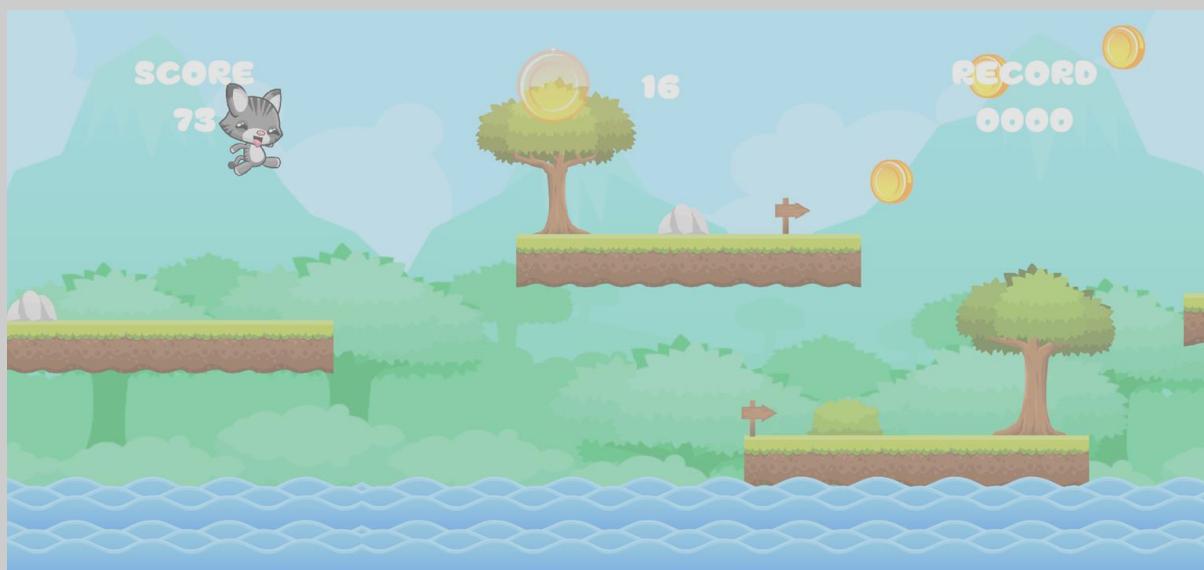


DESARROLLO DE UN JUEGO 2D SENCILLO EN UNITY

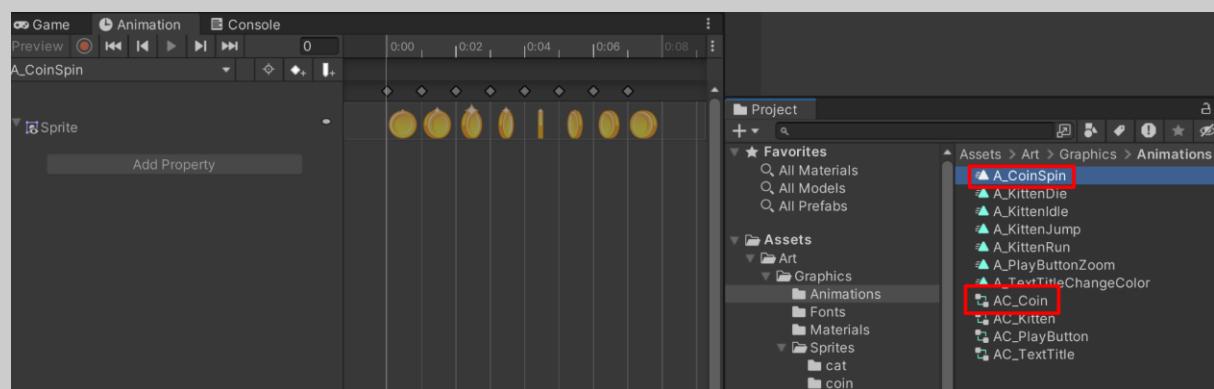
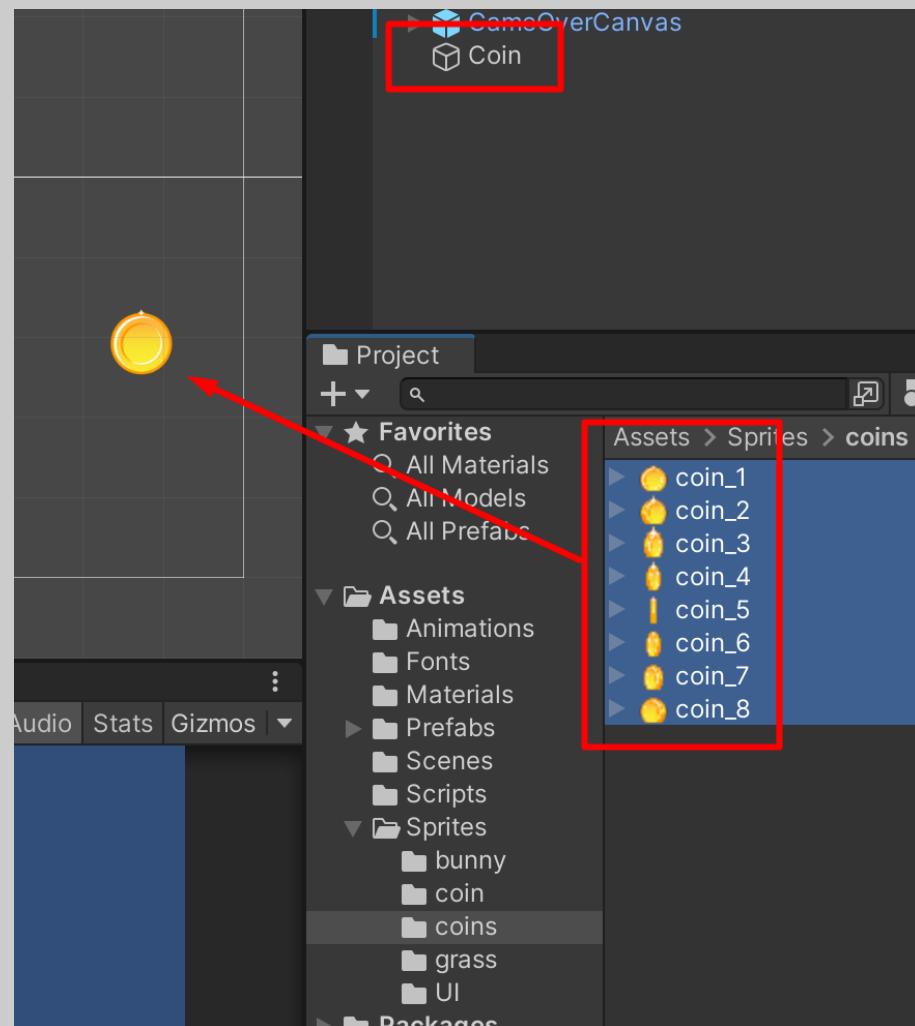
1. Recolección de items	2
2. Música y sonidos	14
3. Fondos y colocación en capas	21



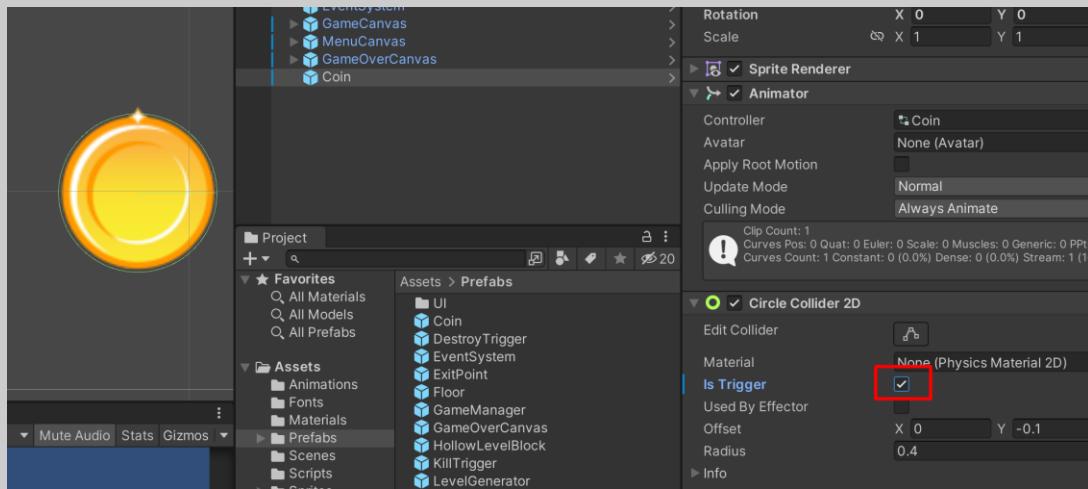
1. Recolección de items

Para terminar nuestro primer juego vamos a añadirle la posibilidad de recolectar ítems, así como actualizar los respectivos marcadores de puntos e items.

