Guida di approfondimento agli script Unity per Platform 2D

Obiettivo della guida

Questa guida approfondisce gli script usati nel progetto platform 2D. L'obiettivo è capire non solo "cosa copiare", ma soprattutto cosa fa ogni riga, perché viene scritta in quel modo e come modificare gli script in autonomia.

Gli script analizzati sono:

- `HelloUnity`;
- `PlayerController`;
- `Coin`;
- `GameManager`;
- `CameraFollow`;
- gestione base delle animazioni tramite `Animator`.

---

1. Che cos'è uno script in Unity

Uno script in Unity è un file C# che permette di aggiungere comportamento a un GameObject.

Un GameObject può rappresentare:

- il giocatore;
- una moneta;
- una piattaforma;
- una camera;
- un nemico;
- un gestore del gioco.

Lo script definisce cosa quell'oggetto deve fare.

Esempi:

- il Player si muove;
- la Coin scompare quando viene raccolta;
- la Camera segue il Player;
- il GameManager aggiorna il punteggio.

---

2. Struttura base di uno script Unity

Esempio minimo:

```CSharp
using UnityEngine;

public class HelloUnity : MonoBehaviour
{
    void Start()
    {
        Debug.Log("Ciao Unity!");
    }
}
```

Analizziamolo riga per riga.

2.1 `using UnityEngine;`

```CSharp
using UnityEngine;
```

Questa riga importa il namespace principale di Unity.

Un namespace è un contenitore di classi. `UnityEngine` contiene moltissime classi utili, per esempio:

- `MonoBehaviour`;
- `GameObject`;
- `Transform`;
- `Rigidbody2D`;
- `Collider2D`;
- `Vector2`;
- `Vector3`;
- `Debug`;
- `Input`;
- `Time`;

- `Animator`.

Senza `using UnityEngine;`, molte istruzioni tipiche di Unity non sarebbero riconosciute.

2.2 Dichiarazione della classe

```CSharp
public class HelloUnity : MonoBehaviour
```

Questa riga dichiara una classe chiamata `HelloUnity`.

`public`

Significa che la classe è accessibile anche da Unity.

`class`

Indica che stiamo dichiarando una classe C#.

Una classe è una struttura che contiene dati e comportamenti.

`HelloUnity`

È il nome della classe.

Importante: in Unity il nome della classe deve coincidere con il nome del file.

Quindi:

```None
HelloUnity.cs
```

contiene:

```CSharp
public class HelloUnity : MonoBehaviour
```

Se il nome non coincide, Unity può generare errori.

`: MonoBehaviour`

Significa che la classe eredita da `MonoBehaviour`.

`MonoBehaviour` è la classe base degli script Unity che possono essere assegnati ai GameObject.

Grazie a `MonoBehaviour`, Unity può chiamare automaticamente metodi come:

- `Awake()`;
- `Start()`;
- `Update()`;
- `FixedUpdate()`;
- `OnCollisionEnter2D()`;
- `OnTriggerEnter2D()`;
- `LateUpdate()`.

2.3 Le parentesi graffe

```CSharp
{

}
```

Le parentesi graffe delimitano un blocco di codice.

In uno script Unity ci sono di solito:

- graffe della classe;
- graffe dei metodi;
- graffe degli `if`;
- graffe dei cicli.

Esempio:

```CSharp
public class Example : MonoBehaviour
{
    void Start()
    {
        if (true)
        {
            Debug.Log("Test");
        }
    }
```

```
}
```

---

3. Variabili

3.1 Che cos’è una variabile

Una variabile è uno spazio in memoria che contiene un valore.

Esempi:

```CSharp
int score = 0;
float speed = 6f;
bool isGrounded = false;
string playerName = "Player";
```

Ogni variabile ha:

- un tipo;
- un nome;
- eventualmente un valore iniziale.

3.2 Tipi di variabili comuni

`int`

Numero intero.

```CSharp
int score = 0;
int lives = 3;
```

Si usa per punteggi, vite, quantità.

