Tutorial Unity 6

Guía completa para montar el proyecto en Unity 6

Índice

1. Introducción

JustTakeIt es un juego de plataformas 2D donde el jugador debe recoger monedas dentro de un escenario lleno de plataformas, enemigos (fantasmas) y trampas (pinchos). La mecánica principal consiste en saltar entre plataformas y usar un arma de retroceso que impulsa al jugador en la dirección contraria a la que apunta, acumulando fuerzas del Rigidbody 2D para alcanzar zonas más altas.

El juego termina cuando el jugador toca un enemigo o un pincho, mostrando una pantalla de Game Over con las monedas recogidas y el tiempo empleado.

El asset de sprites visual usado en el proyecto:

https://assetstore.unity.com/packages/tools/game-toolkits/2d-platformer-229878

---

2. Conceptos básicos de Unity

Antes de empezar, es importante entender los conceptos fundamentales del motor Unity que utilizaremos en este proyecto.

2.1 GameObject

Un **GameObject** es la unidad básica de Unity. Todo lo que existe en la escena es un GameObject: el jugador, las monedas, la cámara, incluso objetos invisibles como el GameManager. Por sí solo, un GameObject no hace nada; necesita **componentes** para tener comportamiento.

2.2 Componente (Component)

Los **componentes** son piezas que se añaden a un GameObject para darle funcionalidad. Algunos componentes son proporcionados por Unity (Transform, Rigidbody, Collider) y otros los creamos nosotros como scripts en C#.

Ejemplos:
- **Transform**: posición, rotación y escala. TODOS los GameObjects lo tienen.
- **SpriteRenderer**: dibuja una imagen 2D en pantalla.
- **Rigidbody2D**: añade física (gravedad, fuerzas, velocidad).
- **Collider2D**: define la forma de colisión del objeto.

2.3 Rigidbody 2D

El **Rigidbody2D** es el componente que conecta un GameObject con el motor de física de Unity. Permite que un objeto sea afectado por:
- **Gravedad**: el objeto cae si no hay nada debajo.
- **Fuerzas**: podemos empujar el objeto con `AddForce()`.
- **Velocidad**: podemos mover el objeto asignando `linearVelocity`.
- **Colisiones**: el objeto reacciona al chocar con otros.

Propiedades importantes:
- **Mass**: masa del objeto (afecta a cómo reacciona a las fuerzas).

- **Gravity Scale**: cuánta gravedad afecta (0 = sin gravedad).
- **Interpolate**: suaviza el movimiento visual. Usamos `Interpolate` para que no se vea a tirones.
- **Collision Detection**: `Discrete` es suficiente para objetos que no se mueven extremadamente rápido.

2.4 Collider 2D

Un **Collider2D** define la "caja" invisible de colisión de un objeto. Hay varios tipos:
- **BoxCollider2D**: forma rectangular.
- **CircleCollider2D**: forma circular.
- **CapsuleCollider2D**: forma de cápsula (usada en nuestro jugador).
- **TilemapCollider2D**: se adapta a los tiles de un Tilemap.

Propiedad "Is Trigger": cuando está activada, el Collider no bloquea físicamente el paso, sino que detecta cuándo otro objeto lo atraviesa. Se usa para monedas, zonas de daño, etc.

- **Colisión normal** (Is Trigger = OFF): los objetos chocan y se detienen.
- **Trigger** (Is Trigger = ON): los objetos se atraviesan, pero se detecta el contacto mediante `OnTriggerEnter2D()`.

2.5 Tags

Los **Tags** son etiquetas que asignamos a los GameObjects para identificarlos desde el código. Ejemplos:
- `Player`: identifica al jugador.
- `Enemy`: identifica a los enemigos.
- `Spike`: identifica a los pinchos.

Desde el código comprobamos el tag así:

```
if (collision.CompareTag("Player"))
{
    // Es el jugador
}
```

2.6 Layers (Capas)

Las **Layers** son similares a los tags pero se usan principalmente para controlar qué objetos interactúan con qué otros en la física. En nuestro proyecto, usamos

una layer llamada `ground` para que el Raycast del jugador solo detecte el suelo y no otros objetos.

2.7 Raycast

Un **Raycast** es un "rayo invisible" que lanzamos desde un punto en una dirección. Si el rayo golpea algo, nos devuelve información sobre lo que tocó. Nosotros lo usamos para detectar si el jugador está tocando el suelo:

```
RaycastHit2D hit = Physics2D.Raycast(origen, dirección, dis
tancia, capa);
isGrounded = hit.collider != null;
```

2.8 Physics Material 2D

Un **PhysicsMaterial2D** define cómo interactúa un objeto físico con las superficies:
- **Friction (Fricción)**: cuánto se "pega" a las superficies. Si es alta, el jugador se queda pegado a las paredes al saltar contra ellas.
- **Bounciness**: cuánto rebota.

En nuestro proyecto usamos dos materiales:
- `DefaultFriction`: fricción normal para caminar por el suelo.
- `NoStuckWall` (No Friction): fricción a 0 para que el jugador no se pegue a las paredes cuando está en el aire.

2.9 Patrón Singleton

El **Singleton** es un patrón de diseño que garantiza que solo exista UNA instancia de una clase y que sea accesible globalmente. Lo usamos en el GameManager:

```
public static GameManager Instance;

private void Awake()
{
    Instance = this;
}
```

Así, desde cualquier script podemos hacer:

```
GameManager.Instance.AddPoint();
```

2.10 Time.timeScale

`Time.timeScale` controla la velocidad del tiempo en Unity:
- `1f` = velocidad normal.
- `0f` = todo pausado (física, animaciones, `Time.deltaTime` = 0).
- `2f` = el doble de velocidad.

Lo usamos para "congelar" el juego cuando aparece la pantalla de Game Over.

2.11 Canvas y UI

El **Canvas** es el contenedor principal de toda la interfaz de usuario (UI) en Unity. Todos los elementos de UI (textos, botones, paneles) deben ser hijos de un Canvas.

- **Canvas Scaler**: controla cómo se escala la UI en diferentes resoluciones.
- **TextMeshPro (TMP)**: sistema avanzado de texto que reemplaza al Text legacy de Unity. Ofrece mejor calidad y más opciones de formato.

2.12 SerializeField y Header

- `[SerializeField]`: hace que una variable privada aparezca en el Inspector de Unity, para poder asignar valores sin cambiar el código.
- `[Header("texto")]`: añade un título/sección en el Inspector para organizar visualmente las variables.
- `[Tooltip("texto")]`: muestra un texto de ayuda al pasar el ratón por encima de la variable en el Inspector.

---

3. Estructura del proyecto

3.1 Carpetas del proyecto

```
Assets/
├── 2D Platformer/       ← Assets del juego (sprites, anima
ciones)
├── phisicsMaterials/     ← Materiales de física (NoStuckWa
ll, DefaultFriction)
```

```
├── Scenes/               ← La escena principal (SampleScen
e)
├── Scripts/              ← Todos los scripts C#
│   ├── GameManager.cs
│   ├── PlayerMovement.cs
│   ├── Coin.cs
│   ├── CameraController.cs
│   └── Enemy.cs
└── Settings/             ← Configuraciones del render pipe
line
```

3.2 Jerarquía de la escena

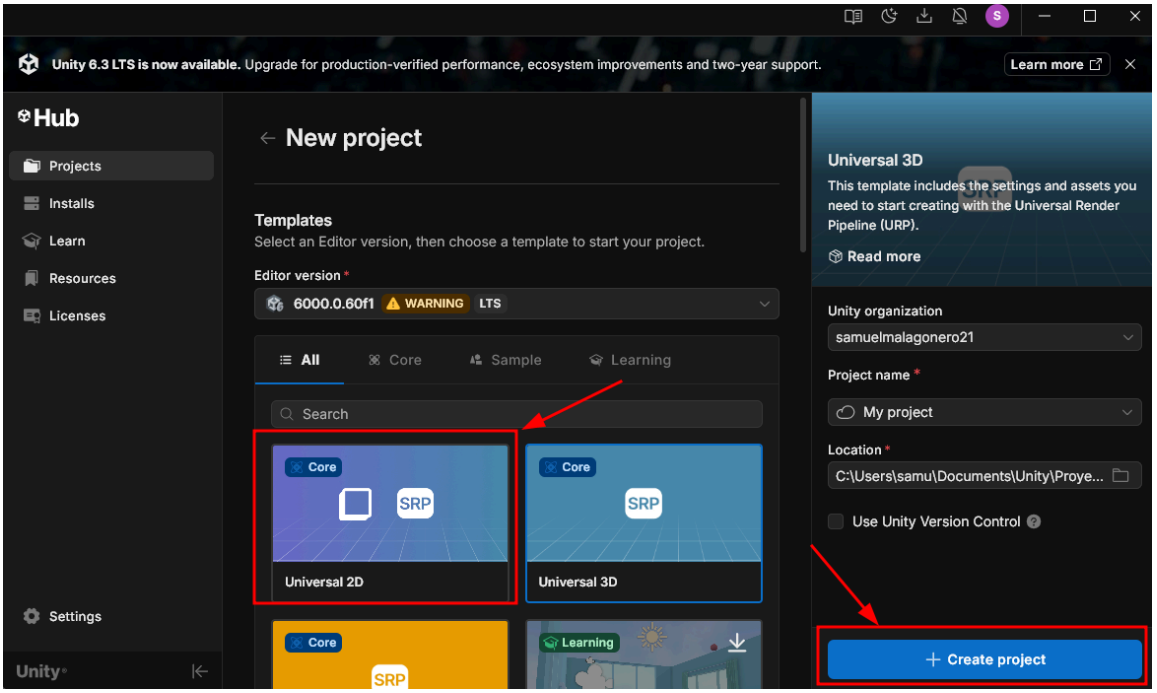
```
SampleScene
├── Main Camera            → CameraController.cs
├── Global Light 2D        → Iluminación 2D
├── Player                 → PlayerMovement.cs + Rigidbody2
D + CapsuleCollider2D
│   ├── Handler Rotation   → Pivote del arma (solo Transfor
m)
│   └── GroundCheck        → Punto desde donde se lanza el
Raycast
├── Spikes                 → Pinchos (tag: Spike, Collider2
D con Is Trigger)
├── Coins                  → Contenedor de monedas (Coin.cs
+ Collider2D trigger)
├── Enemies                → Contenedor de fantasmas
│   ├── Enemy ... Enemy (5) → Enemy.cs + Collider2D trigger
(tag: Enemy)
├── Map
│   ├── Map                → Tilemap (parte VISUAL del map
a)
│   │   └── GroundLayer    → Tilemap pintado con Tile Palet
te (sin colliders)
│   └── Layers             → Colliders INVISIBLES del mapa
│       ├── WallLayers     → Sprites invisibles con BoxColl
ider2D (paredes)
│       │   ├── Wall, Wall (1) ... Wall (13)
```

```
│       └── GroundLayer    → Sprites invisibles con BoxColl
ider2D (suelo)
│           ├── ground, ground (1) ... ground (24+)
├── Canvas                → UI del juego
│   ├── HUD
│   │   ├── ScoreText     → TextMeshPro "Monedas: 0"
│   │   └── TimerText     → TextMeshPro "00:00"
│   └── GameOverPanel     → Panel de muerte (desactivado a
l inicio)
│       ├── GameOverTitle   → TextMeshPro "GAME OVER"
│       ├── FinalScoreText  → TextMeshPro "Monedas: 0"
│       ├── FinalTimeText   → TextMeshPro "Tiempo: 00:00"
│       └── RestartButton   → Botón "Reintentar"
└── GameManager           → GameManager.cs
```

4. Montaje de la escena paso a paso

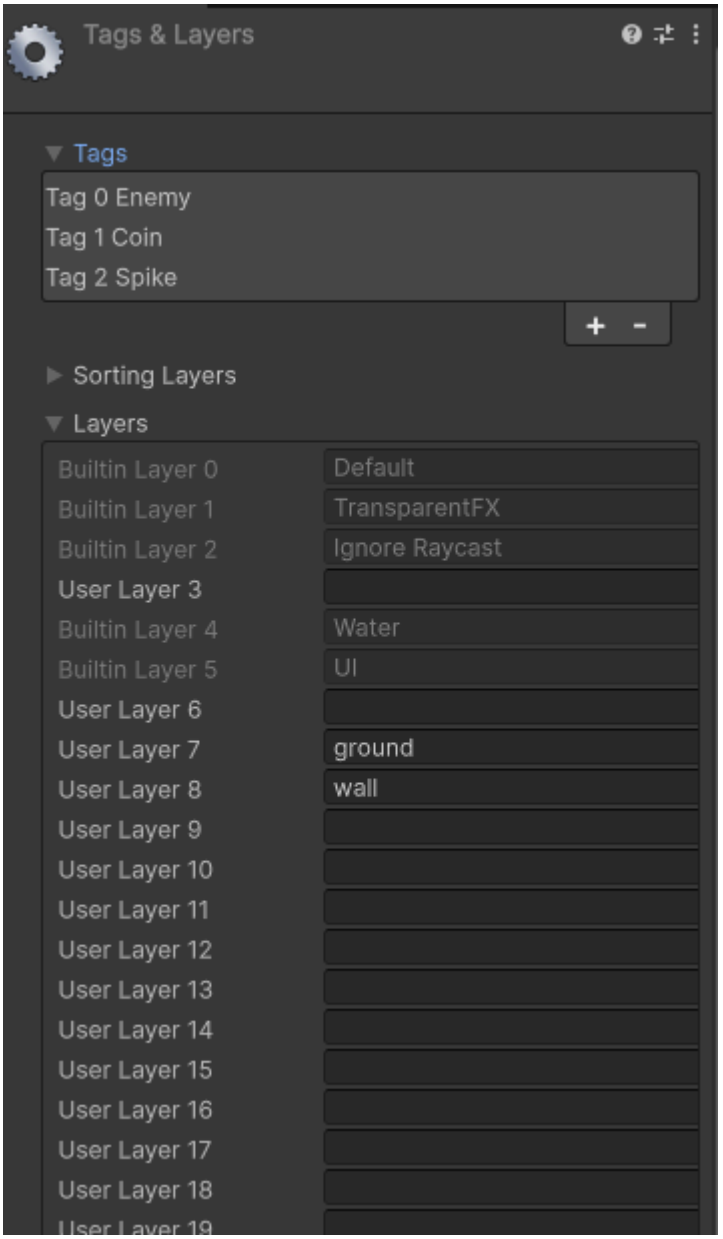
4.1 Crear el proyecto

1. Abre **Unity Hub**.
2. Pulsa **New Project**.
3. Selecciona la plantilla **2D (URP)** — Universal Render Pipeline.
4. Nombre del proyecto: `JustTakeIt`.
5. Pulsa **Create Project**.



