Character Controller 2D

ESTÁS VIENDO LOS APUNTES EN FORMATO PDF. POR TANTO, TE PERDERÁS LOS GIFS. ES RECOMENDABLE VISUALIZARLOS DIRECTAMENTE ONLINE, EN
http://rpmi.metinu.com/character-controller-2d.html

Bienvenidos a los apuntes de cómo conseguir un personaje en movimiento 2D, al estilo de un Super Marioclásico.

La práctica al completo constará de los siguientes puntos:

- Creación del mapa de juego + Colliders del mapa + Plataformas unidireccionales
- Físicas 2D del personaje, pudiendo saltar y correr, sin fallos
- Seguimiento de cámara
- Animaciones del personaje (correr y saltar, principalmente)

Y, para pulir la experiencia, podríamos añadir otro elemento más, muy típico: plataformas móviles.

Con esta base terminada, podremos poblar los mapas de mecánicas interesantes, como saltos difíciles, trampas y enemigos. Luego, combinando estos elementos entre sí, tendremos la posibilidad de crear varios niveles con dificultad ascendente.

Siempre que queramos añadir complejidad al gameplay tendremos que volver a esta etapa y re-programar lo que sea necesario para lograr que los niveles que después creemos puedan contar con elementos más complejos. Por ejemplo, yendo un paso más allá, podríamos añadir físicas 2D del personaje avanzadas, tales como doble salto, salto de pared, dash, nadar, etc.

Crear mapa

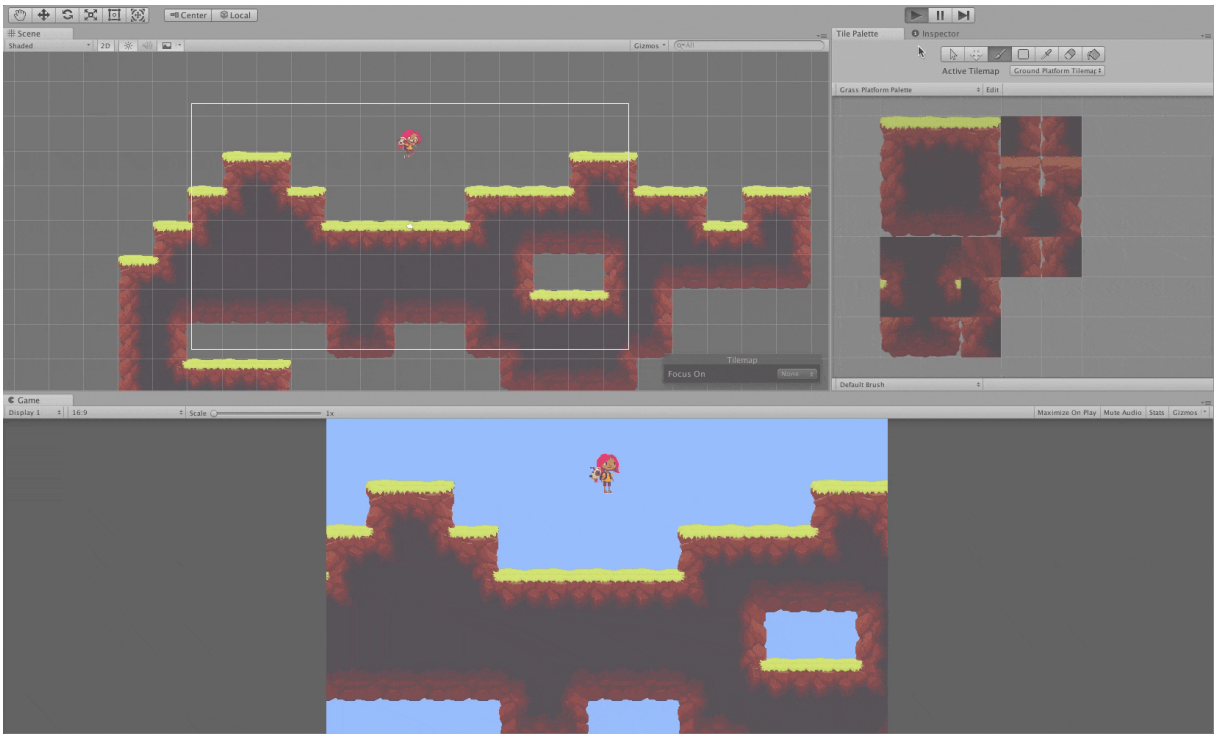
Crear el mapa podría consistir simplemente en arrastrar y soltar nuestros sprites de terreno y ponerles un collider. Sin embargo, cuando se trata de un juego más largo, lo ideal es utilizar el editor de mapas por tilemaps de Unity para aligerar el proceso.

Aunque al principio pueda parecer más costoso que arrastrar y soltar imágenes, a la larga ahorraremos tiempo y nos servirá para más situaciones.

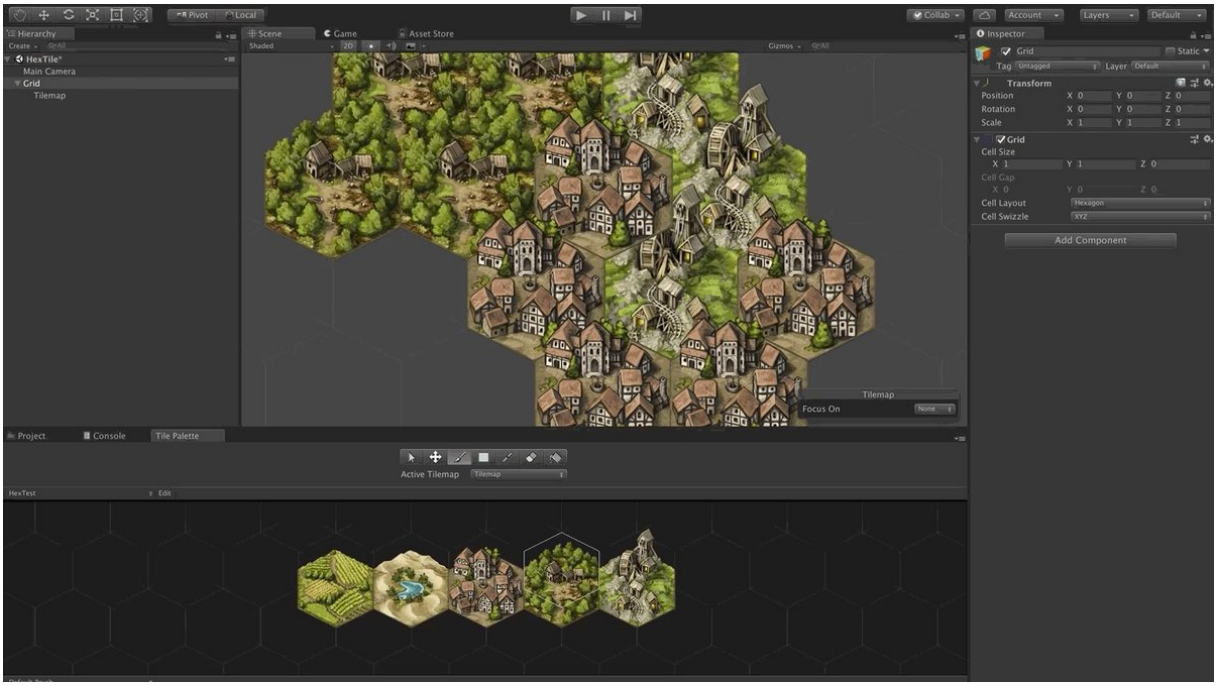
Esto es cierto siempre y cuando el estilo gráfico de nuestro juego se base en tiles reutilizables que se van repitiendo a lo largo de los niveles. Si se trata de un juego con más nivel de detalle, en que cada sprite es único y los niveles no siguen una estructura de casillas, entonces los tilemaps no nos servirán.

Observa la diferencia entre esto (sprite individuales y sin estructura):

y esto (sprites reutilizables y con estructura en tilemap cuadrado):



Esta técnica nos sirve también para crear mapas con estructura en tilemap hexagonales o, incluso, con vista isométrica:



Crear tilemap

Para crear el tilemap, en clase seguimos los pasos del siguiente tutorial de Brackeys:

How to make a 2D Game in Unity.
https://www.youtube.com/watch?v=on9nwbZngyw

Nos descargamos de la Asset Store el mismo pack de assets que usa él en el vídeo y montamos nuestro nivel siguiendo sus mismos pasos.

Si queremos profundizar más en la creación de niveles de Unity, en el aula virtual dejamos varios video tutoriales sobre el tema el primer día. A continuación os dejo también un enlace a otro tutorial más, esta vez

oficial de Unity:

Tilemap: The Basics.
https://www.youtube.com/watch?v=fmNtibNWPhc

¡¡Recuerda tener cuidado con las sorting layers de los sprites y los tilemaps!! Tal y como explica Brackeys en su vídeo, deberías tener el mapeado en una capa más trasera que la del personaje, los items y demás objetos que quieras que aparezcan en primer plano.

Crear colliders

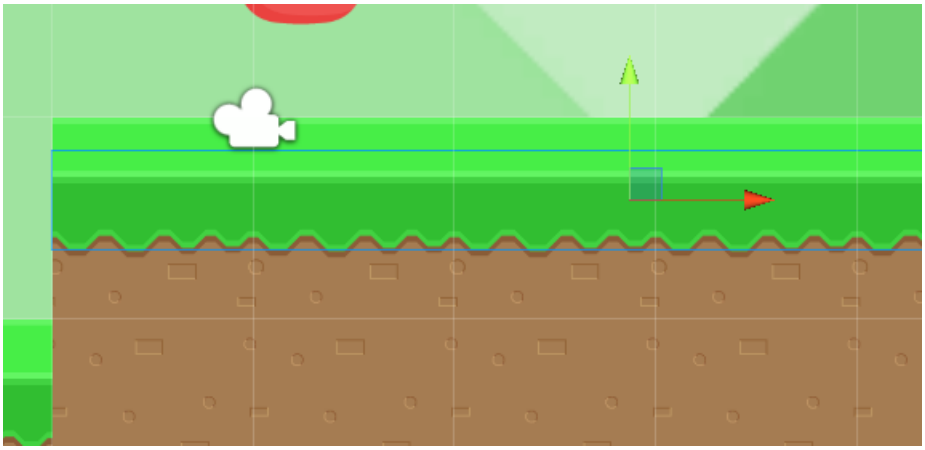
Una vez tengamos el mapa creado, tendremos solo la apariencia. Nos falta rellenarlo con las hitboxes con las que en realidad interactuará nuestro personaje.

¡Importante! Al vestir a nuestro escenario de colliders, debemos poder diferenciar entre los colliders de las paredes y los colliders del suelo, para así, más adelante, que nuestro personaje sea capaz de diferenciar si está tocando suelo o no.

Por comodidad: cambiar color de los colliders

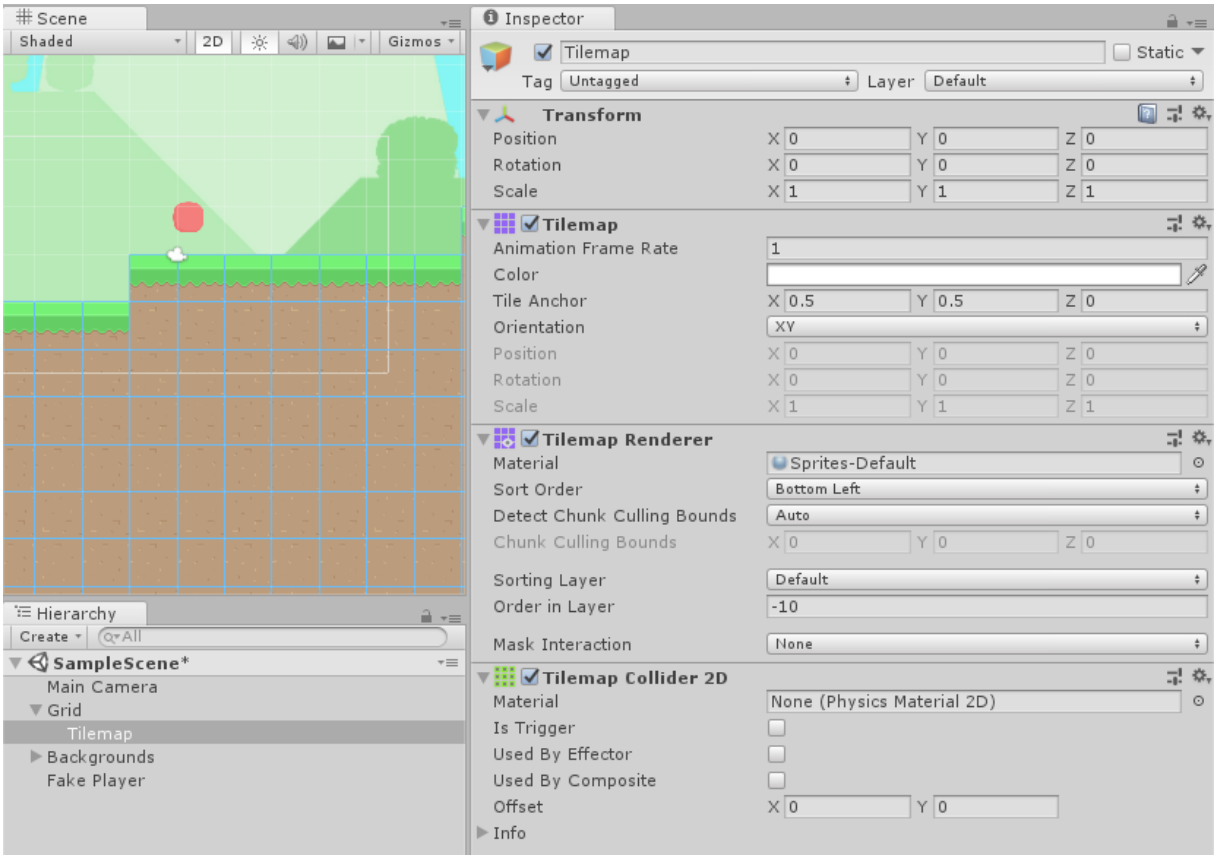
Pero primero, dado que suelo de los sprites suele ser de hierba y el color de los colliders en Unity es también verde, podemos cambiarlo desde **Edit > Project Settings > Physics 2D > Gizmos**.

Como puedes ver en la imagen, yo lo he puesto de color azul, que se distingue mejor contra el verde del escenario.