`float`

Numero decimale.

```CSharp
float moveSpeed = 6f;
float jumpForce = 12f;
```

In C#, i numeri decimali di tipo float usano la `f` finale.

Corretto:

```CSharp
float speed = 6f;
```

Meno corretto per un float:

```CSharp
float speed = 6.0;
```

`bool`

Valore vero/falso.

```CSharp
bool isGrounded = false;
bool isGameOver = false;
```

Si usa per condizioni.

`string`

Testo.

```CSharp
string sceneName = "WinScene";
```

`Vector2`

Coppia di valori X e Y.

```CSharp
Vector2 movement = new Vector2(1f, 0f);
```

Si usa spesso nel 2D.

`Vector3`

Tre valori X, Y e Z.

```CSharp
Vector3 offset = new Vector3(0f, 1f, -10f);
```

Anche nei giochi 2D la camera usa spesso `Vector3`, perché ha posizione anche sull'asse Z.

Riferimenti a componenti

```CSharp
Rigidbody2D rb;
Animator animator;
Transform target;
Text scoreText;
```

Queste variabili non contengono semplici numeri, ma riferimenti ad altri componenti o oggetti.

3.3 Variabili pubbliche e private

Variabile pubblica

```CSharp
public float moveSpeed = 6f;
```

Una variabile `public` appare nell'Inspector di Unity.

Questo permette di modificarla senza cambiare il codice.

Esempio: puoi cambiare la velocità del Player direttamente dall'Inspector.

Variabile privata

```CSharp
private Rigidbody2D rb;
```

Una variabile `private` è usata solo dentro lo script.

Non appare nell'Inspector.

Si usa per dati interni che non devono essere modificati a mano.

Quando usare `public`

Usa `public` per valori che vuoi regolare da Unity:

```CSharp
public float moveSpeed;
public float jumpForce;
public Transform target;
```

Quando usare `private`

Usa `private` per dati tecnici interni:

```CSharp
private Rigidbody2D rb;
private bool isGrounded;
private int score;
```

---

4. Metodi e funzioni

4.1 Che cos'è un metodo

Un metodo è un blocco di codice che esegue un'azione.

Esempio:

```CSharp
void Jump()
{
```

```
    Debug.Log("Salto!");
}
```

Un metodo può:

- non restituire nulla;
- restituire un valore;
- ricevere parametri.

4.2 `void`

CSharp
```
void Start()
```

`void` significa che il metodo non restituisce nessun valore.

Esempio:

CSharp
```
void AddScore()
{
    score += 1;
}
```

Questo metodo modifica una variabile, ma non restituisce nulla.

4.3 Metodi con parametri

CSharp
```
public void AddScore(int amount)
{
    score += amount;
}
```

Qui `amount` è un parametro.

Il metodo riceve un numero intero e lo usa per aumentare il punteggio.

Esempio:

```CSharp
AddScore(1);
AddScore(5);
AddScore(10);
```

4.4 Metodi che restituiscono un valore

```CSharp
public int GetScore()
{
    return score;
}
```

Questo metodo restituisce un numero intero.

La parola `return` indica il valore che viene restituito.

---

5. Metodi speciali di Unity

Unity chiama automaticamente alcuni metodi se sono presenti nello script.

5.1 `Awake()`

```CSharp
void Awake()
{
    rb = GetComponent<Rigidbody2D>();
}
```

`Awake()` viene chiamato quando l'oggetto viene inizializzato.

Si usa spesso per recuperare componenti presenti sullo stesso GameObject.

Esempi:

```CSharp
rb = GetComponent<Rigidbody2D>();
animator = GetComponent<Animator>();
```

5.2 `Start()`

```CSharp
void Start()
{
    UpdateScoreUI();
}
```

`Start()` viene chiamato prima del primo frame di gioco.

Si usa per inizializzare valori, UI o stato iniziale.