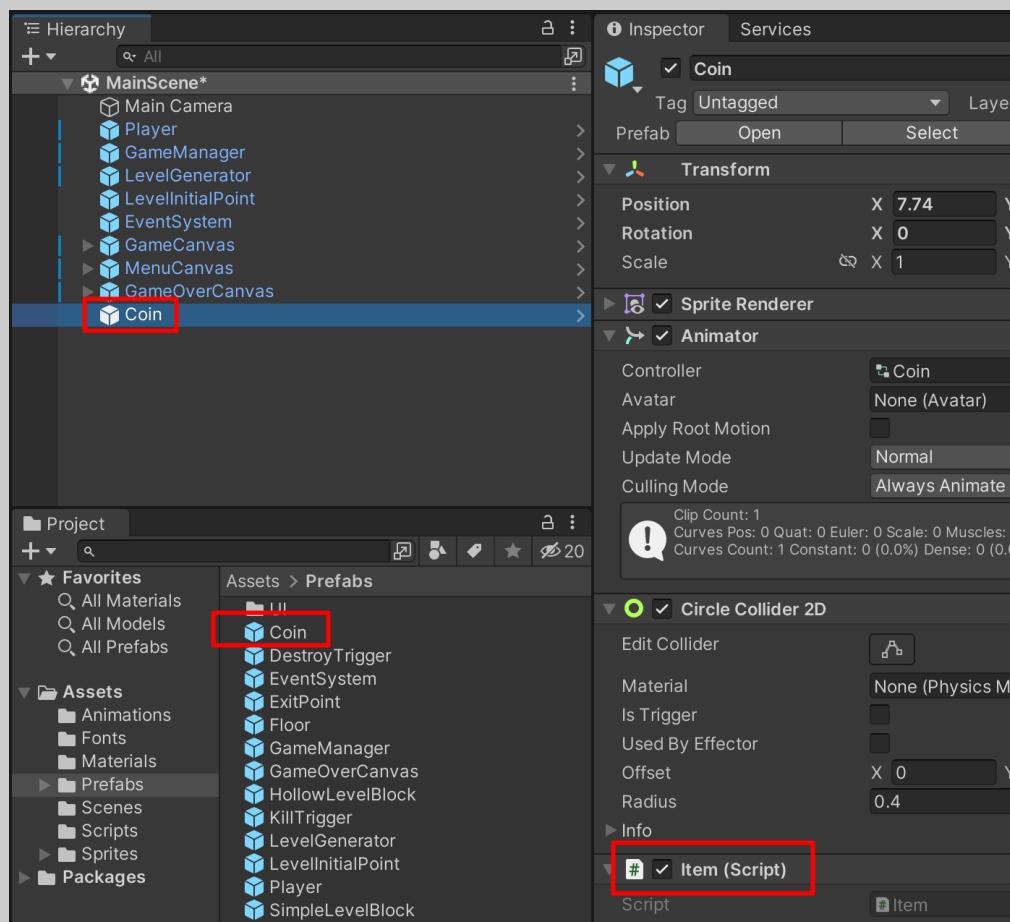
Comenzaremos incorporando a nuestro proyecto varios sprites que representan la rotación de una moneda, las arrastraremos a la Scene y crearemos la animación coin_rotation.anim:

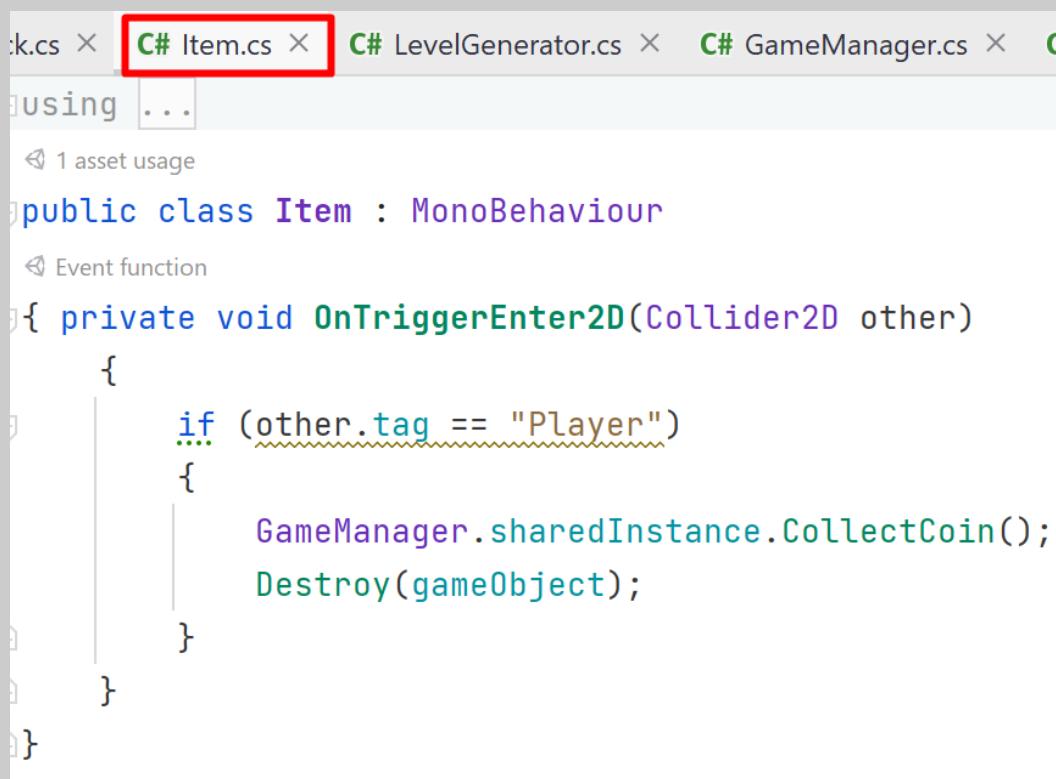


Agregamos a nuestro gameobject un Circle Collider 2D y marcamos el atributo “Is Trigger”:



Lo siguiente será crear un script al que llamaremos Item y se lo asignaremos a nuestro gameobject Coin. Acto seguido, guardamos nuestro prefab Coin y procederemos a editar el script Item.cs:





The screenshot shows a Unity code editor with several tabs at the top: C# Item.cs (highlighted with a red box), C# LevelGenerator.cs, C# GameManager.cs, and others. The Item.cs script contains the following code:

```
using UnityEngine;
public class Item : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.tag == "Player")
        {
            GameManager.sharedInstance.CollectCoin();
            Destroy(gameObject);
        }
    }
}
```

El método CollectCoin() al que llamamos desde el OnTriggerEnter2D será un método del GameManager que se limitará a incrementar en una unidad a la variable entera collectedCoins:



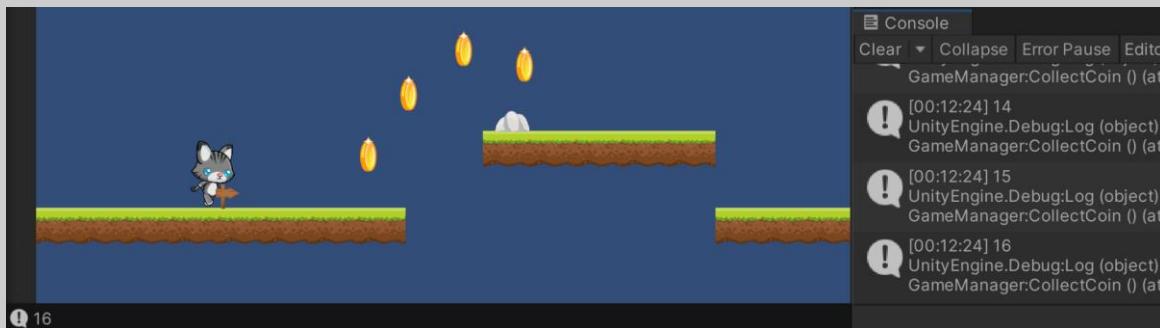
The screenshot shows a Unity code editor with tabs: C# GameManager.cs (highlighted with a red box), C# Item.cs, and others. The GameManager.cs script contains the following code:

```
public int collectedCoins = 0;

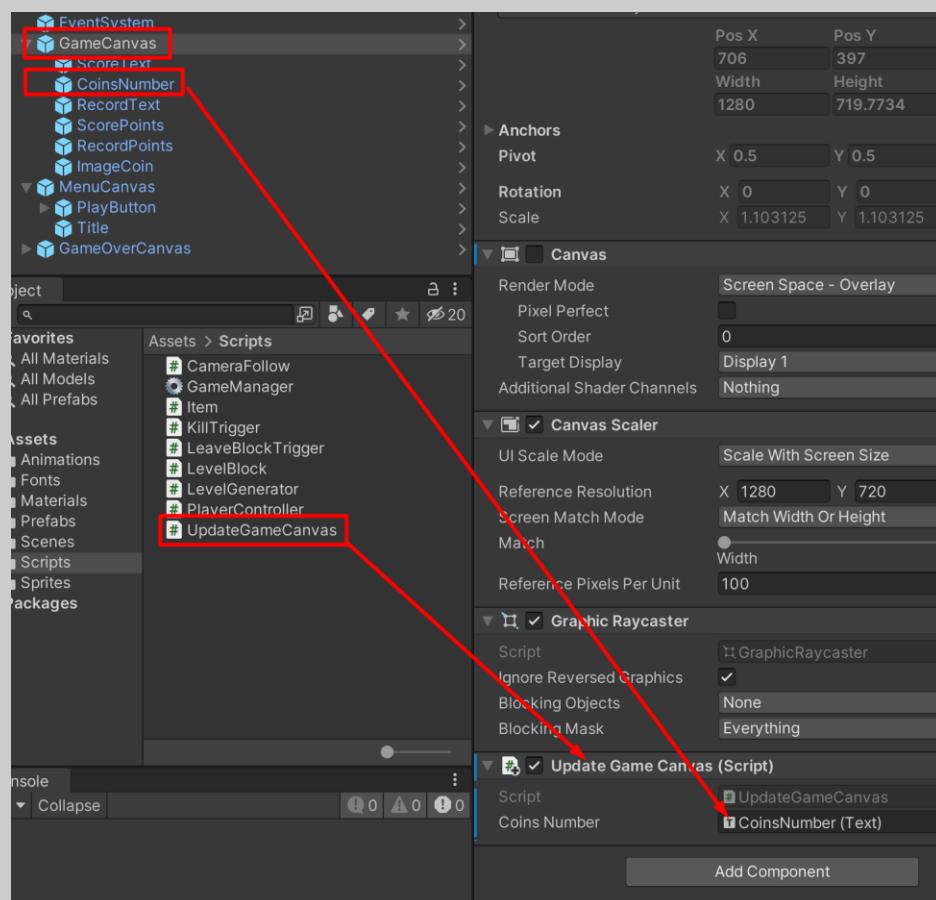
[1 usage]

public void CollectCoin()
{
    collectedCoins++;
    Debug.Log(collectedCoins);
}
```