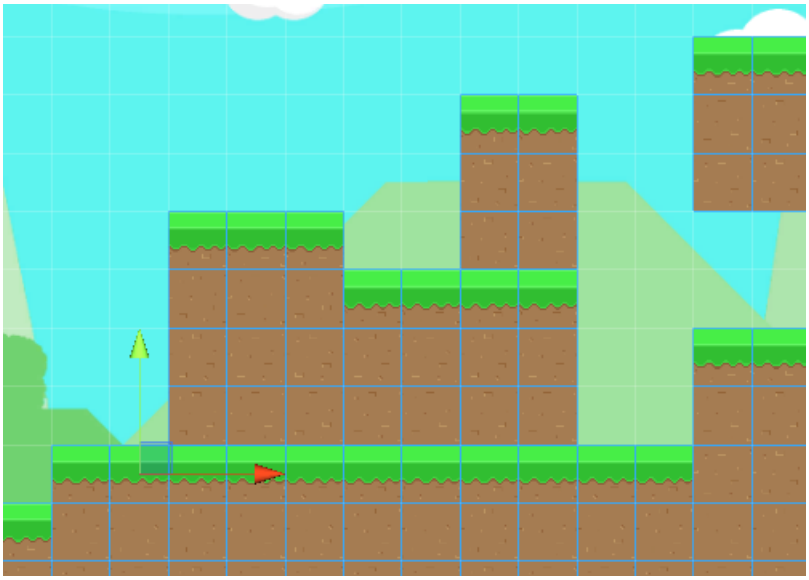
Forma rápida y poco útil

La forma más rápida de ponerle colliders a un tilemap es con el tipo de collider especial **Tilemap Collider 2D**, que, si lo añades al Game Object del Tilemap, generará un collider cuadrado por cada tile que hayas pintado en el tilemap.



Esta forma es poco versátil, ya que nos impide que tengamos partes del escenario en segundo plano.

Por ejemplo, en la siguiente imagen, nuestro personaje chocaría contra la plataforma como si se tratase de una pared.



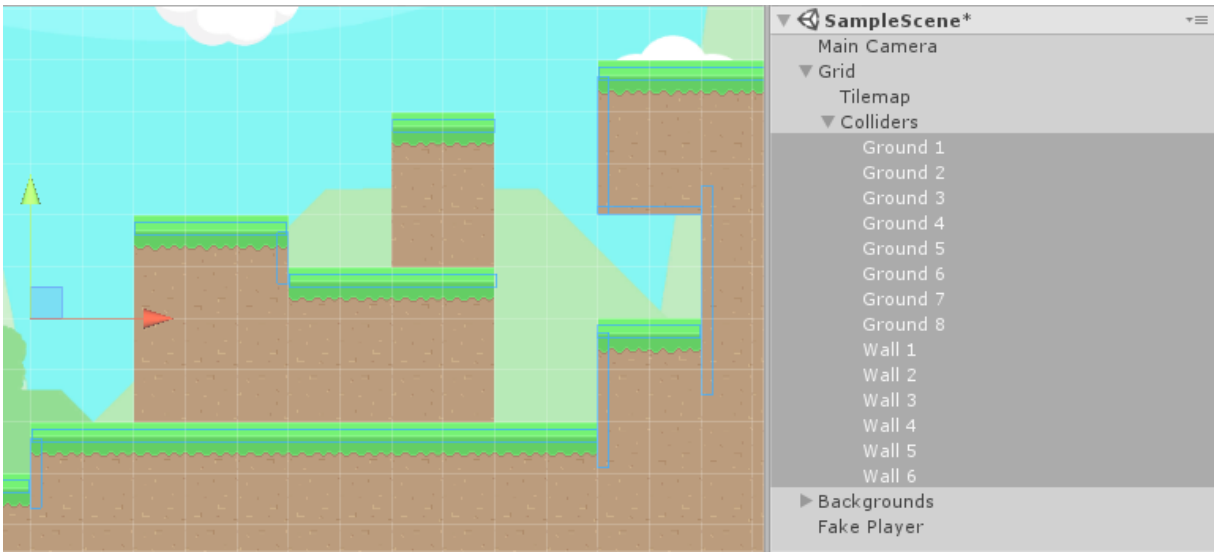
Además, de esta forma no podemos diferenciar entre colliders del suelo y colliders de la pared, lo cual nos resultará útil después para identificar si lo que estamos tocando es suelo o pared.

Forma adecuada

Por tanto, surge la necesidad de poder posicionar nuestros colliders independientemente del aspecto del escenario, como si se tratasen de paredes invisibles que, en su posición correcta, simularán el suelo que la imagen del escenario nos muestra.

BoxCollider2D

Una opción pasaría por vestir al escenario de Box Colliders 2D, como en la imagen:

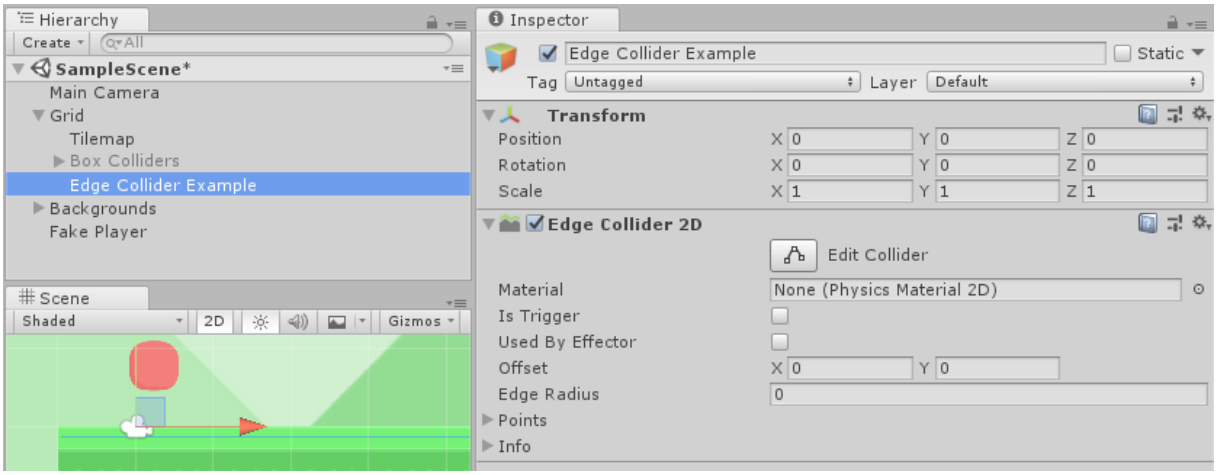


Fíjate cómo he metido todos los colliders dentro de un Game Object vacío llamado “Colliders”, para tenerlos más ordenados.

Además, también he guardado este Game Object como hijo del Grid, para que así, si movemos el escenario, los colliders se muevan con él adecuadamente.

Edge Collider

Otra opción alternativa para vestir al escenario pasaría por usar colliders de tipo Edge Colliders 2D, que son simplemente una línea, tal y como se ve en la siguiente imagen:

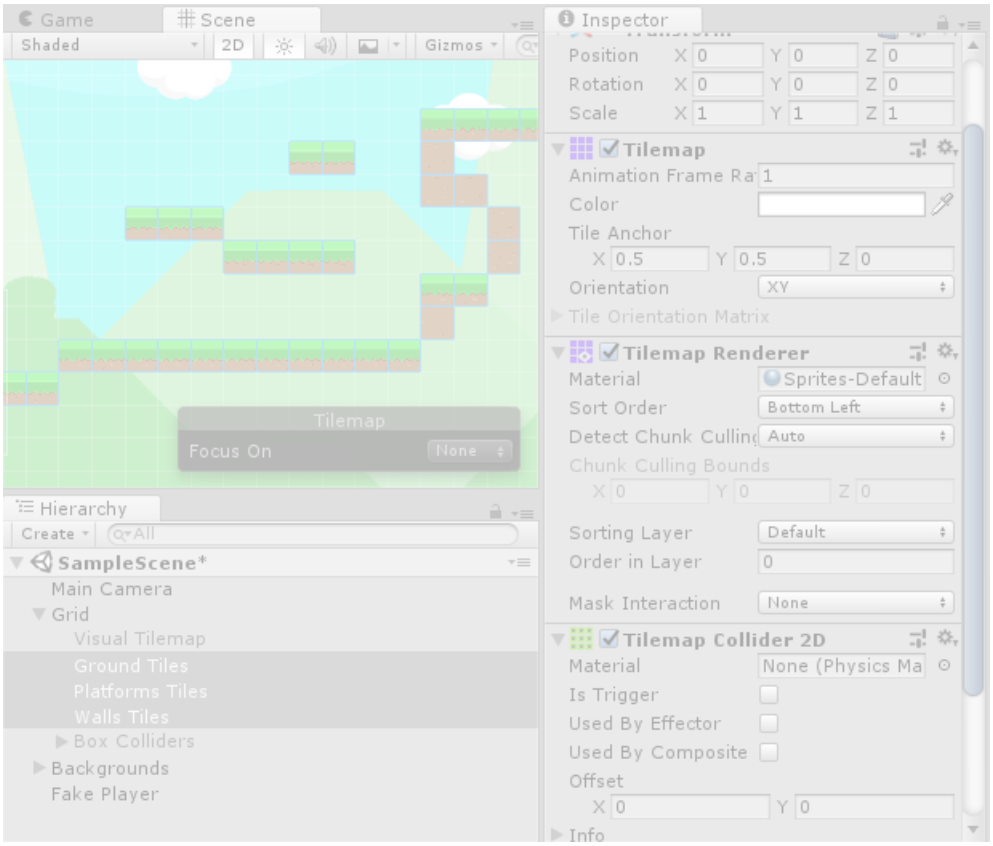


Con ellos se puede conseguir un mayor nivel de detalle, pero los Box Colliders 2D son más fáciles de manipular y visualizar, y para el escenario que estamos construyendo no necesitamos tanto nivel de detalle.

Usando Tilemap Collider 2D, pero con diferentes tilemaps

Otra opción, que es mi preferida para automatizar el proceso, consiste en crear diferentes tilemaps para los diferentes tipos de colisiones (suelos, paredes, plataformas), y entonces poder asignar automáticamente el Tilemap Collider 2D y dejar que Unity se encargue del trabajo duro.

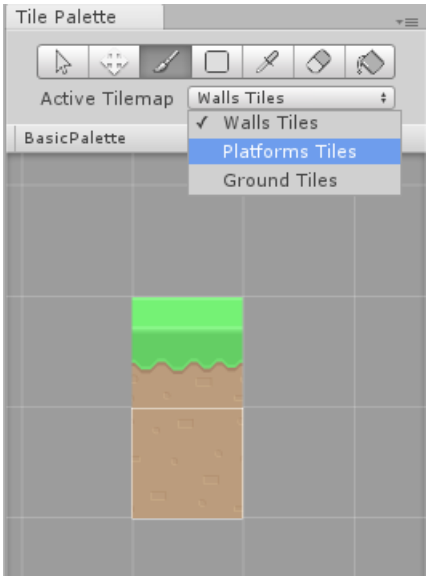
De esta forma, pintaríamos, por un lado, el escenario decorativo, que el jugador nunca llegaría a tocar (espacios subterráneos y paredes de plataformas en segundo plano que queremos que nuestro personaje pueda sobrepasar). Y por otro, en tilemaps diferentes, los tiles con los que sí que interactuará nuestro personaje. Tal que así:



(Tengo desactivado el “Visual Tilemap”. Si lo activara, el escenario se vería igual que siempre.

De esta forma nos ahorramos el trabajo sucio, lo cual es muy importante a la hora de hacer un juego largo porque facilita el avanzar, lo cual ayuda a que la creatividad no se apague entre dificultades técnicas.

Si decides hacerlo así, ten cuidado de seleccionar el tilemap donde quieres pintar en la ventana del Tile Palette. Ya me ha pasado varias veces de estar pintando paredes y suelos en el tilemap equivocado.



Los colliders especiales de las plataformas unidireccionales y móviles las crearemos más adelante, cuando ya tengamos funcionando las físicas del personaje.

Físicas del personaje

Una vez terminado el mapa, ha llegado el momento de montar y programar a nuestro personaje. Para ello, utilizaremos lo aprendido en el Flappy Bird y en el Monkey Coins.

Vamos a crear un script **CharacterController2D**, donde programaremos las físicas de nuestro personaje, tales como:

- Que se mueva si pulsamos WASD
- Que salte si pulsamos espacio
- Que sólo salte si estamos tocando el suelo
- Que cambie la orientación del sprite según la dirección en la que estemos avanzando.
- etc.

En clase, decidimos llamar al script **SimpleCharacterController2D**. Esto es debido a que la red está llena de scripts que ya gestionan el movimiento del personaje con mucho detalle, y lo más común es que nos descarguemos uno de la Asset Store y lo modifiquemos. De momento, vamos a aprender a hacerlo por nuestra cuenta, pero hemos elegido un nombre diferente por si en un futuro queremos utilizar uno más complejo bajado de internet (así, los nombres no se pisarán).

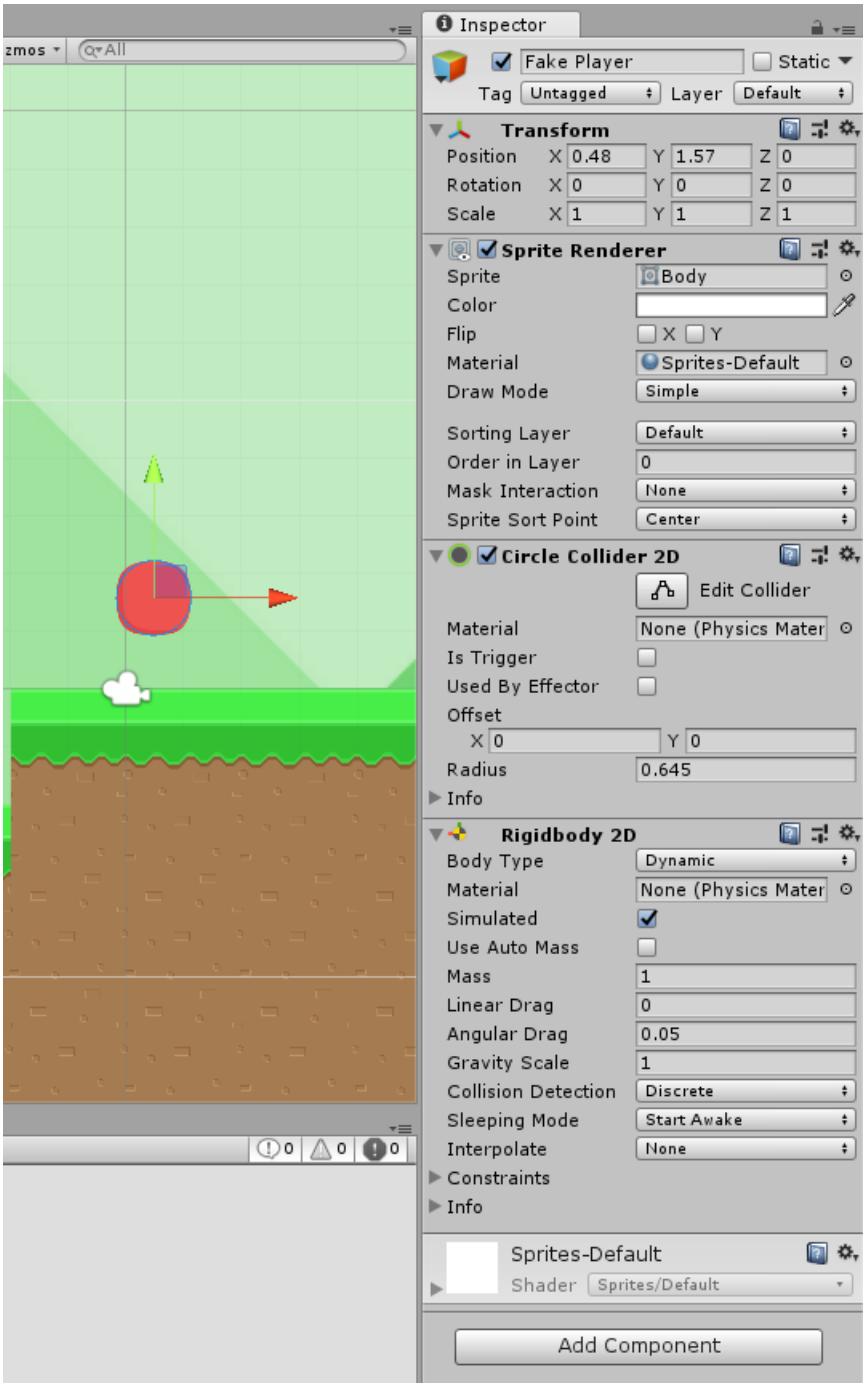
En el siguiente vídeo, Brackeys explica cómo programar el movimiento del personaje utilizando un CharacterController2D.cs programado por él, que nos podemos descargar e incluir en nuestro proyecto para probarlo:

2D Movement in Unity (Tutorial).
https://www.youtube.com/watch?v=dwcT-Dch0bA

De momento, en las próximas secciones veremos como programar el nuestro propio. Una vez sepamos cómo se hace, entonces podremos elegir entre nuestro código, o el de Brackeys.

