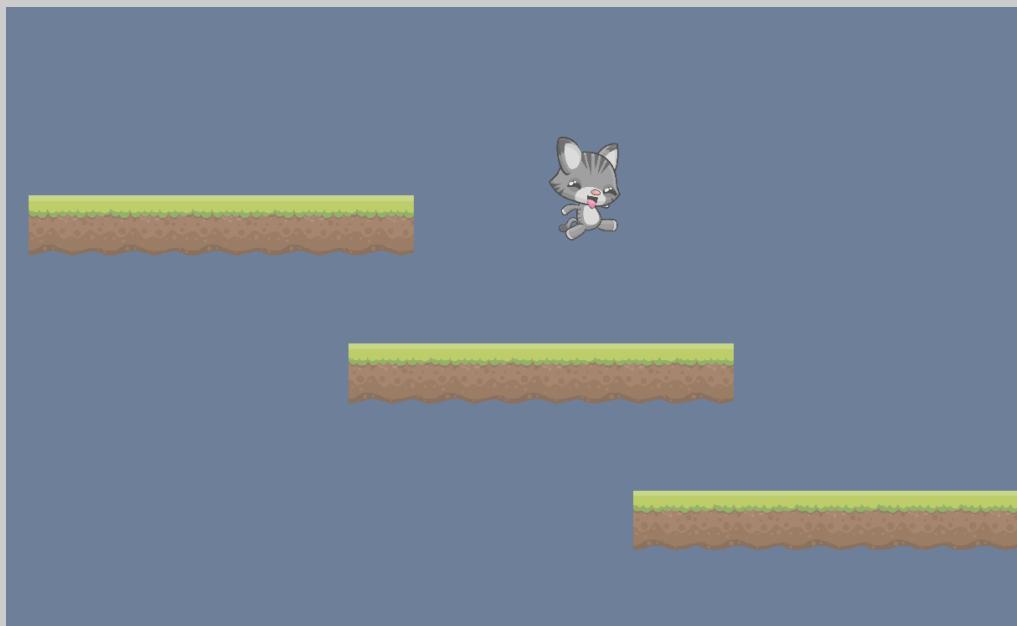


04.- DESARROLLO DE UN JUEGO 2D SENCILLO EN UNITY

1. Introducción	1
2. El Game Design Document (GDD)	2
3. Las tareas a incorporar en el GDD de nuestro juego 2D	4
4. Desarrollo del juego: El Player Controller	6
5. Desarrollo del juego: El Game Manager	22



1. Introducción

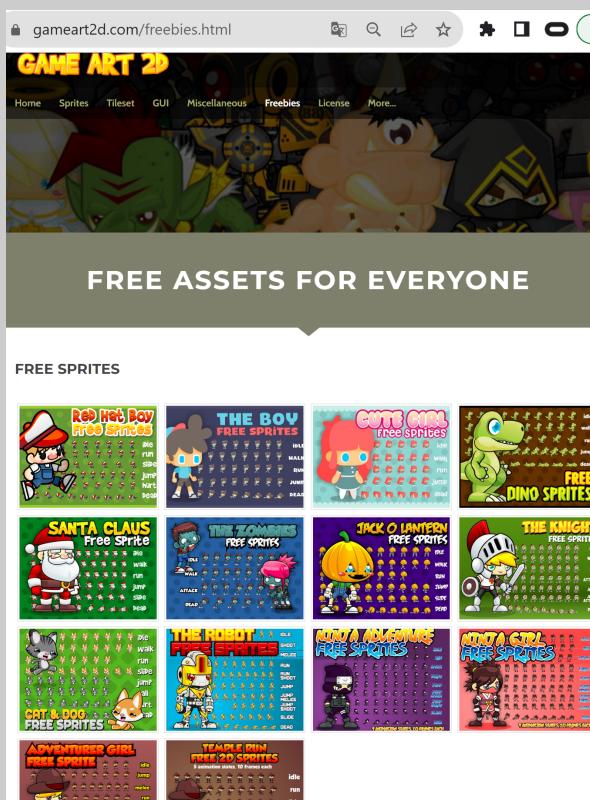
Tras haber cogido experiencia en los bloques anteriores con la programación C# y la interfaz de Unity, es el momento de aplicar todo lo visto y aprendido y crear nuestro primer juego. Va a ser un juego sencillo y modesto que nos permita adquirir soltura con el entorno Unity partiendo de cero, pero que funcione correctamente y sea perfectamente jugable.

Aprenderemos técnicas que nos ayudarán más adelante en desarrollos más complejos, además del manejo de las herramientas que nos ofrece Unity, la física, los sprites, el scroll, crear una interfaz, estructura del GamePlay...y todo lo que rodea a la jugabilidad y el diseño.

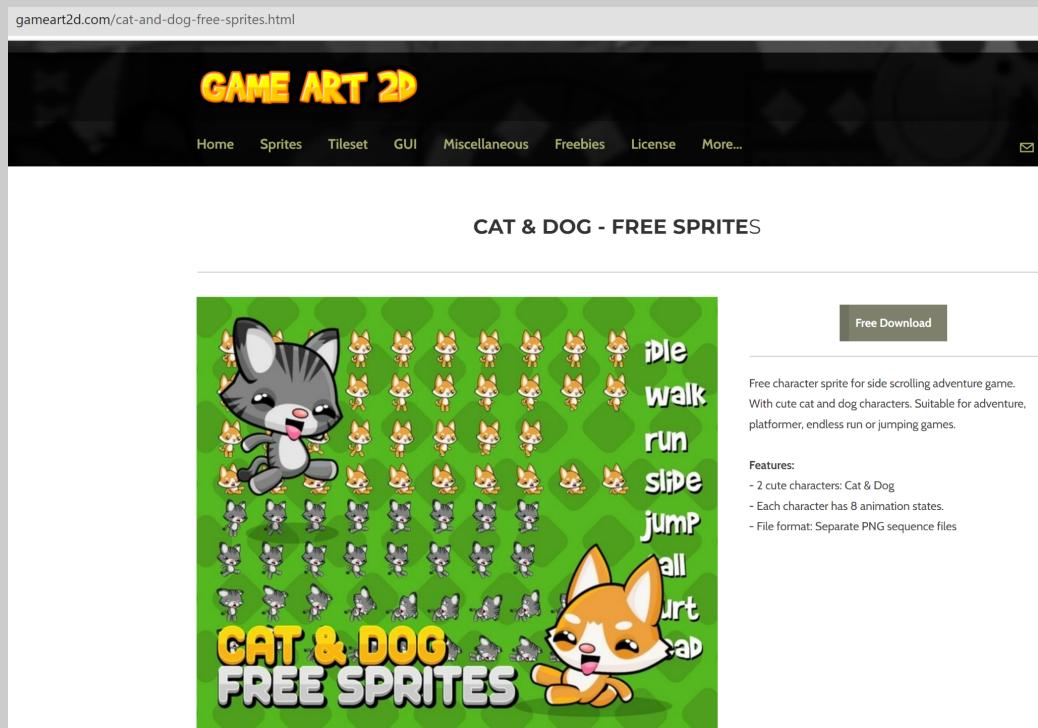
Lo normal es que os atasqueis y que os equivoqueis, pero esa será la oportunidad ideal para aprender a buscar en la documentación de Unity, videotutoriales, y también los foros de desarrollo Unity especializados (los oficiales y los demás) que existen en Internet.

2. El Game Design Document (GDD)

Lo primero de todo que tenemos que tener claro es el videojuego que queremos crear. En nuestro caso va a ser un juego 2D, de plataformas, de corte infantil. Nuestro protagonista podría ser cualquiera de estos personajes de ejemplo que [os pongo en la imagen](#), u otros descargados de otra web, u otros hechos por vosotros mismos en cualquier programa de edición de imágenes o de edición “pixel-art”:



Yo voy a elegir a este gatito: <https://www.gameart2d.com/cat-and-dog-free-sprites.html>



Para el argumento nos bastará con saber que nuestro personaje irá paseando por un bosque, recogiendo comida (por ejemplo, zanahorias) y para ello podrá correr y saltar por las diferentes plataformas que se encuentre, teniendo cuidado de no caerse en los diferentes huecos que se encuentre a su paso.

Para ello dispondremos de los sprites y resto de assets necesarios para su desarrollo y así podremos centrarnos más en programar las diferentes mecánicas del juego.

Concretamente, nuestro juego será del tipo “Infinite Scroll Game”, donde nuestro protagonista podrá ir corriendo infinitamente. Una vez que terminemos la base entonces ya podremos empezar a añadirle más profundidad a la jugabilidad como, por ejemplo añadiendo enemigos, zonas secretas, controles para dispositivos móviles, items que le den poderes al protagonista, la posibilidad de arrojar a sus enemigos los items que haya recogido para eliminar a estos, etc...lo que se nos ocurra. En el desarrollo de un proyecto real, todas estas posibilidades que se nos están pasando por la cabeza, para que el proyecto no se nos vaya de las manos y no lo terminemos nunca, tienen que estar recogidas con anterioridad en un documento llamado Game Design Document (GDD) y así dejar constancia de las características que queremos para nuestro juego e incluso las que no queremos pero que a medio/largo plazo se podrían implementar.

El GDD no tiene una estructura fija, no obstante en Internet se pueden encontrar multitud de plantillas que luego en tu equipo de desarrollo podréis usar o adaptar hasta

tener una a vuestro gusto para vuestro tipo de juego a desarrollar (por ejemplo, serán muy diferentes los GDD de un “Candy Crash” al de un “Metroid Dread”, ya que son juegos muy diferentes, con mecánicas distintas, los mundos son diferentes, los logros, el nivel de exploración, uno tiene monetización (micropagos) y el otro no, los rankings, la necesidad de estar conectado a Internet, destinados a ser jugados en dispositivos distintos, controles distintos...).

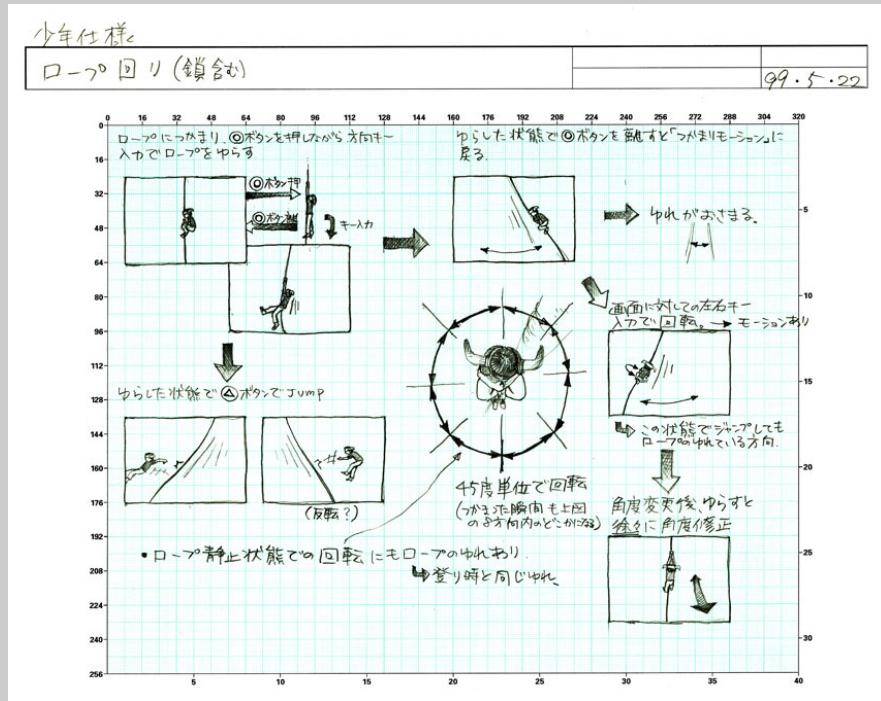
La finalidad principal de un GDD es ser el documento de referencia en el que todas las personas que integran el equipo de desarrollo tengan un rumbo común. Para ello el “Game Designer” tiene que redactar un buen GDD que sea comprensible, consensuado, y que sea lo más específico posible para que todas las personas que realizan el proyecto, e incluso aunque solo seas tu mismo, puedan llegar a buen puerto y que el proyecto no se convierta en algo disperso y que nunca se termina. Eso no significa que este documento sea estático e inalterable a lo largo del desarrollo, ya que pueden surgir ideas nuevas y estas se pueden añadir al GDD en la lista de posibles mejoras a posteriori para cuando tengas un producto mínimo viable. ([This Game Dev Tip Saved Me 137 HOURS! - YouTube](#))

Una vez que se tiene terminado el GDD no es buena idea afrontar el problema directamente, sino que hay dividirlo en pequeñas partes . Por ejemplo, en un sencillo juego de plataformas, tendríamos:

- Por un lado, el diseño del nivel/escenario.
- Por otro lado, el diseño del protagonista, cómo se comporta y sus animaciones.
- Implementación de los controles que nos permitan manejarlos en el juego
- Diseño de “gameplay loop”: como se inicia la partida, que lleva a finalizarla, y como se reinicia una partida.
- Diseño de Items y su influencia en el gameplay.
- Diseño y comportamiento de enemigos.
- Diseño del HUD, que es la interfaz de comunicación con el jugador durante la partida, es decir, toda la información que vemos en pantalla: vida, energía, munición, mini mapa, vida de los enemigos, etc.
- Gestión de la puntuación y funcionamiento de los rankings...

También, otra buena práctica en los desarrollos es el prototipado, esto es, dibujos, esquemas y bocetos de mecánicas del juego, personajes, de interacciones, ideas, de situaciones que se plantean en el juego, del HUD y de los menús... Mismamente, en una

libreta y con un lápiz, o con herramientas ofimáticas que nos ayuden a ello como por ejemplo Sketch, Origami Studio, InVision, Adobe XD, Figma...



3. Las tareas a incorporar en el GDD de nuestro juego 2D

1.- Generar nivel proceduralmente: En nuestro juego va a ser un nivel de diseño 2D, de scroll infinito, de derecha a izquierda (por ejemplo, como el Flappy Bird). Para ello tendremos una cámara fija y un escenario que se irá moviendo hacia la izquierda, por lo que necesitaremos un constructor de escenario a la derecha de la pantalla, que irá generando un nivel aleatorio, y un destructor del escenario a la izquierda una vez que este ya desaparezca de la vista de la cámara.

2.- La animación del personaje: Nuestro protagonista podrá correr, saltar, y al morir se difuminará en forma de humo. Lo haremos a través de Mecanim, que es el sistema de animación implementado en Unity, para el control de los movimientos de los personajes, y que nos permite de una manera fácil configurar las animaciones disponibles, reproduciéndose de una forma predeterminada o bajo ciertas condiciones que vayan cumpliéndose.

3.- Física de nuestro juego: Que va desde elegir los valores de la gravedad deseada, así como el tema de las físicas y colisiones entre objetos, ejecutar (trigger) un evento en un momento determinado, etc... Para ello, veremos que en Unity se utiliza un componente llamado Rigidbody 2D que, al

asignarselo a un objeto del juego, ya podremos configurar su área de colisión y este se verá afectado por todos los eventos físicos del motor de Unity. Por ejemplo, nuestro protagonista será un RigidBody 2D y, por ello, al aplicarle un determinado impulso, podrá llevar a cabo la acción de saltar, y veremos cómo a medida que se acerca de nuevo al suelo será mayor su velocidad de bajada debido a que se ve acelerado por la fuerza de la gravedad.

4.- Los controles: Como se trata de un juego para PC pues utilizaremos para controlar a nuestro personaje el teclado y ratón. Además, le añadiremos un control táctil para exportarlo a nuestro dispositivo móvil.

5.- Diseño de items (o colecciónables): Nuestro protagonista irá recogiendo zanahorias con cuidado de no caerse al vacío. Aquí vuelve a entrar en juego todo el tema de la física, las colisiones con estos items y los triggers. Concretamente, vamos a utilizar dos funciones de la clase Monobehaviour llamadas “OnCollisionEnter2D” (para ejecutar un evento cuando dos objetos “chocan” entre sí, por ejemplo el collider del conejito y el collider de una zanahoria) y “OnTriggerEnter2D” (para ejecutar un evento cuando un objeto entra dentro de una zona, por ejemplo un agujero al vacío)

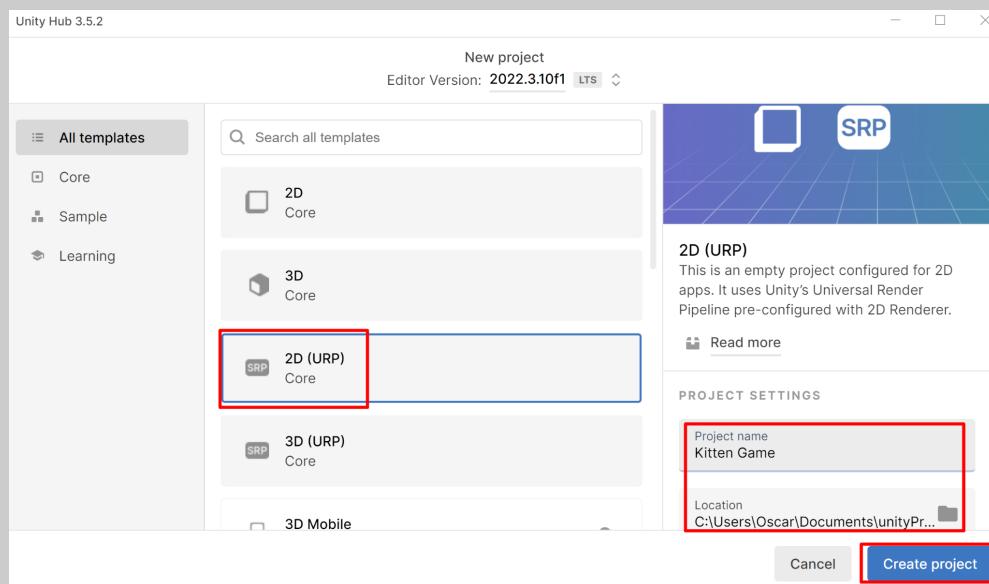
6.- Sistema de puntuación: En nuestro juego puntuará tanto la distancia que estemos recorriendo durante el juego, así como las zanahorias recogidas.

