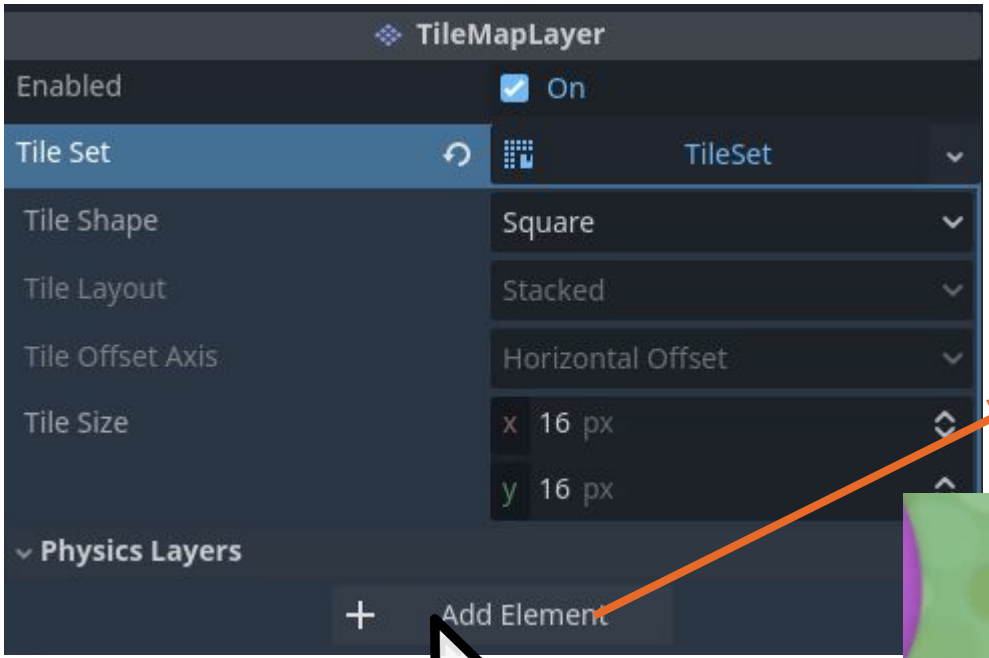
Trascinate il TileSet in formato .png dalla cartella assets nello spazio TileSet dedicato

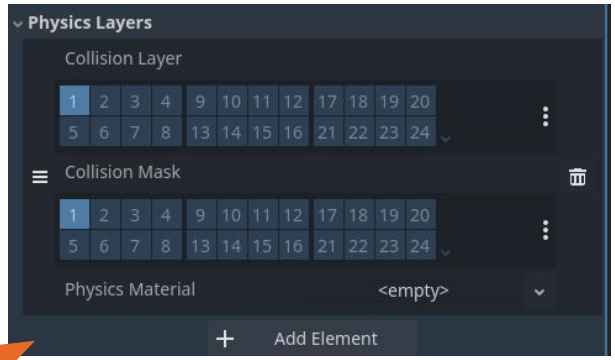


Godot: TileSets e TileMaps



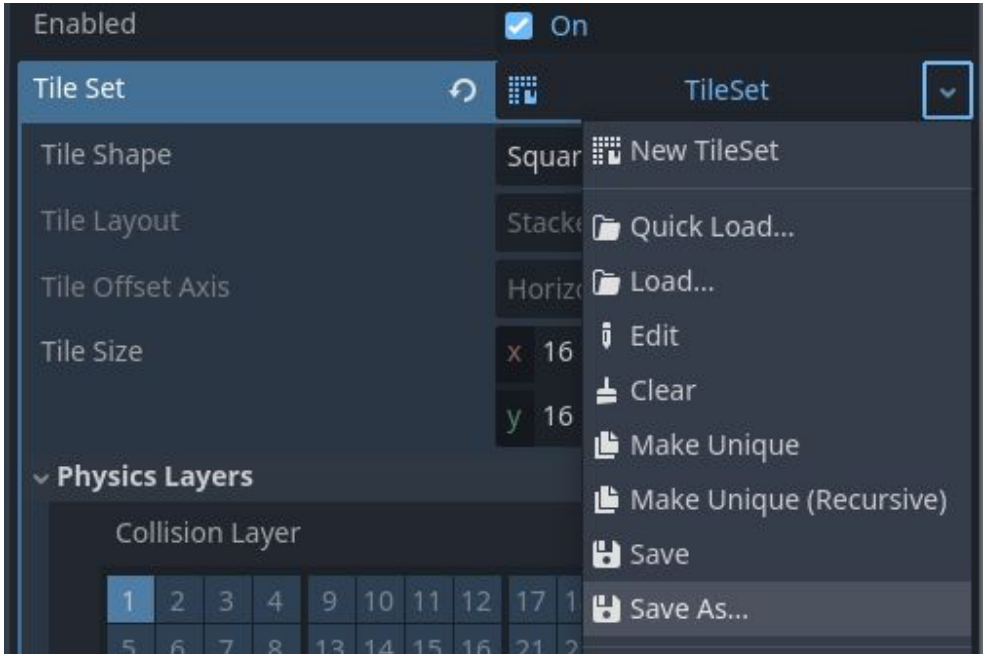
Godot: TileSets e TileMaps



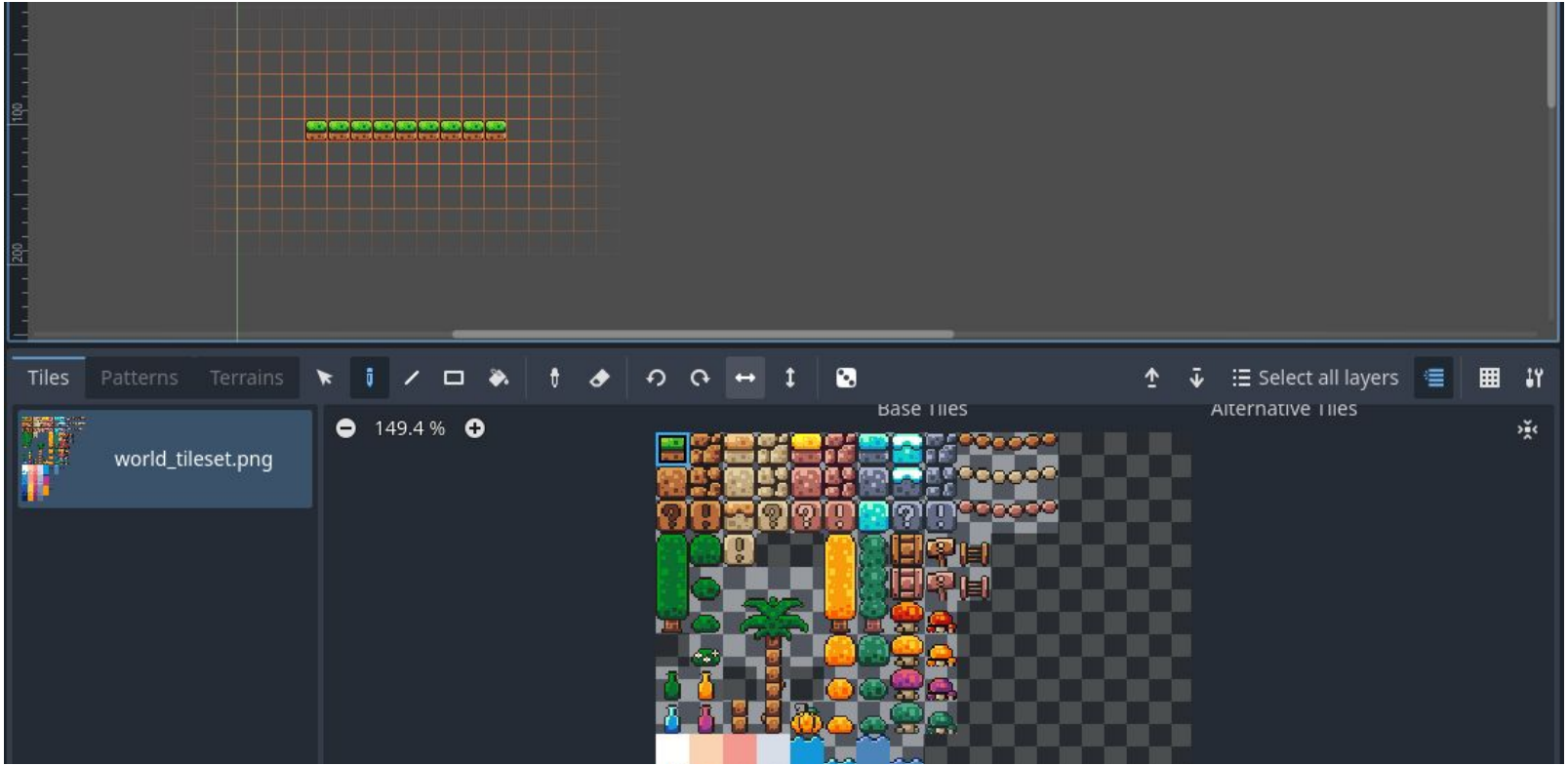


Godot: TileSets e TileMaps

N.B. Un TileSet è una resource, è quindi possibile salvarlo per utilizzarlo in altri TileMapLayer

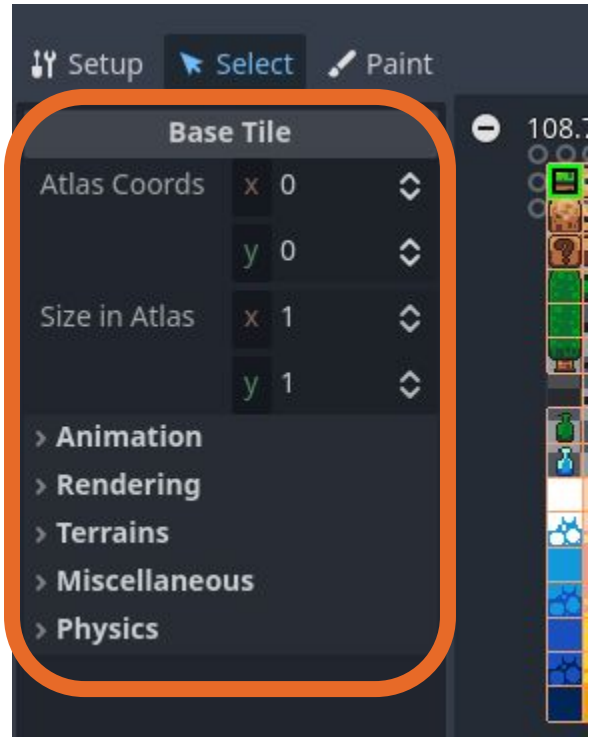


Godot: TileSets e TileMaps



Godot: TileSets e TileMaps

Ogni Tile ha delle proprietà che possono essere modificate dall'editor



Godot: TileSets e TileMaps

Inoltre, è possibile utilizzare una funzionalità chiamata autotiling

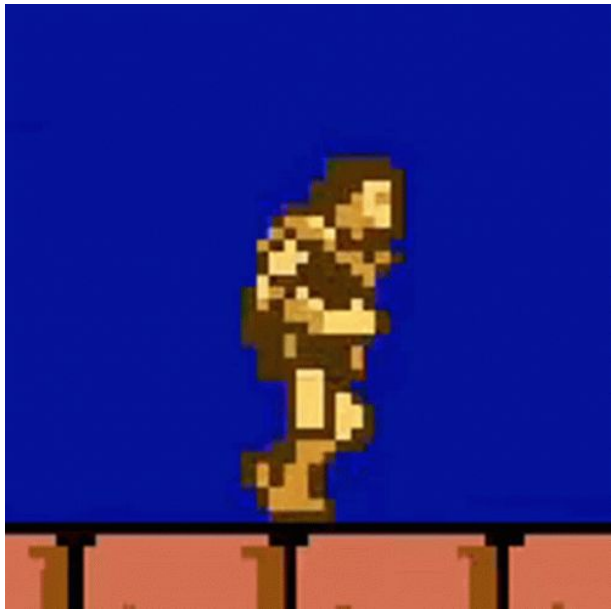
Non andremo nel dettaglio di questa funzionalità, l'idea è di impostare le tile in modo che vengano automaticamente selezionate quelle corrette (e.g. tile ad angolo)

Godot: Collisioni

Nei giochi vi sono spesso situazioni in cui vogliamo controllare se vi sono collisioni tra oggetti di gioco

In

- **Collision**
  Due oggetti di gioco entrano in contatto
- **Collision Response**:
  Reazione a seguito di una collisione



Godot: Collisioni

In Godot, vi sono 4 nodi che possono essere usati per gestire le collisioni (Collision Objects):

- **Area2D**: controlla se vi sono collisioni tra oggetti in una specifica zona
- **StaticBody2D**: permette collision detection; non viene mosso a seguito di collisioni
- **RigidBody2D**: movimento gestito automaticamente dal physics engine
- **CharacterBody2D**: permette collision detection; il movimento deve essere implementato da script

Godot: Collisioni

In Godot, le collisioni vengono gestite dal Physics Engine

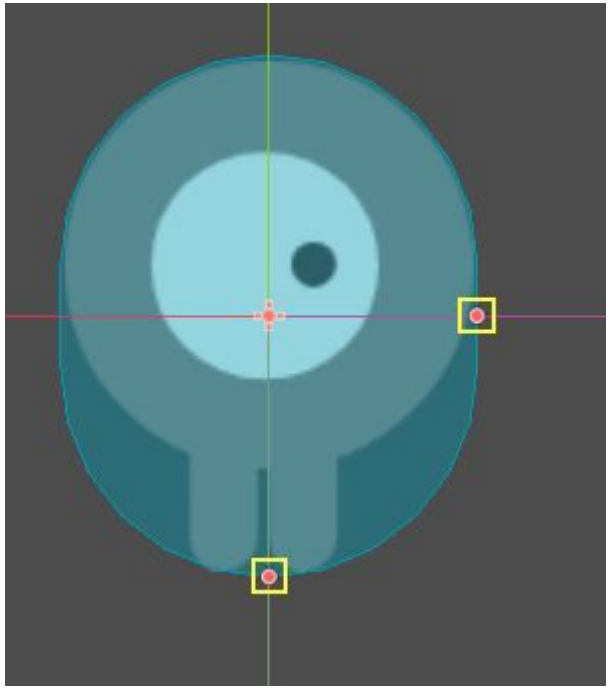
Le collisioni sono **essenziali** per gestire l'intero gioco

Anche il movimento del giocatore su terreni e piattaforme è gestito attraverso le collisioni

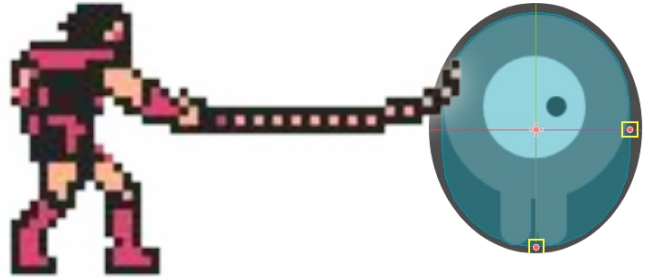
Godot: Collision Shape

Ogni Collision Object può poi avere uno o più **Shape2D** nodi figli

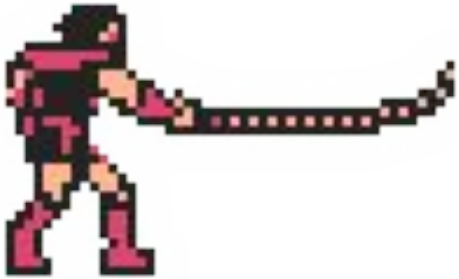
I nodi Shape2D vengono utilizzati per definire i limiti di collisione dell'oggetto e per rilevare i contatti con altri oggetti