Differenza semplice:

- `Awake()` prepara i riferimenti;
- `Start()` avvia la logica iniziale.

5.3 `Update()`

```CSharp
void Update()
{
    float horizontalInput = Input.GetAxisRaw("Horizontal");
}
```

`Update()` viene chiamato una volta per frame.

Si usa per:

- leggere input;
- controllare tasti;
- aggiornare animazioni;
- gestire logica continua.

5.4 `FixedUpdate()`

`FixedUpdate()` viene chiamato a intervalli fissi.

È utile per la fisica.

In questa guida abbiamo usato `Update()` anche per modificare la velocity del Player per semplicità. In progetti più ordinati, spesso si legge l'input in `Update()` e si applica il movimento fisico in `FixedUpdate()`.

5.5 `LateUpdate()`

```CSharp
void LateUpdate()
{
    // movimento camera
}
```

`LateUpdate()` viene chiamato dopo `Update()`.

È utile per la camera, perché permette alla camera di seguire il Player dopo che il Player si è già mosso.

5.6 `OnCollisionEnter2D()`

```CSharp
void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.collider.CompareTag("Ground"))
    {
        isGrounded = true;
    }
}
```

Viene chiamato quando questo oggetto entra in collisione fisica con un altro Collider2D.

Serve per collisioni solide, per esempio:

- Player contro pavimento;

- Player contro muro;
- Player contro nemico.

5.7 `OnTriggerEnter2D()`

```CSharp
void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        Destroy(gameObject);
    }
}
```

Viene chiamato quando questo oggetto entra in contatto con un trigger.

Serve per oggetti che non devono bloccare il Player, per esempio:

- monete;
- power-up;
- porta di fine livello;
- checkpoint.

---

6. Script HelloUnity

Codice completo

```CSharp
using UnityEngine;

public class HelloUnity : MonoBehaviour
{
    void Start()
    {
        Debug.Log("Ciao Unity!");
    }
}
```

Spiegazione

`Debug.Log`

```CSharp
Debug.Log("Ciao Unity!");
```

Scrive un messaggio nella Console di Unity.

È utilissimo per controllare se uno script viene eseguito.

Task — Modifica il messaggio

1. Cambia il testo dentro `Debug.Log`.
2. Premi Play.
3. Controlla la Console.

---

7. Script PlayerController

Codice completo

```CSharp
using UnityEngine;

[RequireComponent(typeof(Rigidbody2D))]
public class PlayerController : MonoBehaviour
{
    public float moveSpeed = 6f;
    public float jumpForce = 12f;

    private Rigidbody2D rb;
    private Animator animator;
    private bool isGrounded;

    void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        animator = GetComponent<Animator>();
    }
```

```
    void Update()
    {
        float horizontalInput = Input.GetAxisRaw("Horizontal");

        rb.velocity = new Vector2(horizontalInput * moveSpeed,
rb.velocity.y);

        if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
            isGrounded = false;
        }

        if (animator != null)
        {
            animator.SetFloat("Speed",
Mathf.Abs(horizontalInput));
            animator.SetBool("IsGrounded", isGrounded);
            animator.SetFloat("VerticalVelocity", rb.velocity.y);
        }
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.collider.CompareTag("Ground"))
        {
            isGrounded = true;
        }
    }
}
```

7.1 [RequireComponent(typeof(Rigidbody2D))]

CSharp

```
[RequireComponent(typeof(Rigidbody2D))]
```

È un attributo di Unity.

Dice a Unity che questo script ha bisogno di un `Rigidbody2D` per funzionare.

Se assegni `PlayerController` a un GameObject senza Rigidbody2D, Unity può aggiungerlo automaticamente.

Serve a ridurre errori di configurazione.

7.2 Variabili pubbliche

```CSharp
public float moveSpeed = 6f;
public float jumpForce = 12f;
```

Queste variabili appaiono nell'Inspector.

`moveSpeed`

Controlla la velocità orizzontale del Player.