4.2 Configurar Tags y Layers

1. Ve a **Edit > Project Settings > Tags and Layers**.
2. En la sección **Tags**, pulsa `+` y añade:
   - `Enemy`
   - `Spike`
3. Comprueba que el tag `Player` ya existe (viene por defecto).
4. En la sección **Layers**, busca un User Layer libre y añade:
   - `ground` (en el User Layer 6, por ejemplo).



4.3 Crear el mapa (Map)

El mapa de JustTakeIt se compone de **dos partes separadas**:

1. **El Tilemap** → es la parte **visual**. Aquí se pintan los tiles que el jugador ve en pantalla (las plataformas, el suelo, las paredes con color). **No tiene colliders.**
2. **Los sprites de colisión** → son sprites **invisibles** colocados encima del Tilemap. Su única función es definir las **zonas de colisión** (dónde puede caminar el jugador, dónde choca con paredes, etc.) mediante BoxCollider2D.

¿Por qué se hace así?

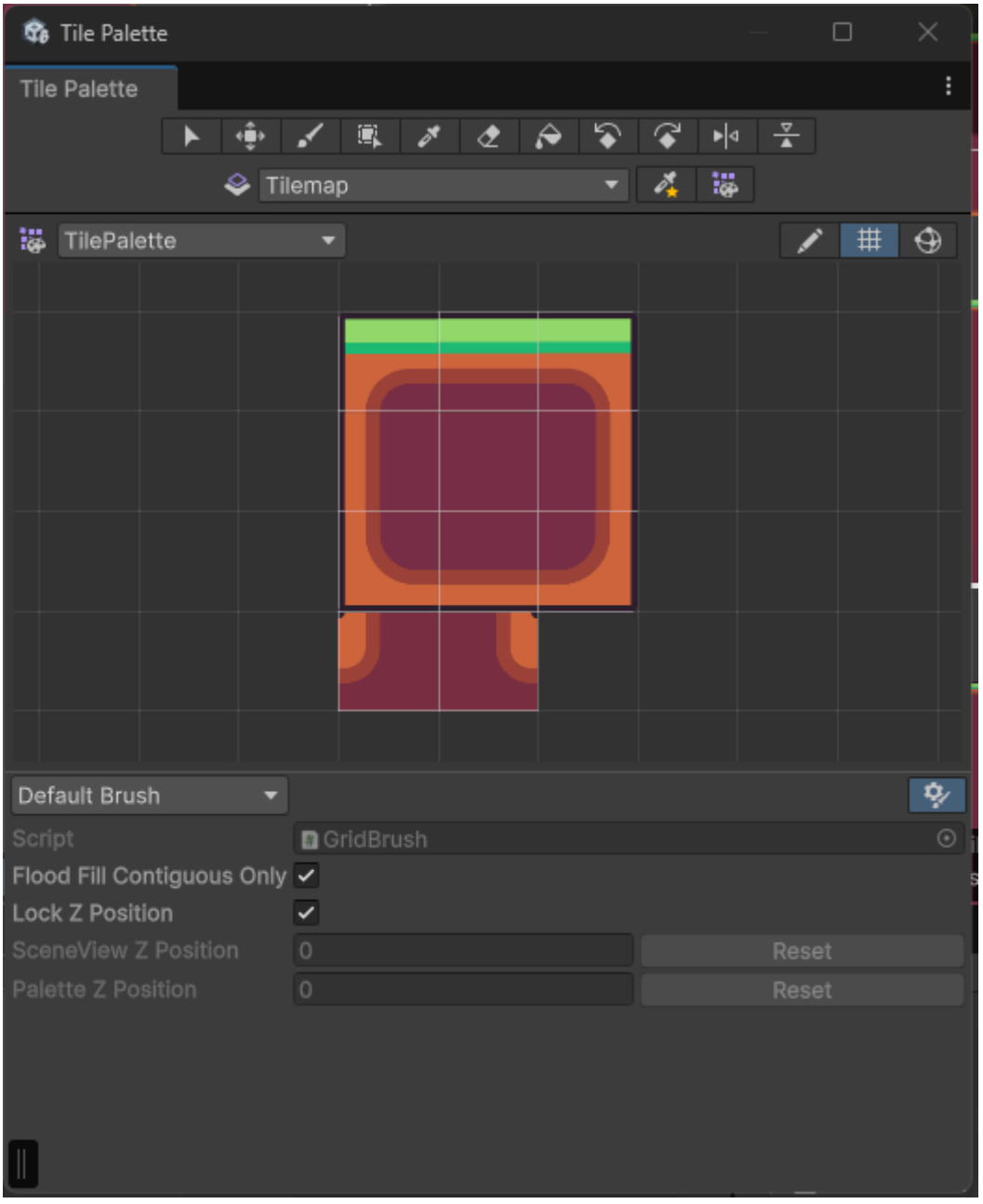
Separar lo visual de las colisiones tiene ventajas:
- **Control total**: puedes definir las colisiones exactamente donde quieras, sin depender de la forma de los tiles.
- **Rendimiento**: un BoxCollider2D simple es más eficiente que un TilemapCollider2D con decenas de tiles.
- **Flexibilidad**: puedes tener zonas decorativas sin colisión, o colisiones sin visual.

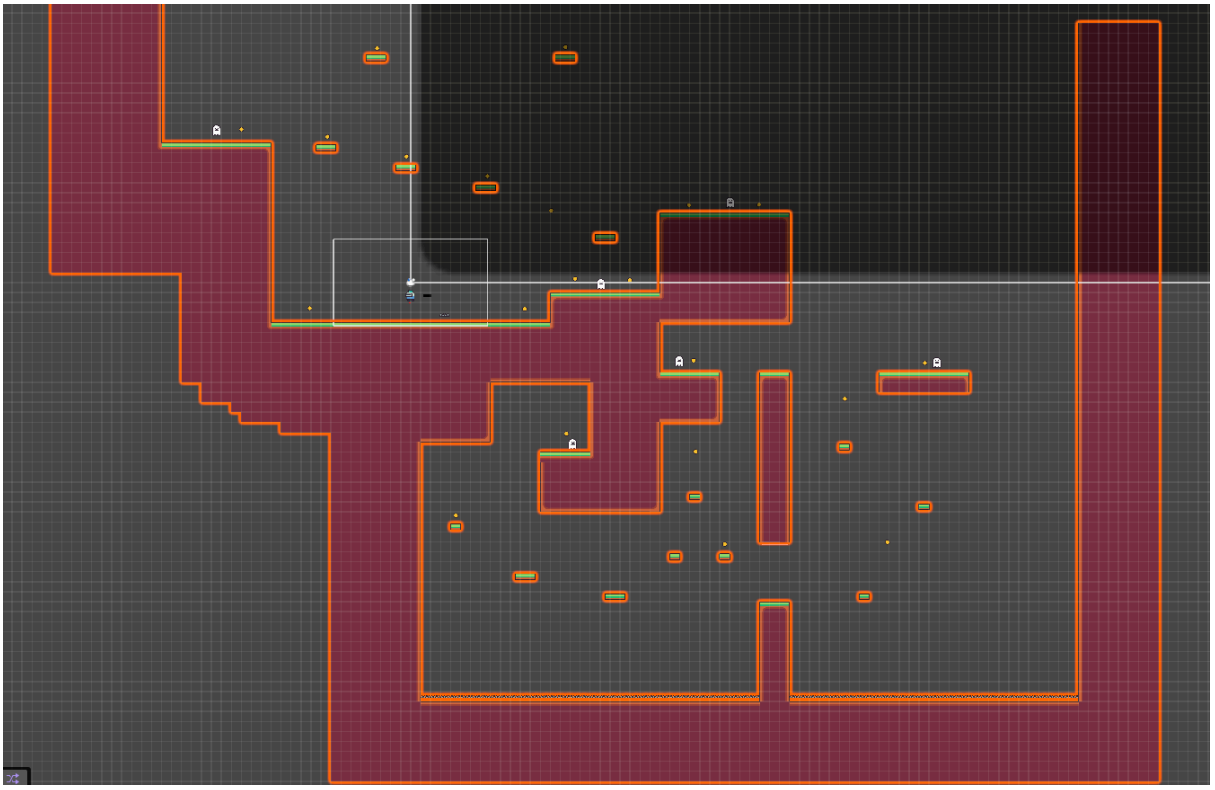
Estructura organizativa

```
Map
├── Map (Tilemap - VISUAL)
│   └── GroundLayer        → Tilemap pintado con la Tile Pa
lette
└── Layers (COLISIONES - invisibles)
    ├── WallLayers         → Sprites con BoxCollider2D para
las paredes
    │   ├── Wall
    │   ├── Wall (1) ... Wall (13)
    └── GroundLayer        → Sprites con BoxCollider2D para
el suelo
        ├── ground
        ├── ground (1) ... ground (24+)
```

Parte 1: Crear el Tilemap (visual)

1. En la Hierarchy, click derecho > **Create Empty** → renombra a `Map`.
2. Dentro de `Map`, click derecho > **2D Object > Tilemap > Rectangular**.
   - Esto crea automáticamente un **Grid** (renómbralo a `Map`) con un **Tilemap** hijo (renómbralo a `GroundLayer`).
3. Abre la **Tile Palette**: ve a **Window > 2D > Tile Palette**.
4. Crea una nueva Palette o usa la existente con tus assets.
5. Pinta el mapa usando la herramienta de pincel en la Tile Palette.
6. **NO añadas TilemapCollider2D** al Tilemap — las colisiones las gestionaremos con los sprites de la carpeta `Layers`.


Tile Palette
Tile Palette
Tilemap
TilePalette
Default Brush
Script
GridBrush
Flood Fill Contiguous Only
Lock Z Position
SceneView Z Position
0
Reset
Palette Z Position
0
Reset



Parte 2: Crear los colliders invisibles

Ahora creamos los sprites que definen las colisiones. Estos sprites se colocan **encima del Tilemap** pero **no se ven en el juego**.

Crear los colliders del suelo

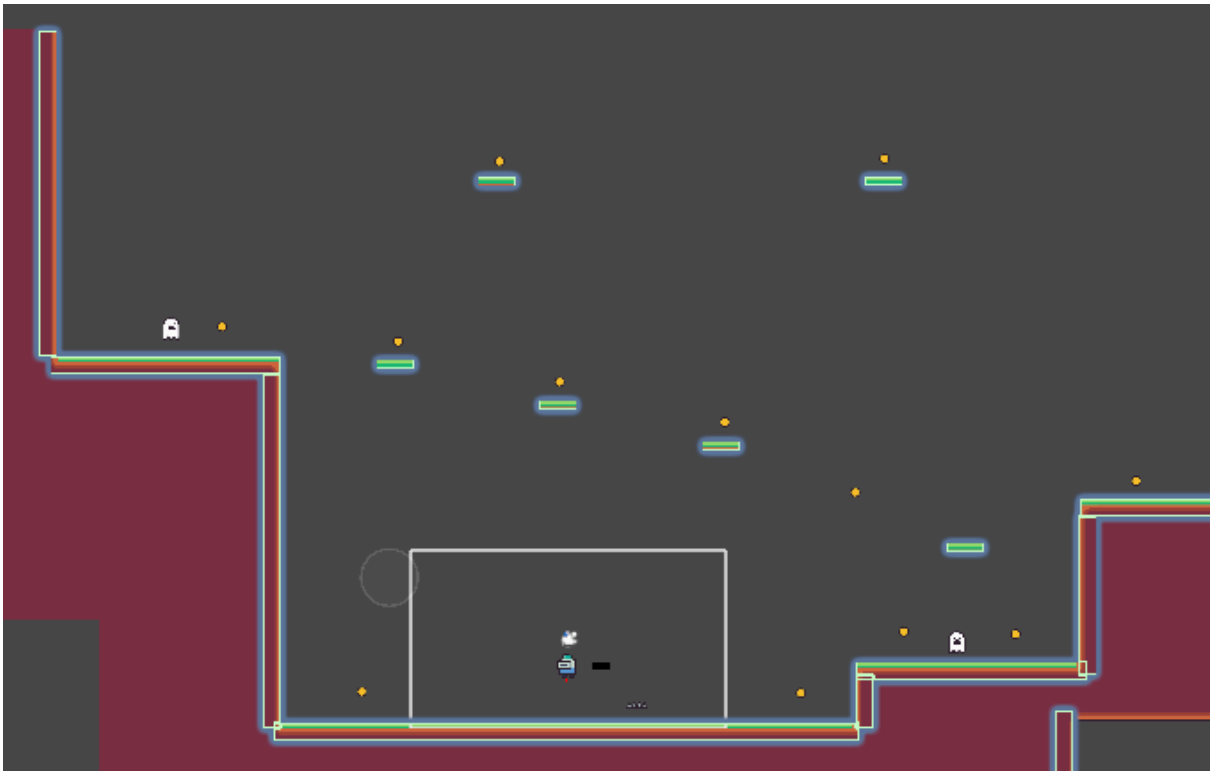
1. Dentro de `Map`, click derecho > **Create Empty** → renombra a `Layers`.
2. Dentro de `Layers`, click derecho > **Create Empty** → renombra a `GroundLayer`.
3. Dentro de `GroundLayer`, click derecho > **2D Object > Sprites > Square**.
4. Renombra a `ground`.
5. En el Inspector:
    - **Transform > Scale**: escálalo para cubrir la zona de suelo del Tilemap. Por ejemplo, `X: 15, Y: 1, Z: 1` para cubrir una plataforma larga.
      > Podemos usar desde la escena la tecla **T** y reescalar directamente la escala
    - **Transform > Position**: colócalo exactamente encima de los tiles de suelo.

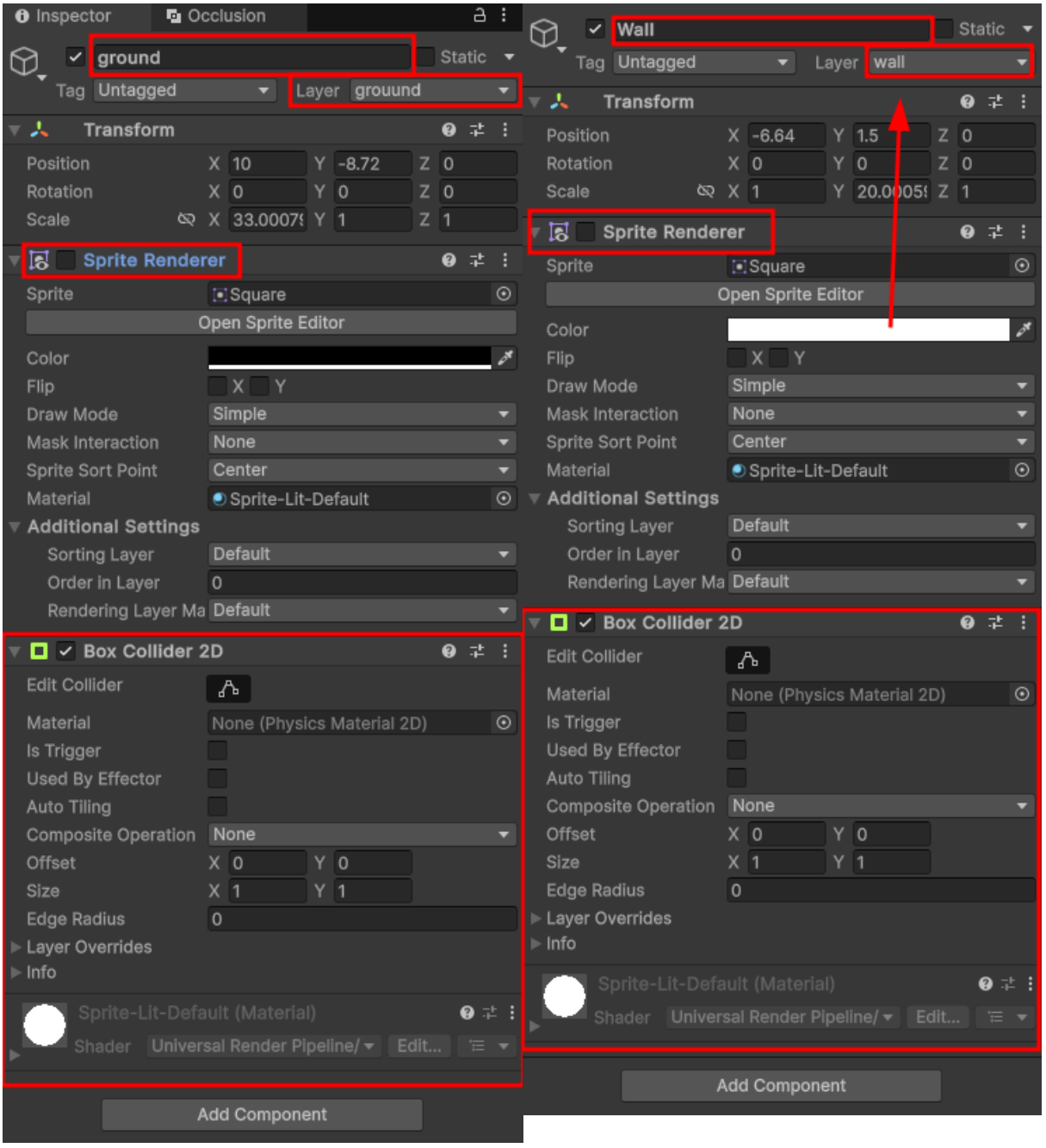
> Podemos usar desde la escena la tecla **W** y mover directamente la posición

6. **Hazlo invisible** para que no se vea en el juego:
   - En el **SpriteRenderer**, cambia el **Color** a transparente (Alpha = 0).
   - O directamente **desactiva el SpriteRenderer** (desmarca el checkbox del componente).
7. Añade un **BoxCollider2D** (Add Component > BoxCollider2D).
   - **NO** actives Is Trigger — El trigger elimina la colisión sólida y el jugador puede traspasar los objetos.
8. Cambia la **Layer** a `ground`.
9. **Duplica** (Ctrl+D) y reposiciona/reescala para cubrir todas las plataformas del mapa.

Crear los colliders de las paredes

1. Dentro de `Layers`, click derecho > **Create Empty** → renombra a `WallLayers`.
2. Repite el mismo proceso: crea sprites cuadrados, escálalos verticalmente para cubrir las paredes del Tilemap, hazlos invisibles, añade BoxCollider2D y asigna Layer `ground`.
3. Duplica y reposiciona hasta cubrir todas las paredes.





¿Por qué la Layer “ground” es importante?

El script `PlayerMovement` lanza un **Raycast** hacia abajo desde el `GroundCheck` para detectar si el jugador está en el suelo. Ese Raycast **solo busca objetos en la Layer** `ground`:

```
RaycastHit2D hit = Physics2D.Raycast(
    groundCheck.transform.position,
```

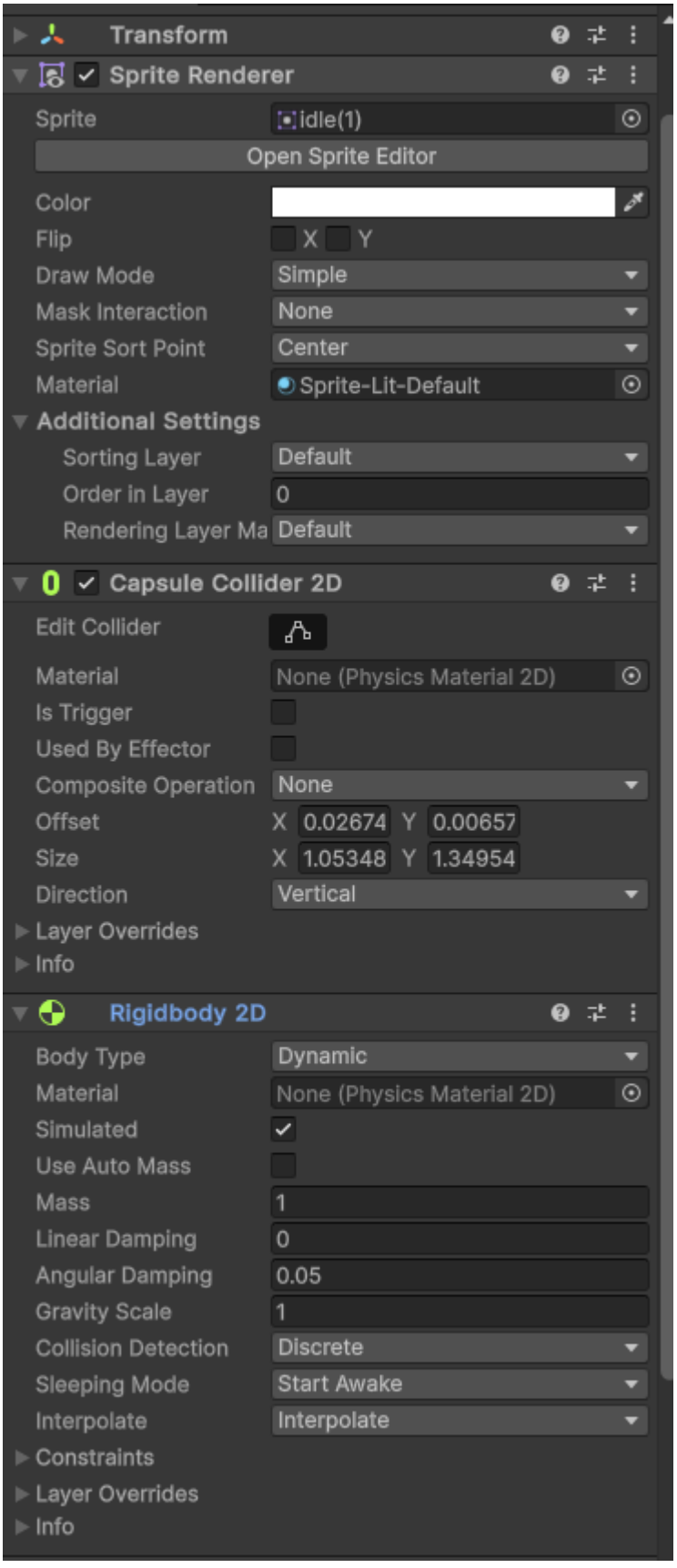
```
    Vector2.down,
    groundDistance,
    whatIsGround  // ← Solo detecta la Layer "ground"
);
isGrounded = hit.collider != null;
```

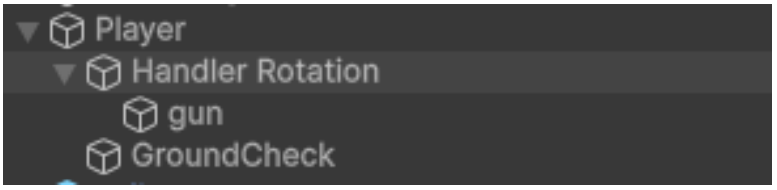
Si los sprites de colisión no tienen la Layer `ground`, el Raycast no los detectará y el jugador **nunca podrá saltar** porque `isGrounded` siempre será `false`.

4.4 Crear el jugador (Player)

1. En la Hierarchy, click derecho > **2D Object > Sprites > Square** (o arrastra tu sprite).
2. Renombra el GameObject a `Player`.
3. Asigna el **Tag**: `Player`.
4. Añade los siguientes componentes desde el botón **Add Component** en el Inspector:
   - **CapsuleCollider2D**: ajusta el tamaño al sprite del jugador.
   - **Rigidbody2D**: configura:
     - Body Type: `Dynamic`
     - Mass: `1`
     - Gravity Scale: `1`
     - Collision Detection: `Discrete`
     - Interpolate: `Interpolate`
   - **PlayerMovement** (nuestro script): arrastra el script o búscalo en Add Component.
5. **Crear el GroundCheck**:
   - Click derecho en `Player` > **Create Empty**.
   - Renombra a `GroundCheck`.
   - Posición: debajo de los pies del jugador (ajusta la Y en negativo, por ejemplo `Y = -0.5`).
6. **Crear el Handler Rotation**:

- Click derecho en `Player` > **Create Empty**.
- Renombra a `Handler Rotation`.
- Posición: `(0, 0, 0)` relativa al Player.
- (Opcional) Añade un sprite hijo que represente el arma (el rectángulo negro).