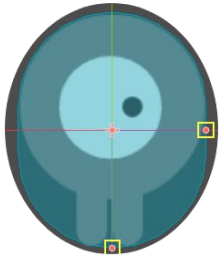


Godot: Collision Shape



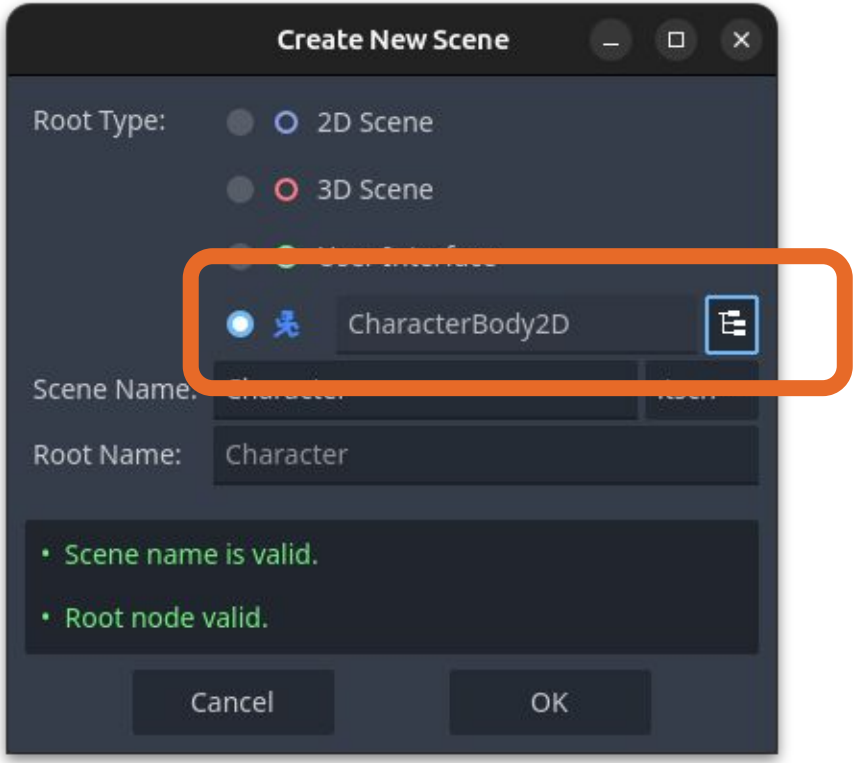
Collision Detected! Something happens





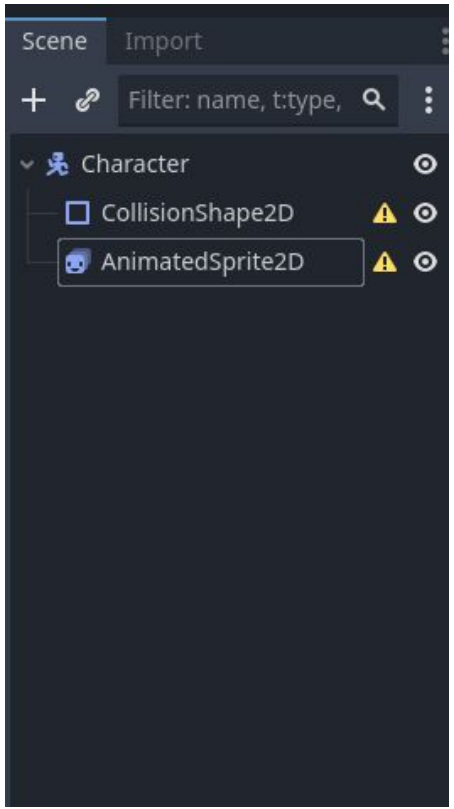
No Collision! Nothing happens

Godot: Creare la scena del personaggio



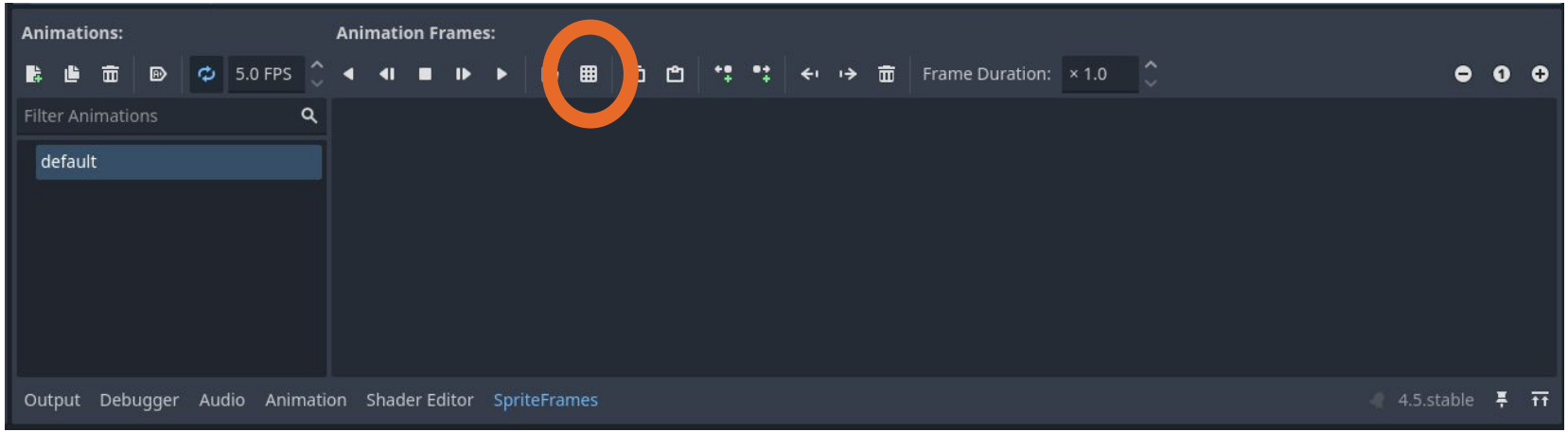
N.B. Definiamo il root node come un nodo CharacterBody2D

Godot: Creare la scena del personaggio



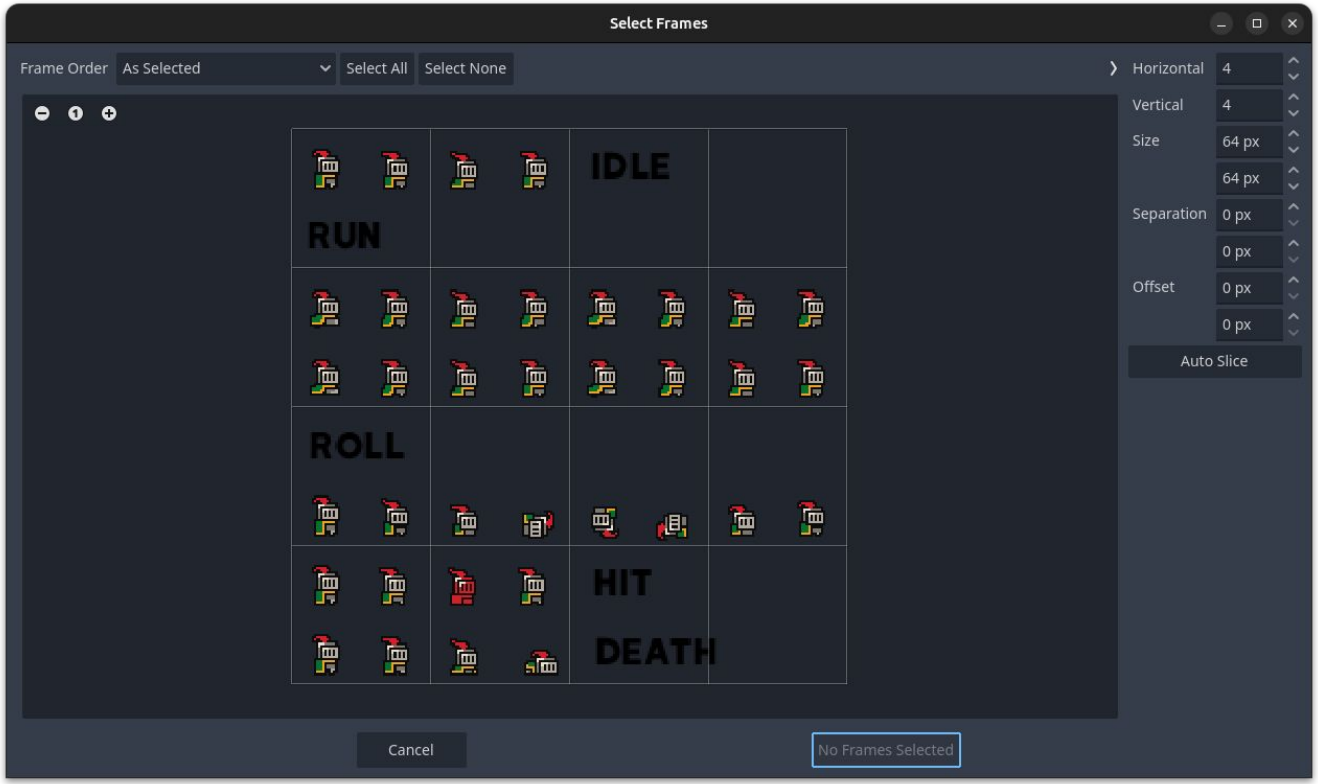
Godot: Creare la scena del personaggio

Per importare i character sheet e selezionare le animazioni



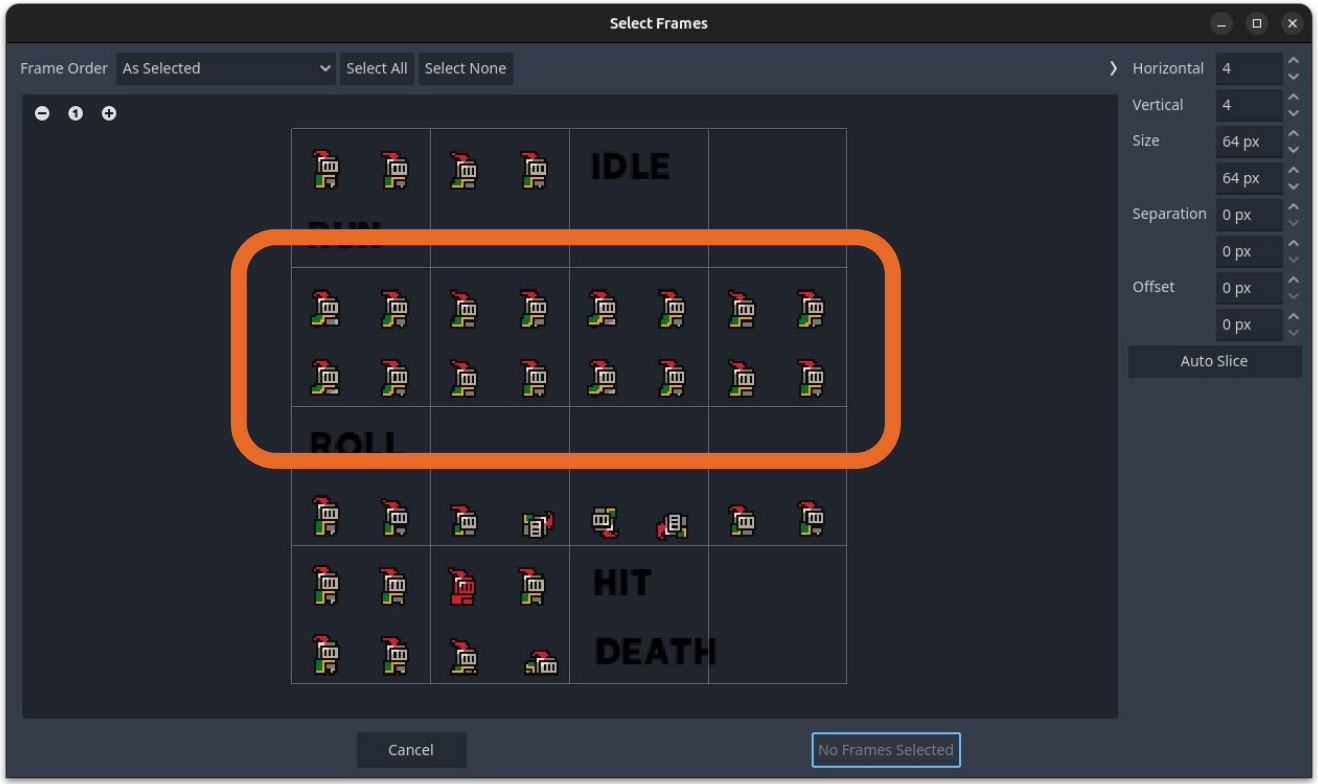
Godot: Creare la scena del personaggio

Possibile selezionare i frame dell'animazione specifica dal character sheet

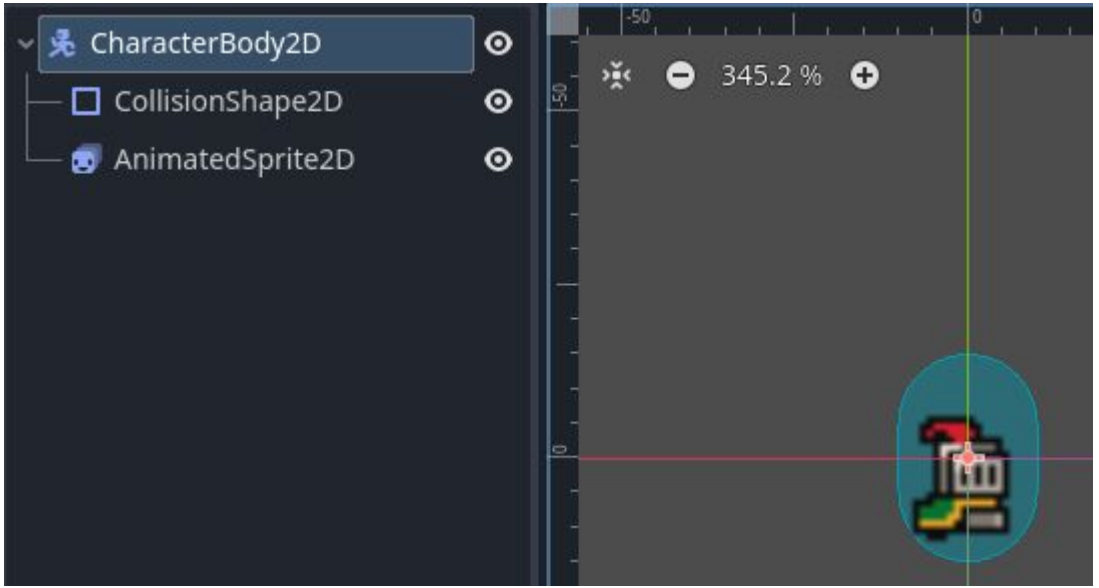


Godot: Creare la scena del personaggio

Possibile selezionare i frame dell'animazione specifica dal character sheet

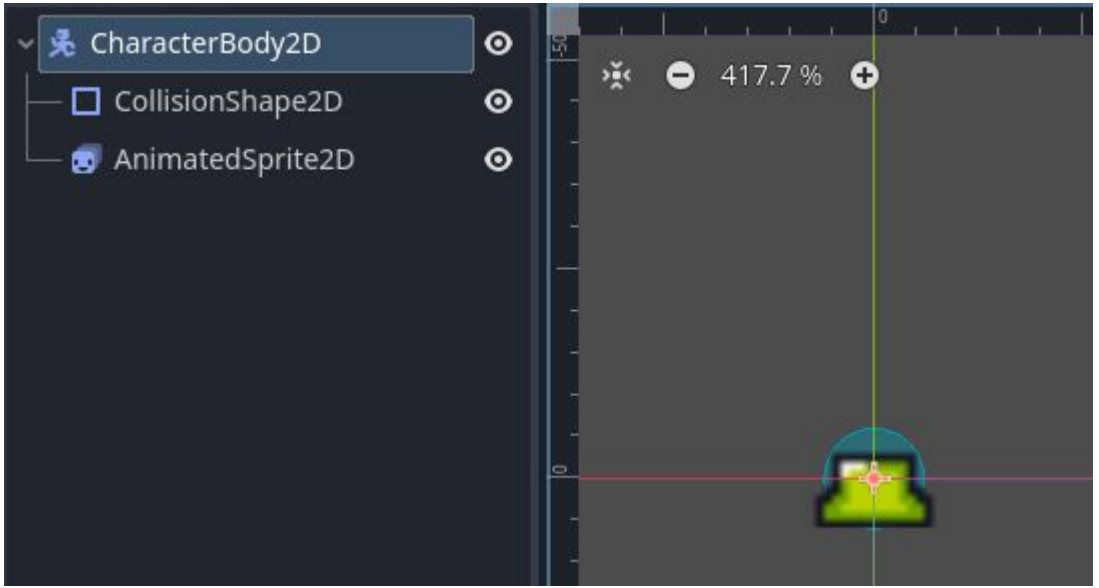


Godot: Creare la scena del personaggio



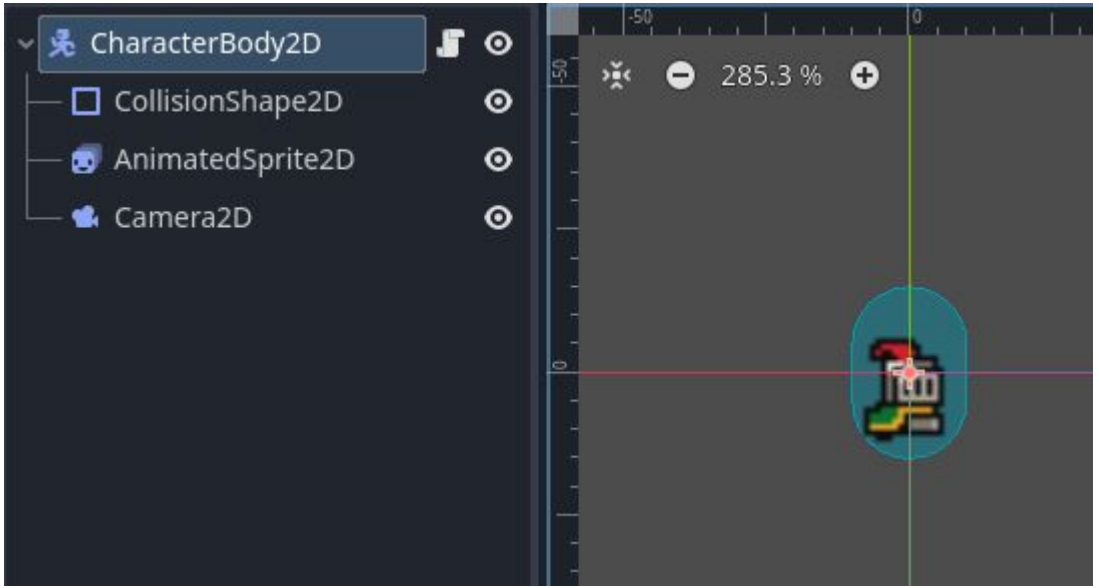
Godot: Creare la scena del nemico

La stessa cosa ma cambiamo lo sprite e la collision shape



Godot: Creare la scena del personaggio

Aggiungete un nodo Camera2D al personaggio principale per gestire automaticamente la camera del gioco



