Práctica: Primer juego con Unity paso a paso

1 Introducción

Teniendo en cuenta los resultados de aprendizaje y competencias que el alumnado debe completar cursando este módulo, este proyecto contribuye a conseguir el siguiente:

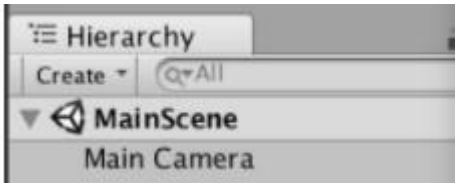
- **RA 5.** *Desarrolla juegos 2D y 3D sencillos utilizando motores de juegos*

2 Iniciando el Proyecto

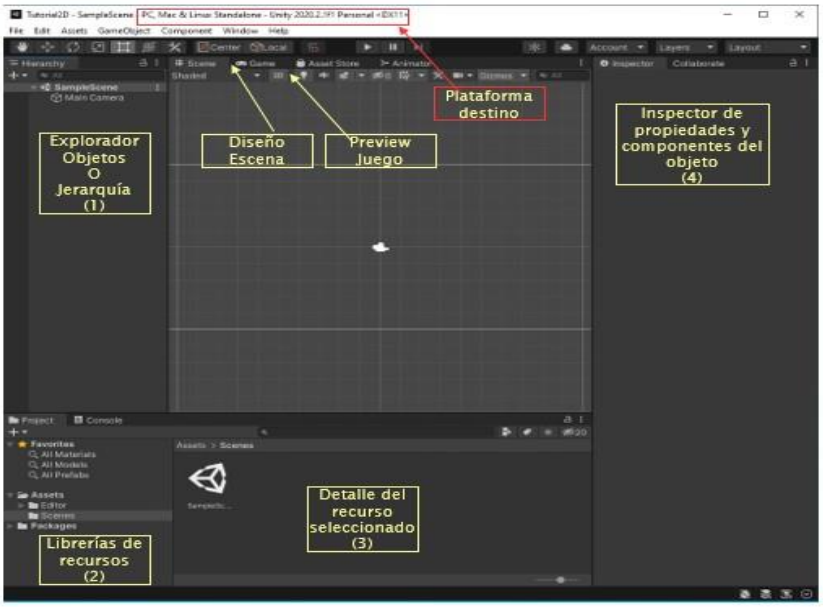
Comenzamos abriendo el Hub y seleccionamos **New** y pulsamos sobre la versión de Unity que queremos utilizar, en caso de tener más de una instalada.

Dentro de los templates, bajamos y pulsamos en **Mobile 2D**. Si no está instalado tendremos que descargarlo, para ello pulsamos en la flecha. Indicamos el nombre del juego. **SunnyLand** puede ser un buen nombre para nuestro juego, aunque tenéis plena libertad para bautizar a vuestra creación.

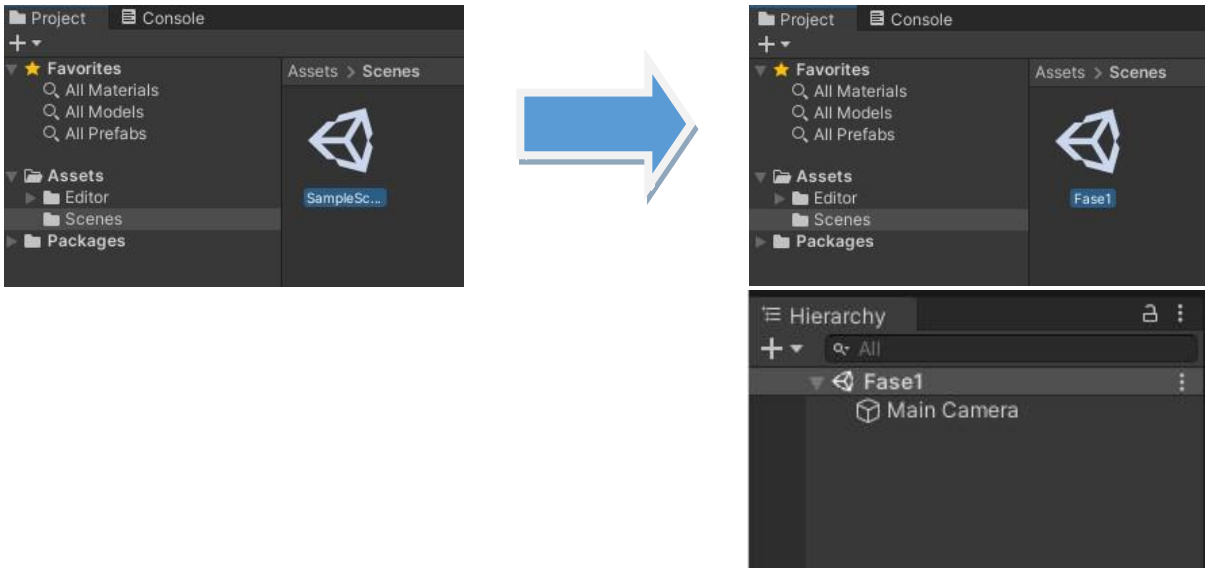
Lo primero que vamos a ver en el navegador jerárquico (izquierda) son:



Unity se basa en **ESCENAS**, por las que puede pasar el personaje, teniendo cada una su cámara correspondiente. Esta es la escena que se crea por defecto y con la que vamos a trabajar. A continuación, se muestran las partes principales del espacio de trabajo.



Seleccionamos en **Librería de Recursos (2)** la carpeta **Scenes** dentro de **Assets** y a la escena **SampleScene** la renombramos como **Fase1** para ponerle un nombre más descriptivo. Hay que tener en cuenta que cuando realicemos cambios como, por ejemplo, el nombre, estaremos modificando los nombres de los ficheros correspondientes. Podemos comprobarlo pulsando sobre **Assets – Show in Explorer.**



Una vez renombrada, pulsamos **File – Build Settings** y seleccionamos Android y hacemos clic en el botón **Switch Platform**. De esta manera cambiaremos la plataforma ahora que no hay ningún recurso añadido y la conversión de plataforma se hace más rápido. Por defecto, Unity crea el juego para PC Desktop. Esta operación puede tardar tiempo.

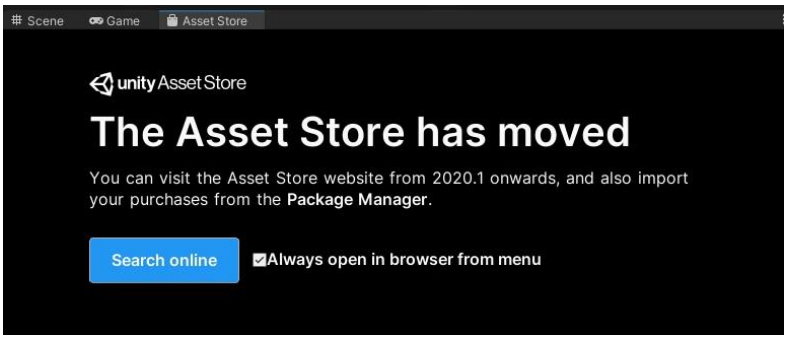
3 La Escena

Para la realización de un juego es necesario combinar no solo la programación sino también otros aspectos como los gráficos y los efectos de sonidos. Si no se pueden abarcar todos estos aspectos, será necesario contar con recursos externos que podamos utilizar (o contratar a alguien que lo pueda hacer). Gracias a Internet podemos encontrar numerosos sitios donde descargar recursos. Lo siguientes son algunos ejemplos:

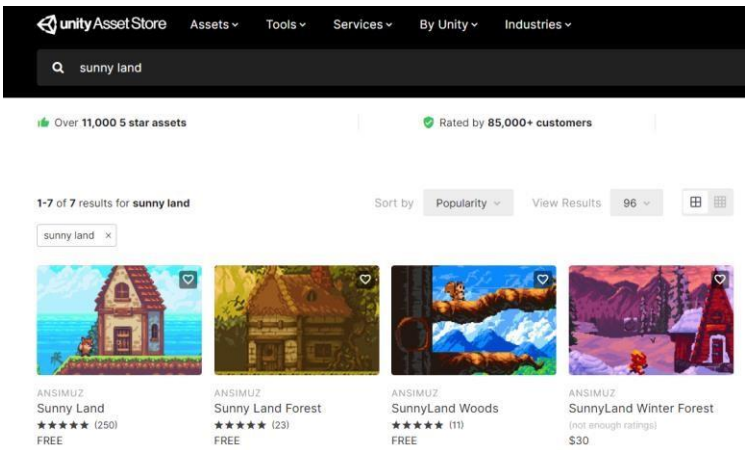
- Unity Asset Store - 2D Environments & Characters | Unity Asset Store
- Kenney - https://www.kenney.nl/assets
- Open Game Art - https://opengameart.org/art-searchadvanced?keys=&field_art_type_tid%5B%5D=9&sort_by=count&sort_order=DESC

En nuestro caso vamos a usar el **SunnyLand** del Unity Store. A continuación, se indican los pasos a seguir:

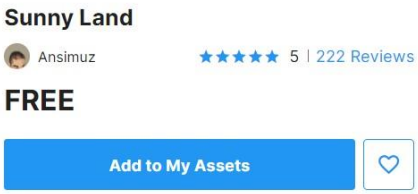
Con la sesión iniciada en Unity, vamos a la opción del menú **Window** y seleccionamos la opción **Asset Store.** Al hacer clic en **Search online** nos abrirá el enlace en el navegador.



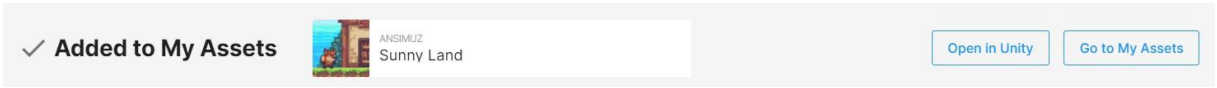
Buscamos el Asset **Sunny Land**.



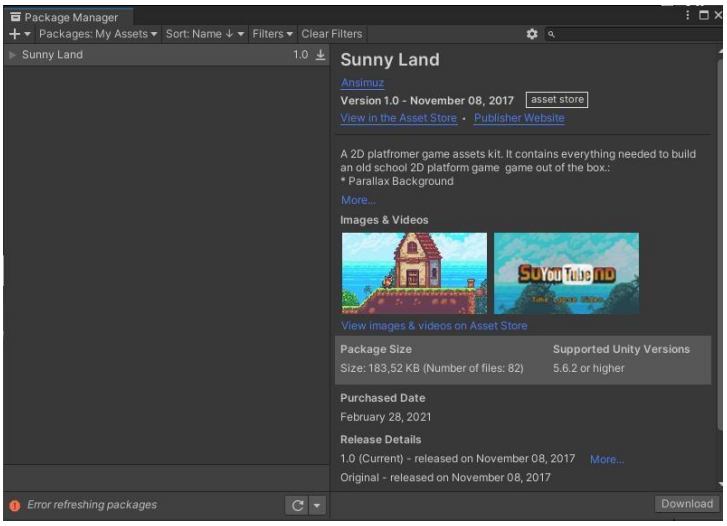
Tras la búsqueda hacemos clic en **Add to my assets**



Tras Aceptar la licencia, hacemos clic en Open in Unity.



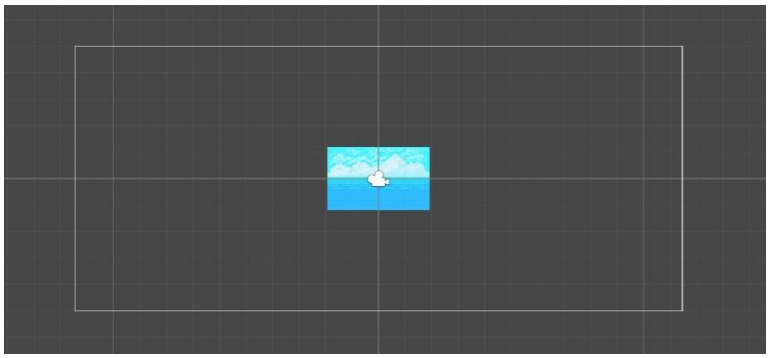
Se abre el **Package manager** (este gestor nos permitirá gestionar todos los recursos instalados en Unity y disponibles para descargar). Vemos que aparece el paquete seleccionado y pulsamos sobre **Download.**



Una vez descargado pulsamos en **Import**, para añadirlo a nuestro proyecto. Esta operación puede tardar tiempo.

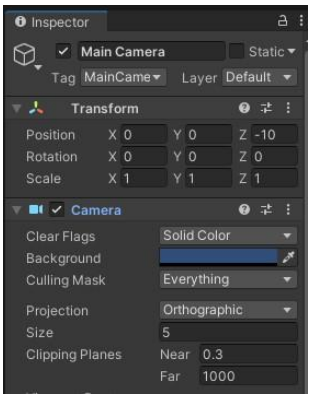
Con todos estos recursos podremos comenzar nuestro juego.

Dentro de la carpeta (2) Sunnyland – Artwork – Environment, arrastramos el fondo back al explorador de objetos (1). Se puede arrastrar a la raíz o al objeto Main Camera para que nos que de asociado a la cámara (no tiene efecto, más que de organización)



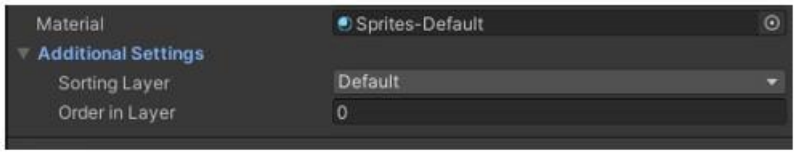
Ajustamos el fondo, seleccionándolo en el explorador y aumentándolo con los tiradores correspondientes. Usa las herramientas ubicadas en la barra superior para mover o ampliar cualquier objeto.

Igualmente podemos ajustar la cámara. Para ello seleccionamos la cámara en el explorador de objetos y desplegamos la **Camera** en el **Inspector** para que nos muestre los tiradores y poder moverlos para ajustar la cámara. En el cuadro inferior derecha puedes ver cómo quedará el resultado final.



Una vez ajustada la cámara podemos hacer zoom a la escena con la rueda del ratón y mover el canvas con el ratón

Si seleccionamos el objeto *back* por defecto, existe una capa (al igual que en otros programas de diseño) llamada **Default** donde podemos ordenar los objetos modificando el **Order in layer**. Pero es recomendable, para este tipo de juegos poner varias capas, que nos permitan más flexibilidad.



En la sección **Additional Settings** del **Inspector** del objeto *back*, desplegamos el **Sorting Layer + Add Sorting Layer**… y pulsando sobre el símbolo + y moviendo las creamos la siguiente estructura:

Una vez creadas las capas, seleccionaremos el objeto Back en el explorador (1) y seleccionaremos **Background** en el **Sorting Layer**. Esto permitirá que la imagen que acabamos de poner quede siempre al fondo y no tape los objetos en primer plano.

Ahora creamos nuestro **Tilemap** (mosaico):

Pulsamos con el botón derecho en el explorador (1) y seleccionamos **2D Object – Tilemap – Rectangular**

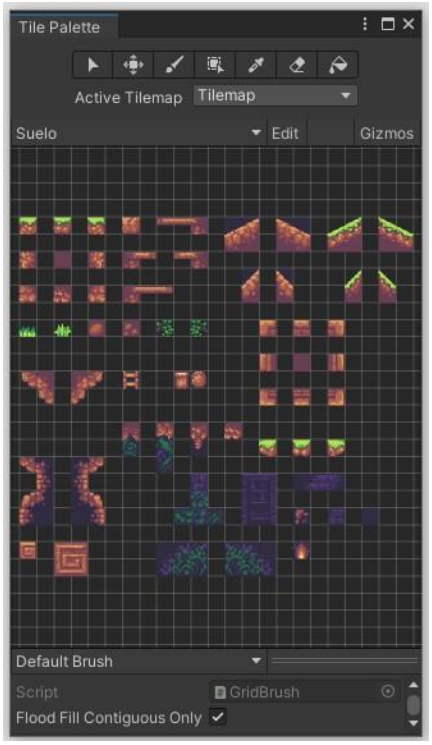
Esto creará una rejilla (grid) que nos permitirá ajustar mosaico. Renombramos grid (menú contextual + Rename) como Tiles para identificarlo como diseño del escenario.

Pulsando sobre el grid, veremos que en la esquina inferior derecha aparece Open Tile Palette que pulsaremos. Si no aparece directamente, se puede acceder a él desde el menú Windows – 2D – Tile Palette.

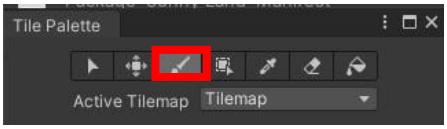
En la nueva ventana seleccionamos Create New Palette, la nombramos como Suelo y la salvamos en la carpeta Assets

Ahora de nuevo vamos a la carpeta (2) Sunnyland – artwork – Environment y pulsamos en la flecha al lado tileset-sliced en (3). Esto abrirá todos los fragmentos dentro de la imagen, que seleccionaremos y arrastraremos al Tile Palette.

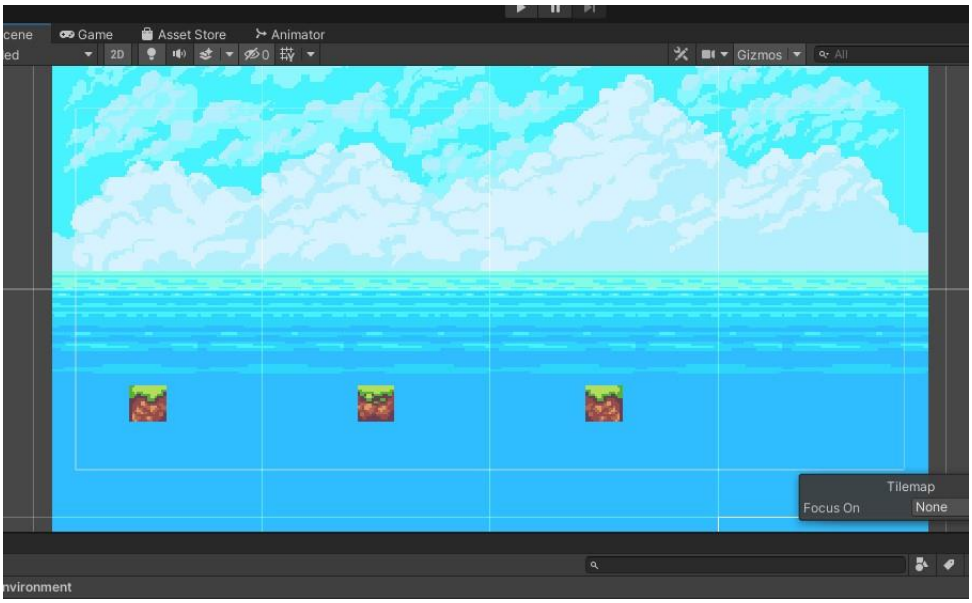
Cuando solicite donde guardar los assets, creamos el directorio Tiles dentro de Assets, para mantener los componentes ordenados. El resultado debería quedar así:



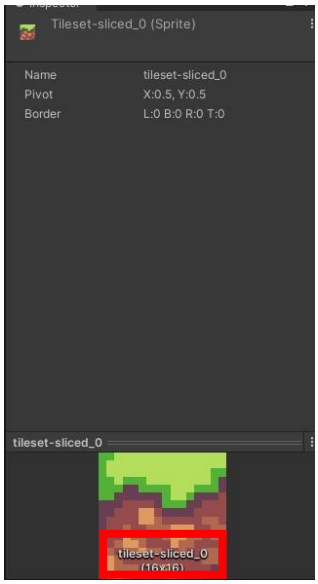
Ahora seleccionando una o varias cuadriculas (arrastrando sobre las cuadrículas de la paleta), pulsando sobre el pincel en la parte superior y marcando la opción Edit, podremos pintar nuestro nivel.

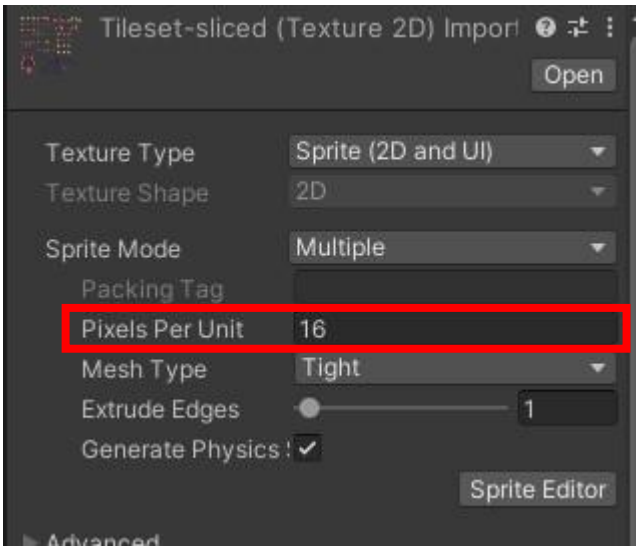


Lo que vemos es que, al pintar las casillas, estas quedan muy separadas.

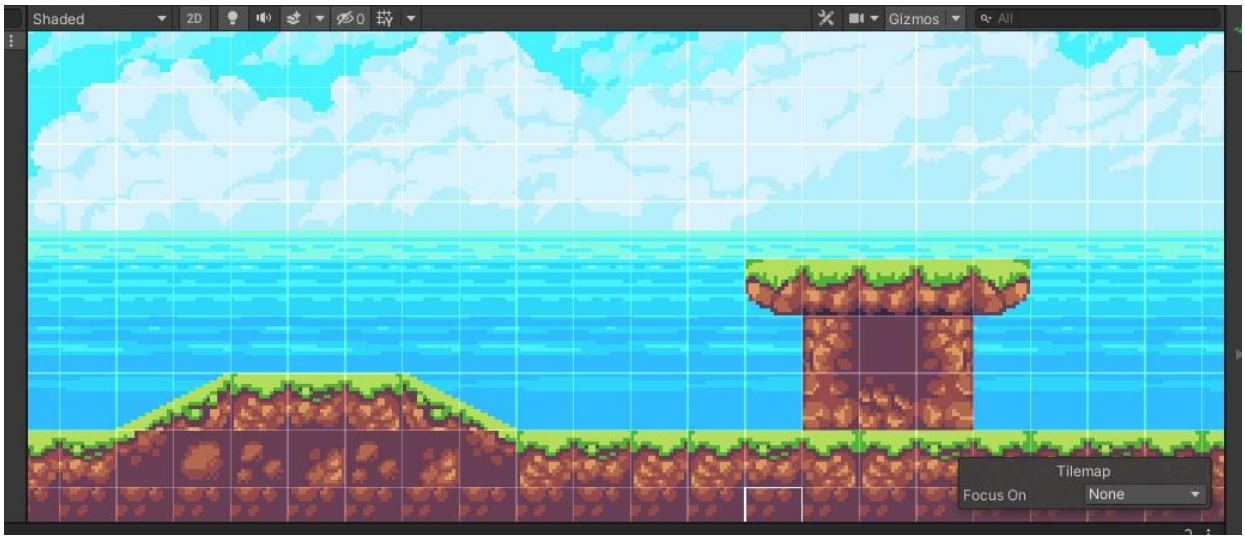


Esto se debe a que no hemos configurado adecuadamente los **PIXELS POR UNIDAD** de las imágenes con las que hemos creado el tileset. Si desplegamos el archivo tileset sliced (dentro de environment) en (3) y seleccionamos cualquier imagen, veremos que en la parte de la derecha se muestra la imagen y la información sobre su tamaño, que en este caso es, 16x16 pixeles. Como queremos que cada cuadro de la rejilla (unidad de medida dentro de Unity) albergue una imagen del mapa, tenemos que configurar el valor de los pixeles por unidad a 16 (16 pixeles / 1 celda = 16), Para ello, seleccionamos el archivo tileset sliced y en el inspector ponemos el valor Pixels per unit en 16 y pulsamos Apply.

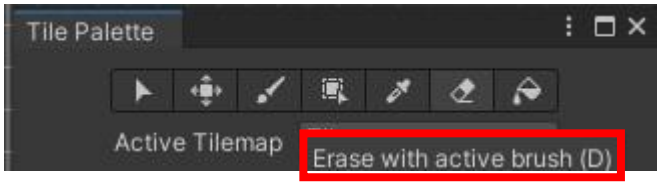




Ahora diseña el nivel a tu gusto. No es necesario hacerlo demasiado complicado ahora. Una cosa como se puede ver en el ejemplo puede ser suficiente:



Si nos equivocamos necesitamos borrar, la herramienta está incluida en **Tile Palette**



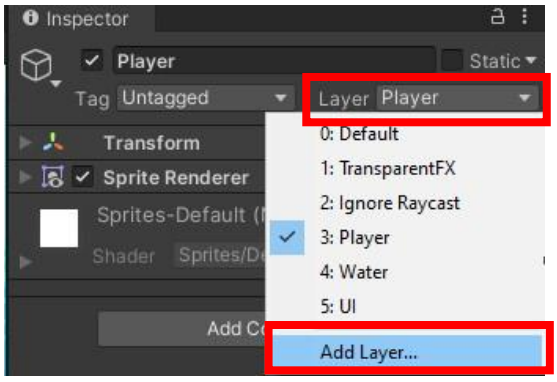
4 Movimiento en 2D

Ahora es momento de añadir nuestro personaje y dotarlo de movimiento y control.

Inicialmente añadiremos un personaje. Desde la carpeta Sunnyland – artwork – sprites – player – idle en (2), seleccionamos las imágenes en la carpeta (3) y ajustamos los pixels por unit a 16, pulsamos Apply y arrastramos el primer fotograma al explorador de objetos.

Lo renombramos como Player.

En el inspector de componentes, pulsamos sobre Layer – Add new layer. Añadimos una capa con nombre Player, para crear una capa donde situaremos a nuestro personaje y los elementos que pueden colisionar con él.

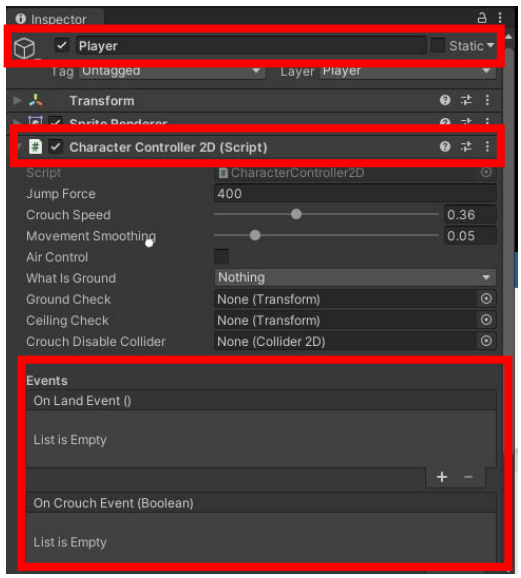


Podemos comenzar el control del personaje desde cero, pero a no ser que tengamos muy claro que queremos hacer, es mejor comenzar por un controlador ya existente que nos permita centrarnos en la creatividad y eliminar tiempo de desarrollo. En este caso, no merece la pena reinventar la rueda.

⭐ Arrastramos el script del controlador suministrado (CharacterController2D.cs) a la carpeta Assets (2).

⭐ Una vez el script está dentro de Unity y compilado correctamente (se compila automáticamente), **seleccionamos el objeto Player** en el explorador y arrastramos desde la librería de recursos (3) el script del controlador al inspector de propiedades (4), y nos aparecerá un nuevo grupo en el Inspector, de nombre el del fichero del script.

⭐ Veremos cómo aparece un nuevo componente de tipo SCRIPT en (4), con una serie de propiedades y eventos que ya vienen predefinidos y que de otra forma tendríamos que programarlos nosotros.



⭐ Vamos a configurarlo:

Jump Force: Fuerza del salto. 700

Crouch Speed: Indica cuánto queremos frenar el personaje cuando se agacha. 0.4 (40% de la velocidad del personaje)

Movement smoothing: Indica la suavidad que vamos a aplicar al movimiento. 0.05

Air Control: Si queremos controlar al personaje en el aire. En este caso lo chequeamos.

What is ground: Indicamos qué se va a considerar como suelo. Seleccionamos everything. Volvemos a abrir dicho desplegable y quitamos el tick de la capa Player, ya que no queremos que el jugador interfiera consigo mismo.

- En el desplegable Layer (parte superior del inspector) seleccionamos la capa Player para añadir al personaje a esa capa.

Ground check y Ceiling Check: Indica qué parte del personaje va a considerarse colindante con el suelo y qué parte se va a considerar la cabeza y, por tanto, chocar con objetos bajos, donde deberemos agacharnos. Para crear estos sensores, pulsamos sobre Player con el botón derecho en el explorador de objetos (1) – Create empty.

- Seleccionamos el objeto recién creado y pulsamos W para poder mover el objeto en pantalla. Aparecerá un eje de coordenadas que moveremos a la altura de la cabeza.

- Duplicamos dicho objeto con CTRL+D y el nuevo objeto lo movemos para identificar los pies.

- Renombramos ambos objetos como Ceilingcheck y Groundcheck respectivamente.
- Seleccionamos Player en el explorador de objetos y arrastramos los dos nuevos objetos del explorador (1) a las casillas correspondientes del inspector (4).

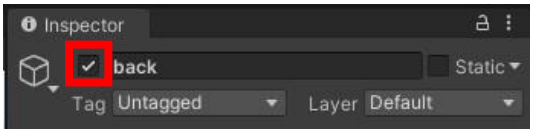
Con todos los parámetros configurados, es momento de añadir física realista al juego a través de las opciones de Unity para este apartado. Vamos a utilizar el componente **RigidBody 2D**, que es el que permite que los objetos respondan a las leyes físicas, como la gravedad.

⭐ Con el Player seleccionado, pulsamos Add Component en el inspector y buscamos Rigidbody 2D (está dentro de Physics 2D).

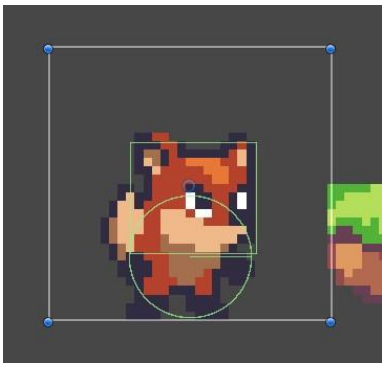
> **Gravity Scale**: 3, para hacer el personaje algo más pesado y darle realismo al juego. (1 es la gravedad standard terrestre)
>
> **Constraints – Freeze rotation – Z**. Bloqueamos la rotación en el eje Z para que el personaje no gire en pantalla y se mantenga siempre recto.

⭐ Si ahora pulsamos el PLAY, veremos que el personaje se cae (empieza a estar bajo el efecto de la gravedad) pero atraviesa el suelo. Esto se debe a que debemos añadir **COLLIDERS** (Colisionadores) o bordes que detecten las colisiones, tanto al personaje como al terreno.

⭐ Seleccionamos el background (objeto back) y lo ocultamos para poder ajustar mejor los colliders.



⭐ Seleccionamos Player y pulsamos la tecla F para hacer zoom sobre el personaje. En el inspector añadimos un nuevo componente, **Box Collider 2D** y pulsamos sobre Edit collider para ajustar la caja aproximadamente a la cara del personaje. Ahora añadimos un **Circle Colider 2D** y lo ajustamos al cuerpo. Para moverlo en el eje Y podemos pulsar en el inspector sobre la "Y" y mover el ratón a izquierda o derecha. Ajustamos el tamaño como antes (tecla F). El resultado deber quedar así:

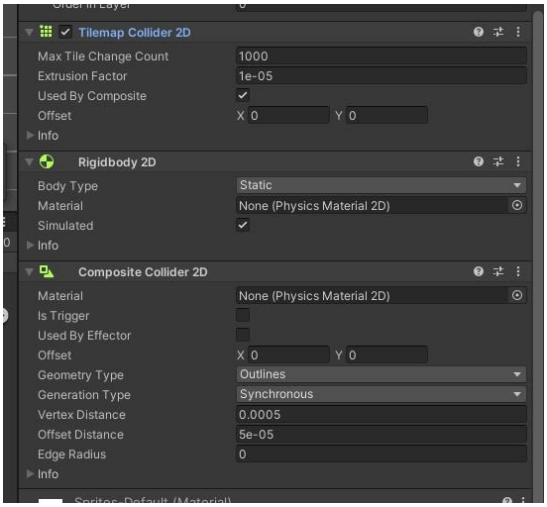


Estos dos collider se usan para que el personaje tenga colisiones más naturales, sobre todo cuando sube o baja por las cuestas, ya que de otra forma subiría a escalones.

Si pulsamos PLAY el personaje seguirá cayendo ya que todavía no tenemos collider asignado al suelo. Para ello, seleccionamos el Tilemap dentro de Tiles en el explorador y añadimos el componente **Tilemap Colider 2D** (vía el Inspector).

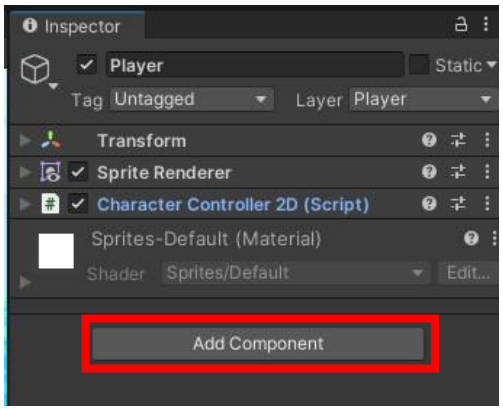
Vemos como Unity dibuja una rejilla verde claro alrededor de nuestro Tilemap y si ahora pulsamos PLAY, el personaje ya no se cae al vacío.

Para mejorar las colisiones, sobre todo en las pendientes, añadimos un nuevo componente al Tilemap llamado **Composite Colider 2D**. Esto añadirá este componente y un RigidBody 2D. En el Tilemap colider 2D, seleccionamos la opción **Used By Composite** con lo que las líneas de colisión quedarán definidas en el borde del suelo. Finalmente, en el RigidBody 2D seleccionamos el tipo **static**, ya que, si no, se nos caería todo el suelo al vacío. Puedes probar las diferentes opciones para ver qué ocurre (dynamic, kinetic, static)



Hecho todo eso ya podemos comenzar a escribir **nuestro script** para mover al personaje.

Con el personaje seleccionado, pulsamos **Add Component** y seleccionamos **New script.** Lo nombramos PlayerMovement (u otro nombre que elijas ya que será el nombre de nuestro componente de script para controlar al personaje). Pulsamos después Create and Add y se abrirá en nuestro editor seleccionado.

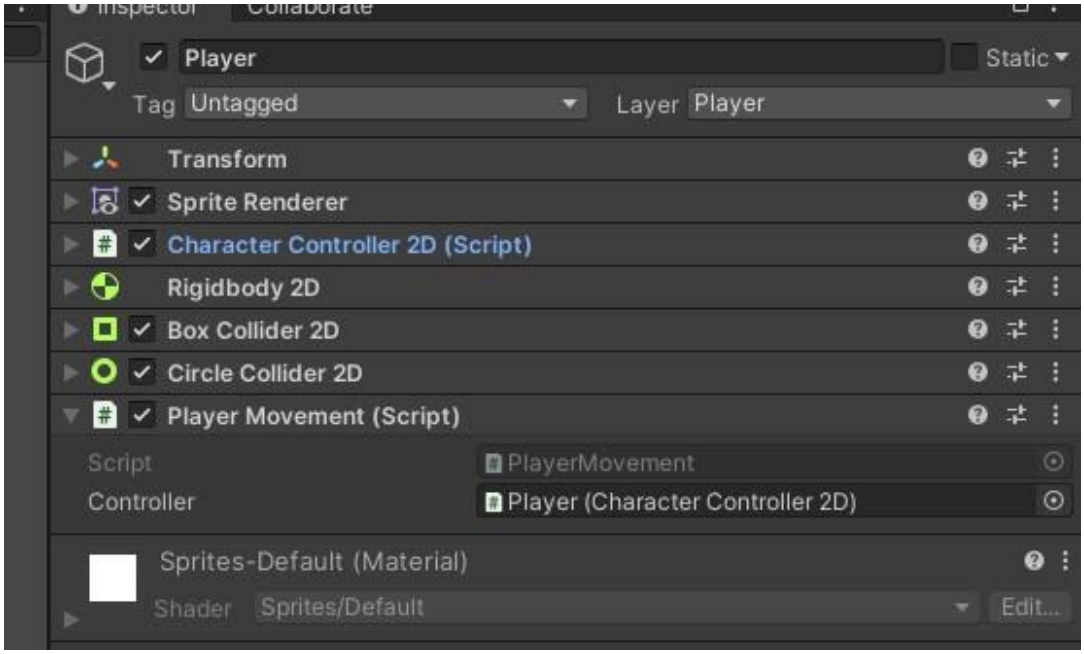


⭐ Lo primero que necesitamos es obtener una referencia al controlador del personaje (que configuramos en los primeros pasos) para poder acceder a sus propiedades y métodos. Para ello declaramos lo siguiente:

```
public CharacterController2D controller;
```

> *Como norma general, podemos decir que cualquier variable/evento declarada en el script como PUBLIC, aparecerá en el explorador de propiedades (4), dentro del componente de script para ser modificado desde Unity.*

⭐ Al salvar y volver a Unity, veremos que, después de compilar, en nuestro componente aparece un control con la variable que hemos puesto pública. Ahora arrastramos desde la parte superior del inspector (4) el controlador (Character Controller 2D) a esta casilla.



Una vez obtenida la referencia al controlador, necesitamos obtener la entrada del usuario para saber hacia dónde se mueve el jugador. La entrada del usuario puede venir desde el teclado, un gamepad, una pantalla táctil...etc. En cualquier caso, usaremos la variable **Input** para acceder a esa información, guardándola en una propiedad de la clase.

```
float horizontalMove=0f;
```

```
    // Update is called once per
frame     void Update()
    {         horizontalMove =
Input.GetAxisRaw("Horizontal");
    }
```

⭐ La función GetAxisRaw devolverá 1 si el personaje se mueve a la derecha y -1 en caso contrario. La función Update se llama una vez por frame (dependiendo cuantos frames o cuadros por segundo tenga el juego, que a su vez suele depender del hardware). Por lo tanto, estaremos escaneando la entrada del usuario por cada frame, asegurando una respuesta adecuada.

Pero ahora para mover el personaje, no es recomendable hacerlo en esta función. Existe otra función destinada a esto que es adecuada para cuando se usa el motor de físicas de Unity. Esta es **FixedUpdate**, que se llama un número de veces fijo por segundo, en vez de por cada frame como Update, evitando "tirones". Dentro de esta función llamaremos al método move del controlador (si no dispusiésemos de este controlador deberíamos llamar a las funciones addForce o MoveTo del objeto Player). Esta es la declaración del método:

```
public void Move(float move, bool crouch, bool jump)
```

Añadimos el siguiente código a la clase:

```
void FixedUpdate() {

controller.Move(horizontalMove*Time.fixedDeltaTime,fals
e,false) ;
}
```

Time.fixedDeltaTime es un valor que nos indica el tiempo desde la última vez que se llamó a esta función y es necesario para ajustar el movimiento, independientemente de las veces que se llame la función por segundo.

⭐ Si ejecutamos el juego veremos que el personaje apenas se mueve. Esto es debido a que la cantidad de movimiento que estamos pasando es 1 o -1 (multiplicado por el deltatime). Necesitamos definir una variable donde almacenemos la velocidad del personaje y multiplicarla por la entrada del usuario. La declararemos pública para que pueda ser modificada desde el componente visual.

```
   public float runSpeed=40f;
// Update is called once per
frame     void Update()
    {
        horizontalMove                                          =
Input.GetAxisRaw("Horizontal")*runSpeed;
    }
```

⭐ Ahora el personaje se debería mover de forma fluida por el escenario. Fíjate que el personaje cambia de lado al cambiar de sentido. Esto es otra ventaja del controlador, que cuando cambia el sentido del personaje, lo transforma en espejo. De nuevo, si no tuviésemos el controlador, tendríamos que codificarlo nosotros.

⭐ Necesitamos implementar el salto, para ello debemos responder a la pulsación del salto. Para ello, añadimos una propiedad a la clase para controlar cuando estamos saltando: `bool jump= false;`

⭐ Ahora actualizamos la variable en la función Update, cuando se detecte la pulsación del salto. `if(Input.GetButtonDown("Jump")) {`

`jump=true; }`

⭐ Añadimos la variable jump como parámetro a la función Move y la reseteamos a false para finalizar el salto.

```
void FixedUpdate() {

controller.Move(horizontalMove*Time.fixedDeltaTime,fals
e,jump);   jump = false;
}
```

⭐ Con el personaje saltando y corriendo, solo falta corregir un par de detalles. Primero vemos que el personaje se hunde ligeramente en el suelo cada vez que salta y cuanto más alto llega más se hunde al aterrizar. Para solucionar esto, cambiamos la propiedad **Colision Detection** dentro del componente Rigidbody 2D del player de continuous a discrete.

Otro problema es que el personaje se “enganche” en las plataformas. Para o solucionar esto crearemos un material que resbale y lo aplicaremos al box collider. Pulsamos sobre el símbolo + de la librería de recursos (2) y seleccionamos **2D – Physics Material 2D**. Lo renombramos como slippery (resbaladizo) y le asignamos la fricción a 0. Ahora arrastramos ese material a Material del Box Colider y del Circle Colider de Player.

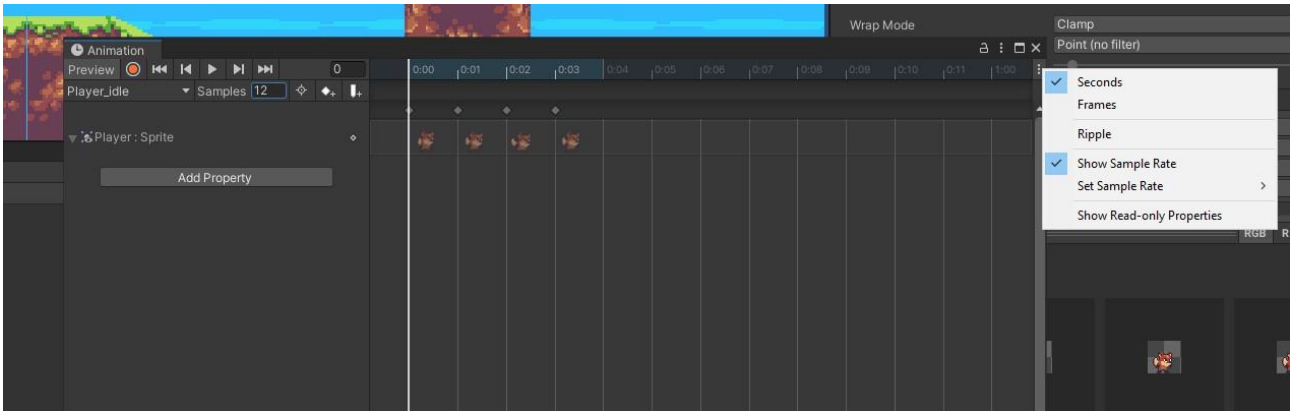
5 Animación del Personaje

Una vez que tenemos el movimiento debemos animar el personaje para dar un aspecto más natural al juego. Se pueden implementar dos tipos de animaciones: la clásica con varios fotogramas o la basada en un solo dibujo con un esqueleto y articulaciones. En este caso usaremos la primera.

⭐ Vamos a abrir la ventana para generar las animaciones. ***Window – Animation – Animation***.

⭐ Seleccionamos Player y pulsamos Create en la ventana de animación. Creamos un directorio para almacenar todas las animaciones (Animations) y creamos la animación con el nombre Player_idle (**el nombre del objeto + _ + animación** como convenio). Creamos inicialmente la animación de idle (inactivo) para que sea también el primer estado del **grafo de animaciones**.

⭐ En la esquina superior derecha de la pantalla pulsamos en el indicador de menú (tres puntos) y seleccionamos Seconds y Show Sample rate.



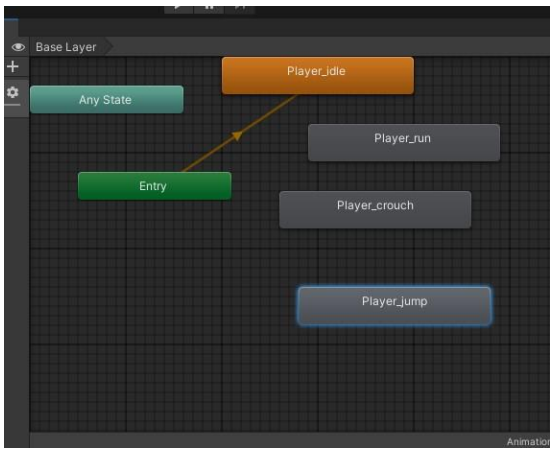
⭐ Seleccionamos las cuatro imágenes de la carpeta Sunnyland – sprites – player – idle (en 3); cambiamos sus pixeles por unidad a 16 y los arrastramos a la ventana de animación, lo cual creará una propiedad del tipo Sprite. Si pulsamos el icono Play de la ventana veremos cómo se anima el personaje. La animación por defecto irá muy rápido, ya que los frames por segundo son 60. Para mejorarlo ajustamos el valor Samples a 12 (12 frames por segundo)

⭐ Ahora pulsamos sobre el desplegable de animaciones (donde pone player_idle) y seleccionamos Create New Clip y repetimos el paso anterior para la animación correr (run).

⭐ Repetimos el paso anterior para agacharse (crouch).

⭐ Hacemos lo mismo para el caso de saltar (jump), pero en este caso ajustamos las Samples a 2 para tratar de que todos los frames se reproduzcan durante el salto (lo ralentizamos).

⭐ Si vamos a la carpeta Animations (3), veremos que se han creado las cuatro animaciones y un objeto llamado Player. Éste es el controlador de animación que decide que animación hay que reproducir y se representa con un grafo con transiciones entre estados. Vamos a hacer doble click sobre dicho objeto para acceder a la ventana Animator.



⭐ En primer lugar, vemos que el primer estado es Player_idle, lo que significa que por defecto será la animación que se reproducirá. Puedes probarlo ejecutando el juego (si activas la ventana de animator, podrás ver en qué estado se encuentra la animación).

⭐ Lo primero que vamos a hacer es crear una transición para el estado de correr. Para ello pulsamos sobre el estado idle con el botón derecho y seleccionamos **Make Transition**, que llevamos hacia el estado run.

⭐ Pulsamos sobre la transición recién creada y vemos en la parte inferior del inspector, debajo del solapamiento de animaciones, la condiciones que se deben dar para hacer esa transición (pasar de un estado a otro).

⭐ Para poder implementar dichas condiciones es necesario definir los parámetros necesarios. Para ello, vemos en la parte izquierda del animator dos pestañas: **Layers y Parameters**. En este caso, seleccionaremos **Parameters**. Pulsando el símbolo + añadimos un parámetro de tipo float llamado **Speed** para saber la velocidad del personaje.

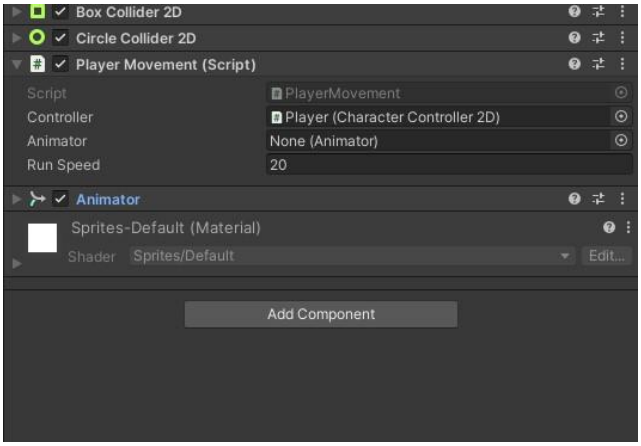
⭐ En las condiciones añadimos: **Speed greater than 0.1**. Es decir, si la velocidad es mayor que cero, se reproducirá la animación correr.
⭐ Como queremos que la transición sea inmediata, desmarcamos **Has Exit time** (que espera a que una animación termine para que empiece la otra) y en **Settings – Transition Duration a 0**.
⭐ Creamos la transición inversa (Run → Idle) con la condición inversa, es decir, Speed Less 0.1.

⭐ Ahora necesitamos pasar dicho parámetro desde nuestro script. Para ello, editamos el script de Player. Lo primero que debemos hacer es acceder, desde el Inspector, al

componente Animator que Unity ha añadido automáticamente a Player al animarlo. Para ello, añadimos:

```
Public Animator;
```

⭐ Al salvar y volver a Unity, vemos dicha propiedad en el controlador de Player y lo único que queda es arrastrar el componente Animator a dicha propiedad.



⭐ En el script, en la función Update, después de calcular la velocidad del personaje (horizontalMove), se lo pasamos a dicho parámetro, usando la función Abs, para obtener siempre el valor absoluto, aunque horizantalMove sea menor que cero (moverse a la izquierda).

```
animator.SetFloat("Speed",Mathf.Abs(horizontalMove));
```

⭐ Ejecutando el juego veremos que la animación funciona correctamente.

⭐ Ahora implementaremos el salto. Este es un estado especial, ya que podemos saltar desde cualquier otro estado. Para no complicar demasiado la máquina de estados, tenemos el estado Any State, que indica precisamente esto. Creamos una transición **Any State → Player_jump**. Para la condición creamos un parámetro **boolean isJumping**. Añadimos dicho parámetro a la condición de la transición (**isJuping is true**). En esta transición, debemos desactivar el tick **Can transition To**, para evitar que se pueda hacer la transición Player_jump → Player_jump de forma continua (no saldría del bucle), mostrando solo el primer frame de la animación del salto.

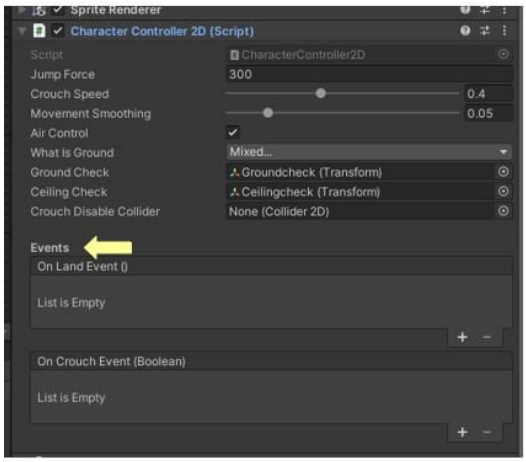
⭐ Ahora creamos dos transiciones:

- Player_Jump → Player_idle con las condiciones speed<0.1 y isJumping false.
- Player_Jump → Player_run con las condiciones speed>0.1 y isJumping false.

⭐ Ahora debemos pasar el parámetro desde el script. Para ello añadimos el siguiente código al pulsar la tecla de salto, lo cual disparará la transición e iniciará la animación del salto.

```
if(Input.GetButtonDown("Jump")) {
    jump=true;
animator.SetBool("isJumping",true);
}
```

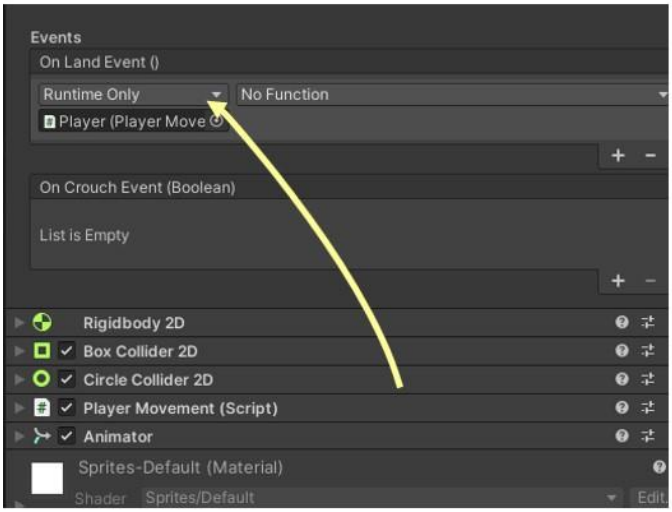
⭐ Ahora para cambiar el estado de esta variable es un poco más complejo, ya que no se puede poner en el fixedUpdate, que solo controla el inicio del salto. Para ello, vemos que nuestro controlador tiene dos eventos predefinidos (On Land Event y On Crouch Event).



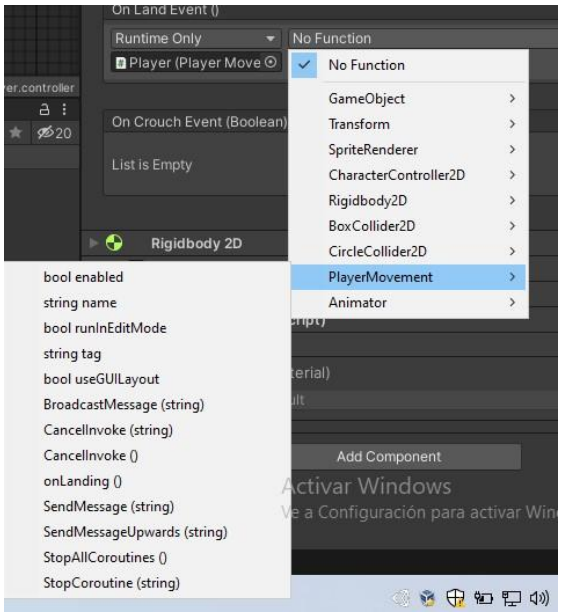
⭐ Vamos a utilizar el evento OnLand, que se disparará cuando el jugador aterrice. Para ello, creamos un método para responder a este evento, que debe ser público, para asignárselo al evento.

```
public void onLanding() {
animator.SetBool("isJumping",false);
    }
```

⭐ Entonces arrastramos el script del Player al On Land Event, para poder seleccionar el código oportuno.



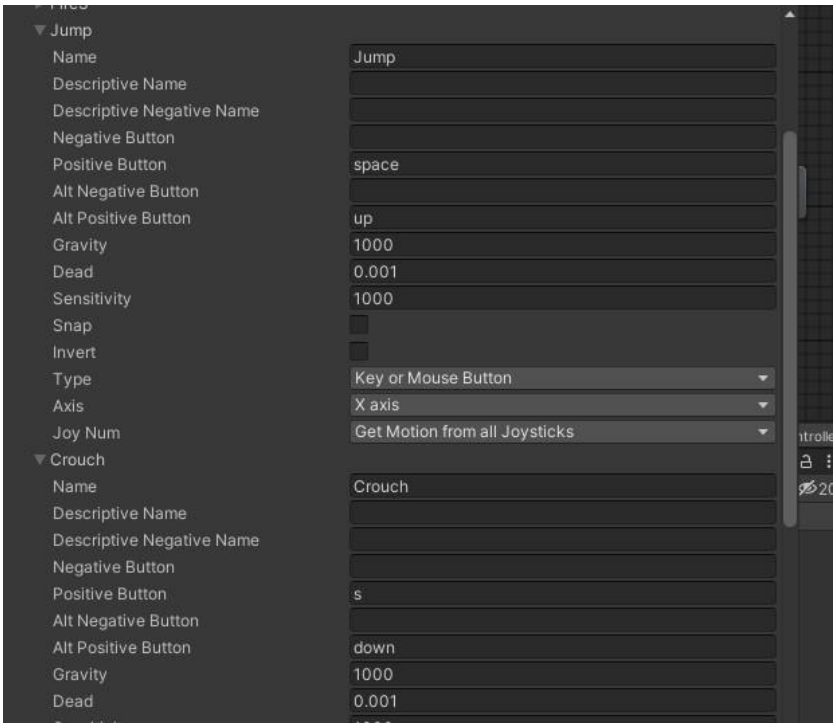
⭐ En la función a asignar al evento seleccionamos la función que acabamos de crear (OnLanding).



⭐ Ahora vamos a retroceder un poco al punto anterior para implementar la opción de **agacharse desde el principio** (implementado la lógica y la animación). Lo primero que tenemos que comprobar es si tenemos alguna tecla asignada a la acción de agacharse. Para ello, en Unity, vamos a ***Edit – Project Settings – Input Manager***. Si te fijas, en esta pantalla se definen las **constantes** que utilizamos en expresiones como:

```
horizontalMove = Input.GetAxisRaw("Horizontal");
```

⭐ Como no existe ninguno asignado a down o crouch, duplicamos el correspondiente a jump con botón derecho – **Duplicate array element** y renombramos la nueva entrada como Crouch y le asignamos la letra s (u otra que quieras). Igualmente asignamos a **Alt Positive button** como “down”. En la entrada de Jump se puede colocar el Alt Positive button como “up”.



⭐ Una vez definida la entrada, vamos a usarla en el script. Primero declaramos la variable para almacenar el estado entre funciones.

```
bool crouch = false;
```

⭐ Después comprobamos si se ha tocado la tecla correspondiente

```
if(Input.GetButtonDown("Crouch")) {
crouch = true;
}else  if(Input.GetButtonUp("Crouch")) {
crouch = false;
}
```

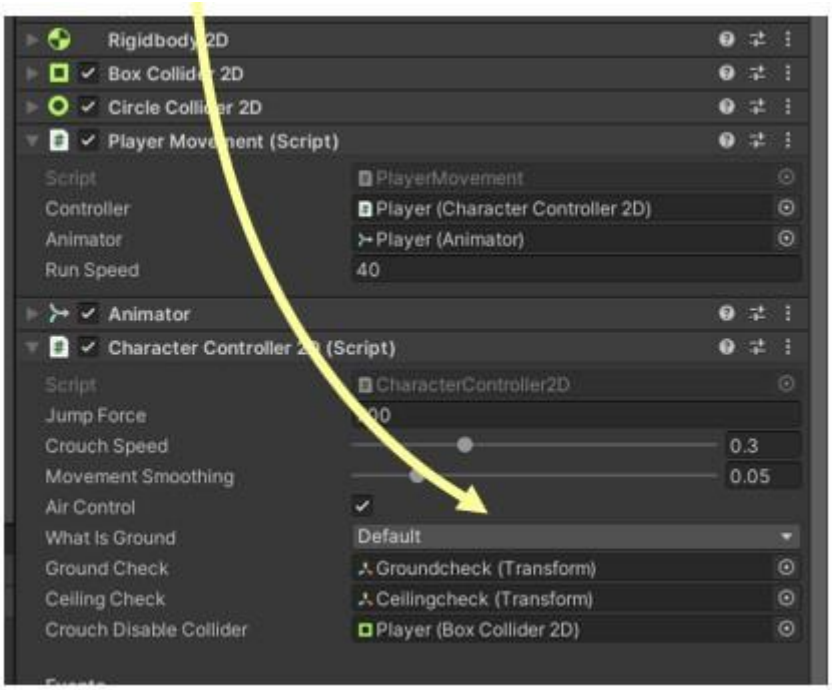
⭐ Como vemos en este caso, al contrario de con Jump, vamos a poner la variable a False cuando se detecte que el jugador suelta la tecla correspondiente. Finalmente modificamos los parámetros de la función Move.

```
controller.Move(horizontalMove*Time.fixedDeltaTime,crouch,jump);
```

⭐ Ahora cuando pulsamos la tecla asignada, el personaje se agachará y cuando se suelte, se levantará. Aunque todavía no tenemos asignada la animación (y no podemos ver un cambio en pantalla), podemos probar el efecto. Si hacemos caminar al personaje y nos agachamos, la velocidad se debería ver reducida al 40% tal como indicamos en el controlador al configurarlo (**Crouch Speed**).

⭐ Ahora vamos a añadir una plataforma baja para terminar de configurar la lógica de la acción de agacharse. Modifica el escenario para añadir la plataforma que se muestra a continuación.

⭐ Si probamos a agacharnos para pasar por debajo de la plataforma comprobaremos que no funciona. Esto se debe a que nos falta indicar al controlador qué collider (colisionador) debemos desactivar para pasar por debajo de la plataforma. En nuestro caso debemos desactivar el collider rectangular (Box Collinder) que se corresponde con la cabeza. Para ello arrastramos dicho collider al campo **Crouch Disable Collider** del controlador.



⭐ Ahora sí debería pasar por debajo de la plataforma. Si no es así, ajusta el tamaño y posición del circle collider. Si ocurre algún problema más, revisa la posición del groundcheck y del ceilingcheck.

⭐ Ahora vamos a configurar la transición dentro del grafo de Animation. Para ello, lo primero, es crear un parámetro booleano denominado **isCrouching** para indicar cuando el personaje está agachado.

⭐ Ahora creamos cuatro transiciones, para completar nuestro grafo de animación:

- Player_idle → Player_Crouch con la condición isCrouching true.

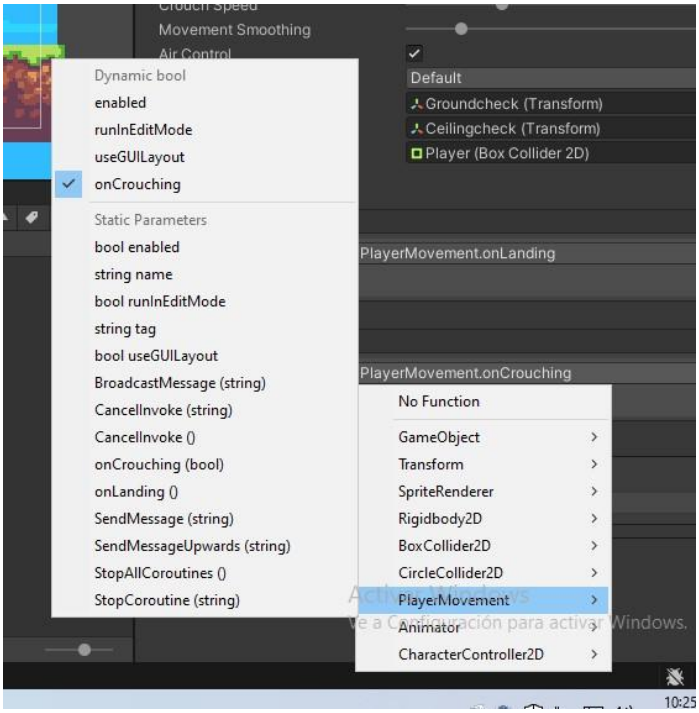
- Player_run → Player_Crouch con la condición isCrouching true.
- Player_Crouch → Player_idle con las condiciones speed<0.1 y isCrouching false.
- Player_Crouch → Player_run con las condiciones speed>0.1 y isCrouching false.

⭐ Ahora, como en los casos anteriores, tenemos que pasar el valor de estos parámetros desde el script. Pero, al contrario de Jump, en este caso el personaje no siempre puede levantarse. Si, por ejemplo, está debajo de una plataforma, no podrá levantarse, aunque soltemos el botón correspondiente. El controlador del personaje lo verifica de forma automática con el ceilingcheck, de forma que, aunque le pasemos crouch=false a la función move, si detecta que ceilingcheck colisiona con una plataforma, forzará crouch a true. Pero la animación la tenemos que controlar nosotros. Para esto vamos a utilizar un nuevo evento que le proporcionamos al controlador, para lo cual creamos la siguiente función:

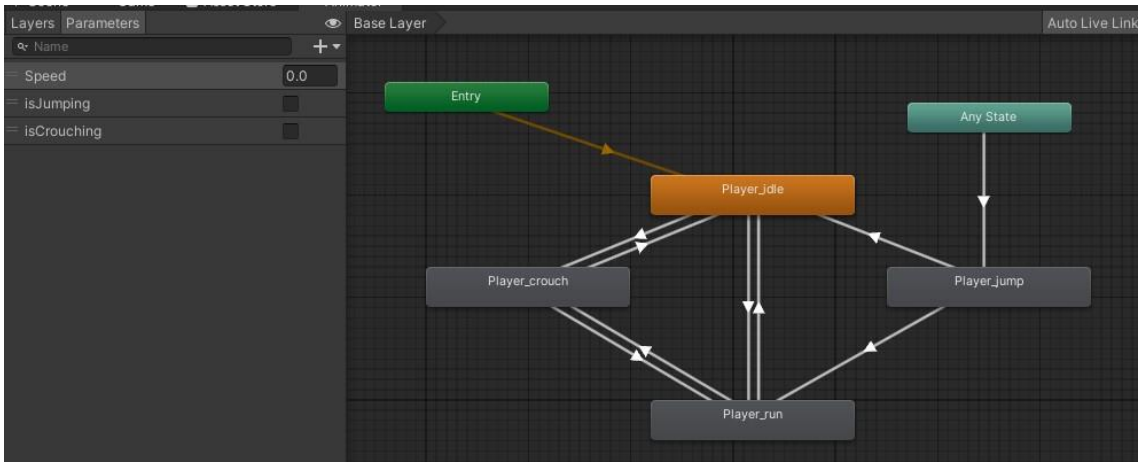
```
public void onCrouching(bool isCrouching) {
    animator.SetBool("isCrouching",isCrouching);
}
```

⭐ Este evento nos informará si el personaje está agachado a través del parámetro isCrouching, que simplemente se lo pasamos al animator.

⭐ Ahora en el Character Controller seleccionamos la función que acabamos de crear para responder al evento onCrouch.



⭐ Con esto terminamos la animación y lógica completa del personaje, que se deberá mover sin problemas por el escenario. El grafo final debería quedar así (si pulsas al Play y vas cambiando los valores de los parámetros, verás cómo va pasando de estado en estado):



6 Testing

El siguiente paso es testear el juego en nuestro dispositivo móvil. Pero antes de hacerlo, debemos asegurarnos de que todo funciona correctamente en el entorno de Unity. Como siempre, dando al icono de PLAY probaremos que todo funciona correctamente y según lo esperado (altura del salto, colocación de las plataformas, la acción de agacharse...etc.). Cuando verifiquemos que todo está correcto, pasaremos a comprobarlo en el dispositivo.

Primero utilizaremos la aplicación **Unity Remote** que se instala en el dispositivo y permite que Unity envíe la salida gráfica del editor al dispositivo y capture desde éste la entrada del usuario. Para ello necesitaremos:

- Comprobar que el JDK, el SDK y el NDK están correctamente instalados y configurados en Unity. Para ello vamos ***Edit – Preferences – External Tools***. Comprobamos que en la sección Android no existen advertencias, lo que indicará que Unity localiza todo lo necesario para compilar y testear la aplicación.
- Comprobar que has iniciado sesión en tu cuenta de Unity en el editor.
- Instalamos en nuestro dispositivo la aplicación Unity Remote:

https://play.google.com/store/apps/details?id=com.unity3d.mobileremote

- Asegúrate de que tienes las opciones de desarrollador activadas en tu dispositivo móvil y la depuración USB activada.
- Conectamos nuestro dispositivo con un cable USB de datos.
- Habilitamos el dispositivo en ***Edit – Project Settings – Editor*** y seleccionamos en la categoría Unity Remote en Device nuestro dispositivo o Any Android Device.
- Ejecutamos nuestro juego de la forma habitual, que ahora podremos ver y probar en el dispositivo móvil.
- Si no conseguimos ver el dispositivo en Device o tras seleccionar Any Android device no podemos ver el juego en el dispositivo móvil, puede ser por alguna de estas causas:
    - Conectar el dispositivo antes de iniciar Unity.

- Si en external tools no encuentra el JDK, SDK o NDK (mostrando una advertencia amarilla) aunque este instalado, desactivamos el checkbox (por ejemplo, Android SDK tools installed with Unity) y lo seleccionamos manualmente (aunque sea del mismo directorio).

- Instalar los Android USB Drivers correspondientes a nuestro dispositivo.

Si necesitas más información o resolver problemas consulta la documentación correspondiente en:
https://docs.unity3d.com/2020.2/Documentation/Manual/UnityRemote5.html

Otra manera de testar el juego, más lenta pero que simula mejor cómo será el juego final (sobre todo a nivel de rendimiento) es hacer un **Build And Run**. De esta forma Unity compilará el juego en una APK y lo desplegará directamente en nuestro dispositivo. Para ello:

- Ir a ***File – Build Settings***… y seleccionar nuestro dispositivo en Run Device.

- Pulsar sobre Build And Run y decimos donde queremos guardar nuestra APK, por ejemplo, en un directorio destinado al Release o Debug.

- Una vez seleccionado el dispositivo, las siguientes veces podemos pulsar directamente ***File – Build And Run*** o CTRL+B.

7 Controles Táctiles

Al probar el juego en nuestro dispositivo hemos podido verificar que no podemos manejar al personaje, ya que inicialmente los hemos diseñado para controlarlo con el teclado (o joystick). Por lo tanto, tendremos que habilitar los controles táctiles y para ello, de nuevo, el Unity Store viene en nuestro auxilio. Vamos a descargar una librería que nos permita implementar rápidamente una palanca en pantalla.

- En el Unity Store buscamos el Joystick Pack (Joystick Pack | Input Management | Unity Asset Store)

- Pulsa para abrirlo con Unity e importarlo al Package Manager. Desde aquí impórtalo en tu proyecto. Recuerda que una vez que lo hayas importado en el package manager ya queda disponible para todos tus futuros proyectos.

- Ahora para poder utilizarlo creamos, en la jerarquía de objetos (1) un nuevo lienzo para los controles. Pulsamos con el botón derecho y seleccionamos **UI – Canvas**

- Desde los Assets (2), en la carpeta Joystick Pack – Prefabs arrastramos desde (3) el Fixed Joystick al canvas recién creado.

- En la pestaña Game vemos el joystick, que podemos modificar a través de sus propiedades en el inspector (4).

- Ahora debemos configurar el script para que responda a los eventos de este elemento visual. Para ello añadimos, la propiedad, al script PlayerMovement:

```
public Joystick;
```

Y sustituimos la línea donde obteníamos el movimiento horizontal del personaje por:

```
horizontalMove = joystick.Horizontal * runSpeed;
```

Que nos devolverá -1 si se mueve a la izquierda y 1 si se mueve a la derecha. Ahora arrastramos el Fixed Joystick a la propiedad recién creada.

Ahora vemos que si probamos el juego el personaje responde de manera precisa al joystick virtual para la acción de correr.

Aprovecharemos el joystick también para saltar y agacharse. Para ello leeremos la posición vertical de este. Primero creamos la variable en la clase:

```
float verticalMove=0f;
```

Y ahora sustituimos nuestro script de salto y agacharse:

```
if(verticalMove >= .05f) {
jump=true;
animator.SetBool("isJumping",true);
            }
    if(verticalMove <= -.05f) {
    crouch = true;
             }else
    crouch = false;
             }
```

En este caso dejamos una zona muerta de .05 para evitar que el personaje sea incontrolable, ya que de otra forma (>=0) el mínimo movimiento de la palanca hacia arriba o abajo haría que el personaje desencadenará el movimiento. Ajusta el tamaño del joystick y la zona muerta para que el control sea el adecuado.

Si quisiésemos usar una pulsación en la pantalla para saltar podemos usar otro código.

```
    if(Input.touchCount>1) {
    Touch t=Input.GetTouch(1);
    jump=true;
    animator.SetBool("isJumping",true);
```

```
}
```

Con este código comprobamos si hay más de una pulsación (ya tenemos una con el correspondiente al joystick) y obtenemos información de esa pulsación (Input.GetTouch con el índice de la pulsación empezando desde 0). En la variable t almacenamos la información de la pulsación por si queremos detectar las coordenadas de la pulsación (que en este caso no usamos). Simplemente, si detectamos una segunda pulsación, saltamos.

De la variable t podemos extraer la posición donde se pulso, aunque necesitamos hacer una transformación entre las unidades de pantalla (t.position) en pixeles a las unidades de nuestro mundo, en unidades, obteniendo un vector con las tres coordenadas (x,y,z)

```
Vector3 touchPosition =
Camera.main.ScreenToWorldPoing(t.position);
```

Otro ejemplo del manejo de varias pulsaciones (a través del array touches del Input) puede ser (pruebalo para ver cuál es el resultado)

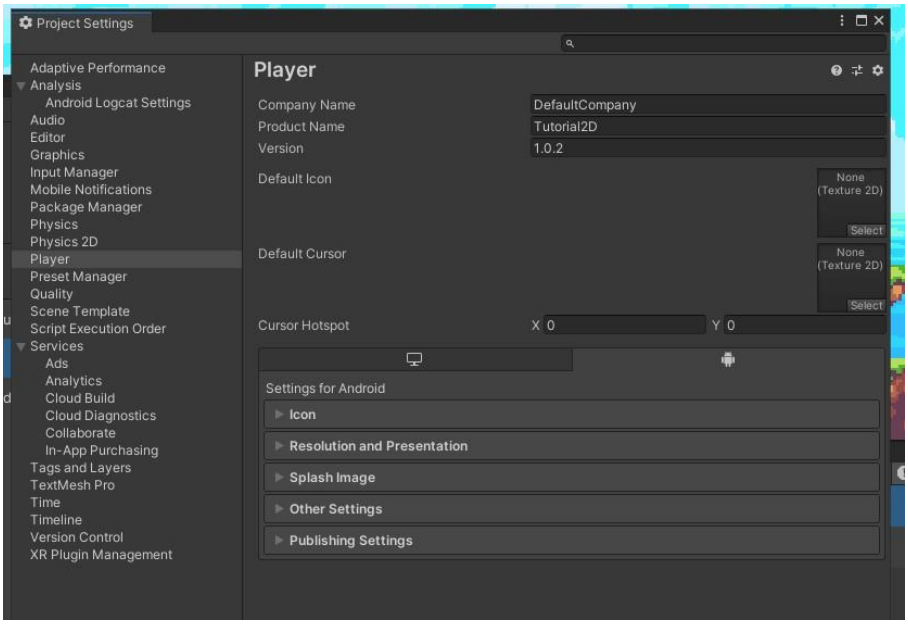
```
for(int i=0;i<Input.touchCount;i++)
{
   Vector3 touchPosition =
Camera.main.ScreenToWorldPoint(Input.touches[i].posit
ion);

Debug.DrawLine(Vector3.zero,touchPosition,Color.red);
}
```

8 Despliegue

Dado que ya tenemos el SDK, JDK y NDK correctamente y hemos podido usar el Unity Remote y Build And Run, lo único que quedaría para desplegar el juego de manera definitiva sería completar todos los detalles relacionados con el icono que se va a mostrar, pantalla de carga (splashscreen), resoluciones, compañía, versión...etc.

Todo esto se configura a través de la sección Player del Project Settings.



En **Publishing Settings** podemos configurar todo lo necesario para poder publicar nuestra aplicación en el Playstore de Android, como el keystore con el que firmaremos digitalmente nuestra aplicación para que sea validada. Pero este tema se escapa del ámbito de esta práctica y es igual para cualquier aplicación que queramos desplegar.

9 Recogiendo Ítems

Es hora de añadir algo más al juego que nuestro personaje. Para ello vamos a añadir algún objeto que recolectar. En este caso, serán cerezas.

⭐ Primero creamos un objeto con los sprites de ***Assets – Sunnyland – Sprites – items – cherry***. Podemos hacerlo igual que hicimos con el personaje principal (primero arrastrar un frame y luego crear la animación con el resto). O si prefieres probar un sistema distinto arrastra todos los frames del sprite a la jerarquía de objetos. Te solicitará un nombre para la animación (Cherry_idle, por ejemplo) que puedes guardar en la carpeta de Animations.

⭐ Acuérdate de indicar los 16 pixels por unidad y establecer un framerate de 12 frames por segundo.

⭐ Una vez creado lo colocamos en pantalla donde queramos. Si lo probamos, vemos que no hay interacción entre el personaje y el ítem. Para que “choquen” le añadimos un BoxCollider 2D a nuestra cereza. Ahora, si lo probamos, veremos que hay colisión.

⭐ Colocamos la cereza en el layer Player en el inspector (4). Esto es para que no se consideré ground (repasar los parámetros del Controller que definimos al principio) y no interactúe con él como si fuese una plataforma.

⭐ Seleccionamos la opción **Is trigger** del BoxCollider 2D de la cereza, que hace que se dispare un evento cuando se choque con el objeto, pero desactiva la física y hace que no choque de manera brusca. Esto en videojuegos, se denomina normalmente como sensores, que detectan cuando un personaje para por un sitio.

🌟 Igualmente necesitamos añadir algo que nos permita identificar con que objeto estamos chocando (¿es un enemigo o un ítem?). Se puede hacer de diversas maneras, pero lo más sencillo es hacerlo mediante Tags (etiquetas). Con la cereza seleccionada, en la parte superior del inspector (4), creamos una nueva etiqueta **"Collectable"** y se la asignamos al objeto.

🌟 Ahora ya podemos crear el script para responder al evento que dispara la colisión.

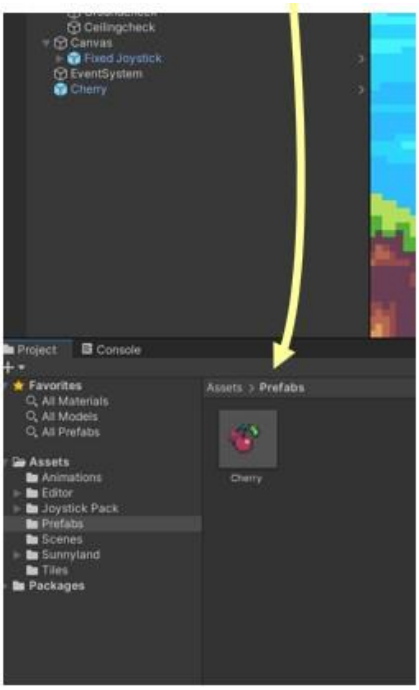
Primero creamos una variable para almacenar las cerezas recogidas en el Player Movement script. `public int cherries=0;`

🌟 Creamos la función para responder al trigger. Respondemos al evento OnTriggerEnter2D, que se disparará cuando se colisiona con un objeto de tipo trigger.

```
    private void OnTriggerEnter2D(Collider2D
collision) {
if(collision.tag=="Collectable") {
Destroy(collision.gameObject);
cherries++;
        Debug.Log(cherries);
 }
}
```

🌟 Si lo probamos, vemos que cuando pasamos por la cereza, esta desaparece y en la consola aparece un uno.

🌟 Vamos a usar el ítem creado para construir un **Prefab**, que es un objeto con propiedades inicializadas que nos permite utilizarlo en el resto del juego. Primero creamos una carpeta prefabs (o como queramos llamarla) dentro de Assets (2) y arrastramos el objeto de la jerarquía de objetos a dicha carpeta. Con esto estamos indicando que queremos usar un objeto con esas características en el resto del juego.



⭐ Ahora desde la carpeta prefabs (3) podemos poner tantas cerezas como queramos en el juego.

SunnyLand.- Muestra el contenido de la variable coins en el juego como un marcador para saber cuántas cerezas ha cogido el personaje. Pista: canvas, etiqueta de texto.

SunnyLand.- Mejora el aspecto de tu nivel haciéndolo más largo añadiendo todo lo que consideres (cuestas, pasadizos, … etc.). En el GRID añade tres TileMaps uno para decorar el fondo (background) con árboles, casas … etc. y otro para el primer plano (foreground) para hierbajos y pequeños detalles. Estos dos tilemaps solo deben servir para mejorar con el juego, pero no deben interaccionar con el personaje.

10 Cinemachine

Ahora que ya podemos movernos libremente y recoger ítems, vamos a añadir un nuevo efecto para que la cámara pueda seguir al personaje lo cual dará un efecto más profesional al juego, haciéndolo menos estático y nos permitirá hacer scroll en todas las direcciones a lo largo de la fase (por ejemplo, subir a un árbol, no solamente en el eje X). Fíjate que ahora el personaje puede salir del encuadre por la izquierda o derecha. Unity implementa las herramientas necesarias para hacer esto a través de una opción denominada **Cinemachine**. Cinemachine no viene instalado por defecto, por lo que tendremos que hacer lo siguiente:

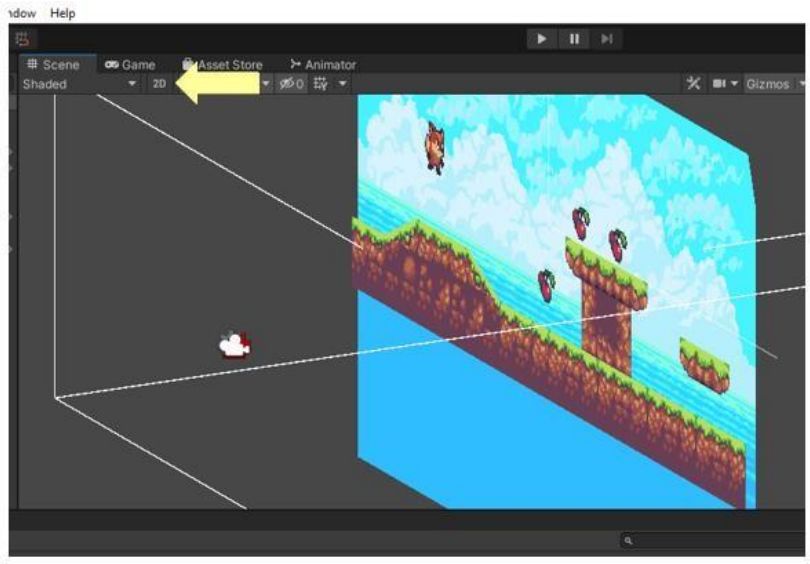
⭐ Seleccionamos ***Window – Package Manager***
⭐ En la ventana del gestor de paquetes, seleccionamos en la parte superior ***Unity Registry*** para que nos muestre todos los paquetes disponibles para instalar (y no solo los que estamos usando actualmente)

⭐ Seleccionamos Cinemachine y pulsamos Install. Cuando la descarga e instalación termine, aparecerá un nuevo menú Cinemachine.

Ahora vamos a usarlo y ver las distintas opciones.

⭐ Primero seleccionamos Cinemachine – Create 2D Camera.

⭐ Aparentemente no ha pasado nada, pero vemos que, en nuestra jerarquía de objetos, se ha añadido un objeto **OM Vcam** que es una cámara virtual que está en la misma posición que la cámara "real". Para comprobarlo pulsa sobre el botón 2D de la parte superior, que nos permite ver la escena desde cualquier eje. En imagen se muestra el botón 2D y las dos cámaras; la real (blanca) y la virtual (roja)



⭐ Ahora tenemos que indicar a la cámara virtual a que personaje tiene que seguir. Para ello arrastramos el objeto Player desde la jerarquía hasta el control **Follow** de la cámara virtual.

⭐ Con ello vemos que al pulsar Play, la cámara siempre sigue al personaje encuadrándolo, para que podamos comprobar el enfoque. Igualmente verás un punto amarillo indicando el punto de enfoque de la cámara.

⭐ Ya tenemos lo más importante hecho. Ahora es probable que tengas que retocar el nivel ya que, posiblemente, se vean partes sin escenario.

Vamos a ver más opciones de Cinemachine. Para ello despliega la sección Body de la cámara virtual.

***Dead zone**, es el área del cuadro en que cinemachine va a mantener siempre el objetivo dentro. La dead zone se identifica al ser la parte más nítida cuando se testea (no está sombreada) y que siempre envuelve al personaje.*

Para explorar las opciones, coloca X, Y, Z Dumping a cero. Se explicará posteriormente el uso. Notarás que movimiento de la cámara es más brusco, menos suave.

⭐ **Screen X e Y:** Posición de la pantalla respecto a la dead zone.. Por defecto, están en el medio (0.5). Moviendo el Y a un valor como 0.46, tendremos al personaje un poco por encima de la mitad lo que suele dar un acabado mejor.

⭐ **Dead Zone Width y Height:** Permite modificar el tamaño de la dead zone y que le permite moverse sin que la cámara le siga. Prueba distintos tamaños para ver el efecto. Veras que cuando el personaje llega al límite de la dead zona, la cámara le empieza a seguir.

⭐ **X, Y, Z Dumping:** Nos permite indicar cuanto queremos que salga el objetivo (punto amarillo) de la dead zone antes de empezar a mover la cámara. Simula el retraso que tendría un operador de cámara para reaccionar al movimiento del personaje. Como ahora lo tenemos a cero, en cuanto el marcador amarillo toca el borde de la dead zone la cámara le sigue (brusco). Si ponemos un valor como 0,1 en X e Y veremos que el marcador amarillo puede salir ligeramente de la dead zone antes de que la cámara se mueva. Esto le da suavidad al enfoque. En resumen, hace que la cámara vaya un poco retrasada respecto al movimiento del personaje.

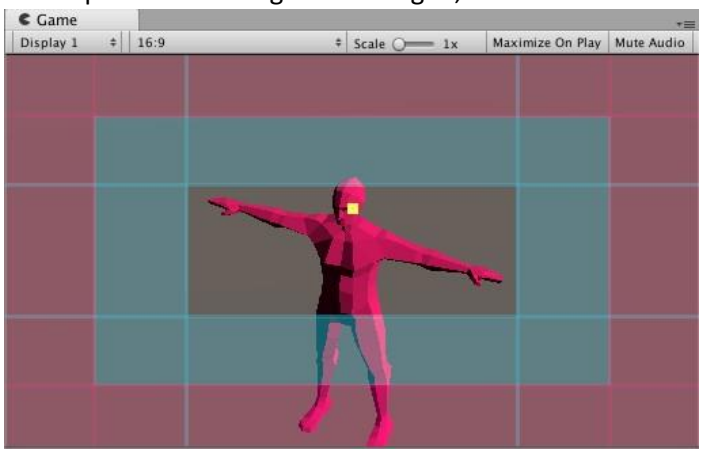
⭐ **Lookahead time:** Ahora mismo resulta difícil saber que viene por delante al personaje. Con esta opción, Unity reconoce la dirección en la que está mirando el personaje y nos permite ver adelantado entre 0 y 1 segundo. Si le ponemos un valor mayor que cero, veremos que el objetivo (donde enfoca la cámara) mira un poco más adelante del jugador, permitiendo ver lo que viene.

⭐ **Lookahead smoothing:** Nos permite seleccionar la suavidad con la que realizará el efecto anterior. Cuanto mayor lo pongamos, más le “costará” al marcador separarse del personaje.

⭐ **Lookahead Y:** Permite ignorar el efecto en el eje Y (sobre todo al saltar) ya que puede resultar incómodo para el jugador, dejándolo solo activo en el eje X.

⭐ **Soft zone:** Si el objetivo entra en esta zona (relleno azul), la cámara se reorientará para poner al personaje de nuevo en la dead zone, con la suavidad indicada en el dumping. Normalmente, solo se modifica para personajes que son extremadamente rápidos (Sonic)

Vemos las distintas partes explicadas en la siguiente imagen, a modo de resumen:



El área despejada indica la **dead zone**. El área teñida de azul indica la soft zone. La posición de las dead y soft zones indica la posición de la pantalla. El área teñida de rojo indica el área de no-paso, en la que el objetivo nunca entra. El cuadrado amarillo indica el objetivo.

11 Ampliación - Enemigos

Solo nos queda añadir algo de peligro, para ello están los enemigos. Vamos a implementar la lógica típica de unas plataformas sin disparo. Cuando chocamos lateralmente con el enemigo, morimos y cuando saltamos sobre él lo eliminamos.

⭐ Antes de nada, crea el personaje Rana y su animación Idle. Si tienes alguna duda, repasa lo visto anteriormente.

⭐ Acuérdate de añadir un Collider 2D y añadirlo a la capa Player (como en las cerezas)

⭐ Añádele una etiqueta (Tag) **Enemy**

⭐ Añadimos una dependencia al script, para usar el gestor de escenas.

```
using UnityEngine.SceneManagement;
```

⭐ Añadimos la siguiente función al script de nuestro personaje. Este evento se disparará siempre que colisionemos con otro objeto del juego (si tiene collider, claro). Lo primero es obtener el objeto con el que colisionamos a través de la propiedad collider de la Collisio2D. Una vez verificado que chocamos contra un enemigo, necesitamos conocer la dirección en la que hemos chocado, para ello usamos el método **GetContact** de la colisión normalizado. Esto nos devolverá un Vector con x=1 si chocamos por la derecha del objeto y x=-1 si chocamos por la izquierda. La coordenada Y será 1 si chocamos arriba y -1 si es abajo. En nuestro caso, obtenemos el valor absoluto y si chocamos lateralmente (derecha o izquierda) reiniciamos el nivel y si saltamos sobre el personaje lo eliminamos.

```
 void OnCollisionEnter2D(Collision2D col){
Collider2D collider=col.collider;
if(collider.tag=="Enemy") {
          Vector2 direction = col.GetContact(0).normal;
     if(Mathf.Abs(direction.x)==1)
SceneManager.LoadScene(SceneManager.GetActiveScene().name);
          if(Mathf.Abs(direction.y)==1)
Destroy(collider.gameObject);
      }
    }
```

Con esto y un poco de imaginación ya tenemos todos los ingredientes para crear un buen juego en nuestro móvil.

Ampliación: Haz que la rana salte cada cierto tiempo, incluyendo la animación y la lógica. (Pista: puedes intentar usar el mismo controlador del personaje o alguna función del RigidBody2D como AddForce)

12 Para seguir practicando

- Juego con efecto parallax lateral:
  - https://pixelnest.io/tutorials/2d-game-unity/ (lectura/impresión)
  - https://www.youtube.com/watch?v=kByEbJItcwg (video)
- Otros tutoriales - https://pixelnest.io/tutorials/