Se aumenti il valore, il Player corre più veloce.

`jumpForce`

Controlla la forza del salto.

Se aumenti il valore, il Player salta più in alto.

7.3 Variabili private

```CSharp
private Rigidbody2D rb;
private Animator animator;
private bool isGrounded;
```

`rb`

Contiene il riferimento al componente `Rigidbody2D` del Player.

Serve per modificare la fisica, in particolare la velocità.

`animator`

Contiene il riferimento al componente `Animator`.

Serve per aggiornare i parametri delle animazioni.

`isGrounded`

Indica se il Player è a terra.

- `true`: il Player può saltare;
- `false`: il Player è in aria.

7.4 `Awake()` nel PlayerController

```CSharp
void Awake()
{
    rb = GetComponent<Rigidbody2D>();
    animator = GetComponent<Animator>();
}
```

`GetComponent<T>()`

Cerca un componente sullo stesso GameObject.

```CSharp
rb = GetComponent<Rigidbody2D>();
```

Significa:

“Prendi il Rigidbody2D presente su questo oggetto e salvalo nella variabile rb”.

Questo evita di cercarlo ogni frame.

7.5 Lettura dell’input

```CSharp
float horizontalInput = Input.GetAxisRaw("Horizontal");
```

Legge l’input orizzontale.

Valori possibili:

- `-1`: sinistra;
- `0`: nessun input;
- `1`: destra.

Con i controlli standard Unity, funziona con:

- frecce sinistra/destra;
- A/D.

7.6 Movimento orizzontale

```CSharp
rb.velocity = new Vector2(horizontalInput * moveSpeed,
rb.velocity.y);
```

Questa riga imposta la velocità del Rigidbody2D.

`rb.velocity` è un `Vector2`, quindi ha:

- X: velocità orizzontale;
- Y: velocità verticale.

```CSharp
horizontalInput * moveSpeed
```

Calcola la velocità X.

Esempio:

- input destra: `1 * 6 = 6`;
- input sinistra: `-1 * 6 = -6`;
- nessun input: `0 * 6 = 0`.

```CSharp
rb.velocity.y
```

Mantiene invariata la velocità verticale.

Questo è importante: se mettessimo Y a 0, il Player non cadrebbe e non salterebbe correttamente.

7.7 Salto

```CSharp
if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
{
    rb.velocity = new Vector2(rb.velocity.x, jumpForce);
    isGrounded = false;
}
```

`Input.GetKeyDown(KeyCode.Space)`

Controlla se il tasto Spazio è stato premuto in quel frame.

`&&`

Significa “e anche”.

La condizione è vera solo se:

- è stato premuto Spazio;
- `isGrounded` è `true`.

Perché controllare `isGrounded`

Serve a impedire il salto infinito in aria.

Movimento verticale del salto

```CSharp
rb.velocity = new Vector2(rb.velocity.x, jumpForce);
```

Mantiene la velocità X e imposta la velocità Y al valore del salto.

Perché `isGrounded = false`

Dopo il salto, il Player non è più a terra. Quindi non deve poter saltare di nuovo finché non tocca il Ground.

7.8 Aggiornamento Animator

```CSharp
if (animator != null)
{
    animator.SetFloat("Speed", Mathf.Abs(horizontalInput));
    animator.SetBool("IsGrounded", isGrounded);
    animator.SetFloat("VerticalVelocity", rb.velocity.y);
}
```

`animator != null`

Controlla che il riferimento all'Animator esista.

Se il Player non ha Animator, il codice dentro l'if non viene eseguito.

Questo evita errori.

`SetFloat`

```CSharp
animator.SetFloat("Speed", Mathf.Abs(horizontalInput));
```

Imposta un parametro float dell'Animator.

`Speed` può essere usato per passare da Idle a Run.

`Mathf.Abs`

Restituisce il valore assoluto.