Transform
Sprite Renderer
Sprite
idle(1)
Open Sprite Editor
Color
Flip
X
Y
Draw Mode
Simple
Mask Interaction
None
Sprite Sort Point
Center
Material
Sprite-Lit-Default
Additional Settings
Sorting Layer
Default
Order in Layer
0
Rendering Layer Ma
Default
Capsule Collider 2D
Edit Collider
Material
None (Physics Material 2D)
Is Trigger
Used By Effector
Composite Operation
None
Offset
X 0.02674 Y 0.00657
Size
X 1.05348 Y 1.34954
Direction
Vertical
Layer Overrides
Info
Rigidbody 2D
Body Type
Dynamic
Material
None (Physics Material 2D)
Simulated
Use Auto Mass
Mass
1
Linear Damping
0
Angular Damping
0.05
Gravity Scale
1
Collision Detection
Discrete
Sleeping Mode
Start Awake
Interpolate
Interpolate
Constraints
Layer Overrides
Info



Código: PlayerMovement.cs

Crea un nuevo script en la carpeta `Scripts` llamado `PlayerMovement.cs` y pega el siguiente código:

```
using UnityEngine;

///<summary>
/// Controla el movimiento del jugador: caminar, saltar y a
puntar/disparar el arma de retroceso.
/// Requiere un Rigidbody2D en el mismo GameObject.
///</summary>
public class PlayerMovement : MonoBehaviour
{
    // --- Componentes ---
    private Rigidbody2D rb;

    [Header("Materiales de física")]
    [Tooltip("Material sin fricción para evitar que el juga
dor se pegue a las paredes en el aire")]
    [SerializeField] private PhysicsMaterial2D noFriction;
    [Tooltip("Material con fricción normal para cuando el j
ugador está en el suelo")]
    [SerializeField] private PhysicsMaterial2D defaultFrict
ion;

    [Header("Estadísticas del jugador")]
    [Tooltip("Velocidad horizontal del jugador")]
    [SerializeField] private float speed = 5f;

    [Header("Salto")]
    [Tooltip("Fuerza del salto aplicada como impulso")]
```

```
    [SerializeField] private float jumpForce = 5f;
    [Tooltip("Capa(s) que el Raycast considera como suel
o")]
    [SerializeField] private LayerMask whatIsGround;
    [Tooltip("Distancia del Raycast hacia abajo para detect
ar el suelo")]
    [SerializeField] private float groundDistance = 0.2f;
    [Tooltip("Objeto hijo desde donde se lanza el Raycast h
acia abajo")]
    [SerializeField] private GameObject groundCheck;
    [Tooltip("Multiplicador de fuerza del retroceso del arm
a en el aire")]
    [SerializeField] private float airMultiplier = 4f;
    [SerializeField] private bool isGrounded;

    [Header("Arma de retroceso")]
    [Tooltip("Transform hijo que rota apuntando hacia el ra
tón (pivote del arma)")]
    [SerializeField] private Transform handleRotation;
    [Tooltip("Munición máxima del arma (se recarga al tocar
el suelo)")]
    [SerializeField] private int gunAmmo = 3;
    private int currentGunAmmo;

    ///<summary>
    /// Awake se ejecuta antes que Start. Obtenemos el comp
onente Rigidbody2D.
    ///</summary>
    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        currentGunAmmo = gunAmmo;
    }

    ///<summary>
    /// Update se ejecuta cada frame. Desde aquí llamamos a
toda la lógica del jugador.
    ///</summary>
```

```
    private void Update()
    {
        // Si el juego ha terminado, no hacer nada
        if (GameManager.Instance != null && GameManager.Ins
tance.IsGameOver())
            return;

        // Cambiamos el material de física según si estamos
en el suelo o en el aire.
        // Esto evita que el jugador se "pegue" a las pared
es al saltar.
        rb.sharedMaterial = isGrounded ? defaultFriction :
noFriction;

        HandleMovement();
        HandleLook();
        Jump();
    }

    ///<summary>
    /// Mueve al jugador horizontalmente usando el eje "Hor
izontal" (A/D o flechas).
    /// La velocidad vertical del Rigidbody se mantiene int
acta para no afectar saltos/caídas.
    ///</summary>
    private void HandleMovement()
    {
        float hor = Input.GetAxis("Horizontal");

        // Si no hay input horizontal, no modificamos la ve
locidad
        if (hor == 0f)
            return;

        rb.linearVelocity = new Vector2(hor * speed, rb.lin
earVelocity.y);
    }
```

```
    ///<summary>
    /// Gestiona el salto del jugador. Usa un Raycast hacia
abajo desde el groundCheck
    /// para detectar si el jugador está tocando el suelo.
    ///</summary>
    private void Jump()
    {
        // Solo podemos saltar si estamos en el suelo y pul
samos el botón de salto
        if (Input.GetButtonDown("Jump") && isGrounded)
        {
            rb.AddForce(Vector2.up * jumpForce, ForceMode2
D.Impulse);
        }

        // Raycast: lanza un rayo invisible hacia abajo par
a comprobar si hay suelo
        RaycastHit2D hit = Physics2D.Raycast(
            groundCheck.transform.position,  // Origen del
rayo
            Vector2.down,                    // Dirección:
hacia abajo
            groundDistance,                  // Distancia
máxima del rayo
            whatIsGround                     // Solo detec
ta objetos en esta capa
        );

        isGrounded = hit.collider != null;
    }

    ///<summary>
    /// Rota el pivote del arma hacia la posición del ratón
y gestiona el disparo.
    /// El "disparo" no lanza proyectiles: aplica una fuerz
a de retroceso al jugador
    /// en la dirección contraria a donde apunta, permitien
do impulsarse por el aire.
```

```
    ///</summary>
    private void HandleLook()
    {
        // Convertimos la posición del ratón en pantalla a
coordenadas del mundo
        Vector3 mousePos = Camera.main.ScreenToWorldPoint(I
nput.mousePosition);
        Vector2 direction = (mousePos - transform.positio
n).normalized;

        // Calculamos el ángulo en grados para rotar el piv
ote del arma
        float angle = Mathf.Atan2(direction.y, direction.x)
* Mathf.Rad2Deg;
        handleRotation.rotation = Quaternion.Euler(0f, 0f,
angle);

        // Al hacer clic izquierdo, si tenemos munición, ap
licamos retroceso
        if (Input.GetKeyDown(KeyCode.Mouse0) && currentGunA
mmo > 0)
        {
            // La fuerza se aplica en dirección CONTRARIA a
donde apunta el arma
            rb.AddForce(direction * (-jumpForce * airMultip
lier), ForceMode2D.Impulse);
            currentGunAmmo--;
        }

        // La munición se recarga automáticamente al tocar
el suelo
        if (isGrounded)
        {
            currentGunAmmo = gunAmmo;
        }
    }

    ///<summary>
```

```
    /// Método llamado cuando el jugador colisiona con un e
nemigo o trampa.
    /// Se activa con triggers (Collider2D marcado como "Is
Trigger").
    ///</summary>
    private void OnTriggerEnter2D(Collider2D collision)
    {
        // Si tocamos un objeto con tag "Enemy" o "Spike",
morimos
        if (collision.CompareTag("Enemy") || collision.Comp
areTag("Spike"))
        {
            Die();
        }
    }

    ///<summary>
    /// Gestiona la muerte del jugador: desactiva el objeto
y llama al Game Over.
    ///</summary>
    private void Die()
    {
        // Notificamos al GameManager que el juego ha termi
nado
        if (GameManager.Instance != null)
        {
            GameManager.Instance.GameOver();
        }

        // Desactivamos el sprite del jugador (desaparece d
e la pantalla)
        gameObject.SetActive(false);
    }

    ///<summary>
    /// Dibuja el rayo del groundCheck en la vista Scene (s
olo visible en el editor,
    /// no aparece en el juego compilado). Útil para depura
```

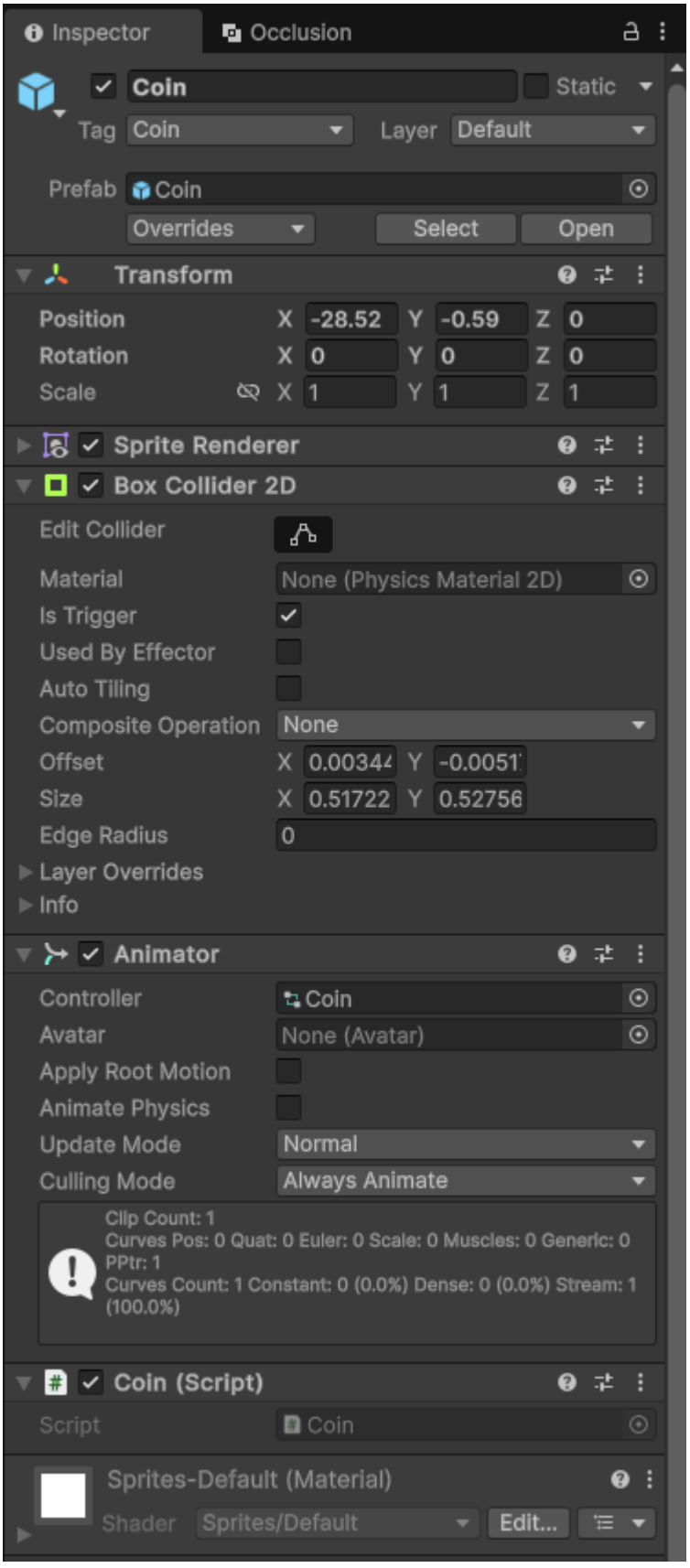
```
r.
    ///</summary>
    private void OnDrawGizmos()
    {
        if (groundCheck == null) return;

        Gizmos.color = Color.red;
        Gizmos.DrawLine(
            groundCheck.transform.position,
            groundCheck.transform.position + Vector3.down *
groundDistance
        );
    }
}
```

1. **Configurar las variables del script PlayerMovement** en el Inspector:
    - **No Friction**: arrastra el material `NoStuckWall` desde la carpeta `phisicsMaterials`.
    - **Default Friction**: arrastra el material `DefaultFriction`.
    - **Speed**: `5`
    - **Jump Force**: `5`
    - **What Is Ground**: selecciona la layer `ground`.
    - **Ground Distance**: `0.2`
    - **Ground Check**: arrastra el GameObject `GroundCheck`.
    - **Air Multiplier**: `4`
    - **Handle Rotation**: arrastra el GameObject `Handler Rotation`.
    - **Gun Ammo**: `3`

4.5 Crear las monedas (Coins)

1. Crea un GameObject vacío en la Hierarchy, renómbralo a `Coins` (contenedor).

2. Dentro de `Coins`, click derecho > **2D Object > Sprites > Circle** (o usa tu sprite de moneda).
3. Renombra a `Coin`.
4. Añade componentes:
    - **CircleCollider2D** (o el que se adapte a tu sprite): activa **Is Trigger** ✅.
    - **Coin** (nuestro script).
    - (Opcional) **Animator** si tienes animación de la moneda girando.
5. Duplica la moneda (Ctrl+D) y colócala en diferentes posiciones del mapa.
6. (Opcional) Convierte la moneda en **Prefab**: arrastra el GameObject desde Hierarchy a la carpeta Assets. Así podrás reutilizarla fácilmente.