Colocamos varias monedas en los prefabs que componen nuestro escenario (en el bloque original y en su variantes) y comprobamos que todo funciona correctamente:



En vez de que se nos muestre el contador de monedas en la consola, vamos a implementar que se actualice el marcador de monedas del GameCanvas. Para ello vamos a crear el script UpdateGameCanvas.cs que asignaremos al GameCanvas, y dentro de él la variable CoinsNumber que asignaremos al gameobject CoinsNumber:



¡Ojo! Como lo estamos haciendo con un TextMeshPro, el tipo de objeto pertenece a la clase TextMeshProUGUI:



```

    using TMPro;
    using UnityEngine;

    public class UpdateGameCanvas : MonoBehaviour
    {
        [SerializeField] private TextMeshProUGUI coinsNumber; // Variable highlighted with a red box
        // Changed in 1 asset

        void Update()
        {
            if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
            {
                coinsNumber.text = GameManager.sharedInstance.CollectedCoins.ToString();
            }
        }
    }

```

El siguiente paso será implementar el funcionamiento tanto del marcador de puntos como el marcador de la máxima puntuación lograda. Ten en cuenta que la puntuación máxima se debe guardar entre diferentes partidas (debe persistir) por lo que el motor Unity nos ofrece la clase Player Preferences, cuyos métodos nos sirven para almacenar pequeños datos concretos (máxima puntuación, música activada/desactivada, nivel de dificultad...)

Vamos a crear un método en el PlayerController.cs que nos devuelva la distancia recorrida por el jugador para así utilizarlo luego en nuestro UpdateGameCanvas.cs donde asignaremos su atributo scorePoints con el gameobject ScorePoints del GameCanvas:



```

    public float distanceTravelled = 0; // Variable highlighted with a red box
    // Frequently called 1 usage
    public float GetDistanceTravelled()
    {
        distanceTravelled =
            Vector2.Distance(a: new Vector2(startPosition.x, y: 0), b: new Vector2(this.transform.position.x, y: 0));
        return distanceTravelled;
    }

```

Fíjate que he utilizado la función Distance que me devuelve la distancia que hay entre dos objetos Vector2:

The screenshot shows the Unity Documentation for the `Vector2.Distance` method. The sidebar lists various Unity classes, and the main content area shows the declaration and description of the `Distance` method.

```
public static float Distance(Vector2 a, Vector2 b);
```

Returns the distance between `a` and `b`.
`Vector2.Distance(a,b)` is the same as `(a-b).magnitude`.

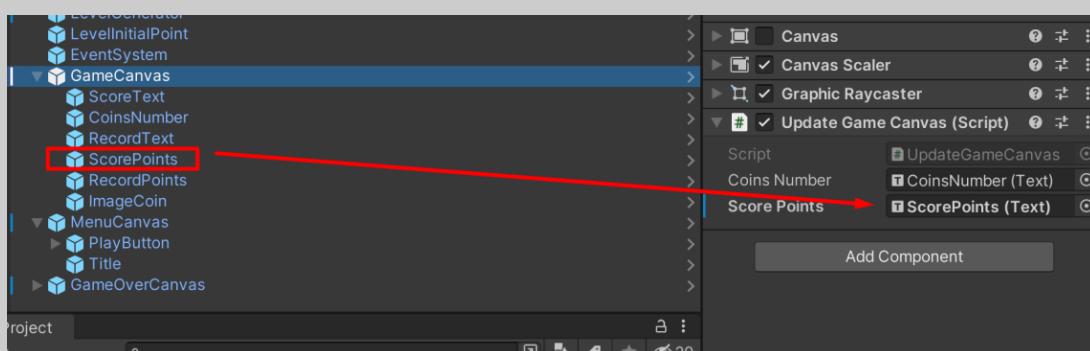
Ahora solo falta actualizar el marcador de puntuación:

```
C# UpdateGameCanvas.cs
public class UpdateGameCanvas : MonoBehaviour
{
    public Text coinsNumber;
    public Text scorePoints;

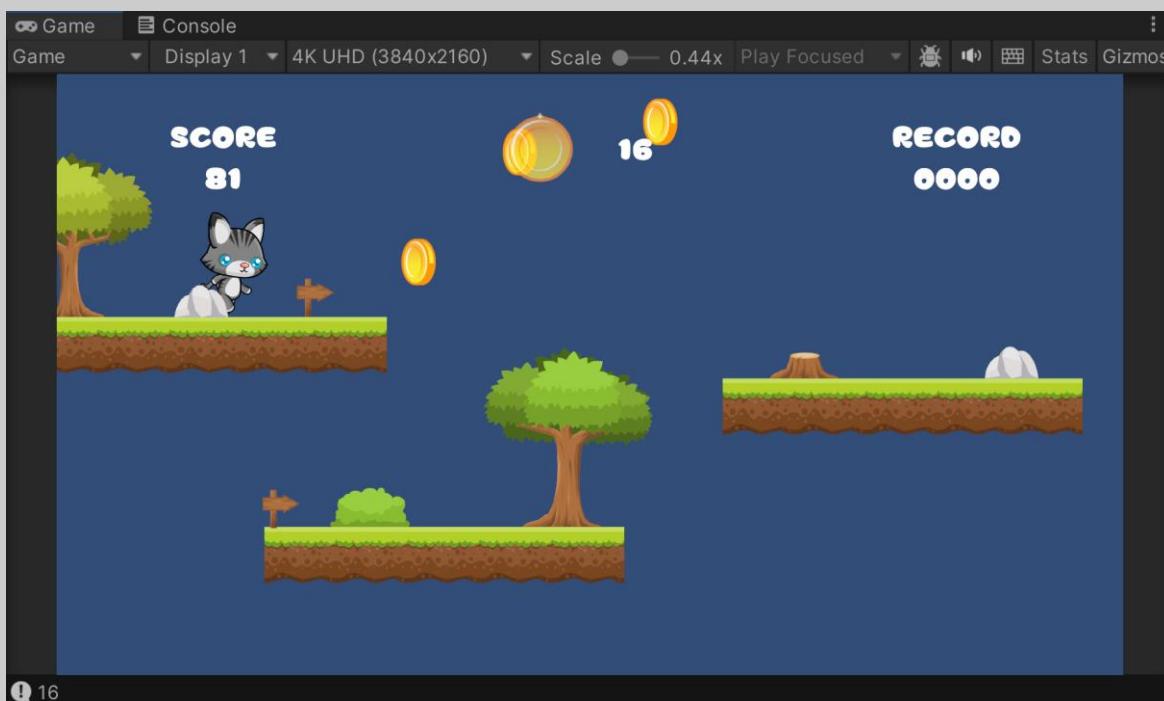
    void Update()
    {
        if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
        {
            coinsNumber.text = GameManager.sharedInstance.collectedCoins.ToString();

            //formateo la salida para que me aparezcan 0 decimales
            scorePoints.text = PlayerController.sharedInstance.GetDistanceTravelled().ToString("f0");
        }
    }
}
```

Recuerda asignarle el gameobject ScorePoints:



Y comprobamos el resultado:



Una vez que muere nuestro protagonista deberemos comprobar si hemos logrado un récord de puntuación máxima. Para ello, consultaremos dentro de PlayerPrefs el campo "highscore" con un método de tipo get, y lo sobreescrivimos con un método de tipo set:

```
public void KillPlayer()
{
    GameManager.sharedInstance.GameOver();
    animator.SetBool(name: "isAlive", value: false);
    Invoke(methodName: "SleepPlayer", time: 1f);
    if (PlayerPrefs.GetFloat(key: "highscore", defaultValue: 0) < distanceTravelled)
    {
        PlayerPrefs.SetFloat("highscore", distanceTravelled);
    }
}
[1 usage]
public void SleepPlayer()
{
    GetComponent<Rigidbody2D>().Sleep();
}
```