0. Crear sprite + rigidbody + collider

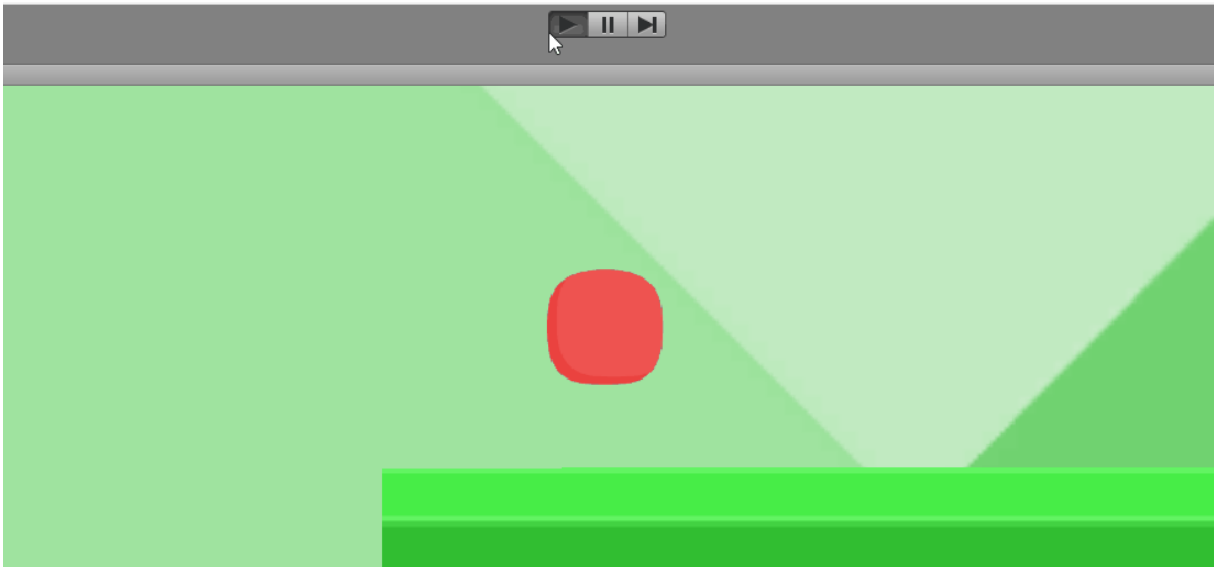
Lo primerísimo que tenemos que hacer es arrastrarnos una imagen a la escena, para que nos cree un **Sprite Renderer** y así poder ver a nuestro personaje. Después, vamos a añadirle un **Circle Collider 2D** y un **Rigidbody 2D** para que responda a físicas, gravedad y colisiones.



¿Por qué Circle Collider en lugar de Box Sollider?

Para programar las físicas del personaje, en los pies se usan siempre formas de colliders redondeadas para evitar que se atasque o que le afecten bruscamente las irregularidades del terreno.

Ya tenemos lo equivalente a una pelota que cae y choca contra el suelo:



1. Hacer que salte

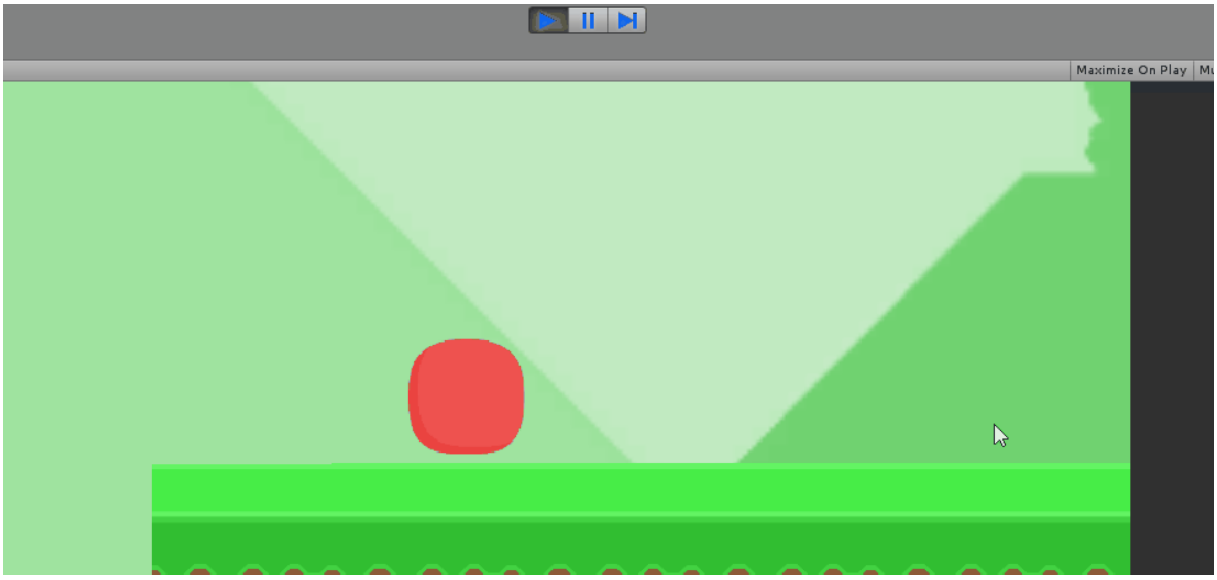
Primero, le agregamos el componente script **Character Controller 2D** a nuestro personaje-pelota.

Vamos a decirle que, en todo momento, esté comprobando **si pulsamos el botón de saltar**, y si lo pulsamos, entonces modificar nuestra velocidad hacia arriba, al igual que hacíamos en el Flappy Bird.

Para ello, **recuerda que necesitarás una referencia al rigidbody, para poder hablarle desde el código**.

```
// En todo momento
void  Update()
{
 // Comprobar si se ha pulsado espacio
 if (Input.GetKeyDown(KeyCode.Space))
 {
  // Si se ha pulsado, modificar velocidad hacia arriba con una fuerza de 5
  myRigidbody.velocity  =  Vector2.up  *  5.0f;
 }
}
```

Multiplicamos por 5 para que el salto sea más fuerte. Si no lo multiplicamos por nada, se quedaría como multiplicado *1 y saltaría poquísimo.



Ya tenemos al personaje saltando al pulsar espacio.

> No copies el código a ciegas. ¿Has inicializado tu variable de tipo Rigidbody2D en el Start, al igual que hacíamos en el Flappy Bird? Si no lo has hecho, a qué esperas. Tienes dos opciones:
>
> - Usa `GetComponent<Rigidbody2D>()` en el método `Start()`, como hacíamos en el Flappy Bird.
> - O haz la variable pública y arrástrale la referencia desde el inspector.

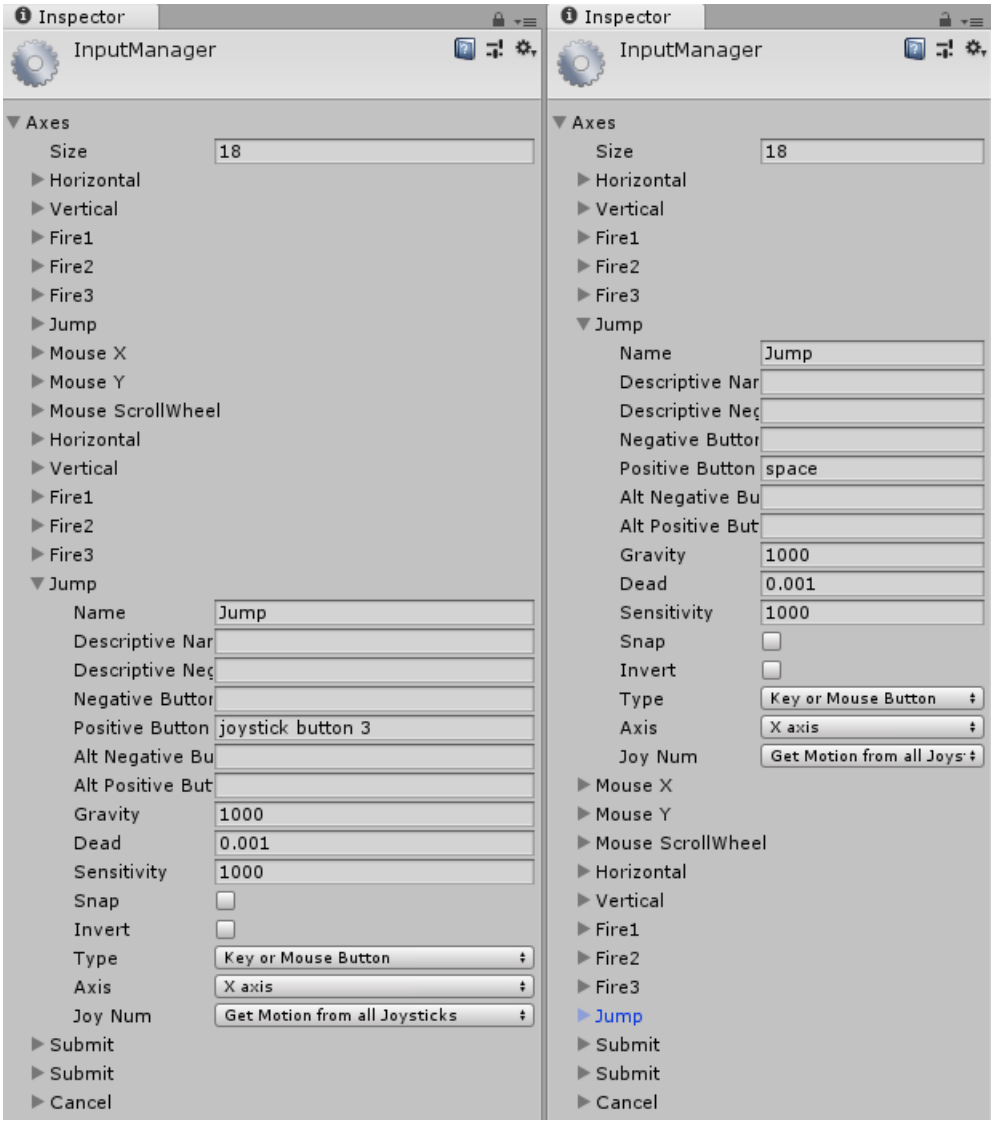
1.2. GetButtonDown("Jump") VS GetKeyDown(KeyCode.Space)

Vamos a introducir una mejora. En lugar de usar el método **GetKeyDown(KeyCode.Space)**, utilizaremos **GetButtonDown("Jump")**, que **también funciona con gamepads**. ¿Por qué?

Al utilizar GetKeyDown, nos estamos comunicando directamente con el teclado del ordenador, y le estamos preguntando si se ha pulsado la tecla espacio.

Al utilizar **GetButtonDown, nos estamos comunicando con el sistema de Input Manager de Unity**, que nos permite mapear acciones a más de un botón. En este caso, la acción "Jump" está mapeada tanto a la tecla espacio del teclado, como al botón A del mando de Xbox, o al botón X del mando de PS4.

Podemos comprobarlo accediento al Input Manager desde **Edit > Project Settings > Input > Axes > Jump**. En ese menú, podemos encontrar mapeada la acción "Jump" a dos botones diferentes, espacio por un lado, y joystick button 3 por otro.



(Me he abierto dos ventanas de inspector para mostrártelo, pero tú podrás comprobarlo con una sola).

Por tanto, nuestro método Update nos quedará como:

```
// En todo momento
void  Update()
{
 // Comprobar si se ha pulsado espacio
 if (Input.GetButtonDown("Jump"))
 {
  // Si se ha pulsado, modificar velocidad hacia arriba
  myRigidbody.velocity  =  Vector2.up  *  5.0f;
 }
}
```

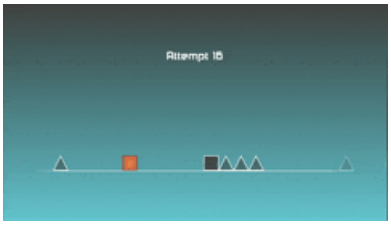
Prueba a enchufarle un mando de Xbox a Unity y ya verás como funciona.

1.3. ¿Por qué GetButtonDown y no GetButtonUp, o GetButton a secas?

Si usamos el sufijo “Up”, el programa reaccionará cuando levantemos el dedo del botón, por lo que la acción se retrasará unos instantes aunque no nos demos cuenta, lo cual puede traducirse en frustración en el jugador, al caerse por el precipicio a pesar de haber pulsado el botón.

Si no usamos ningún sufijo, el programa reaccionará constantemente mientras tengamos pulsado el botón, por lo que el personaje estará saltando continuamente, y volverá a saltar tan pronto como vuelva a tocar el suelo (siempre que hayamos programado ya que sólo salte de nuevo al volver a tocar el suelo). Esto tampoco lo queremos, ya que suele ser incómodo, y no suele ser realista ver al personaje saltando todo el rato.

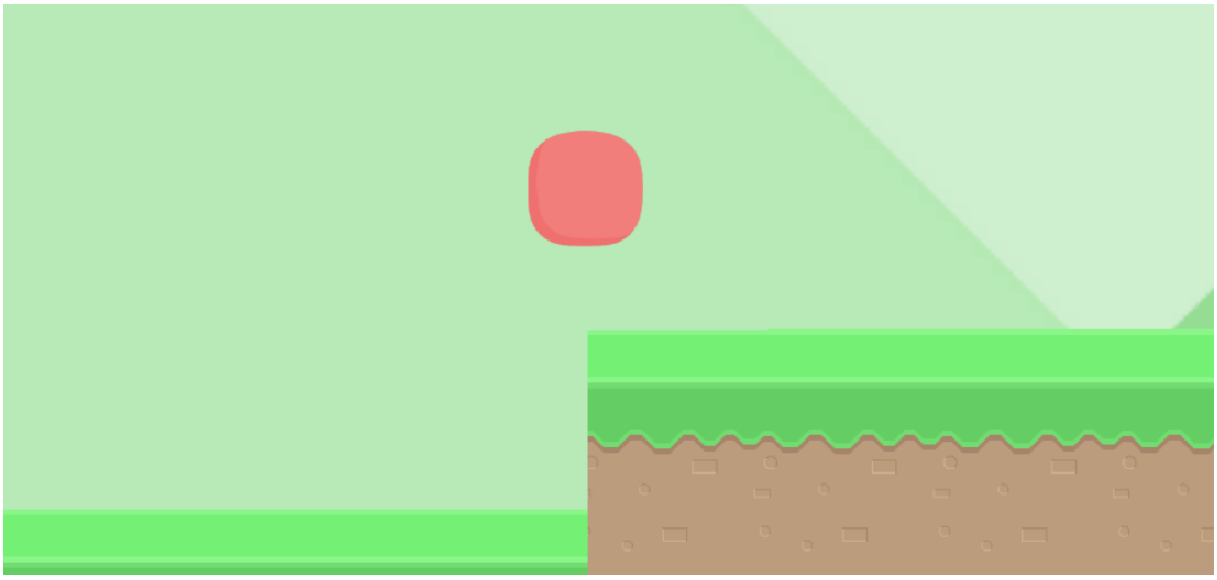
Hay otros juegos, como Geometry Dash, en que el jugador sí que salta constantemente si mantiene pulsado el botón, pero eso tiene sentido en el Geometry Dash por como está estructurado su gameplay, no en un plataformas estilo Super Mario.