7.- La interfaz gráfica del usuario: Donde entrarán el diseño de los menús, las diferentes pantallas y la navegación entre ellas. Concretamente, en nuestro juego, tendremos 2 pantallas:

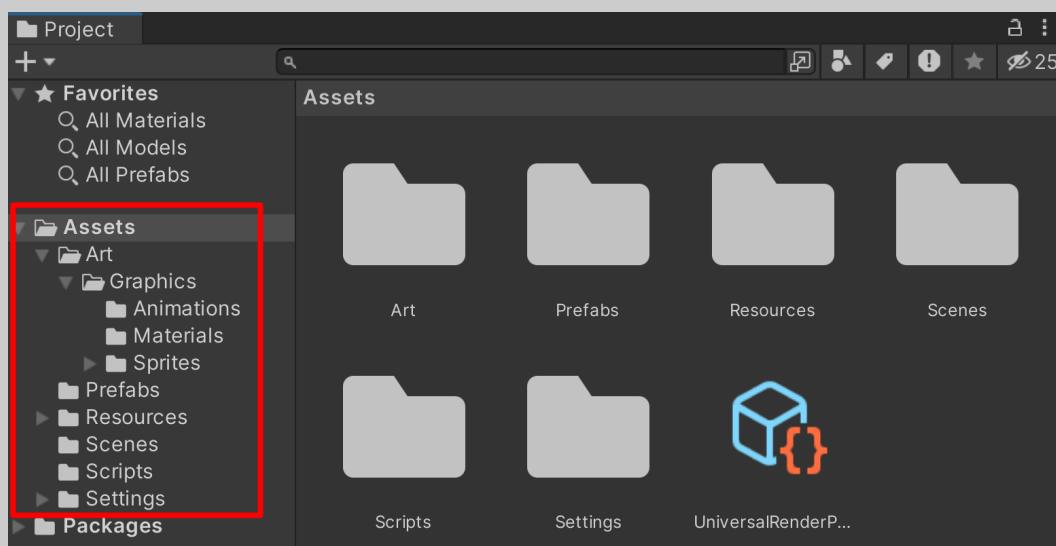
- Pantalla Principal o de Inicio: Con una imagen de fondo, el nombre del juego y un botón para empezar la partida. En un futuro se le podrían añadir pequeños iconos de compartición en redes sociales, desactivar el sonido, etc... pero esto no formaría parte del núcleo principal de nuestro juego.
- Pantalla donde se desarrolla el juego: Tendremos un HUB superior con nuestros puntos actuales, con nuestro récord de puntos guardado y con el número de zanahorias recogidas.
- Pantalla de Game Over: Donde además del letrero de “Game Over”, debajo vendrá la puntuación obtenida en la partida y el número de zanahorias recogidas y, por último, más abajo, dos botones: el de Jugar de nuevo y el de Volver a la Pantalla de Inicio.

4. Desarrollo del juego: El Player Controller

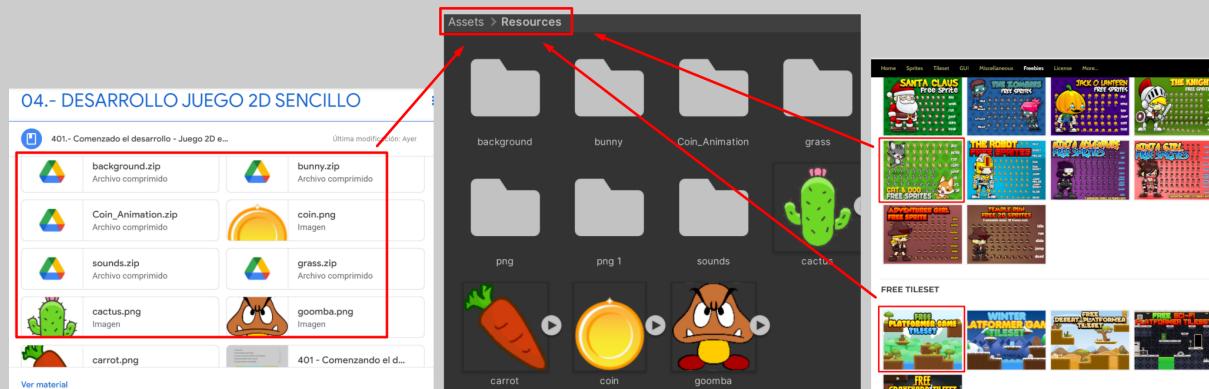
Comenzamos creando un nuevo proyecto 2D en Unity, por ejemplo en la versión 2022 LTS, dentro de una carpeta de nuestro equipo que veamos adecuada. Le ponemos un nombre al proyecto, como podría ser “*The Kitten Game*”, lo abrimos y elegimos una distribución de ventanas (Layout), con la que nos sintamos a gusto para trabajar con las herramientas del motor.



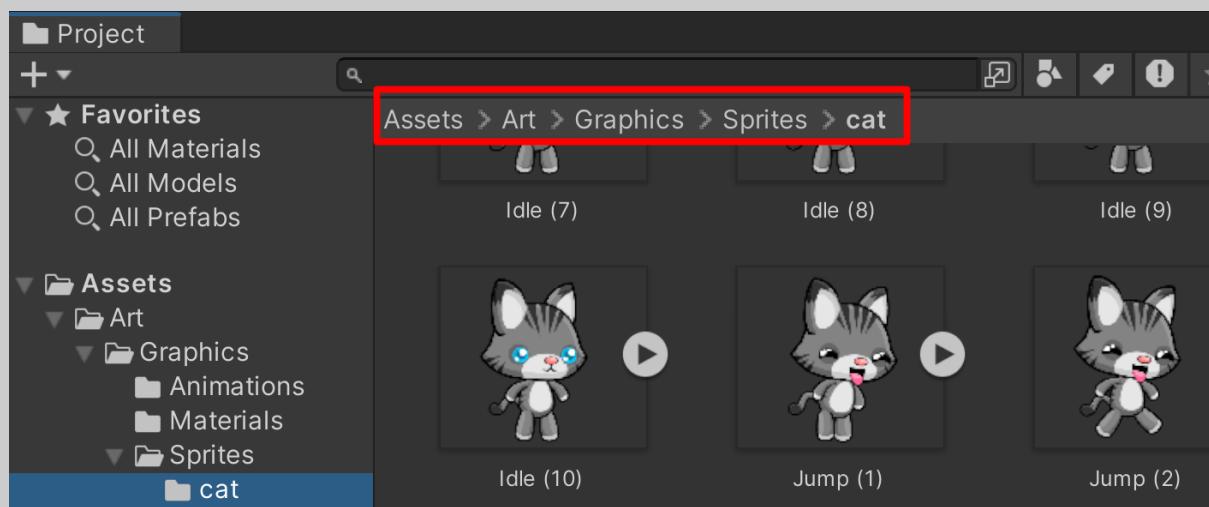
Tendremos en cuenta que todos los archivos de nuestro proyecto los colocaremos dentro de la carpeta Assets. Para ello, según vayamos añadiendo recursos a nuestro juego, iremos creando carpetas, e incluso subcarpetas, para tenerlo todo bien organizado y con nombres significativos, para que al final nos quede todo bien estructurado como nos se nos va indicando en la guía: [“308.- \[Unity\] Coding guidelines & Basic Best Practices”](#). En un futuro iremos formando una estructura como esta que veis en la imagen:



Descargamos de Classroom (y/o de aquí: <https://www.gameart2d.com/freebies.html>) los recursos y assets que utilizaremos en nuestro proyecto. Los iremos guardando todos en un “cajón desastre” que será la carpeta Resources para, más adelante, ir poco a poco moviendo lo que vayamos necesitando a las diferentes carpetas del proyecto. El objetivo es que, en las últimas fases del desarrollo del proyecto, borremos esta carpeta Resources cuando su contenido ya solo sean “los restos” que ya no vayamos a utilizar:

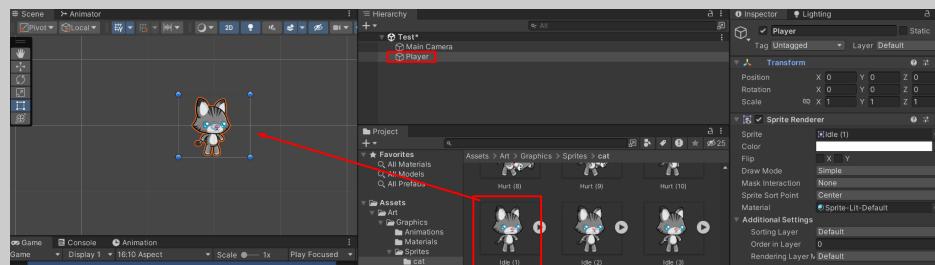


Una vez tengamos la carpeta de Resources llena de assets descargados, comenzaremos cojendo los sprites de nuestro protagonista y los moveremos (o copiaremos) desde Resources (nuestro “cajón desastre”) a la carpeta Assets>Art>Graphics>Sprites>cat:

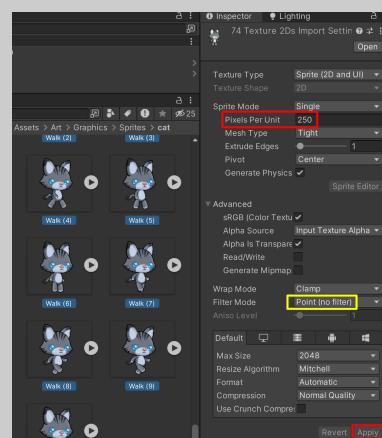


El siguiente paso será crear un Prefab de nuestro personaje. El Prefab es un tipo de GameObject (como los cubos, esferas, etc... que ya hemos visto y utilizado) pero hecho por nosotros mismos y sobre el que estableceremos una serie de características/componentes como las colisiones, la gravedad, las animaciones, estados, etc..

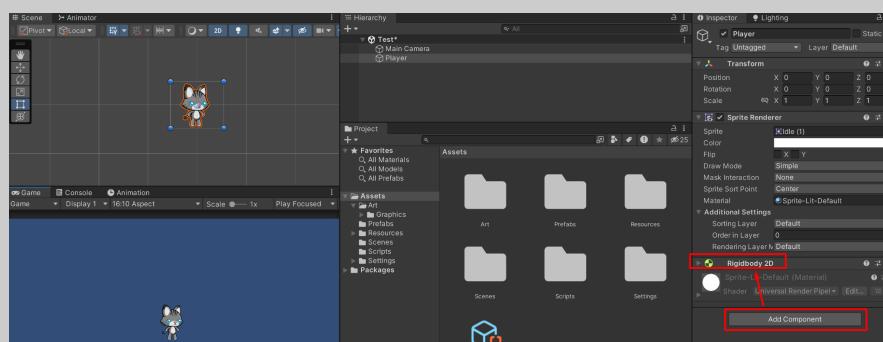
Para tener un punto de partida, comenzaremos arrastrando el sprite “Idle_1” a nuestra escena de juego (Scene) para que así se nos cree un GameObject con un componente “Sprite Renderer”. Este gameobject lo renombramos como “Player” y lo colocamos en la posición (0,0,0)



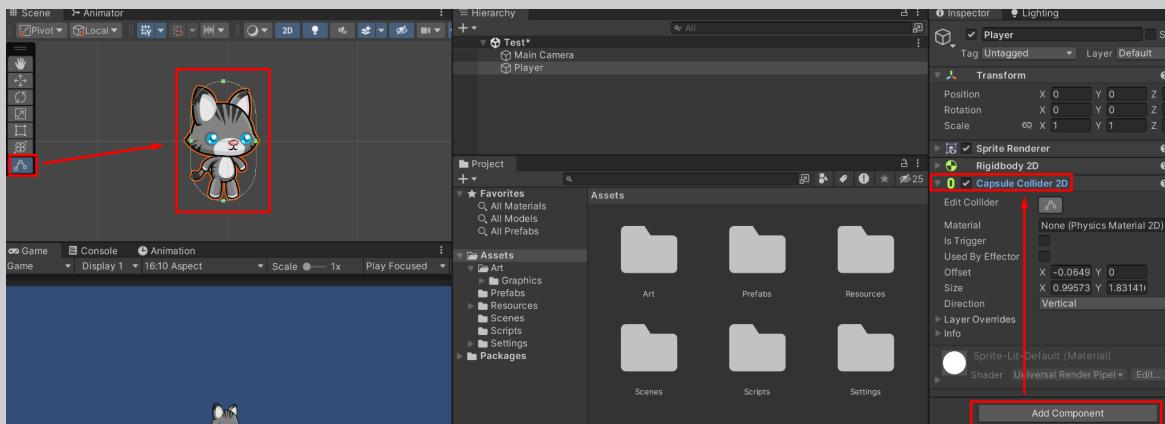
Nota: Si el tamaño del sprite en nuestro juego no nos convence, en vez de tocar la escala del transform del gameobject, es preferible seleccionar todos los sprites y cambiar su atributo “Pixels Per Unity” en su Inspector:



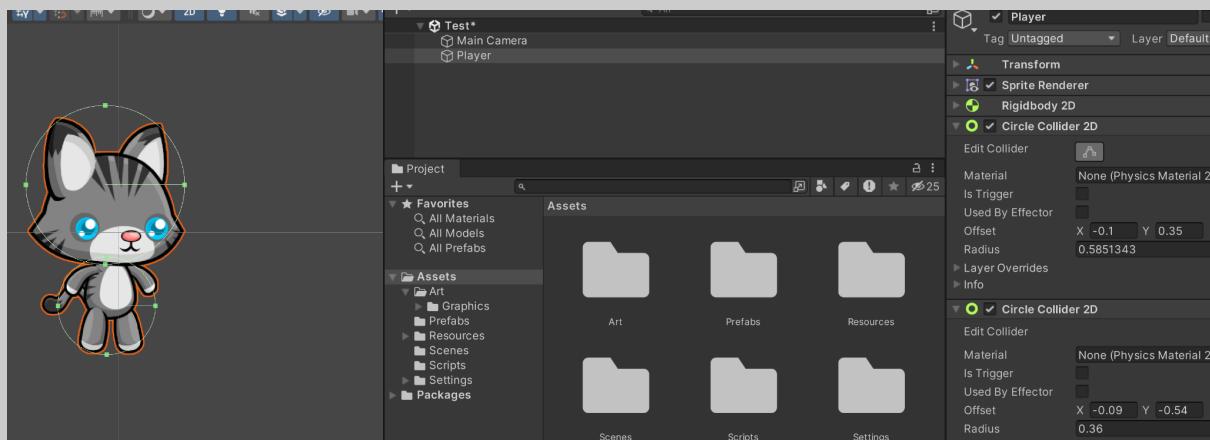
Si pulsamos el botón de “Play” veremos como se queda flotando en el aire en la escena de juego (Game) nuestro protagonista. Bastará con que le añadamos el componente “Rigidbody 2D”, desde el botón “Add Component” del Inspector, para que le afecten las físicas 2D del motor de Unity (*Edit > Proyect Settings > Physics 2D*), como por ejemplo, para que se vea afectado por la gravedad. Además el Rigidbody tiene una serie de atributos que nos permitirán controlar la física de nuestro personaje como la masa, el material, etc... así como aplicarle desde código fuerzas, velocidad...



A continuación también le añadiremos un Capsule Collider 2D, lo que nos permitirá establecer la zona de colisión de nuestro personaje, lo que le facultará para colisionar con otros GameObjects de nuestro juego que tengan un collider:

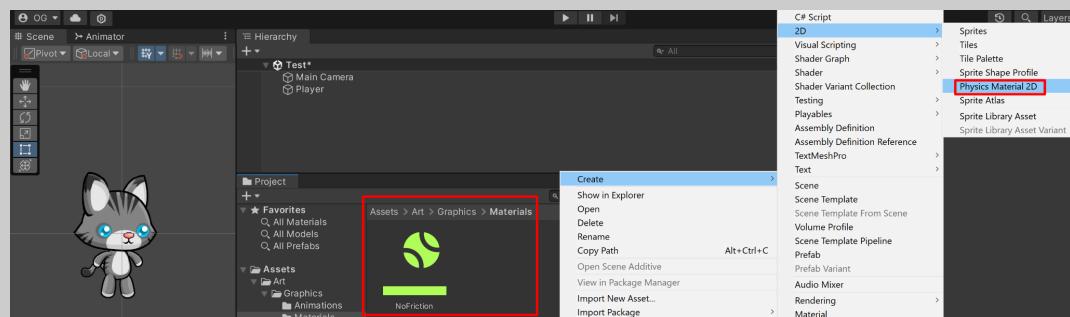


Nota: Si nos interesa, podríamos afinar más y ponerle varios Colliders con la finalidad, por ejemplo, de realizar diferentes acciones en función de collider que colisione:

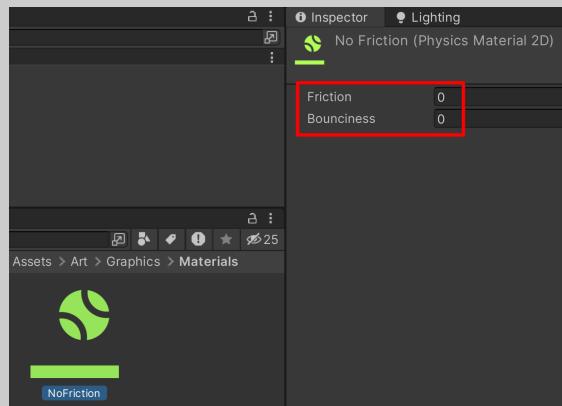


Para terminar con nuestro protagonista vamos a añadirle un “Physic Material 2D” sin fricción, de tal forma que cuando contacte/choque con otro gameobject se “resbale” y no haya fricción entre ellos.