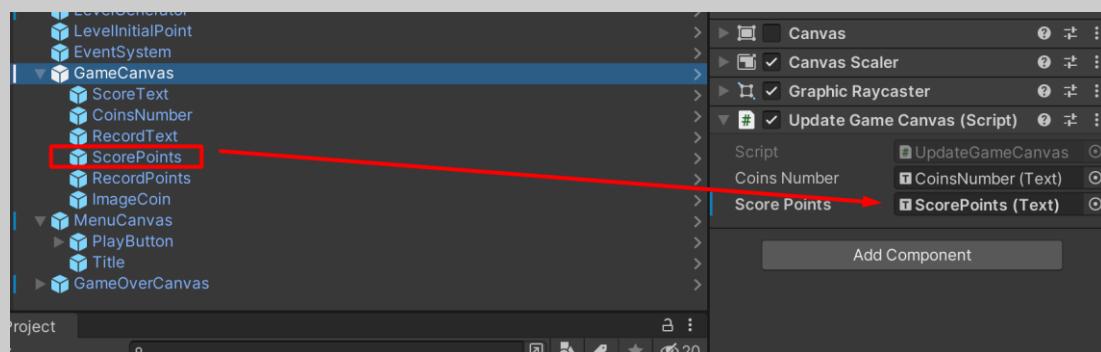
A continuación lo mostraremos en el marcador del Canvas correspondiente:

```
l.cs × C# UpdateGameCanvas.cs × C# PlayerController.cs × C# GameManager.cs × C# Item.cs × C# LevelGenerator.cs × C# CameraFollow.cs
using ...
public class UpdateGameCanvas : MonoBehaviour
{
    public Text coinsNumber; // Changed in 1 asset
    public Text scorePoints; // Unchanged
    public Text recordPoints; // Unchanged
    void Update()
    {
        if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
        {
            coinsNumber.text = GameManager.sharedInstance.collectedCoins.ToString();

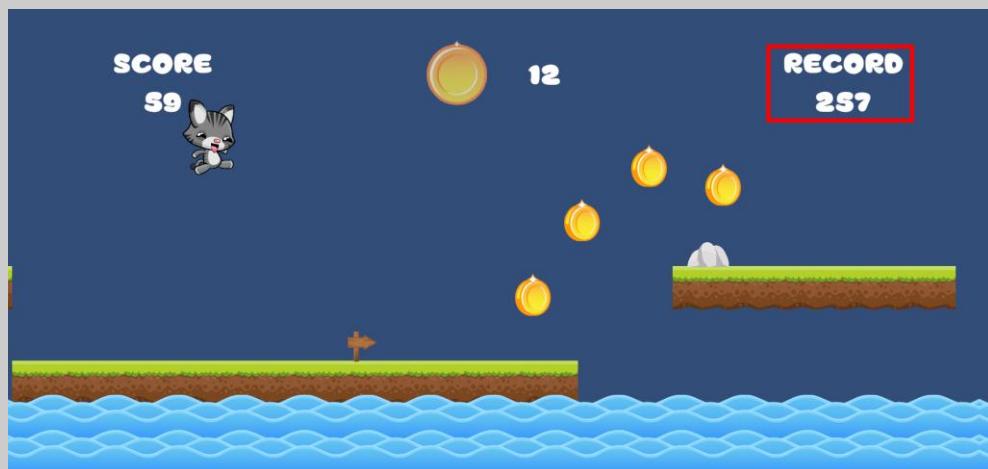
            //formateo la salida para que me aparezcan 0 decimales
            scorePoints.text = PlayerController.sharedInstance.GetDistanceTravelled().ToString("f0");

            recordPoints.text = PlayerPrefs.GetFloat("highscore", 0).ToString("f0");
        }
    }
}
```

Recuerda asignarle el gameobject RecordPoints:



Y comprobamos el resultado:



Sería interesante que, durante el juego, cuando rebasemos el record, este se vaya actualizando junto a nuestra puntuación. Lo implementará el alumno en su proyecto.

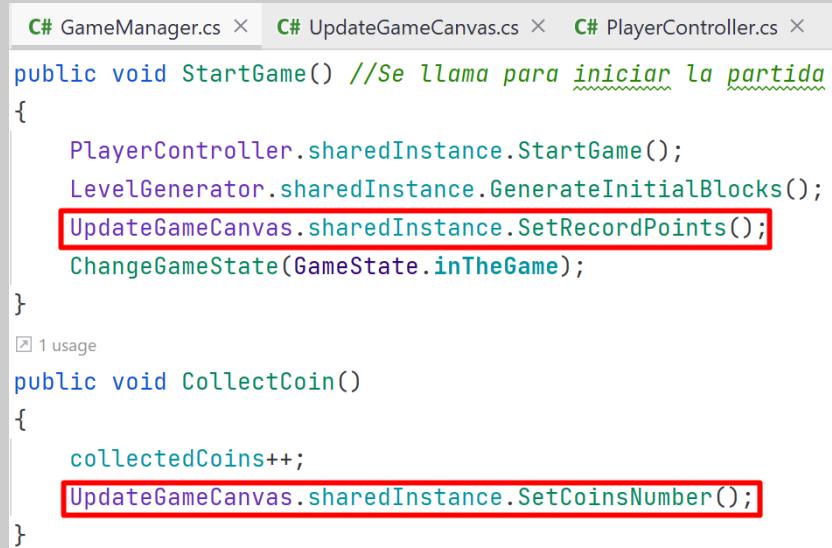
Ahora que todo funciona, vamos a hacer una pequeña modificación para hacer el código más eficiente en el UpdateCanvas.cs, ya que no tiene mucho sentido refrescar en cada frame el marcador de monedas y el del récord de puntos. En lugar de eso, vamos a utilizar el patrón singleton y modificaremos estos dos marcadores solo cuando se produzca un cambio, y no en cada frame:

```
ck.cs × C# UpdateGameCanvas.cs × C# GameManager.cs × C# PlayerController.cs × C# Item.cs × C# LevelGenerator.cs × C# CameraFollow.cs ×
    ↗ 1 asset usage  ↗ 3 usages  ↗ 1 exposing API
public class UpdateGameCanvas : MonoBehaviour
{
    public static UpdateGameCanvas sharedInstance;

    public Text coinsNumber;  ↗ Changed in 1 asset
    public Text scorePoints;  ↗ Changed in 1 asset
    public Text recordPoints;  ↗ Changed in 1 asset
    ↗ Event function
    private void Awake()
    {
        sharedInstance = this;
    }
    ↗ 1 usage
    public void SetRecordPoints()
    {
        recordPoints.text = PlayerPrefs.GetFloat(key: "highscore", defaultValue: 0).ToString(format: "f0");
    }
    ↗ 1 usage
    public void SetCoinsNumber()
    {
        coinsNumber.text = GameManager.sharedInstance.collectedCoins.ToString();
    }
    ↗ Event function
    void Update()
    {
        if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
        {
            //con "f0" formateo la salida para que me aparezcan 0 decimales
            scorePoints.text = PlayerController.sharedInstance.GetDistanceTravelled().ToString(format: "f0");
        }
    }
}
```

Ahora bastará con llamar a esos métodos en el momento que se necesiten, y no 60 veces por cada frame como pasaba antes, sobrecargando innecesariamente el dispositivo que ejecuta el juego.

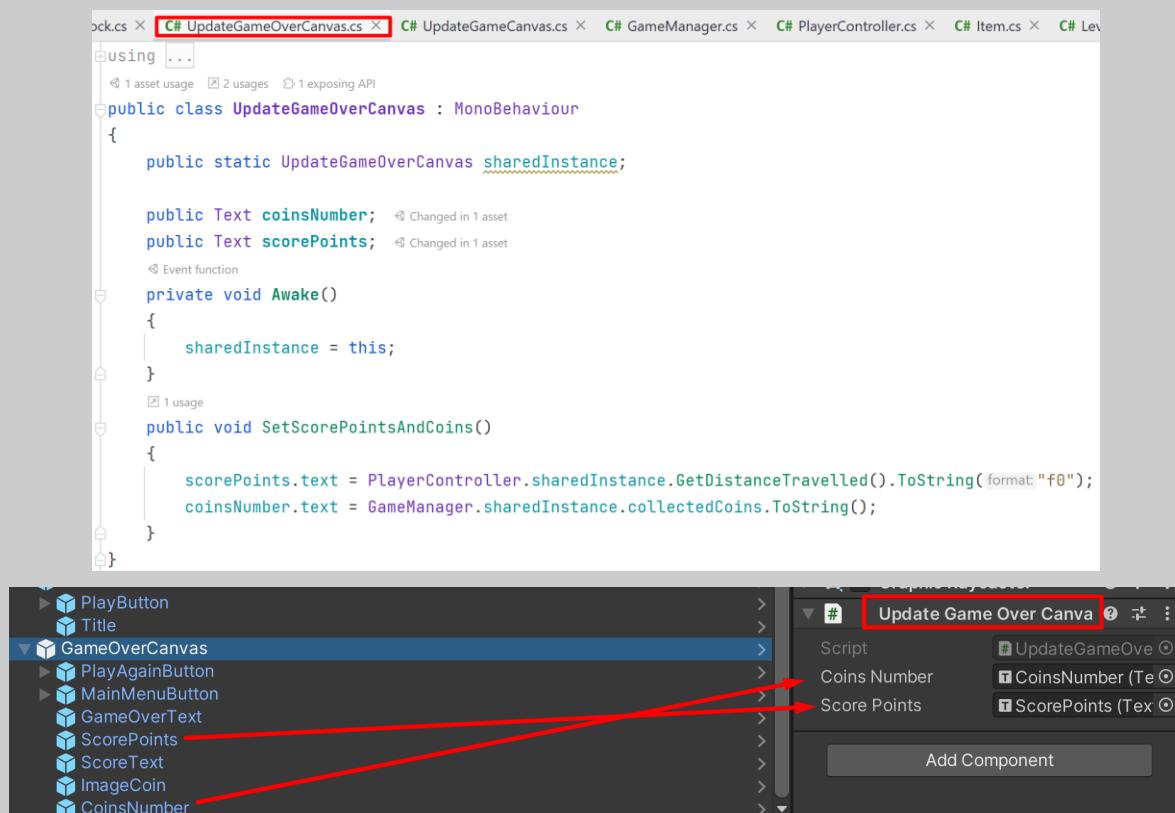
Al método setRecordPoints() le llamaremos una sola vez en el método StartGame() del GameManager.cs, y al método SetCoinsNumber le llamaremos en el método CollectCoin() cada vez que recojamos una moneda:



```
C# GameManager.cs × C# UpdateGameCanvas.cs × C# PlayerController.cs ×
public void StartGame() //Se llama para iniciar la partida
{
    PlayerController.sharedInstance.StartGame();
    LevelGenerator.sharedInstance.GenerateInitialBlocks();
    UpdateGameCanvas.sharedInstance.SetRecordPoints();
    ChangeGameState(GameState.inTheGame);
}

[1 usage]
public void CollectCoin()
{
    collectedCoins++;
    UpdateGameCanvas.sharedInstance.SetCoinsNumber();
}
```