En Geometry Dash, sí que utilizaríamos GetButton(“Jump”) a secas, ya que querríamos que el cuadrado siguiese saltando mientras no soltemos el botón.

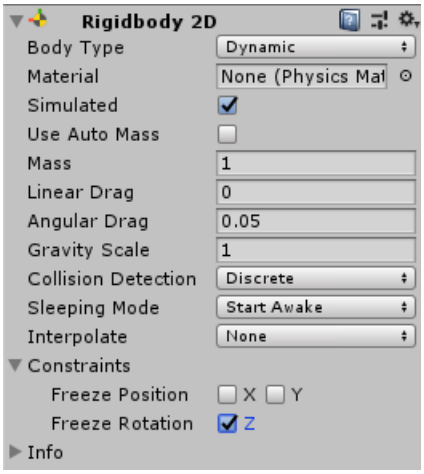
2. Evitar que ruede

Si ahora mismo soltamos al jugador sobre una pendiente, se torcerá y reaccionará como una pelota.



Y no queremos eso, a no ser que nuestro personaje sea una pelota.

Para evitarlo, basta con **desplegar el apartado Constraints del Rigidbody 2D, y congelar la rotación alrededor del eje Z.**



De esta forma, el Game Objet ya no reaccionará a las físicas que lo hagan rotar, y siempre mirará hacia arriba.

3. Movimiento horizontal

Vamos a hacer que se mueva hacia izquierda y derecha, tal y como hicimos en el proyecto del **Monkey Coins**, utilizando el método **Input.GetAxis("Horizontal")** que nos devuelve -1 si pulsamos hacia la izquierda y +1 si pulsamos hacia la derecha (y 0 si no pulsamos nada).

- Si pulsamos izquierda, **Input.GetAxis("Horizontal") = -1**
- Si pulsamos derecha, **Input.GetAxis("Horizontal") = +1**
- Si no pulsamos nada, **Input.GetAxis("Horizontal") = 0**

Además, **El método GetAxis también responde ante el Input Manager de Unity**, por lo que **funciona tanto con teclado como con mando**. Con teclado responde tanto a WASD como a las flechas de dirección, y con mando está configurado para responder al Joystick Izquierdo.

Para lograr el movimiento, vamos a programar que lo que le metamos por el Input.GetAxis("Horizontal") sea la velocidad en X del Rigidbody.

```
void Update()
    {
        Vector2 playerInput;

  // Recogemos el input del jugador
        playerInput.x = Input.GetAxis("Horizontal") * 10f;

  // Recordamos la velocidad vertical del Rigidbody
        playerInput.y = myRigidbody.velocity.y;

  // Si pulsamos saltar
        if (Input.GetButtonDown("Jump"))
        {
         // Nos da igual la velocidad vertical anterior, ahora queremos que vaya para arriba
            playerInput.y = 5f;
        }

  // Asignamos los inputs recogidos al rigidbody
        myRigidbody.velocity = playerInput;
    }
```

Multiplicamos el Input por 10 para que se mueva más rápido. Si no lo multiplicamos por nada, se quedaría como multiplicado *1 y sería muy lento.

Posible BUG: mi personaje cae muy lento
Un bug muy típico sería haber programado:

```
playerInput.y = 0f;
```

Como queriendo no modificar (por eso el 0) la velocidad vertical del rigidbody. ¡¡Craso error!! Claro que la estamos modificando, por 0. Por tanto, en cada frame, estamos combatiendo la fuerza de la gravedad, obligando al cuerpo a pararse en el aire.

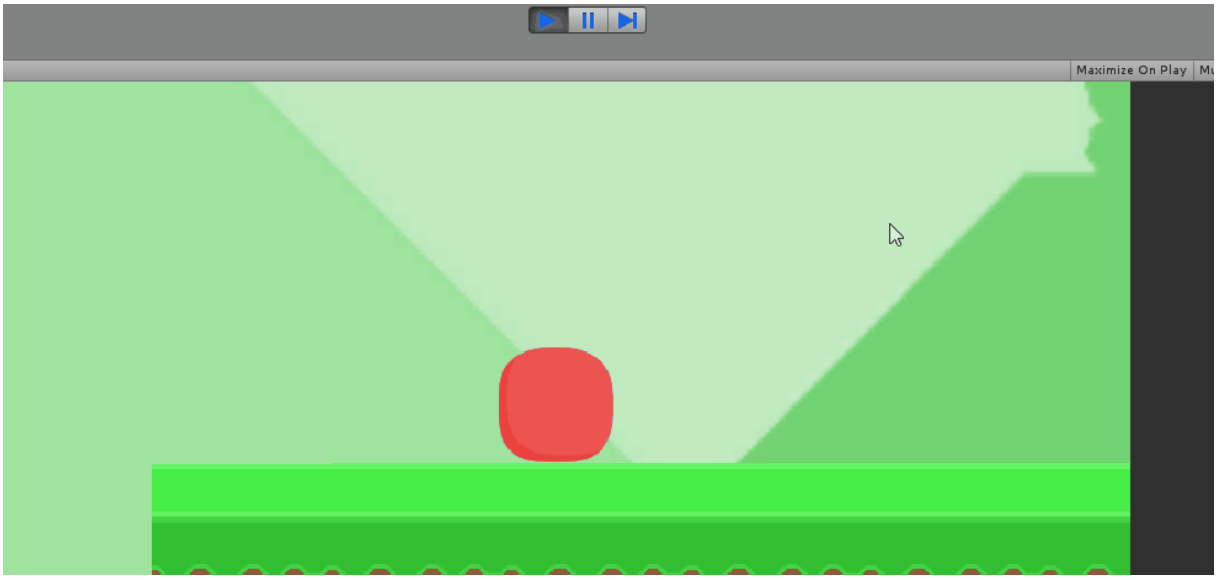
Por ello, programamos

```
playerInput.y = myRigidbody.velocity.y;
```

para que así, en cada frame, mientras no pulsemos saltar, la velocidad va a seguir siendo la misma de antes.

3.1. Hacer movimiento instantáneo (quitar suavizado)

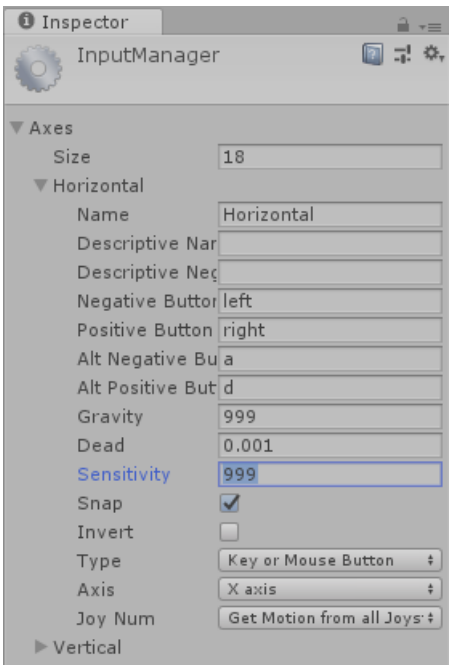
Como no tenemos con qué comparar, no nos daremos cuenta, pero **El Input Manager está manipulando los valores de las teclas WASD para simular que estamos usando un joystick de mando**. ¿Qué pasa con esto? Que nuestro personaje tarda más en acelerar y más en frenar, como suavizando el movimiento.



Esto es un efecto que mola si somos nosotros, con un mando, los que lo controlamos. Pero tenerlo predefinido en las teclas es incómodo, y un signo de que somos novatos usando Unity y no sabemos modificar el Input Manager. Cambiarlo es muy fácil, basta con acceder a los ejes Verticales y Horizontales del Input Manager y quitarle el suavizado para que el cambio sea instantáneo.

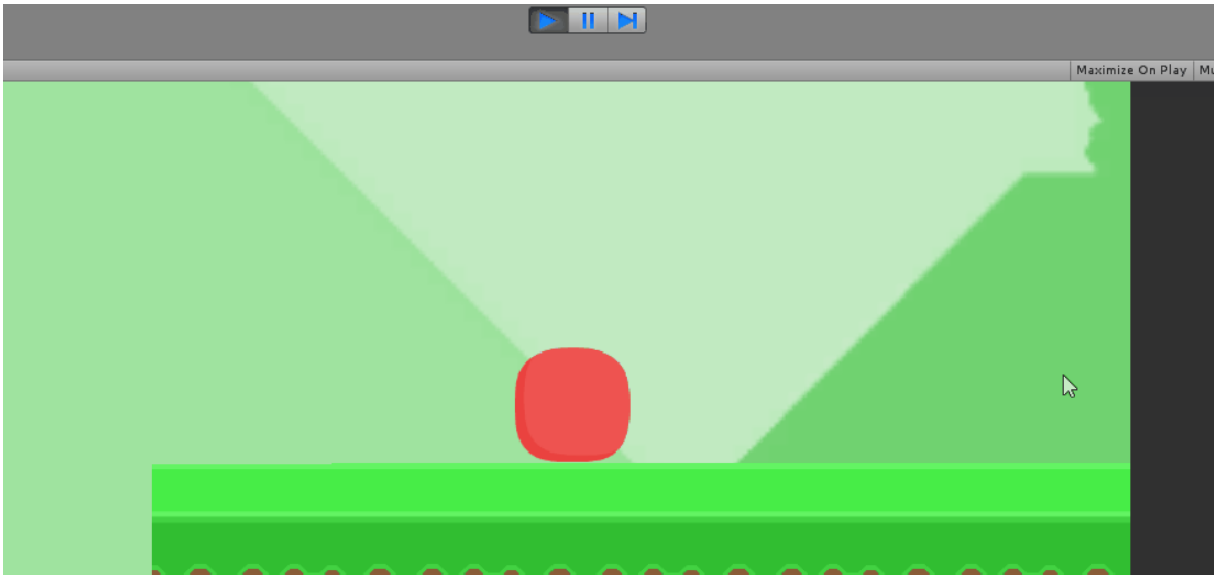
Recuerda: Edit > Project Settings > Input > Axis > Horizontal / Vertical

Modifica los valores “Gravity” y “Sensitivity” por valores muy muy altos, como 999, para que los botones del teclado dejen de creerse joysticks.



Aunque de momento solo estemos usando el eje horizontal, cámbiaselo también al vertical.

Fíjate en cómo se siente el nuevo movimiento:

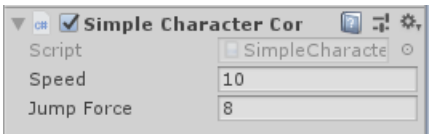


¿Notas la diferencia? Si no la notas en el gif, créeme que el jugador sí que la nota. Y si no, sé betatester de tu propio juego, pruébalo tú mismo y decide.

4. Hacer variables públicas

Hasta ahora, la velocidad de movimiento y la fuerza del salto las hemos dejado *hardcodeadas*, escritas a la fuerza en el código como 10f y 5f respectivamente. Sería interesante permitir que el diseñador las modificase desde el inspector haciendo esos valores públicos:

```
public float speed = 10f;
public float jumpForce = 8f;
```



Para que el diseñador lo tenga más fácil, y no nos meta valores negativos ni extremadamente altos, podemos obligarlo a ceñirse a un rango de valores con el atributo `[Range(min,max)]`.

```
[Range(1f, 20f)] public float speed = 10f;
[Range(1f, 20f)] public float jumpForce = 8f;
```

> En programación orientada a objetos, atributo tiene varios significados. El más típico es que un atributo es una variable pública cualquiera de nuestro Script, a la que podemos *atribuirle* un valor desde el inspector. Pero también se le llama atributo a códigos especiales encerrados entre llaves [ ] que se

> ponen junto a las variables públicas para personalizar cómo se muestran en el inspector. A continuación te dejo algunos útiles, pruébalos para ver cómo funcionan:
>
> - [Tooltip("Lorem ipsum sit amet")]
> - [Header("Lorem ipsum sit amet")]
> - [Space(20)]
> - [Range(min, max)]
>
> Cambia los parámetros entre paréntesis para ver los resultados

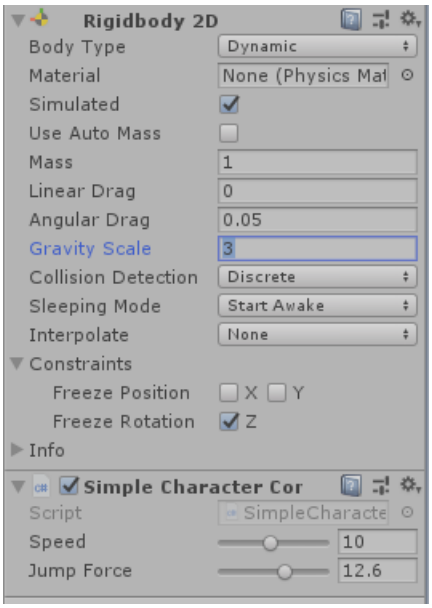
5. Hacer que caiga más rápido

En videojuegos de plataformas, las leyes de la física no tienen por qué ser las del mundo real. De hecho, no lo son nunca, porque las físicas del mundo real son aburridas, y porque sería imposible saltar 3 veces nuestra altura, como hace Super Mario.