Godot: Creare una scena per il giocatore

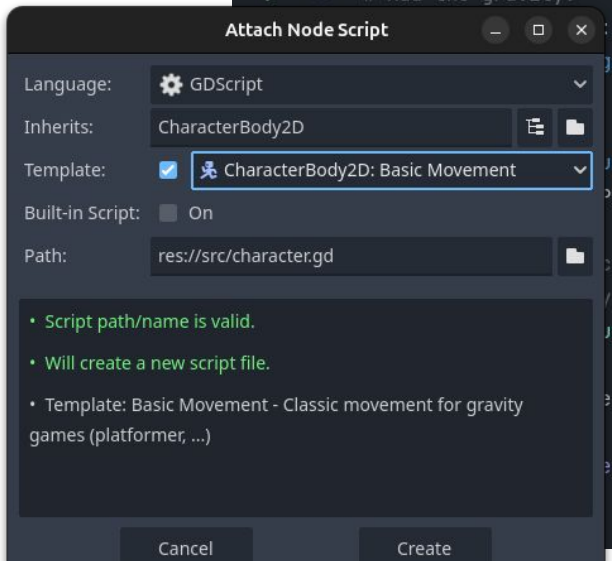
I nodi fondamentali per un scena dedicata al giocatore sono i seguenti:

- → **CharacterBody2D (root)** (abbiamo visto prima)
- → **CollisionShape2D**: specializza Shape2D
- → **AnimatedSprite2D**: Permette di impostare delle animazioni usando più frame (queste animazioni sono poi identificate da nomi)
- → **Camera2D**
- → **RayCast2D** *(opzionale)*

Godot: A breve vedremo...

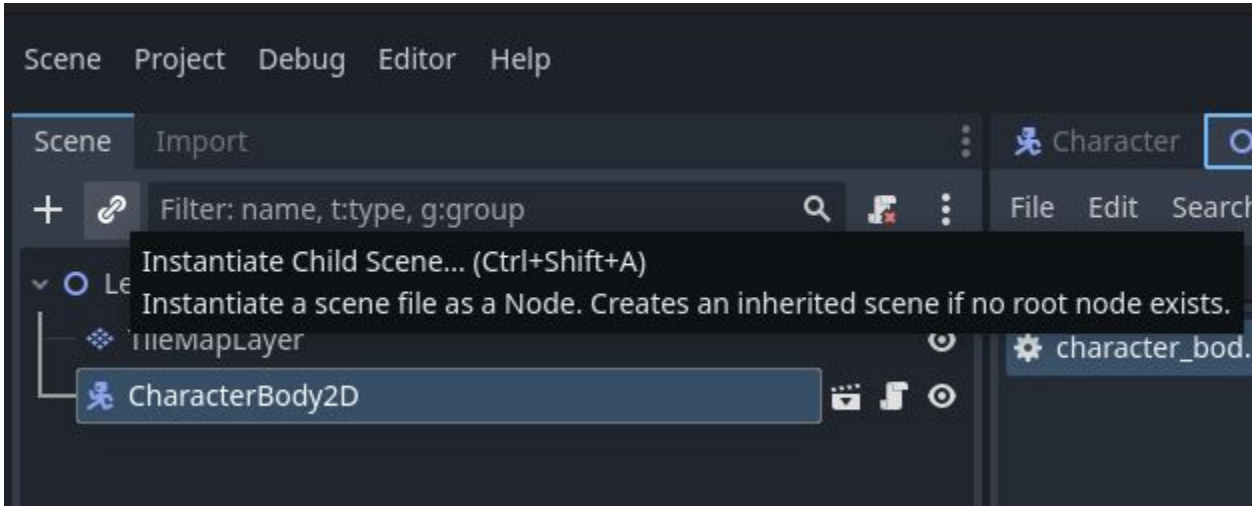
Ok, ma come faccio a far muovere questi personaggi?

```
1   extends CharacterBody2D
2
3
4   const SPEED = 300.0
5   const JUMP_VELOCITY = -400.0
6
7
8   func _physics_process(delta: float) -> void:
9       # Add the gravity.
```



Godot: Istanziare una scena

Per istanziare una scena salvata in memoria all'interno di un'altra scena è sufficiente usare la seguente funzionalità dall'interfaccia grafica



Godot: CharacterBody2D

I nodi CharacterBody2D sono ideali per qualsiasi entità che debba essere mossa da script (e.g. giocatore)

Per supportare il movimento, vi sono due possibili funzioni:

- ➔ **move_and_slide**
- ➔ **move_and_collide**

Entrambe le funzioni permettono di spostare il CharacterBody2D verso un vettore direzione e di rilevare collisioni

Godot: CharacterBody2D

La principale differenza tra i due metodi è nella gestione delle collisioni

move_and_collide non effettua alcuna operazione di collisione in automatico (collisione = ferma movimento)

move_and_slide è un caso specifico molto comune in cui vogliamo che un CharacterBody2D scivoli sull'oggetto con cui sta collidendo (collisione = continua movimento)

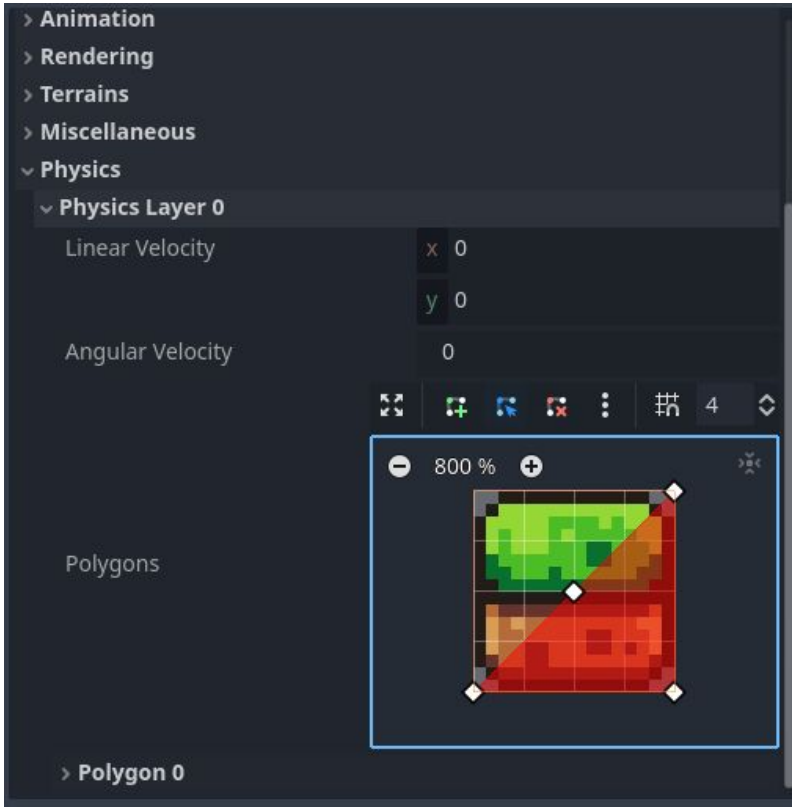
Godot: Callbacks

Il physics engine esegue il processing ad un intervallo di tempo fissato (60 volte al secondo). Questo intervallo di tempo è diverso dal frame rate (numero di frame processati al secondo)

- → **Idle Processing**: il nodo viene aggiornato ad ogni frame, quindi dipende da Hardware e ottimizzazione
- → **Physics Processing**: il nodo viene aggiornato ad un intervallo fisso di 60 volte al secondo

Godot: Collisioni con Tile

Per ogni Tile è poi possibile definire una Shape per le collisioni!



Godot: Collision Layer e Collision Mask

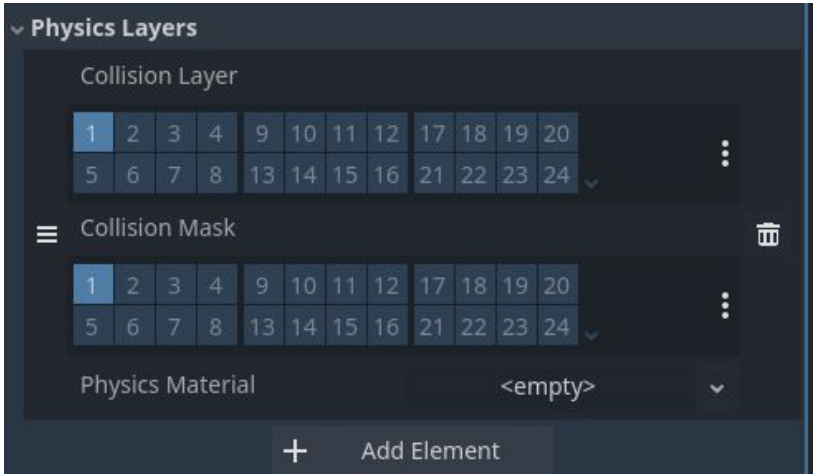
Tutti i Collision Object interagiscono su diversi layer

Questo è necessario in modo tale che possano essere gestite interazioni specifiche tra oggetti di determinate tipologie

Godot: Collision Layer e Collision Mask

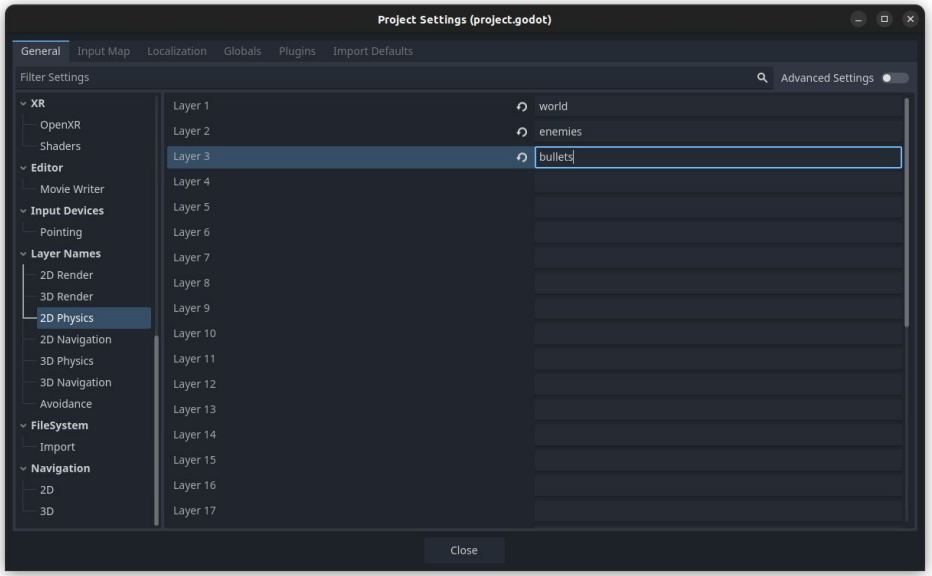
Per gestire queste situazioni, è possibile impostare le seguenti proprietà:

- → **Collision Layer**: I layer in cui gli oggetti di gioco si trovano
- → **Collision Mask**: I layer da analizzare per collisioni



Godot: Collision Layer e Collision Mask

Viene inoltre data la possibilità di dare dei nomi ai vari Collision Layer (per semplificare l'organizzazione)



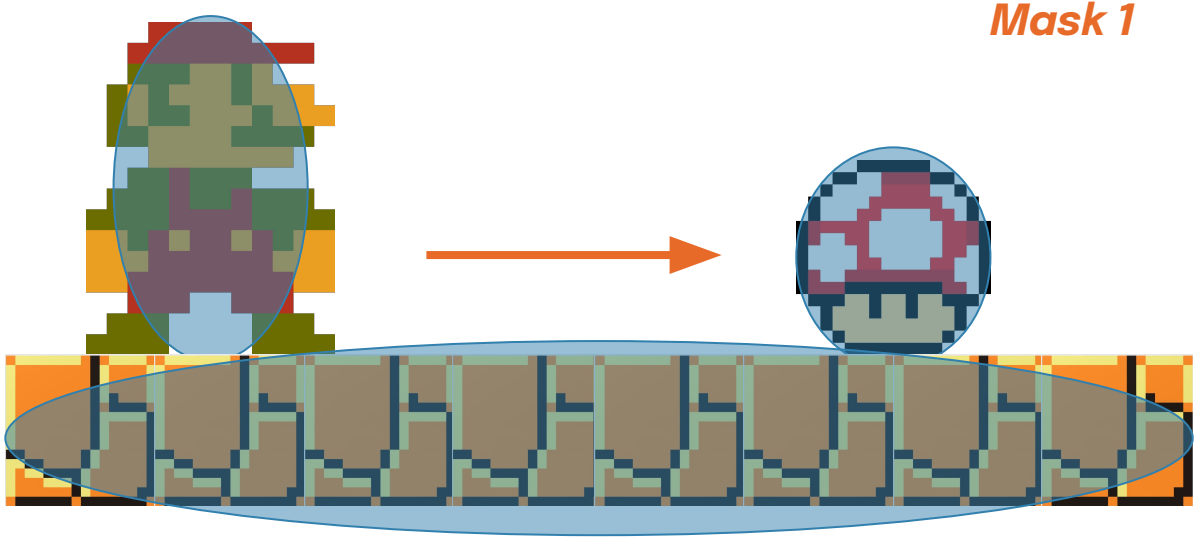
Godot: Collision Layer e Collision Mask

Esempio:

Cosa succede?