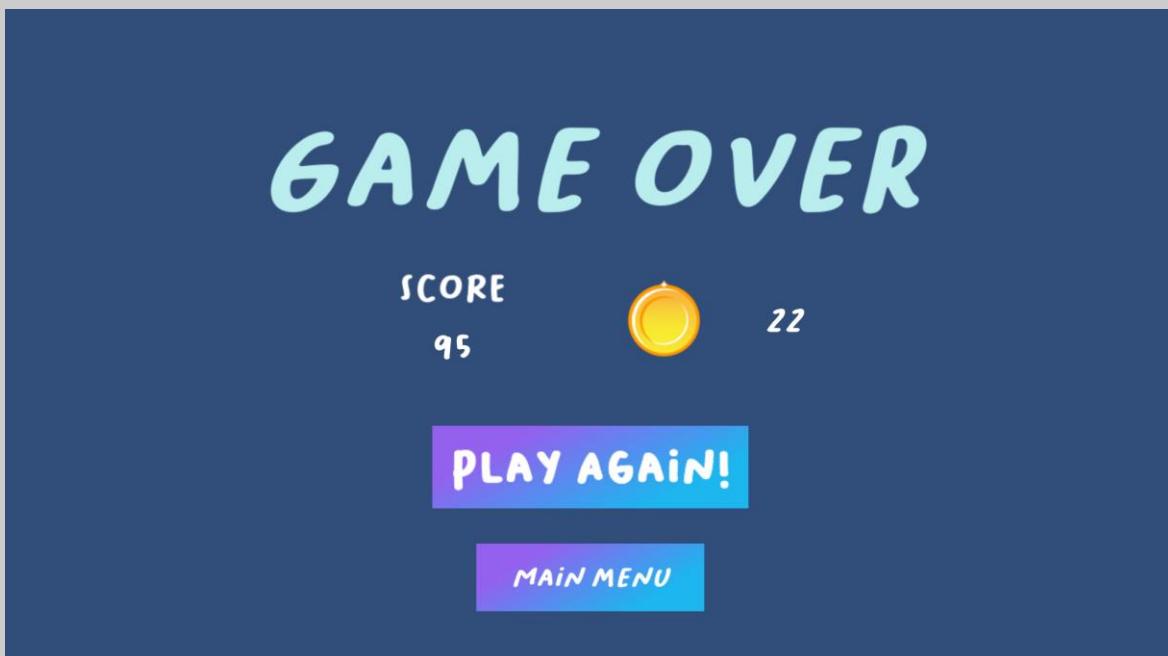
Para terminar este punto del tema, solo nos faltaría crear un script, muy similar al “UpdateGameCanvas.cs”, al que llamaremos “UpdateGameOverCanvas.cs” y se lo asignaremos al gameobject GameOverCanvas:



Recuerda sobreescribir el prefab. Luego, solo nos faltaría llamar al método SetScorePointsAndCoins() desde el método GameOver del GameManager.cs:

```
C# GameManager.cs × C# UpdateGameOverCanvas.cs × C# UpdateGameCanvas.cs × C#
public void GameOver() //Se llama cuando el jugador muere
{
    ChangeGameState(GameState.gameOver);
    LevelGenerator.sharedInstance.RemoveAllBlocks();
    UpdateGameOverCanvas.sharedInstance.SetScorePointsAndCoins();
}
```

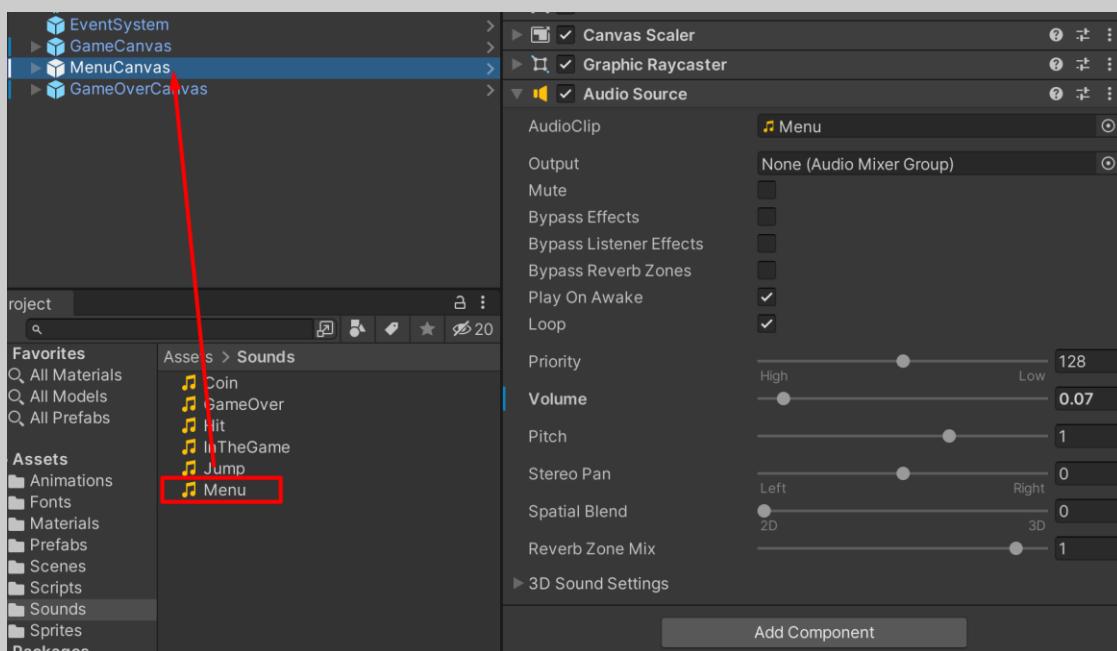
Comprobamos que todo esté funcionando correctamente:



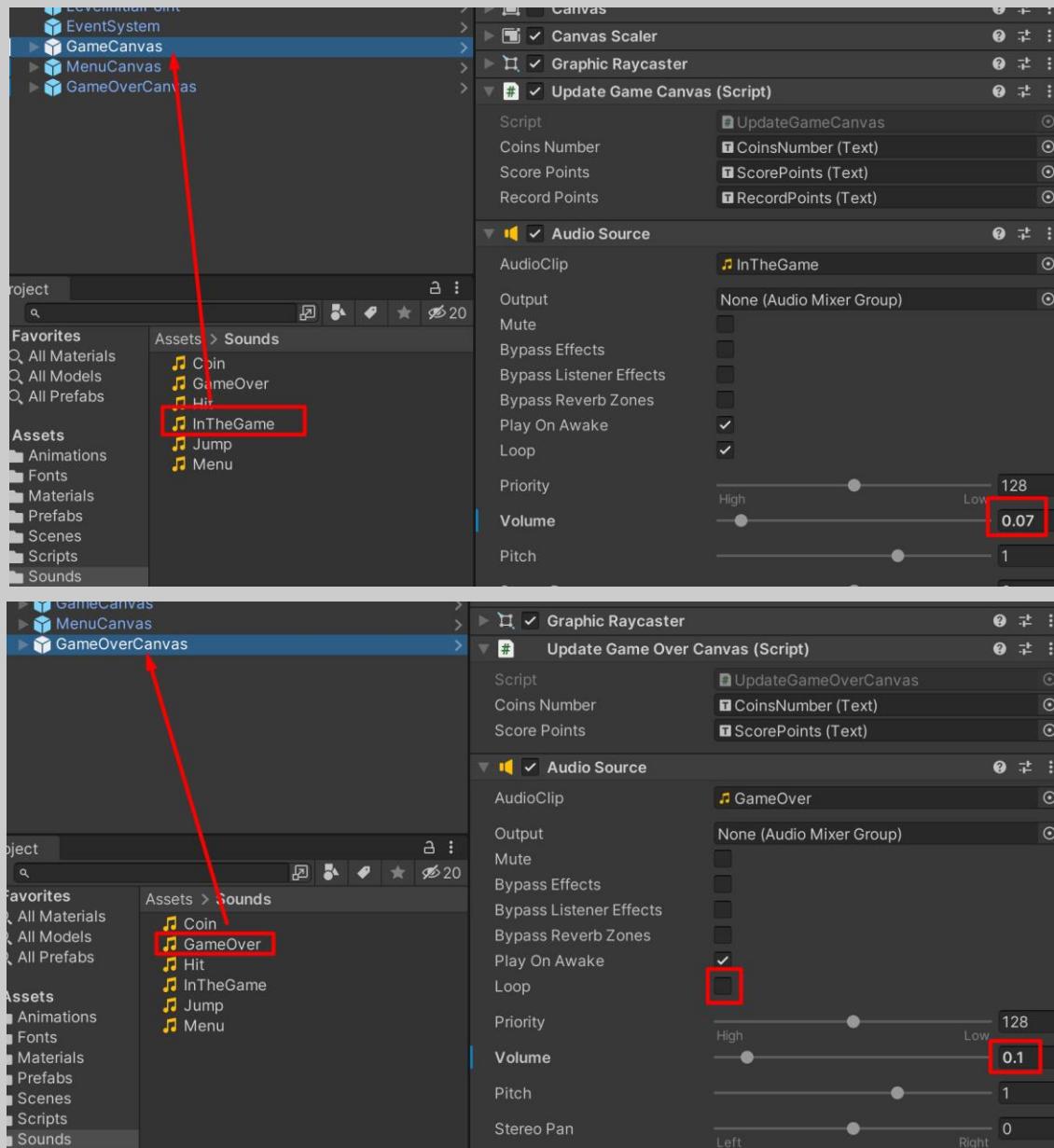
2. Música y sonidos

Para ir finalizando el desarrollo de nuestro juego vamos a añadirle algunos sonidos que hagan las partidas más amenas y reconfortantes, aumentando el feedback con el jugador. Estos sonidos los guardaremos en la carpeta Art>Audio>Music en el caso de las melodías, y en Art>Audio>Sounds en el caso de los sonidos FX.