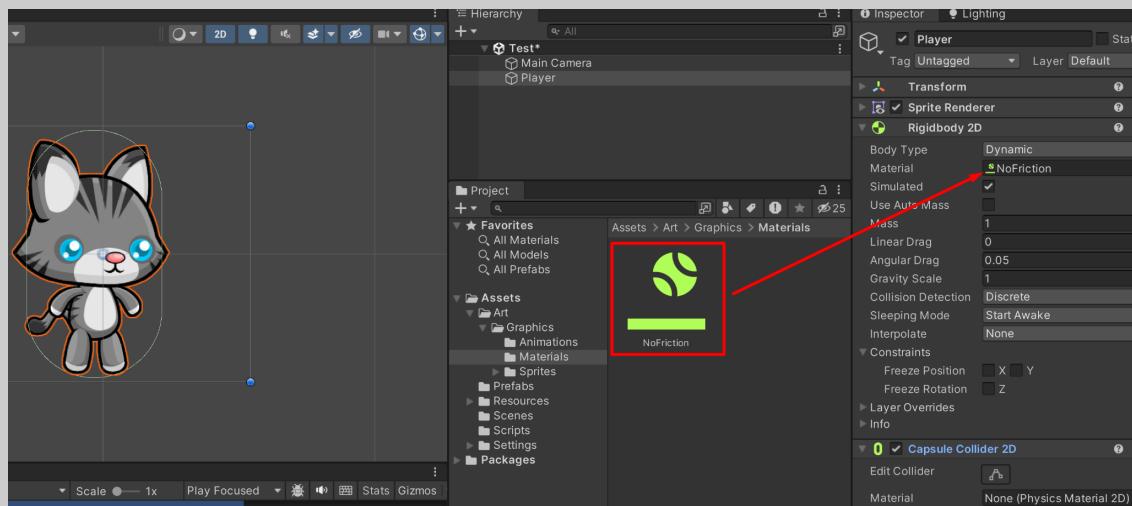
El primer paso consiste en crear el material dentro de la carpeta Assets>Art>Graphics>Materials pinchando con el botón derecho del ratón y eligiendo Create>2D>Physics Material 2D:



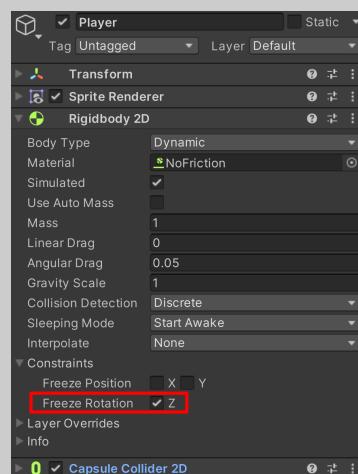
A continuación, le reducimos la fricción a 0, y el rebote se lo dejamos en 0:



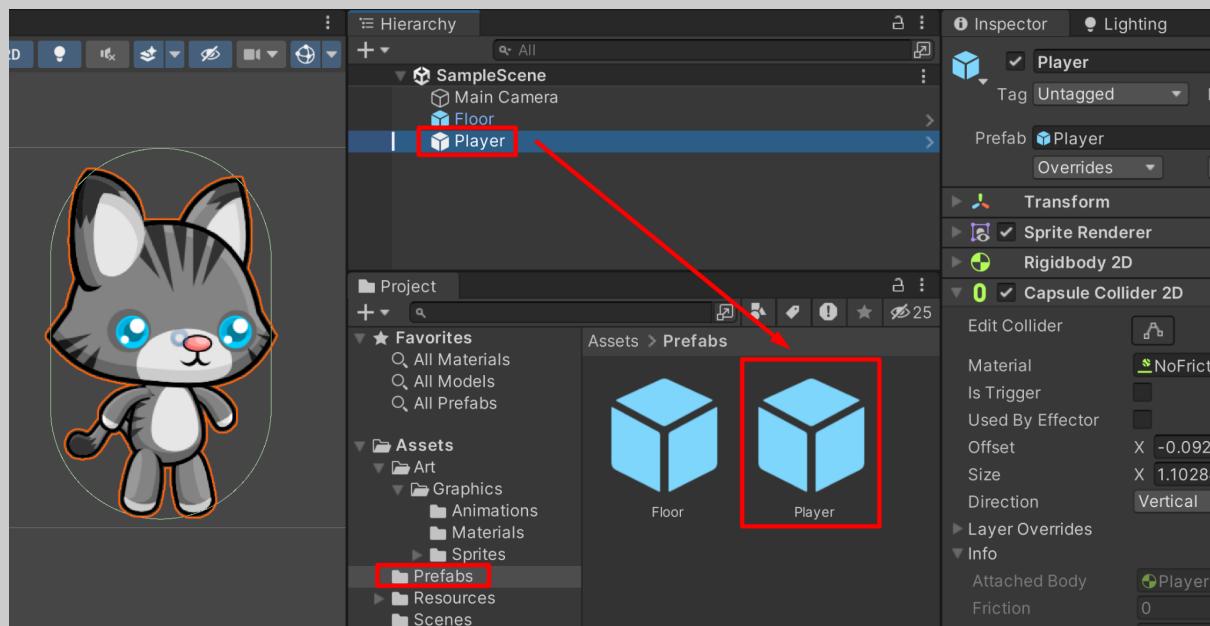
Ahora asignaremos ese material al Rigidbody2D de nuestro personaje arrastrándolo a su atributo material. Otra opción válida sería arrastrarlo a cada Collider, en el caso de que tuviéramos varios Colliders y quisieramos un comportamiento distinto para cada uno:



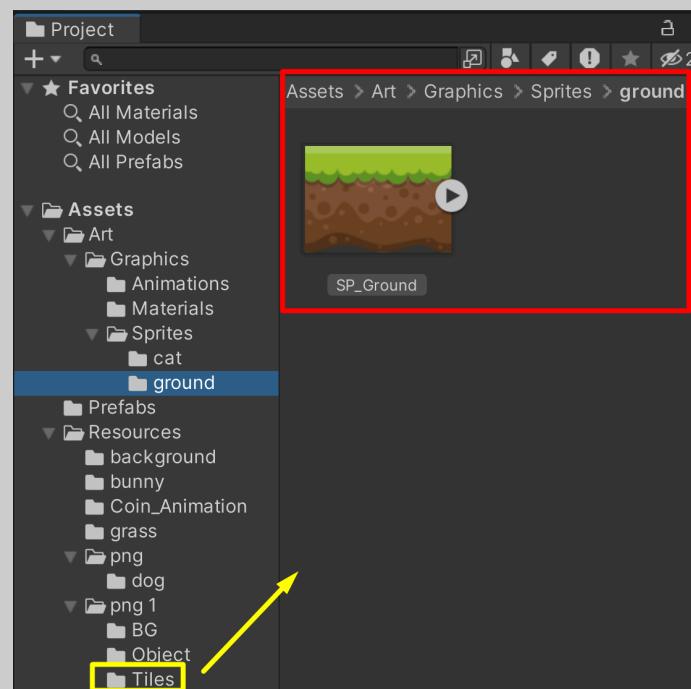
Por último, vamos a congelar la rotación en el eje Z de nuestro Player, ya que no me interesa que las físicas afecten a este eje del personaje al ser un juego 2D “puro” y además le ahorro cálculos innecesarios al motor de Unity:



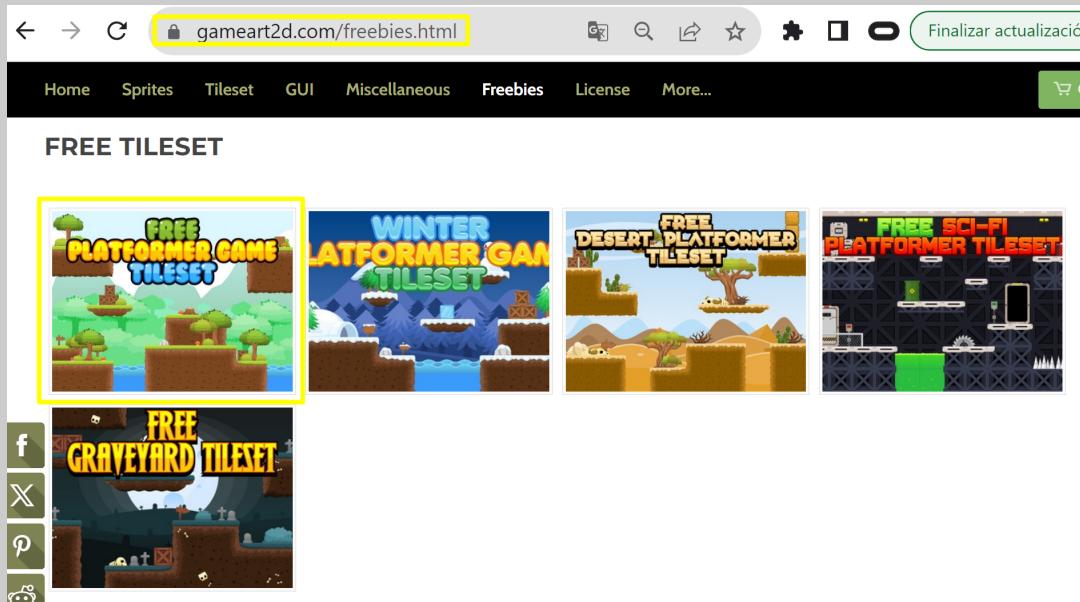
Una vez hecho esto, ya tenemos nuestro GameObject "Player" listo para ser guardado como un asset más del proyecto, y eso lo haremos en forma de "Prefab" arrastrándolo a una carpeta a la que llamaremos "Prefabs". Así, aunque borremos el Player de la escena siempre lo tendremos guardado como prefab con todas las características que le hemos ido dando:



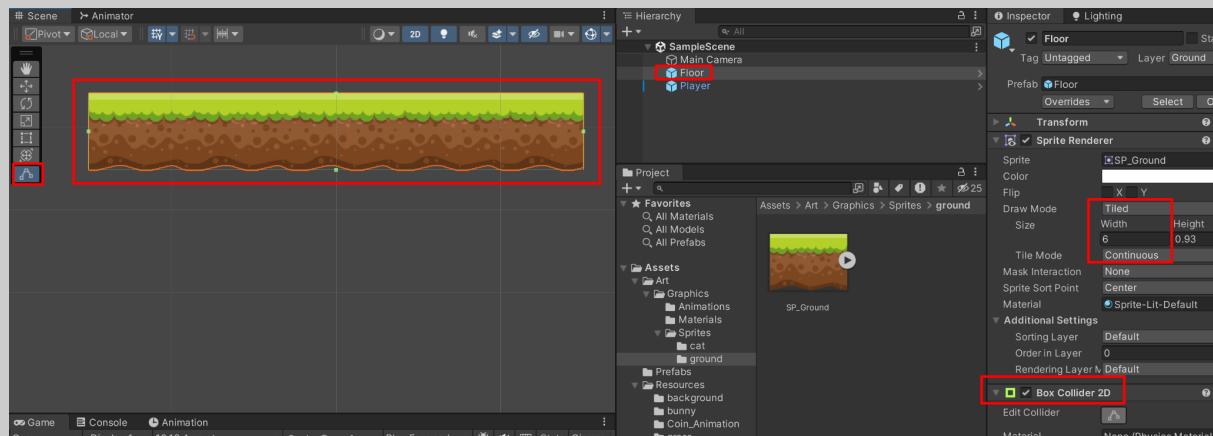
Ahora, para que nuestro personaje no se caiga al vacío cada vez que ejecutamos el juego, vamos a añadirle un suelo a nuestro gusto. Para ello nos descargamos el sprite de Classroom (o lo movemos desde la carpeta Resources si ya lo habíamos descargado allí antes) y lo colocamos en una carpeta de nuestro proyecto:



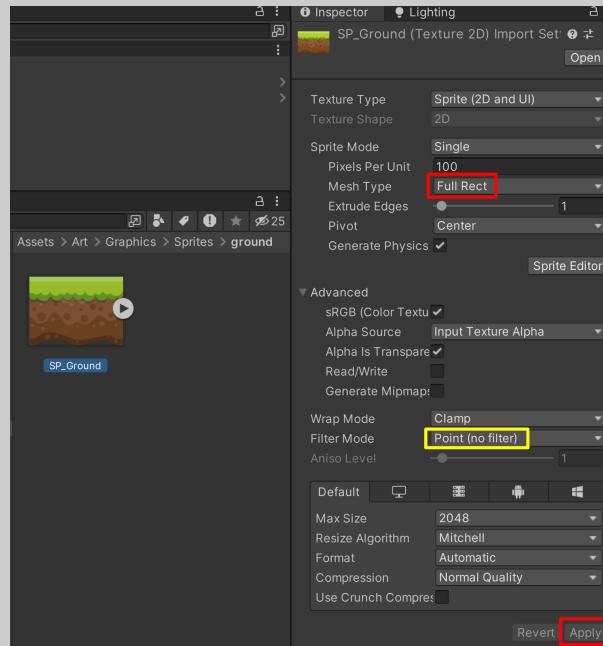
Otra opción es utilizar un tileset distinto, por ejemplo, uno de estos de Internet:



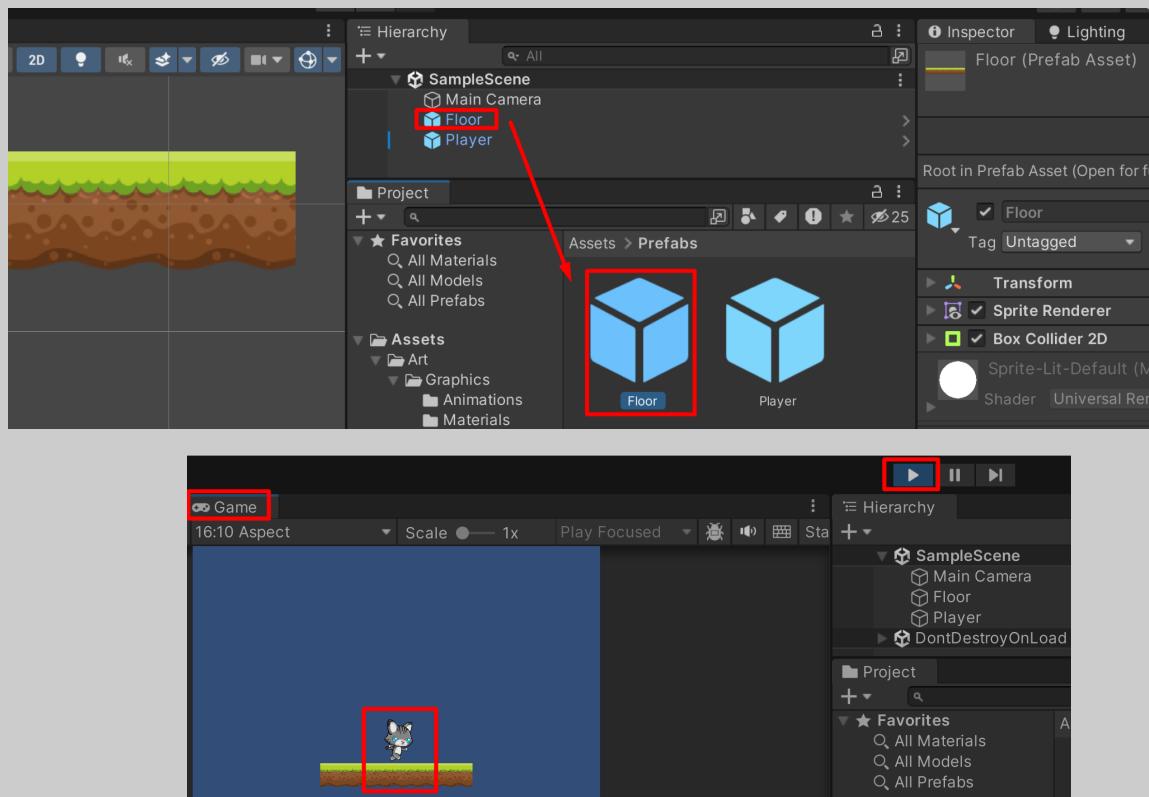
Lo arrastramos a la escena de juego, configuramos su componente Sprite Renderer modificando el atributo “Draw Mode” como Tiled (veremos que nos da un mensaje de advertencia que luego arreglaremos), aumentamos el campo Width y luego le añadimos un Box Collider:



Tal y como nos indica el mensaje de advertencia, cambiamos el Mesh Type a “Full Rect”:

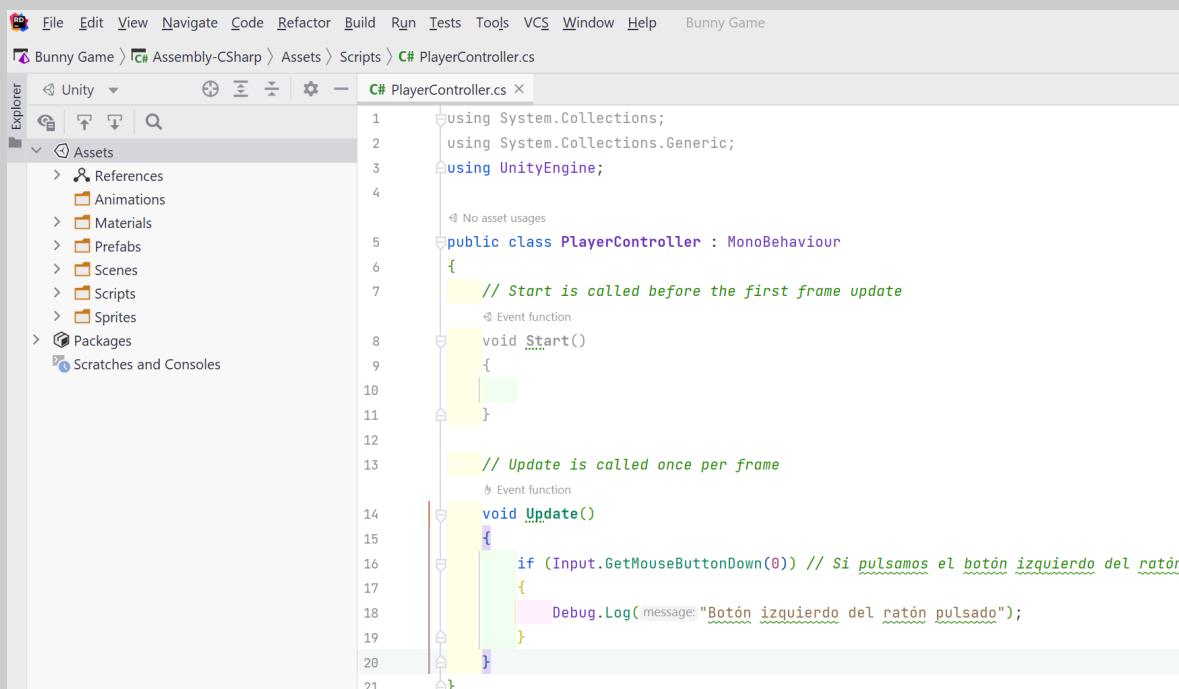


Si al dar Play vemos que el conejito se queda encima del suelo, ya podemos guardar ese suelo como Prefab y le llamamos, por ejemplo, “Floor”:



Una vez guardados y configurados nuestros 2 prefabs, ha llegado la hora de programar los comportamientos. Para ello en la carpeta “Scripts” creamos un C# script llamado PlayerController. Ese script se lo vamos a arrastrar a nuestro protagonista y lo abrimos para editar su código.