Layer 1
Mask 1

Layer 1
Mask 1

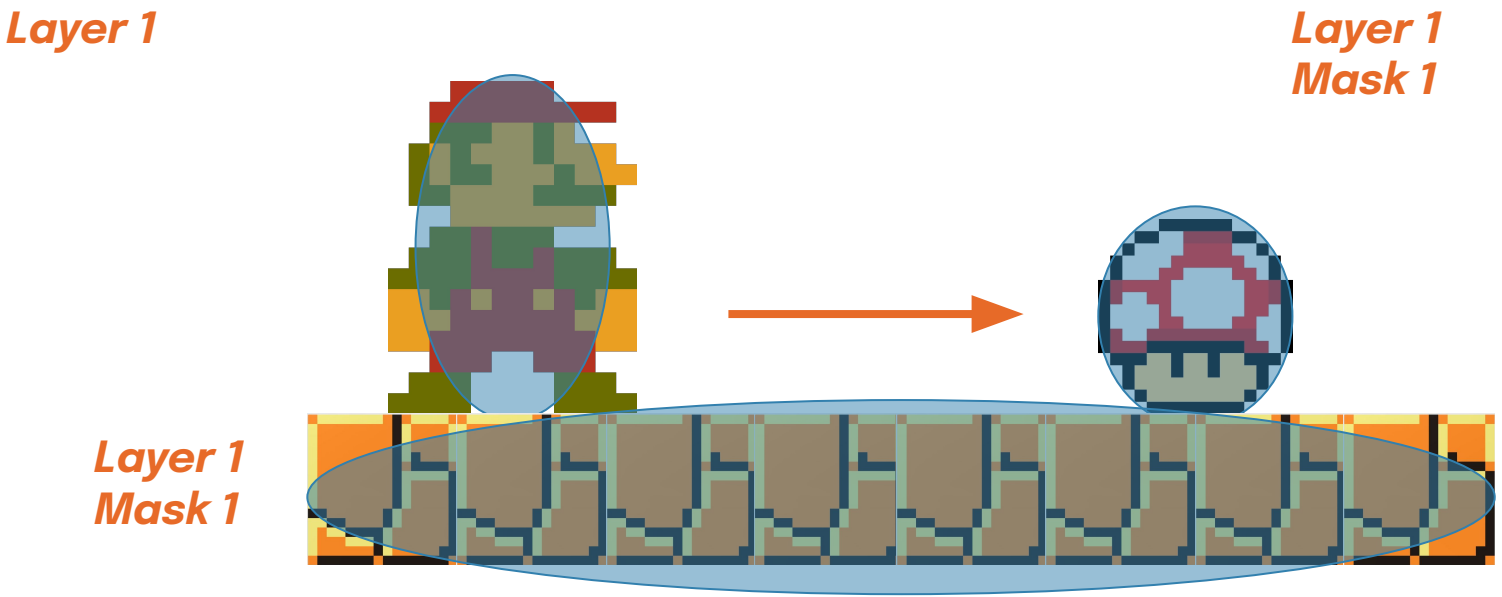


Layer 1
Mask 1

Godot: Collision Layer e Collision Mask

Esempio:

Cosa succede?



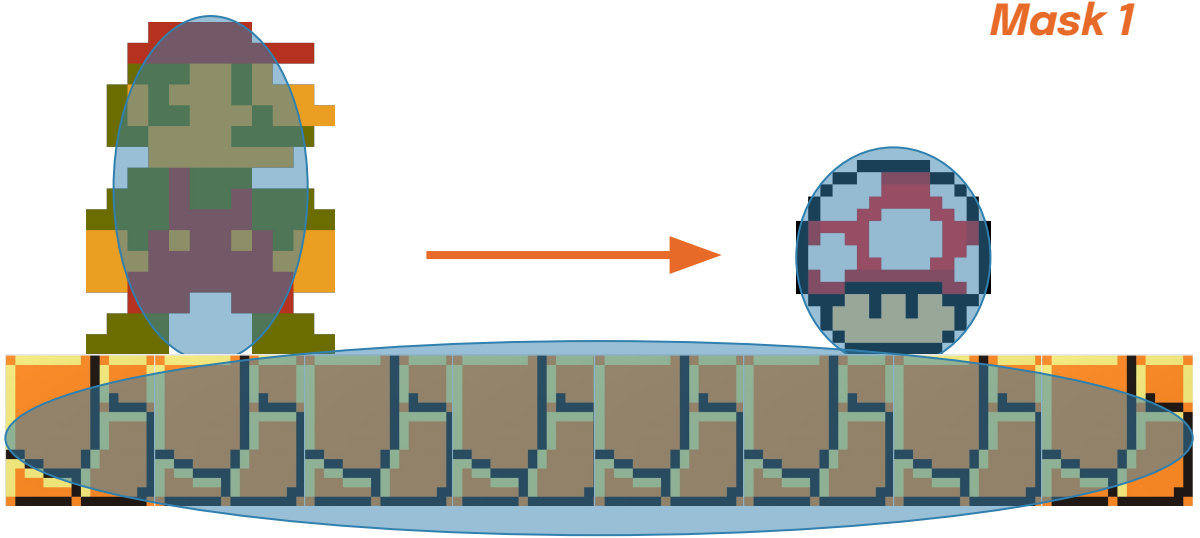
Godot: Collision Layer e Collision Mask

Esempio:

Cosa succede?

Mask 1

Layer 1
Mask 1

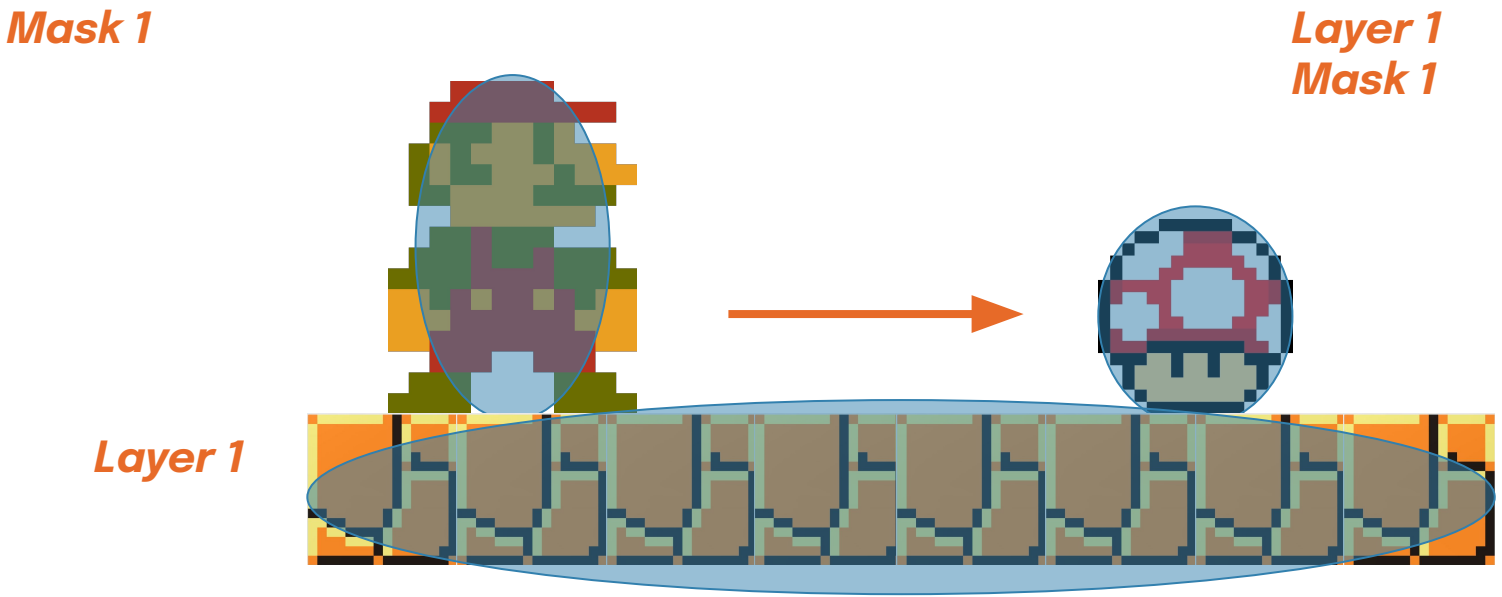


Layer 1
Mask 1

Godot: Collision Layer e Collision Mask

Esempio:

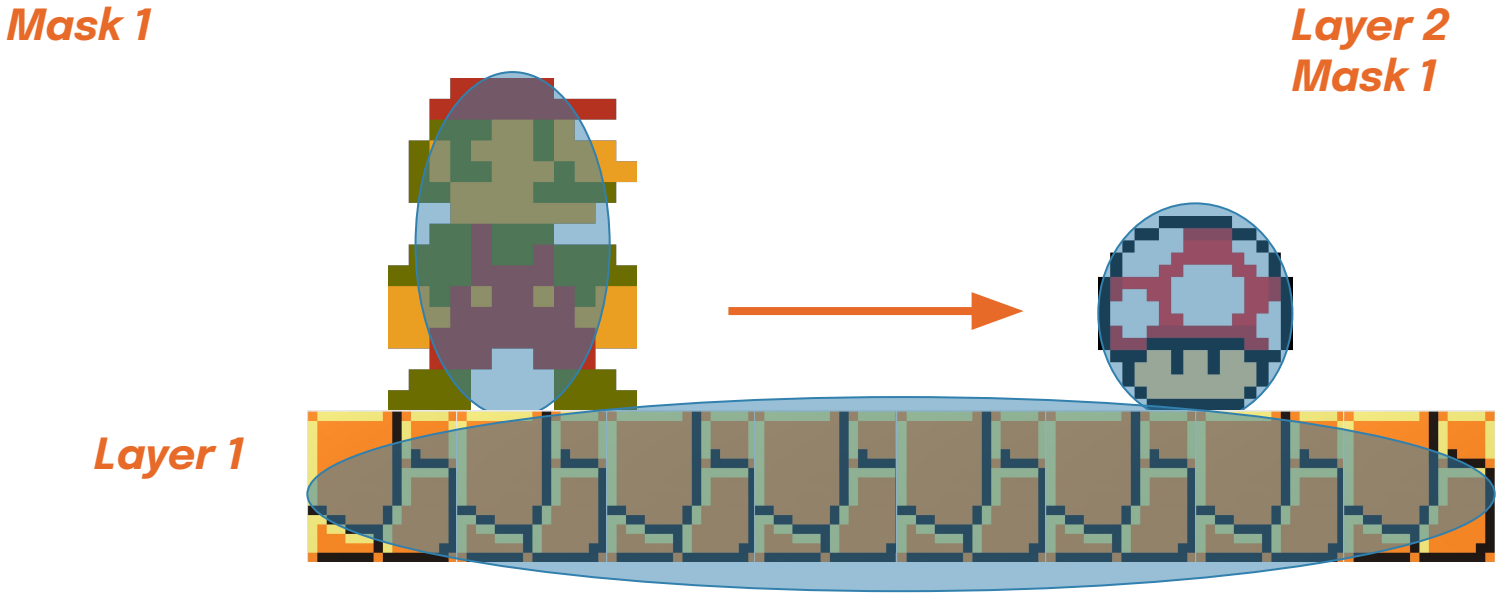
Cosa succede?



Godot: Collision Layer e Collision Mask

Esempio:

Cosa succede?



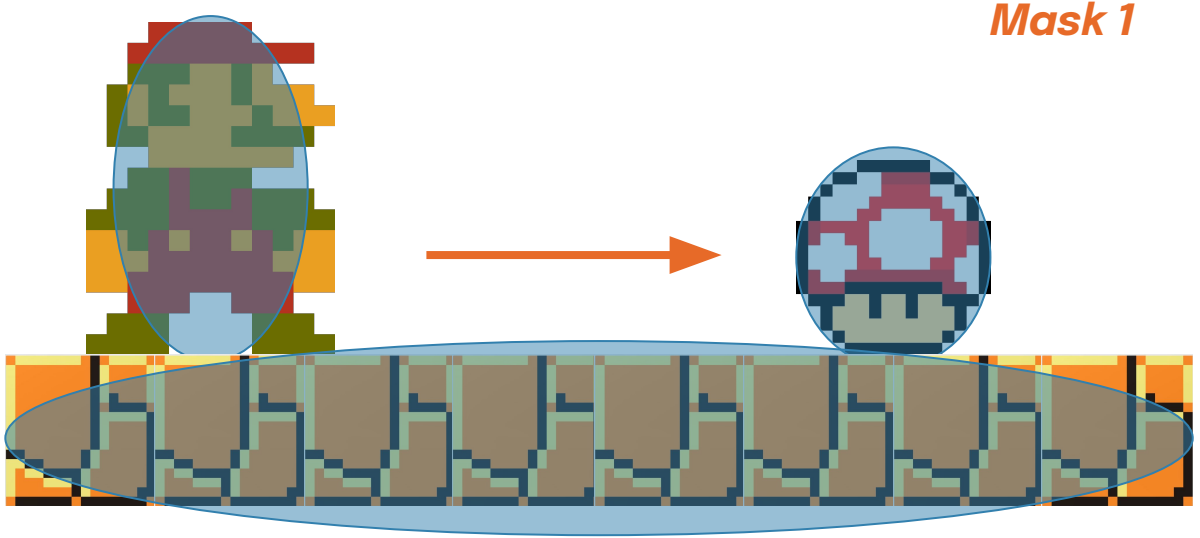
Godot: Collision Layer e Collision Mask

Esempio:

Cosa succede?

Mask 1,2

Layer 2
Mask 1



Layer 1

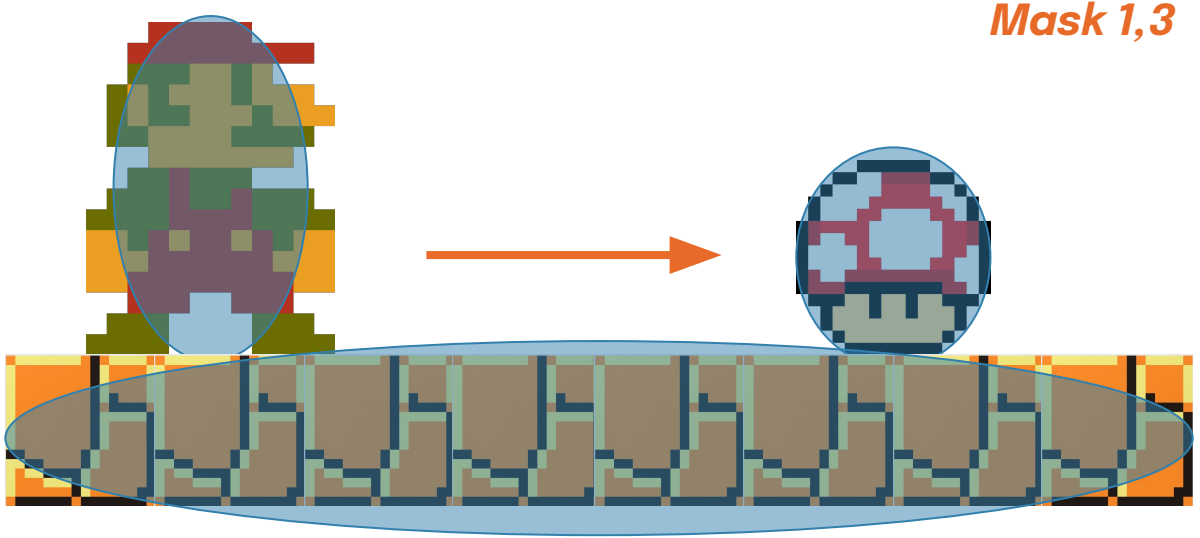
Godot: Collision Layer e Collision Mask

Esempio:

Cosa succede?