Esempi:

- `Mathf.Abs(-1)` restituisce `1`;
- `Mathf.Abs(0)` restituisce `0`;
- `Mathf.Abs(1)` restituisce `1`.

Serve perché per l'animazione Run non importa se il Player va a sinistra o destra: importa solo che si stia muovendo.

`SetBool`

```CSharp
animator.SetBool("IsGrounded", isGrounded);
```

Imposta un parametro booleano dell’Animator.

Può essere usato per distinguere animazioni a terra e in aria.

`VerticalVelocity`

```CSharp
animator.SetFloat("VerticalVelocity", rb.velocity.y);
```

Indica se il Player sta salendo o scendendo.

Può servire per animazioni Jump e Fall.

7.9 Collisione con il Ground

```CSharp
void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.collider.CompareTag("Ground"))
    {
        isGrounded = true;
    }
}
```

Quando il Player collide con un oggetto, Unity chiama questo metodo.

`Collision2D collision`

Contiene informazioni sulla collisione.

`collision.collider`

È il Collider2D dell’oggetto toccato.

`CompareTag("Ground")`

Controlla se l’oggetto toccato ha tag `Ground`.

Se sì, il Player è considerato a terra.

---

8. Script Coin

Codice completo

CSharp
```
using UnityEngine;

public class Coin : MonoBehaviour
{
    public int value = 1;

    void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            GameManager.Instance.AddScore(value);
            Destroy(gameObject);
        }
    }
}
```

8.1 Variabile `value`

CSharp
```
public int value = 1;
```

Indica quanti punti vale la moneta.

È pubblica, quindi puoi cambiare il valore dall'Inspector.

Esempi:

- moneta normale: `1`;
- moneta speciale: `5`;
- gemma rara: `10`.

8.2 `OnTriggerEnter2D`

```CSharp
void OnTriggerEnter2D(Collider2D other)
```

Viene chiamato quando qualcosa entra nel trigger della moneta.

La moneta deve avere:

- `Collider2D`;
- `Is Trigger` attivo.

Il Player deve avere:

- `Collider2D`;
- normalmente anche `Rigidbody2D`.

8.3 Controllo del Player

```CSharp
if (other.CompareTag("Player"))
```

Controlla se l'oggetto entrato nel trigger ha tag `Player`.

Serve a evitare che la moneta venga raccolta da altri oggetti, come nemici o piattaforme.

8.4 Aggiunta punteggio

```CSharp
GameManager.Instance.AddScore(value);
```

Questa riga chiama il metodo `AddScore` del GameManager.

Gli passa il valore della moneta.

Esempio:

```CSharp
value = 1;
```

allora viene chiamato:

```CSharp
AddScore(1);
```

8.5 Distruzione della moneta

```CSharp
Destroy(gameObject);
```

Distrugge il GameObject a cui è assegnato lo script.

In questo caso distrugge la moneta raccolta.

`gameObject`

Rappresenta il GameObject su cui si trova lo script.

Se lo script `Coin` è sulla moneta, `gameObject` è quella moneta.

---

9. Script GameManager

Codice completo

```CSharp
using UnityEngine;
using UnityEngine.UI;

public class GameManager : MonoBehaviour
{
    public static GameManager Instance { get; private set; }

    public Text scoreText;
    private int score;

    void Awake()
    {
        if (Instance != null && Instance != this)
        {
```

```
            Destroy(gameObject);
            return;
        }

        Instance = this;
    }

    void Start()
    {
        UpdateScoreUI();
    }

    public void AddScore(int amount)
    {
        score += amount;
        UpdateScoreUI();
    }

    private void UpdateScoreUI()
    {
        if (scoreText != null)
        {
            scoreText.text = "Score: " + score;
        }
    }
}
```

9.1 `using UnityEngine.UI;`

CSharp

```
using UnityEngine.UI;
```

Importa le classi UI classiche di Unity.

Serve per usare `Text`.