Para asignarle un archivo de sonido a alguno de los componentes de nuestro juego, necesitamos que ese gameobject tenga un componente de tipo “ AudioSource ”. No obstante, si arrastramos un sonido directamente a uno de nuestros gameobjects de juego, Unity automáticamente le creará el componente “ AudioSource ” con la fuente de sonido asignada. Por ejemplo, voy a asignarle al gameobject MenuCanvas el sonido llamado “ Menu ”. Veremos que automáticamente se le crea el componente Audio Source y aprovecho para ajustar el volumen:



A continuación hago lo mismo para el GameCanvas y para el GameOverCanvas (fíjate que en este último no me interesa que el sonido se repita en bucle, por lo que desactivo el atributo Loop en el AudioSource:



Si ahora probamos nuestro juego veríamos como todos los sonidos se reproducen a la vez, ya que hemos dejado marcada la casilla Play On Awake, lo que hace que se reproduzca el sonido nada más que el gameobject aparezca en la jerarquía de la escena.

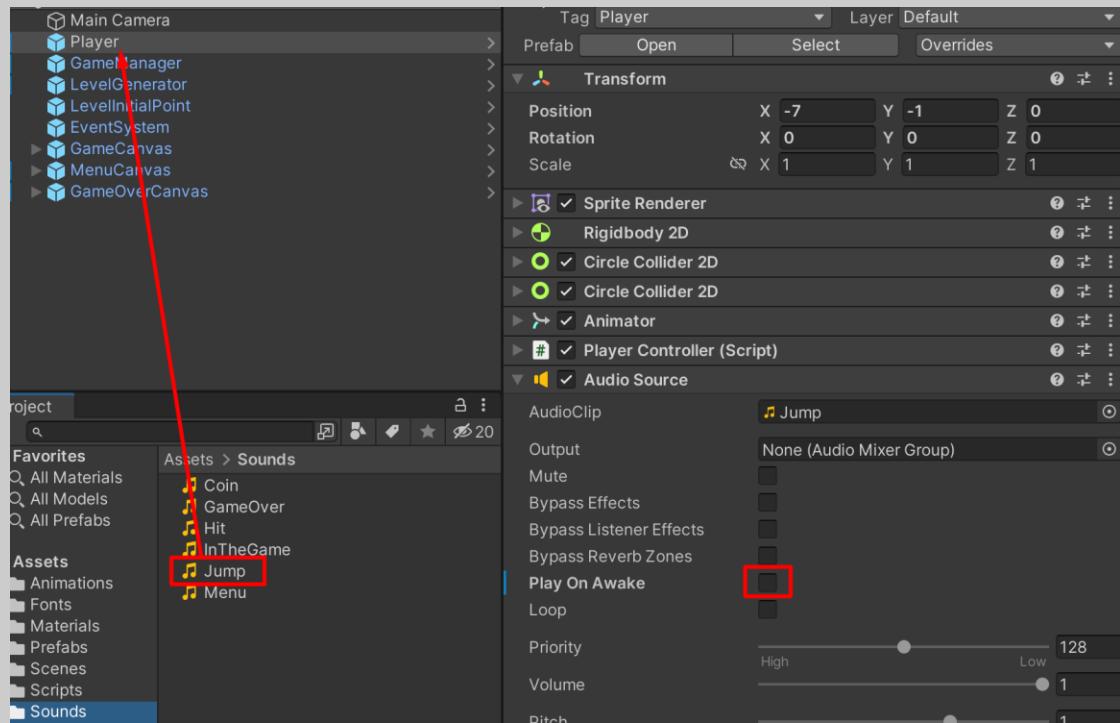
Como este no es el efecto deseado, procederemos a modificar por código que el sonido de cada Canvas se reproduzca en función de que esté activado o desactivado (podemos dejar marcada Play On Awake si queremos, ya que no nos va a afectar):

```

; × C# GameManager.cs × C# UpdateGameOverCanvas.cs × C# UpdateGameCanvas.cs × C# Playe
3 usages
void ChangeGameState(GameState newGameState)
{
    if (newGameState == GameState.menu)
    {
        //La escena de Unity deberá mostrar el menú principal
        menuCanvas.enabled = true;
        menuCanvas.GetComponent< AudioSource >().Play();
        gameCanvas.enabled = false;
        gameCanvas.GetComponent< AudioSource >().Stop();
        gameOverCanvas.enabled = false;
        gameOverCanvas.GetComponent< AudioSource >().Stop();
    }
    else if (newGameState == GameState.inTheGame)
    {
        currentState = GameState.inTheGame;
        //La escena de Unity deberá configurarse para mostrar el juego en si
        menuCanvas.enabled = false;
        gameCanvas.enabled = true;
        gameOverCanvas.enabled = false;
        menuCanvas.GetComponent< AudioSource >().Stop();
        gameCanvas.GetComponent< AudioSource >().Play();
        gameOverCanvas.GetComponent< AudioSource >().Stop();
    }
    else if (newGameState == GameState.gameOver)
    {
        currentState = GameState.gameOver;
        //La escena de Unity deberá mostrar el menú de fin de partida
        menuCanvas.enabled = false;
        gameCanvas.enabled = false;
        gameOverCanvas.enabled = true;
        menuCanvas.GetComponent< AudioSource >().Stop();
        gameCanvas.GetComponent< AudioSource >().Stop();
        gameOverCanvas.GetComponent< AudioSource >().Play();
    }
}
IEnumerator EventFunction
void Start()
{
    //Me aseguro de que tenga el valor GameState.menu no sea que lo haya cambiado en el inspector
    currentState = GameState.menu;
    menuCanvas.enabled = true;
    gameCanvas.enabled = false;
    gameOverCanvas.enabled = false;
    menuCanvas.GetComponent< AudioSource >().Play();
    gameCanvas.GetComponent< AudioSource >().Stop();
    gameOverCanvas.GetComponent< AudioSource >().Stop();
}

```

Ahora vamos con el sonido del salto. Para ello le asignamos el sonido "Jump" a nuestro gameobject Player, desactivamos "Play On Awake" y guardamos el prefab:



Luego añadiremos en el código que se reproduzca el sonido cuando nuestro protagonista salte (y de paso vamos a cambiar la asignación del botón de salto):

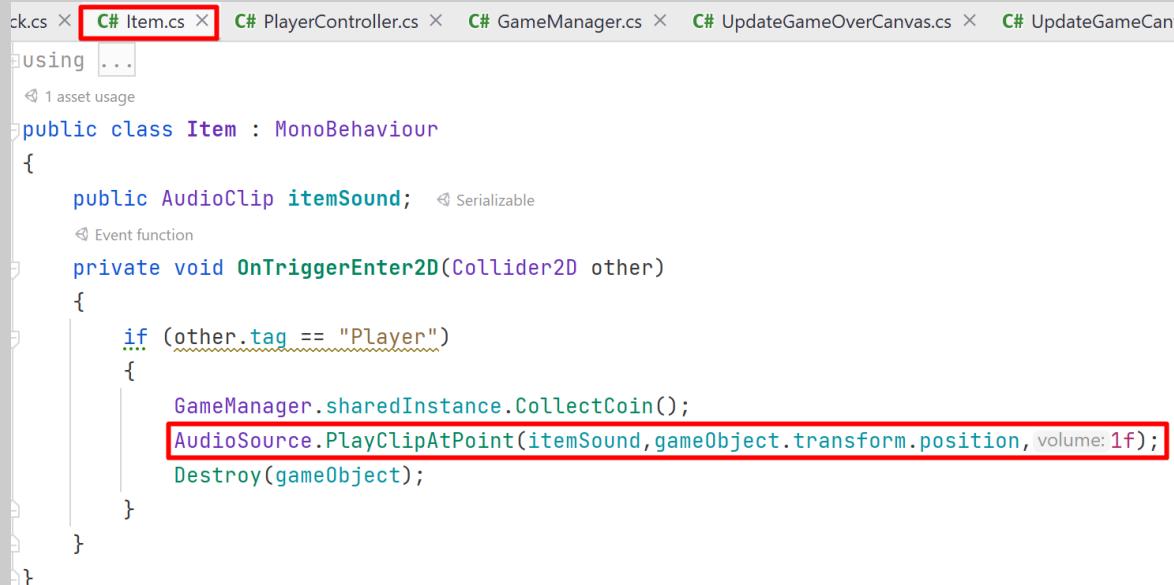
```

C# PlayerController.cs × C# GameManager.cs × C# UpdateGameOverCanvas.cs × C# UpdateGameCar

void Update()
{
    //Solo salta si estamos en el estado inTheGame
    if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
    {
        animator.SetBool(name: "isGrounded", value: isOnTheFloor());
        if (Input.GetButtonDown("Fire1")) //
        {
            if (isOnTheFloor())
            {
                Jump();
                GetComponent< AudioSource >().Play();
            }
        }
    }
}