Layer 3
Mask 1,2

Layer 2
Mask 1,3



Layer 1

Godot: CharacterBody2D - Vector2D

Vector2D (Variant) è una struttura dati usata per mantenere coordinate 2D. Essenziale per qualsiasi caso in cui vogliamo interagire con l'ambiente 2D (e.g. posizioni)

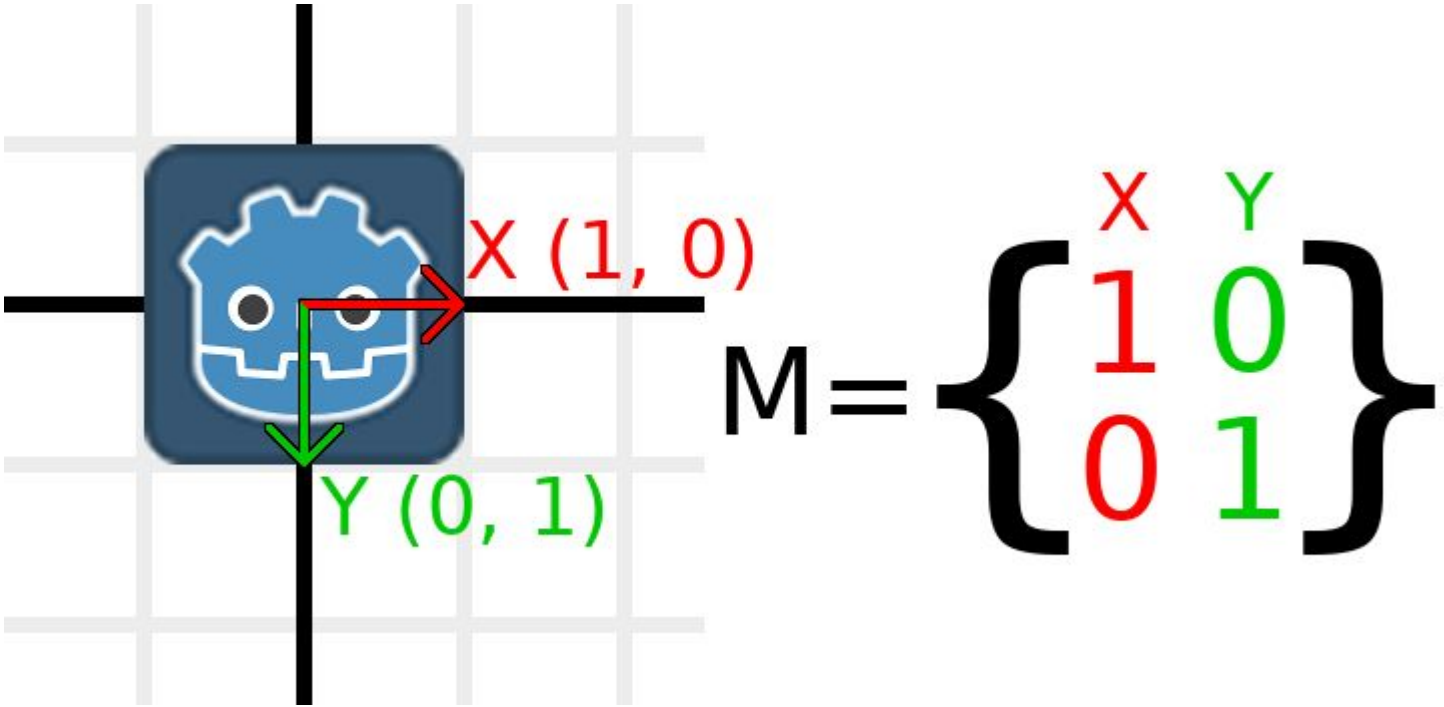
Fornisce diversi metodi utili all'interno del gioco (e.g. calcolare il vettore distanza tra due Vector2D)

Godot: Transform2D

Transform2D è una matrice 2x3 che rappresenta una trasformazione in uno spazio 2D

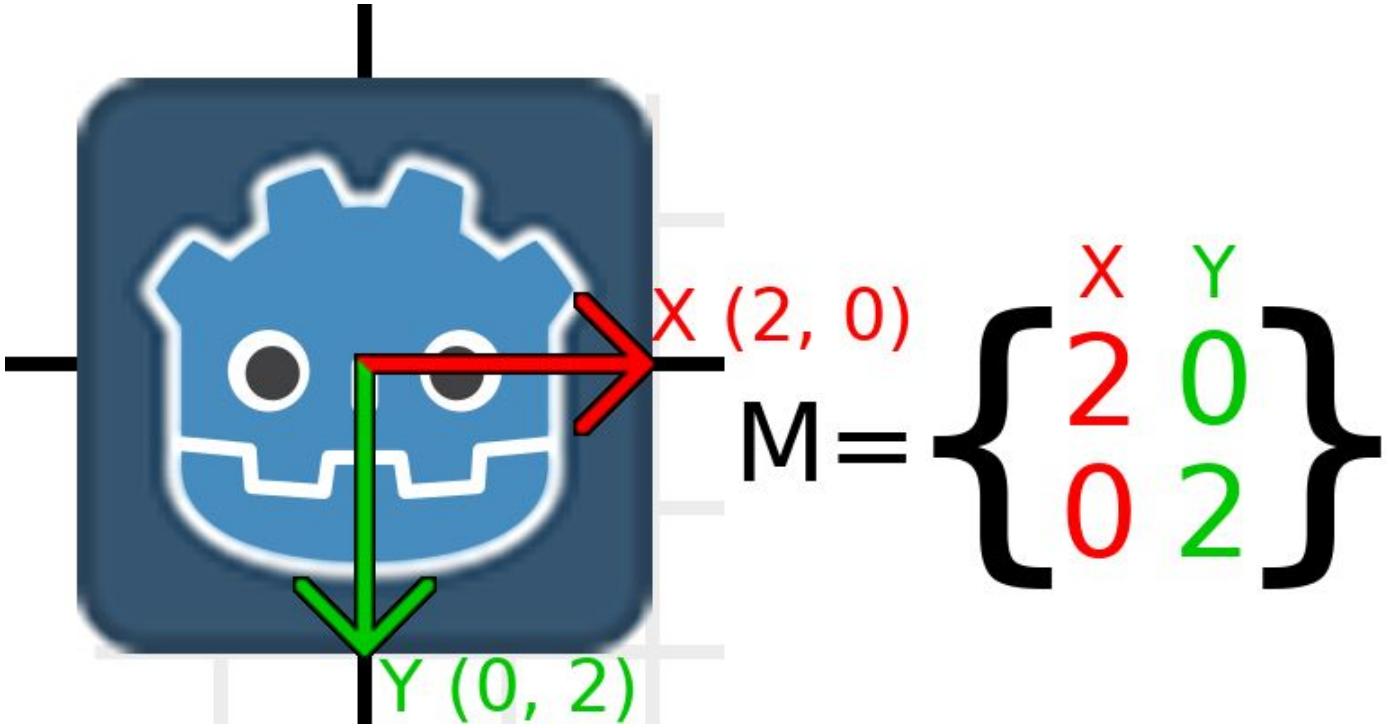
Viene utilizzata per applicare diverse trasformazioni al Node2D a cui è associata

Godot: Transform2D



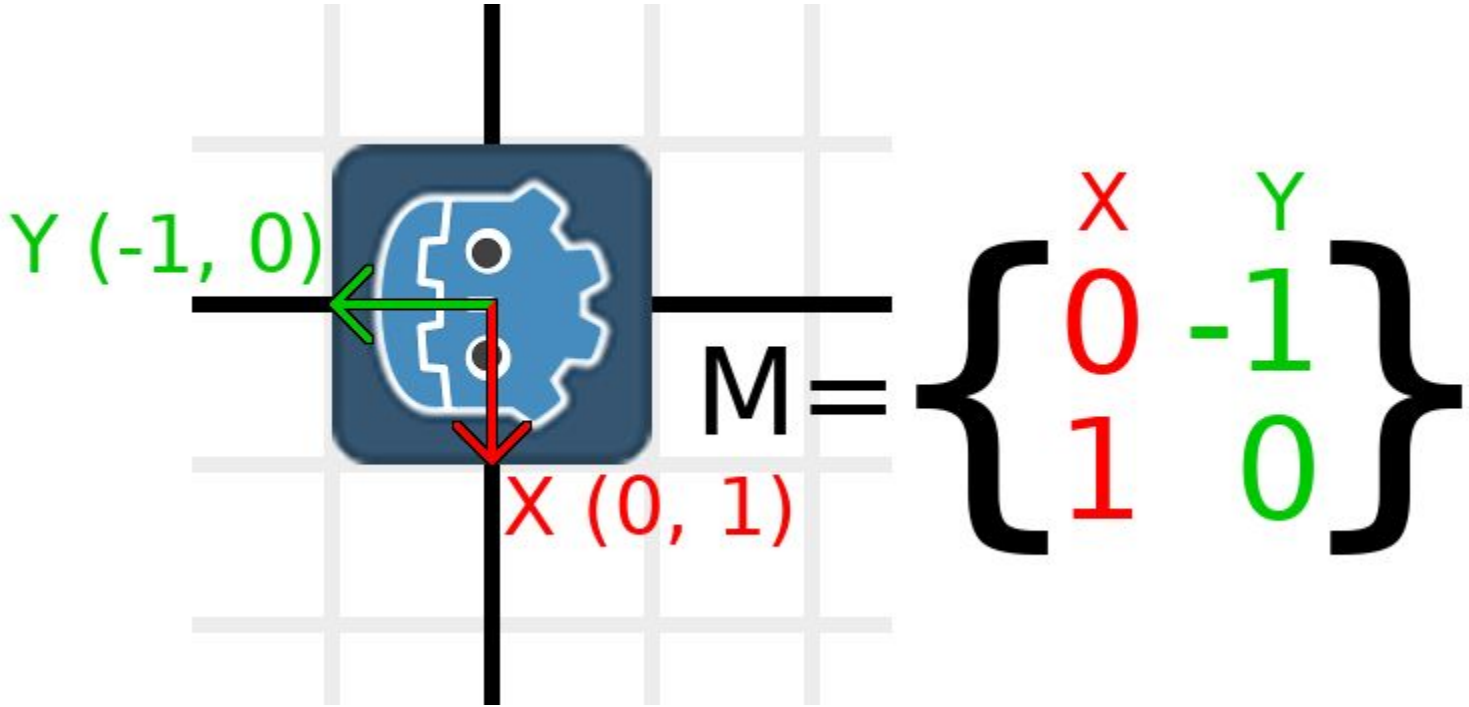
https://docs.godotengine.org/en/stable/tutorials/math/matrices_and_transforms.html

Godot: Transform2D



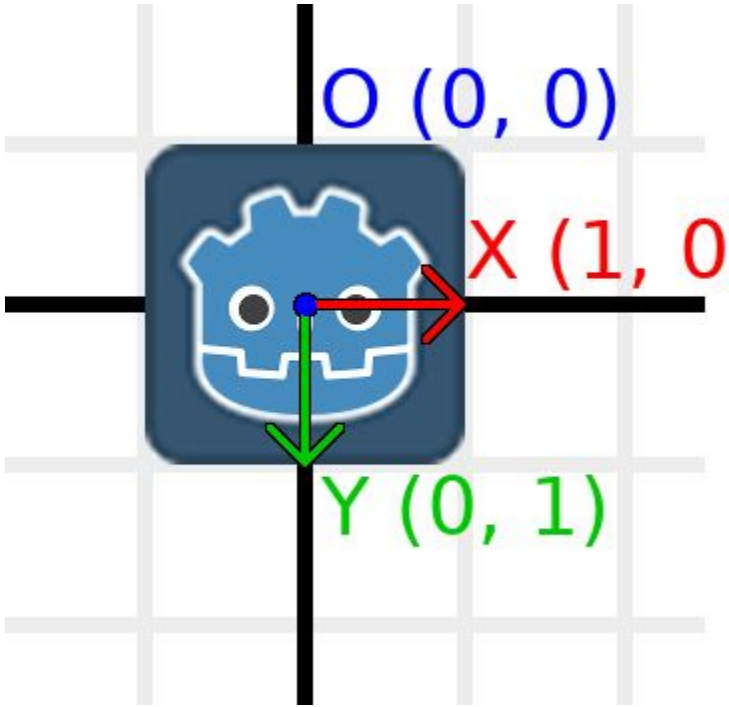
https://docs.godotengine.org/en/stable/tutorials/math/matrices_and_transforms.html

Godot: Transform2D



https://docs.godotengine.org/en/stable/tutorials/math/matrices_and_transforms.html

Godot: Transform2D

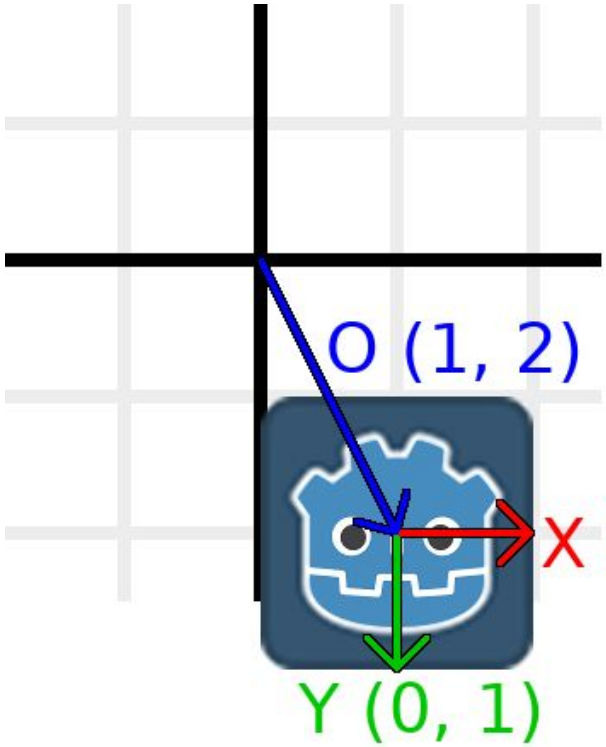


$$M = \begin{matrix} X & Y \\ 1 & 0 \\ 0 & 1 \end{matrix}$$

$$\text{Origin} = \begin{matrix} 0 \\ 0 \end{matrix}$$

https://docs.godotengine.org/en/stable/tutorials/math/matrices_and_transforms.html

Godot: Transform2D



$$M = \begin{matrix} X & Y \\ 1 & 0 \\ 0 & 1 \end{matrix}$$

$$\text{Origin} = \begin{matrix} 1 \\ 2 \end{matrix}$$

https://docs.godotengine.org/en/stable/tutorials/math/matrices_and_transforms.html

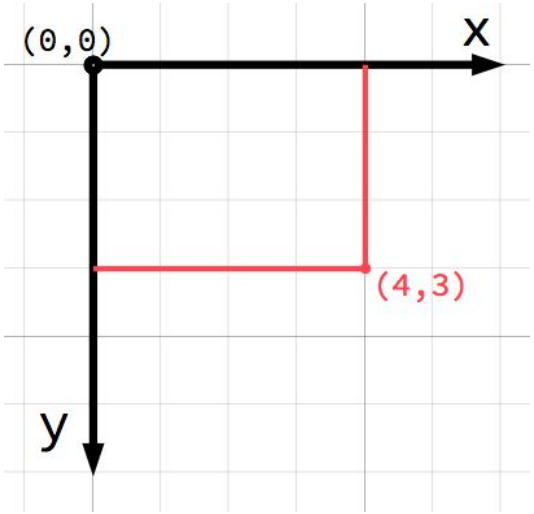
Godot: Viewport

Un **Viewport** è una superficie su cui vengono disegnati gli elementi di gioco (finestra principale dell'editor di Godot)

Inoltre, è possibile definire **SubViewport**, delle superfici aggiuntive in modo tale che vi siano più superfici sulle quali disegnare (e.g. utilizzare più camere nella stessa scena)

Godot: Viewport

NOTA BENE: l'asse y punta in basso e non in alto. Questa è una comune pratica in applicazioni di computer grafica (origine in alto a sinistra)



Godot: Script

Gli script non sono tecnicamente classi. Sono delle risorse che comunicano all'engine una serie di inizializzazioni da eseguire per le classi built-in

Le classi di Godot registrano i loro dati all'interno di **ClassDB**. Questo database fornisce accesso a informazioni relative alla classe a runtime: Proprietà, Metodi, Costanti, Segnali

Associare uno script all'oggetto estende le informazioni disponibili nel ClassDB

Godot: Script di Movimento Built-In

```gdscript
extends CharacterBody2D


const SPEED = 300.0
const JUMP_VELOCITY = -400.0


func _physics_process(delta: float) -> void:
    # Add the gravity.
    if not is_on_floor():
        velocity += get_gravity() * delta

    # Handle jump.
    if Input.is_action_just_pressed("ui_accept") and is_on_floor():
        velocity.y = JUMP_VELOCITY

    # Get the input direction and handle the movement/deceleration.
    # As good practice, you should replace UI actions with custom gameplay action
    var direction := Input.get_axis("ui_left", "ui_right")
    if direction:
        velocity.x = direction * SPEED
    else:
        velocity.x = move_toward(velocity.x, 0, SPEED)

    move_and_slide()
```

Godot: Script

Metodi essenziali:

- _ready: invocato quando il nodo e i suoi figli entrano nella scena
- _process: invocato durante lo step di processing del main loop di gioco (delta NON costante)
- _physics_process: invocato durante lo step di processing della fisica del main loop di gioco (delta costante)

Godot: Script

Metodi essenziali:

- ➔ add_child: aggiunge un nodo figlio al nodo che invoca questo metodo
  - N.B.: per aggiungere nodi figli al Nodo radice di una scena è possibile chiamare la radice dell'albero con get_tree().root
- ➔ queue_free: mette in coda il nodo per essere rimosso alla fine del frame attuale (anche i nodi figli di questo nodo sono rimossi)

Godot: Script

Annotazioni:

- @export: permette di rendere le proprietà modificabili dall'editor (nota: export supporta diverse funzionalità come grouping, range, ...)
- @onready: inizializza una variabile prima della chiamata del metodo _ready

https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_exports.html

Godot: Script

Da script è possibile accedere ai nodi figli del root node a cui è associato lo script

```
func _animation_setter() -> void:
    $AnimatedSprite2D.play("idle")
```

Per farlo è sufficiente preporre il carattere $ (abbreviazione del metodo get_node("NodePath")) al nome del nodo

Questa funzionalità è essenziale per chiamare da script metodi associati a nodi figli del nodo radice