Senza questa riga, Unity non riconoscerebbe:

```CSharp
public Text scoreText;
```

9.2 Singleton semplice

```CSharp
public static GameManager Instance { get; private set; }
```

Questa riga crea un punto di accesso globale al GameManager.

Permette di scrivere da altri script:

```CSharp
GameManager.Instance.AddScore(value);
```

`static`

Significa che la proprietà appartiene alla classe, non a una singola variabile locale.

`get; private set;`

Significa:

- altri script possono leggere `Instance`;
- solo `GameManager` può modificarla.

9.3 Controllo duplicati

```CSharp
if (Instance != null && Instance != this)
{
    Destroy(gameObject);
    return;
}

Instance = this;
```

Questo blocco assicura che ci sia un solo GameManager.

`Instance != null`

Controlla se esiste già un GameManager.

`Instance != this`

Controlla se l'istanza già esistente non è questo oggetto.

`Destroy(gameObject)`

Distrugge il duplicato.

`return`

Interrompe il metodo.

`Instance = this`

Salva questo GameManager come istanza principale.

9.4 Variabile `scoreText`

```CSharp
public Text scoreText;
```

È un riferimento al testo della UI.

Essendo pubblica, appare nell'Inspector.

Devi trascinare l'oggetto `ScoreText` nel campo dello script.

9.5 Variabile `score`

```CSharp
private int score;
```

Contiene il punteggio attuale.

È privata perché deve essere modificata solo dal GameManager.

9.6 Metodo `AddScore`

```CSharp
public void AddScore(int amount)
{
    score += amount;
    UpdateScoreUI();
}
```

Aggiunge punti al punteggio.

`score += amount`

È una forma abbreviata di:

```CSharp
score = score + amount;
```

Dopo aver cambiato il punteggio, aggiorna la UI.

9.7 Metodo `UpdateScoreUI`

```CSharp
private void UpdateScoreUI()
{
    if (scoreText != null)
    {
        scoreText.text = "Score: " + score;
    }
}
```

Aggiorna il testo visibile sullo schermo.

`scoreText != null`

Controlla che il campo sia stato assegnato.

Se non è stato assegnato, evita un errore.

`scoreText.text`

Modifica il contenuto del testo UI.

---

10. Script CameraFollow

Codice completo

```CSharp
using UnityEngine;

public class CameraFollow : MonoBehaviour
{
    public Transform target;
    public float smoothSpeed = 5f;
    public Vector3 offset = new Vector3(0f, 1f, -10f);

    void LateUpdate()
    {
        if (target == null) return;

        Vector3 desiredPosition = target.position + offset;
        transform.position = Vector3.Lerp(transform.position,
desiredPosition, smoothSpeed * Time.deltaTime);
    }
}
```

10.1 Variabile `target`

```CSharp
public Transform target;
```

È l’oggetto da seguire.

Nel nostro caso è il Transform del Player.

Essendo pubblica, puoi trascinare il Player nel campo `target` dall’Inspector.

10.2 Variabile `smoothSpeed`

```CSharp
public float smoothSpeed = 5f;
```

Controlla quanto velocemente la camera raggiunge il Player.

Valore basso:

- camera più lenta e morbida.

Valore alto:

- camera più rapida e rigida.

10.3 Variabile `offset`

```CSharp
public Vector3 offset = new Vector3(0f, 1f, -10f);
```

Indica la distanza tra camera e Player.

Valori:

- X: spostamento orizzontale;
- Y: spostamento verticale;
- Z: distanza della camera.

Nel 2D, la camera di solito deve stare a Z = -10.

10.4 `LateUpdate()`

```CSharp
void LateUpdate()
```

La camera si aggiorna dopo il movimento del Player.

Questo riduce scatti e ritardi visivi.

10.5 Controllo target

```CSharp
if (target == null) return;
```

Se non è stato assegnato nessun target, il metodo si interrompe.

Serve a evitare errori.

10.6 Posizione desiderata

```CSharp
Vector3 desiredPosition = target.position + offset;
```

Calcola dove dovrebbe andare la camera.