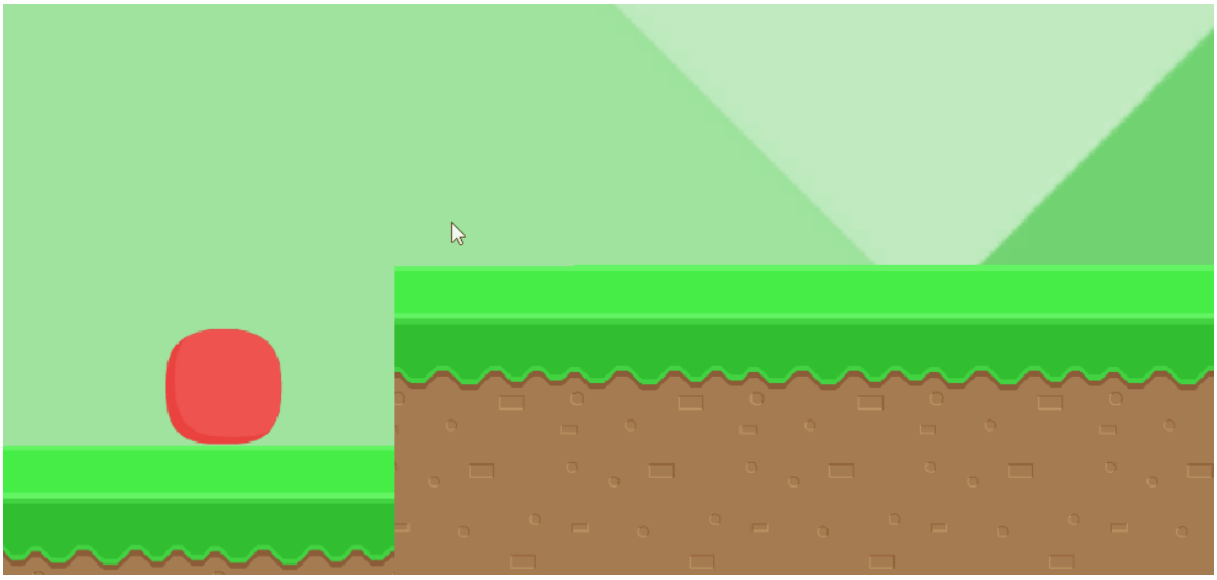
Ahora que tenemos al personaje moviéndose, como Game Designers deberíamos darnos cuenta de que cae demasiado lento.

¿Demasiado lento? En realidad no. Está cayendo a la velocidad física real de la simulación a la que caería un objeto con la gravedad terrestre. Pero eso, en videojuegos, es aburrido.

Para arreglarlo, basta con hacer algo más grande el valor de la gravedad del componente Rigidbody2D.



Juega con el multiplicador de la gravedad del rigidbody y con la fuerza del salto que hiciste pública en el inspector hasta conseguir unos valores que se sientan divertidos.

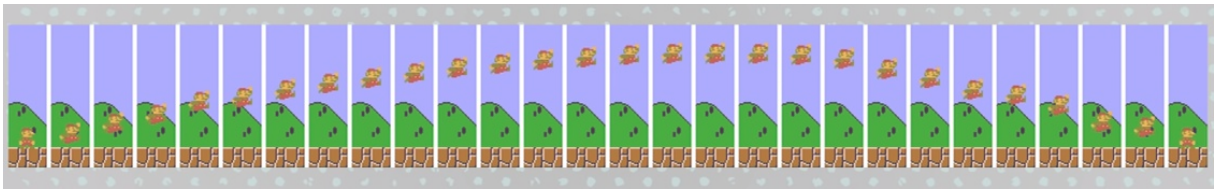


¿Notas la diferencia?

5.1. Hacer que manteniendo pulsado llegues más alto

En realidad, lo único que diferencia las físicas de videojuegos de las físicas reales es mucho más que una gravedad más fuerte. **¿No has notado nunca como, en los juegos de plataformas, a más tiempo dejes pulsado el botón saltar, más altura alcanzas en el salto? Y, si lo pulsas y despulsas rápido, tu salto es más pequeño.**

Si no me equivoco, ese tipo de salto lo inventó Super Mario, y fue lo que lo hizo triunfar y sentirse tan divertido en aquella época, cuando aún en los juegos de la Spectrum ni siquiera se podía corregir la trayectoria del salto una vez lo ejecutabas (como en la vida real, ¿no?).



Para quien quiera hacer su juego más divertido, dejo aquí un tutorial que explica como conseguirlo con 4 sencillas líneas de código:

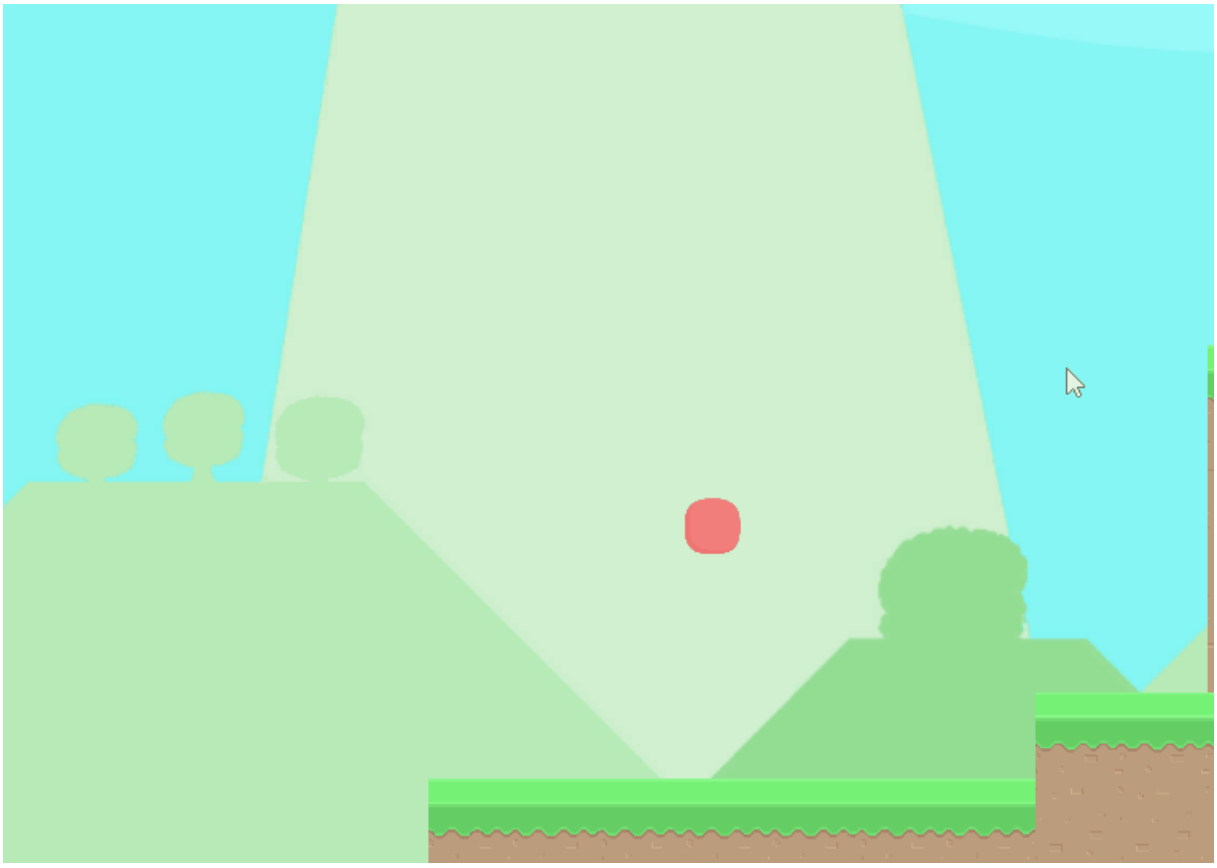
Better Jumping in Unity With Four Lines of Code
https://www.youtube.com/watch?v=7KiK0Aqtmzc

6. Detectar suelo

BUG: salta infinito // FIX: Permitir saltar sólo cuando esté en el suelo

Ahora mismo, nuestro personaje salta infinitamente, al igual que hacía el Flappy Bird. Vamos, que

técnicamente vuela.



Para evitar que salte en el aire, basta con comprobar si estamos tocando el suelo. Y para saber si estamos tocando el suelo, nos hace falta una forma de detectarlo. Los ingredientes son:

- Una variable booleana llamada **grounded**, que esté en **false si el personaje está en el aire, y en true si está en el suelo**
- Sólo **saltará si pulsa espacio Y (&&) está en el suelo**
- **Al entrar en colisión con el suelo (OnCollisionEnter), grounded pasa a ser true**.
- **Al dejar de colisionar con el suelo (OnCollisionExit), grounded pasa a ser false**.

Por tanto, comienza **declarándote una variable privada grounded, que empiece en false** (ya que el personaje spawnea en el aire, normalmente).

```
public class SimpleCharacterController2D : MonoBehaviour
{
 [...]
    bool grounded = false;
    [...]
```

Y antes de permitir que salte, comprueba que grounded sea true.

```
// Si pulsamos espacio Y el personaje está en el suelo
if (Input.GetButtonDown("Jump") && grounded == true)
{
 // Saltar
    playerInput.y = jumpForce;
}
```

Detectar suelo con OnCollisionEnter2D y OnCollisionExit2D

Utilizando las funciones que detectan colisiones, podemos darnos cuenta de si hemos tocado el suelo o no.

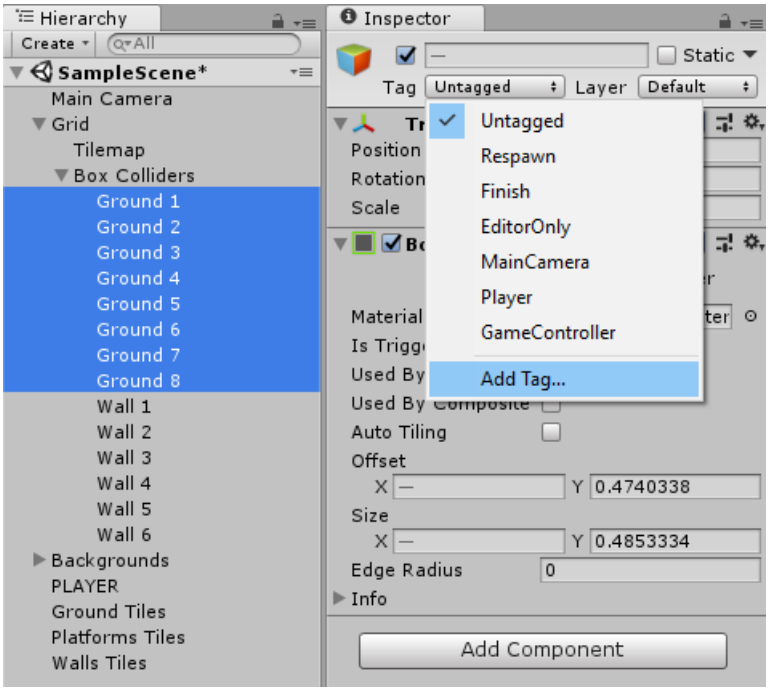
```
void OnCollisionEnter2D(Collision2D collision)
{
    grounded = true;
}

void OnCollisionExit2D(Collision2D collision)
{
    grounded = false;
}
```

¿Qué pasa ahora? Que, **choque con lo que choque, va a detectar que está en el suelo**. Sea o no realmente un suelo (podría ser la pared, el techo, o incluso un enemigo o moneda). Vamos a corregirlo.

BUG: Detecta las paredes como suelo // FIX: Poner tag “Suelo” a los colliders del suelo

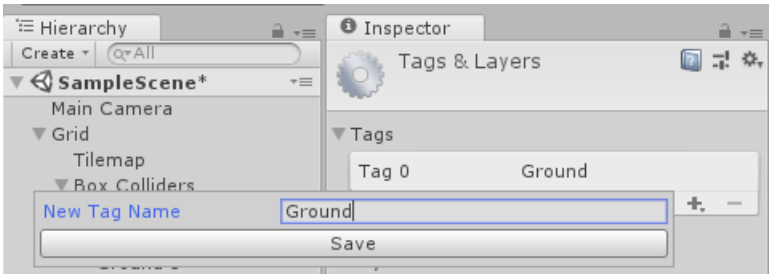
Por esta razón es importante dividir los colliders entre suelos y paredes.

Debemos crear un nuevo tag, una etiqueta, para identificar los colliders del suelo:

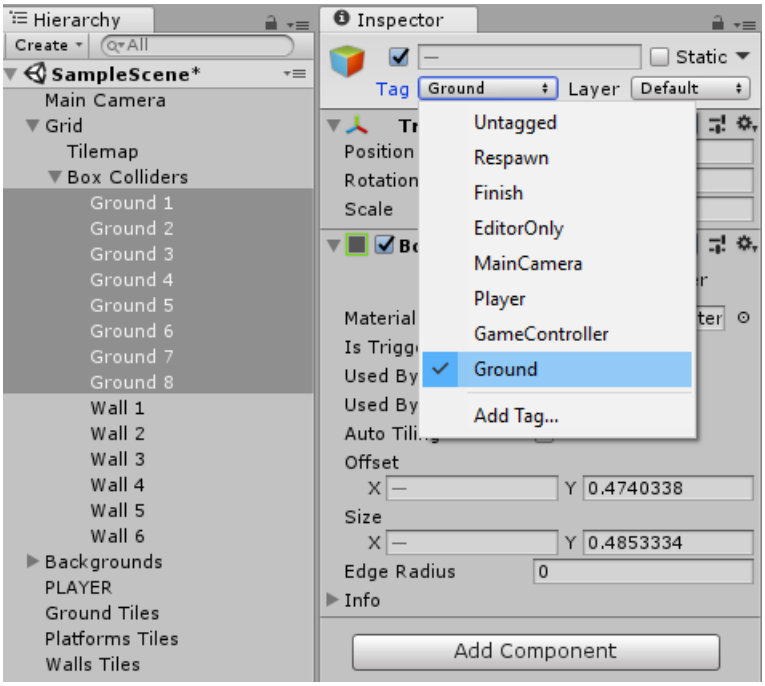


(Al principio no podemos añadírsela porque no existe, vamos a crearla. Pulsamos en **Add Tag…**)

En la ventana que se nos abre, "Tags & Layers" **pulsamos sobre el símbolo + y creamos el tag "Ground"**.



Gracias a esto, ya sí que podremos asignarle la tag "Ground" a los colliders del suelo:



De esta forma, en código podremos identificar si el objeto con el que estamos chocando es suelo o no y, solamente si es suelo, ponemos grounded a true:

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

// En collision exit no hace falta hacer la comprobación
void OnCollisionExit2D(Collision2D collision)
{
    grounded = false;
}
```

BUG: Al resbalar por la pared y tocar el suelo, no salta // FIX: Usar OnCollisionStay2D

Todavía hay ciertos bugs que nos pueden ocurrir y que pueden bloquear a nuestro personaje e impedir que salte. Por ejemplo, si estamos tocando la pared y el suelo al mismo tiempo, en una esquina, es posible que OnCollisionEnter sea ejecutado a la vez por la pared y por el suelo, detecte la pared, no se ponga grounded en true y por tanto siga siendo false aún estando en el suelo, e impidiéndonos saltar.

Para evitar este tipo de bugs, **debemos comprobar continuamente, en cada frame, si estamos tocando el suelo o no**. No nos basta con comprobarlo solo cuando chocamos con algo. **Esto podemos hacerlo mediante el método OnCollisionStay.**

Por tanto, nos basta con cambiar nuestro código para usar OnCollisionStay en lugar de OnCollisionEnter:

```
void OnCollisionStay2D(Collision2D collision)
{
 if (collision.collider.tag == "Ground")
 {
     grounded = true;
 }
}

// En collision exit no hace falta hacer la comprobación
void OnCollisionExit2D(Collision2D collision)
{
    grounded = false;
}
```

Por qué OnCollisionStay

En la práctica es exactamente lo mismo en casi todos los casos. Simplemente, OnCollisionStay se está ejecutando continuamente, todo el rato, en cada frame, como el método Update, mientras que OnCollisionEnter sólo se llama una vez, al chocar con algo. Si esa única vez perdemos la detección del suelo, grounded queda en false por siempre y ya no podemos volver a saltar. Utilizando OnCollisionStay forzamos al jugador a estar comprobando continuamente si está en contacto o no con el collider del suelo, y tenemos la seguridad de que en cualquier momento, si está en contacto, grounded será true.

7. Evitar que se atasque en las paredes

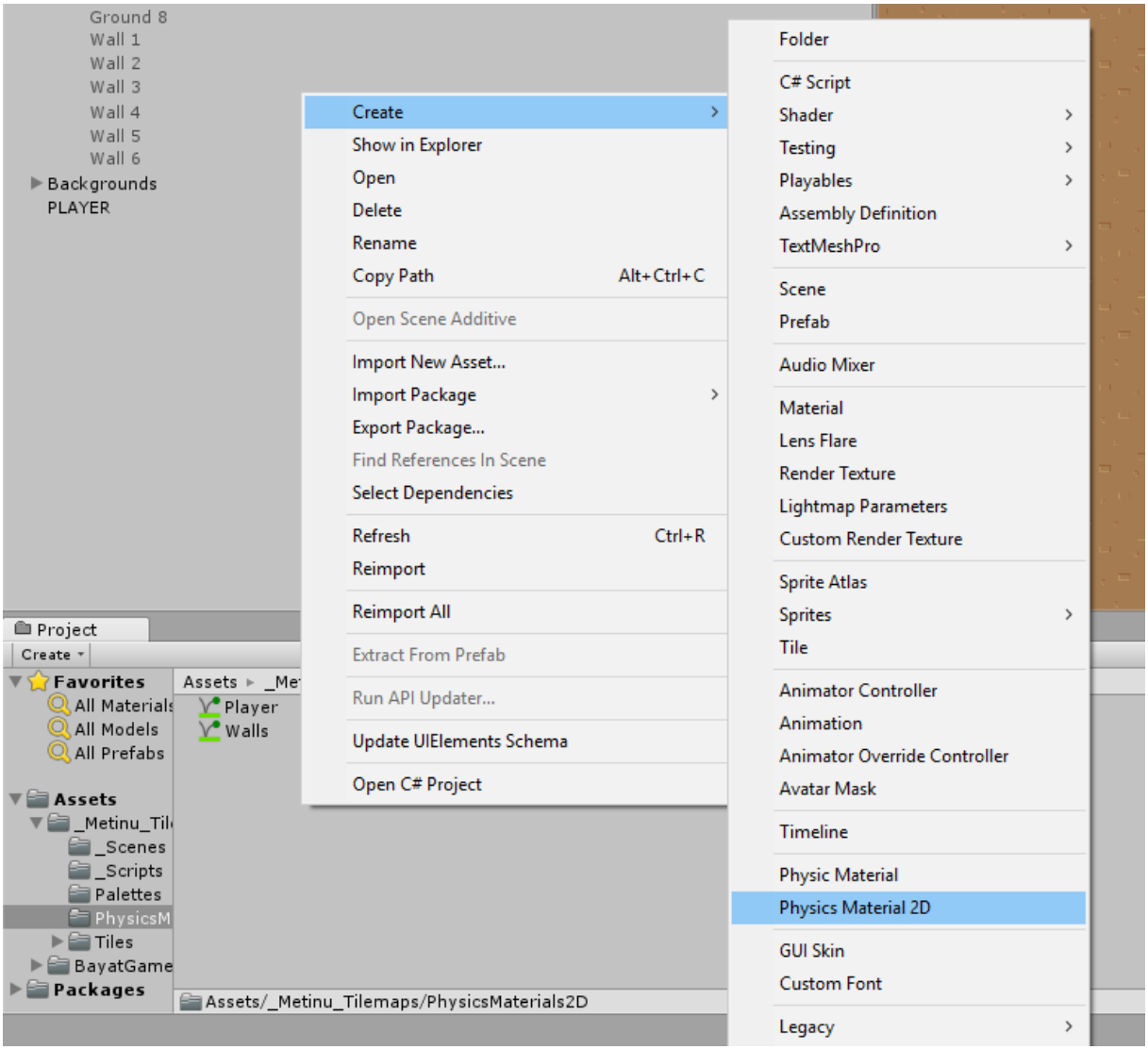
BUG: Se atasca en las paredes // FIX: Materiales físicos deslizantes

Ahora mismo nuestro personaje ya está bastante fino. ¿Con cuántas líneas de código? 44, cuento yo, algo bastante asequible.

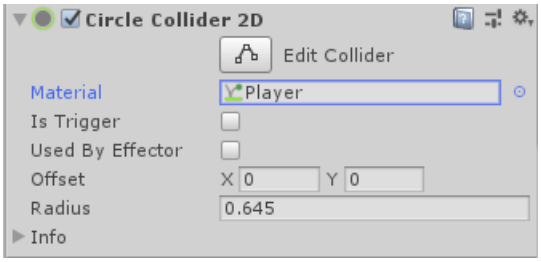
El bug con el que nos encontramos ahora, **otro bug muy típico**, es que nuestro personaje se queda atascado en las paredes al saltar y empujar contra ellas. Esto sucede porque, como estamos permitiendo controlar a nuestro personaje en el aire, **como si tuviera un jetpack**, al pulsar hacia la pared estamos aplastando al personaje contra la pared, lo que impide que éste caiga.

Para solucionarlo, basta con dotar de físicas de fricción a los materiales de los colliders, como si los dotásemos de una rugosidad virtual. Para evitar que nuestro personaje se quede pegado a las paredes, **vamos a hacerlas resbaladizas, como si fueran hielo**. Esta es **otra razón por la que es útil separar los colliders del suelo y de las paredes**.

Para crear un material físico, pulsa botón derecho en la ventana del proyecto > Create > Physics Material 2D

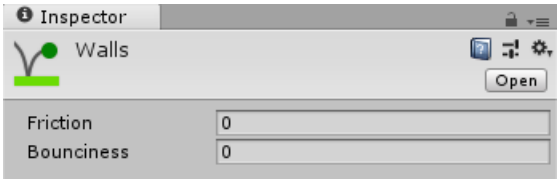


Los **materiales físicos** son diferentes de los materiales que creamos en los programas de edición 3D. Aquí, **olvídate de albedo, metallic, emission, normal map, etc**. Esos otros son los materiales visuales, y se aplican al 3D y al renderer. **Los materiales físicos, en cambio, se aplican sobre los colliders**.

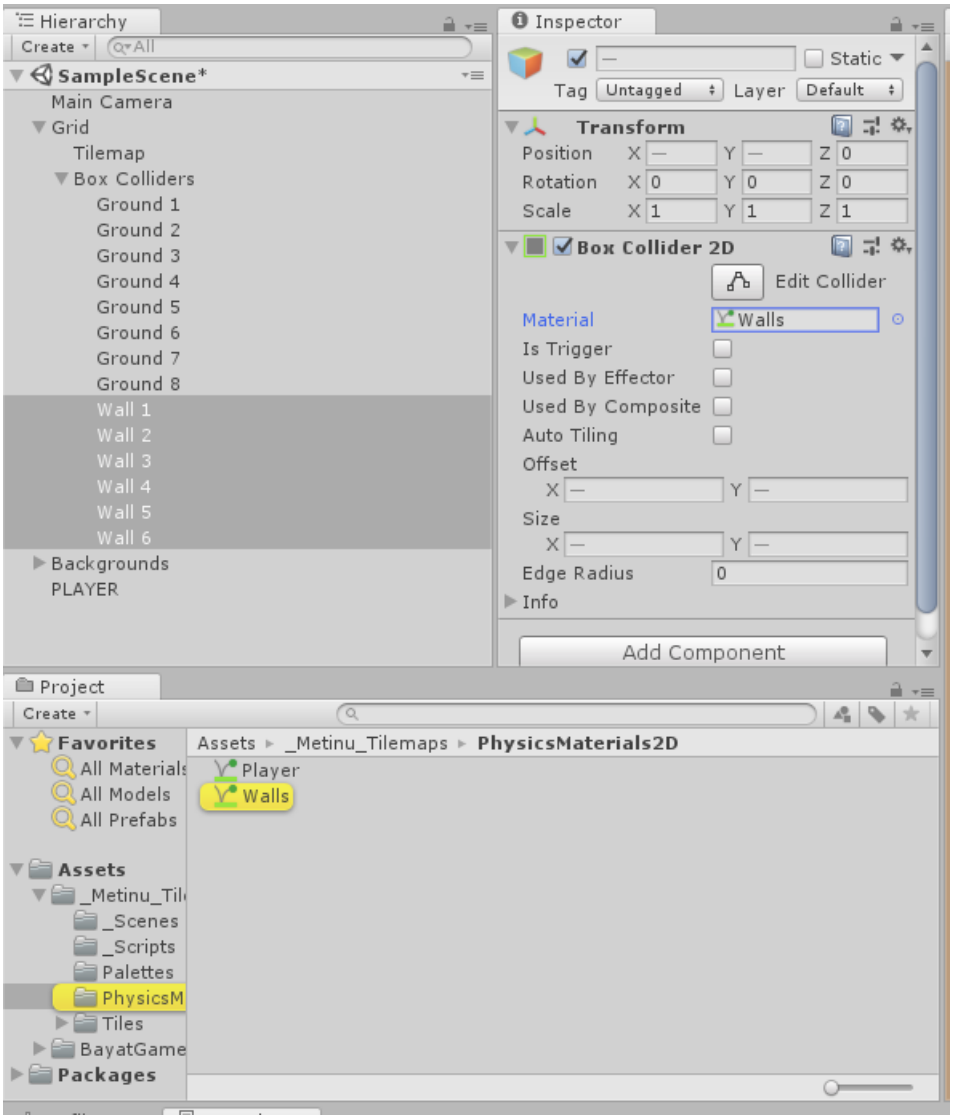


7.1. Material físico ultra-resbaladizo a las paredes

Crea un material físico con fricción 0 llamado “Walls”, “Resbaladizo”, “Ice”, o como tu decidas, y aplícaselo a los colliders de las paredes.



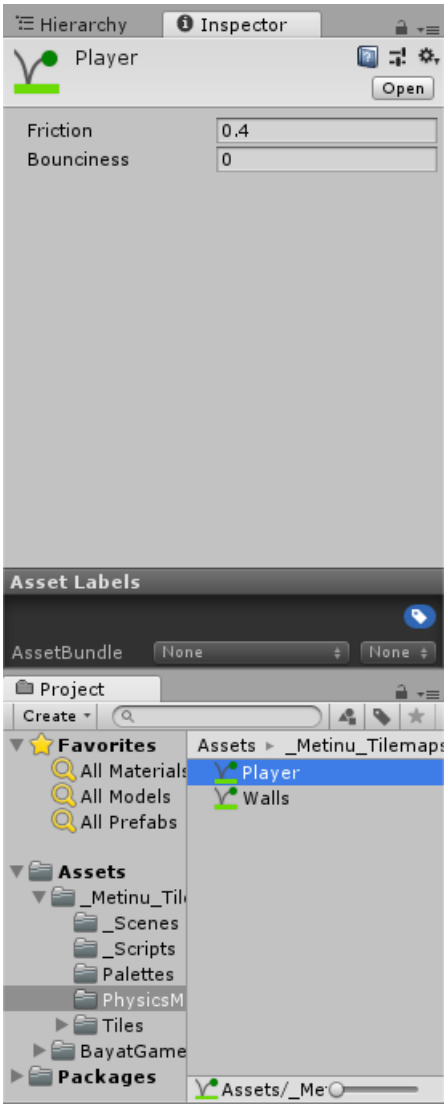
Sé que soy pesado, pero he aquí **otra muestra de por qué es importante separar los colliders de suelos y paredes**.



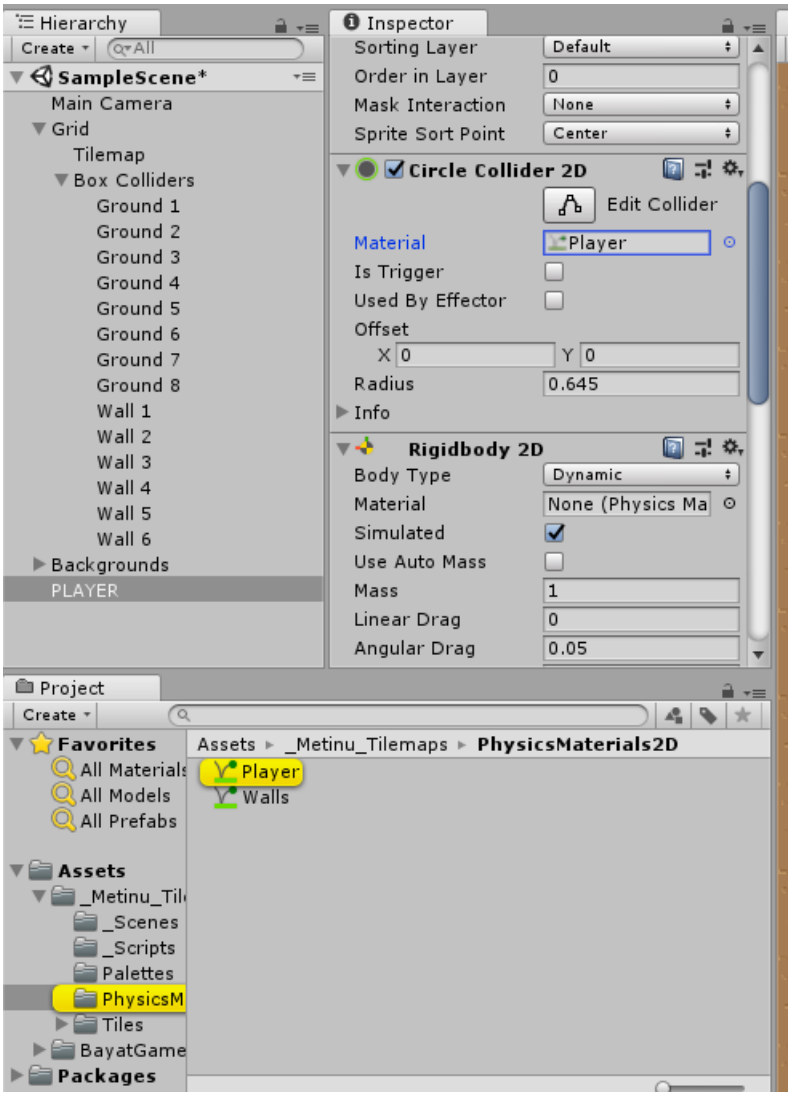
7.2. Material físico por defecto al personaje

Para que la interacción funcione, ambos colliders tienen que tener material físico, aunque sea uno por defecto.

Por tanto, crea un material físico para el jugador llamado “Player”, “Default”, “Standard”, o como tú decidas, y aplícaselo al jugador. Deja los valores por defecto (es decir, fricción en 0.4).



Asígnalo al collider del personaje:



8. Seguimiento de cámara sencillo

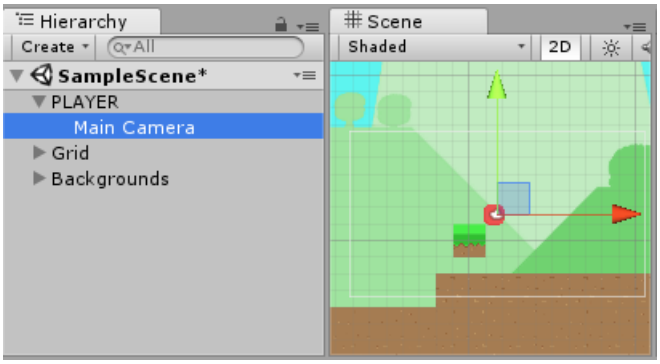
Ya tenemos al personaje listo para moverse por el escenario y explorar. Pero cuando se sale del encuadre de la cámara, lo perdemos. **Vamos a hacer que la cámara siga al personaje**.

El seguimiento de cámara es un arte en sí mismo, y hay muchas formas de conseguirlo. Normalmente, implica programación. Por suerte, Unity lanzó hace poco “Cinemachine”, un Asset Externo y oficial con componentes especiales preparados para hacer un seguimiento de cámara sin necesidad de programación.

Nosotros dejaremos esos berenjenales para más adelante, aunque, a continuación, dejo el link a un tutorial de brackeys acerca de cómo configurar este tipo de seguimiento:

2D Camera in Unity (Cinemachine Tutorial).
https://www.youtube.com/watch?v=2jTY11Am0Ig

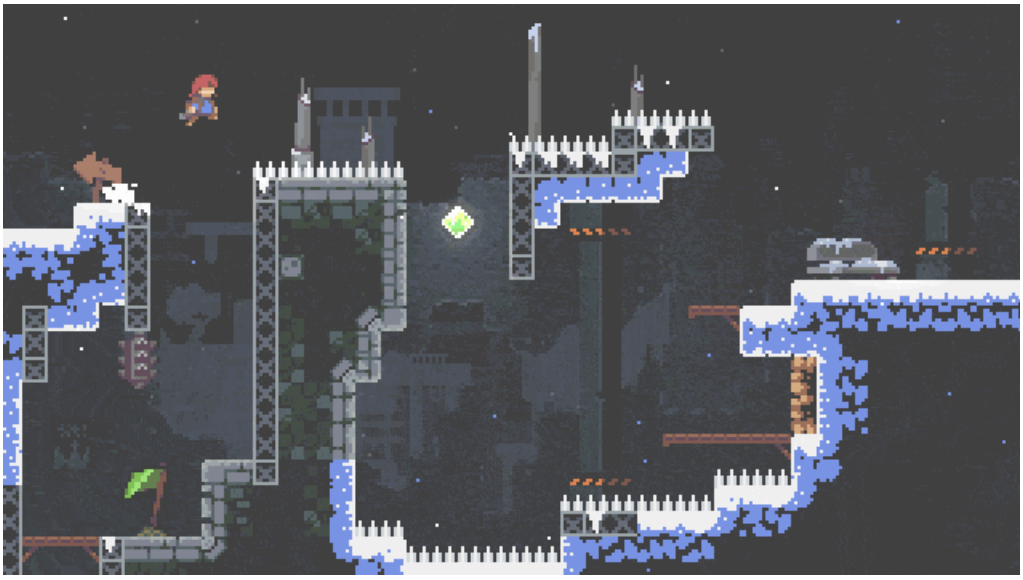
¿Cómo lo vamos a hacer nosotros? Simple y sencillo: vamos a emparentar la cámara al jugador, de forma que, cuando el jugador se mueva, la cámara irá tras de él.



Ten cuidado de, antes de emparentarlos, alinear la cámara con el personaje para dejar a éste en el centro. Si no, nunca lo verás.

9. Plataformas unidireccionales

Un tipo de plataformas muy común en videojuegos, son las plataformas unidireccionales, o **one-way platforms**, que te permiten atravesarlas sólo en una dirección (normalmente, de abajo hacia arriba), pero al caer desde arriba actúan como suelo y colisionas con ellas normalmente.



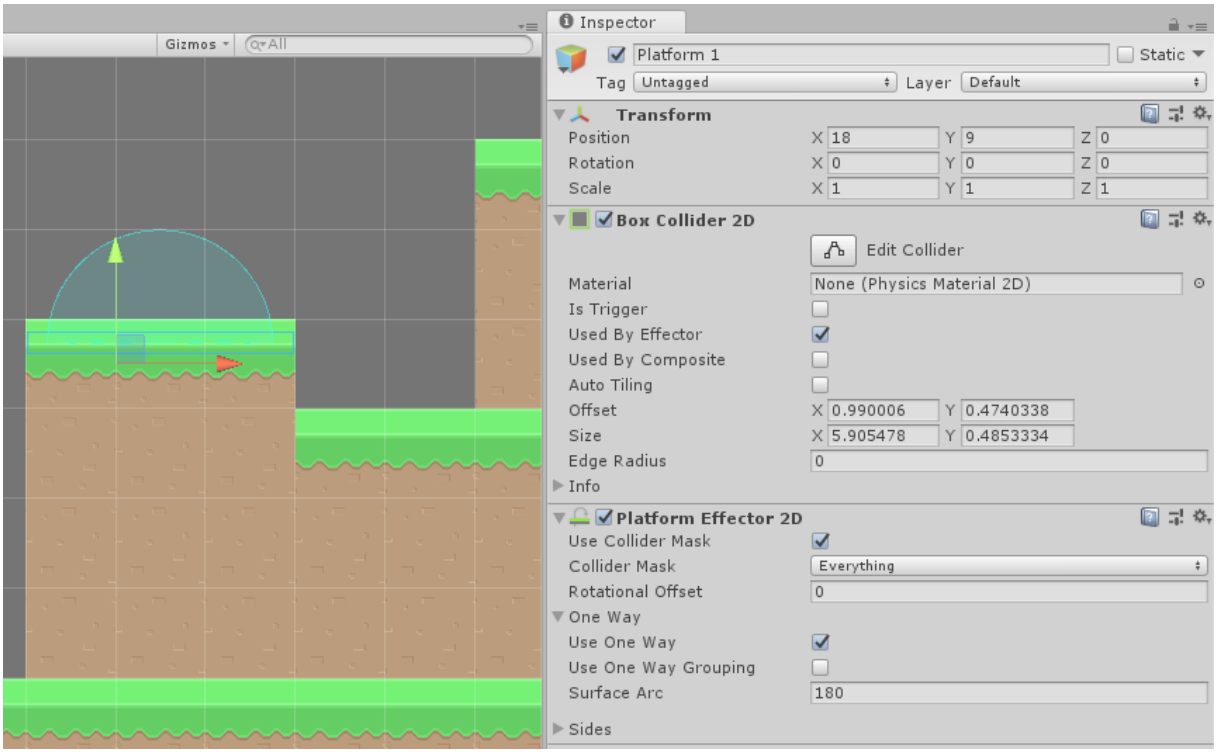
En esta captura de pantalla del videojuego “Celeste” hay 3 plataformas unidireccionales. Si no puedes identificarlas fácilmente no te preocupes, es porque has jugado muy poco. Por suerte, hay un lugar en el mundo para ti.

9.1 Cómo crearlas en Unity

Mediante el componente **Platform Effector 2D** que, añadido a cualquier collider, hace que éste sólo detecte colisiones en una dirección.

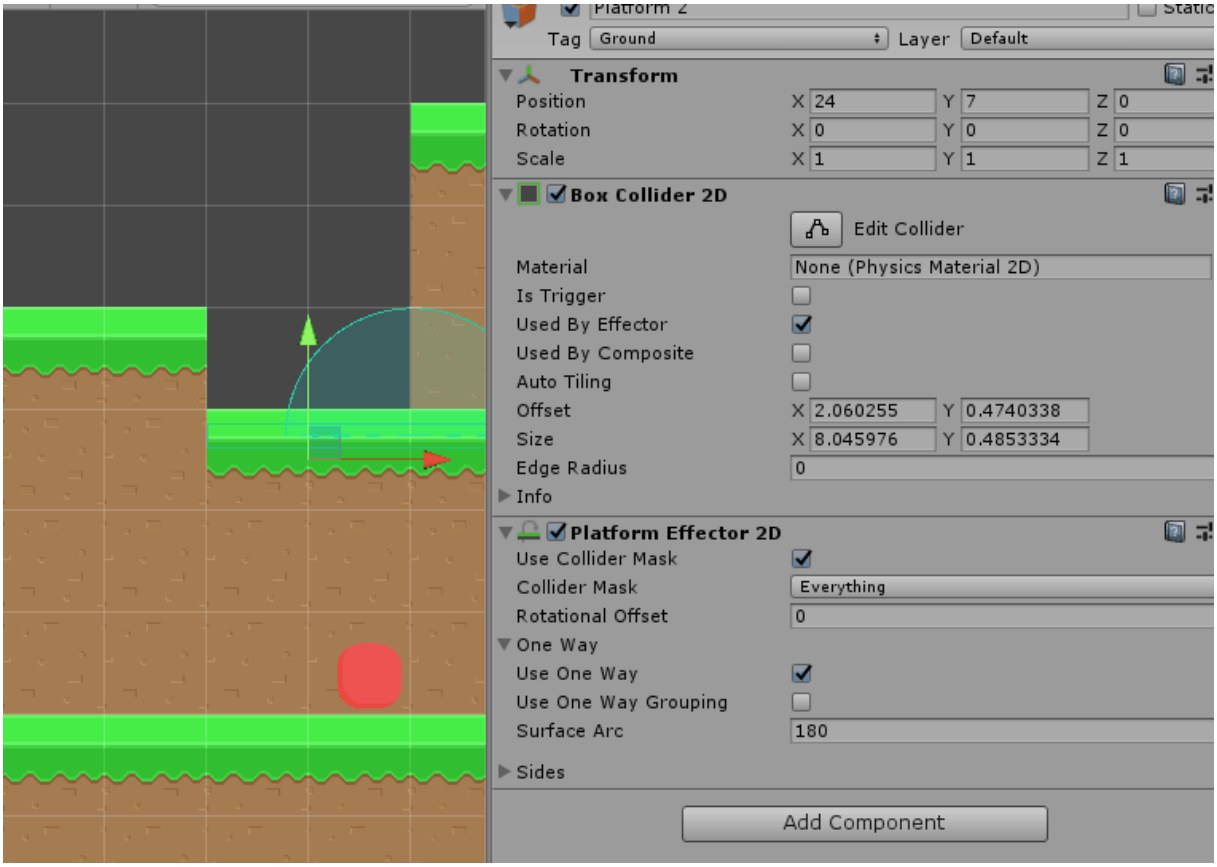
Añádelo a los colliders de las plataformas, **que también debes tenerlos en game objects independientes y**

taggeados como "Ground" para así detectarlos como suelo y que grounded sea true.



Cuando lo añadas, verás aparecer ese abanico al seleccionar el Game Object. Significa que las colisiones solo serán detectadas si vienen desde esa posición.

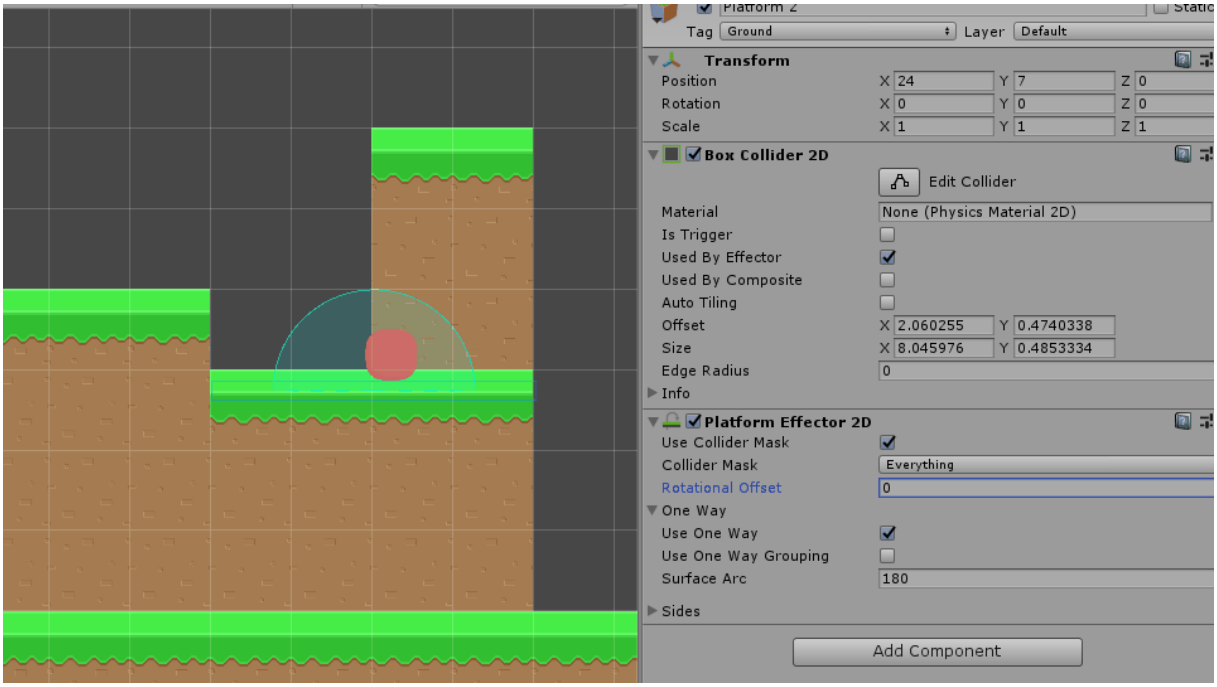
Este componente es un effector, **no es un collider**. Para que funcione, tiene que ir acompañado de un collider que tenga marcada la casilla **"Used By Effector"**.



9.2 Bajar de la plataforma al pulsar abajo

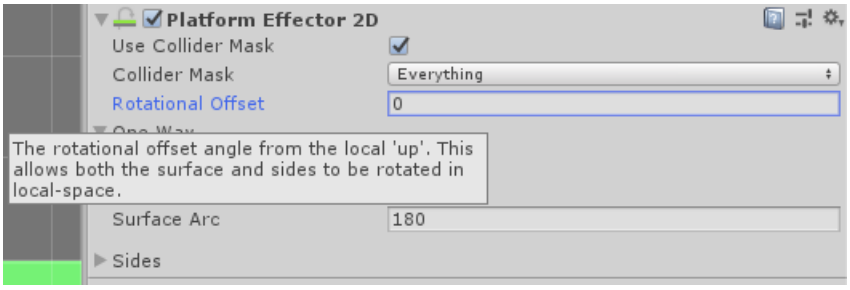
Para bajar de la plataforma, tendremos que **desactivar su collider al pulsar abajo**. Sin embargo, si hacemos esto, corremos el riesgo de que el collider vuelva a activarse mientras nosotros estamos en medio, lo que puede dar lugar a que el motor de físicas lo interprete como un golpe muy fuerte y nuestro personaje salga disparado.

Para evitar ese bug, lo solucionaremos **invirtiendo el abanico del platform effector al pulsar abajo**. Por tanto, tendremos que programar un script que haga lo siguiente:



A este script lo podemos llamar **VerticalPlatform.cs**, por ejemplo, y lo agregaremos a todas las plataformas en las que queramos conseguir este efecto.

En este script, accederemos al componente **PlatformEffector2D**, inicializándolo mediante `GetComponent<PlatformEffector2D>()` en el método `Start()`. Dentro de él, queremos acceder a su variable **Rotational Offset** como hacemos manualmente en el gif anterior, y cambiarla a 180º cuando pulsemos hacia abajo.

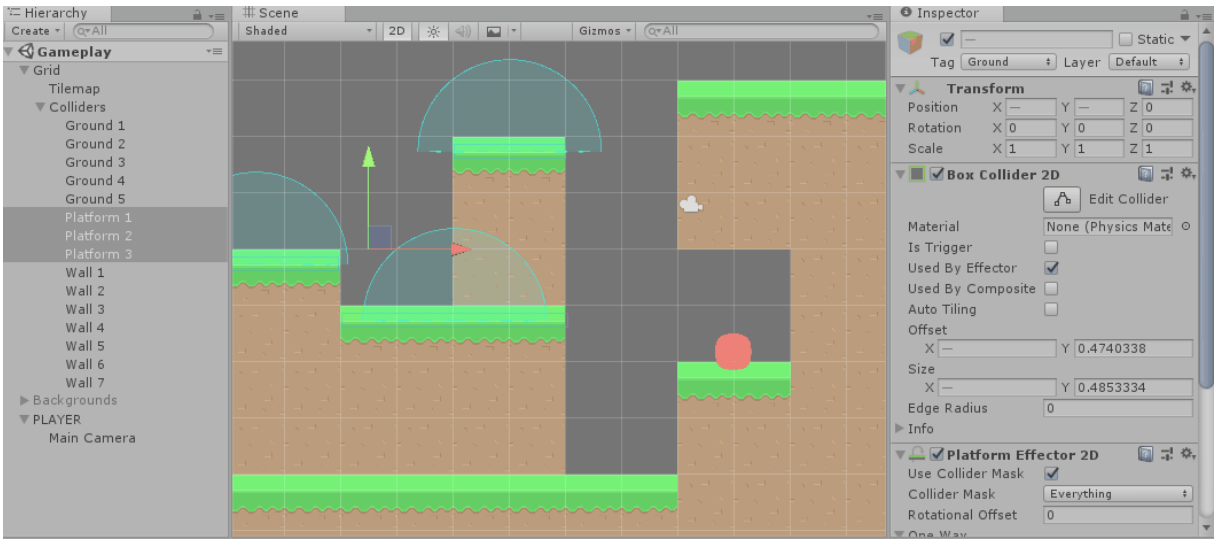


Para comprobar si estamos pulsando hacia abajo, en todo momento (en `Update()`) comprobaremos si el Input del eje vertical es menor que 0, mediante **Input.GetAxis("Vertical")**. Si es menor que 0, igualaremos el rotational offset a 180. En cualquier otro caso (else), igualaremos el rotational offset a 0.

```
void Update()
{
 // Si pulsamos hacia abajo
    if (Input.GetAxis("Vertical") < 0f)
    {
     // Invertir platform effector
        myPlatformEffector.rotationalOffset = 180f;
    }
    else // En cualquier otro caso:
    {
     // El platform effector se queda recto
        myPlatformEffector.rotationalOffset = 0f;
    }
}
```

> No copies el código a ciegas. ¿Has inicializado tu variable de tipo PlatformEffector2D en el Start, al igual que hacíamos en el Flappy Bird con el rigidbody? Si no lo has hecho, a qué esperas. Tienes dos opciones:
>
> - Usa `GetComponent<PlatformEffector2D>()` en el método `Start()`, como hacíamos en el Flappy Bird.
> - O haz la variable pública y arrástrale la referencia desde el inspector.

Una vez agregamos los componentes a las plataformas, obtenemos el siguiente efecto al pulsar abajo:



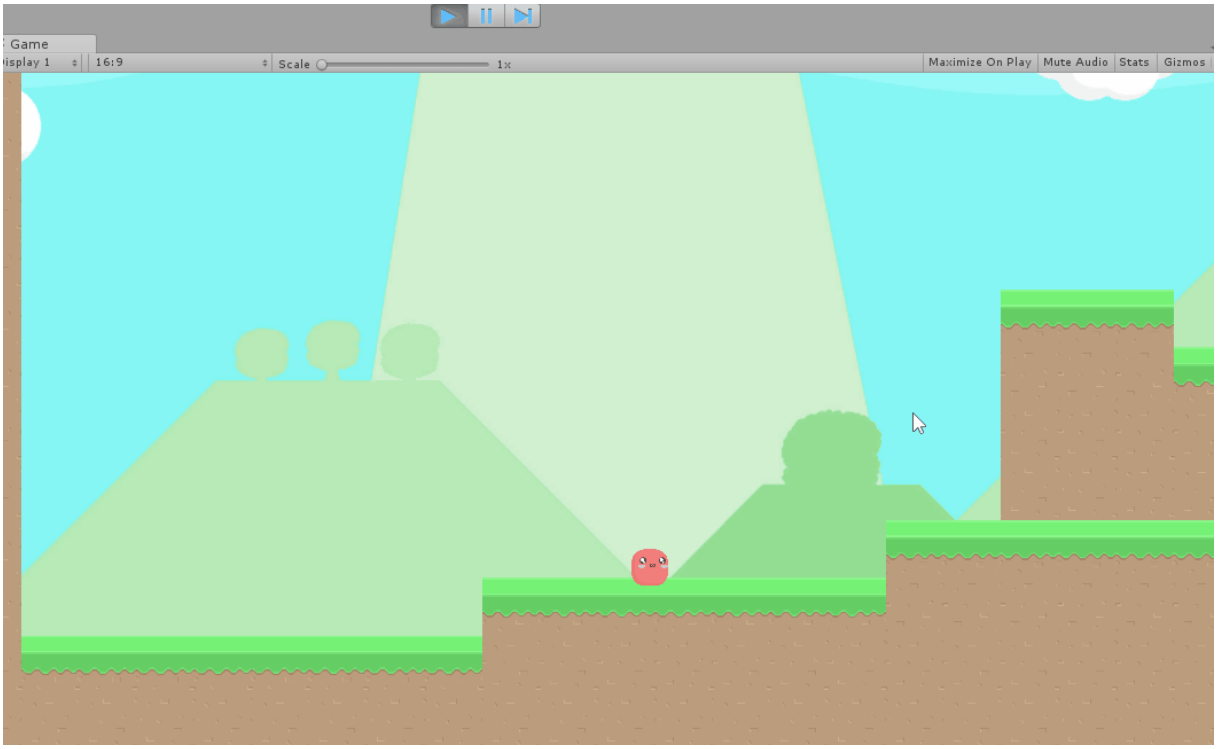
10. Invertir sprite del personaje al moverse

Hasta ahora nuestro personaje era solo una bola de carne rosa, por lo que no nos hemos dado cuenta de que en ningún momento se está girando para mirar hacia derecha o izquierda. Antes de programar esta funcionalidad, voy a modificar mi sprite para que tenga personalidad y enfrente una dirección (que por defecto será la derecha). Haz tú lo mismo con el tuyo.

10.1. Dotar al personaje de dirección

Para ello, me he abierto la imagen del sprite con photoshop, me he abierto Google y he buscado por “kawaii faces”, me he descargado una con transparencia, y se la he puesto encima. Me ha quedado así de cuqui:

Ahora ya podemos darnos cuenta de que, efectivamente, al caminar hacia la izquierda no se da la vuelta, por lo que hace un moonwalk extraño.

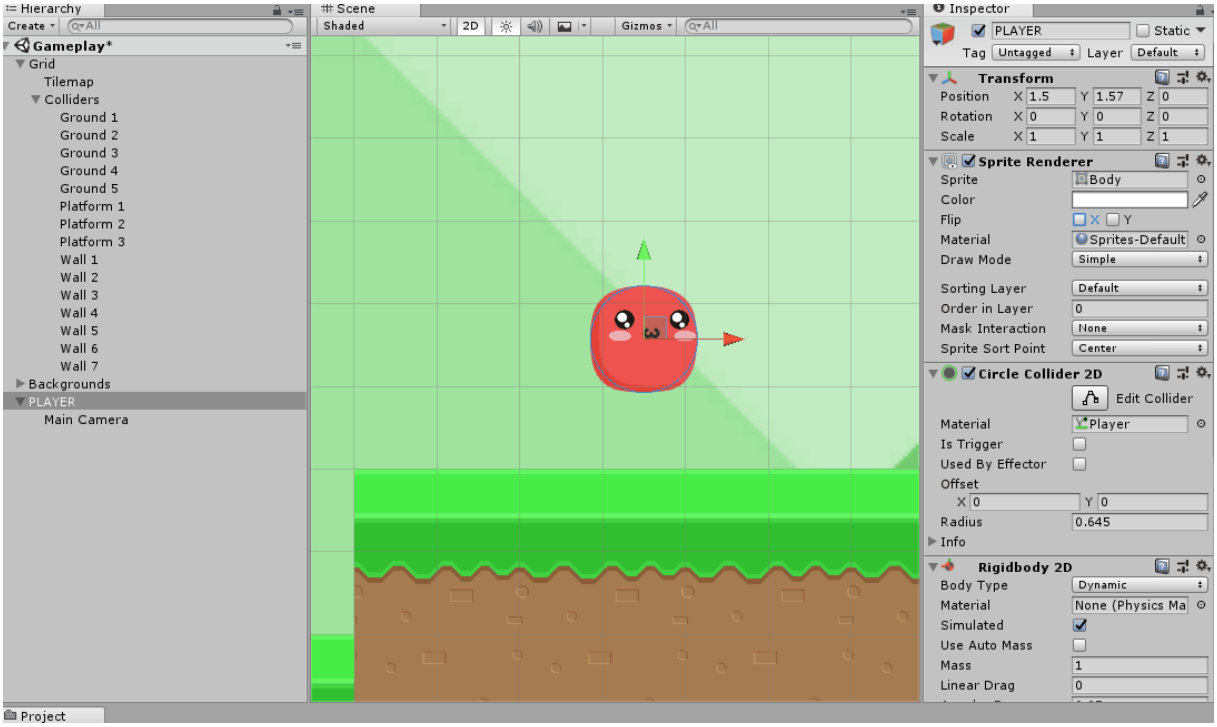


Es normal que no se de la vuelta, en ningún momento le hemos dicho que lo haga. Vamos a solucionarlo.

10.2. Invertir dirección del sprite al caminar

BUG: camina de espaldas // FIX: dar la vuelta al sprite al caminar

Vamos a solucionarlo tildando la casilla de **“Flip X”** como hago manualmente en el siguiente gif, pero automáticamente, mediante código.

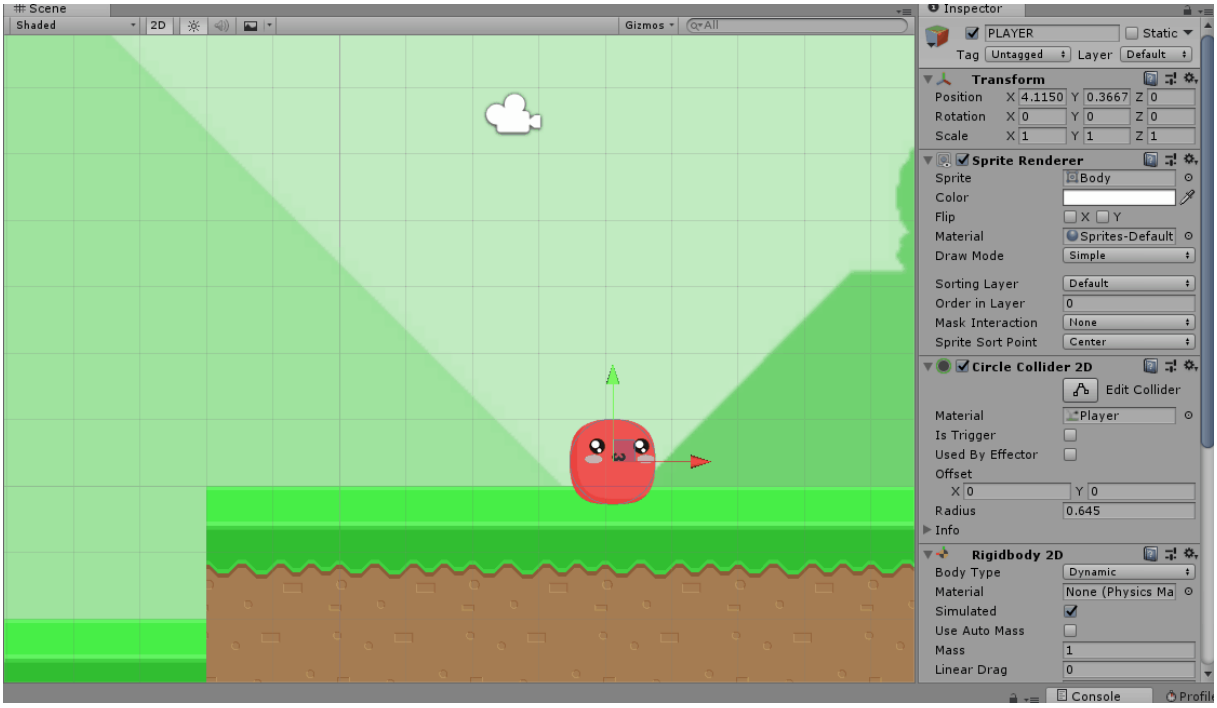


¿Cómo le decimos al script **CharacterController.cs** que haga eso? Accedemos al componente SpriteRenderer con `GetComponent<SpriteRenderer>()`, al `Start()` y, dentro de él, a su variable pública **mySpriteRender.flipX**, que podemos poner en true o en false según nos convenga.

En todo momento (en el `Update()`), dependiendo del Input horizontal leído, si es mayor que 0 miramos hacia la derecha (flipX = false), y si es menor que 0 miramos hacia la izquierda (flipX = true).

```
[...]
// Si vamos hacia la izquierda
if (playerInput.x < 0f)
{
 // Miramos hacia la izquierda
    mySpriteRenderer.flipX = true;
}

// Si vamos hacia la derecha
if (playerInput.x > 0f)
{
 // Miramos hacia la derecha
    mySpriteRenderer.flipX = false;
}
[...]
```

Con este poquito de código, ya tenemos al personaje mirando en la dirección en que camina:



En este gif puedes incluso comprobar, si miras a la ventana del Inspector, cómo el script hace su trabajo automáticamente, tildando automáticamente la casilla FlipX por nosotros. **¿No es flipante lo listo que es el código?**

Continuará

A partir de aquí, ya tenemos al personaje más que listo para moverse por el escenario.

Como se dijo en la introducción, ahora se avecinan cosas interesantes:

- Crear físicas del personaje avanzadas (salto de pared, doble salto, dash, nadar, volar...)
- Ponerle animaciones al personaje
- Crear power ups para saltar más, o más vidas, o armas, magias, etc.
- Crear plataformas móviles
- Poblar el mapa con trampas y enemigos
- Mejorar el seguimiento de la cámara para que sea más suave
- Crear más niveles, a cada cual más difícil, y con diferentes estéticas.
- Crear un boss final.
- Crear menús.

Y ya tendríamos el juego completo.

¿Vas imaginándote cómo hacer todas estas cosas? Si tienes mucha prisa, pregúntame y te doy recursos para correr más, y si te da vergüenza preguntarme, ya sabes, usa Google y YouTube...

Continuaremos en el tercer trimestre.