Godot: Script Cose Simpatiche

Array e Dizionari:

```
var d = {4: 5, "A key": "A value", 28: [1, 2, 3]}
d["Hi!"] = 0
d = {
    22: "value",
    "some_key": 2,
    "other_key": [2, 3, 4],
    "more_key": "Hello"
}
```

```
var arr = []
arr = [1, 2, 3]
var b = arr[1] # This is 2.
var c = arr[arr.size() - 1] # This is 3.
var d = arr[-1] # Same as the previous line, but shorter.
arr[0] = "Hi!" # Replacing value 1 with "Hi!".
arr.append(4) # Array is now ["Hi!", 2, 3, 4].
```

Godot: Script Cose Simpatiche

Godot fornisce la possibilità di salvare delle funzioni in delle variabili e chiamarle successivamente usando il metodo call

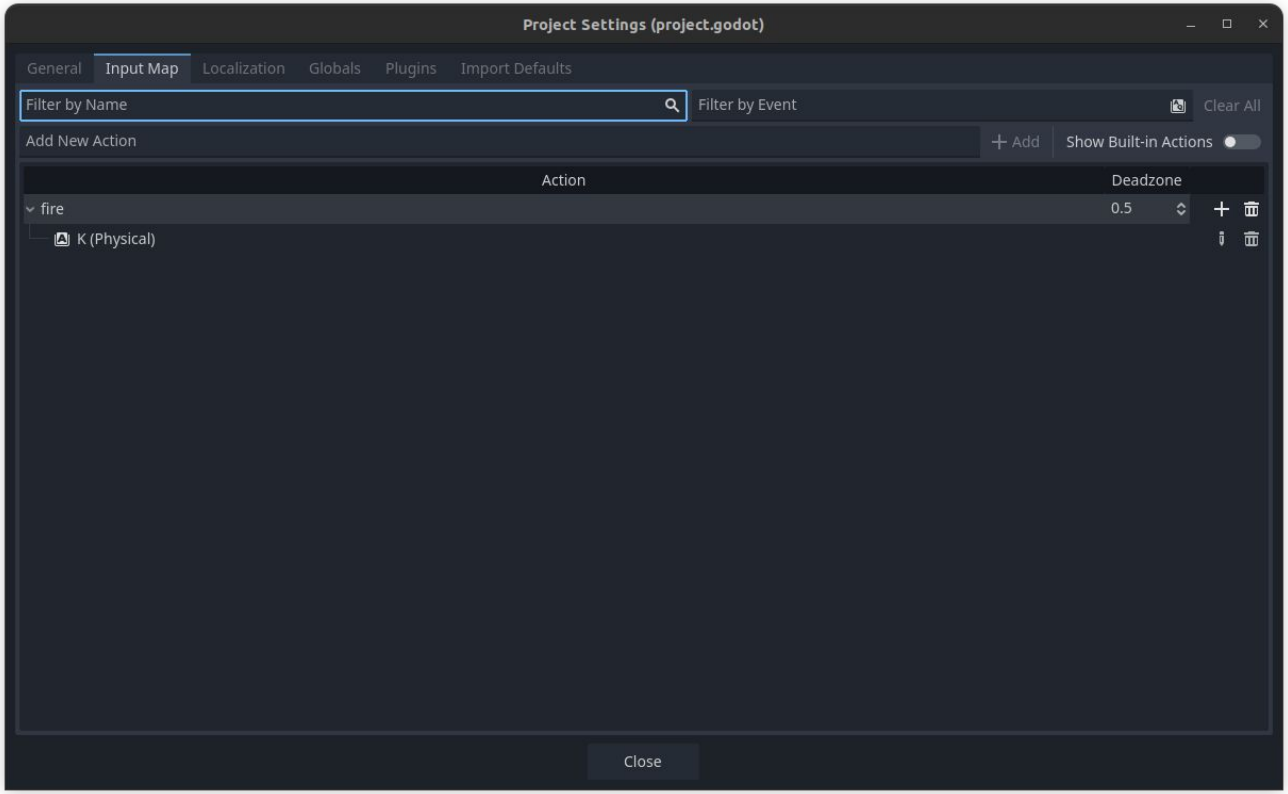
Viene data la possibilità di salvare sia metodi di nodi sia lambda function definite sul momento

Godot: Script Cose Simpatiche

Per castare un oggetto da una classe padre ad una classe figlia

```
func _on_area_2d_body_entered(body: Node2D) -> void:
    if body is Character:
        (body as Character).bounce(self.global_position)
```

Godot: Input


Project Settings (project.godot)
General
Input Map
Localization
Globals
Plugins
Import Defaults
Filter by Name
Filter by Event
Clear All
Add New Action
Add
Show Built-in Actions
Action
Deadzone
fire
0.5
K (Physical)
Close

Godot: Gestione Input da Script

Godot fornisce la classe **Input** per gestire input da tastiera, gamepad e così via

I metodi più importanti a riguardo sono:

- → **is_action_just_pressed**: True se il giocatore ha iniziato a premere l'azione nel frame attuale
- → **is_action_pressed**: True se l'azione sta venendo premuta

Godot: Script Estendere Classi

Viene data la possibilità di definire delle nuovi classi a partire da quelle fornite da Godot

```
class_name Character extends CharacterBody2D


const SPEED = 300.0
const JUMP_VELOCITY = -400.0
```

Questo permette di generalizzare alcuni controlli (vedremo dopo)

Godot: Caricare e Istanziare una scena

In alternativa, da codice è possibile istanziare una scena nel seguente modo:

- Usare il metodo *load* o *preload* per caricare una scena da file
- Questo crea un oggetto PackedScene
- Chiamare il metodo *instantiate* di PackedScene

Godot: Caricare e Istanziare una scena

Esempio:

```
# Use load() instead of preload() if the path isn't known at compile-time.
var scene = preload("res://scene.tscn").instantiate()
# Add the node as a child of the node the script is attached to.
add_child(scene)
```

Godot: Istanziare un nodo da codice

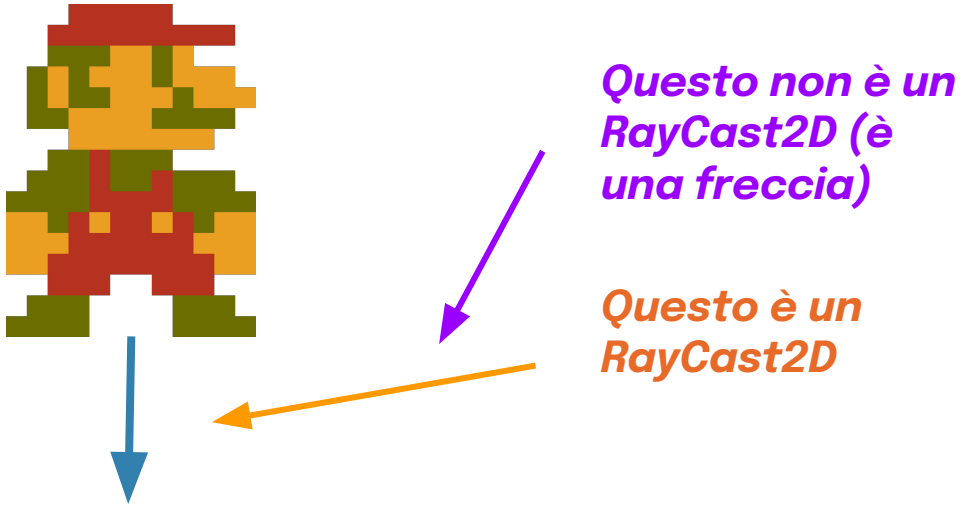
Esempio:

```
var sprite2d

func _ready():
    var sprite2d = Sprite2D.new() # Create a new Sprite2D.
    add_child(sprite2d) # Add it as a child of this node.
```

Godot: RayCast2D

RayCast2D è un nodo che permette di creare un raggio in uno spazio 2D. Il raggio causerà poi una collisione con il primo CollisionObject che raggiunge



Godot: RayCast2D

Il nodo RayCast2D ha solo una collision mask, non si trova su nessun collision layer

Questo perchè la sua unica funzione è quella di trovare delle collisioni con altri oggetti su layer specificati

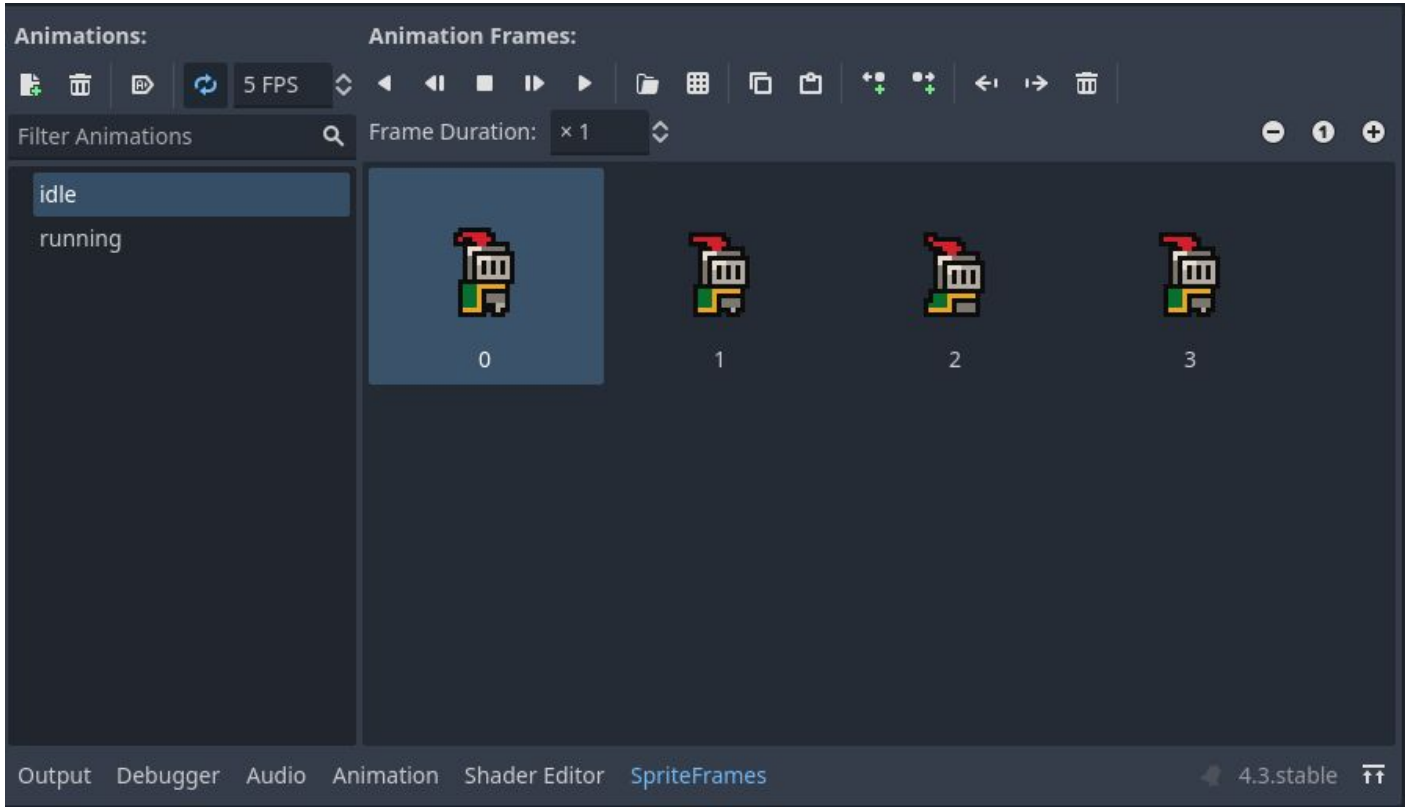
Estremamente utile in qualsiasi situazione vogliamo che una entità cerchi per delle collisioni seguendo la direzione specificata dal raycast

Godot: Animazioni

Il nodo per gestire nel modo più diretto le animazioni è **AnimatedSprite2D**

Questo nodo permette di creare diverse animazioni (identificate da un nome), ognuna con diverse immagini per ogni frame

Godot: Animazioni



Godot: Animazioni

Inoltre, AnimatedSprite2D ha due proprietà ***flip_v*** e ***flip_h*** che permettono di capovolgere lo sprite verticalmente o orizzontalmente

N.B.: utile per disegnare lo sprite nella direzione opposta verticale, in modo da non dover avere frame dedicati

Godot: Animazioni

Una volta create le animazioni è poi possibile eseguirle da script, richiamando il nodo e usando il metodo play (passando come argument il nome dell'animazione da eseguire)

```
func _animation_setter() -> void:
    $AnimatedSprite2D.play(&"idle")

    if movable:
        if velocity.x > 0:
            $AnimatedSprite2D.flip_h = false
        elif velocity.x < 0:
            $AnimatedSprite2D.flip_h = true
```

Godot: Animazioni

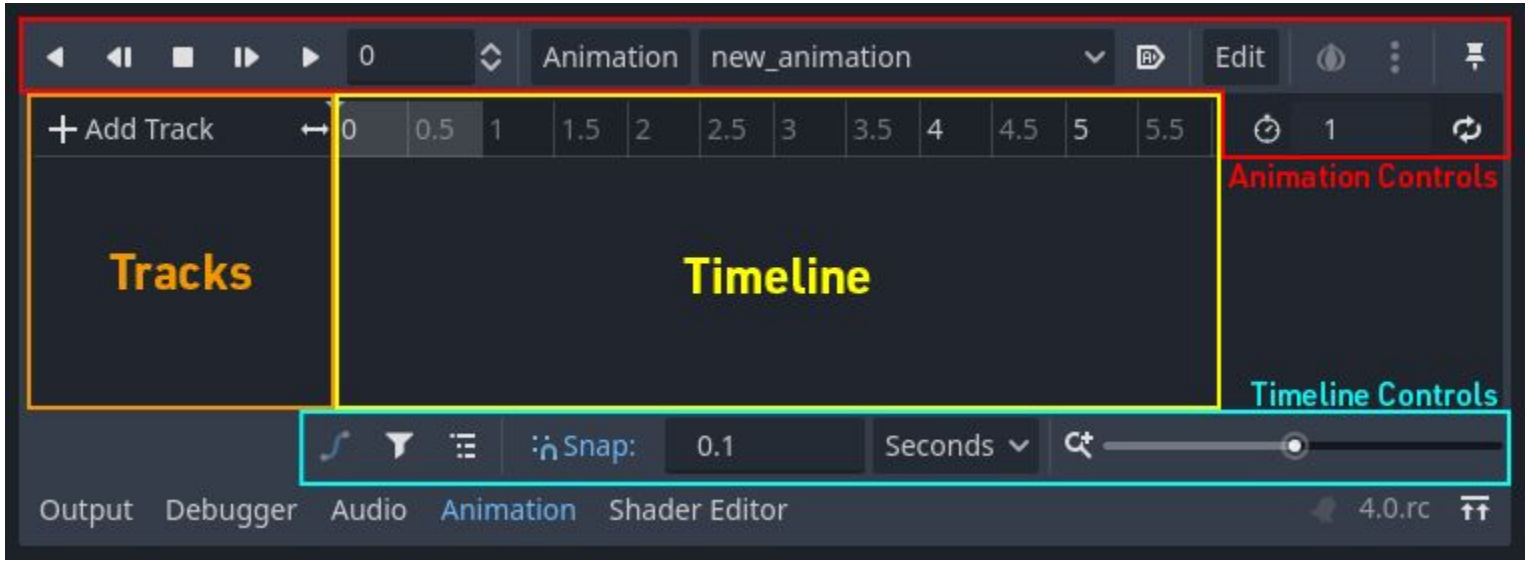
Problema: Come gestire animazioni complesse che richiedono la modifica di proprietà di altri nodi all'interno della scena? (e.g. posizione)

Godot: Animazioni

Soluzione: **AnimationPlayer**

Godot: Animazioni

Finestra Animation dell'animation player



Godot: Animazioni

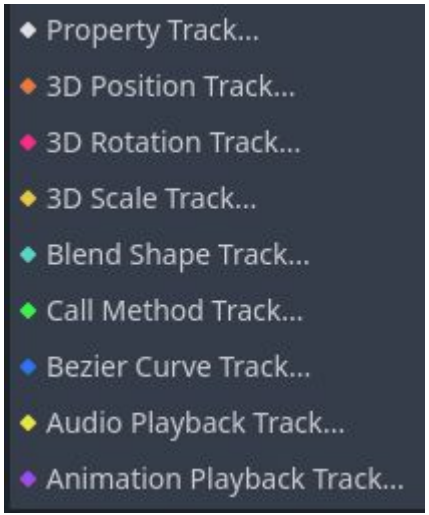
AnimationPlayer permette di impostare dei valori che determinate proprietà devono avere in certi punti temporali dell'animazione (keyframe)