```

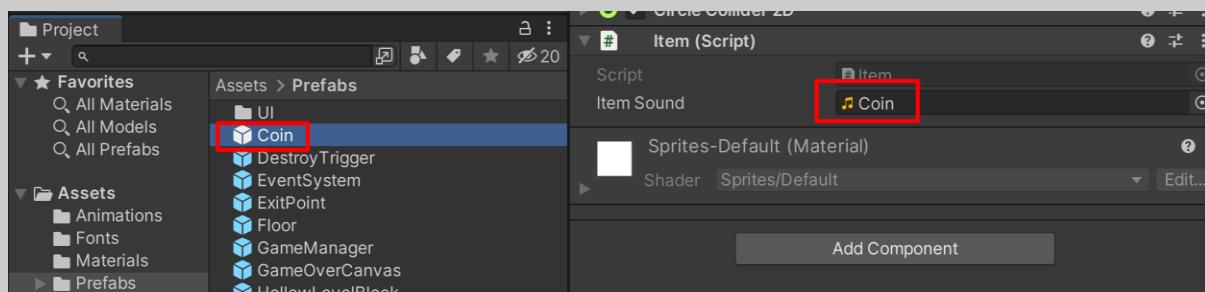
Ya solo nos falta añadirle a nuestro juego el sonido “Coin” al recoger las monedas y el sonido “Hit” cuando nos caemos. Fíjate que en el caso de las monedas no le vamos a poder asignar el sonido directamente a la moneda, ya que esta al ser recogida es destruida por lo que no daría tiempo a escuchar el sonido. En su lugar, tendremos que usar el método estático AudioSource.PlayClipAtPoint, al que le tendremos que pasar un AudioClip y la posición desde donde suena el sonido (que puede ser la posición del mismo gameobject):



```

Item.cs X C# Item.cs X C# PlayerController.cs X C# GameManager.cs X C# UpdateGameOverCanvas.cs X C# UpdateGameCan...
using ...
public class Item : MonoBehaviour
{
    public AudioClip itemSound; // Serializable
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.tag == "Player")
        {
            GameManager.sharedInstance.CollectCoin();
            AudioSource.PlayClipAtPoint(itemSound, gameObject.transform.position, volume: 1f);
            Destroy(gameObject);
        }
    }
}

```



Lo mismo ocurre con el sonido “Hit” que no se lo podemos pasar directamente al gameobject KillPlayer, ya que cuando morimos se destruyen todos los bloques del escenario y no daría tiempo a escucharlo. Por ello, aquí también tendremos que usar el método estático AudioSource.PlayClipAtPoint pasandole un AudioClip al script:

```
cs < C# KillTrigger.cs < C# Item.cs < C# PlayerController.cs < C# LevelBlock.cs < C# LevelGenerator.cs < C# UpdateGameOverCanvas.cs < C#
using ...  

↳ 1 asset usage  

public class KillTrigger : MonoBehaviour  

{  

    public AudioClip hit; // Serialized  

    Event function  

    private void OnTriggerEnter2D(Collider2D other)  

    {  

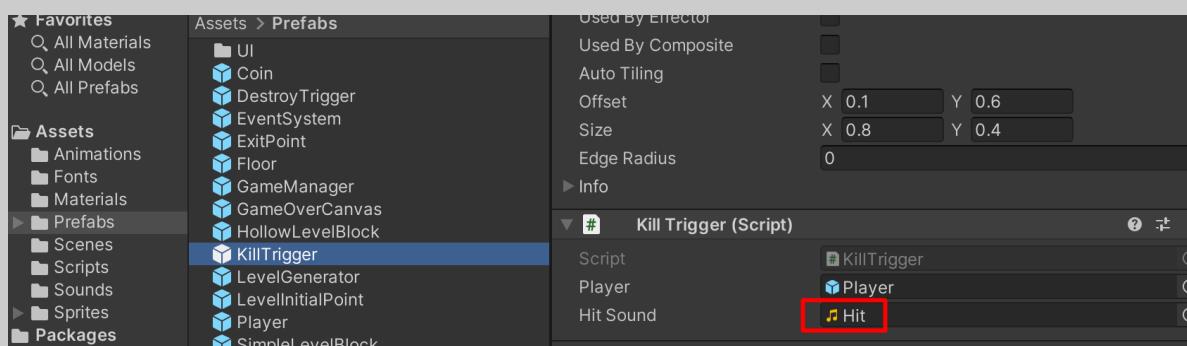
        if (other.tag == "Player")  

        {  

            AudioSource.PlayClipAtPoint(hit, transform.position);  

            PlayerController.sharedInstance.KillPlayer();  

        }
    }
}
```

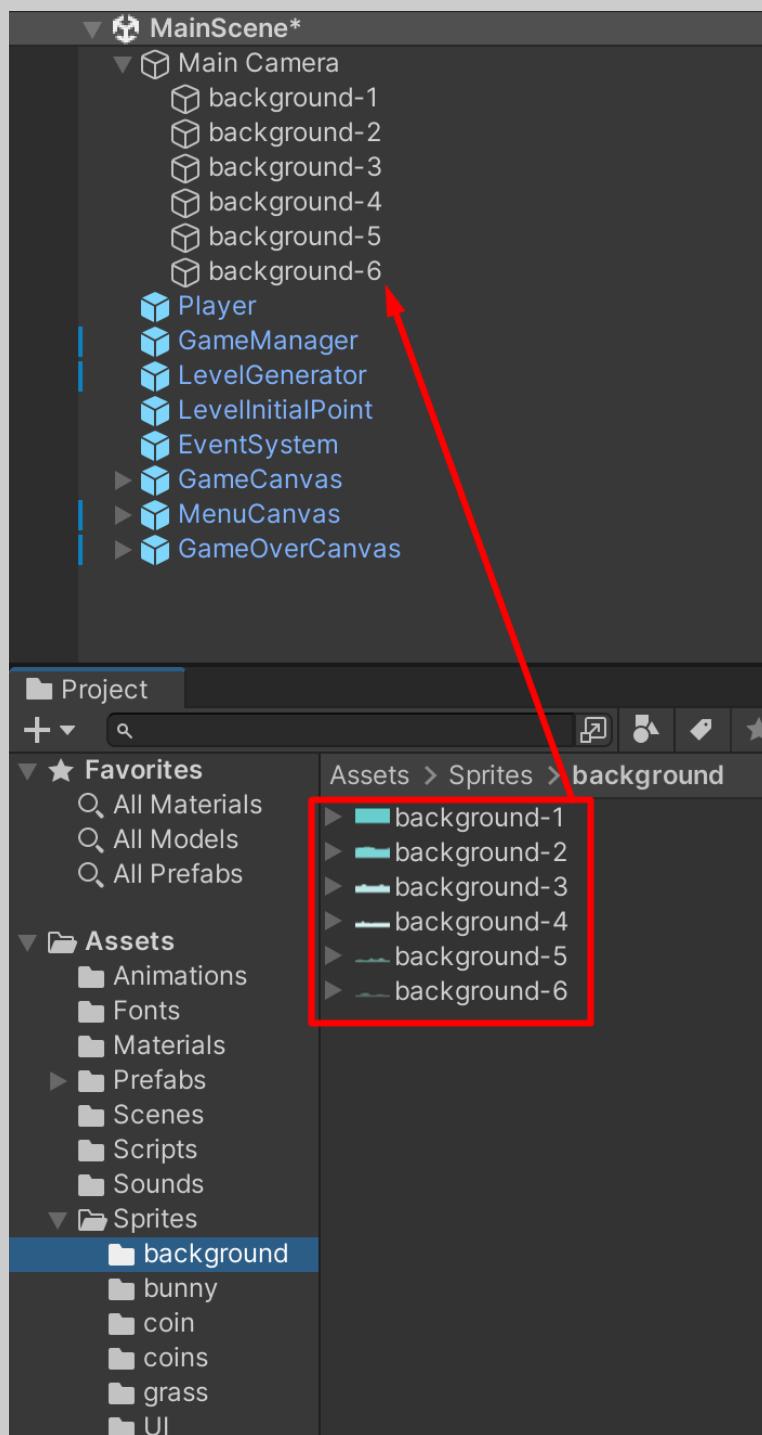


Para que se siga viendo la animación de hit vamos a hacer un pequeño cambio en el PlayerController.cs, retrasando también la llamada al método GameOver():