Esempio:

- Player a `(5, 2, 0)`;
- offset `(0, 1, -10)`;
- posizione desiderata camera `(5, 3, -10)`.

10.7 Movimento morbido con `Lerp`

```CSharp
transform.position = Vector3.Lerp(transform.position,
desiredPosition, smoothSpeed * Time.deltaTime);
```

`Vector3.Lerp` interpola tra due posizioni.

Significa che la camera non salta subito alla posizione desiderata, ma si avvicina gradualmente.

Parametri:

```CSharp
Vector3.Lerp(posizioneIniziale, posizioneFinale, fattore)
```

`transform.position`

Posizione attuale della camera.

`desiredPosition`

Posizione che la camera vuole raggiungere.

`smoothSpeed * Time.deltaTime`

Controlla quanto velocemente avviene lo spostamento.

`Time.deltaTime` rende il movimento più indipendente dal numero di frame al secondo.

---

11. Animator nello script

11.1 Che cos'è l'Animator

L'Animator gestisce le animazioni di un GameObject.

Lo script non riproduce direttamente i singoli frame. Lo script aggiorna parametri, e l'Animator decide quale animazione mostrare.

11.2 Parametri usati

Nel nostro PlayerController abbiamo usato:

```CSharp
animator.SetFloat("Speed", Mathf.Abs(horizontalInput));
animator.SetBool("IsGrounded", isGrounded);
animator.SetFloat("VerticalVelocity", rb.velocity.y);
```

`Speed`

Indica se il Player si sta muovendo.

- `0`: fermo;
- maggiore di `0`: in movimento.

`IsGrounded`

Indica se il Player è a terra.

`VerticalVelocity`

Indica se il Player sale o scende.

- valore positivo: sale;
- valore negativo: scende;
- circa zero: stabile.

11.3 Nomi dei parametri

I nomi nello script devono essere identici ai nomi nell'Animator.

Se nello script scrivi:

```CSharp
animator.SetFloat("Speed", 1f);
```

nell'Animator deve esistere un parametro chiamato esattamente:

```None
Speed
```

Attenzione a:

- maiuscole;
- minuscole;
- spazi;
- errori di battitura.

---

12. Parole chiave importanti negli script

`public`

Rende una variabile o un metodo accessibile dall'esterno.

In Unity, una variabile pubblica appare nell'Inspector.

`private`

Rende una variabile o un metodo accessibile solo dentro la stessa classe.

`static`

Indica qualcosa legato alla classe, non a una singola istanza.

Usato nel GameManager per `Instance`.

`void`

Indica che un metodo non restituisce valori.

`return`

Interrompe un metodo o restituisce un valore.

Esempio:

```CSharp
if (target == null) return;
```

significa: se manca il target, esci dal metodo.

`new`

Crea un nuovo oggetto o valore.

Esempio:

```CSharp
new Vector2(1f, 0f);
new Vector3(0f, 1f, -10f);
```

`this`

Indica l'oggetto corrente.

Nel GameManager:

```CSharp
Instance = this;
```

significa: questa istanza è il GameManager principale.

---

13. Errori comuni negli script

Nome file diverso dal nome classe

Errore:

```None
PlayerMovement.cs
```

ma dentro:

```CSharp
public class PlayerController : MonoBehaviour
```

Soluzione: nome file e nome classe devono coincidere.

Mancano le graffe

Errore comune quando si copia codice.

Controlla sempre che ogni `{` abbia la sua `}`.

Punto e virgola mancante

Errore:

```CSharp
public float speed = 5f
```

Corretto:

```CSharp
public float speed = 5f;
```

Tag scritto male

Errore:

```CSharp
CompareTag("player")
```

ma il tag è:

```None
Player
```

Unity distingue maiuscole e minuscole.

Riferimento non assegnato

Errore tipico:

```None
NullReferenceException
```

Significa che stai usando qualcosa che non esiste o non è stato collegato.