Il valore di questa proprietà tra due keyframe è uguale al valore impostato nel keyframe iniziale

Godot: Animazioni

L'AnimationPlayer permette di creare diversi tipi di track per modificare diverse proprietà Ogni track ha diverse funzionalità:

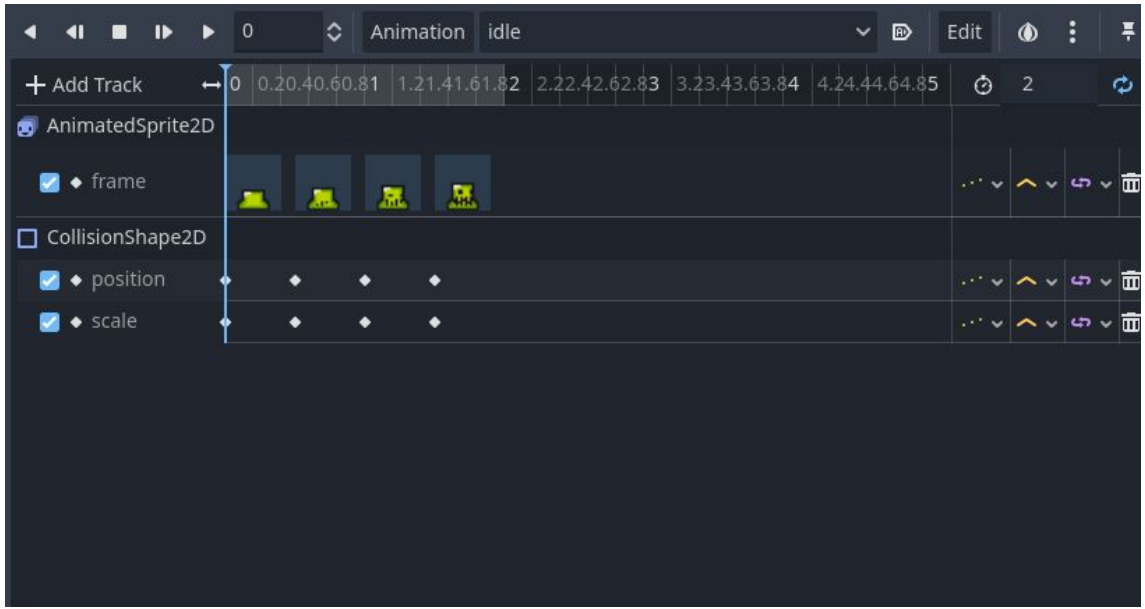
- **Property**: modificare il valore di una proprietà di un nodo
- **Call Method**: chiama un metodo all'interno di uno script
- ...



https://docs.godotengine.org/en/stable/tutorials/animation/animation_track_types.html

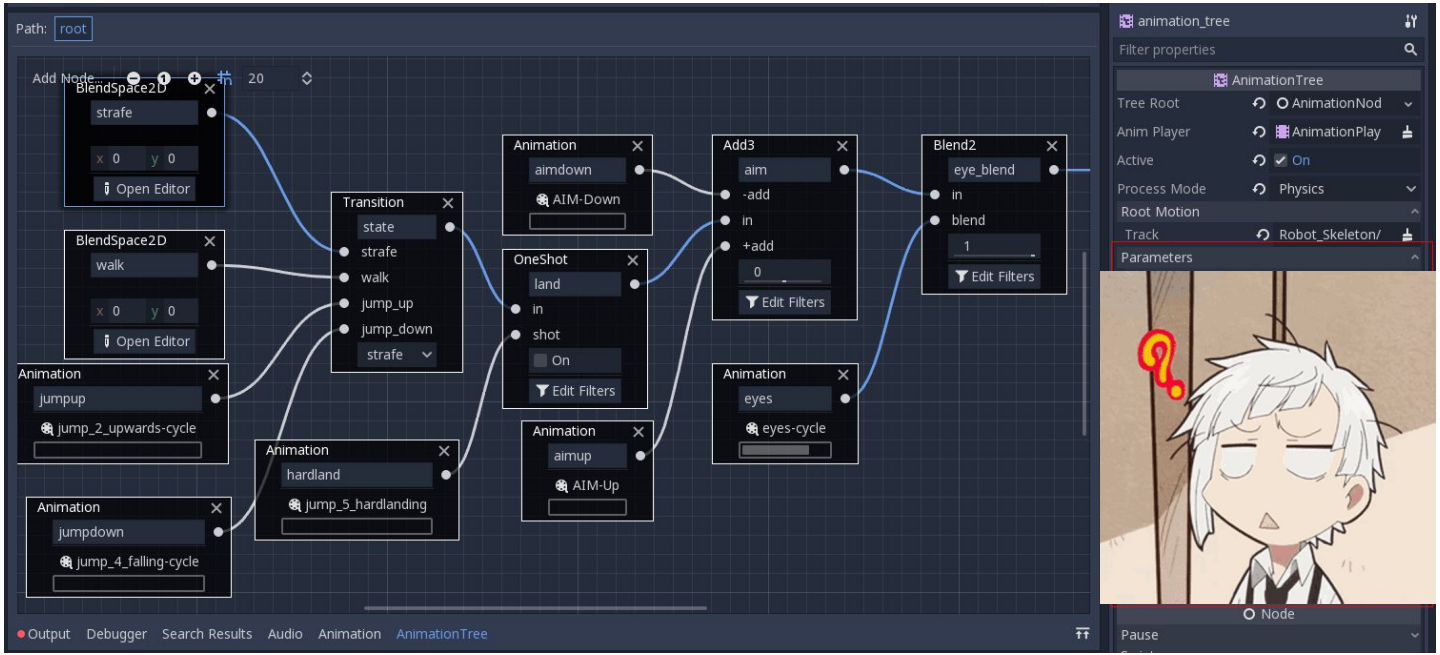
Godot: Animazioni - Property Track Esempio

Utilizzeremo la classe Enemy che abbiamo creato prima. Aggiungiamo un nodo AnimationPlayer alla scena



Godot: Animazioni

Soluzione ancora più potente: AnimationTree



https://docs.godotengine.org/en/latest/tutorials/animation/animation_tree.html

Godot: Creare una scena per un nemico

I nodi fondamentali per un scena dedicata a collezionabili:

- → **CharacterBody2D (root)** (abbiamo visto prima)
- → **CollisionShape2D**
- → **AnimatedSprite2D**

Godot: Creare una scena per collezionabili

I nodi fondamentali per un scena dedicata a collezionabili:

- → **Area2D (root)**
- → **CollisionShape2D**
- → **AnimatedSprite2D**

Godot: Creare una scena per un proiettile

I nodi fondamentali per un scena dedicata a collezionabili:

- → **CharacterBody2D (root)** (forse? Dipende dal gioco)
- → **CollisionShape2D**
- → **AnimatedSprite2D**

Godot: Scrivere il codice per sparare

Modificare lo script associato al giocatore in modo tale che sia possibile sparare dei proiettili di fuoco

Lo script dovrebbe fare quanto segue:

- Caricare la scena del proiettile creata precedentemente
- Istanziarla (**ATTENZIONE!**)
- Impostare la velocità (orizzontale e verticale) rispetto alla direzione del giocatore

Godot: Scrivere il codice per il nemico

Modificare lo script associato al nemico in modo tale che attacchi il giocatore quando si trova in un certo range (e.g., saltando addosso al giocatore)

Lo script dovrebbe fare quanto segue:

- Ottenere la posizione del giocatore
- Impostare la velocità (orizzontale e verticale) rispetto alla direzione del giocatore

Godot: Segnali

Esempio:

Collision Detected! Something happens

Come codificare la collision response?

Godot: Segnali

In Java (o anche in Godot stesso) per gestire questa situazione servirebbe una classe di controllo (e.g. GameController) che controlli per ogni physics callback se avvengono delle collisioni

Dovrebbe poi gestire le collisioni in base alle classi dei due oggetti che stanno collidendo per determinare quale comportamento dovrebbe avvenire

Godot: Segnali

Problema: Questo tipo di gestione è poco pratica e porterebbe a diverse situazioni di controlli hard-coded

Soluzione: Segnali!

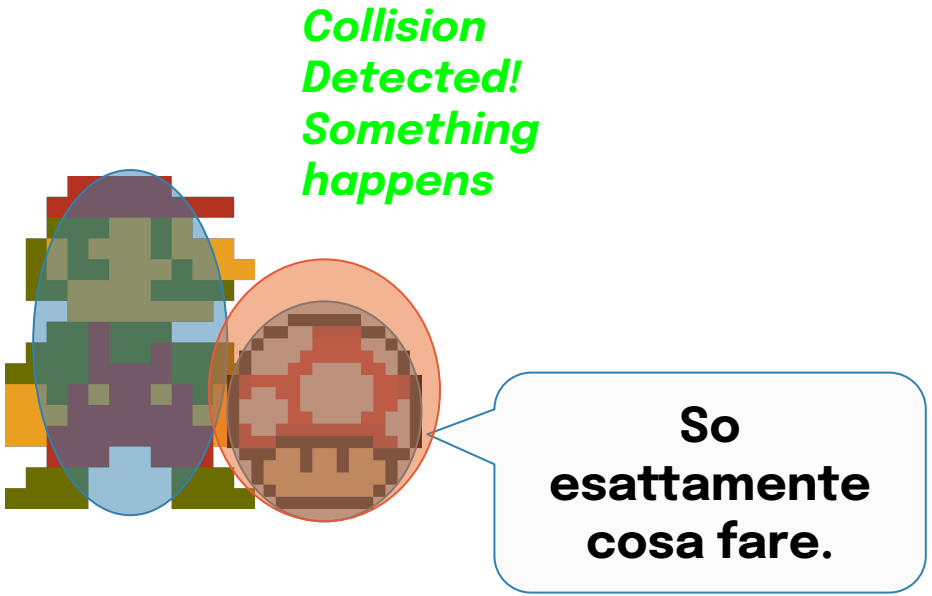
Godot: Segnali

I **Segnali** in Godot sono dei messaggi che vengono emessi dai nodi quando avviene un evento specifico

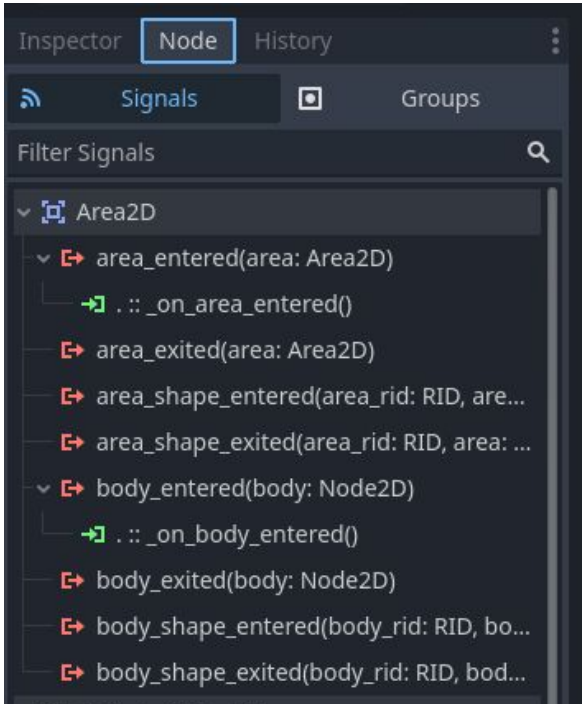
Sono utili per diminuire l'accoppiamento (coupling) dei moduli di un gioco e a rendere il funzionamento delle varie componenti più flessibile

Godot: Segnali

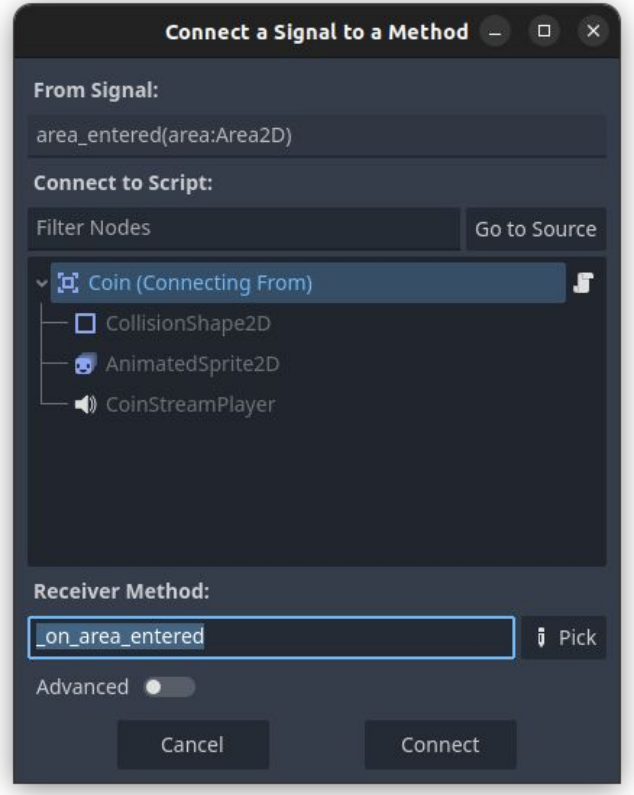
Esempio:



Godot: Segnali



Godot: Segnali



Godot: Segnali

```
func _on_body_entered(body: Node2D) -> void:
    if is_instance_of(body, Player) and available:
        var player := body as Player
        set_deferred("$CollisionShape2D.disabled", true)
        self.visible = false
        available = false
        player.increase_score(1)
        $CoinStreamPlayer.play()
```

Controllare che il Body sia una istanza di un nodo definito da script

Godot: Segnali

Inoltre, è possibile collegare un segnale direttamente da codice

```
func _ready():
    var timer = get_node("Timer")
    timer.timeout.connect(_on_timer_timeout)
```

```
func _on_timer_timeout():
    visible = not visible
```

Godot: Segnali

Infine, è possibile definire dei segnali personalizzati da codice

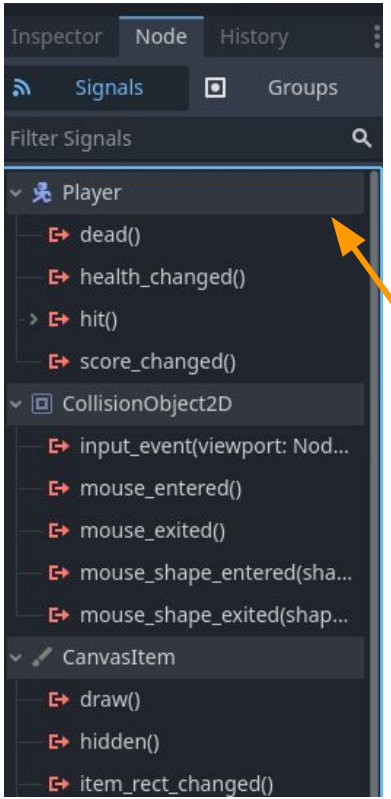
```
extends Node2D

signal health_depleted

var health = 10
```

```
func take_damage(amount):
    health -= amount
    if health <= 0:
        health_depleted.emit()
```

Godot: Segnali



I nuovi segnali definiti da script compaiono anche nell'inspector

Godot: Suoni - AudioStream

Un nodo **AudioStream** è la classe base per riprodurre effetti sonori e musica

Vi sono tre tipi di AudioStream:

- ➔ **AudioStreamPlayer**
- ➔ **AudioStreamPlayer2D**
- ➔ **AudioStreamPlayer3D**

Godot: Timer

Il nodo **Timer** è un nodo utile per impostare degli intervalli di tempo, per modificare qualcosa dopo X secondi

Una volta che il Timer termina, è poi possibile informare un nodo attraverso un apposito segnale

Godot: Timer


```
func _ready() -> void:
    _animation_setter()
    $Timer.start(2)
```

(Inoltre, è possibile metterlo in pausa)
N.B.: di default i timer si riavviano automaticamente

Godot: Timer

N.B.: è anche possibile creare un one-shot Timer chiamato **SceneTreeTimer** da codice

Utile per creare dei delay

```
func some_function():
    print("Timer started.")
    await get_tree().create_timer(1.0).timeout
    print("Timer ended.")
```

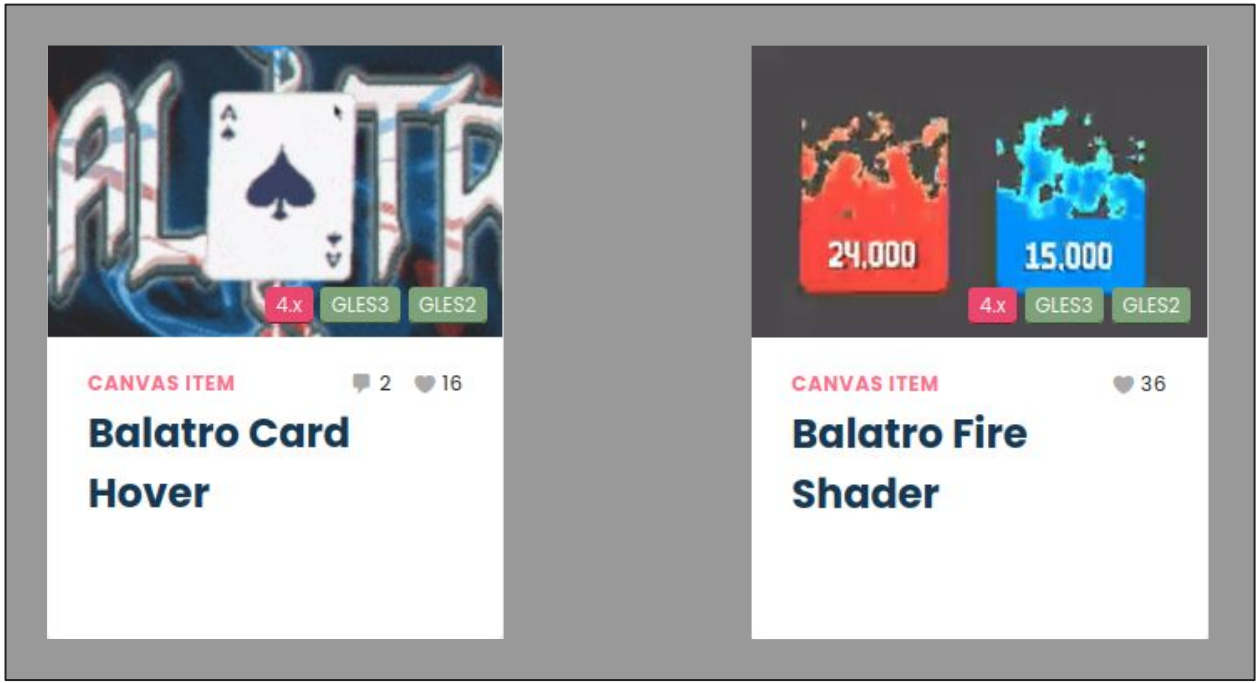
Godot: Shader

Uno Shader è un programma speciale che viene eseguito su GPU

Il codice per uno Shader viene eseguito su ogni pixel separatamente, grazie alla parallelizzazione offerta dall'esecuzione di queste operazioni su GPU

Godot supporta un linguaggio di programmazione dedicato agli Shader e basato su OpenGL

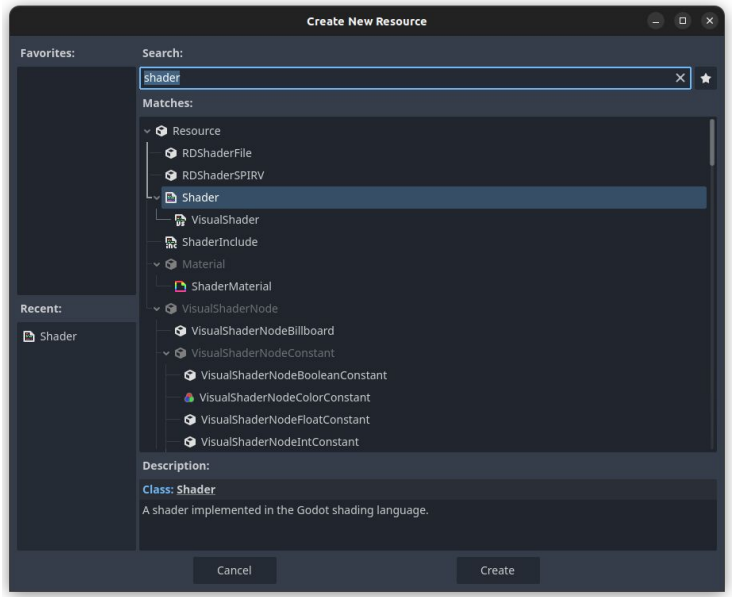
Godot: Shader



https://godotshaders.com/

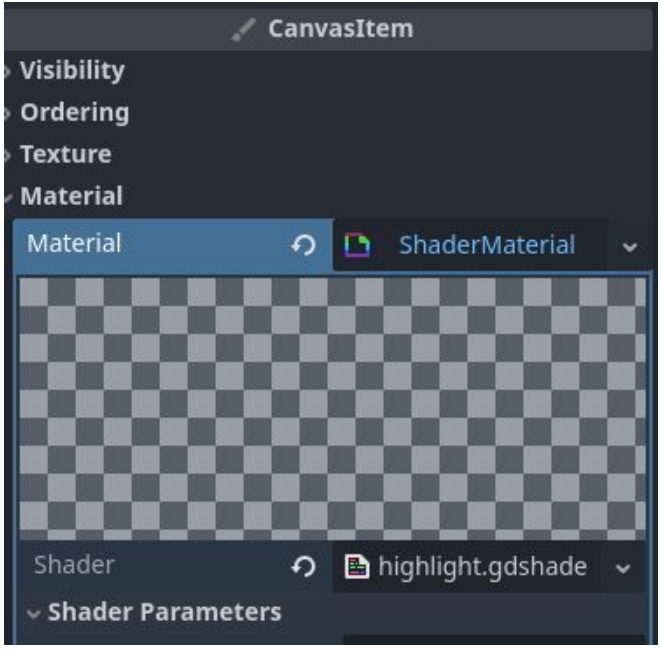
Godot: Come usare uno shader in Godot

Assumendo di avere il codice per uno shader, creare una risorsa Shader



Godot: Come usare uno shader in Godot

A questo punto creare un material per l'oggetto su cui si vuole applicare lo shader



Godot: Come usare uno shader in Godot

Inoltre, è possibile modificare da script gli attributi presenti all'interno di uno shader

```
$AnimatedSprite2D.get_material().set_shader_parameter("active", true)
```

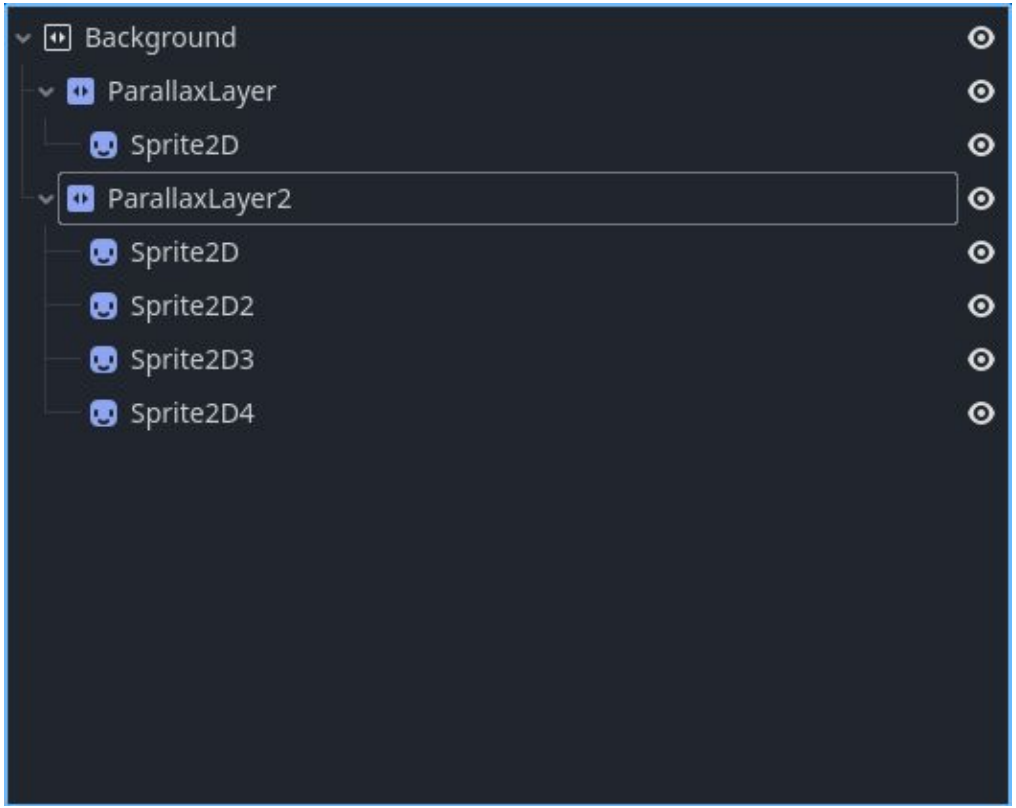
Può essere utile durante il gioco per ottenere effetti particolari o per abilitare/disabilitare lo shader

Godot: Parallax Background

ParallaxBackground è un nodo che permette di creare un background con effetto parallasse (ovvero, le immagini di sfondo si muovono più lentamente rispetto a quelle in primo piano)

Vengono poi usati diversi nodi **ParallaxLayer** per creare diversi livelli di background con diverse velocità (modificando la property motion_scale)

Godot: Parallax Background



Godot: GUI

Nel mondo ideale tutti gli schermi avrebbero esattamente la stessa risoluzione

Sfortunatamente non viviamo nel mondo ideale :(

Dobbiamo quindi soffrire per realizzare delle interfaccia che mantengano le posizioni dei singoli elementi consistenti indipendentemente dalle impostazioni grafiche

Godot: CanvasLayer

Prima di iniziare, qualche dettaglio aggiuntivo su come Godot disegna sul viewport gli elementi di gioco 2D

CanvasLayer è un nodo utilizzato per renderizzare tutti gli elementi di gioco 2D

Node2D e Control ereditano entrambi da CanvasLayer

Come gestire diversi layer su cui vogliamo che diversi transform siano applicati?

Godot: GUI - Anchor

Il posizionamento dei nodi della GUI si basa sulle Ancore

Ogni nodo di Controllo ha 4 ancore che servono a definire dei margini rispetto ai quali verranno poi disegnati gli elementi della GUI

Fortunatamente, Godot fornisce diversi pre-set di ancore

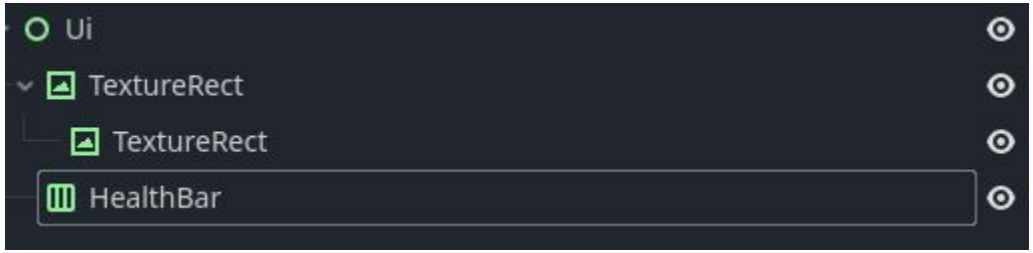
Godot: GUI

Creiamo un Control node

Un Control node ha: una forma rettangolare che definisce i suoi estremi, una posizione ancora relativa al Control node padre o alla viewport corrente, e dei margini relativi all'ancora

Godot fornisce poi diversi tipi di Container

Godot: GUI



Esempio banale ma efficace

Ciò che viene mostrato nella GUI (esempio, numero di monete collezionate) può poi essere aggiornato usando i segnali

Godot: GUI Nodi Comunemente Usati

I nodi più usati per realizzare interfacce grafiche sono i seguenti:

- **Label**: per mostrare testo
- **Button**: bottone interagibile che può contenere del testo e una icona
- **ColorRect**: mostra un rettangolo di un colore definito
- **TextureRect**: mostra una texture

Godot: GUI - Container

Tuttavia, più diventano complesse le interfacce più diventa complesso gestire le ancore per ogni singolo nodo relativo all'interfaccia grafica

Per facilitare la realizzazione di interfacce con molte componenti, Godot mette a disposizione dei nodi di Controllo chiamati **Container**

Un Container dispone automaticamente i suoi nodi figli seguento una certa formattazione

Godot: GUI Container Comunemente Usati

I Container più usati per realizzare interfacce grafiche sono i seguenti:

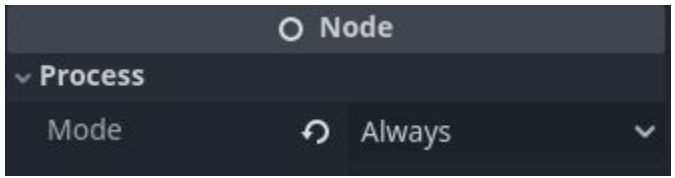
- **CenterContainer**: mantiene nodi figli orientati al centro
- **MarginContainer**: mantiene dei margini attorno ai nodi figli
- **VBoxContainer**: orienta i nodi figli verticalmente
- **HBoxContainer**: orienta i nodi figli orrizontalmente

Godot: Menu

- Per mettere in pausa la scena (e.g. menù di opzioni o death screen):

```
get_tree().paused = not get_tree().paused
```

- Modificare poi la process mode del menu in modo tale che sia sempre interagibile



Godot: Generiche cose utili - Motion Mode

Una delle proprietà di CharacterBody2D è **Motion Mode**

Questa proprietà è utile quando si vuole modificare il funzionamento del Physics Engine per lavorare con giochi 2D top-down invece di platform

Godot: Generiche cose utili - Cambiare Scena

```
extends Node

func _on_texture_button_pressed() -> void:
    get_tree().change_scene_to_file("res://scenes/Level.tscn")


func _on_texture_button_2_pressed() -> void:
    get_tree().quit()
```

Godot: Segnali Esercizio

Modificare il progetto in modo tale che:

- Il giocatore possa raccogliere i collezionabili
- I proiettili colpiscano ed eliminino i nemici
- (**BONUS**): I nemici respingano il giocatore quando questo collide con loro

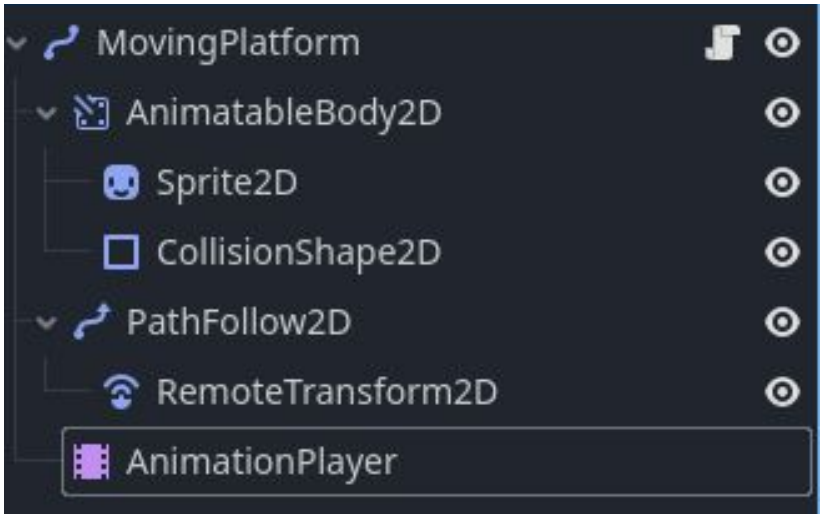
Suggerimento: Usare una Area2D per controllare la HitBox dei nemici, invece della CollisionShape2D del CharacterBody

Godot: Menu e GUI Esercizio

Modificare il progetto in modo tale che:

- Il giocatore abbia una barra della vita e un contatore per le monete raccolte
- Vi sia un menù di pausa

Godot: Creare una scena per una piattaforma in movimento (bonus)



Godot: GitHub del Progetto

https://github.com/swapUnibaAcademy/SviluppoVG-Godot

Grazie a tutti per l'attenzione! C:

Notate che molte delle informazioni che sono presenti in questa presentazione sono state estratte dalla documentazione ufficiale di Godot

https://docs.godotengine.org/en/stable/index.html

Per capire ulteriori funzionalità (e.g. networking, occlusion e navigation), vi consiglio caldamente di utilizzarla

UniBa

SWAP researchgroup