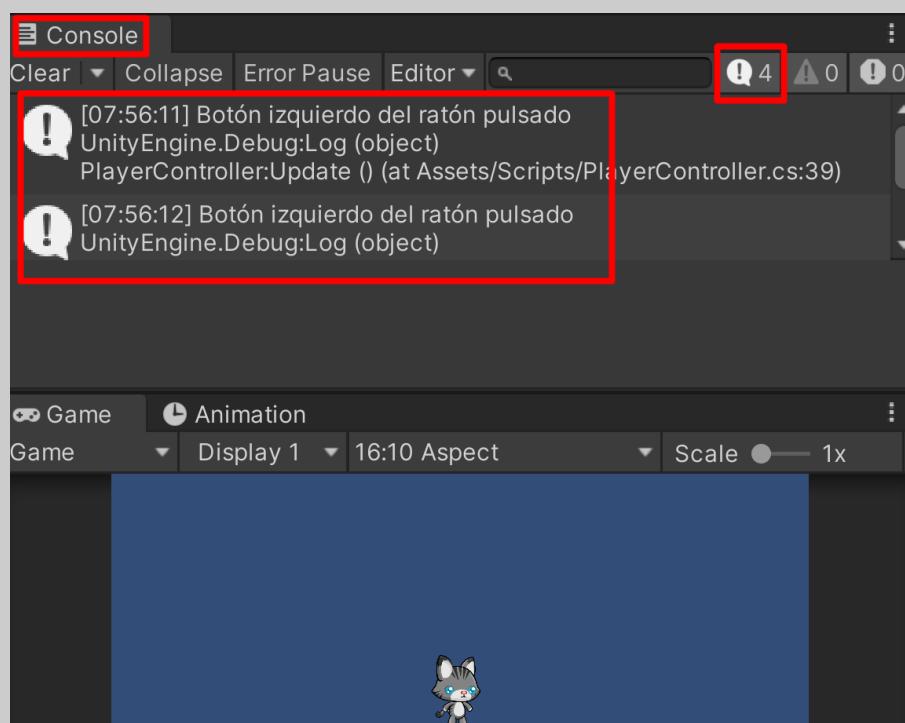
Vamos a empezar programando que nuestro protagonista salte cuando pulsemos el botón del ratón. De momento me mostrará un mensaje en la vista Console cada vez que pulse el botón izquierdo del ratón en la vista Game:



```

File Edit View Navigate Code Refactor Build Run Tests Tools VCS Window Help Bunny Game
Bunny Game > Assembly-CSharp > Assets > Scripts > C# PlayerController.cs
C# PlayerController.cs
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class PlayerController : MonoBehaviour
6 {
7     // Start is called before the first frame update
8     void Start()
9     {
10     }
11
12     // Update is called once per frame
13     void Update()
14     {
15         if (Input.GetMouseButtonDown(0)) // Si pulsamos el botón izquierdo del ratón
16         {
17             Debug.Log(message: "Botón izquierdo del ratón pulsado");
18         }
19     }
20 }

```

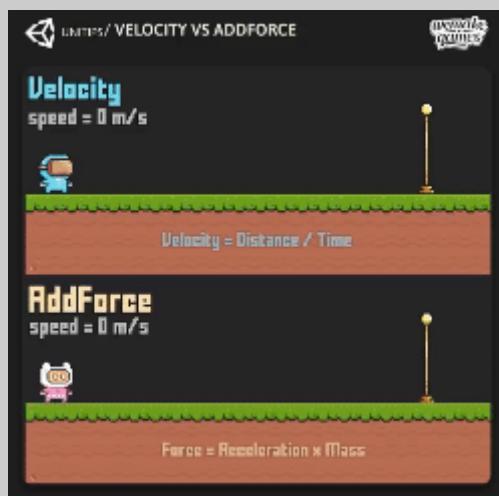


Como vemos que Debug.Log funciona, procedemos a programar la acción de saltar:

```
public class PlayerController : MonoBehaviour
{
    public float jumpForce = 20.0f; // Unchanged
    private Rigidbody2D _rigidbody2D;
    // Event function
    private void Awake()
    {
        this._rigidbody2D = GetComponent<Rigidbody2D>();
    }
    // Event function
    void Start()
    {
    }
    // Event function
    void Update()
    {
        if (Input.GetMouseButtonDown(0)) // Si pulsamos el botón izquierdo del ratón
        {
            Debug.Log(message: "Botón izquierdo del ratón pulsado");
            Jump();
        }
    }
    // Frequently called 1 usage
    void Jump()
    {
        _rigidbody2D.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
        //_rigidbody2D.AddForce(new Vector2(0, 1 * jumpForce), ForceMode2D.Impulse);
    }
}
```

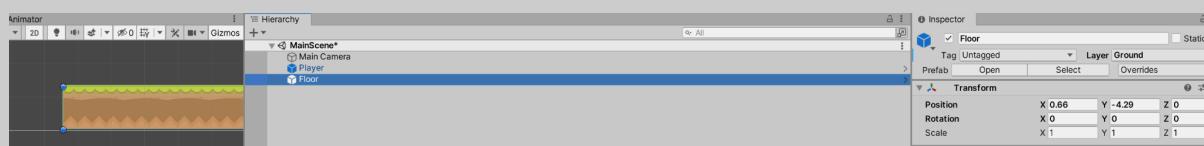
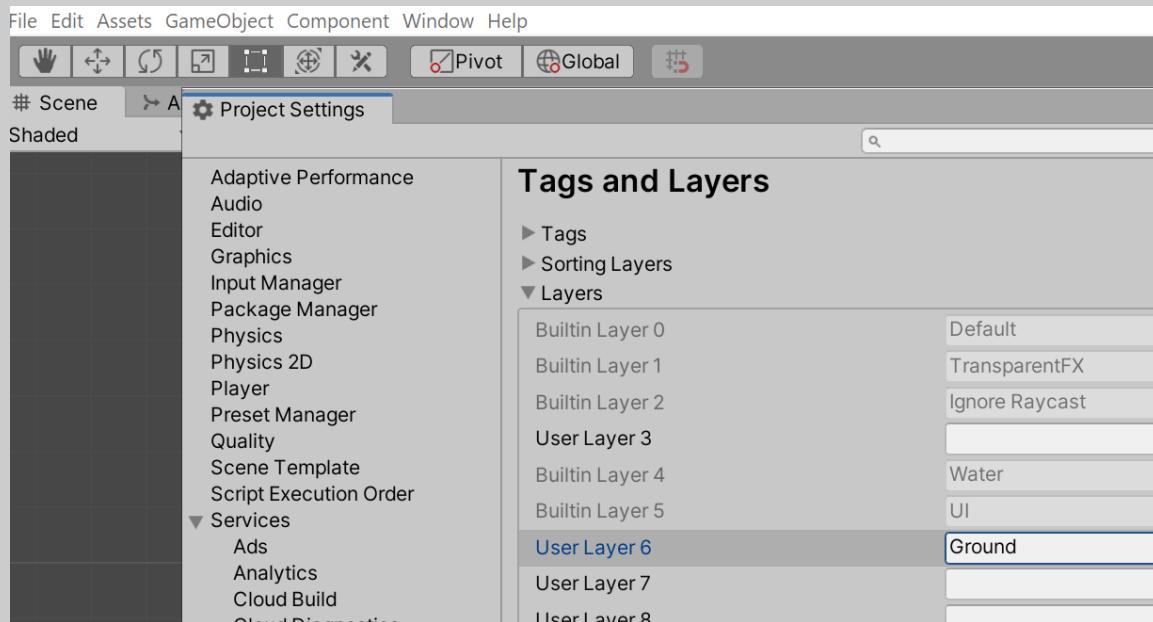
Fíjate que `Vector2.up` es la “abreviatura” de `new Vector2 (0.0f, 1.0f)`. [Realmente es un atributo estático \(o propiedad estática, si así lo prefieres\) de la clase Vector2.](#)

Nota: Diferencia gráfica entre aplicar una velocidad o una fuerza a un RigidBody2D:



```
//_rigidbody2d.velocity = new Vector2(0.0f, 1.0f) *jumpForce;
//_rigidbody2d.velocity = Vector2.up *jumpForce;
```

Ahora mismo, nuestro protagonista ya salta, pero nos interesa que solo pueda volver a saltar cuando haya tocado el suelo. Para ello, comenzaremos creando una capa (Layer) que llamaremos “Ground” (Edit>Project Settings>Tags and Layers) y que a continuación se la asignaremos al suelo:



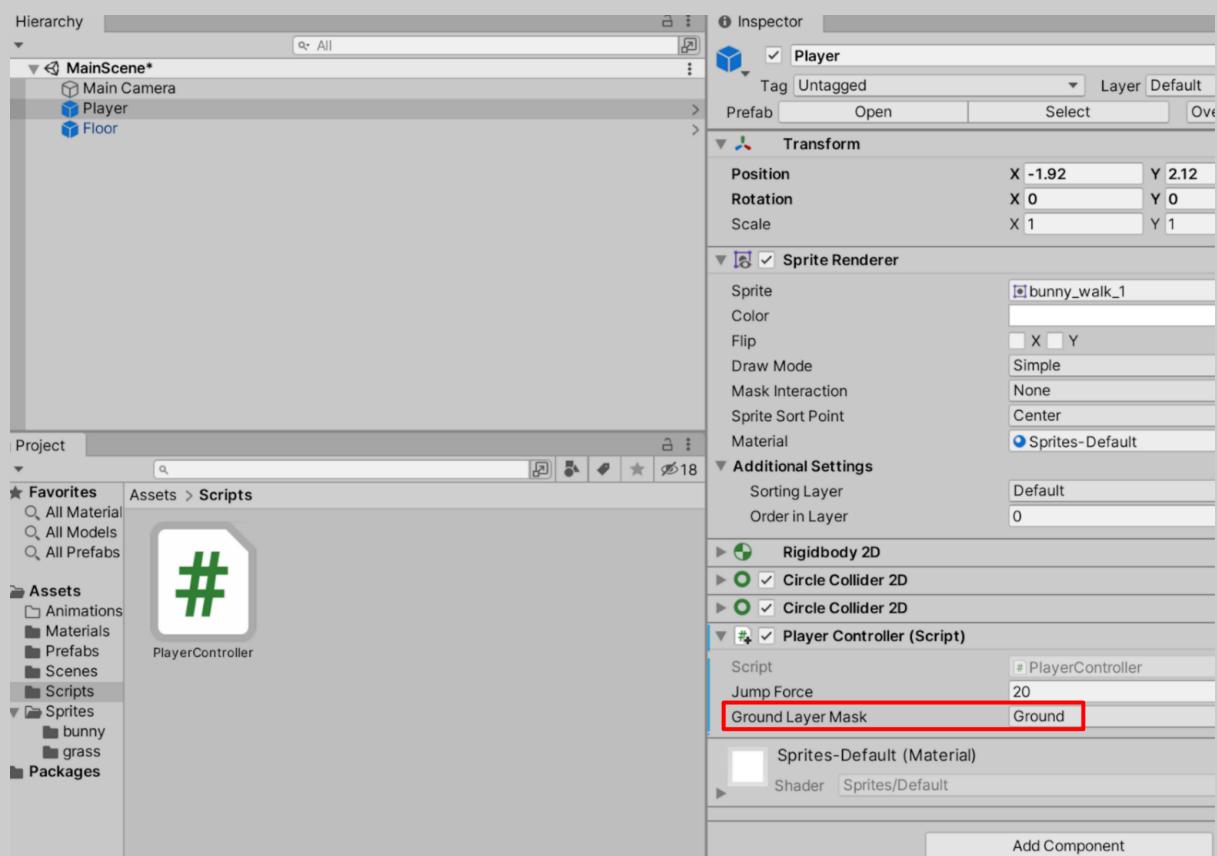
Ahora, continuaremos editando el script PlayerController en el que utilizaremos el método **RayCast** de la clase Physics2D, que nos permite lanzar un rayo desde donde está mi protagonista (`this.transform.position`) hacia abajo (`Vector2.down`) con una longitud `_distanceRaycast` de 1.0f y ver si choca con el `groundLayerMask.value` (donde desde el inspector indicaremos que es la capa “Ground” que creamos antes. Veamos el resultado:

```
C# PlayerController.cs ×
1  ^ o  public class PlayerController : MonoBehaviour
2  {
3      public float jumpForce = 6.0f; //Fuerza de salto para multiplicarsela a Vector2.up  ↵ Unchanged
4      private Rigidbody2D _rigidbody2d; //Para guardar la referencia al componente Rigidbody2D del Player
5      private float _distanceRaycast = 1.0f; //Longitud del Raycast
6      public LayerMask groundLayerMask; //Seleccionaremos en el inspector la capa deseada  ↵ Serializable
7
8      // Event function
9      private void Awake()
10     {
11         _rigidbody2d = GetComponent<Rigidbody2D>();
12     }
13
14     // Event function
15     void Start()
16     {
17     }
18
19     // Event function
20     void Update()
21     {
22         if (Input.GetMouseButtonDown(0)) //Si pulsamos el botón izquierdo del ratón
23         {
24             if (isOnTheFloor())
25             {
26                 Jump();
27             }
28         }
29
30         // Frequently called  ↵ 1 usage
31         public bool isOnTheFloor()
32         {
33             bool isOnTheFloor=false;
34             if (Physics2D.Raycast((Vector2) origin: this.transform.position, direction: Vector2.down, _distanceRaycast, (int) groundLayerMask))
35             {
36                 isOnTheFloor = true;
37             }
38             return isOnTheFloor;
39         }
40
41         // Frequently called  ↵ 1 usage
42         public void Jump()
43         {
44             _rigidbody2d.AddForce(Vector2.up*jumpForce, ForceMode2D.Impulse);
45         }
46
47         /*private void OnDrawGizmos()
48         {
49             Gizmos.DrawWireSphere(this.transform.position, _distanceRaycast);
50         }*/
51 }
```

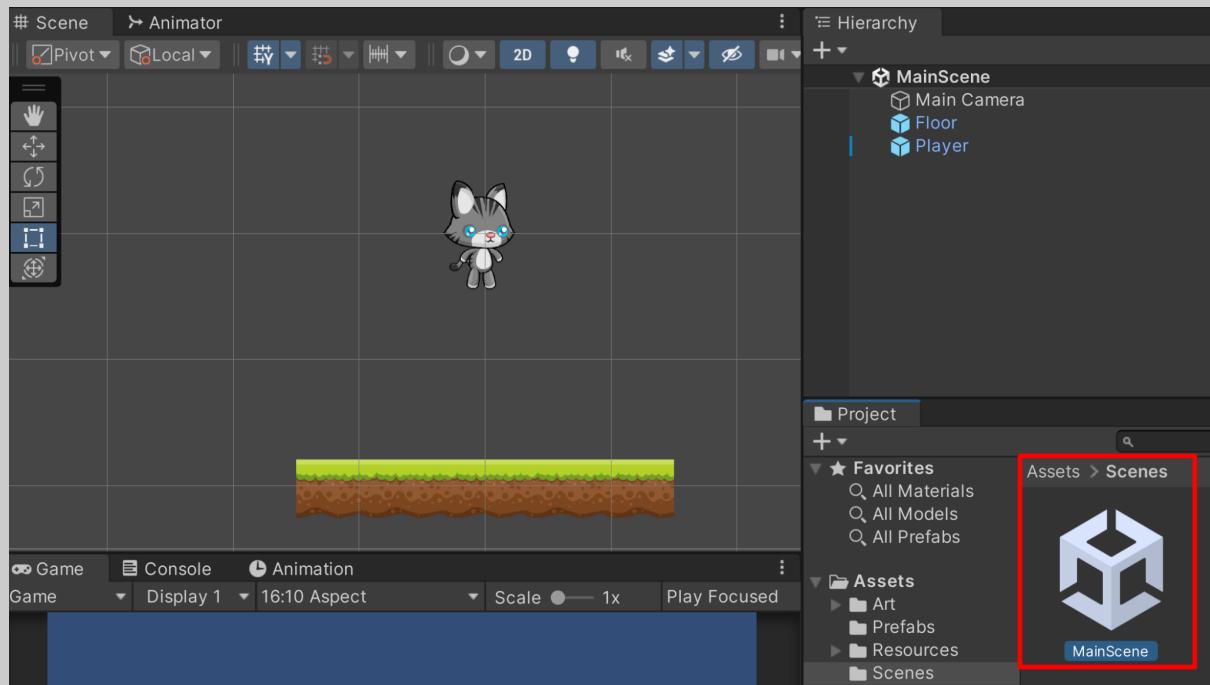
Nota: Representación gráfica del Raycast2D que hemos implementado, donde la distancia sería la variable `_distanceRaycast` que vale `1.0f` y que chocará con la capa `Ground` cuando nuestro personaje esté tocando el suelo:



Procedemos a elegir `Ground` en el atributo `Ground Layer Mask` de nuestro script `Player Controller`:



Probamos que nuestro protagonista salta de forma correcta y le damos a Archivo > Guardar y así quedará guardada nuestra escena en la carpeta correspondiente. Si quieres, puedes renombrarla como “MainScene”:

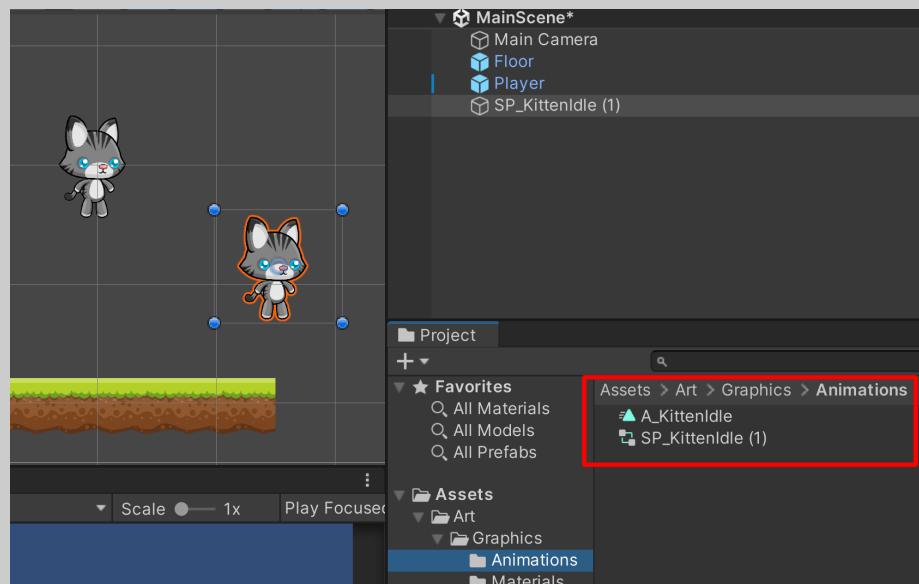
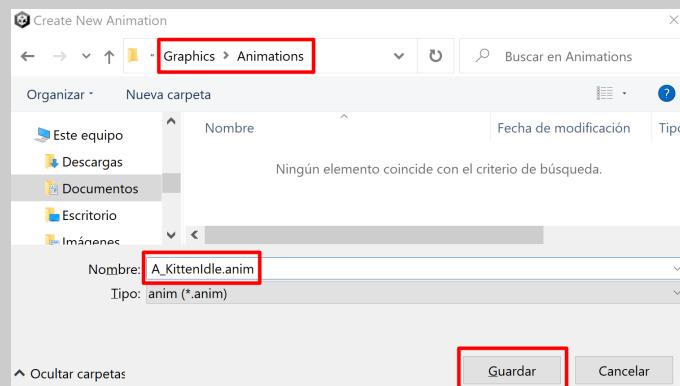
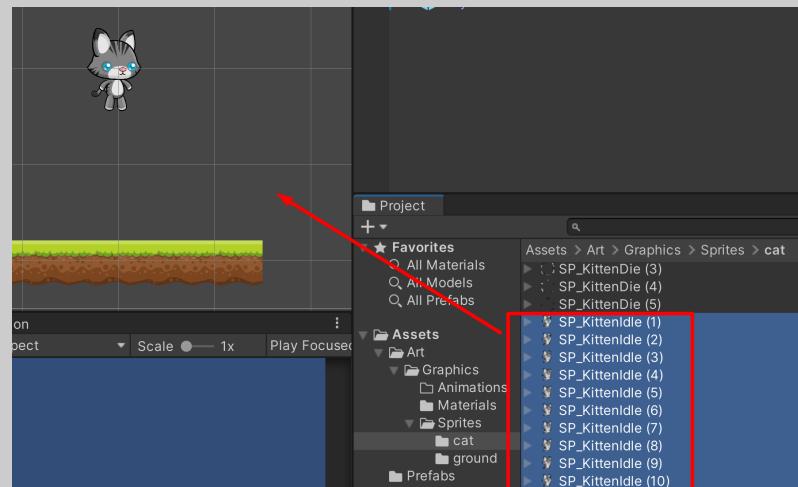


Nota: Si quisieramos ver dibujado una esfera de un radio de 1.0f en la escena (no en Game, ojo) para hacernos a la idea del alcance Raycast, podríamos usar el método OnDrawGizmos de la siguiente manera:

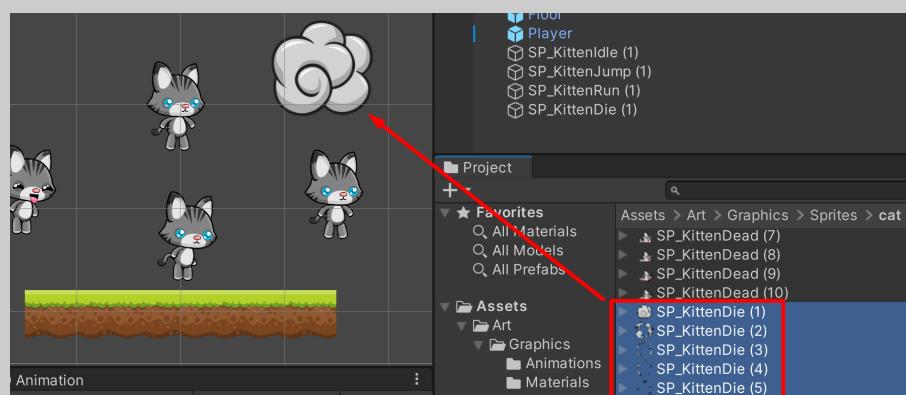
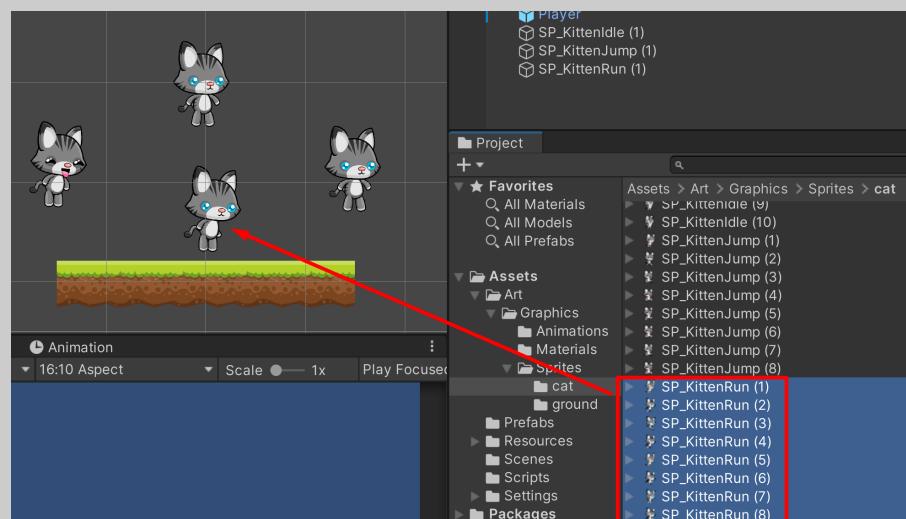
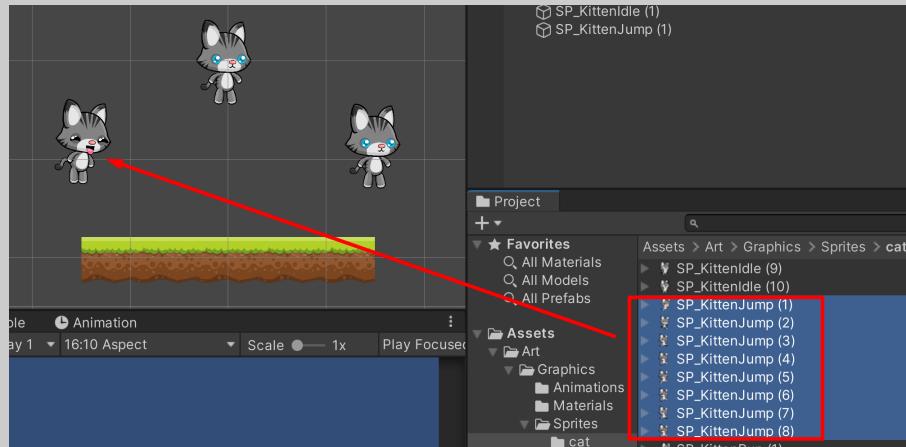
```
private void OnDrawGizmos(){
    Gizmos.DrawWireSphere(this.transform.position, _distanceRaycast)
}

//uso la variable isGroundedRange en vez de usar el 0.1f "a pelo" ->"hardcoding"
```

Continuamos el desarrollo de nuestro proyecto comenzando a crear las animaciones del protagonista. Concretamente, vamos a crear cuatro animaciones: A_KittenIdle, A_KittenJump y A_KittenRun y A_KitteDie. Para ello bastará con seleccionar todos los sprites que forman parte de la animación, arrastrarlos a la escena y luego guardar la animación en la carpeta correspondiente:

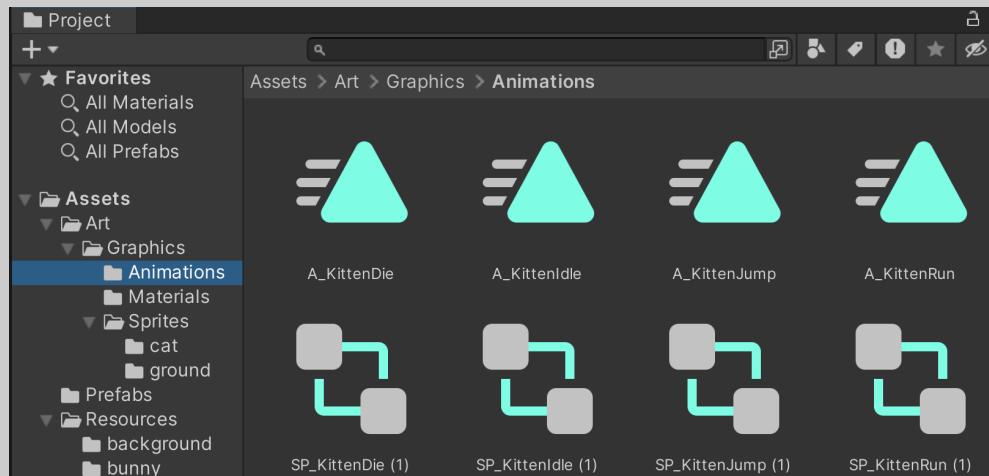


Procedemos a realizar la misma operación para las otras 3 animaciones:

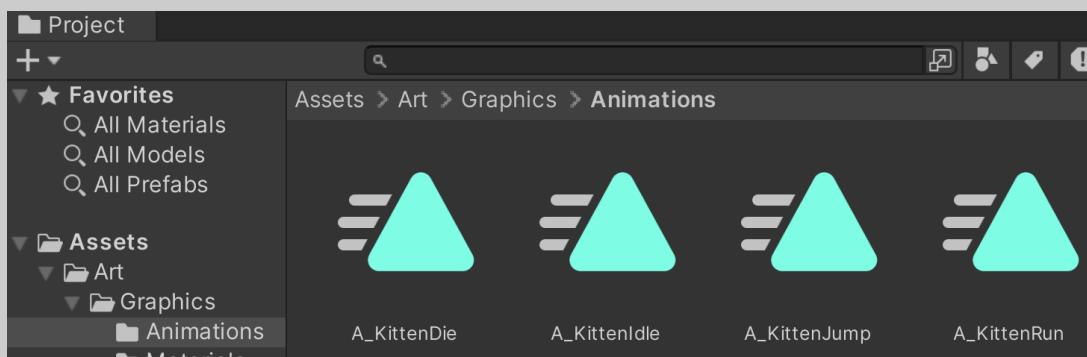


Nota: La animación de SP_KittenDie la he copiado de la carpeta bunny de la carpeta Resources.

Veremos que dentro de la carpeta Animations, además de las animaciones (los triángulos), se nos han creado varios “Animation Controller” (que si les dieramos doble click se abrirían en la ventana Animator):



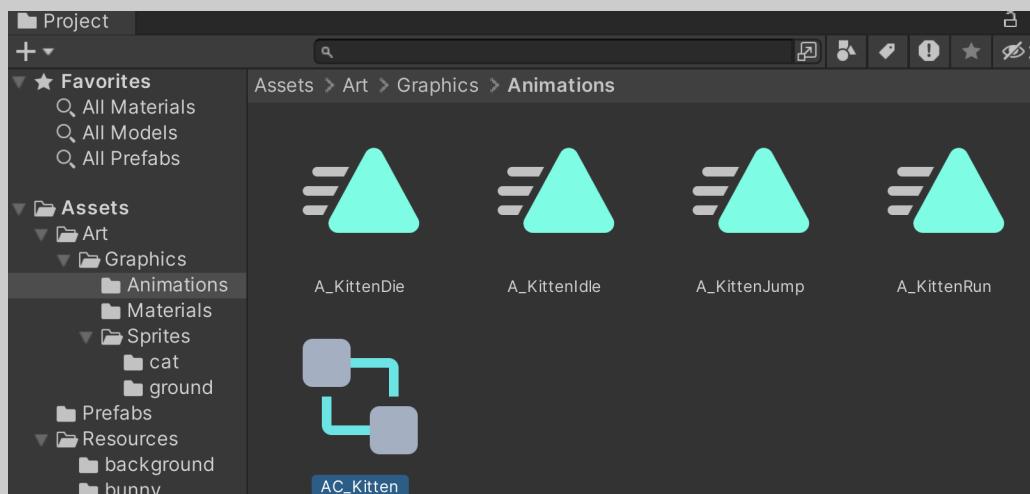
Borramos todos los Animators Controller y nos quedamos solo con las animaciones:



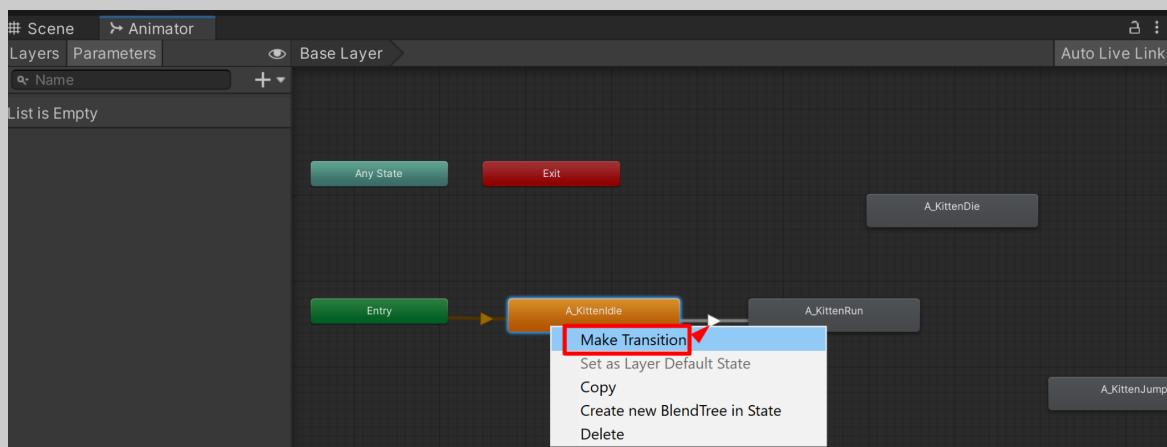
También podemos borrar de la escena todos los gameobjects que se crearon al hacer las animaciones, ya no los vamos a necesitar más:



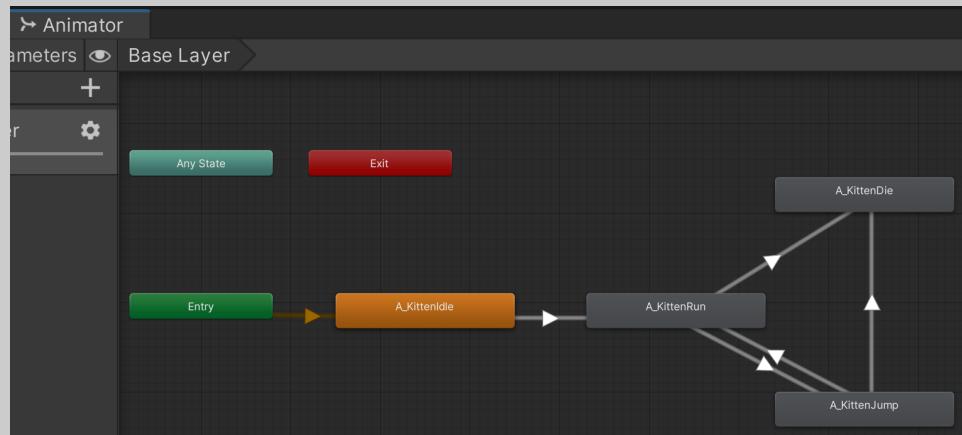
Vamos a crear un solo Animator Controller que “gobierne” todas las animaciones de nuestro personaje. Para ello, comenzaremos arrastrando, por ejemplo, la animación A_KittenIdle a nuestro GameObject Player, y veremos como se nos crea un Animator Controller al que nombraremos como AC_Kitten:



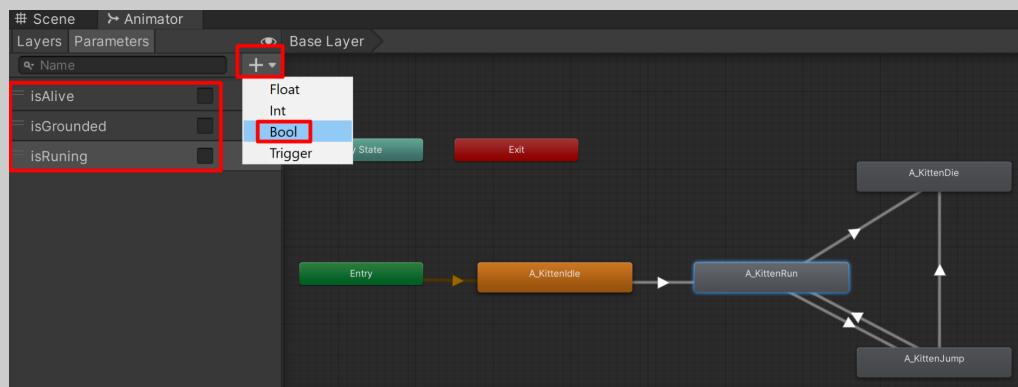
Hacemos doble click sobre el AC_Kitten y arrastramos el resto de animaciones a la ventana Animator. Luego, bastará con crear las transiciones entre estados pulsando con el botón derecho en una animación y seleccionando Make Transition y luego pinchamos en la animación que iría a continuación:



De la misma manera, creamos las transiciones entre el resto de animaciones de nuestro controlador de animaciones, de tal forma que todo cobre sentido:

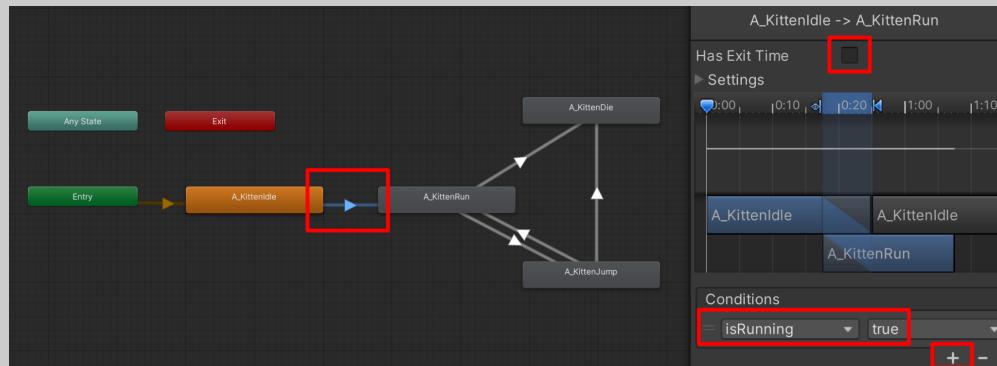


Para controlar el paso de una animación a otra a través de las transiciones, necesitaremos crear 3 parámetros booleanos: `isGrounded`, `isAlive` y `isRunning`.



Ahora ya podemos establecer todas las condiciones del paso de un estado a otro y en todas desactivaremos la casilla “Has Exit Time” para favorecer que empiece a ejecutarse una animación aunque no haya terminado la anterior. Vamos a ello:

1.- Para pasar de Idle a Run la condición será que `isRunning` se ponga en true:



2.- Para pasar de Run a Die la condición será que `isAlive` se ponga en false.

3.- Para pasar de Jump a Die la condición será que `isAlive` se ponga en false.

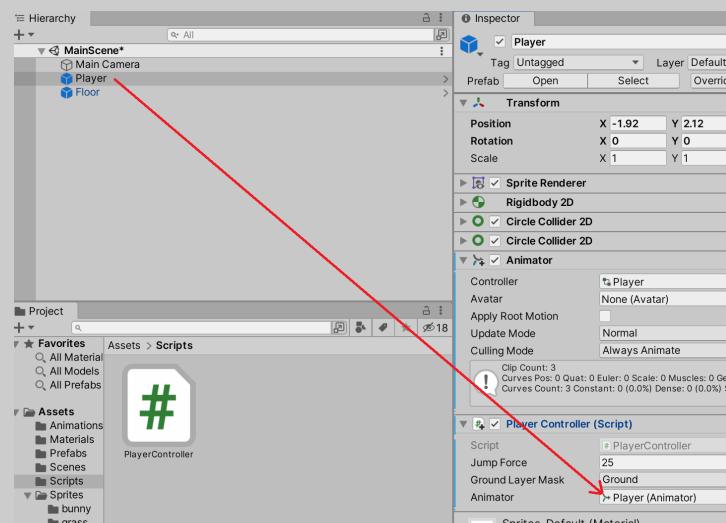
4.- Para pasar de Jump a Run la condición será que isGrounded se ponga en true.

5.- Para pasar de Run a Jump la condición será que isGrounded se ponga en false.

Gracias a estos parámetros que acabamos de establecer en cada transición, ya podemos controlar el cambio de estado de nuestro jugador desde el código y mediante la componente Animator:



A continuación arrastramos nuestro Player (que tiene un componente Animator) a nuestro Player Controller Script:



Nota: Una forma alternativa de hacer esta operación es que, en vez de arrastrar al Script el Animator en el inspector, hacerlo por código dentro del mismo script, concretamente en el método Awake a través del método GetComponent:

```

private void Awake()
{
    _animatorPlayer = GetComponent<Animator>();
}

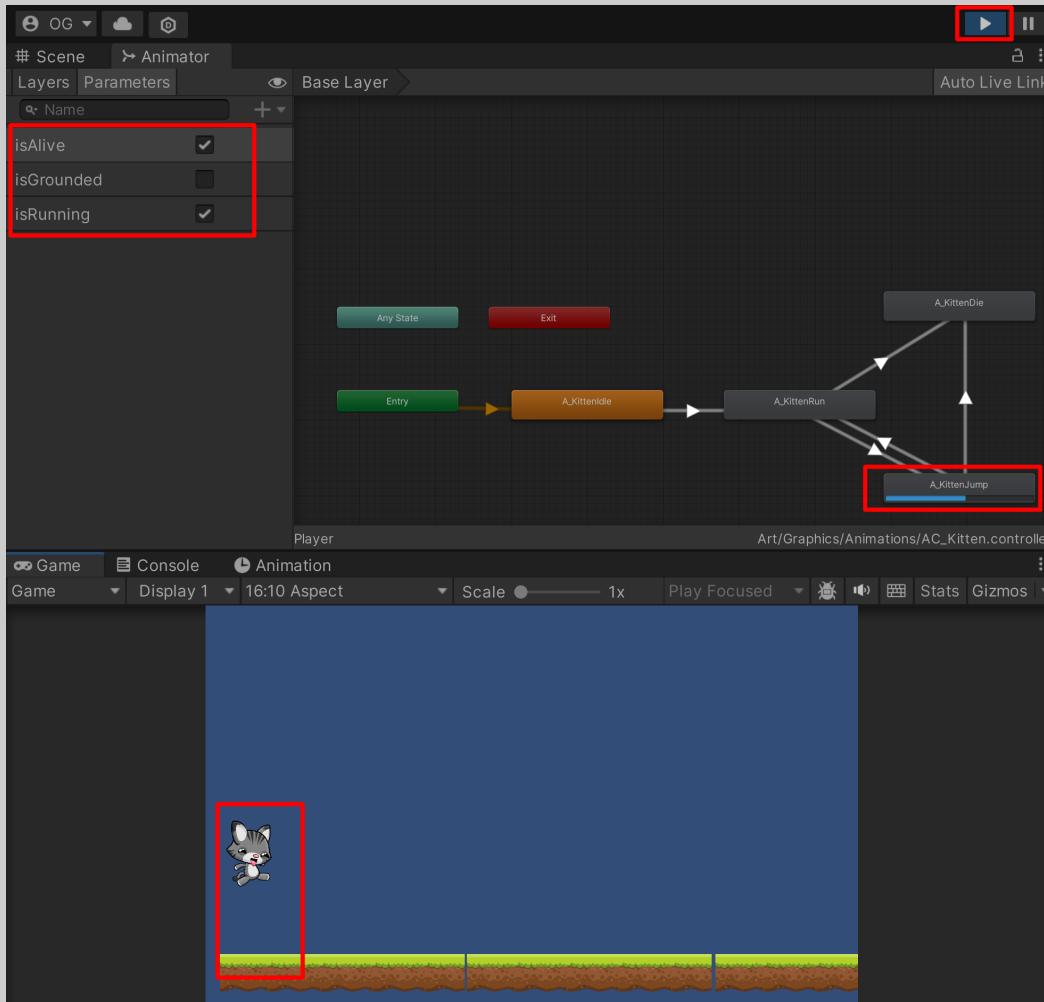
```

Como hemos explicado en clase, a efectos de rendimiento nos da igual hacerlo de un modo u otro pero, si elegimos hacerlo por código, en un futuro nos ayudaría a trasladar fácilmente este script a otros proyectos.

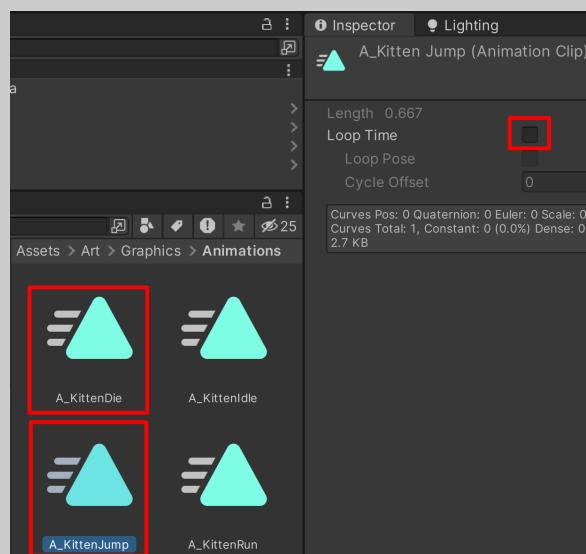
Ahora ya podemos acceder a los parámetros del Animator y empezar a programarlo de la siguiente manera:

```
C# PlayerController.cs
1  using UnityEngine;
2  public class PlayerController : MonoBehaviour
3  {
4      public float jumpForce = 6.0f; //Fuerza de salto para multiplicarsela a Vector2.up
5      private Rigidbody2D _rigidbody2d; //Para guardar la referencia al componente Rigidbody2D del Player
6      private float _distanceRaycast = 1.0f; //Longitud del Raycast
7      public LayerMask groundLayerMask; //Seleccionaremos en el inspector la capa deseada
8      private Animator _animatorPlayer; //Para guardar la referencia al componente Animator del Player
9      private bool _isRunning = false; //Para controlar que empiece parado y al pulsar un botón empiece a correr
10     private void Awake()
11     {
12         _animatorPlayer = GetComponent<Animator>();
13         _rigidbody2d = GetComponent<Rigidbody2D>();
14     }
15     void Start()
16     {
17         _animatorPlayer.SetBool("isAlive", true);
18         /*_animatorPlayer.SetBool("isRunning", false); No haría falta ya que, por defecto, los parámetros del
19          Animator Controller se inicializan en falso */
20     }
21     void Update()
22     {
23         _animatorPlayer.SetBool("isGrounded", value: isOnTheFloor());
24         if (Input.GetMouseButton(0)) //si pulsamos el botón izquierdo del ratón
25         {
26             if (_isRunning == false) //Solo ocurre al principio de la partida
27             {
28                 _isRunning = true;
29                 _animatorPlayer.SetBool("isRunning", _isRunning);
30             }
31             else
32             {
33                 if (isOnTheFloor())
34                 {
35                     Jump();
36                 }
37             }
38         }
39     }
}
```

Comprobamos que los cambios de estado se realicen de manera correcta:



Fíjate que, mientras que las animaciones de Idle y de Run sí que nos interesa que se repitan en bucle indefinidamente, la de Jump y la de Die no, estás últimas solo quiero que se ejecuten una sola vez, por lo que les desmarcamos su casilla de "Loop Time":



Para terminar este punto solo nos falta que el conejito se desplace.

Nota: Si fuese un plataformas 2D normal (no un Infinite Runner como el que estamos haciendo), cogeríamos la lectura del mando (izquierda o derecha) y moveríamos al personaje:

```
void Update () {
    float inputX = Input.GetAxis("Horizontal") * moveSpeed;
    _rigidbody2D.velocity = new Vector2(inputX,_rigidbody2D.velocity.y);
}
```

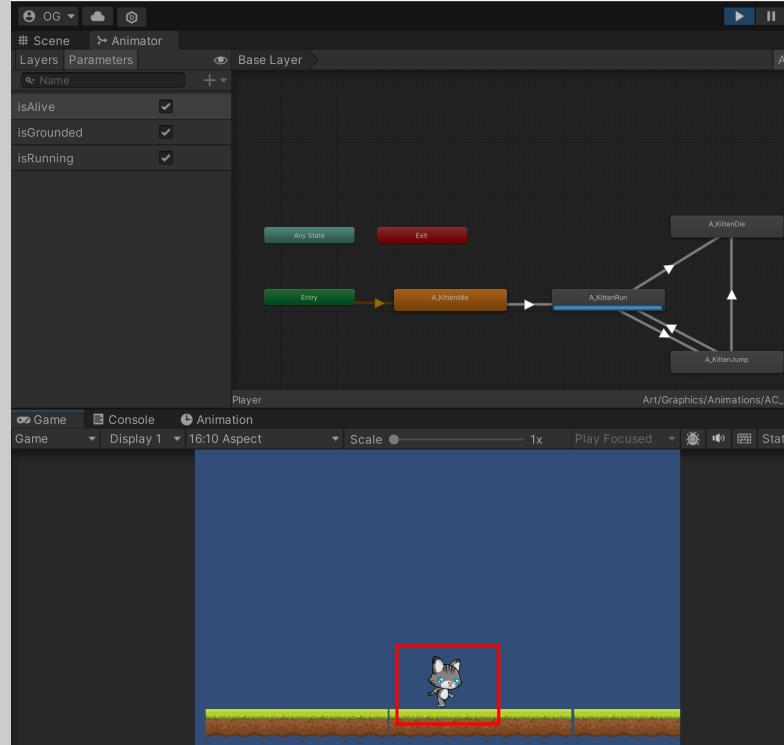
Para que el personaje mire en la dirección hacia la que se mueve bastaría con activar el FlipX del componente Sprite Renderer o también, otra solución, es poner a -1 el eje X del parámetro Scale del Transform:

```
if (_rigidbody2D.velocity.x<0){
    transform.localScale = new Vector3(-1,1,1); }
else if (_rigidbody2D.velocity.x>0){
    transform.localScale = new Vector3.one; }
```

Igual que para saltar teníamos que aplicar una fuerza en un momento dado (con la pulsación del ratón), ahora para correr tendremos que aplicarle una velocidad constante hacia la derecha. Esta velocidad, en vez de aplicarla en el método Update, lo haremos en el FixedUpdate...¿por qué? Porque el método Update, a pesar de ejecutarse cada frame, estos frames no tienen que estar equiespaciados, por lo que podríamos ver tirones en el scroll del juego. En cambio, el método FixedUpdate si se ejecuta cada un tiempo fijo por lo que es el método correcto donde aplicar fuerzas constantes a la física (por ejemplo: para velocidades constantes o para movimientos que siempre se estén ejecutando). En nuestro caso aplicaremos una velocidad en el eje x de 2.0f, y dejaremos la velocidad que tenía en el eje y:

```
C# PlayerController.cs ×
1  using UnityEngine;
2  public class PlayerController : MonoBehaviour
3  {
4      public float jumpForce = 6.0f; //Fuerza de salto para multiplicarsela a Vector2.up
5      private Rigidbody2D _rigidbody2D; //Para guardar la referencia al componente Rigidbody2D del Player
6      private float _distanceRaycast = 1.0f; //Longitud del Raycast
7      public LayerMask groundLayerMask; //Seleccionaremos en el inspector la capa deseada
8      private Animator _animatorPlayer; //Para guardar la referencia al componente Animator del Player
9      private bool _isRunning = false; //Para controlar que empieze parado y al pulsar un botón empieze a correr
10     public float runningSpeed = 6.0f; //Unchanged
11
12     private void FixedUpdate()
13     {
14         if (_isRunning==true) //Para que no empiece a correr nada más empezar el juego (en estado Idle)
15         {
16             if (_rigidbody2D.velocity.x < runningSpeed)
17             {
18                 _rigidbody2D.velocity = new Vector2(runningSpeed, _rigidbody2D.velocity.y);
19             }
20         }
21     }
}
```

Comprobamos que todo funcione correctamente:



Antes de continuar codificando, vamos a optimizar el código que ya tenemos hecho hasta ahora. Concretamente, es mucho más eficiente que el método SetBool del Animator reciba un entero en vez de una cadena y además esto nos evitaría errores de “caligrafía” más adelante. Para ello, vamos a comenzar creandnos tres nuevas variables utilizando el método StringToHash:

```
C# PlayerController.cs < C# PlayerController2.cs >
1  using UnityEngine;
2  public class PlayerController : MonoBehaviour
3  {
4      public float jumpForce = 6.0f; //Fuerza de salto para multiplicarsela a Vector2.up
5      public LayerMask groundLayerMask; //Seleccionaremos en el inspector la capa deseada
6      public float runningSpeed = 6.0f;
7
8      private Rigidbody2D _rigidbody2d; //Para guardar la referencia al componente Rigidbody2D del Player
9      private float _distanceRaycast = 1.0f; //Longitud del Raycast
10     private Animator _animatorPlayer; //Para guardar la referencia al componente Animator del Player
11     private bool _isRunning = false; //Para controlar que empiece parado y al pulsar un botón empiece a correr
12
13     private int _animIdIsAlive = Animator.StringToHash("isAlive");
14     private int _animIdIsRunning = Animator.StringToHash("isRunning");
15     private int _animIdIsGrounded = Animator.StringToHash("isGrounded");
}

```

Y ahora, procedemos a utilizarlas en los métodos SetBool del Animator:

```
C# PlayerController.cs ×

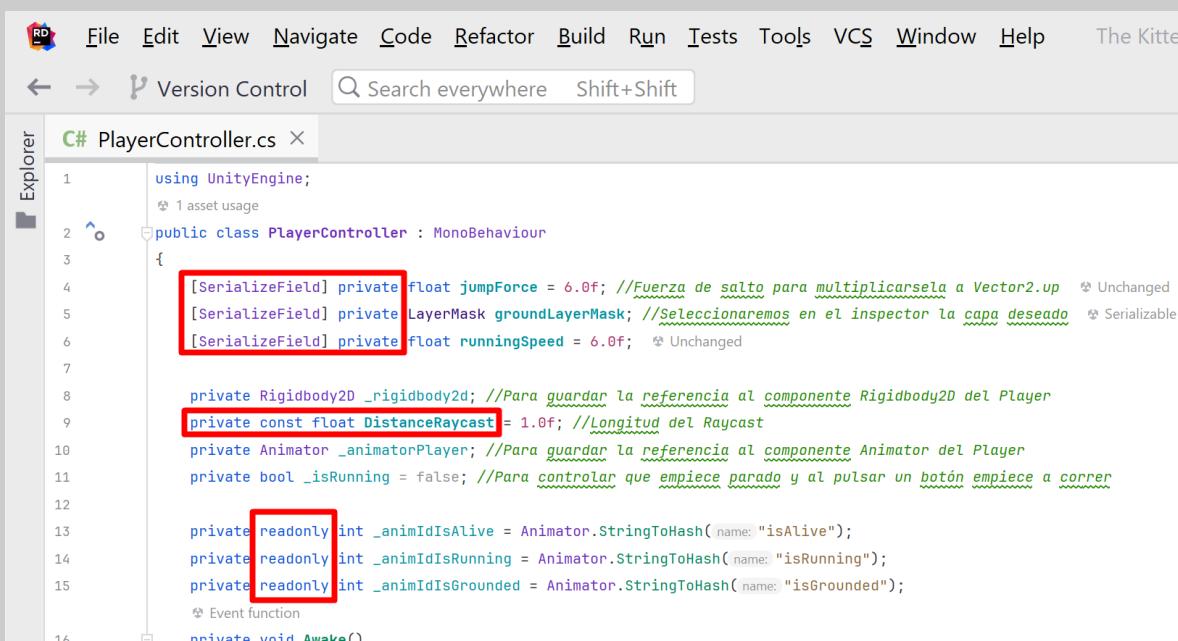
1  using UnityEngine;
  ↵ 1 asset usage
2  public class PlayerController : MonoBehaviour
3  {
4      public float jumpForce = 6.0f; //Fuerza de salto para multiplicarsela a Vector2.up ↵ Unchanged
5      public LayerMask groundLayerMask; //Seleccionaremos en el inspector la capa deseada ↵ Serializable
6      public float runningSpeed = 6.0f; ↵ Unchanged
7
8      private Rigidbody2D _rigidbody2D; //Para guardar la referencia al componente Rigidbody2D del Player
9      private float _distanceRaycast = 1.0f; //Longitud del Raycast
10     private Animator _animatorPlayer; //Para guardar la referencia al componente Animator del Player
11     private bool _isRunning = false; //Para controlar que empiece parado y al pulsar un botón empiece a correr
12
13     private int _animIdIsAlive = Animator.StringToHash( name: "isAlive" );
14     private int _animIdIsRunning = Animator.StringToHash( name: "isRunning" );
15     private int _animIdIsGrounded = Animator.StringToHash( name: "isGrounded" );
  ↵ Event function
16     private void Awake()
17     {
18         _animatorPlayer = GetComponent<Animator>();
19         _rigidbody2D = GetComponent<Rigidbody2D>();
20     }
  ↵ Event function
21     void Start()
22     {
23         _animatorPlayer.SetBool( _animIdIsAlive, value: true );
24     }
  ↵ Event function
25     void Update()
26     {
27         _animatorPlayer.SetBool( _animIdIsGrounded, value: isOnTheFloor() );
28         if ( Input.GetMouseButton(0) //Si pulsamos el botón izquierdo del ratón
29         {
30             if ( _isRunning == false ) //Solo ocurre al principio de la partida
31             {
32                 _isRunning = true;
33                 _animatorPlayer.SetBool( _animIdIsRunning, _isRunning );
34             }
35             else
36             {
37                 if ( isOnTheFloor() )
38                 {
39                     Jump();
  
```

Otra optimización que le podemos hacer a nuestro código es que siempre es preferible que las variables sean privadas para así evitar modificarlas accidentalmente desde otras clases. En nuestro caso tenemos 3 variables públicas que, en su momento, nos interesó hacerlas así para que estuvieran “expuestas” en el inspector y así modificar su valor cómodamente desde él, ya que si fueran variables privadas no aparecerían en el inspector. Existe una manera de que una variable, aunque sea privada, aparezca expuesta en el inspector, y esta manera es “serializandola”. Para ello, bastará con poner delante de ella la instrucción [SerializeField]. (“serializar” es el proceso de convertir el estado de un objeto en un formato que se pueda almacenar o transportar. El complemento de serialización y deserialización, que convierte una secuencia en un objeto. Juntos, estos procesos permiten almacenar y transferir datos.)

Por último, también Rider nos ofrece un par de consejos a los que también podremos hacer caso si así lo creemos conveniente: 1: establecer la variable _distanceRaycast como constante y 2: establecer las variables del tipo _animIdRun como readonly.

Nota: Diferencia entre readonly y const: - readonly: Solo se pueden asignar en la declaración de la variable o en el constructor de la clase. Luego ya NO se pueden modificar. - const: Solo se pueden asignar en la declaración de la variable y pertenecen a la clase (como static) NO son atributos de un objeto sino de la clase.

Siguiendo estos 2 consejos (stringtohash y serializar), más los 2 que nos ofrece Rider, vamos a modificar nuestro script consecuentemente:



```

C# PlayerController.cs ×
1  using UnityEngine;
2  public class PlayerController : MonoBehaviour
3  {
4      [SerializeField] private float jumpForce = 6.0f; //Fuerza de salto para multiplicarsela a Vector2.up
5      [SerializeField] private LayerMask groundLayerMask; //Seleccionaremos en el inspector la capa deseada
6      [SerializeField] private float runningSpeed = 6.0f; //Unchanged
7
8      private Rigidbody2D _rigidbody2D; //Para guardar la referencia al componente Rigidbody2D del Player
9      private const float DistanceRaycast = 1.0f; //Longitud del Raycast
10     private Animator _animatorPlayer; //Para guardar la referencia al componente Animator del Player
11     private bool _isRunning = false; //Para controlar que empiece parado y al pulsar un botón empiece a correr
12
13     private readonly int _animIdIsAlive = Animator.StringToHash(name: "isAlive");
14     private readonly int _animIdIsRunning = Animator.StringToHash(name: "isRunning");
15     private readonly int _animIdIsGrounded = Animator.StringToHash(name: "isGrounded");
16     private void Awake()

```

5. Desarrollo del juego: El Game Manager

En este punto desarrollaremos toda la lógica del juego, lo que comúnmente llamamos el “Game Play”: Inicio del juego, el “game over”, el restart, que todos los actores de nuestro juego se comporten como es debido en la escena, eventos que ocurren en el juego...

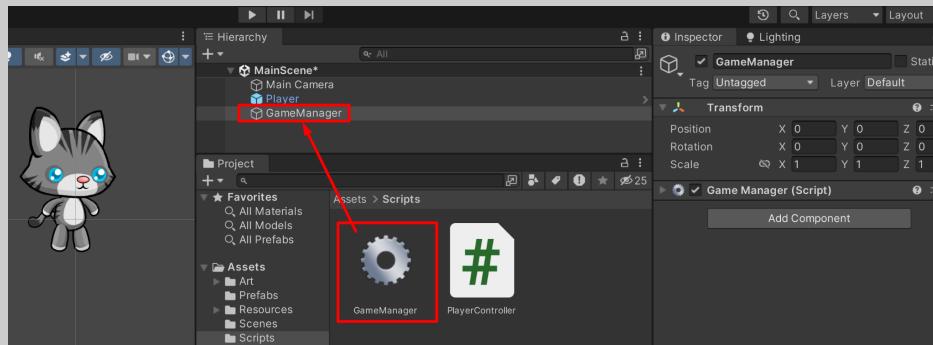
Lo normal será tener corriendo en un juego una única instancia del objeto gameManager que controle los eventos del “Game Play”, salvo que nuestro juego tenga variedad de situaciones donde tendría sentido tener varios Game Manager (por ejemplo, en un juego tipo GTA donde en un momento dado estás conduciendo un vehículo, o en una galería de tiro, o en un partido de tenis...)

Crearemos el script GameManager en la carpeta correspondiente y le vamos a introducir 3 métodos, uno para iniciar la partida, otro cuando el jugador muere y otro cuando el jugador decide finalizar y volver al menú principal:



```
C# GameManager.cs ×
1 using ...
4 ◌ No asset usages
5 public class GameManager : MonoBehaviour
6 {
7     public void StartGame()
8     {
9     }
10 }
11 public void GameOver()
12 {
13 }
14 }
15 }
16 public void BackToMainMenu()
17 {
18 }
19 }
20 }
```

Este script se lo asignaremos a un objeto vacío que vamos a crear en la escena con el nombre, por ejemplo, “GameManager” también:



Ahora, vamos a mejorar el script GameManager de tal forma que tengamos un método que gestione los 3 métodos que hemos visto antes y sea capaz de decidir cual debe ejecutarse. Esto es lo que se llama un “Game State” y para su gestión se podría utilizar, por ejemplo, un tipo de dato enumerado que en nuestro juego tomaría 3 valores: menu, inTheGame y gameOver:

```
C# PlayerController.cs x C# GameManager.cs x
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public enum GameState
7  {
8      menu,
9      inTheGame,
10     gameOver
11 }
12
13 public class GameManager : MonoBehaviour
14 {
15     //declaramos currentState del tipo enumerado GameState y lo inicializamos al valor menu
16     public GameState currentState = GameState.menu; // Unchanged
17
18     void Start()
19     {
20         StartGame();
21     }
22
23     public void StartGame() //Se llama para iniciar la partida
24     {
25     }
26     public void GameOver() //Se llama cuando el jugador muere
27     {
28     }
29
30     //lo llamamos cuando el jugador decide finalizar y volver a menú principal
31     public void BackToMainMenu()
32     {
33     }
34 }
```

Si te fijas en la captura, en el método Start llamaríamos al método StartGame. En ese método cambiaríamos el estado de nuestro juego a "inTheGame" ya que lo habíamos inicializado al estado "menu". En el método GameOver cambiaríamos el estado de nuestro juego a "gameOver", y en el método BacktoMainMenu cambiaríamos el estado de nuestro juego a "menu".

Realmente, los 3 métodos hacen lo mismo, esto es, cambian el estado del juego, por lo tanto mejor voy a hacer un método privado (ChangeGameState) que se encargue de ello:

```

public class GameManager : MonoBehaviour
{
    //declaramos currentState del tipo enumerado GameState y lo inicializamos al valor menu
    public GameState currentState = GameState.menu; ◄ Unchanged

    ◄ Event function
    void Start()
    {
        StartGame();
    }

    □ 1 usage
    public void StartGame() //Se llama para iniciar la partida
    {
        ChangeGameState(GameState.inTheGame);
    }

    public void GameOver() //Se llama cuando el jugador muere
    {
        ChangeGameState(GameState.gameOver);
    }

    //lo llamamos cuando el jugador decide finalizar y volver a menú principal
    public void BackToMainMenu()
    {
        ChangeGameState(GameState.menu);
    }

    □ 3 usages
    void ChangeGameState(GameState newGameState)
    {
        if (newGameState == GameState.menu)
        {
            //La escena de Unity deberá mostrar el menú principal
        }
        else if (newGameState == GameState.inTheGame)
        {
            //La escena de Unity deberá configurarse para mostrar el juego en si
        }
        else if (newGameState == GameState.gameOver)
        {
            //La escena de Unity deberá mostrar el menú de fin de partida
        }
    }
}

```

Ya tenemos preparado el GameManager. De momento aún no hace nada ya que tenemos programado por un lado el GameManager y por otro el PlayerController, cada uno por su lado, por lo que tendremos que escribir un poco más de código para que estas dos clases se conozcan y que el GameManager gobierne al PlayerController. La forma típica de hacer esto sería ir a nuestro conejito y crearle en su Script PlayerController una variable pública donde arrastraremos el GameManager y así ya tendríamos una instancia de este. El problema es que en este caso no podemos hacer esto ya que entonces tendríamos que arrastrar el GameManager a cada gameobject de nuestro juego (a los enemigos, a los items, etc...) y hemos dicho que lo ideal es tener un único GameManager, no múltiples instancias de este. Para solucionar esto, tenemos que saber que cuando necesitamos compartir un objeto entre varias clases de nuestro juego, se suele utilizar el patrón de programación Singleton.

Si aplicamos el patrón Singleton al GameManager podremos acceder a este desde cualquier punto de nuestro proyecto y tener una única instancia de este.

Singleton o instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón Singleton se implementa creando en nuestra clase un método que crea una instancia del objeto solo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con modificadores de acceso como protegido o privado). Fuente: [Wikipedia](#)

Para ello, bastará con que en la clase GameManager creamos un atributo estático del tipo GameManager al que luego, en el método Awake le demos como valor la misma clase en la que se encuentra:

```

Controller.cs × C# GameManager.cs ×
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
[9 usages] [4 exposing APIs]
public enum GameState
{
    menu,
    inTheGame,
    gameOver
}
[1 asset usage] [1 usage] [1 exposing API]
public class GameManager : MonoBehaviour
{
    //shareInstance será única y compartida con toda la escena de Unity
    public static GameManager sharedInstance;

    //declararemos currentState del tipo enumerado GameState y lo inicializamos al valor menu
    public GameState currentState = GameState.menu; [Unchanged]
    [Event function]
    private void Awake()
    {
        sharedInstance = this;
    }
}

```

Podemos mejorar el método Awake asegurándonos de que solo va a ver una única instancia compartida, incluso podemos añadir la instrucción DontDestroyOnLoad(gameObject) si nos interesa que el GameManager no se destruya al cambiar de una escena a otra (en este primer proyecto nos va a dar igual ya que únicamente vamos a tener una escena)

```

private void Awake()
{
    if (sharedInstance!=null && sharedInstance!=this)
    {
        Destroy(obj:this);
    }
    else
    {
        sharedInstance = this;
        DontDestroyOnLoad(gameObject); //En este proyecto no nos hará falta
    }
}

```

La primera operación que vamos a hacer es controlar en el PlayerController que el conejito comience a desplazarse si entra en el estado inTheGame, y que mientras esté en el estado menú no se desplace. Para ello modificamos el script PlayerController:

```
C# PlayerController.cs × C# GameManager.cs ×

14     private void FixedUpdate()
15     {
16         //Solo corre si estamos en el estado inTheGame
17         if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
18         {
19             if (_rigidbody2D.velocity.x < runningSpeed)
20             {
21                 _rigidbody2D.velocity = new Vector2(runningSpeed, _rigidbody2D.velocity.y);
22             }
23         }
24     }
25 
26     void Update()
27     {
28         //Solo salta si estamos en el estado inTheGame
29         if (GameManager.sharedInstance.currentGameState == GameState.inTheGame)
30         {
31             animator.SetBool(name: "isGrounded", value: isOnTheFloor());
32             if (Input.GetMouseButton(0)) // Si pulsamos el botón izquierdo del ratón
33             {
34                 if (isOnTheFloor())
35                 {
36                     Jump();
37                 }
38             }
39         }
40     }
41 }
```

El siguiente paso será el de modificar el GameManager de tal forma que cuando pulsemos un botón (recuerda, la lectura de entradas la hacemos en el método Update) el juego entre en el estado inTheGame. (de momento, mientras no tengamos un menú con botones y opciones lo haremos así, por lo que más adelante lo cambiaremos)

Para la lectura de entradas del jugador se suelen utilizar uno de estos 2 métodos:

- `Input.GetKeyDown`
- `Input.GetButtonDown`

En la documentación de Unity se nos muestran sus diferencias:

- <https://docs.unity3d.com/ScriptReference/Input.GetKeyDown.html>

Input.GetKeyDown

Declaration

```
public static bool GetKeyDown(string name);
```

Description

Returns true during the frame the user starts pressing down the key identified by `name`.

Call this function from the Update function, since the state gets reset each frame. It will not return true until the user has released the key and pressed it again.

For the list of key identifiers see Conventional Game Input. When dealing with input it is recommended to use Input.GetAxis and Input.GetButton instead since it allows end-users to configure the keys.

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown("space"))
        {
            print("space key was pressed");
        }
    }
}
```

- <https://docs.unity3d.com/ScriptReference/Input.GetButtonDown.html>

Input.GetButtonDown

[Leave feedback](#)

Declaration

```
public static bool GetButtonDown(string buttonName);
```

Description

Returns true during the frame the user pressed down the virtual button identified by `buttonName`.

Call this function from the Update function, since the state gets reset each frame. It will not return true until the user has released the key and pressed it again.

Use this only when implementing action like events IE: shooting a weapon.

Use Input.GetAxis for any kind of movement behavior.

To edit, set up, or remove buttons and their names (such as "Fire1"): 1. Go to Edit > Project Settings > Input Manager to bring up the Input Manager. 2. Expand Adds by clicking the arrow next to it. This shows the list of the current buttons you have. You can use one of these as the parameter "buttonName". 3. Expand one of the items in the list to access and change aspects such as the buttons name and the key, joystick or mouse movement that triggers it. 4. For more information about buttons, see the Input Manager page

```
using UnityEngine;
using System.Collections;

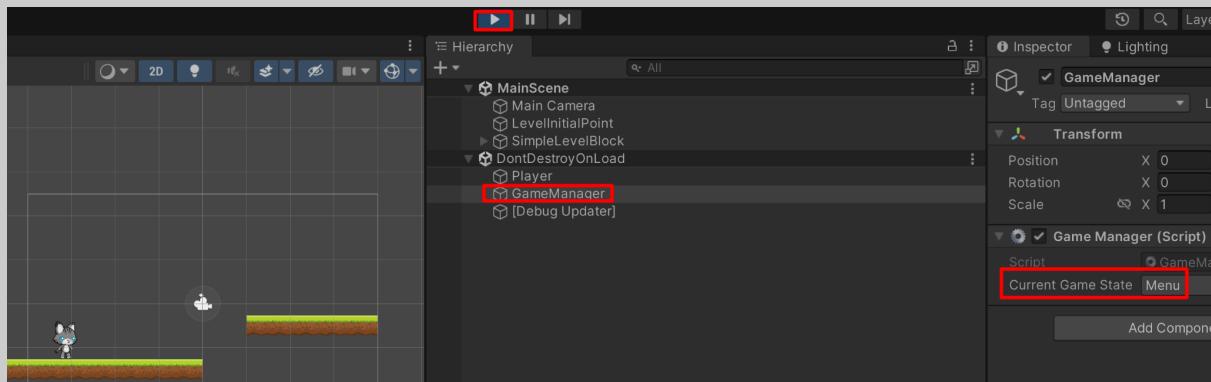
public class ExampleClass : MonoBehaviour
{
    public GameObject projectile;
    void Update()
    {
        if (Input.GetButtonDown("fire1"))
            Instantiate(projectile, transform.position, transform.rotation);
    }
}
```

En nuestro caso vamos a usar `Input.GetButtonDown` ya que nos va a dar más versatilidad si en un futuro queremos usar, por ejemplo, un Gamepad y no solo limitarnos a usar el teclado. Por defecto, si entramos en Edit>Project Settings>Input Manager veremos como nuestro juego acepta 30 entradas distintas. Podemos dejarlo así, y utilizar, por ejemplo, la de "Fire 1" (botón Ctrl Left y como alternativo el Botón derecho de ratón), o podemos dejar solo una entrada en vez de las 18.

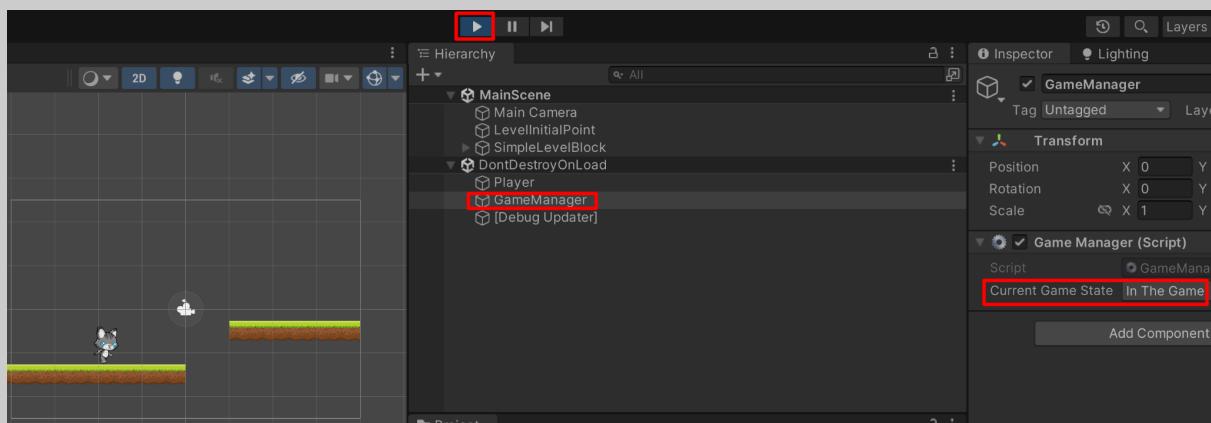
```
ontroller.cs × C# GameManager.cs ×
private void Start()
{
    currentGameState = GameState.menu; //El juego inicia en este estado
}
⌚ Event function
private void Update()
{
    if (currentGameState != GameState.inTheGame)
    {
        Debug.Log(message: "Comienzo");
        StartGame();
    }
}
⌚ Frequently called 1 usage
public void StartGame() //Se le llama para iniciar la partida
{
    ChangeGameState(GameState.inTheGame);
}
⌚ 1 usage
public void GameOver() //Se le llama cuando el jugador muere
{
    ChangeGameState(GameState.gameOver);
}

public void BackToMainMenu() //Se le llama cuando el jugador decide finalizar y volver al menú ppal.
{
    ChangeGameState(GameState.menu);
}

⌚ Frequently called 3 usages
void ChangeGameState(GameState newGameState)
{
    if (newGameState == GameState.menu)
    {
        //Mostrar canvas/menú principal
        currentGameState = GameState.menu;
    }else if (newGameState == GameState.inTheGame)
    {
        //Mostrar el gatito en acción
        currentGameState = GameState.inTheGame;
    }else if ((newGameState == GameState.gameOver))
    {
        //Mostrar canvas/menú gameover
        currentGameState = GameState.gameOver;
    }
}
```

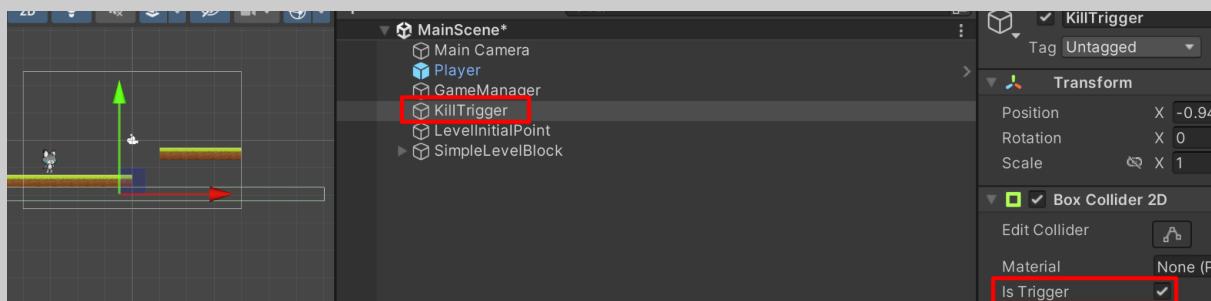


El personaje no se desplaza. Pero pulsamos la tecla CtrlLeft y...



Ahora vamos a programar cuándo debe entrar en el estado gameOver nuestro juego. Para ello vamos a configurar un nuevo elemento en la pantalla, concretamente un Collider que va a funcionar a modo de Trigger (desencadenante). Lo que queremos lograr es que cuando se establezca un contacto entre nuestro personaje y una zona de caída debajo de la escena, se lance un Trigger de un determinado evento; en nuestro caso el evento será el estado de gameOver.

Para ello, creamos un GameObject vacío, le llamamos, por ejemplo, KillTrigger, le añadimos un componente BoxCollider2D y lo colocamos debajo de la escena. Recuerda marcar la opción "Is Trigger" para que el conejito atraviese el KillTrigger y no se quede andando sobre él:



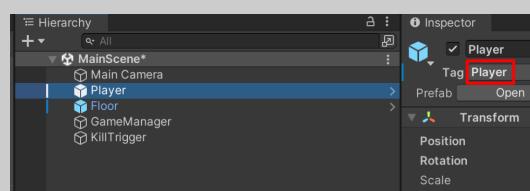
A continuación vamos a programar la “muerte” del protagonista: lo que queremos es que se desencadene una acción a partir de colisionar el Player con el KillTrigger. De los scripts que tenemos, en el GameManager no tiene mucho sentido programar la lógica del gameObject KillTrigger, ya que este script solo conoce de estados y se encarga de dirigir la acción del juego; en el PlayerController pues tampoco tiene sentido porque es el que se encarga de manejar los comportamientos del jugador, que de momento, solo sabe ir para adelante y saltar, y en el que más adelante añadiremos el comportamiento de morir.

Lo correcto será crear un nuevo script al que vamos a llamar KillTrigger (el desencadenante de la muerte) donde no necesitaré los métodos ni de Start ni de Update sino que utilizaré el método OnTriggerEnter2D, que es un método de la clase MonoBehaviour al que se le llama automáticamente cuando nuestro GameObject (KillTrigger) entra en contacto con otro (Player):

<https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter2D.html>

The screenshot shows the Unity Documentation interface. On the left, there's a sidebar with a tree view of Unity classes under 'Version: 2020.3'. The 'MonoBehaviour' node is expanded, showing 'OnTriggerEnter2D(Collider2D)' as a child. The main content area has the title 'MonoBehaviour.OnTriggerEnter2D(Collider2D)'. Below it, there's a 'Parameters' section with a single parameter 'other' described as 'The other Collider2D involved in this collision.' There's also a 'Description' section with detailed information about the method's behavior and notes about trigger events.

Comenzamos poniendo al gameobject Player la etiqueta (tag) “Player”, que es una de las etiquetas que ya nos trae unity por defecto:



A continuación implementamos en siguiente script, se lo asignamos al gameObject KillTrigger y probamos su funcionamiento:

The screenshot shows the Unity Editor with three tabs open: 'PlayerController.cs', 'GameManager.cs', and 'KillTrigger.cs'. The 'KillTrigger.cs' tab is active and contains the following C# code:

```

using UnityEngine;
public class KillTrigger : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.CompareTag("Player"))
        {
            Debug.Log(message: "Nuestro protagonista Player ha muerto");
        }
    }
}

```

Si funciona correctamente y muestra el mensaje por consola, entonces vamos a nuestro PlayerController donde junto a las comportamientos de correr y saltar añadiremos el comportamiento de morir. Lo haremos en un método público KillPlayer (que será llamado desde el método OnTriggerEnter2D de la clase KillTrigger) dondé:

- Cambiaremos el estado del juego a gameOver (para que el GameManager tenga constancia)
- Cambiaremos a falso la variable “isAlive” del animator.

```
s × C# PlayerController.cs ×
public void KillPlayer()
{
    GameManager.sharedInstance.GameOver();
    animator.SetBool(name: "isAlive", value: false);
    Debug.Log(message: "You died");
}
```

Ahora, para llamar a este método desde OnTriggerEnter2D podemos volver a hacer el mismo “truco” que hicimos con el GameManager, el de crear un Singleton, esto es, una variable estática apuntando a mi mismo, ya que solo voy a tener un único personaje protagonista en el juego (no es lo mismo que los enemigos, que puede haber varios)

```
public class PlayerController : MonoBehaviour
{
    public static PlayerController sharedInstance;
    public float jumpForce = 20.0f;     ↗ "30"
    private Rigidbody2D _rigidbody2D;
    public LayerMask groundLayerMask;   ↗ Serializable
    public Animator animator;          ↗ Changed in 1 asset
    public float runningSpeed = 2.0f;   ↗ Unchanged
    ↗ Event function
    void Awake()
    {
        sharedInstance = this;
        this._rigidbody2D = GetComponent<Rigidbody2D>();
    }

    public void KillPlayer()
```

Gracias a esto, podemos ir al método OntriggerEnter2D y llamar al método KillPlayer del PlayerController a través de su instancia:

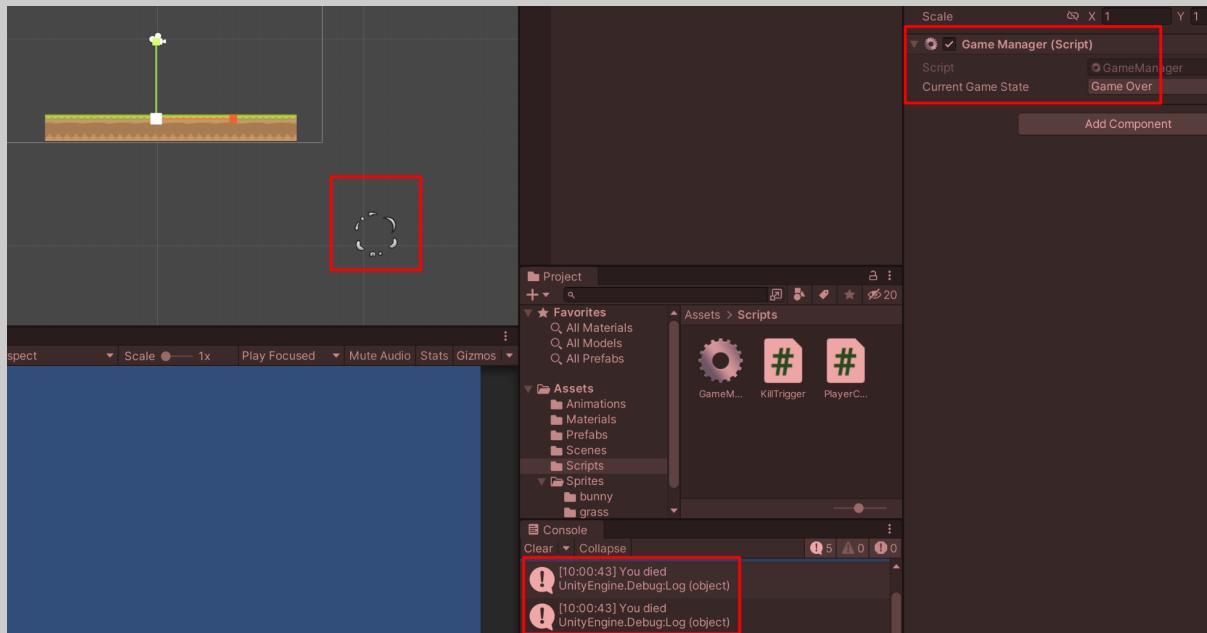


```

C# KillTrigger.cs × C# PlayerController.cs ×
1  using ...
2
3  public class KillTrigger : MonoBehaviour
4  {
5
6      public GameObject player;
7
8      private void OnTriggerEnter2D(Collider2D other)
9      {
10         if (other.tag == "Player")
11         {
12             PlayerController.sharedInstance.KillPlayer();
13         }
14     }
15 }
16

```

Vamos a comprobar que, efectivamente, el GameManager cambia su estado a GameOver y que se ejecuta la animación correspondiente:

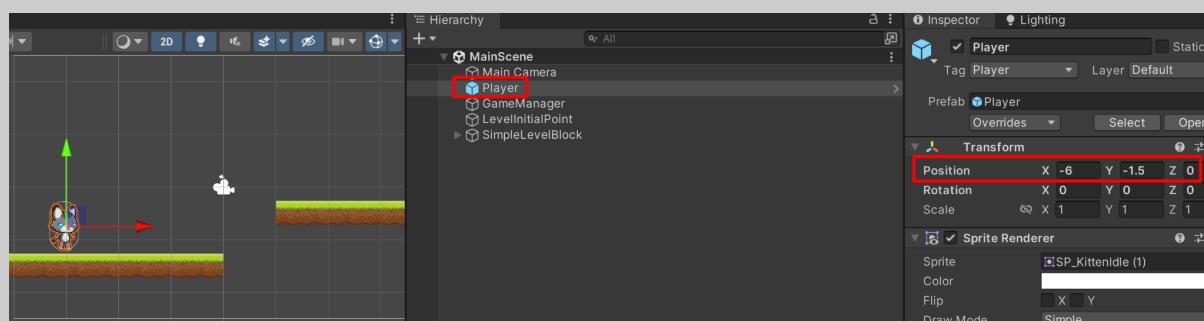


Para terminar este punto solo nos quedaría programar que el juego vuelva a empezar, un “restart game”, para así volver a jugar las veces que quiera el usuario. Para ello, el usuario pulsará un botón en la interfaz o, como no tenemos todavía una interfaz, pues pulsando un botón (como hicimos anteriormente para empezar partida y que el conejo comenzara a desplazarse)

Cada vez que nuestro juego empiece hay que “resetear” todas las variables al valor que tenían antes de empezar, como por ejemplo, la posición que tenía el protagonista. Para ello, vamos a crear una variable startPosition de tipo Vector3 que la iniciamos en el método Awake con la posición inicial que tiene nuestro personaje al iniciar el videojuego:

```
public class PlayerController : MonoBehaviour
{
    public static PlayerController sharedInstance;
    public float jumpForce = 20.0f; // "30"
    private Rigidbody2D _rigidbody2D;
    public LayerMask groundLayerMask; // Serializable
    public Animator animator; // Changed in 1 asset
    public float runningSpeed = 2.0f; // Unchanged
    private Vector3 startPosition;

    void Awake()
    {
        sharedInstance = this;
        startPosition = this.transform.position;
        this._rigidbody2D = GetComponent<Rigidbody2D>();
    }
}
```



Ahora vamos a hacer un pequeño cambio en el PlayerController renombrando el método Start a, por ejemplo, StartGame para que así no lo llame Unity automáticamente al iniciar el juego, sino que tome el mando el GameManager y lo llame cuando a nosotros nos interese, y así aprovechar para reiniciar la posición del conejito:

```
public float runningSpeed = 2.0f;     ↗ Unchanged
private Vector3 startPosition;
                                     ↗ Event function
void Awake()
{
    sharedInstance = this;
    startPosition = this.transform.position;
    this._rigidbody2D = GetComponent<Rigidbody2D>();
    animator.SetBool( name: "isAlive", value: true);
}
                                     ↗ Frequently called  ↗ 1 usage
public void StartGame()
{
    animator.SetBool( name: "isAlive", value: true);
    this.transform.position = startPosition;
}
                                     ↗ 1 usage
public void KillPlayer()
```

Ahora ya, desde el método StartGame del GameManager, llamaré a este nuevo método (que también se llama StartGame):

```
Assembly-CSharp > Assets > Scripts > C# GameManager.cs
C# PlayerController.cs X C# GameManager.cs X
                                     ↗ Frequently called  ↗ 1 usage
public void StartGame() //Se llama para iniciar la partida
{
    PlayerController.sharedInstance.StartGame();
    ChangeGameState(GameState.inTheGame);
}
                                     ↗ 1 usage
public void GameOver() //Se llama cuando el jugador muere
{
    ChangeGameState(GameState.gameOver);
}
//Lo llamamos cuando el jugador decide finalizar y volver a menú principal
public void BackToMainMenu()
```

Por último, en el método Update del GameManager indicamos que se reinicie la partida pero solo cuando no estemos en el estado inTheGame, ya que sino se reiniciaría el juego cada vez que pulsaramos el botón “Fire1” incluso aunque no hubieramos muerto:

```
private void Update()
{
    if (Input.GetButtonDown("Fire1"))
    {
        if (currentGameState != GameState.inTheGame)
        {
            StartGame();
        }
    }
}
```