Esempi:

- `scoreText` non assegnato nell'Inspector;
- `target` della camera vuoto;
- manca `Animator`;
- manca `Rigidbody2D`.

---

14. Task di approfondimento

Task A — Cambia velocità e salto

1. Apri `PlayerController`.
2. Cambia i valori iniziali di `moveSpeed` e `jumpForce`.
3. Premi Play.
4. Poi cambia gli stessi valori dall'Inspector.

Domanda: quale valore viene usato da Unity quando lo script è già assegnato a un GameObject?

Risposta: Unity usa il valore salvato nell'Inspector, non necessariamente quello scritto nel codice dopo la prima assegnazione.

Task B — Aggiungi doppio punteggio

Modifica una moneta speciale impostando `value = 5` dall'Inspector.

Obiettivo: capire perché `value` è pubblica.

Task C — Messaggi di debug

Aggiungi nel PlayerController:

```CSharp
Debug.Log("Input orizzontale: " + horizontalInput);
```

Osserva la Console mentre premi A/D.

Obiettivo: usare `Debug.Log` per capire cosa succede durante il gioco.

Task D — Camera più rigida

Nel componente `CameraFollow`, aumenta `smoothSpeed` a 20.

Poi abbassalo a 1.

Osserva la differenza.

Task E — Protezione dagli errori

Rimuovi temporaneamente il target dalla Main Camera.

Premi Play.

Grazie a:

```CSharp
if (target == null) return;
```

lo script non genera errore.

---

15. Versione commentata del PlayerController

```CSharp
using UnityEngine;

// Questo attributo assicura che l'oggetto abbia un Rigidbody2D
[RequireComponent(typeof(Rigidbody2D))]
public class PlayerController : MonoBehaviour
{
    // Velocità orizzontale del personaggio, modificabile
dall'Inspector
    public float moveSpeed = 6f;

    // Forza del salto, modificabile dall'Inspector
    public float jumpForce = 12f;

    // Riferimento al componente Rigidbody2D
    private Rigidbody2D rb;
```

```
    // Riferimento al componente Animator
    private Animator animator;

    // Indica se il Player è a terra
    private bool isGrounded;

    void Awake()
    {
        // Recupera il Rigidbody2D presente sul Player
        rb = GetComponent<Rigidbody2D>();

        // Recupera l'Animator, se presente
        animator = GetComponent<Animator>();
    }

    void Update()
    {
        // Legge input orizzontale: -1 sinistra, 0 fermo, 1
destra
        float horizontalInput = Input.GetAxisRaw("Horizontal");

        // Applica la velocità orizzontale mantenendo la velocità
verticale
        rb.velocity = new Vector2(horizontalInput * moveSpeed,
rb.velocity.y);

        // Se premo Spazio e sono a terra, salto
        if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
        {
            rb.velocity = new Vector2(rb.velocity.x, jumpForce);
            isGrounded = false;
        }

        // Se esiste un Animator, aggiorno i parametri
        if (animator != null)
        {
            animator.SetFloat("Speed",
Mathf.Abs(horizontalInput));
            animator.SetBool("IsGrounded", isGrounded);
```

```
            animator.SetFloat("VerticalVelocity", rb.velocity.y);
        }
    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        // Se tocco un oggetto con tag Ground, posso saltare di
nuovo
        if (collision.collider.CompareTag("Ground"))
        {
            isGrounded = true;
        }
    }
}
```

---

16. Metodo di studio consigliato

Per imparare davvero gli script, non copiarli soltanto.

Per ogni script:

1. leggilo riga per riga;
2. cambia un solo valore alla volta;
3. premi Play;
4. osserva cosa cambia;
5. controlla la Console;
6. ripristina il codice se qualcosa si rompe.

La domanda più importante da farsi è sempre:

“Questa riga quale dato modifica? Quale oggetto controlla? In quale momento viene eseguita?”