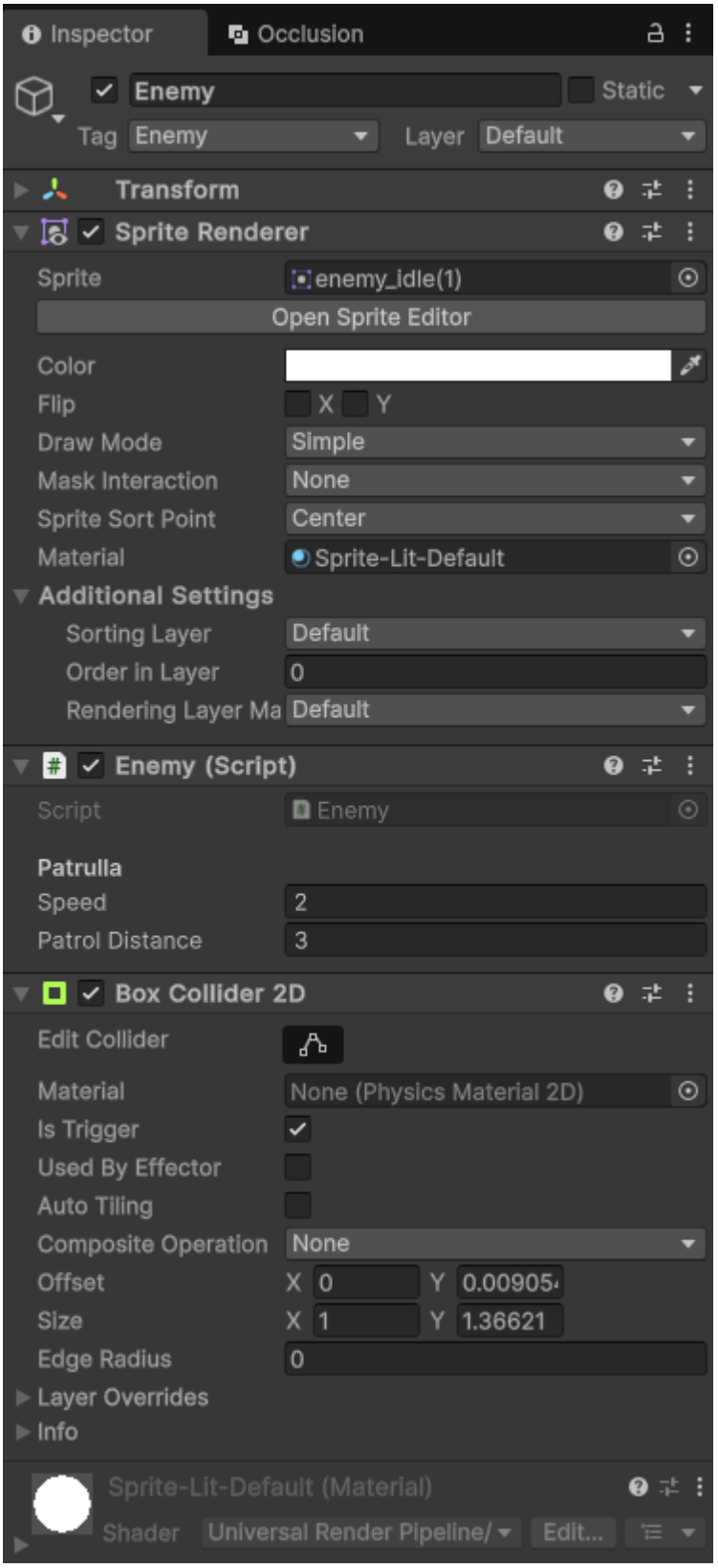
Código: Coin.cs

Crea un nuevo script en la carpeta `Scripts` llamado `Coin.cs` y pega el siguiente código:

```
using UnityEngine;

///<summary>
/// Controla el comportamiento de una moneda coleccionable.
/// Cuando el jugador la toca, suma un punto y se destruye.
/// Requiere un Collider2D marcado como "Is Trigger" en el
mismo GameObject.
///</summary>
public class Coin : MonoBehaviour
{
    ///<summary>
    /// OnTriggerEnter2D se ejecuta automáticamente cuando
otro Collider2D
    /// entra en contacto con el trigger de este objeto.
    ///</summary>
    ///<param name="collision">El Collider2D del objeto que
ha entrado en el trigger.</param>
    private void OnTriggerEnter2D(Collider2D collision)
    {
        // Comprobamos si el objeto que nos tocó tiene el t
ag "Player"
        if (collision.CompareTag("Player"))
        {
            // Sumamos un punto a través del GameManager
            GameManager.Instance.AddPoint();

            // Destruimos esta moneda (desaparece de la esc
ena)
            Destroy(gameObject);
        }
    }
}
```

4.6 Crear los enemigos (Enemies)

1. Crea un GameObject vacío llamado `Enemies` (contenedor).
2. Dentro, crea un sprite para el fantasma o usa tu asset.
3. Renómbralo a `Enemy`.
4. Asigna el **Tag**: `Enemy`.
5. Añade componentes:
    - **Collider2D** (BoxCollider2D o CircleCollider2D): activa **Is Trigger** ✅.
    - **Enemy** (nuestro script nuevo).
6. Configura las variables del script Enemy:
    - **Speed**: `2` (velocidad de patrulla).
    - **Patrol Distance**: `3` (distancia de ida y vuelta).
7. Duplica el enemigo para tener 6 en total (Enemy, Enemy (1)... Enemy (5)).
8. Coloca cada uno en la posición deseada del mapa.



Código: Enemy.cs

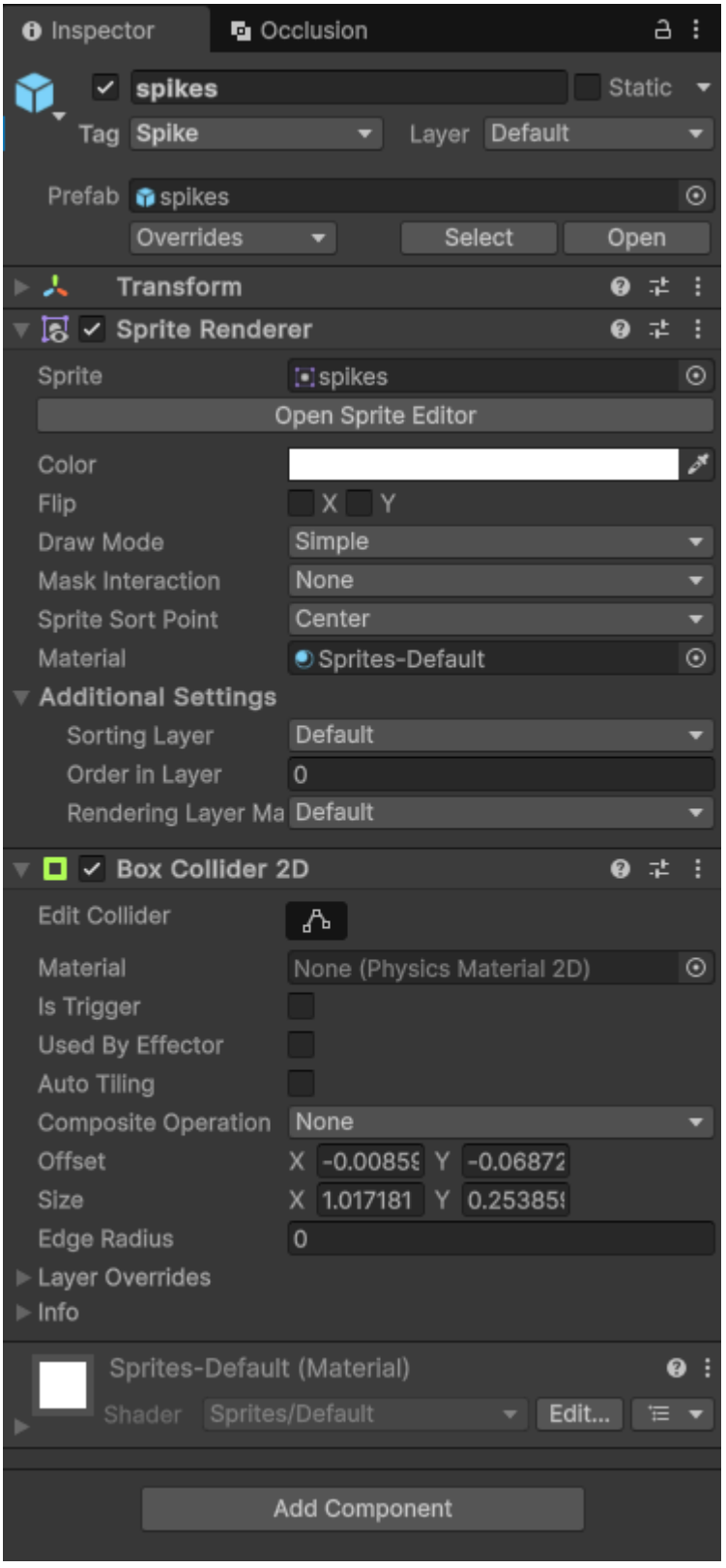
Crea un nuevo script en la carpeta `Scripts` llamado `Enemy.cs` y pega el siguiente código:

```
using UnityEngine;

///<summary>
/// Comportamiento básico de un enemigo (fantasma).
/// Se mueve en patrulla horizontal entre dos puntos.
/// El jugador muere al tocarlo (gestionado desde PlayerMov
ement).
/// Requiere un Collider2D marcado como "Is Trigger" y el t
ag "Enemy".
///</summary>
public class Enemy : MonoBehaviour
{
    [Header("Patrulla")]
    [Tooltip("Velocidad de movimiento del enemigo")]
    [SerializeField] private float speed = 2f;
    [Tooltip("Distancia que recorre en cada dirección desde
su posición inicial")]
    [SerializeField] private float patrolDistance = 3f;

    // Posición inicial del enemigo (se guarda al empezar)
    private Vector3 startPosition;
    // Dirección actual del movimiento (1 = derecha, -1 = i
zquierda)
    private int direction = 1;

    ///<summary>
    /// Start se ejecuta al inicio. Guardamos la posición d
onde se colocó el enemigo.
    ///</summary>
    private void Start()
    {
        startPosition = transform.position;
    }

    ///<summary>
    /// Update se ejecuta cada frame. Movemos al enemigo de
lado a lado.
    ///</summary>
```

```
    private void Update()
    {
        // Si el juego terminó, el enemigo se detiene
        if (GameManager.Instance != null && GameManager.Ins
tance.IsGameOver())
            return;

        // Movemos al enemigo en la dirección actual
        transform.Translate(Vector2.right * (direction * sp
eed * Time.deltaTime));

        // Si se aleja demasiado de su punto de inicio, cam
bia de dirección
        if (Vector2.Distance(startPosition, transform.posit
ion) >= patrolDistance)
        {
            direction *= -1; // Invertimos la dirección

            // Volteamos el sprite para que mire hacia dond
e camina
            Vector3 scale = transform.localScale;
            scale.x *= -1;
            transform.localScale = scale;
        }
    }
}
```

4.7 Crear los pinchos (Spikes)

1. Crea un GameObject con tu sprite/asset de pinchos.
2. Renómbralo a `Spikes`.
3. Asigna el **Tag**: `Spike`.
4. Añade un **Collider2D** (normalmente BoxCollider2D): activa **Is Trigger** ✅.
5. Colócalos en las posiciones peligrosas del mapa.

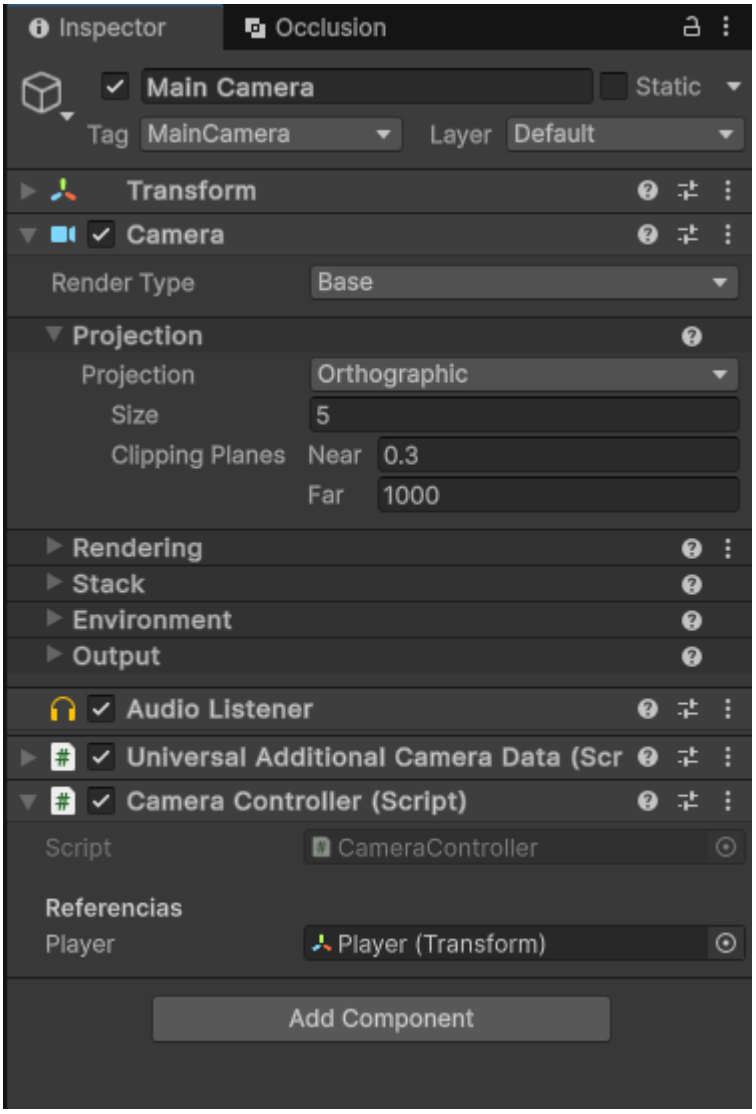


Los pinchos **no necesitan script propio**. La detección de muerte se gestiona desde `PlayerMovement.OnTriggerEnter2D()`, que comprueba si el objeto tiene el tag `Spike`:

```
// Fragmento de PlayerMovement.cs
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Enemy") || collision.CompareT
ag("Spike"))
    {
        Die();
    }
}
```

4.8 Configurar la cámara

1. Selecciona **Main Camera** en la Hierarchy.
2. Añade el componente **CameraController** (nuestro script).
3. En el Inspector, arrastra el **Player** al campo `Player` del script.
4. Asegúrate de que la posición Z de la cámara es `10` (para ver la escena 2D).



Código: CameraController.cs

Crea un nuevo script en la carpeta `Scripts` llamado `CameraController.cs` y pega el siguiente código:

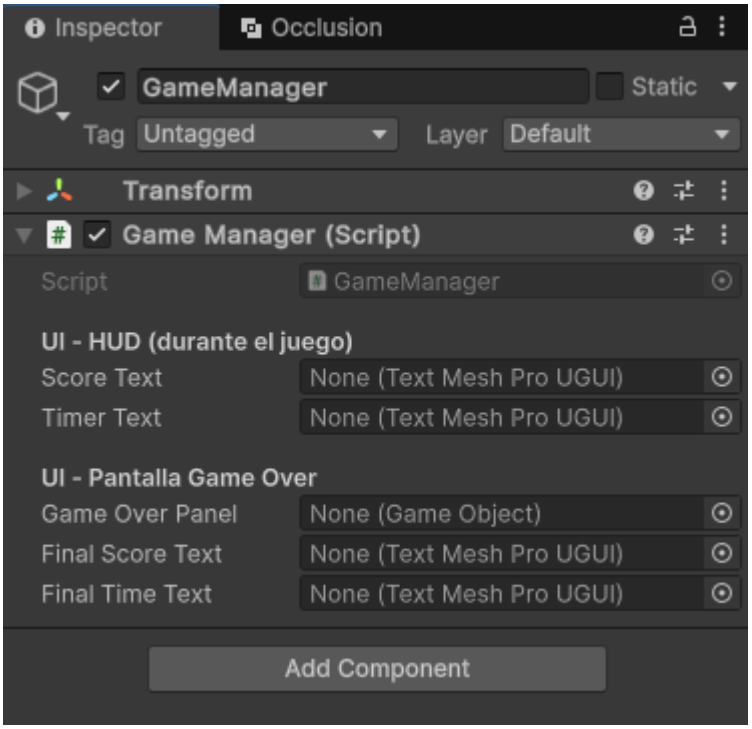
```
using UnityEngine;

///<summary>
/// Controla la cámara para que siga al jugador con un lige
ro desplazamiento
/// hacia la dirección del ratón, dando una sensación de "m
irar adelante".
/// Se asigna al GameObject de la Main Camera.
///</summary>
public class CameraController : MonoBehaviour
```

```
{
    [Header("Referencias")]
    [Tooltip("Transform del jugador al que la cámara debe s
eguir")]
    public Transform player;

    // Referencia cacheada a la cámara principal
    private Camera _camera;

    ///<summary>
    /// Start se ejecuta una vez al inicio. Guardamos la re
ferencia a la cámara.
    ///</summary>
    void Start()
    {
        _camera = Camera.main;
    }

    ///<summary>
    /// Update se ejecuta cada frame. Calculamos la posició
n objetivo de la cámara
    /// y la movemos suavemente con Lerp.
    ///</summary>
    void Update()
    {
        // Si el jugador ha sido destruido/desactivado, no
hacer nada
        if (player == null) return;

        // Si el juego ha terminado, dejamos la cámara quie
ta
        if (GameManager.Instance != null && GameManager.Ins
tance.IsGameOver())
            return;

        // Convertimos la posición del ratón en pantalla a
posición en el mundo
        Vector3 mousePos = _camera.ScreenToWorldPoint(Inpu
```

```
t.mousePosition);

        // Calculamos un offset (desplazamiento) normalizad
o desde el jugador hacia el ratón.
        // Esto hace que la cámara se adelante ligeramente
en la dirección
        // hacia donde apunta el ratón, permitiendo ver mej
or lo que hay delante.
        Vector3 offset = (mousePos - player.position).norma
lized;

        // La posición objetivo es el jugador + el offset
(Z=-10 para que la cámara vea la escena)
        Vector3 targetPos = new Vector3(
            player.position.x + offset.x,
            player.position.y + offset.y,
            -10f
        );

        // Lerp: movimiento suave. 0.1f controla la velocid
ad (más bajo = más suave)
        _camera.transform.position = Vector3.Lerp(
            _camera.transform.position,
            targetPos,
            0.1f
        );
    }
}
```

4.9 Crear el GameManager

1. Click derecho en la Hierarchy > **Create Empty**.
2. Renombra a `GameManager`.
3. Añade el componente **GameManager** (nuestro script).
4. **NO asignes las variables de UI todavía** — primero hay que crear el Canvas (paso siguiente).



Código: GameManager.cs

Crea un nuevo script en la carpeta `Scripts` llamado `GameManager.cs` y pega el siguiente código:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;

///<summary>
/// GameManager es un Singleton que controla el estado gene
ral del juego:
/// puntuación, tiempo, estados de juego (jugando / game ov
er) y la UI.
///</summary>
public class GameManager : MonoBehaviour
{
    // --- Singleton ---
    // El patrón Singleton asegura que solo exista UNA inst
ancia de GameManager
    // en toda la escena, accesible desde cualquier script
con GameManager.Instance
    public static GameManager Instance;
```

```
    // --- Estado del juego ---
    private int score = 0;            // Monedas recogidas
    private float elapsedTime = 0f;   // Tiempo transcurrid
o en segundos
    private bool isGameOver = false;  // ¿Ha terminado la p
artida?

    // --- Referencias UI (se asignan desde el Inspector) -
--
    [Header("UI - HUD (durante el juego)")]
    [SerializeField] private TextMeshProUGUI scoreText;
// Texto de monedas en pantalla
    [SerializeField] private TextMeshProUGUI timerText;
// Texto del cronómetro en pantalla

    [Header("UI - Pantalla Game Over")]
    [SerializeField] private GameObject gameOverPanel;
// Panel que se muestra al morir
    [SerializeField] private TextMeshProUGUI finalScoreTex
t;  // Puntuación final
    [SerializeField] private TextMeshProUGUI finalTimeText;
// Tiempo final

    ///<summary>
    /// Awake se ejecuta antes que Start. Aquí configuramos
el Singleton.
    ///</summary>
    private void Awake()
    {
        // Si ya existe una instancia, destruimos este dupl
icado
        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }
        Instance = this;
```

```
    }

    ///<summary>
    /// Start se ejecuta una vez al inicio. Inicializamos l
a UI.
    ///</summary>
    private void Start()
    {
        // Nos aseguramos de que el juego esté en marcha
        Time.timeScale = 1f;
        isGameOver = false;

        // Ocultamos el panel de Game Over al empezar
        if (gameOverPanel != null)
            gameOverPanel.SetActive(false);

        // Actualizamos la UI con los valores iniciales
        UpdateScoreUI();
        UpdateTimerUI();
    }

    ///<summary>
    /// Update se ejecuta cada frame. Aquí actualizamos el
cronómetro.
    ///</summary>
    private void Update()
    {
        // Solo contamos el tiempo si el juego NO ha termin
ado
        if (!isGameOver)
        {
            elapsedTime += Time.deltaTime;
            UpdateTimerUI();
        }
    }

    ///<summary>
    /// Añade un punto a la puntuación. Llamado por las mon
```

```
edas al recogerlas.
    ///</summary>
    public void AddPoint()
    {
        if (isGameOver) return; // No sumar puntos si el ju
ego terminó

        score++;
        UpdateScoreUI();
    }

    ///<summary>
    /// Activa el estado de Game Over. Llamado cuando el ju
gador muere.
    ///</summary>
    public void GameOver()
    {
        if (isGameOver) return; // Evitar llamar dos veces

        isGameOver = true;

        // Congelamos el juego (todo se pausa: física, anim
aciones, etc.)
        Time.timeScale = 0f;

        // Mostramos la pantalla de Game Over con los datos
finales
        if (gameOverPanel != null)
        {
            gameOverPanel.SetActive(true);

            if (finalScoreText != null)
                finalScoreText.text = "Monedas: " + score;

            if (finalTimeText != null)
                finalTimeText.text = "Tiempo: " + FormatTim
e(elapsedTime);
        }
```

```
    }

    ///<summary>
    /// Reinicia la escena actual. Se conecta al botón "Rei
ntentar".
    ///</summary>
    public void RestartGame()
    {
        // Restauramos el tiempo antes de recargar (importa
nte porque lo pusimos a 0)
        Time.timeScale = 1f;
        SceneManager.LoadScene(SceneManager.GetActiveScene
().buildIndex);
    }

    ///<summary>
    /// Devuelve si el juego ha terminado. Útil para que ot
ros scripts
    /// dejen de funcionar cuando el jugador muere.
    ///</summary>
    public bool IsGameOver()
    {
        return isGameOver;
    }

    // --- Métodos privados de UI ---

    ///<summary>
    /// Actualiza el texto de puntuación en el HUD.
    ///</summary>
    private void UpdateScoreUI()
    {
        if (scoreText != null)
            scoreText.text = "Monedas: " + score;
    }

    ///<summary>
    /// Actualiza el texto del cronómetro en el HUD.
```

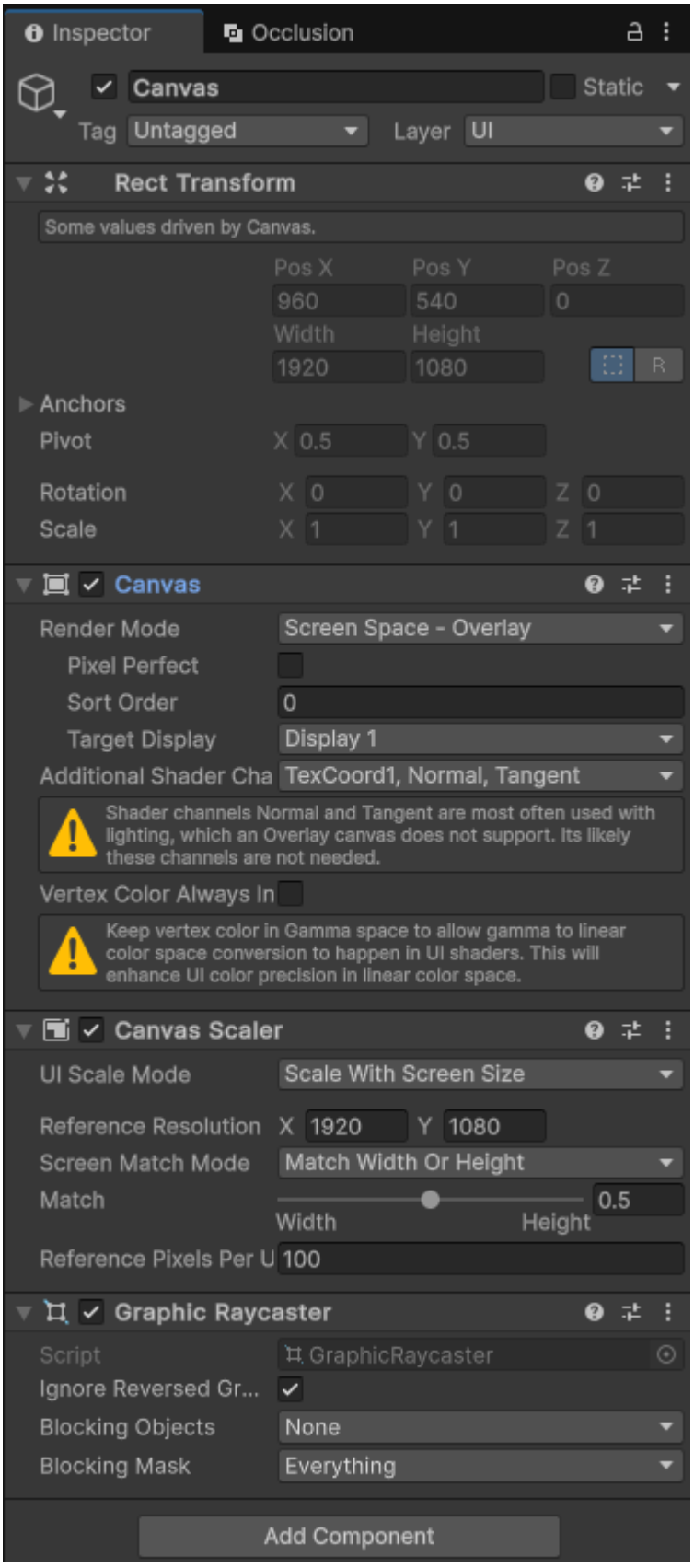
```
    ///</summary>
    private void UpdateTimerUI()
    {
        if (timerText != null)
            timerText.text = FormatTime(elapsedTime);
    }

    ///<summary>
    /// Convierte segundos a formato "MM:SS" legible.
    /// Ejemplo: 125.7 segundos → "02:05"
    ///</summary>
    private string FormatTime(float timeInSeconds)
    {
        int minutes = Mathf.FloorToInt(timeInSeconds / 60
f);
        int seconds = Mathf.FloorToInt(timeInSeconds % 60
f);
        return string.Format("{0:00}:{1:00}", minutes, seco
nds);
    }
}
```

4.10 Crear la interfaz de usuario (UI)

Esta es la parte más larga. Sigue cada paso con cuidado.

4.10.1 Crear el Canvas

1. Click derecho en la Hierarchy > **UI > Canvas**.
2. Selecciona el Canvas y en el Inspector:
    - **Canvas Scaler**:
        - UI Scale Mode: `Scale With Screen Size`
        - Reference Resolution: `1920 x 1080`
        - Screen Match Mode: `Match Width Or Height`
        - Match: `0.5`



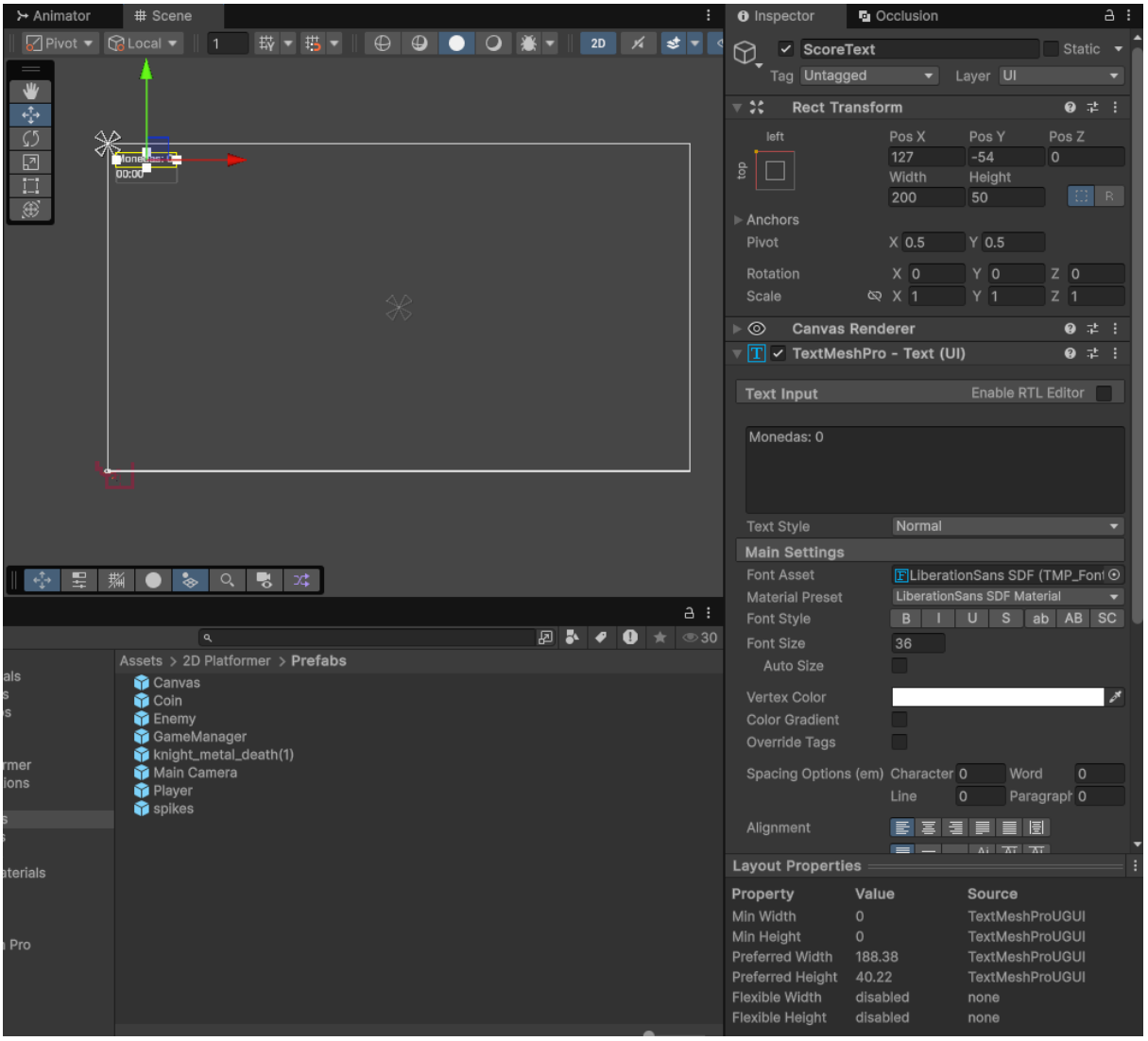
4.10.2 Crear el HUD (puntuación y tiempo en pantalla)

1. Click derecho en el Canvas > **Create Empty** → renombra a `HUD`.

2. Selecciona `HUD` y en el **Rect Transform**:
    - Anchor Preset: **Stretch - Stretch** (mantén Alt pulsado para también ajustar posición).
    - Left, Top, Right, Bottom: todo a `0`.

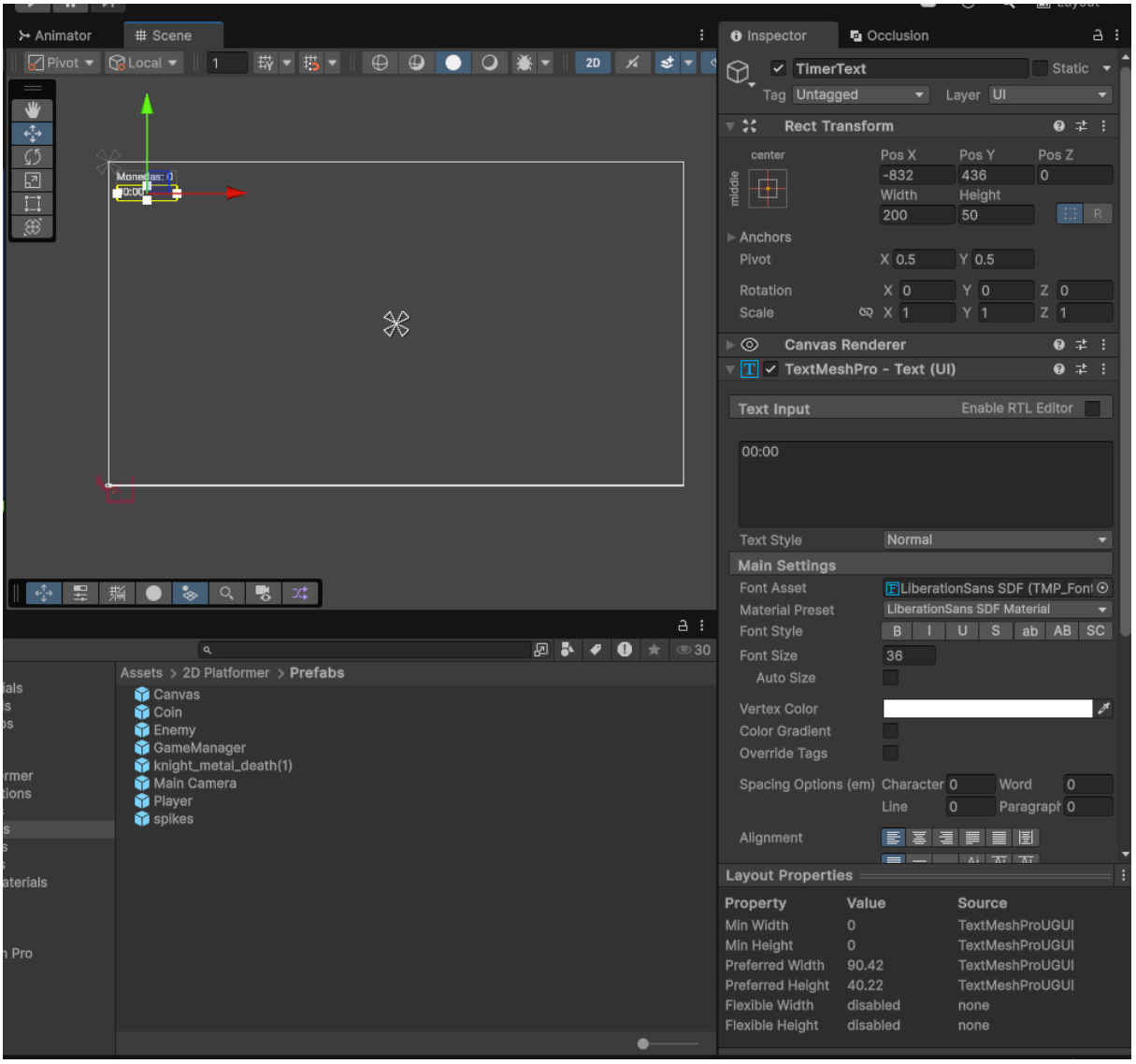
Texto de puntuación (ScoreText)

1. Click derecho en `HUD` > **UI > Text - TextMeshPro** → renombra a `ScoreText`.
    - Si es la primera vez que usas TextMeshPro, Unity te pedirá importar los esenciales. Pulsa **Import TMP Essentials**.
2. Configura el Rect Transform:
    - Anchor Preset: **Top-Left**.
    - Pos X: `30`, Pos Y: `30`.
    - Width: `300`, Height: `50`.
3. En el componente TextMeshPro:
    - Text: `Monedas: 0`
    - Font Size: `36`
    - Color: blanco
    - Alignment: izquierda



Texto del cronómetro (TimerText)

1. Click derecho en `HUD` > **UI > Text - TextMeshPro** → renombra a `TimerText`.
2. Configura el Rect Transform:
   - Anchor Preset: **Top-Right**.
   - Pos X: `30`, Pos Y: `30`.
   - Width: `200`, Height: `50`.
3. En el componente TextMeshPro:
   - Text: `00:00`
   - Font Size: `36`
   - Color: blanco
   - Alignment: derecha



4.10.3 Crear la pantalla de Game Over

Panel oscuro de fondo

1. Click derecho en el Canvas > **UI > Panel** → renombra a `GameOverPanel`.
2. En el componente **Image**:
   - Color: negro con transparencia → RGBA: `(0, 0, 0, 200)` (el 200 sobre 255 da un negro semitransparente).
3. Rect Transform: Anchor Preset **Stretch - Stretch**, todo a `0`.

Título “GAME OVER”

1. Click derecho en `GameOverPanel` > **UI > Text - TextMeshPro** → renombra a `GameOverTitle`.

2. Rect Transform:
    - Anchor Preset: **Middle-Center**.
    - Pos Y: `120`.
    - Width: `600`, Height: `100`.
3. TextMeshPro:
    - Text: `GAME OVER`
    - Font Size: `72`
    - Alignment: centrado
    - Color: rojo (`#FF0000`) o blanco, como prefieras.
    - Font Style: **Bold**

Puntuación final

1. Click derecho en `GameOverPanel` > **UI > Text - TextMeshPro** → renombra a `FinalScoreText`.
2. Rect Transform:
    - Anchor Preset: **Middle-Center**.
    - Pos Y: `30`.
    - Width: `400`, Height: `60`.
3. TextMeshPro:
    - Text: `Monedas: 0`
    - Font Size: `48`
    - Alignment: centrado
    - Color: blanco

Tiempo final

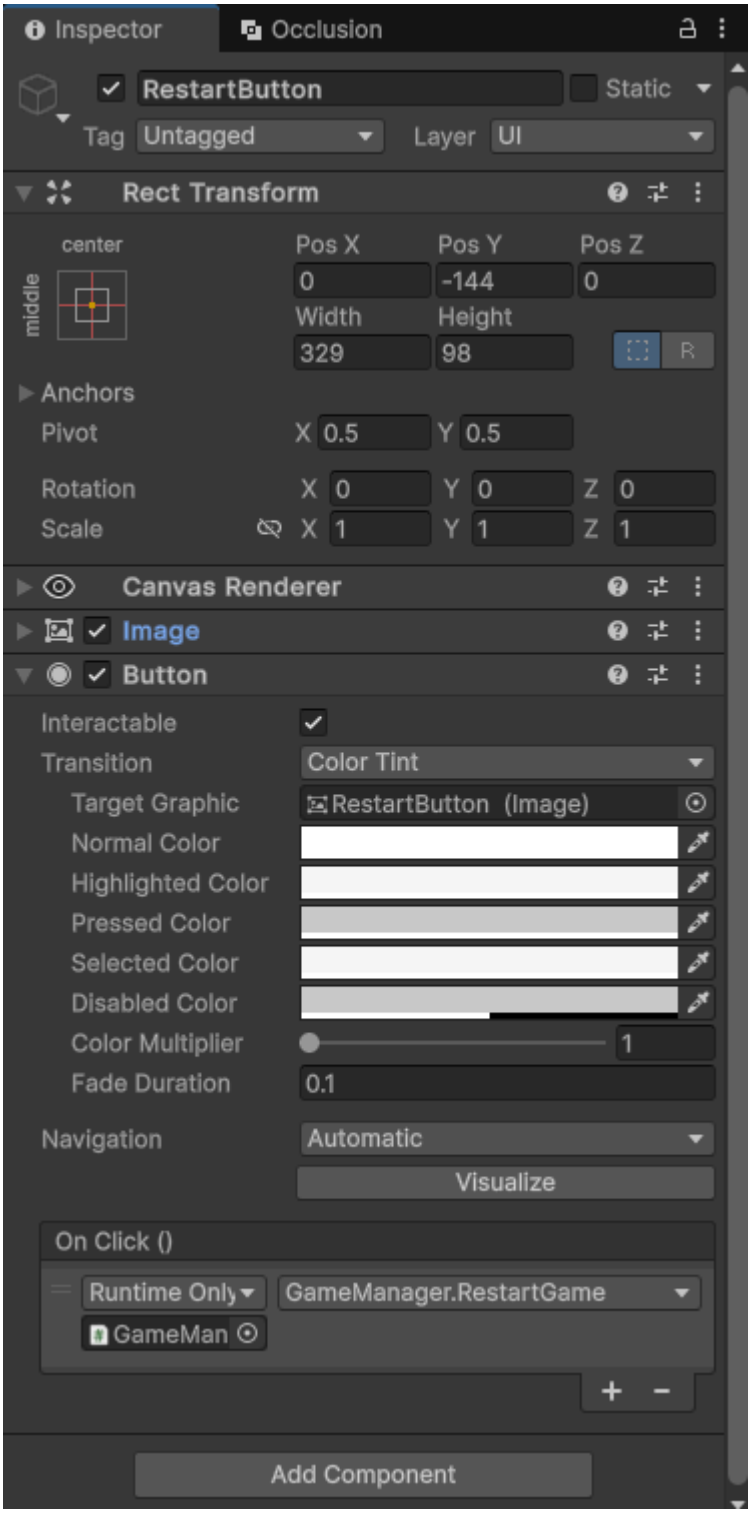
1. Click derecho en `GameOverPanel` > **UI > Text - TextMeshPro** → renombra a `FinalTimeText`.
2. Rect Transform:
    - Anchor Preset: **Middle-Center**.

- Pos Y: `30`.
- Width: `400`, Height: `60`.

3. TextMeshPro:
   - Text: `Tiempo: 00:00`
   - Font Size: `48`
   - Alignment: centrado
   - Color: blanco

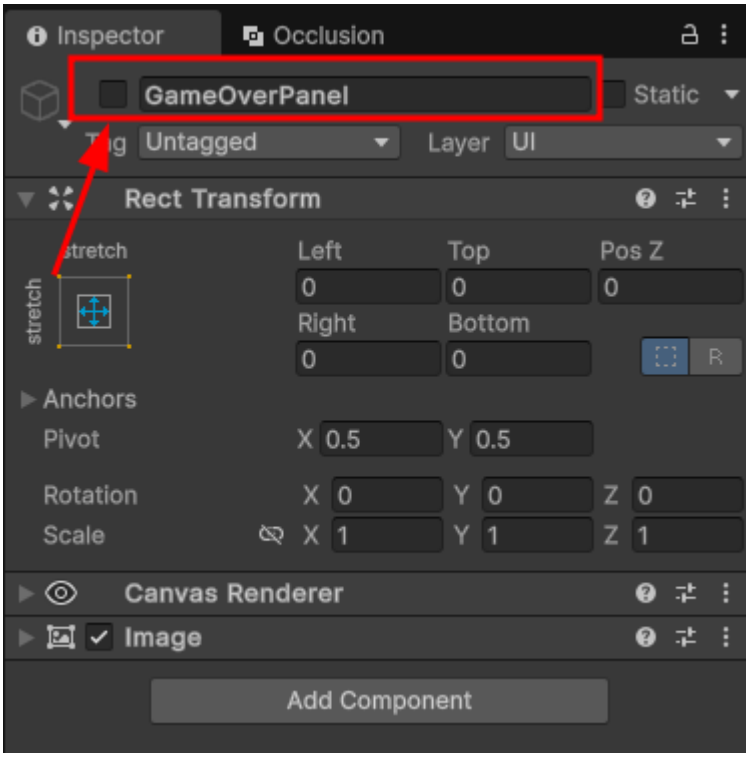
Botón Reintentar

1. Click derecho en `GameOverPanel` > **UI > Button - TextMeshPro** → renombra a `RestartButton`.
2. Rect Transform:
   - Anchor Preset: **Middle-Center**.
   - Pos Y: `120`.
   - Width: `250`, Height: `60`.
3. Selecciona el **texto hijo** del botón (normalmente se llama `Text (TMP)`) y cambia:
   - Text: `Reintentar`
   - Font Size: `32`
4. Selecciona `RestartButton` y en el componente **Button > On Click()**:
   - Pulsa `+` para añadir un nuevo evento.
   - Arrastra el GameObject `GameManager` de la Hierarchy al campo “None (Object)”.
   - En el dropdown de función, selecciona: **GameManager > RestartGame()**.



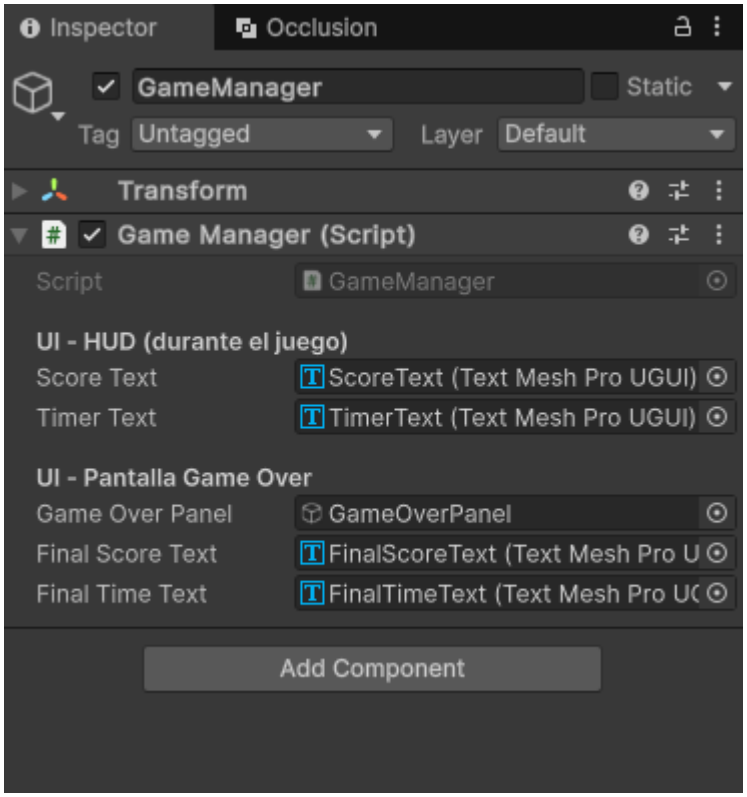
Desactivar el panel

1. **MUY IMPORTANTE**: selecciona `GameOverPanel` y **desmarca el checkbox** al lado de su nombre en el Inspector (esto lo desactiva). El panel se activará automáticamente por código cuando el jugador muera.



4.10.4 Asignar las referencias de UI al GameManager

1. Selecciona el `GameManager` en la Hierarchy.
2. En el Inspector, en el script GameManager, arrastra:
   - `ScoreText` → campo **Score Text**
   - `TimerText` → campo **Timer Text**
   - `GameOverPanel` → campo **Game Over Panel**
   - `FinalScoreText` → campo **Final Score Text**
   - `FinalTimeText` → campo **Final Time Text**



5. Explicación de los scripts

5.1 GameManager.cs

Propósito: controlar el estado global del juego (puntuación, tiempo, game over).

Patrón Singleton: permite que cualquier script acceda al GameManager con `GameManager.Instance`.

Flujo de ejecución:
1. `Awake()` → establece la instancia Singleton.
2. `Start()` → inicializa la UI y desactiva el panel de Game Over.
3. `Update()` → cada frame suma tiempo al cronómetro (si no es Game Over).
4. `AddPoint()` → llamado por las monedas, suma puntos y actualiza la UI.
5. `GameOver()` → congela el juego (`Time.timeScale = 0`), muestra el panel final.
6. `RestartGame()` → recarga la escena (conectado al botón Reintentar).

Métodos auxiliares:
- `UpdateScoreUI()`: actualiza el texto de monedas.
- `UpdateTimerUI()`: actualiza el cronómetro.
- `FormatTime()`: convierte segundos a formato `MM:SS`.
- `IsGameOver()`: devuelve si el juego ha terminado (usado por otros scripts).

5.2 PlayerMovement.cs

Propósito: gestionar todo el movimiento del jugador.

Componentes requeridos: Rigidbody2D, CapsuleCollider2D.

Flujo de ejecución:
1. `Awake()` → obtiene el Rigidbody2D.
2. `Update()` → si no es Game Over, ejecuta:
- Cambia el material de física (con/sin fricción).
- `HandleMovement()`: lee input horizontal y asigna velocidad.
- `HandleLook()`: rota el arma hacia el ratón, dispara retroceso.
- `Jump()`: gestiona salto y detección de suelo (Raycast).
3. `OnTriggerEnter2D()` → detecta colisión con enemigos/pinchos → `Die()`.
4. `Die()` → llama a `GameManager.GameOver()` y desactiva al jugador.

Mecánica del arma de retroceso: al hacer clic izquierdo, se aplica una fuerza en dirección contraria al ratón. Esto permite impulsarse por el aire. La munición se recarga al tocar el suelo.

Materiales de física: se cambian dinámicamente para evitar que el jugador se pegue a las paredes en el aire (fricción 0 en el aire, fricción normal en el suelo).

5.3 Coin.cs

Propósito: detectar cuándo el jugador toca una moneda.

Requiere: Collider2D con **Is Trigger** activado.

Funcionamiento: cuando el jugador (identificado por el tag `Player`) entra en el trigger de la moneda, se suma un punto al GameManager y la moneda se destruye con `Destroy(gameObject)`.

5.4 CameraController.cs

Propósito: la cámara sigue al jugador con un desplazamiento (offset) suave hacia la dirección del ratón.

Funcionamiento:
- Calcula un offset normalizado desde el jugador hacia la posición del ratón.
- La posición objetivo de la cámara es el jugador + ese offset.
- Usa `Vector3.Lerp()` para moverse suavemente (interpolación lineal).
- Se detiene si el juego termina o el jugador es destruido.

5.5 Enemy.cs

Propósito: enemigo que patrulla horizontalmente entre dos puntos.

Requiere: Collider2D con **Is Trigger** activado, Tag `Enemy`.

Funcionamiento:
- Guarda su posición inicial al empezar.
- Se mueve en una dirección con `transform.Translate()`.
- Cuando se aleja más de `patrolDistance` de su punto de inicio, invierte la dirección.
- Voltea el sprite (`localScale.x *= -1`) para que mire hacia donde camina.
- Se detiene si el juego termina.

La muerte del jugador al tocar al enemigo NO está en este script — está en `PlayerMovement.OnTriggerEnter2D()`.