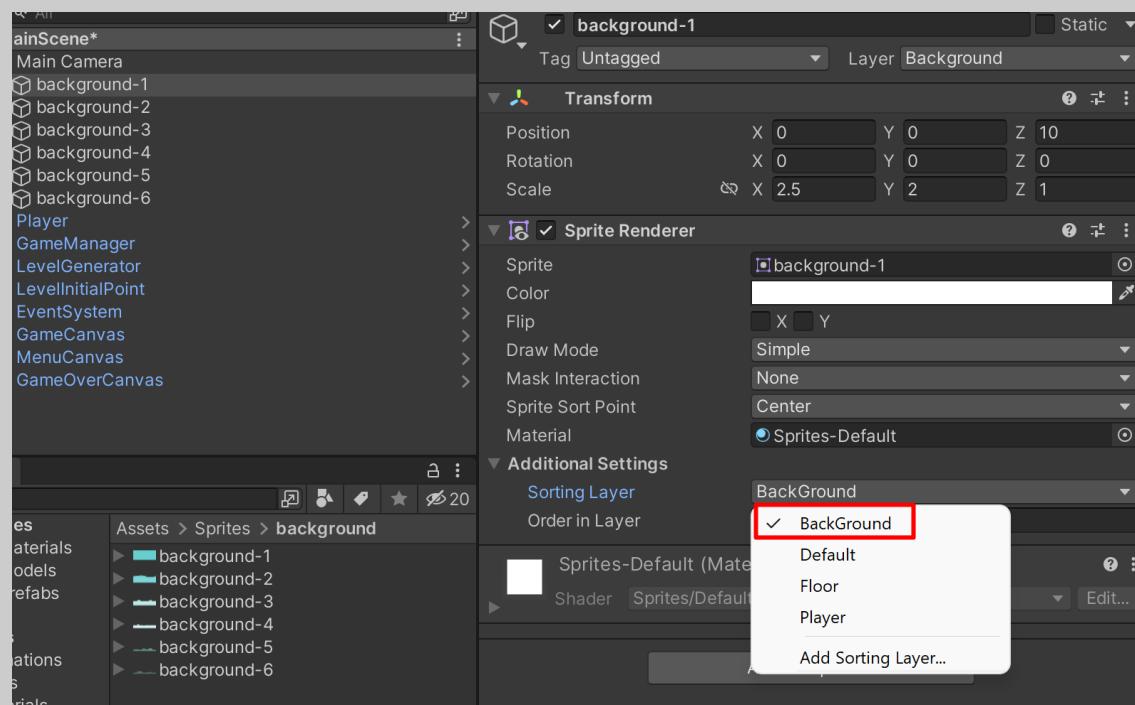
```
trigger.cs × C# Item.cs × C# PlayerController.cs × C# LevelBlock.cs × C# LevelGenerator.cs × C# UpdateGameOverCanvas.cs × C# UpdateGameCanvas.cs × C# LeaveBlockTrigger.cs × C# Camera.cs ×
suggestion
OK with that No, whitespace should insert suggestion
public void KillPlayer()
{
    animator.SetBool(name: "isAlive", value: false);
    Invoke(methodName: "DelayGameOver", time: 1f);
    if (PlayerPrefs.GetFloat(key: "highscore", defaultValue: 0) < distanceTravelled)
    {
        PlayerPrefs.SetFloat("highscore", distanceTravelled);
    }
}
1 usage
public void DelayGameOver()
{
    GetComponent<Rigidbody2D>().Sleep();
    GameManager.sharedInstance.GameOver();
}
```

3. Fondos y colocación en capas

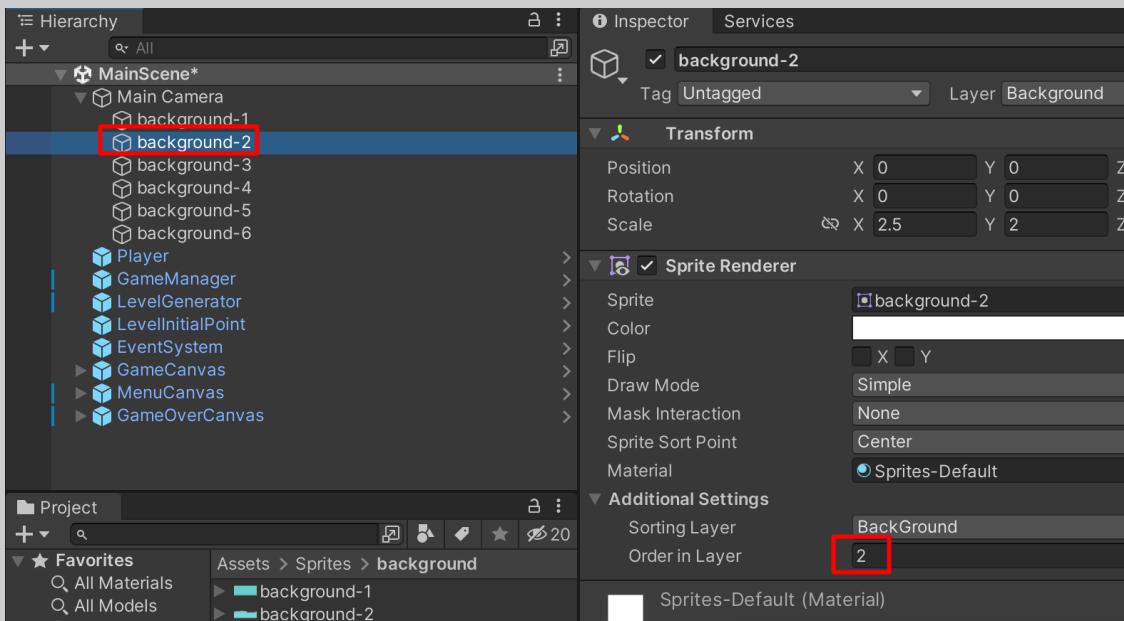
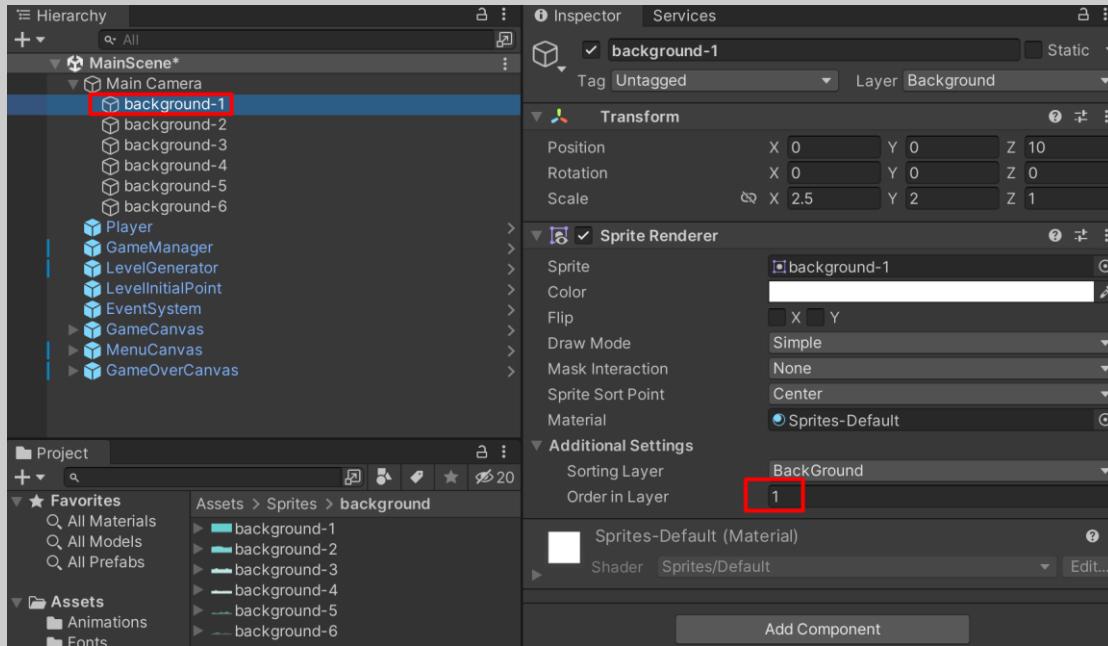
El último punto del tema consiste en colocar un fondo personalizado a nuestro juego, para hacerlo más atractivo y darle a la escena una mayor sensación de velocidad. Para ello, la forma más sencilla, es utilizar varias imágenes de fondo sobreuestas una sobre otra, para que den sensación de profundidad al jugador. Las colocaremos todas como hijas de la cámara, para que así se vayan desplazando a la par de esta:



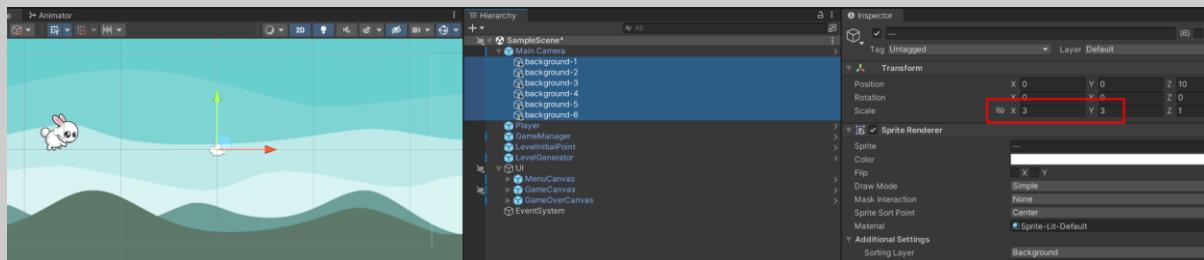
Las pondremos en un orden determinado en una nueva capa (Sorting Layer) en el fondo de la escena a la que llamaremos, por ejemplo, “BackGround”:



Y colocamos los diferentes backgrounds en orden mediante el campo “Order in Layer”:



En caso de que los fondos no te ocupen la totalidad del fondo del juego, deberás escalarlos:



Y aquí tenemos el resultado final:

