

04.- DESARROLLO DE UN JUEGO 2D SENCILLO EN UNITY

1. Ampliando nuestro juego: La clase Collectable (Item)	2
2. Ampliando nuestro juego: Utilización de una corutina (IEnumerator)	7
3. Ampliando nuestro juego: Creación de una pequeña animación	9
4. Ampliando nuestro juego: Creación de un enemigo	17
5. Ampliando nuestro juego: Cinemachine	29



1. Ampliando nuestro juego: La clase Collectable (Item)

En nuestro juego solo podemos recoger monedas, que como hemos visto reciben el nombre de item o colecciónable. Si en nuestro juego tuviéramos la posibilidad de recoger otros tipos de colecciónables (por ejemplo, zanahorias, pociónes, estrellas...) sería muy interesante que todos pertenecieran a una clase llamada, por ejemplo, “Collectable” para ser tratados todos de forma general, ya que su comportamiento va a ser muy parecido.

Por ello, vamos a modificar ligeramente nuestro juego para añadir esta manera de proceder y, una vez que funcione correctamente para las monedas, añadiremos un nuevo colecciónable, concretamente, una zanahoria.

En vez de crear una clase llamada “Collectable”, vamos a trabajar sobre la clase Item que ya teníamos creada. Quedará de la siguiente manera (explicaciones en comentarios):

```
C# Item.cs x C# GameManager.cs x C# UpdateGameCanvas.cs x
1  using UnityEngine;
2 ^o public class Item : MonoBehaviour
3 {
4     public AudioClip itemSound; //En vez de coinSound
5     private bool isCollected = false; //Para saber si ha sido recogido o no
6     public int value; //Para que tenga un valor el item recogido
7
8     //Método para activar el item y su collider
9     void Show()
10    {
11        this.GetComponent<SpriteRenderer>().enabled = true; //activa la imagen del item
12        this.GetComponent<CircleCollider2D>().enabled = true; //activa el collider del item
13        isCollected = false;
14    }
15    //Método para desactivar el item y su collider. (más óptimo que Destroy)
16    void Hide()
17    {
18        this.GetComponent<SpriteRenderer>().enabled = false; //activa la imagen del item
19        this.GetComponent<CircleCollider2D>().enabled = false; //activa el collider del item
20    }
21
22    void Collect()
23    {
24        isCollected = true;
25        Hide();
26        GameManager.sharedInstance.CollectorsItem(value); //En vez de CollectCoin. Además ahora le pasamos un value
27        AudioSource.PlayClipAtPoint(itemSound, transform.position);
28    }
29
30    // Event function
31    private void OnTriggerEnter2D(Collider2D other)
32    {
33        if (other.tag == "Player")
34        {
35            Collect();
36        }
37    }
38}
```

Fíjate que los cambios que acabo de hacer también afectan al GameManager.cs, al UpdateGameCanvas.cs y al UpdateGameOverCanvas.cs:

```

C# GameManager.cs X C# UpdateGameCanvas.cs X

using ...
16 usages 4 exposing APIs

public enum GameState
{
    menu,
    inTheGame,
    gameOver
}

1 asset usage 8 usages 1 exposing API

public class GameManager : MonoBehaviour
{
    public static GameManager sharedInstance;

    public GameState currentGameState = GameState.menu; Unchanged

    public Canvas MenuCanvas; Changed in 0+ assets
    public Canvas GameCanvas; Changed in 0+ assets
    public Canvas GameOverCanvas; Changed in 0+ assets

    public int collectObject=0; Unchanged

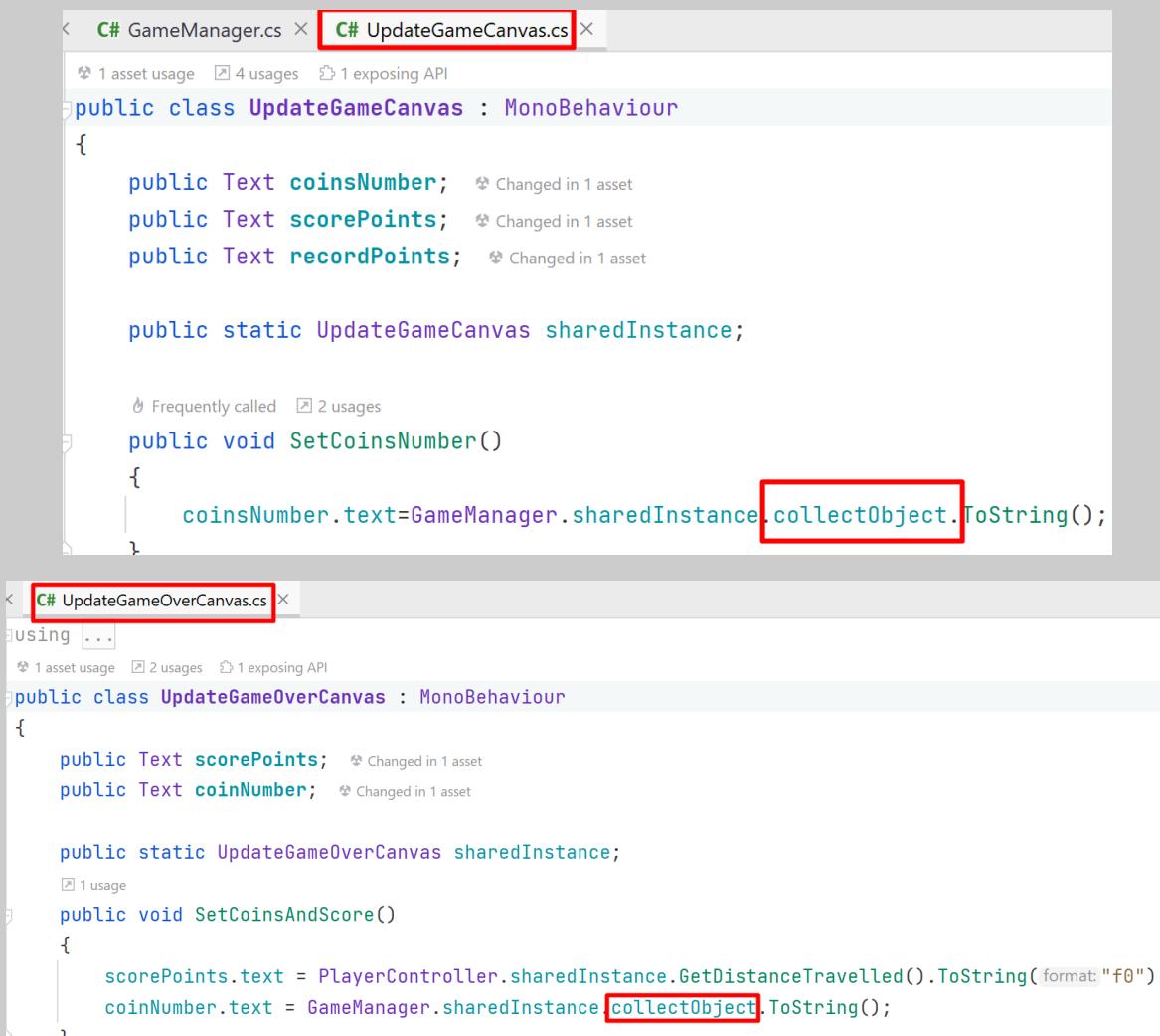
    1 usage

    public void CollectItem(int objectValue)
    {
        collectObject+=objectValue;
        UpdateGameCanvas.sharedInstance.SetCoinsNumber();
    }

    Frequently called 1 usage

    public void StartGame()
    {
        ChangeStateGame(GameState.inTheGame);
        PlayerController.sharedInstance.StartGame();
        LevelGenerator.sharedInstance.GenerateInitialBlocks();
        UpdateGameCanvas.sharedInstance.SetRecordPoints();
        collectObject = 0;
        UpdateGameCanvas.sharedInstance.SetCoinsNumber();
    }
}

```



```

< C# GameManager.cs > < C# UpdateGameCanvas.cs >
  1 asset usage  4 usages  1 exposing API
public class UpdateGameCanvas : MonoBehaviour
{
    public Text coinsNumber;  Changed in 1 asset
    public Text scorePoints;  Changed in 1 asset
    public Text recordPoints;  Changed in 1 asset

    public static UpdateGameCanvas sharedInstance;

    Frequently called  2 usages
    public void SetCoinsNumber()
    {
        coinsNumber.text=GameManager.sharedInstance.collectObject.ToString();
    }
}

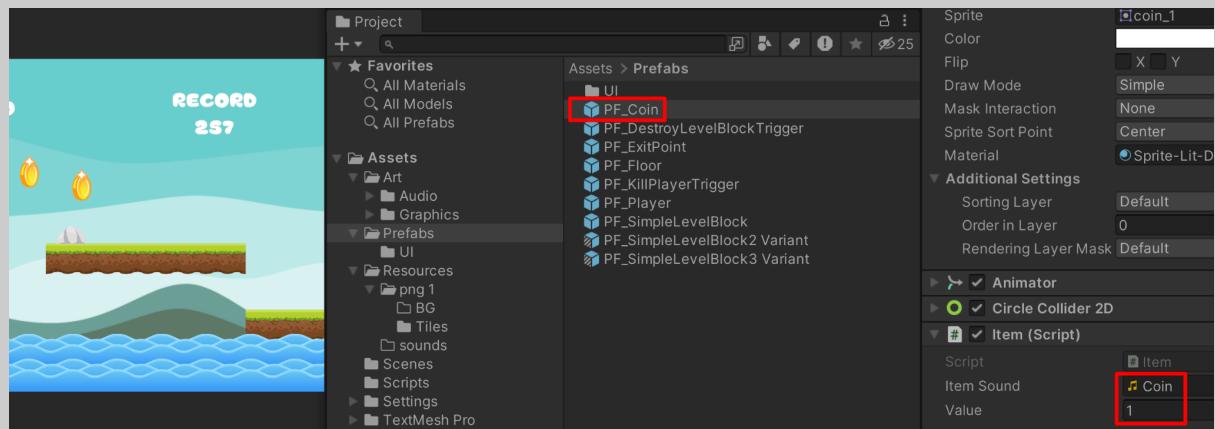
< C# UpdateGameOverCanvas.cs >
using ...
  1 asset usage  2 usages  1 exposing API
public class UpdateGameOverCanvas : MonoBehaviour
{
    public Text scorePoints;  Changed in 1 asset
    public Text coinNumber;  Changed in 1 asset

    public static UpdateGameOverCanvas sharedInstance;

    1 usage
    public void SetCoinsAndScore()
    {
        scorePoints.text = PlayerController.sharedInstance.GetDistanceTravelled().ToString(format:"f0");
        coinNumber.text = GameManager.sharedInstance.collectObject.ToString();
    }
}

```

Recuerda asignar el sonido de la moneda y su valor de 1 al script Item del prefab Coin, y comprueba que todo sigue funcionando correctamente:



Ahora vamos a añadir un nuevo colecciónable, algo que le dé energía al protagonista, por ejemplo “un pez”. La forma más cómoda de seguir añadiendo ítems de diferentes tipos a nuestro juego es clasificando estos colecciónables con un enumerado (enum) y así será muy sencillo de seguir ampliando estos en un futuro:

```

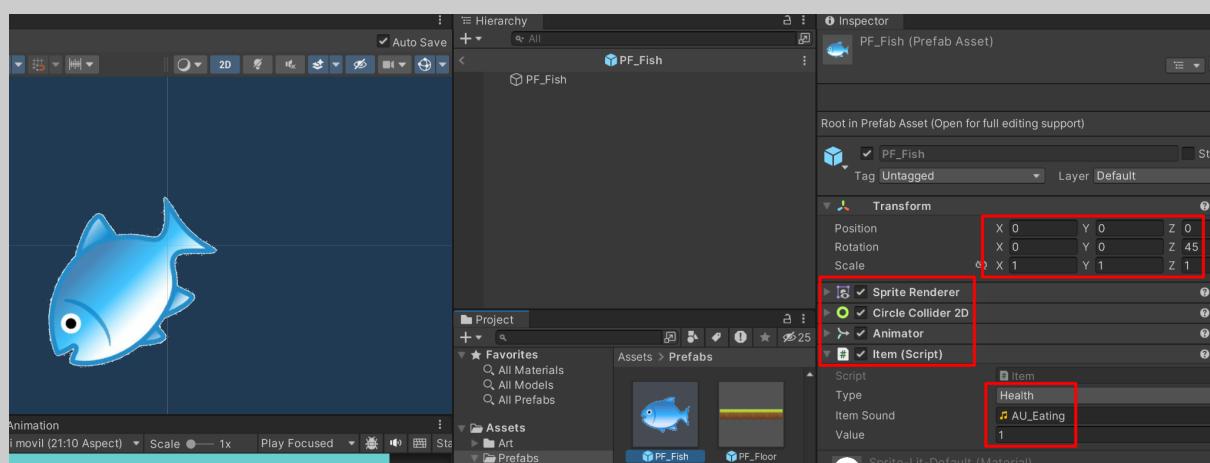
Manager.cs X C# Item.cs X
using UnityEngine;
[3 usages 3 exposing APIs]
public enum ItemType
{
    health,
    money
}
[2 asset usages]
public class Item : MonoBehaviour
{
    public ItemType type; //En el inspector seleccionaré el tipo de ítem
    public AudioClip itemSound; //En vez de coinSound
}

```

Establecemos el tipo de enumerado para el prefab de la moneda:



A continuación vamos a crearnos un prefab para un ítem de tipo “Health” por ejemplo un pez. Este prefab contendrá los elementos que vemos en la siguiente imagen:



Ahora mismo, si probamos el juego, vemos que el comportamiento de los ítems de tipo “money” (como las monedas) y los ítems de tipo “health” (como el pez), es el mismo ya que ambos incrementan el contador de monedas. Deberemos modificar el método Collect de la clase Item para que así tengan efectos distintos sobre el juego, por ejemplo, las monedas seguirán aumentando en uno el contador de monedas y los peces que incrementen en uno una variable “healthPlayer” (que se verá reflejado en un futuro marcador de cansancio/salud). Para ello debemos modificar los scripts Item.cs y el PlayerController.cs:

```
Editor: Item.cs X Editor: DestroyLevelBlockTrigger.cs X Editor: GameManager.cs X Editor: UpdateGameCanvas.cs X Editor: UpdateGameOverCanvas.cs X Editor: CameraFollow.cs X Editor: KillPlayerTrigger.cs X Editor: LevelBlock.cs X Editor: LevelGame.cs X
void Collect()
{
    isCollected = true; //En este juego no lo usamos, pero en otro juego puede sernos de utilidad
    Hide(); //Ocultamos
    //Destroy(gameObject); Ocultar o destruir, a elección del desarrollador
    AudioSource.PlayClipAtPoint(itemSound,gameObject.transform.position);
    switch (type)
    {
        case ItemType.health:
        {
            //Aumentar la salud. Como es algo del jugador (no del juego) lo suyo es hacerlo en el PlayerController
            PlayerController.sharedInstance.CollectHealth(value);
            break;
        }
        case ItemType.money:
        {
            //Además de pasarle un value, a CollectCoin lo renombramos como CollectMoney, para que así sea más general
            GameManager.sharedInstance.CollectMoney(value);
            break;
        }
    }
}
```

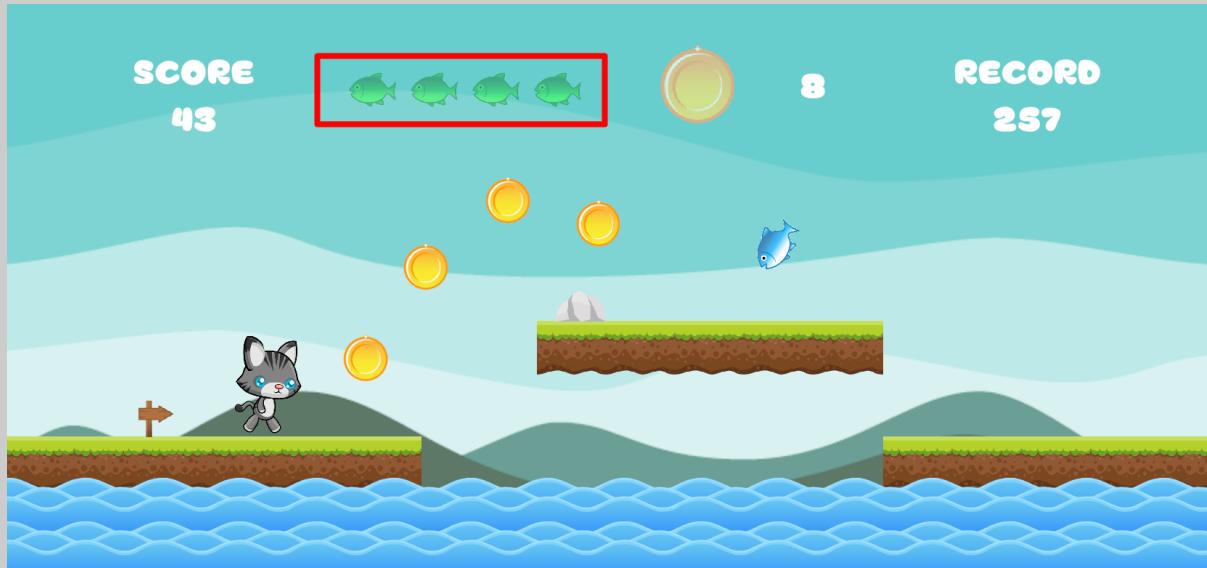
```
nager.cs X C# PlayerController.cs X C# Item.cs X
public class PlayerController : MonoBehaviour
{
    public static PlayerController sharedInstance;
    public float jumpForce = 20.0f; //15
    private Rigidbody2D _rigidBody2D;
    public LayerMask layerMaskGround; // Serializable
    public Animator animator; // Changed in 1 asset
    public float runningSpeed=2.0f; //5
    private Vector3 startPosition;

    private float distanceTravelled = 0;
    private float multiplo = 100;
    private int healthPlayer=6; //Puntos de vida
    [1 usage]

    public void CollectHealth(int objectValue)
    {
        if (healthPlayer < 6) //No se pueden tener más de 6 puntos de vida
        {
            healthPlayer+=objectValue;
            //UpdateGameCanvas.sharedInstance.SetHealthNumber();
        }
    }

    public void Frequently called [1 usage]
    public void StartGame()
    {
        animator.SetBool(name: "isAlive", value: true);
        this.transform.position = startPosition;
        healthPlayer = 6;
    }
}
```

En nuestro juego podríamos implementar que se pierda un punto de vida cuando el conejo impacte contra un obstáculo (rocas, pinchos...), cuando impacte contra un enemigo, cada vez que ejecute 5 saltos (porque se va cansando), etc... Bastaría con decrementar el valor de la variable `healthPlayer` y luego actualizar el Canvas (`UpdateGameCanvas.sharedInstance.SetHealthNumber()`) para que muestre una zanahoria menos:



La lógica de la actualización del Canvas con el número de zanahorias se deja para que lo implemente el alumno en el proyecto final del tema.

2. Ampliando nuestro juego: Utilización de una corutina (IEnumerator)

Explicado de forma sencilla, una corutina es un tipo de función capaz de gobernarse a sí misma, y que sigue su propia lógica de ejecución independiente de la ejecución normal del programa. Para ver cómo funcionan, vamos a implementar una corutina que sea capaz de esperar 5 segundos, restar un punto de vida, esperar 5 segundos, restar otro punto de vida... y así sucesivamente, emulando el cansancio de nuestro protagonista.

Por definición, IEnumerator “*declara una interacción simple a través de una colección no genérica*”, es decir, que va a hacer una iteración de sí misma hasta que alguien decida pararla. Veamos cómo sería su código (con comentarios):

```
s × C# PlayerController.cs × C# DamageTrigger.cs × C# TriggerMovement.cs × C# Enemy.cs × C# LevelGenerator.cs × C# LevelBlock.cs × C# KillTrigger.cs × C# Item.cs ×
private float distanceTravelled = 0;
private float multiplo = 100;
private int healthPlayer; //Puntos de vida
IEnumerator TirePlayer()
{
    while (healthPlayer > 0)
    {
        healthPlayer--;
        Debug.Log(message: "Puntos de vida restantes: " + healthPlayer);
        //UpdateGameCanvas.sharedInstance.SetHealthNumber(); Lo hará el alumno
        yield return new WaitForSeconds(5f); //Duermete durante 5 segundos antes de volver a comprobar la condición del while
    }
    yield return new WaitForSeconds(5f); //si no cumple la condición del while duérmete 5 segundos antes de volver a comprobar la condición
}

public void StartGame()
{
    animator.SetBool(name: "isAlive", value: true);
    this.transform.position = startPosition;
    healthPlayer = 7; //Empiezo en 7 porque la corutina me va a restar 1 cada vez más empezar
    StartCoroutine(methodName: "TirePlayer"); //Comienza la corutina a ejecutarse
}
```

Otra forma mejor:

```
IEnumerable TirePlayer()
{
    while (true)
    {
        yield return new WaitForSeconds(5f);
        if (_healthPlayer > 0)
        {
            _healthPlayer--;
            Debug.Log(message: "Puntos restantes: " + _healthPlayer);
        }
    }
}

public void StartGame()
{
    _healthPlayer = 6;
    StartCoroutine(routine: TirePlayer());
```

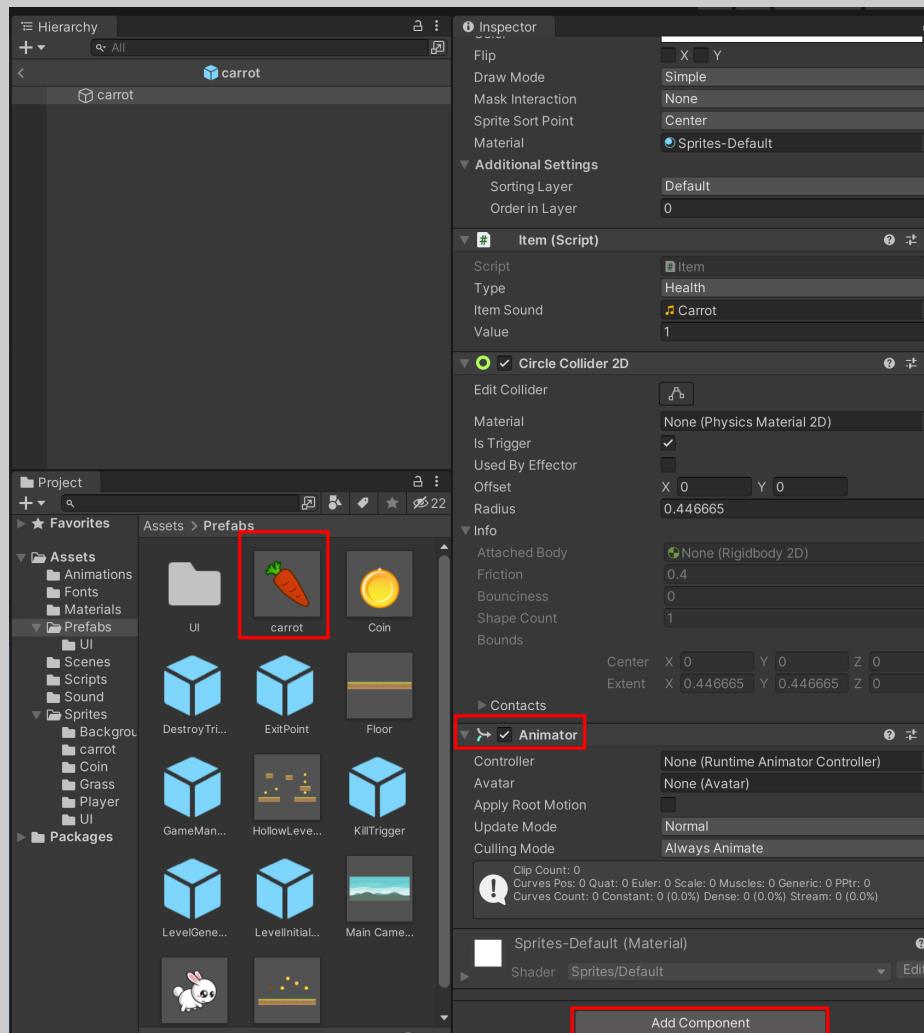
Sería interesante que, cuando te quedes sin comida, suene una pequeña alarma de fondo permanente, el conejito se ponga rojo (por ejemplo) y que su fuerza del salto se reduzca a la mitad; así hasta que logre coger al menos una zanahoria. Se deja para que lo implemente el alumno en el proyecto final del tema.

3. Ampliando nuestro juego: Creación de una pequeña animación

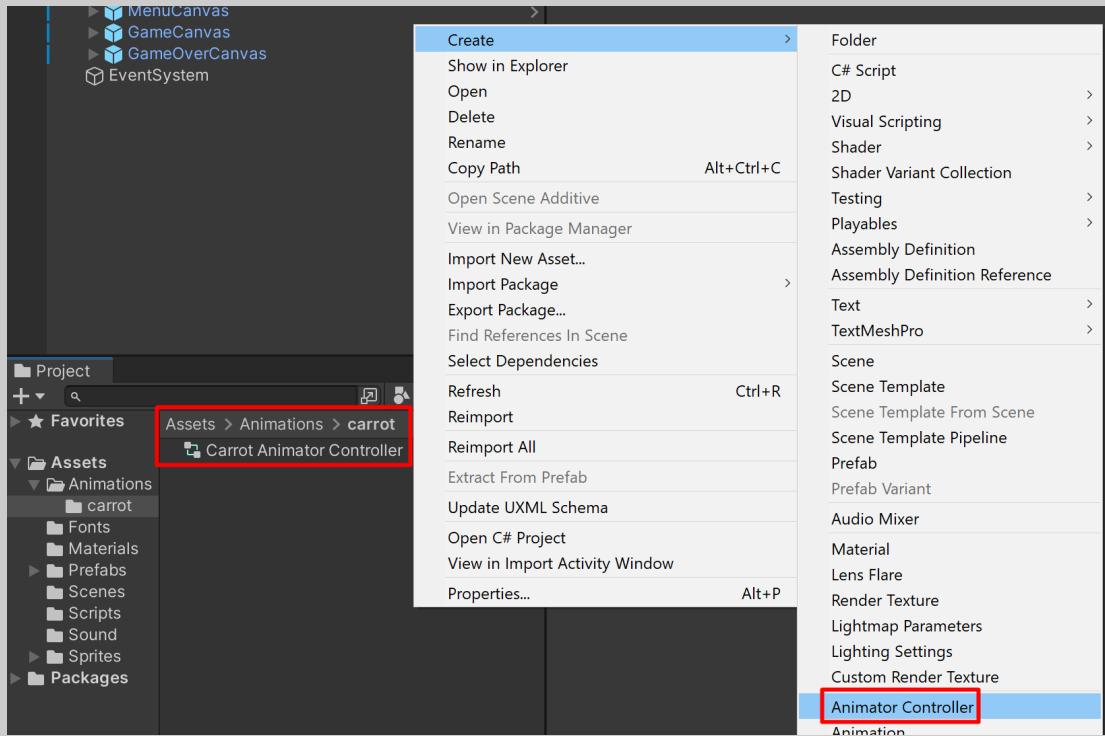
El proceso de animación de los objetos de nuestro juego es muy sencillo, bastará con crear un Animator Controller para uno de estos objetos y, dentro de él, crear las distintas animaciones junto a sus transiciones.

Vamos a hacer un pequeño ejemplo que consistirá en animar un ítem de tipo “health”, por ejemplo una zanahoria, que podría recoger nuestro personaje. Esta animación hará que el ítem sea más llamativo/attractivo para el jugador.

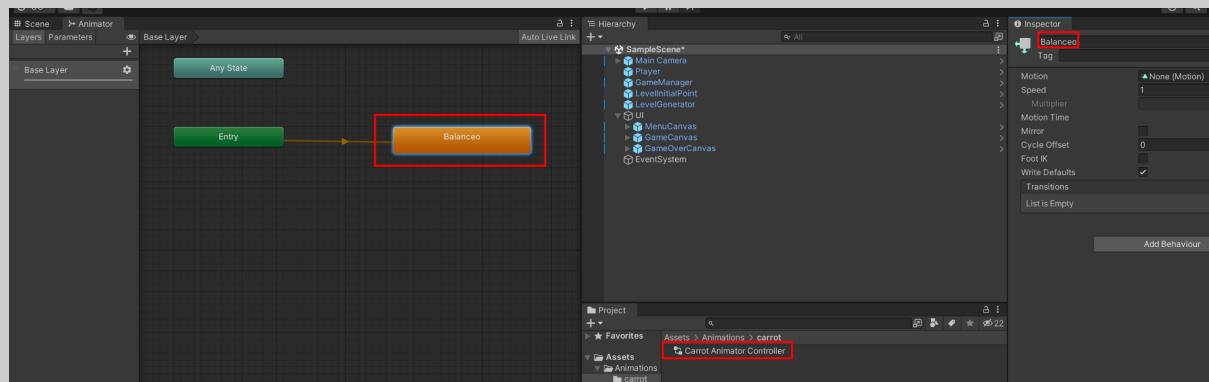
Comenzamos haciendo doble click sobre el Prefab de la zanahoria para así poder editarlo y añadirle un componente Animator:



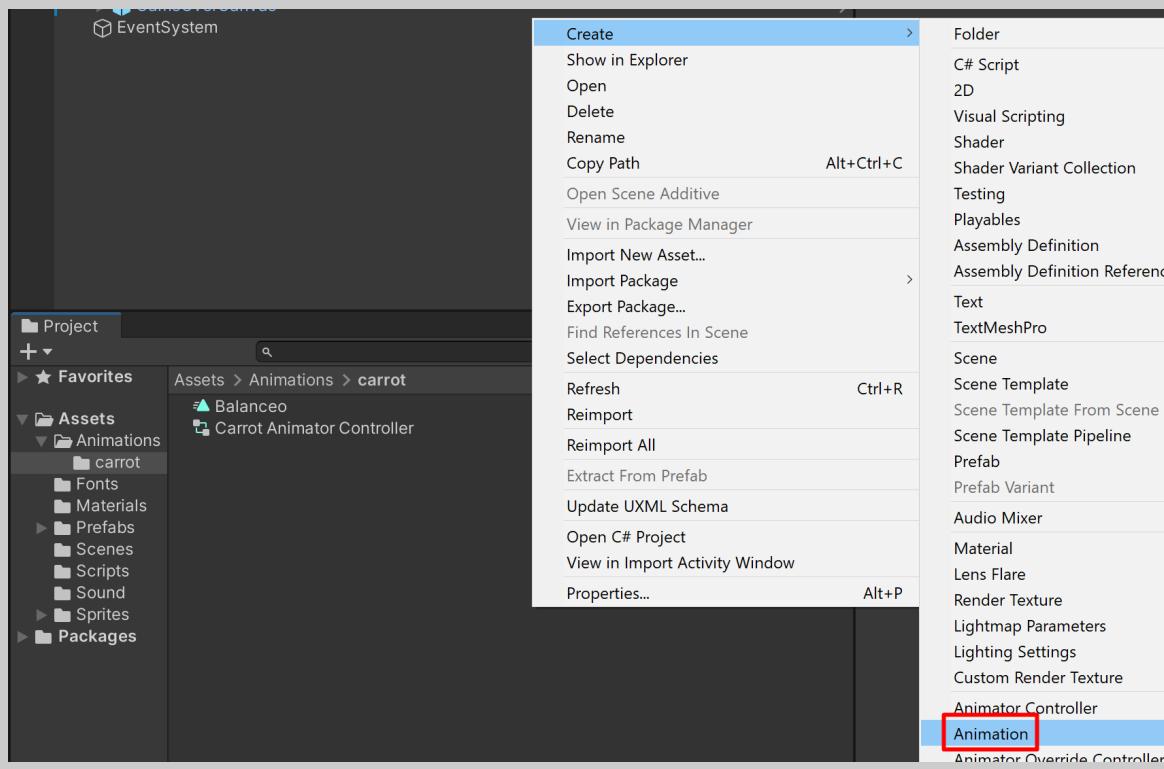
Lo siguiente que tendremos que hacer es crear un Animator Controller dentro de la carpeta Animations>Carrot llamado, por ejemplo “AC_Carrot”



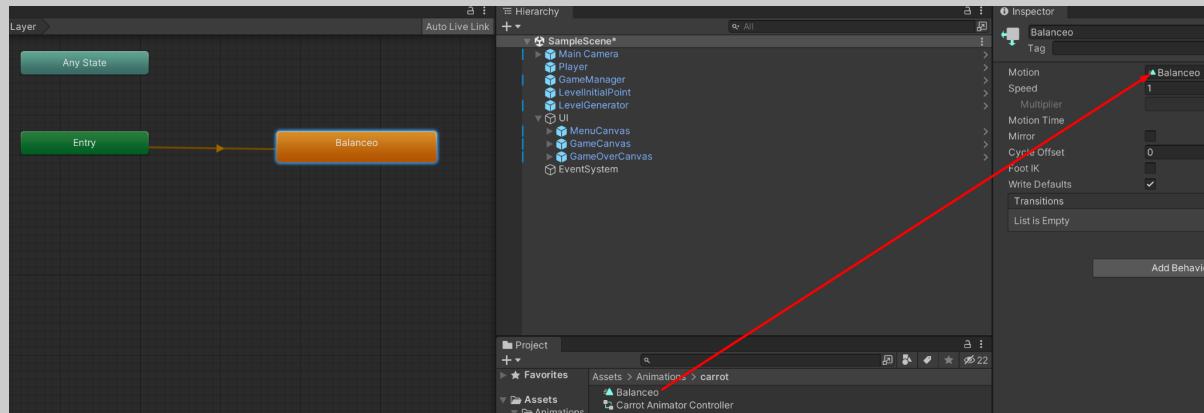
Hacemos doble click sobre el Carrot Animator Controller para que se abra y con el botón derecho del ratón creamos un nuevo estado al que llamaremos “Balanceo” o “Swing”:



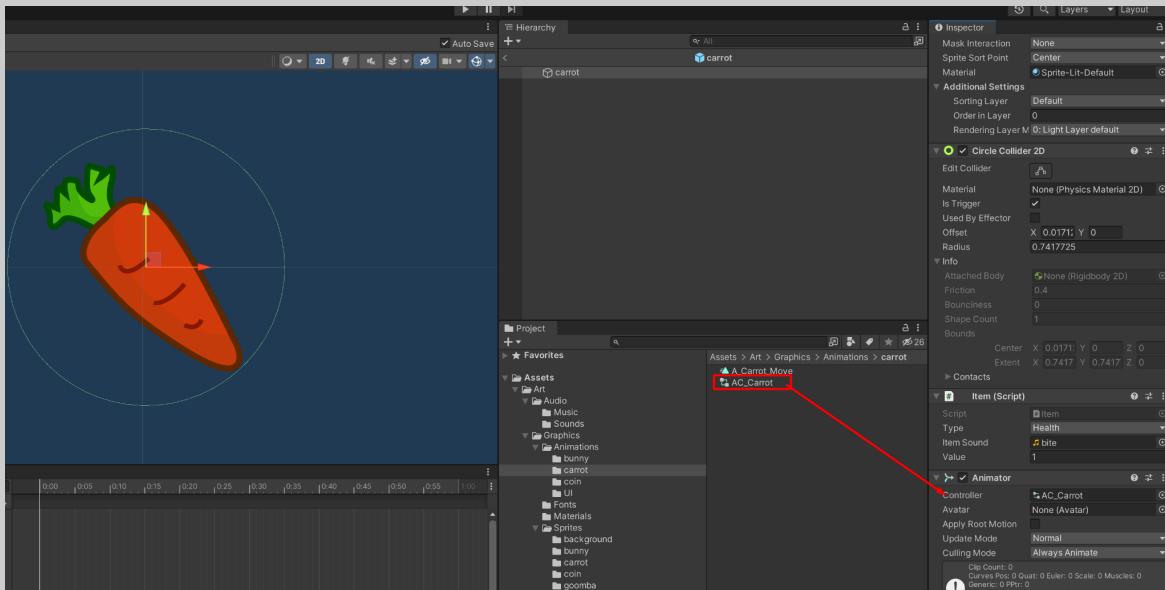
Procedemos a crear la animación del estado Balanceo a la que llamaremos balanceo también:



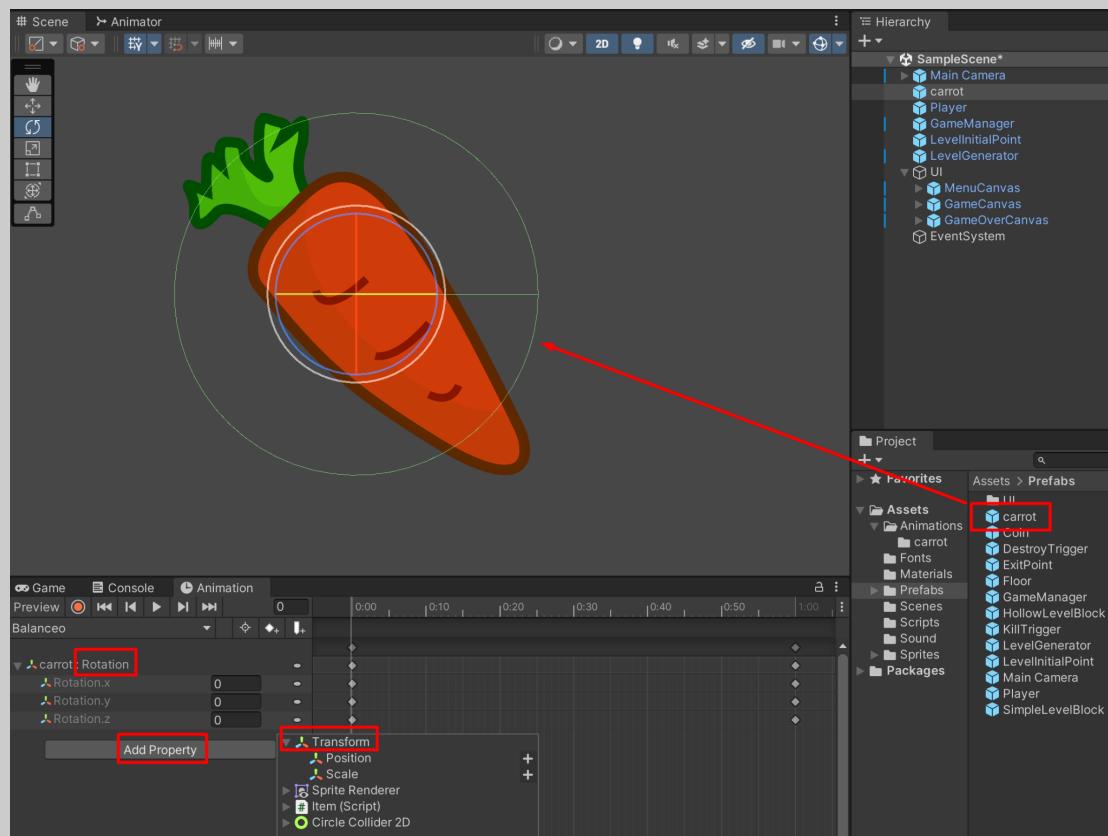
Asignamos la animación creada a su estado correspondiente:



Procedemos a asignar el Animator Controller que creamos anteriormente al componente Animator de la zanahoria:

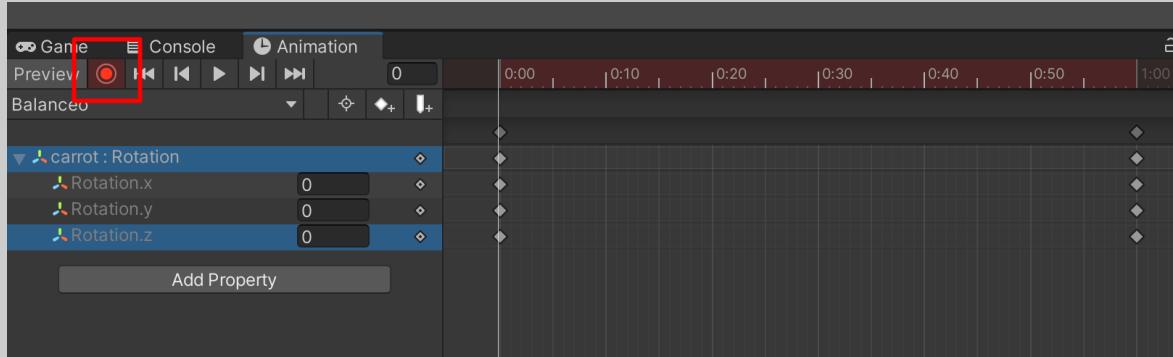


Arrastramos el prefab de la zanahoria a la escena y pulsamos el botón "Add Property" de panel Animation. En nuestro caso, como queremos que haga una pequeña animación de rotación elegiremos Transform>Rotation:



Ahora seguimos los siguientes pasos:

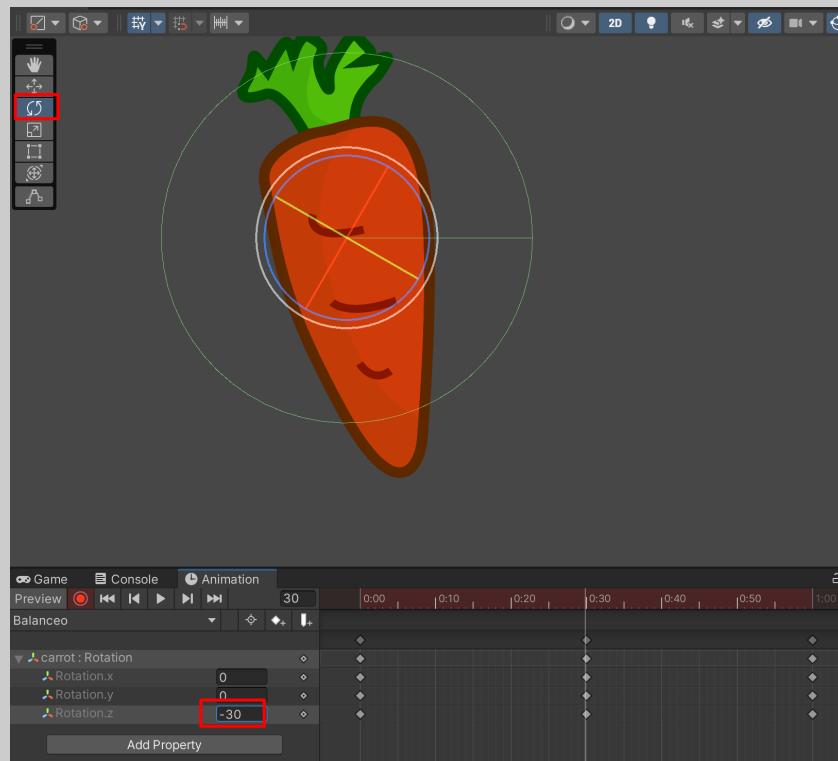
1.- Activamos el “recording mode”:



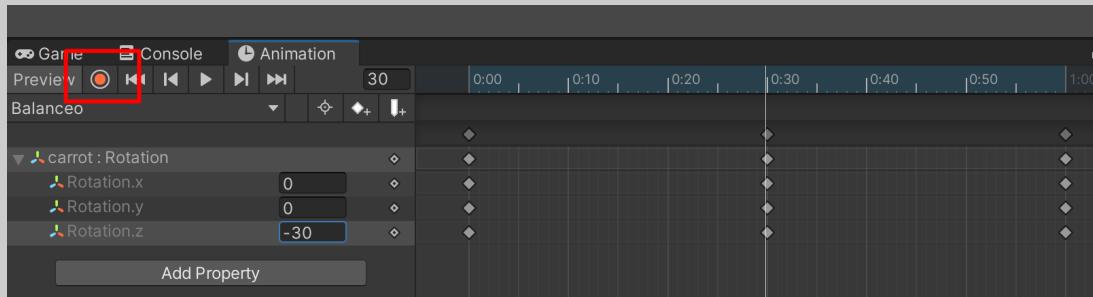
2.- Colocamos la escala de tiempo por la mitad:



3.- Rotamos ligeramente la zanahoria:



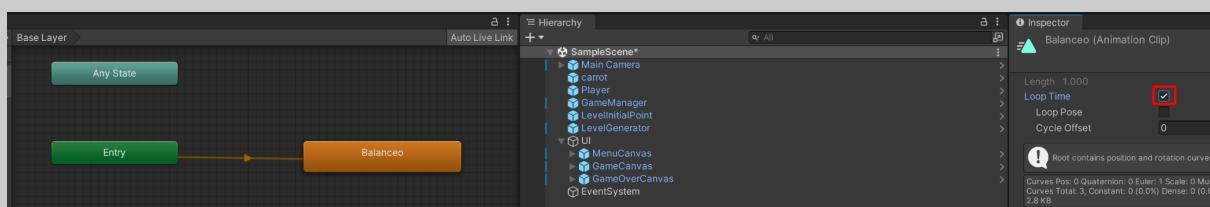
4.- Desactivamos el “recording mode”:



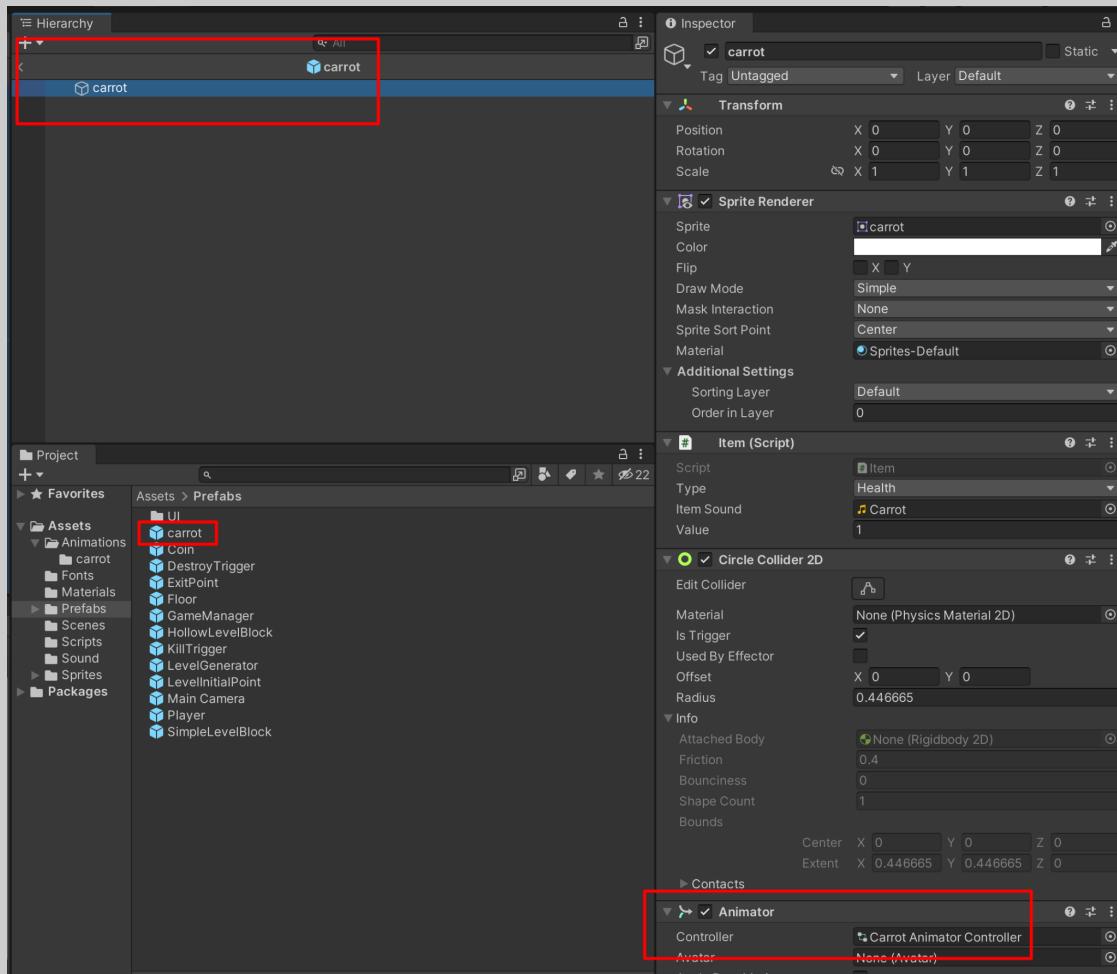
5.- Si pulsamos el botón “play” veremos el resultado:



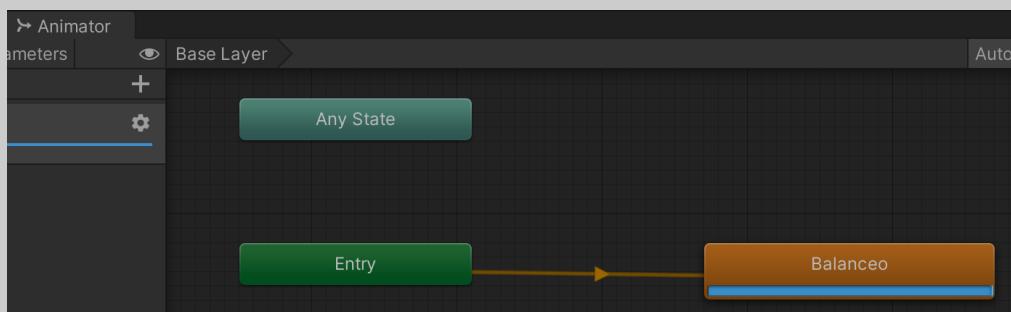
Para que la animación se ejecute indefinidamente durante el juego, pulsaremos en la opción “Loop time”, que aparecerá haciendo doble click sobre el estado “Balanceo”:



Asegurate de que se han quedado guardados todos los cambios en el prefab con su Animator Controller asignado en el componente Animator:



Ejecuta tu juego para comprobar que la rotación ya está activa en todas las zanahorias del juego. Además, podrás ver que la animación se ejecuta indefinidamente (incluso en la máquina de estados del Animator):

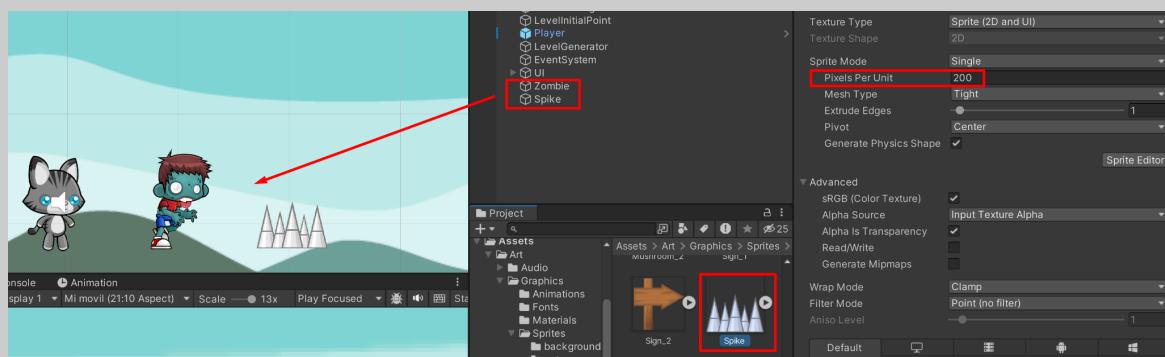


Sería interesante que el alumno implementara en su proyecto alguna plataforma móvil, de manera sencilla, mediante una animación tal y como acabamos de ver.

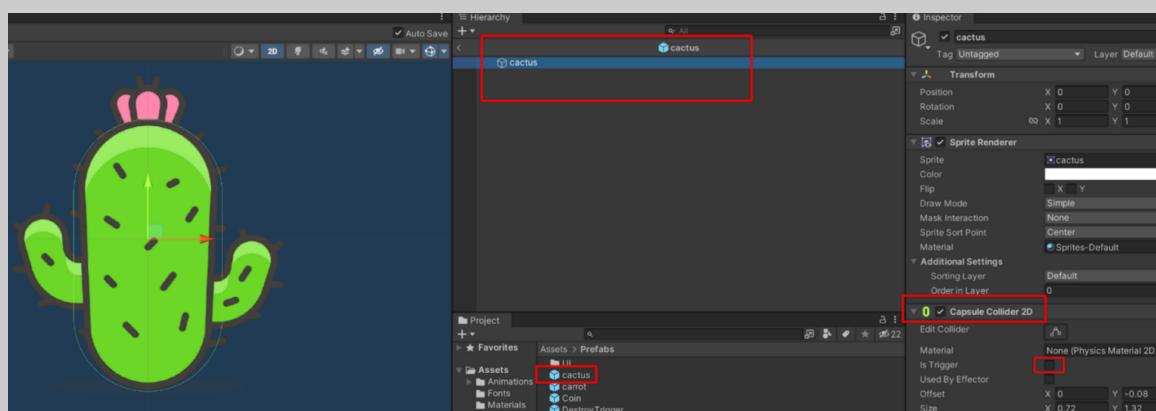
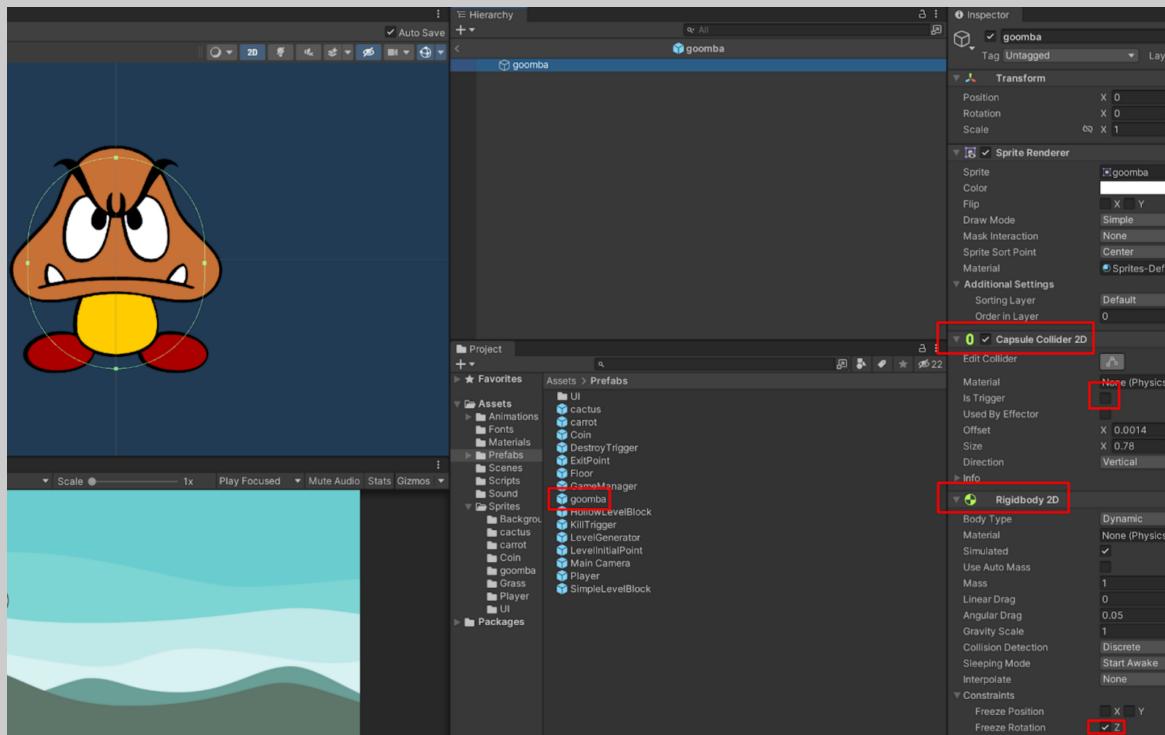
4. Ampliando nuestro juego: Creación de un enemigo

Existe otro tipo de movimiento que podemos implementar que no es en base a las animaciones sino que es en base a triggers. Consiste en un tipo de movimiento sencillo en el que, por ejemplo, un enemigo o una plataforma se mueve en una dirección hasta que choca contra algo y se da la vuelta. Es el movimiento clásico en los juegos de plataformas 2D, como por ejemplo las tortugas o los goombas en el Super Mario Bros.

Comenzaremos introduciendo dos nuevos sprites en nuestro juego: un obstáculo (por ejemplo, unos pinchos metálicos) y un enemigo (por ejemplo, un zombie). Seguramente que sean demasiado grandes debido a su resolución por lo que jugaremos con el atributo “Pixels Per Unit” para ponerlos en un tamaño que consideremos apropiado, por ejemplo así:

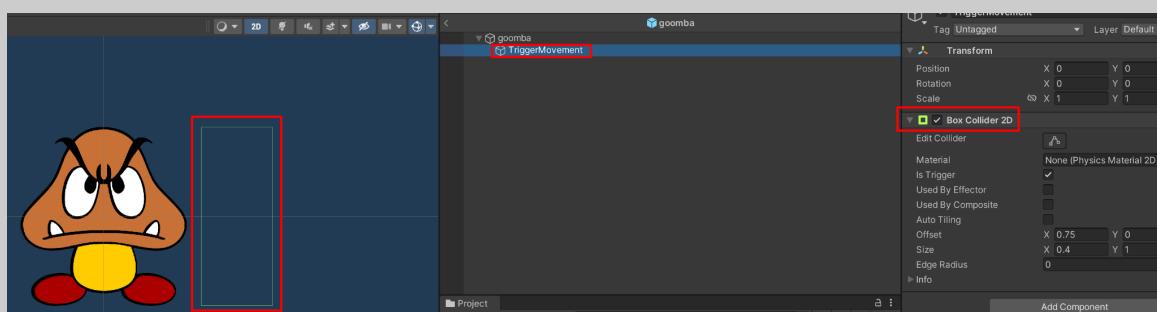


A continuación crearemos los prefabs correspondientes y aprovecharemos para añadirles unos “Capsule Collider 2D” y, además, como goomba se va a mover, le añadiremos a este último un Rigidbody 2D donde poderle aplicar una velocidad posteriormente por código. Ah! No te olvides de congelar la rotación en el eje Z del goomba para ahorrar tiempo de cálculo:



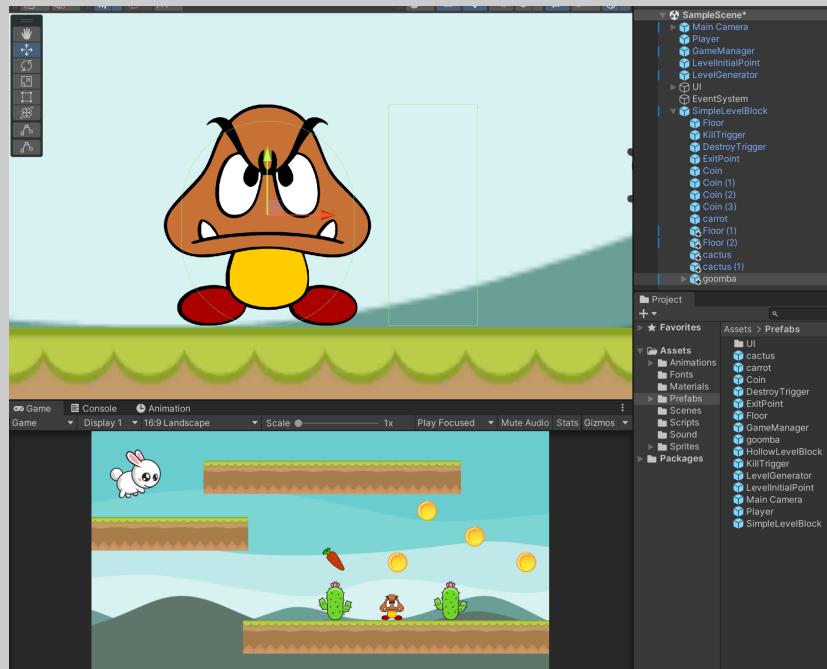
Ahora vamos a añadirle al prefab enemigo un hijo llamado, por ejemplo, "TriggerMovement" que tenga un pequeño Box Collider por delante de él para que, cuando detecte que se va a chocar con algo, se dé la vuelta. (*Una forma más elegante de hacerlo sería lanzando un Raycast, pero como eso ya lo hemos hecho para detectar que el conejito detecte el suelo pues me interesa que lo veais de esta otra forma para que tengais diferentes recursos a la hora de enfrentarnos a un problema*)

La razón de añadir este nuevo BoxCollider2D como un objeto hijo en vez de añadirlo directamente al padre es porque este hijo controlará el comportamiento del padre, por lo que así tendremos el comportamiento por separado y será fácilmente exportable, por ejemplo, a otros enemigos:



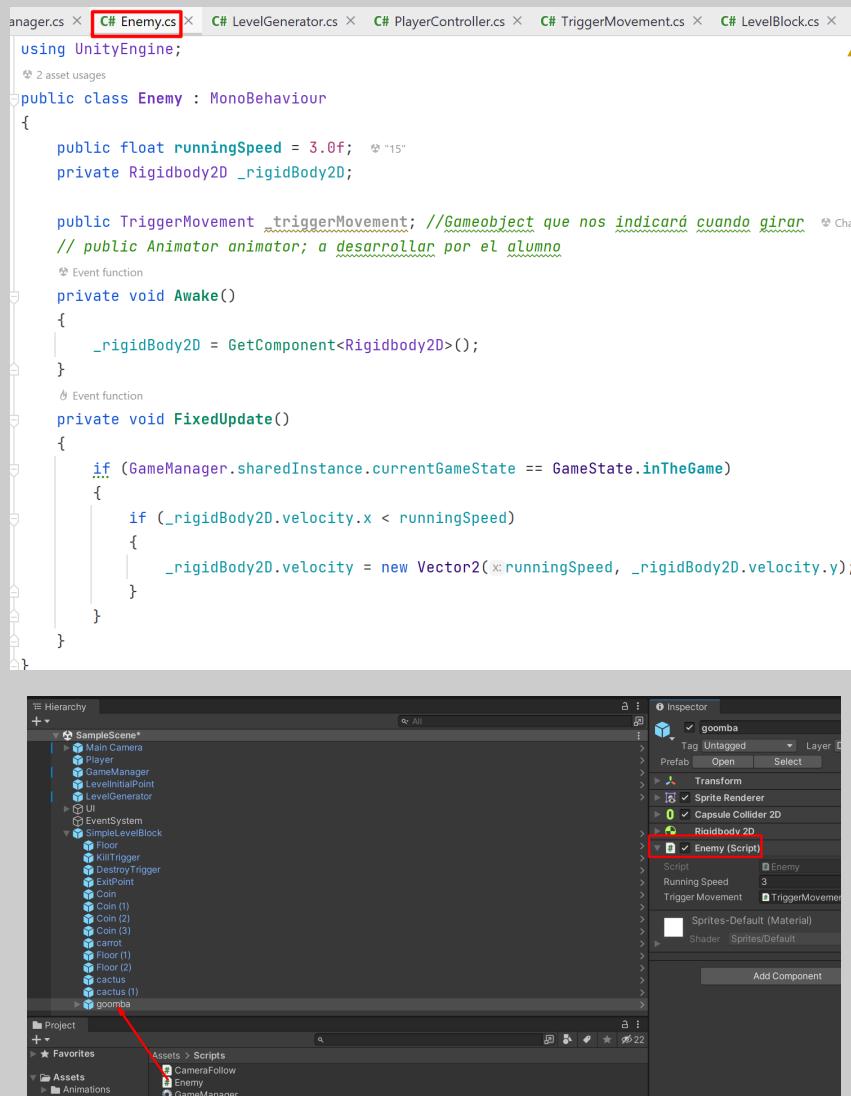
Nuestro enemigo comenzará, por ejemplo, andando hacia la derecha con cierta velocidad, y cuando choque con el cactus rotará 180 grados sobre el eje y (para que así rote también su hijo TriggerMovement con su Collider) y la velocidad pasará a ser negativa para que ande hacia la izquierda.

Preparamos una pequeña escena, donde el conejo está atrapado, para así observar cómodamente el comportamiento del goomba al chocar con los cactus:



Ahora, que tenemos los prefabs preparados, toca implementar este comportamiento mediante código añadiendo dos scripts a nuestro proyecto: uno para el enemigo en sí (para que se mueva, haga daño, etc...) y otro para el trigger que notificará al enemigo que hay un obstáculo cercano para que dé la vuelta.

Comenzaremos creando un Script al que llamaremos, por ejemplo, "Enemy.cs" y se lo asignaremos al enemigo. Fíjate que el enemigo va a desplazarse al igual que hace el Player, por lo que podremos reutilizar parte de su código:



The screenshot shows the Unity Editor interface. At the top, there are several tabs: Manager.cs, C# Enemy.cs (which is currently selected and highlighted with a red box), C# LevelGenerator.cs, C# PlayerController.cs, C# TriggerMovement.cs, and C# LevelBlock.cs. Below the tabs is the code editor window containing the Enemy.cs script. The script defines a public class Enemy that inherits from MonoBehaviour. It includes fields for runningSpeed (set to 3.0f) and _rigidBody2D. It also includes triggerMovement and animator fields, both commented out. The Awake() method initializes _rigidBody2D. The FixedUpdate() method checks if the current game state is 'InTheGame'. If so, it checks if the rigidbody's velocity.x is less than runningSpeed. If true, it sets the rigidbody's velocity to a new Vector2 with x:runningSpeed and y:_rigidBody2D.velocity.y. The bottom part of the screenshot shows the Unity interface with the Hierarchy panel (containing SampleScene*, Main Camera, Player, GameManager, LevelGenerator, UI, EventSystem, SimpleLevelBlock, Floor, Trigger, DestroyTrigger, ExitPoint, Coin, cactus, and goomba), the Inspector panel (showing the goomba object with its components: Transform, Sprite Renderer, Capsule Collider 2D, Rigidbody 2D, and Enemy (Script)), and the Project panel (Assets > Scripts > Enemy).

```

using UnityEngine;

public class Enemy : MonoBehaviour
{
    public float runningSpeed = 3.0f;
    private Rigidbody2D _rigidBody2D;

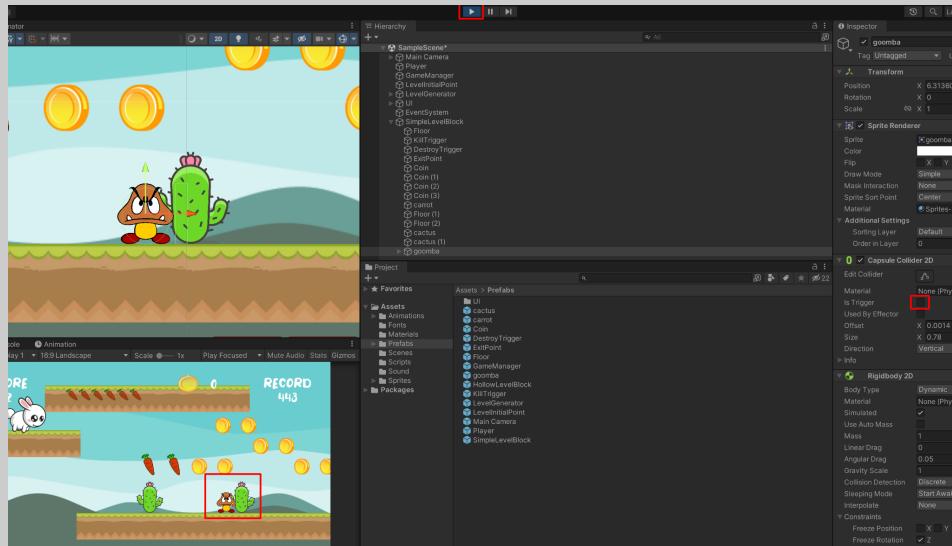
    public TriggerMovement triggerMovement; // GameObject que nos indicará cuando girar
    // public Animator animator; a desarrollar por el alumno

    private void Awake()
    {
        _rigidBody2D = GetComponent<Rigidbody2D>();
    }

    private void FixedUpdate()
    {
        if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
        {
            if (_rigidBody2D.velocity.x < runningSpeed)
            {
                _rigidBody2D.velocity = new Vector2(x:runningSpeed, _rigidBody2D.velocity.y);
            }
        }
    }
}

```

Si ejecutamos el juego veremos como el goomba se desplaza hacia la derecha hasta chocar con el cactus: (recuerda no tener marcado Is Trigger ni en los cactus ni en el goomba)

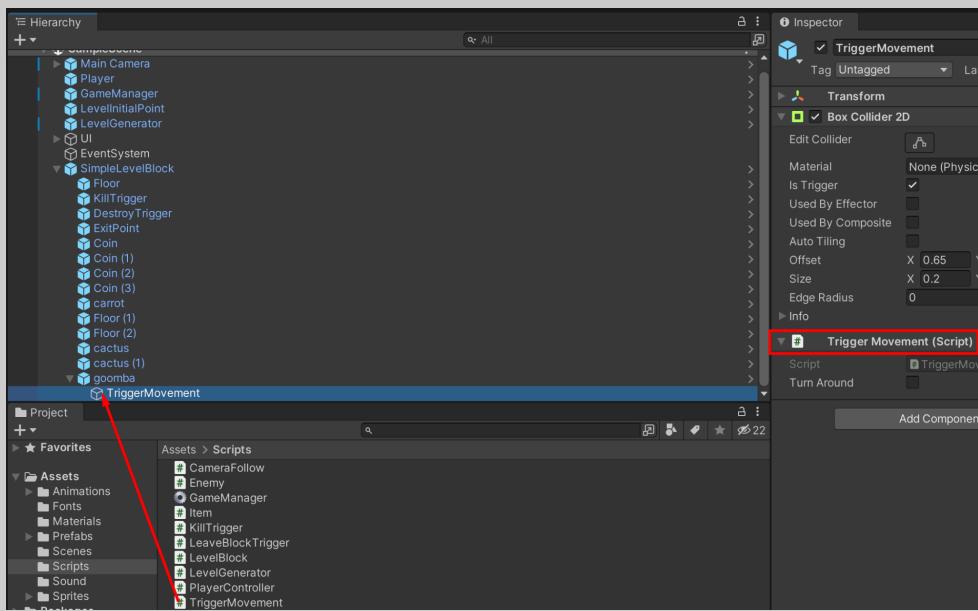


Implementaremos un script llamado, por ejemplo, TriggerMovement, que se lo asignaremos al gameobject TriggerMovement hijo del goomba:

```
manager.cs x C# TriggerMovement.cs x C# Enemy.cs x C# LevelGenerator.cs x C# Plat
using UnityEngine;

public class TriggerMovement : MonoBehaviour
{
    public bool turnAround; //Cambia de estado en cada colisión

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (turnAround)
        {
            turnAround = false;
        }
        else
        {
            turnAround = true;
        }
    }
}
```



Gracias a la variable booleana turnAround del TriggerMovement, el Enemigo sabrá cuándo girar. Para ello vamos a modificar el script Enemy.cs:

```

using UnityEngine;
public class Enemy : MonoBehaviour
{
    public float runningSpeed = 3.0f;
    private Rigidbody2D _rigidBody2D;

    public TriggerMovement _triggerMovement; //Gameobject que nos indicará cuando girar
    // public Animator animator; a desarrollar por el alumno

    private void Awake()
    {
        _rigidBody2D = GetComponent<Rigidbody2D>();
    }

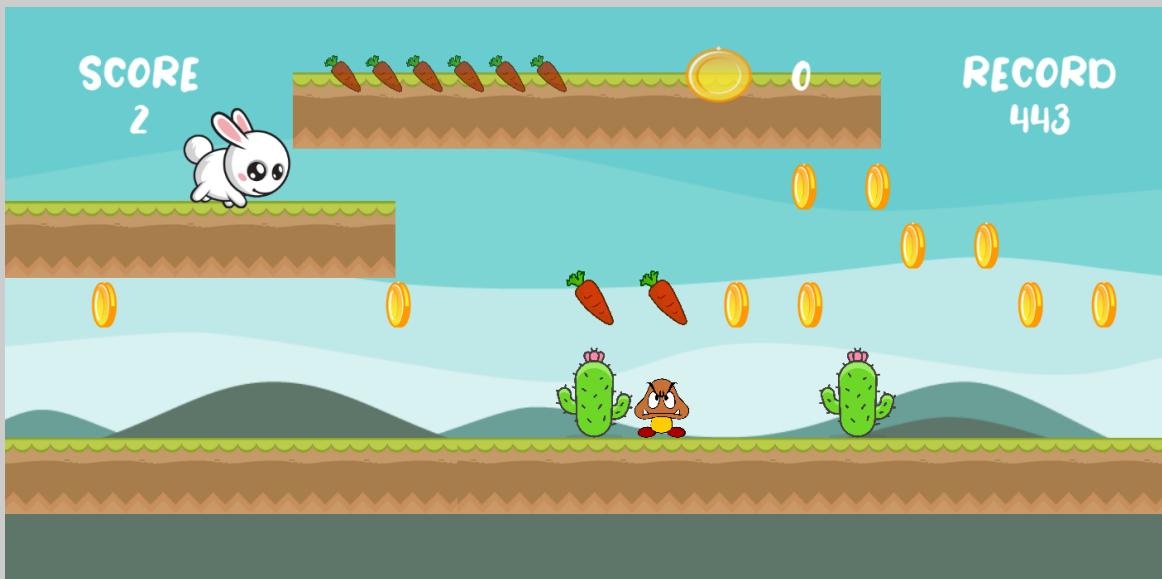
    private void FixedUpdate()
    {
        float currentRunningSpeed = runningSpeed;

        if (_triggerMovement.turnAround == true)
        {
            currentRunningSpeed = -runningSpeed; //Con la velocidad negativa el enemigo se desplazará a la izquierda
            transform.eulerAngles = new Vector3(0, 180f, 0); //giramos 180 grados en el eje y
        }
        else
        {
            transform.eulerAngles = Vector3.zero; //volvemos a los 0 grados en el eje y
        }

        if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
        {
            if (_rigidBody2D.velocity.x < runningSpeed) && (_rigidBody2D.velocity.x > -runningSpeed))
            {
                _rigidBody2D.velocity = new Vector2(currentRunningSpeed, _rigidBody2D.velocity.y);
            }
        }
    }
}

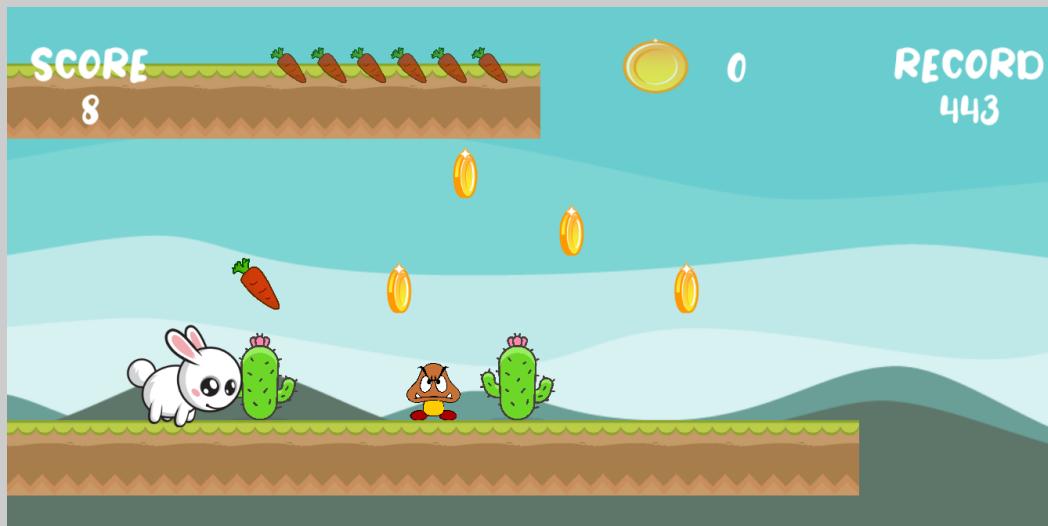
```

Comprobamos que el goomba se mueva de derecha a izquierda entre los cactus:

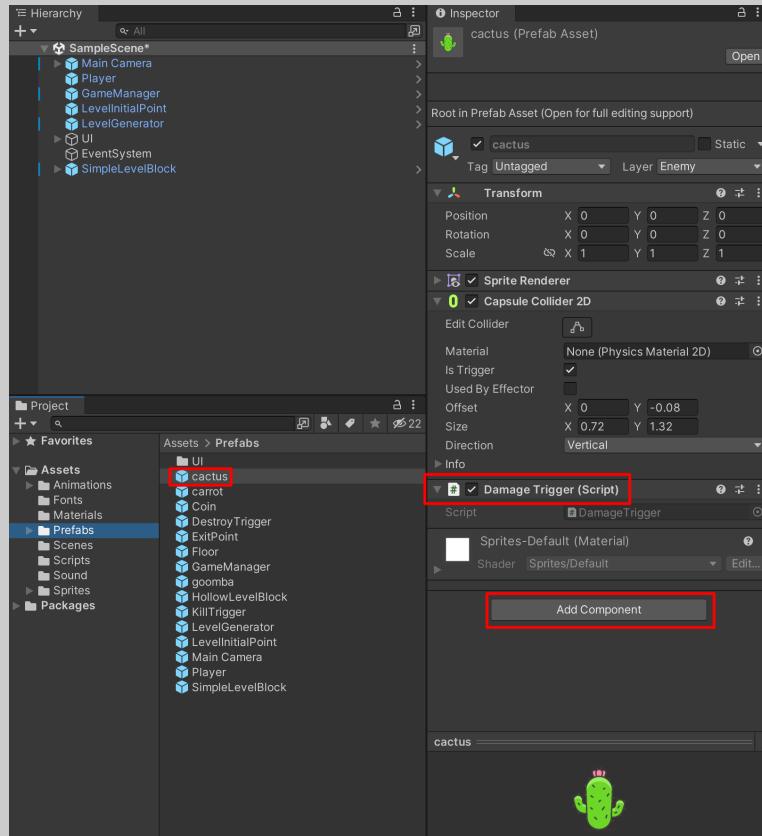


Para terminar programaremos que nuestro conejo al tocar un obstáculo o un enemigo pierda un punto de vida. Además, no me interesa que el conejo al colisionar con un cactus o enemigo pierda velocidad, sino que solo reciba daño, y que esté uno o dos segundos en estado invulnerable.

Comenzaremos por los cactus. Ahora mismo, si no tuviésemos marcado el atributo “is Trigger” del Collider del obstáculo, si chocara contra él se quedaría así:



Procedemos a crear un nuevo script llamado DamageTrigger y se lo asignamos al prefab del cactus. Además, si no lo tuviésemos marcado, activaremos el atributo “Is Trigger” en Collider.



Implementamos el script:



The screenshot shows a Unity code editor with the tab bar at the top containing several scripts: Manager.cs, C# DamageTrigger.cs (highlighted with a red box), C# TriggerMovement.cs, C# Enemy.cs, and C# LevelGe. The main code area displays the following C# script:

```
using UnityEngine;

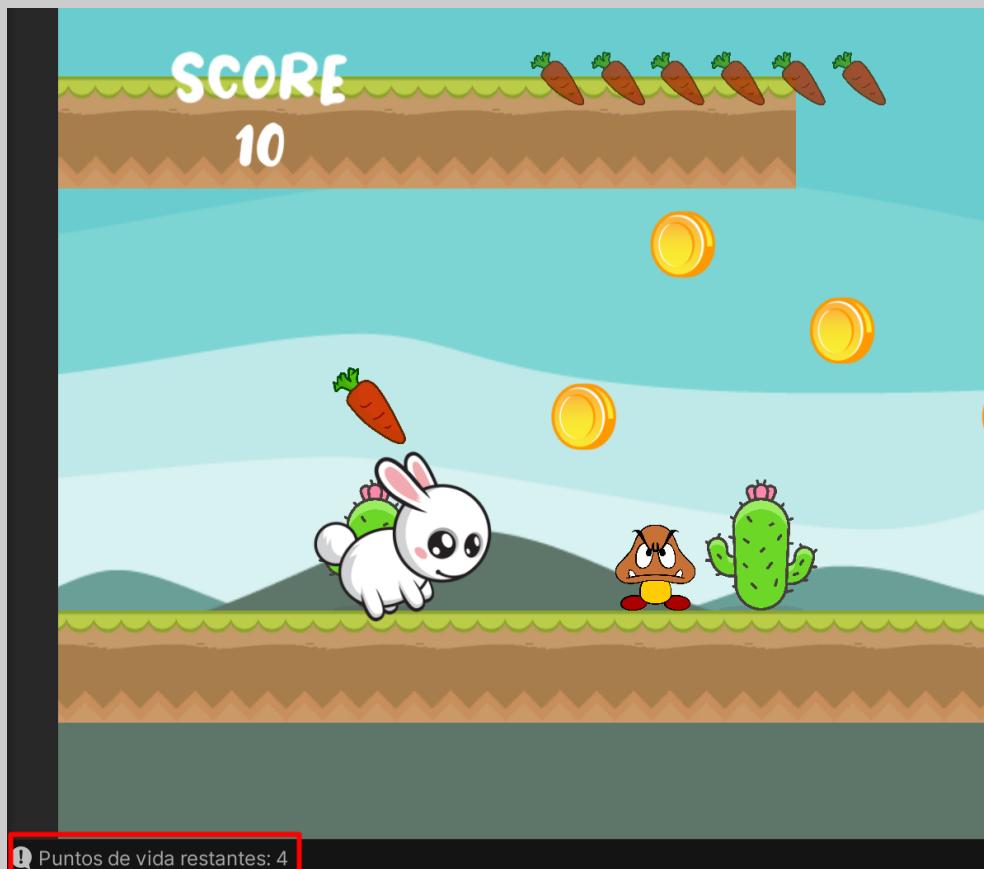
public class DamageTrigger : MonoBehaviour
{
    //public AudioClip hit; lo hará el alumno
    [Event function]
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.tag == "Player")
        {
            // AudioSource.PlayClipAtPoint(hit, transform.position);
            PlayerController.sharedInstance.DamagePlayer();
        }
    }
}
```

Ahora añadimos el método DamagePlayer al PlayerController:

```
.cs × C# PlayerController.cs × C# DamageTrigger.cs × C# TriggerMovement.cs × C# Enemy.cs × C# Level.cs
private float distanceTravelled = 0;
private float multiplo = 100;
private int healthPlayer; //Puntos de vida

public void DamagePlayer()
{
    //animator.SetBool("isDamaging",true); Lo hará el alumno
    if (healthPlayer > 0)
    {
        healthPlayer--;
        Debug.Log(message: "Puntos de vida restantes: "+healthPlayer);
        //UpdateGameCanvas.sharedInstance.SetHealthNumber(); Lo hará el alumno
    }
}
```

Probamos y vemos que funciona correctamente, lo único que perdemos primero uno y luego otro punto de vida, ya que aún no hemos implementado la invulnerabilidad tras haber recibido daño:



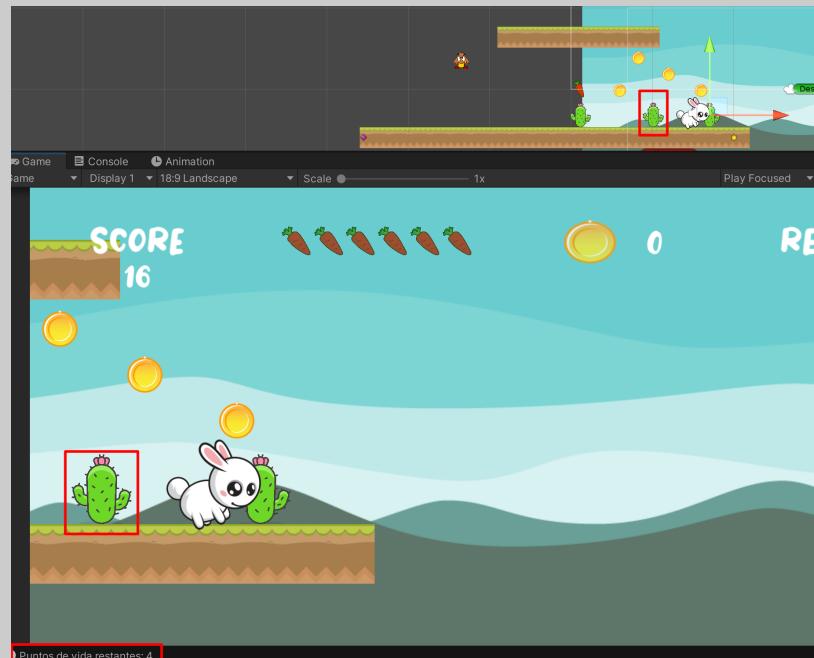
Vamos a implementar que nuestro personaje se haga invulnerable durante un segundo y medio al recibir daño. La forma más sencilla que tendríamos es utilizando el método Invoke, (aunque también se podría hacer mediante una corutina) vamos a ello:

```
.cs × C# PlayerController.cs × C# DamageTrigger.cs × C# TriggerMovement.cs × C# Enemy.cs × C# LevelGene
private float distanceTravelled = 0;
private float multiplo = 100;
private int healthPlayer; //Puntos de vida
private bool invulnerability;

[1 usage]
public void DamagePlayer()
{
    //animator.SetBool("isDamaging",true); Lo hará el alumno
    if (invulnerability == false)
    {
        if (healthPlayer > 0)
        {
            healthPlayer--;
            invulnerability = true;
            Debug.Log(message:"Puntos de vida restantes: " + healthPlayer);
            //UpdateGameCanvas.sharedInstance.SetHealthNumber(); Lo hará el alumno
            Invoke(methodName: "DelayInvulnerability", time: 1.5f);
        }
    }
}

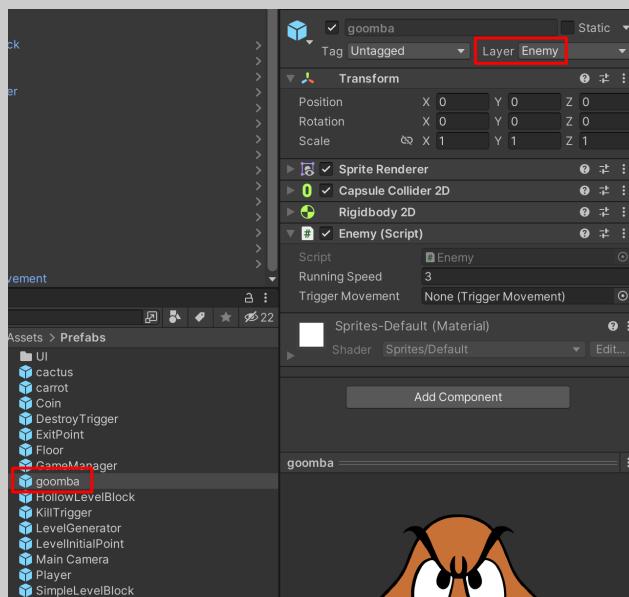
[1 usage]
public void DelayInvulnerability()
{
    invulnerability = false;
}
```

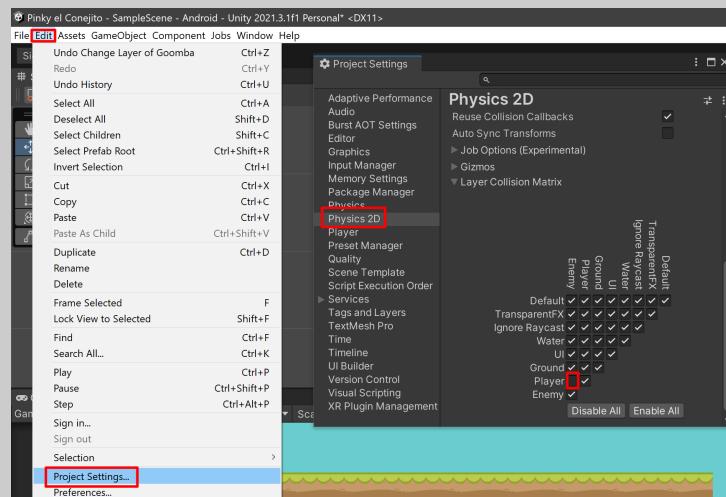
Fíjate, comenzamos con 6 puntos de vida y me hacen daño el primer y el tercer cactus, el segundo no pues estaba en estado invulnerable:



Quedan pendientes, para un posterior desarrollo por parte del alumno en su proyecto, las siguientes tareas:

- El sonido cuando impactamos con un obstáculo.
 - La animación del protagonista cuando impacta con el obstáculo (puedes reutilizar la del humo)
 - La actualización del canvas, perdiendo un punto de vida cuando impacta con el cactus.
 - La animación del enemigo cuando impacta con el jugador.
 - Al igual que hemos hecho con el obstáculo, implementamos el daño al impactar con el enemigo. Como ocurría con el cactus, el enemigo no nos debe frenar, lo atravesaremos pero sufriremos daño con su correspondiente sonido, animación, pérdida de punto de vida y actualización del canvas.
- Pista: Hazlo como creas oportuno, pero es posible que necesites hacer algo así si lo haces como yo tengo en mente:*



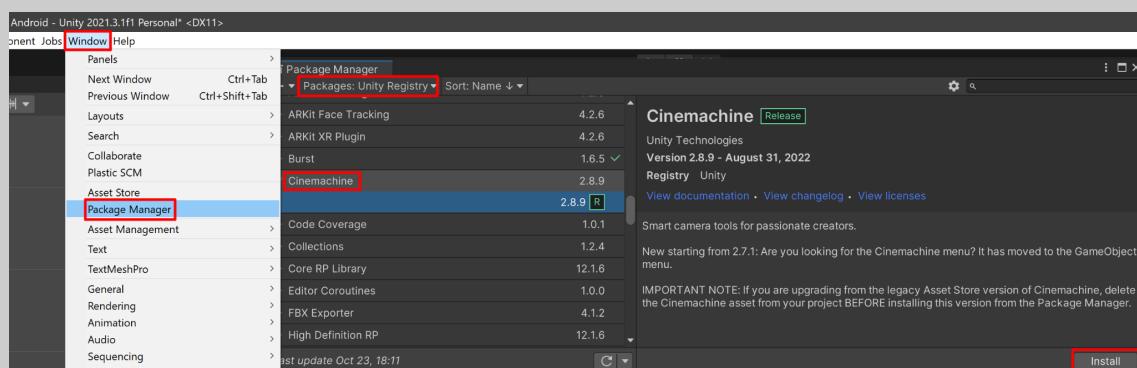


5. Ampliando nuestro juego: Cinemachine

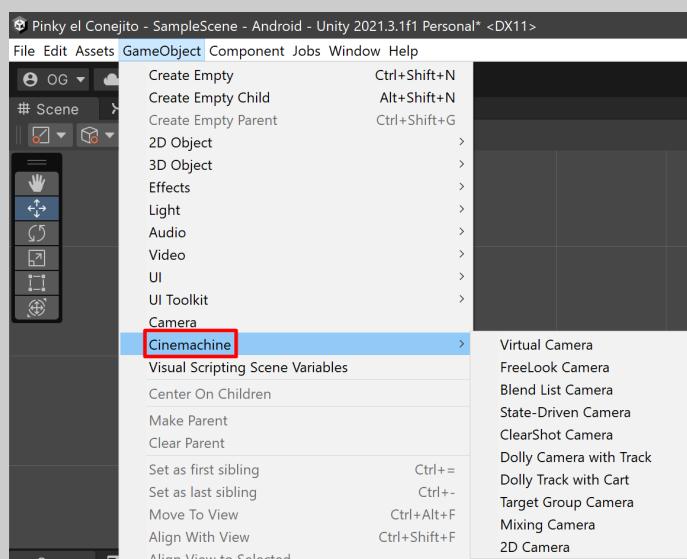
Hay muchas formas de conseguir que la cámara del juego siga a nuestro protagonista. La más sencilla de todas sería colocando la cámara como hija del protagonista y, aunque esta forma no se recomienda para proyectos serios debido a la brusquedad del movimiento (scroll) que va a tener la cámara, sí que nos puede servir de forma temporal y para hacer pruebas, hasta que implementemos una mejor forma.

Otra forma de hacerlo más elegante y sin necesidad de añadir nuevos componentes a nuestro proyecto sería, como hemos visto en la hoja 14 del documento [402](#), es utilizando la función Vector3.SmoothDamp, lo que nos permite hacer un seguimiento del protagonista más suave, agradable al jugador y sin tirones ni cambios bruscos.

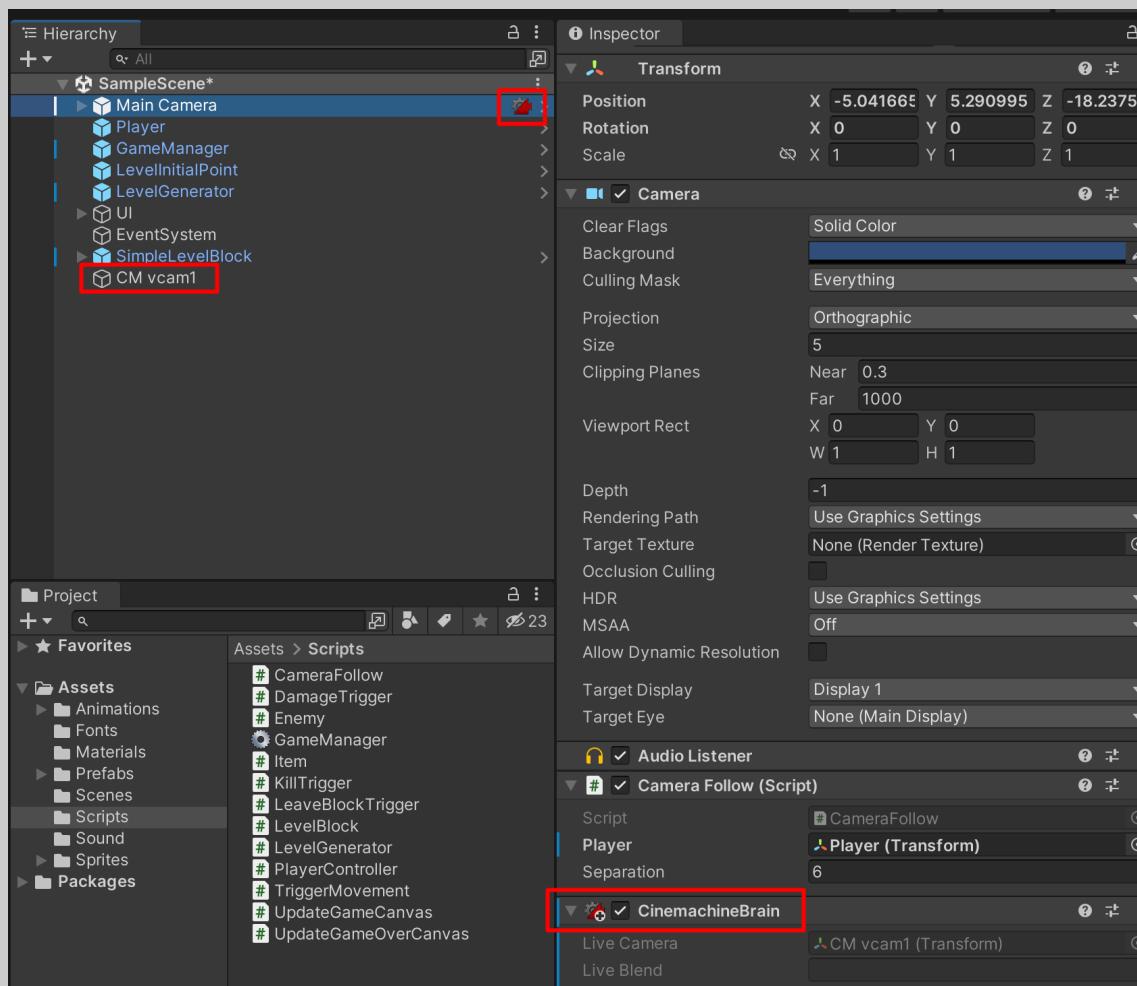
No obstante, la forma “profesional” de hacerlo y que nos ayudará a crear cualquier tipo de cámara para cualquier tipo de juego, es agregando a nuestro proyecto un paquete que recibe el nombre de Cinemachine:



Una vez instalado el paquete veremos que nuestro menú de Unity “GameObject” ha cambiado añadiendo un nuevo componente:

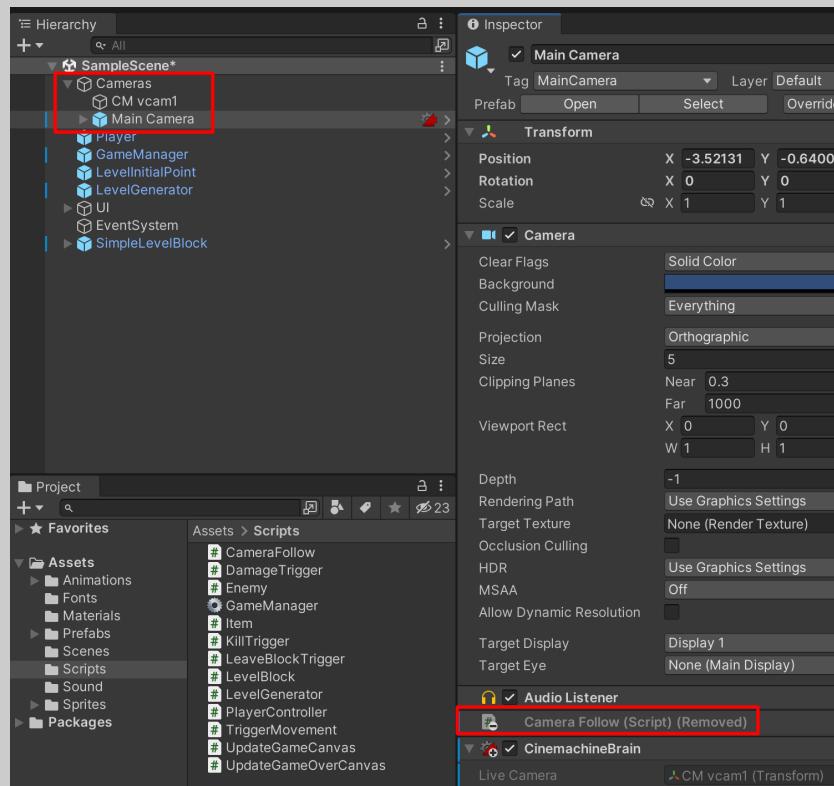


Vamos a seleccionar GameObject>Cinemachine>2D Camera y fíjate lo que ha cambiado en el proyecto: Se le ha añadido automáticamente un “cerebro” CinemachineBrain a nuestra cámara, así como se ha creado una VirtualCamera “CM vcam1” en nuestra jerarquía:

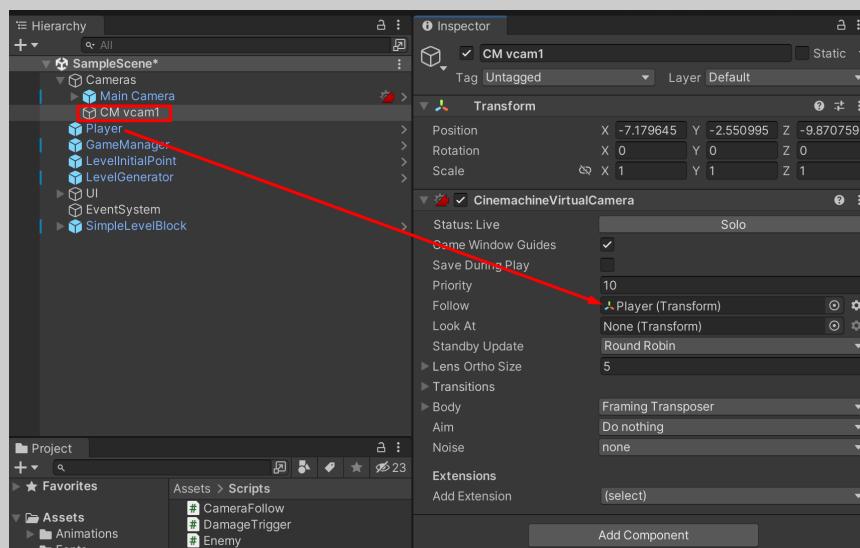


La forma de trabajar con “Cinemachine” consiste en crear varias cámaras virtuales con las que el desarrollador, por ejemplo, podría distribuir por el escenario para que Cinemachine se encargue de cambiar suavemente entre ellas. Además, estas cámaras virtuales son muy flexibles permitiendo cambiar su comportamiento, su tamaño, ángulo...

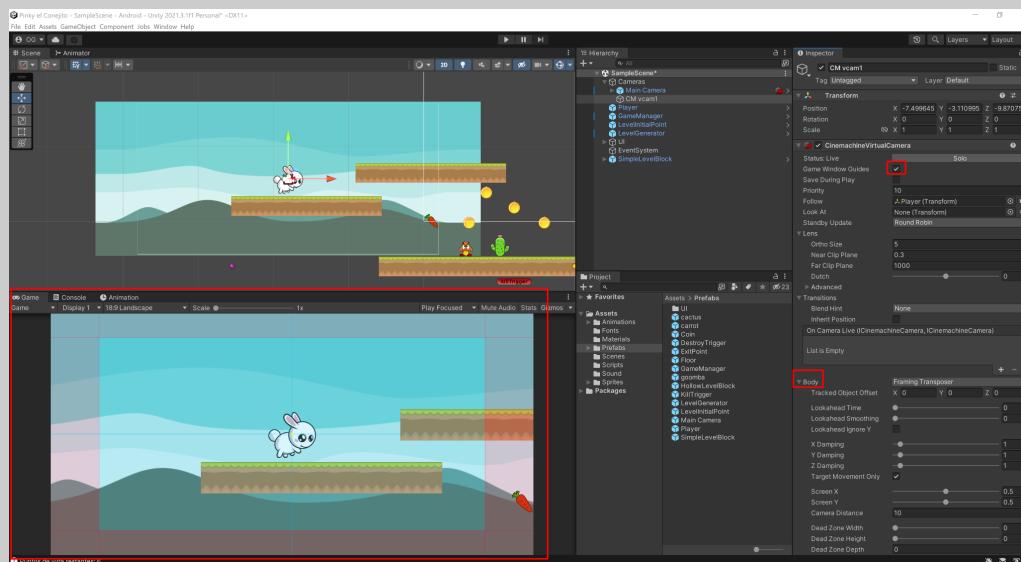
En el caso que nos ocupa, será suficiente con una sola cámara virtual, concretamente utilizaremos la que se nos ha creado automáticamente llamada "Virtual Camera". Para organizar nuestro proyecto vamos a crear un Gameobject vacío llamado "Cameras" y dentro meteremos la Main Camera (ya de paso, desactiva el script "Camera Follow") y la cámara virtual "CM vcam1":



Comenzaremos indicando a la cámara virtual que siga a nuestro personaje. Bastará con arrastrar el gameobject Player al atributo Follow de la cámara virtual:

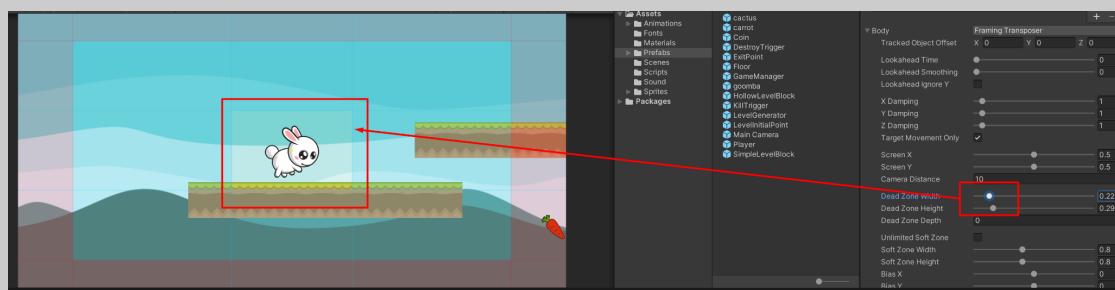


Vamos a activar las “Game Window Guides” y a desplegar el atributo Body. Verás como en la vista Game han aparecido unas guías roja, otra azul, y tendremos una blanca también en el centro. Cada una de estas guías van a ser la manera de indicar a nuestra cámara virtual el tipo de seguimiento que queremos sobre nuestro personaje.



Podremos configurar las opciones del seguimiento a través de las variables que contiene el atributo Body: tiempo en que adelanta al personaje (lookahead time), cantidad de suavizado (lookahead smoothing), que sea más lento el seguimiento (X Damping), etc...

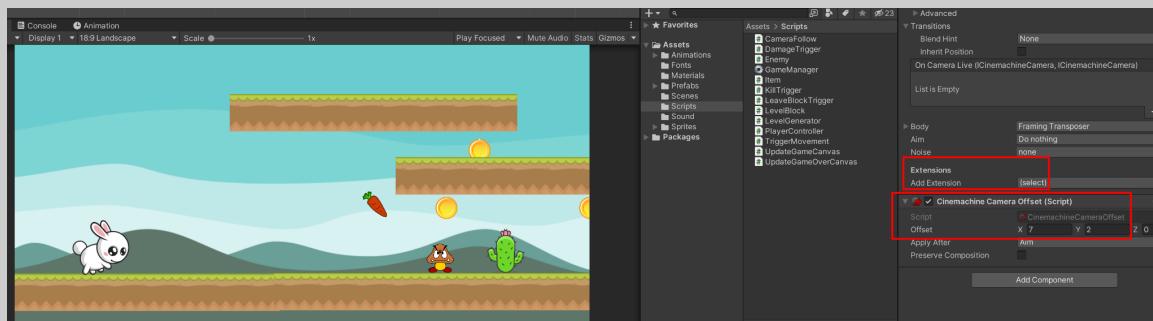
De entre ellas, es muy interesante la de configurar una zona muerta indicando su alto y ancho (dead zone Width/height) qué es una zona donde, si está nuestro protagonista, la cámara no se mueve y le da un poco de tranquilidad al jugador y a las mecánicas:



En nuestro caso, como es un Infinity Runner no necesitaremos ninguna zona muerta así que lo dejaremos en 0.

También es muy curioso el efecto de la cámara corriendo detrás del personaje si le hemos puesto un X Damping muy alto y nos aproximamos a la zona roja de las guías. También podemos configurar el tamaño de esta zona roja modificando las variables Soft Zone Width/Height.

Realmente, para nuestro juego, las opciones que trae por defecto son perfectas, solo bastará con ponerle un suavizado a la cámara (lookahead smoothing), por ejemplo, de 10 y ya casi estaría listo. Digo “casi” porque nos falta añadirle un pequeño desplazamiento (offset) a la cámara para que el personaje no esté centrado en la pantalla, sino hacia la derecha y ligeramente hacia arriba. Para arreglarlo necesitaremos de una de las extensiones incluidas en Cinemachine que deberemos añadir para poder utilizarla, y se llama “Cinemachine Camera Offset”:

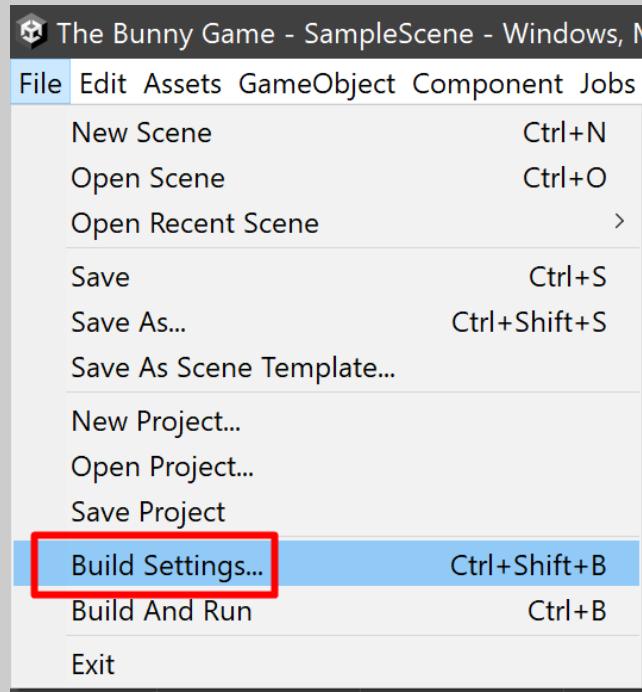


Apenas hemos rozado la superficie de todas las funcionalidades que nos ofrece Cinemachine. Podríamos hacer cosas muy interesantes como por ejemplo tener otra cámara virtual asociada a una animación para que por ejemplo se cambie a ella cuando recibamos un impacto y que esté más cerca del personaje, lo que daría como resultado un efecto de zoom al pasar de una cámara a otra. También nos podría parecer más interesante que la cámara haga una pequeña vibración cuando el personaje choque con un obstáculo, o cuando se quede sin zanahorias haga un efecto de “ruido”.... Más información:

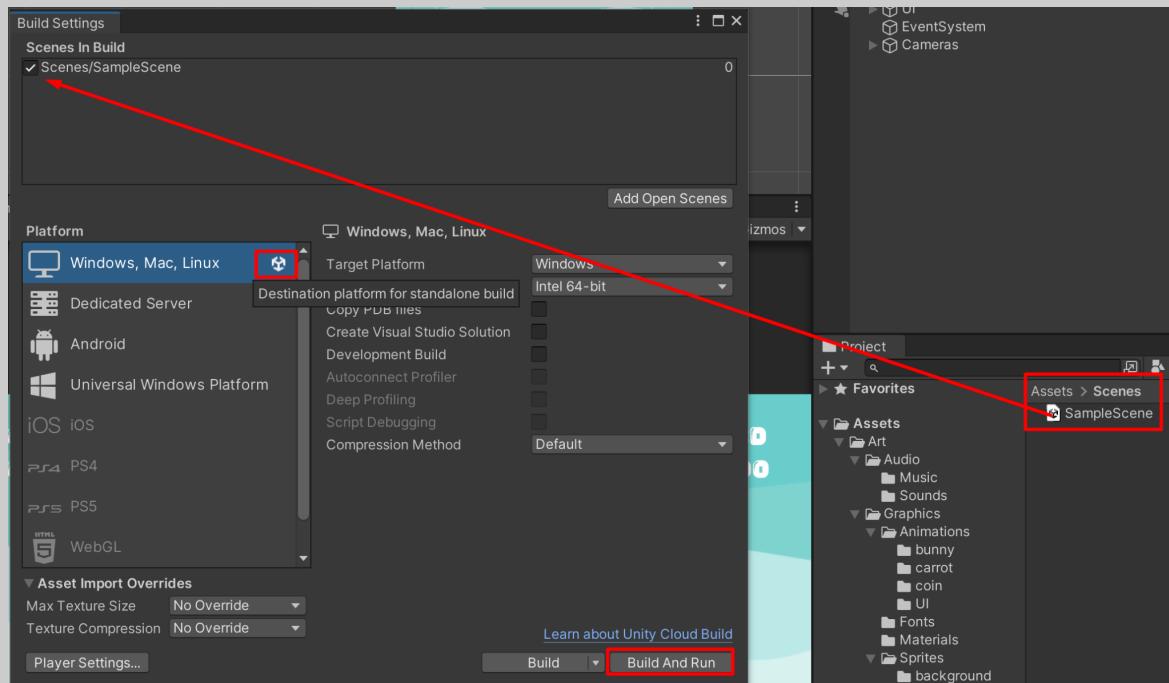
- Cámara persigue al Jugador - CineMachine - #1: <https://www.youtube.com/watch?v=TrZYt5pS1pY&t=476s>
- Transición entre Cámaras y Confinar Área Visible - CineMachine - #2: <https://www.youtube.com/watch?v=e8WkzvsBRC4>
- Vibración (Shake) de Cámara con Impulsos - CineMachine - #3: <https://www.youtube.com/watch?v=53wttnhyhhA>
- Vibración con Impulsos sin Colisiones - CineMachine - #4: <https://www.youtube.com/watch?v=C9yaFP0eWmo>
- Cambiar Cámara dentro de un Área - CineMachine - #5: https://www.youtube.com/watch?v=RWxMboV8_8s

Sería interesante que el alumno, tras ver los videos, implementara en su proyecto algún efecto curioso ayudándose de alguna de las muchas funciones de Cinemachine.

Para finalizar el proyecto, de momento, vamos a exportarlo para nuestra plataforma Windows (en el tema siguiente veremos cómo exportar a Android y a XBOX Series) y así comprobar su correcto funcionamiento. Pulsamos sobre el menú File>Build Settings...



Arrastramos nuestra única escena a la ventana "Scenes In Build". Nos aseguramos que la plataforma elegida sea Windows y pulsamos el botón "Build" o "Build And Run":



Seleccionamos una carpeta y en unos instantes ya tendremos nuestro archivo ejecutable junto al resto de librerías necesarias:

Este equipo > Disco local (C:) > Usuarios > Oscar > Escritorio > bunny				
	Nombre	Fecha de modificación	Tipo	Tamaño
do	MonoBleedingEdge	02/11/2022 17:52	Carpeta de archivos	
s	The Bunny Game_BurstDebugInformation_DoN...	02/11/2022 17:52	Carpeta de archivos	
tos	The Bunny Game_Data	02/11/2022 17:52	Carpeta de archivos	
	The Bunny Game.exe	02/11/2022 17:52	Aplicación	639 KB
	UnityCrashHandler64.exe	02/11/2022 17:52	Aplicación	1.099 KB
	UnityPlayer.dll	02/11/2022 17:52	Extensión de la aplic...	28.154 KB

Si damos doble click sobre el ejecutable veremos que se nos abre nuestro juego tras la Splash Screen de Unity:

