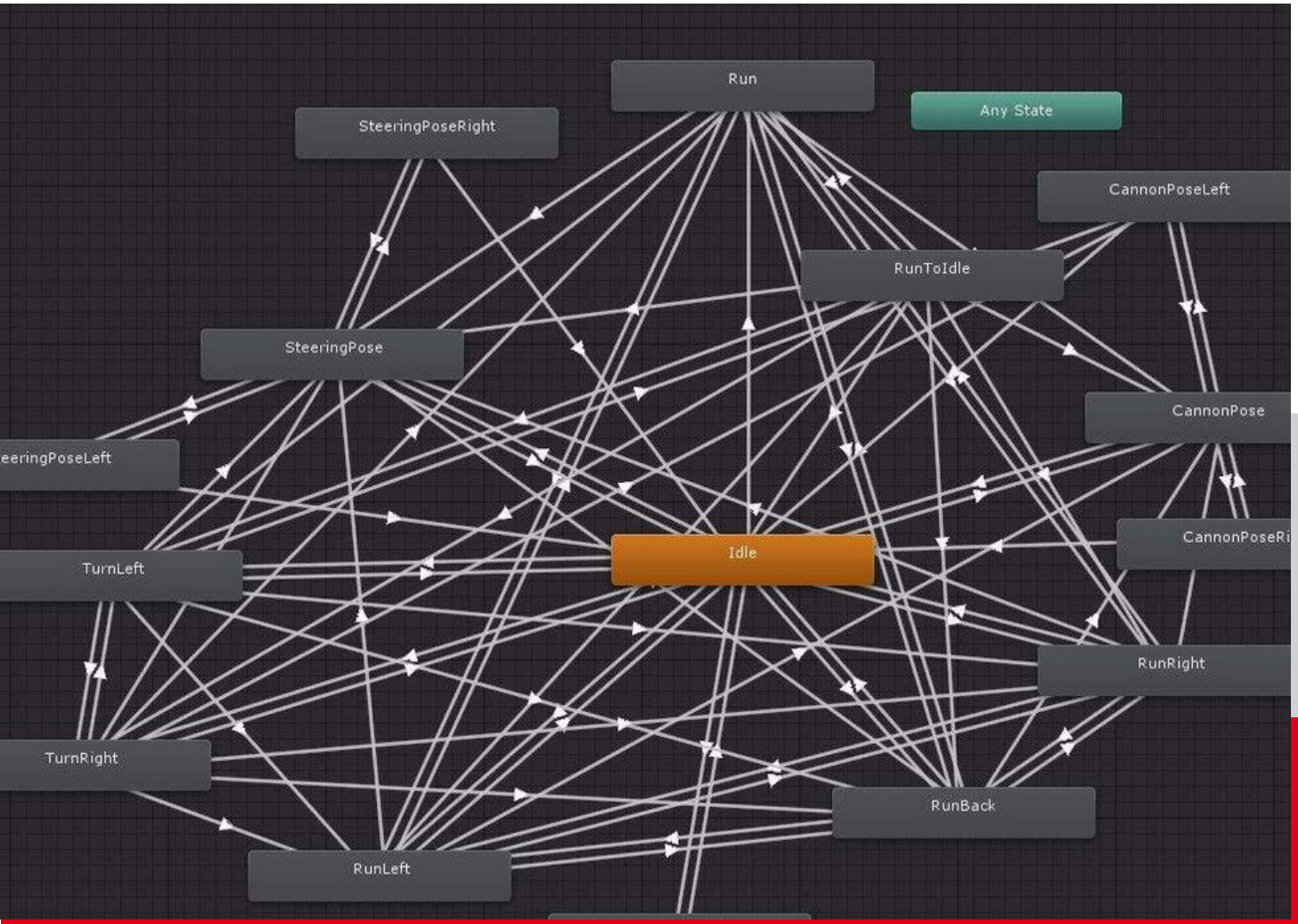
Álvaro Holguera
CIFP José Luis Garci
(Madrid)

UD 05
TRABAJANDO EN 2D

Desarrollo de Entornos
Interactivos Multidispositivo

Álvaro Holguera
CIFP José Luis Garci
(Madrid)



Parte 1
ANIMATOR & ANIMATION

Antes de ponernos a trabajar en un juego 2D, tendremos que aprender cómo funcionan estas dos herramietas de Unity

ANIMATION y ANIMATOR

Vamos a conocer las herramientas que usa Unity para controlar crear y controlar objetos animados, desde elementos estáticos que se mueven solos (como las plataformas móviles en un juego 2D) hasta las complejas animaciones de personajes.

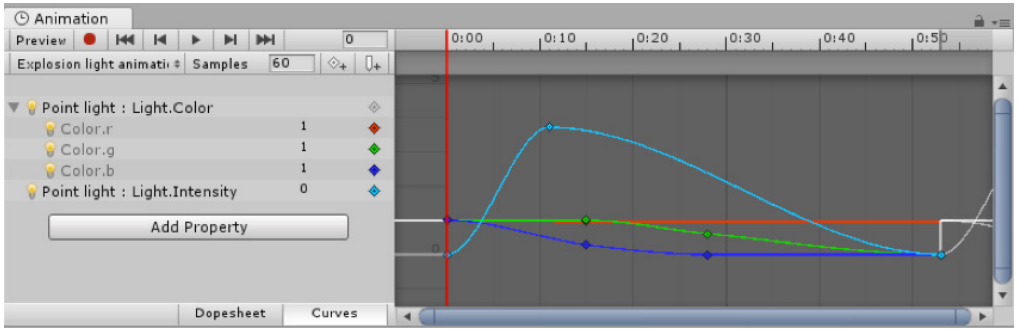
Veremos cómo funcionan las animaciones para posteriormente aplicarlo primero en la creación de un sencillo juego de plataformas y, más tarde, en cómo controlar un personaje 3D, para terminar incorporando un seguimiento con cámara mediante el paquete Cinemachine.

Unity tiene dos herramientas para crear animaciones, complementarias entre sí pero que cumplen funciones diferentes

Animation

Permite crear animaciones propias, tanto de elementos 2D como elementos 3D, manejando sus parámetros básicos (posición, escala, rotación), y en algunos casos otros no tan básicos (por ejemplo ,las luces).

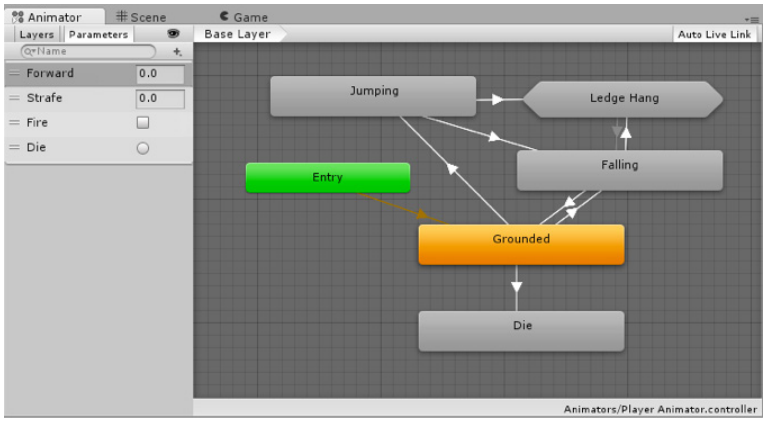
Cuenta con un completo sistema de línea de tiempo, con posibilidad de creación de fotogramas clave, interpolación y edición de curvas



Animator (ó Animator Controller)

Sistema nodal que nos permite crear transiciones entre las animaciones, tanto las creadas en Unity con el Animation, como las importadas desde programas externos (como modelos .fbx)

Este componente se añade por defecto a nuestro Game Object al crear una animación

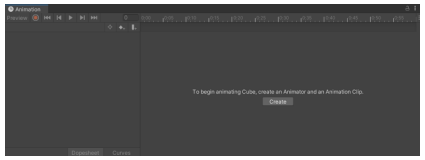


ANIMATION (parte 1)

Unity dispone de una herramienta para crear animaciones mediante fotogramas clave, tanto de elementos 2D como 3D. Esta herramienta está pensada tanto para animar sprites, como la animación 2D tradicional, como para animar las propiedades de un solo elemento (por ejemplo, una nube que pasa).

Pasos a seguir

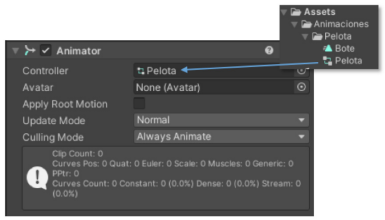
1. Abrimos el panel: Window->Animation->animation



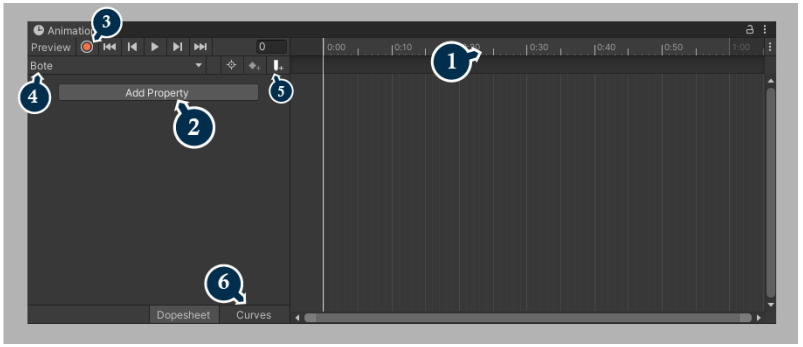
2. Deberemos seleccionarun GameObject de nuestra escena para animar (o si ya tiene alguna animación asociada, verla en el panel).

3. Si no tiene una animación nos pedirá crearla, en el botón "Create" del panel Animation, eligiendo donde guardarla en nuestro proyecto de Unity. Es importante llevar un orden, ten en cuenta que un objeto puede tener asociadas muchas animaciones.

IMPORTANTE: como verás, se ha creado automáticamente un componente "Animator" en el objeto, y en la propiedad "Controller" verás que se ha creado también un "Animator Controller", con el mismo nombre y que contrendrá todas las animaciones, como veremos más adelante.

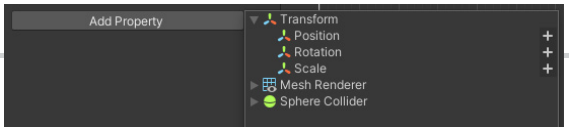


Ahora se mostrará el panel de Animation con la animación que acabamos de crear:



- Línea de tiempo con los fotogramas clave (1). Podemos hacer zoom sobre ella con la rueda del ratón, crear fotogramas clave, grabar cualquier modificación que se produzca, o incluso añadir eventos que podremos vincular a métodos públicos en un script vinculado. También podemos movernos con los botones de reproducción o ir a un fotograma específico.

- Si pulsamos el botón de "Add Property" (2) nos permitirá modificar los parámetros de sus componentes, especialmente el de Transform, pulsando en el icono "+"



- Otra opción muy cómoda es pulsar el botón de "grabar" (3), y a partir de ese momento cualquier propiedad que toquemos quedará registrada como Key Frame. No te olvides de desactivarlo luego.

- Verás un desplegable con la animación en la que estamos trabajando (4), y si tiene varias te permitirá elegir otra. Un mismo objeto puede tener tantas como queramos. Incluso se reproducirán la jerarquía del objeto en la escena.

- Para añadir una nueva, despliégalo y pulsa en "Create new clip" que deberás guardar en la carpeta correspondiente.

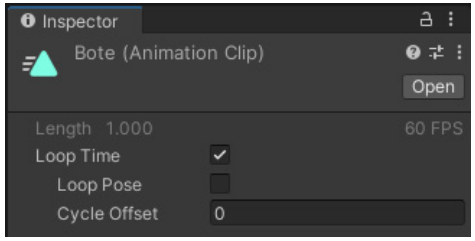
- Junto al botón para añadir fotogramas claves verás la opción de añadir "eventos" (5) que permite vincular fotogramas de la animación con funciones en scripts.

- Podemos cambiar el modo de animación a curvas (6), para crear interpolados más suavizados (por defecto todas las animaciones tienen algún tipo de suavizado).

ANIMATION (parte 2)

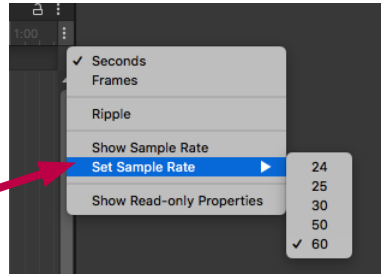
Animation clip

Si seleccionamos la animación en el proyecto, veremos en su panel del Inspector varias opciones que debemos conocer:



- **Loop Time:** la animación se ejecutará en bucle. Nota: si le damos a play en el panel de Animation se verá en bucle siempre, pero si ejecutamos el juego se ejecutará en bucle solo si está marcada esta casilla
- **Loop Pose**: si hemos creado una animación repitiendo el primero y el último fotograma, debemos marcar esta casilla para que el último y el primero se fundan. También sirve para que animaciones 3D se ejecuten en el sitio, sin moverse. Lo veremos más adelante.
- **Cycle Offset**: si queremos que la animación empiece en un fotograma distinta.

NOTA: si nos fijamos, por defecto las animaciones tienen una velocidad de 60fps. Si queremos cambiarlo deberemos usar el menú de hamburguesa que hay en el panel de Animation:



AYUDAS para crear la animación:

- Si se hace doble click en la propiedad, se crean automáticamente todos los keyframes con los valores actuales
- Si seleccionamos todos los keyframes, podemos mover los últimos ó los primeros y el resto se redistribuyen proporcionalmente.
- Podemos copiar y pegar keyframes, por ejemplo para crear una pausa entre dos movimientos.

IMPORTANTE: si creamos una animación modificando los parámetros de "Position" de nuestro objeto, recuerda que las coordenadas que das son respecto a su elemento padre, y si no tiene, respecto al escenario 3D. Por eso, si indicas que en el fotograma 1 esté en unas coordenadas, irá a ellas aunque muevas el objeto de posición. Si quieres luego poder mover el elemento animado por el escenario, hazlo hijo de un padre y así las coordenadas serán relativas a ese padre, que será el que movamos.

Eventos

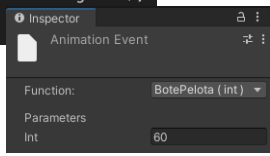
Podemos ampliar las posibilidades de las animaciones mediante la creación de "eventos", los cuales permiten llamar directamente a funciones que se encuentran en el código vinculado a ese Game Object,

Para añadirlo, solo tenemos que poner el puntero en el momento exacto de la línea de tiempo, y pulsar el botón de "Evento".

Tras añadirlo, o al volver a seleccionarlo, aparecerá en el inspector un desplegable que nos permite seleccionar la función entre las disponibles en el script asociado a ese Game Object, así como los parámetros que podemos pasar a esa función.

En el siguiente ejemplo, se ha creado un script para este objeto, y se ha llamado a través del evento (como se puede ver, se pueden pasar variables a la función):

```
void BotePelota(int fotograma)
{
    print("Se acaba de ejecutar un evento en el frame: " + fotograma);
}
```

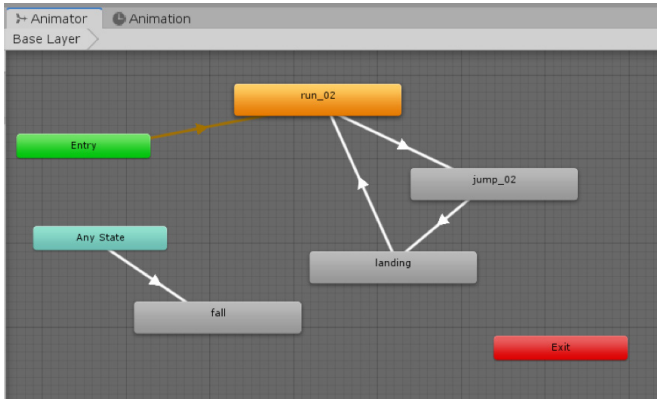


Si nos ponemos encima de una marca de evento, veremos un Tooltip con la función a la que llama.

ANIMATOR CONTROLLER (parte 1)

El controlador de las animaciones (Animator Controller) es la herramienta de Unity para gestionar las transiciones entre unas animaciones y otras.

Una vez creadas las animaciones, mediante el panel de Animator podremos asignar transiciones desde una animación a otra.



Podemos arrastrar las animaciones directamente desde el panel de proyecto al panel de Animator y aparecerá como un nodo nuevo.

TRUCO: para moverte por el panel, puedes hacer Zoom con la rueda del ratón, y desplazarlo pulsando Alt+arrastrar.

Por defecto se ejecutará una, la que está unida al estado "Entry" y aparece de naranja, pero podemos cambiarla por otra pulsando con el botón derecho > Set as Layer Default State.

Para crear una transición en un estado, pulsamos con el botón derecho en un nodo > Make Transition, y la arrastramos al nuevo nodo. Aparecerá una conexión con una flecha indicando la dirección.

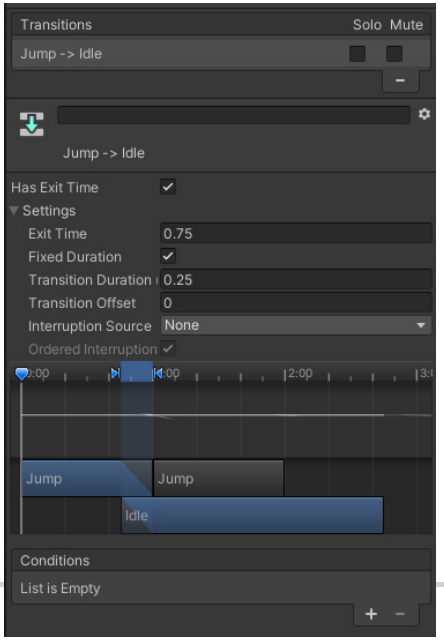
Estas transiciones las podemos crear desde un estado a otro (por ejemplo, desde correr a saltar y vuelta a correr). Incluso podemos crear dos transiciones de un estado a otro, y que cada una dependa de factores independientes (en este caso, aparecen dos triángulos en lugar de uno en la unión entre nodos).

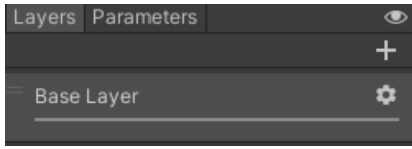
Pero también podemos crear transiciones desde cualquier estado al que nosotros queramos, desde el "Any state", de forma que esté siempre disponible.

Si seleccionamos la transición aparecerá en azul y podremos configurar algunas cuestiones en el panel de inspector:

Además de poder indicar cuánto dura la transición, hay un parámetro importante: si desmarcamos la casilla "Hass Exit Time" la transición no esperará a que termine la animación para ejecutar la transición, algo que a menudo nos es conveniente (no apetece tener que terminar el ciclo de caminado para empezar un salto, por ejemplo).

A la izquierda del panel podemos organizar las animaciones por capas, algo habitual para animar diferentes partes del cuerpo. Al añadir una nueva capa, podemos crear todo un nuevo set de animaciones





ANIMATOR CONTROLLER (parte 2)

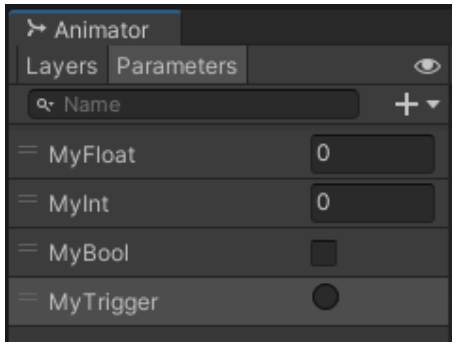
Lanzar las transiciones entre estados

Al ejecutar el juego, veremos en el panel cómo se reproducen las animaciones y las transiciones, pero antes, tenemos que determinar qué provoca que se pase de una animación a otra (si no lo hacemos, las transiciones saltan automáticamente)

Para eso se usan los parámetros y las condiciones:

- **Parámetros**: en la pestaña de "Parameters" podemos añadir tantos como queramos, y pueden ser de tipo Float, Int, Bool y Trigger. Es importante nombrarlos correctamente, para luego acceder a ellos por código.

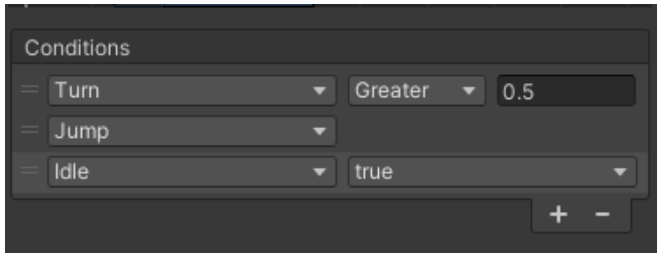
- La diferencia entre un parámetro Trigger de uno Bool, es que se "dispara" una vez y vuelve a su estado inicial, muy útil para cosas como saltos.



- **Condiciones**: en el inspector de la transición, podemos añadir las condiciones que se deben cumplir para que esa transición se lance, las cuales van vinculadas a los parámetros creados previamente.

Dependiendo del tipo de parámetro, nos permitirá asignar una condición (un valor mayor o menor qué, que sea true o false, o que directamente se lance en el caso de los trigger).

En este ejemplo, se tienen que dar estas tres condiciones para que se produzca la transición (si queremos que se ejecute si se da una u otra condición, deberemos crear varias transiciones):



Si ejecutamos el juego, podemos comprobar si las transiciones se ejecutan cambiando los parámetros en tiempo real. Algo muy útil para verificar en tiempo real si se está ejecutando bien el Controlador de Animaciones.

ANIMATOR CONTROLLER (parte 3)

Las animaciones y los parámetros añadidos están dentro del Componente "Animator" creado en nuestro GameObject. Eso significa que podemos acceder a él mediante código, y de esta forma cambiar las animaciones.

Podemos hacerlo de dos formas: creando una variable pública o serializada de tipo Animator que contendrá nuestro componente y arrastramos el Game Object que contiene el componente:

```
public Animator anim
```

O bien, si el componente lo tiene el mismo Game Object que contiene el script, accediendo a él en el método Start (después de crear la variable Animator, que ahora puede ser privada):

```
anim = GetComponent<Animator>();
```

A partir de ese momento, podemos acceder a todos los parámetros del componente, incluyendo las variables que determinan la ejecución de las transiciones, mediante los métodos públicos de esta clase. Algunos ejemplos:

Para lanzar un parámetro de tipo trigger llamado "jump":

```
anim.SetTrigger("jump");
```

Para modificar el parámetro de un booleano llamado "isGrounded":

```
anim.SetBool("isGrounded",true);
```

Para dar un valor a un parámetro float llamado "giro", le daremos el valor que tiene el eje de nuestro mando ("AxisX")

```
anim.SetFloat("giro", AxisX);
```

Obtener los datos de la animación actual

Una vez tenemos la variable de tipo Animator, y le hemos adjudicado el componente correspondiente, podemos no solo establecer los valores de sus parámetros, sino también obtenerlos.

Igual que usamos los métodos "Set…" podemos usar el método Animator.Get… para obtener el valor de uno de los parámetros. Ejemplo:

```
bool isCrouched = animator.GetBool("Crouch");
```

También podemos obtener el estado actual de cada uno de los parámetros de la animación, e incluso sus valores concretos en el caso de los "float". Podemos:

- Obenter la información de sus parámetros mediante

```
GetCurrentAnimatorStateInfo(0).
nombreDelParametro;
```

- Comprobar en qué estado estamos en ese momento mediante

```
GetCurrentAnimatorStateInfo(0).
IsName("nombre");
```

Controlar una animación mediante código

Podemos acceder a cualquier animación (Animation) y a sus correspondientes clips, para ello usaremos la una variable "Animation"

Si sabemos el nombre del clip de animación, podemos llamarlo dentro del array de esta forma:

```
VariableAnimation["nombreDelClip"];
```

Una vez hemos accedido al clip de animación, a través del AnimationState podemos llevar a cabo múltiples operaciones: reproducirlo, pararlo, cambiar su velocidad (-1 es hacia atrás, 1 hacia adelante y 0 detenerlo), etc.

```
anim["Walk"].time = 0.0f;
```

NOTA: al acceder al componente Animation de nuestro GameObject, nos devolverá un array. Eso significa que podemos hacer un bucle por los elementos del array,

Álvaro Holguera
CIFP José Luis Garci
(Madrid)

Trabajando en 2D

Ahora que ya sabemos controlar animaciones, vamos a aplicarlo a un juego 2D

TRABAJANDO EN 2D

Vamos a practicar lo aprendido, y a aprender cosas nuevas, creando un sencillo juego de plataformas 2D. Para ello, aprenderemos algunas cosas básicas en este tipo de juegos.

Proyecto en 3D vs. 2D

Cuando comenzamos un nuevo proyecto, debemos indicar si será en 3D ó 2D

En cualquier caso, siempre podemos cambiar esa configuración una vez creado el proyecto, en Edit>Project Settings>Editor

La diferencia entre trabajar en uno o en otro sistema afecta a cómo se interpretan algunos elementos (las imágenes en modo 2D son interpretadas como sprites por defecto, no como texturas), los controles de la cámara y los Game Objects (bloqueA el movimiento y escalado en el eje Z, así como rotar la cámara) o a la iluminación. Puedes comprobar todas las diferencias aquí

NOTA: Es bueno recordar que tenemos una herramienta para escalar en 2D, muy útil para trabajar con sprites, aunque el proyecto sea 3D, la Rect Tool.

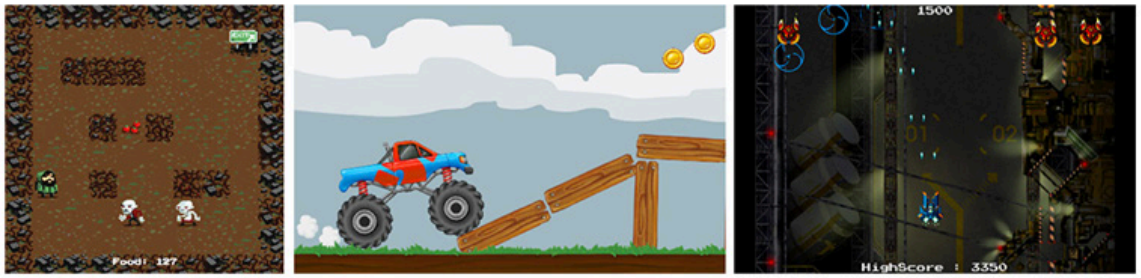
Elegir un modo u otro depende de nuestro proyecto:

- Juegos en 3D y en vista ortográfica, así como juegos en 2D con geometrías 3D, deberán usar el editor 3D
- Juegos en 2D, que utilizan básicamente sprites para crear el escenario, deberán usar el editor 2D, incluso cuando se crea profundidad mediante una cámara en perspectiva

Full 3D

Orthographic 3D

Full 2D

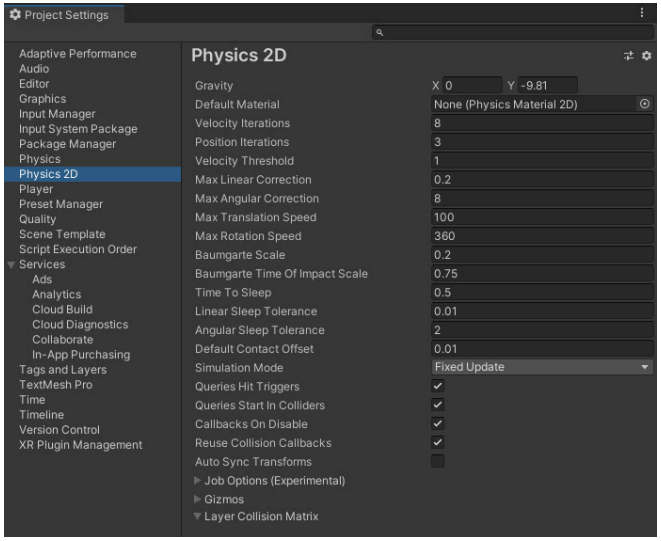


FÍSICAS EN 2D

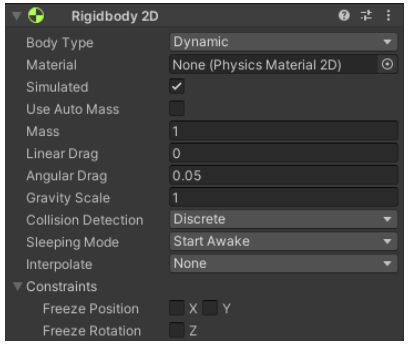
El motor de físicas 2D de Unity funciona de forma muy similar al de 3D, de hecho, tanto los elementos como los scripts usados para uno suelen ser iguales para el otros, solo que añadiendo "2D" al final (RigidBody2D, BoxCollider2D, OnTriggerEnter2D, etc.)

IMPORTANTE: Aunque podemos mezclar físicas 2D y 3D en una misma escena, no son compatibles entre sí, y un elemento 2D no puede colisionar con un elemento 3D.

Igual que las físicas normales, las 2D podemos configurarlas en "Edit>Project Settings>Physics 2D". Las opciones son prácticamente las mismas que en 3D, incluyendo la interacción entre capas para evitar colisiones entre ellas.



Al igual que en 3D, es necesario añadir un "Rigid Body 2D" a nuestro Sprite para añadir físicas y colisiones (al menos uno de los dos elementos que colisionan tienen que tener un Rigid Body, y siempre se aconseja que sea el que se mueve, y ambos deben tener colisionadores 2D).



El componente para 2D es prácticamente igual que para 3D, con algunas variaciones, la más importante es la gravedad: en lugar de un checkbox para activarla, aquí nos encontramos un parámetro llamado "Gravity Scale", que puede ir desde 0 (no le afecta), hasta 1 (se le aplica la gravedad normal de la escena), o incluso valores superiores de forma que el Sprite es "más pesado".

Otro parámetro exclusivo es que en el apartado de "constrains", podemos evitar que el objeto gire en el eje Z, algo muy recomendable para evitar que se gire en profundidad.

Colisiones 2D

Al igual que en 3D, podemos añadir componentes de colisión a nuestros Sprites 2D. Los llamados "Colliders2D":

- **Circle / Box / Capsule Colliders**: añaden formas geométricas.
- **Polygon Collider**: Formas complejas, y funciona similar al "mesh collider". Creará una forma que se adapta a las zonas opacas de nuestra imagen. Útil en elementos estáticos.

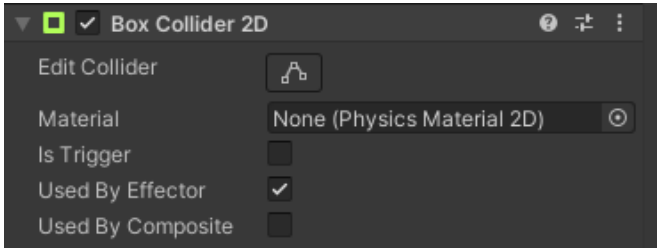
Podemos modificar la forma haciendo click en los vértices creados, o creando nuevos al hacer click, o borrando si hacemos Ctrl+click

- **Edge Collider**: permite crear desde simples líneas hasta formas complejas (creando y moviendo puntos, mediante Ctrl + Shift). SE usa para crear colisionadores simples en un solo lado (por ejemplo un suelo o una pared), ya que permite crear formas abiertas.
- **Composite Collider:** permite sumar las zonas marcadas por un Box Collider o Polygon Collider (ambos tienen un checkbox llamado "Used By Composite" para que lo sume)

EFECTORES 2D

Los efectores 2D (2D Effectors) son componentes que nos permiten añadir fuerzas a nuestros objetos 2D

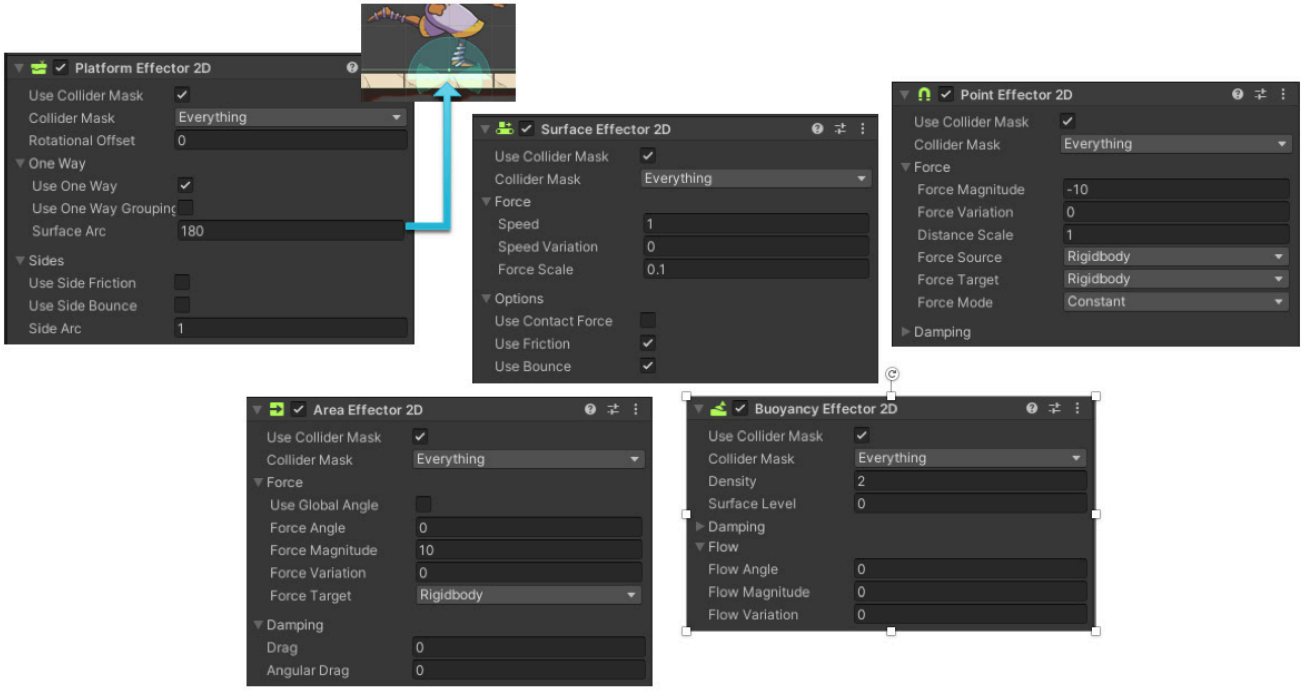
Para que pueda aplicarse, el Sprite debe tener un Rigid Body 2D, y al menos un Collider 2D con la casilla de "Used By Effector" aplicada (si no existe ninguno, saltará una alerta).



Existen 5 tipos de efectores:

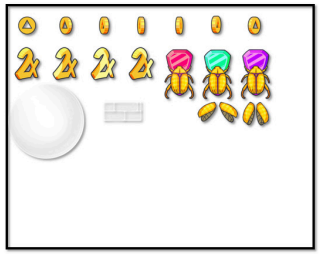
- Platform Effector 2D: pensados para crear efectos propios de plataformas, por ejemplo que colisionen solo por un lado (One Way), uno o todos los colisionadores del objeto que colisiona (Use One Way Group), así como el lado tomado como colisionador (Surface Arc)
- Surface Effector: permite añadir una velocidad tangencial al objeto que entre en contacto con él (speed), con variación aleatoria (Speed Variation), o incluso que se aplique al punto de contacto provocando giros en el objeto contactado
- Point Effector 2D*: aplica fuerzas de atracción o repulsión sobre ese punto.
- Area Effector 2D*: aplica una fuerza en una dirección y sobre el área indicada por el collider.
- Buoyancy Effector 2D*: simula el efecto de flotabilidad y fluidez por ejemplo sobre un fluido de un objeto sobre un área.

IMPORTANTE: Algunos requieren que el Collider usado para el Effector tenga la casilla de "Is Trigger" activada.



TRABAJANDO CON SPRITES

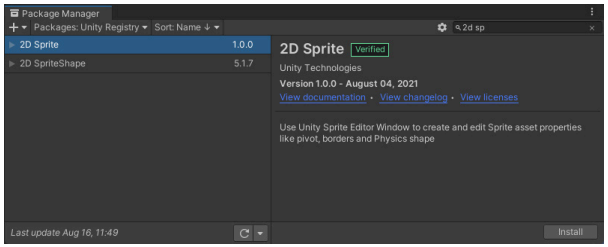
En juegos 2D es obligatorio trabajar con Sprites, elementos 2D que pueden presentarse de forma aislada o animada. En cualquier caso, lo mejor es importar hojas de sprites, en lugar de imágenes independientes. Un ejemplo de hojas de sprites:



Para los siguientes ejercicios de práctica, usaremos el paquete gratuito de "Endless Runner" disponible en la Asset Store. Adquiérelo, descárgalo e impórtalo a tu proyecto.

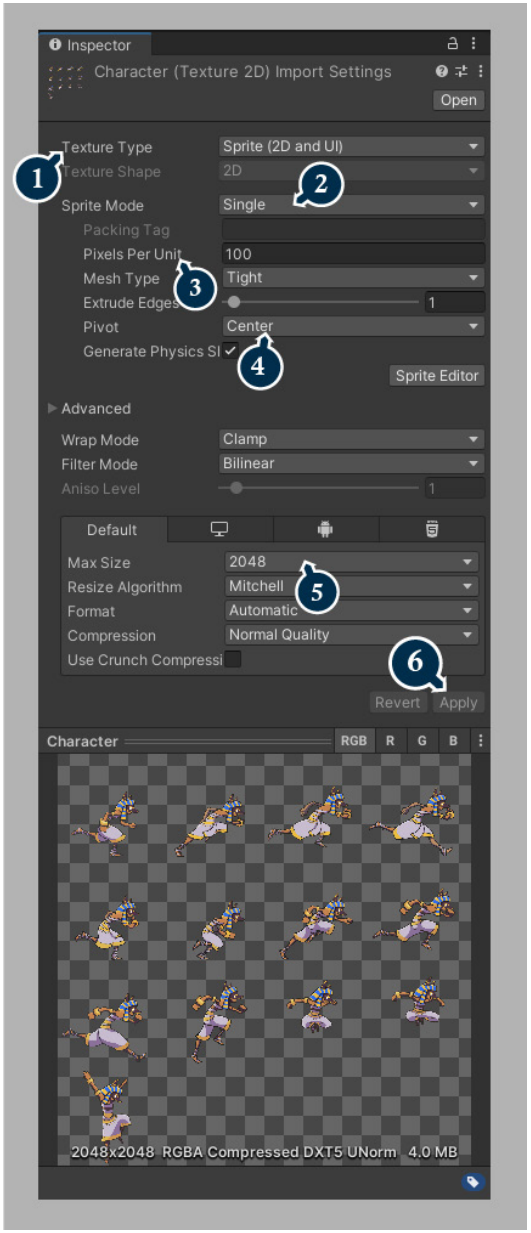
Un Sprite es un elemento 2D que podemos importar a nuestro proyecto como imagen, ya sea para crear elementos 2D o como parte del User Interface (UI).

IMPORTANTE: para poder editarlo, debemos instalarnos el paquete "2d Editor"



Una vez importada una imagen a Unity, podemos cambiar sus parámetros en el inspector

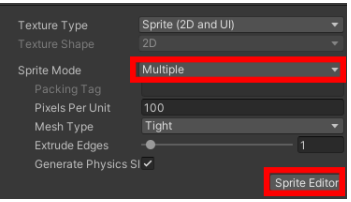
- Lo primero que tenemos que hacer (1) es indicar que es un tipo de textura "Sprite (2D and UI)". Si estamos en un proyecto 2D, lo tendrá configurado así por defecto.
- Sprite Mode (2): si es una imagen sencilla, lo dejaremos en "simple", pero si es una hoja de Sprites lo pondremos en "Multiple"
- Pixels Per Unit (3): configura el tamaño relativo de la imagen con respecto a las unidades de Unity: una imagen de 200x200px. ocupará en este caso 2 unidades de Unity (= 2 mts.)
- Pivot (4): punto de anclaje de la imagen (si la configuramos como múltiple, ese punto lo pondremos a cada imagen de la hoja de sprites por separado)
- Podemos indicar un tamaño máximo para la imagen (5), para ahorrar espacio en la compilación. Si es necesario, podemos hacerlo independiente para diferentes sistemas, es decir, para diferentes "builds".
- Una vez ajustado, debemos pulsar el botón de "Apply" (6). En la parte inferior vereos una previsualización de la imagen y nos dirá el tamaño que ocupará en la compilación.



EDITOR DE SPRITES

Una hoja de sprites nos permite incluir varias imágenes en un solo archivo, ya sea para crear animaciones o para aumentar la eficiencia del juego (es mejor importar una sola y "trocearla" que importar varias imágenes por separado). A continuación, veremos cómo crear múltiples imágenes independientes desde una hoja de sprites, mediante el "Sprite Editor".

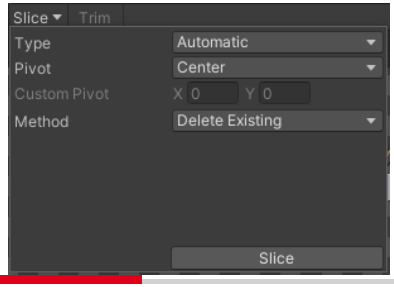
En el inspector seleccionamos como tipo de Sprite "multiple", y pulsamos en el botón de "Sprite Editor" (antes debemos dar a "Apply" para que se guarden los cambios).



NOTA: si has importado sprites de la Asset Store y ya están editados, puedes resetearlos borrando el archivo .meta que aparecerá junto al archivo del proyecto y con el mismo nombre.

Se nos abrirá la ventana para editar los Sprites:

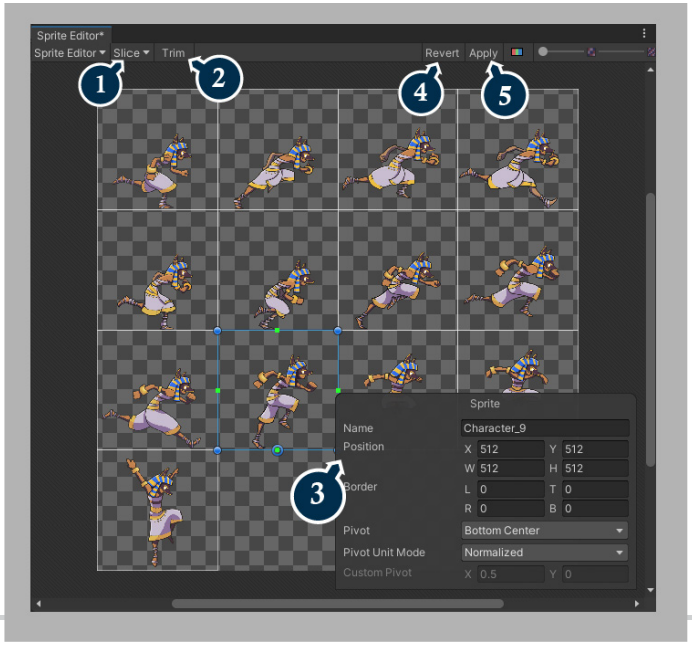
En la opción de Slice (1) se nos permite varias opciones para recortar:



Type:

- **Automática**: se crean las celdas en base a las zonas transparentes (útil para sprites desordenados)
- **Grid By Cell Size**: si conocemos el tamaño de nuestra cuadrícula, lo podemos indicar (así como el offset si existe)
- **Grid By Cell Count**: indicamos el nº de filas y columnas que tiene la hoja.

IMPORTANTE: Debemos el elegir el punto de anclaje de cada celda (podemos incluso configurarlo manualmente según sus coordenadas).



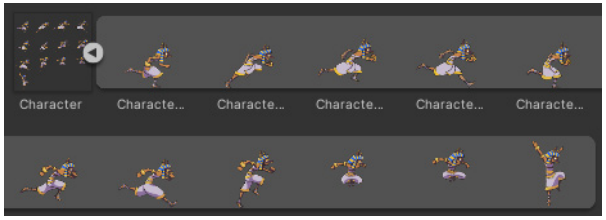
Si la hoja ya está editada y queremos conservar las ediciones podemos indicarlo en la opción de "Method".

Al pulsar en "Slice" podremos ver el resultado. Podemos incluso editarlo manualmente seleccionando los "trozos recortados" (3). Si necesitamos verlo con detalle, podemos hacer Zoom con la rueda del ratón y con Ctrl+Click y Allt+Click movernos por el panel (a veces necesitamos ajustar el pixel)

El botón de "Trim" (2) permite ajustar la zona de recorte a las partes no transparentes de la imagen.

Si después de ejecutar un "recorte" no estamos contentos, podemos pulsar en "Revert" (4)

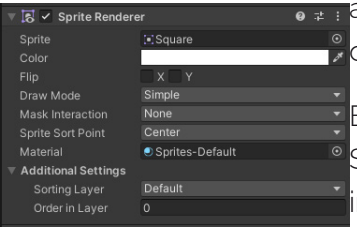
Al pinchar en "Apply" (5), se crearán tantas imágenes virtuales como cuadrículas tiene la hoja, cada una con un nombre asignado que podremos ver desplegando el Sprite original en el panel de proyecto de Unity.



CAPAS DE SPRITES

Si no disponemos de una imagen todavía, podemos usar un sprite a modo de "placeholder" que luego podremos sustituir por la imagen definitiva. Tenemos varias formas para elegir:

El componente "Sprite Renderer" se añade por defecto a los sprites, y funciona de forma silimar al "mesh renderer" de los objetos 3D.



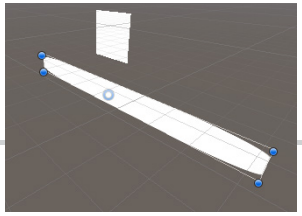
En la opción de "Sprite" del Sprite Renderer ubicaremos la imagen (sprite)

Nos permite otras opciones, como teñir la imagen, voltearla en un eje, cambiar las propiedades del canal Alpha o incluso modificar el material.

NOTA: todos los parámetros pueden cambiarse mediante código. Por ejemplo, voltear un sprite.

Por defecto los sprites tienen el material "Sprite-Default", que no recibe la iluminación de escena, pero podemos asignarle otros

Podemos escalarlo para cambiar su forma original, usando la herramienta de escalado 2D (rect tool)



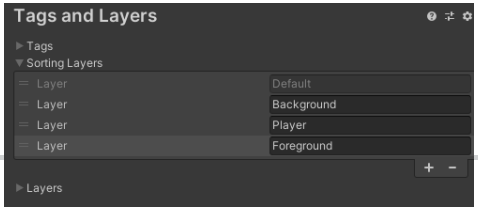
Sorting layers

Algo muy importante cuando trabajamos con Sprites en un entorno 2D es organizarlos en capas, para determinar cómo se ven.

Una opción para organizar los sprites es moverlos en el eje Z. Si la cámara es ortográfica, no tendrá la sensación de que se alejan, pero sí que determinará qué sprites se superponen a otros. Pero cuando se trabaja con un gran nº de elementos, este método es poco recomendable.

El parámetro Sorting Layer, dentro de Additional Settings, nos permitirá organizar los sprites en caso de coincidir en su posición del eje Z

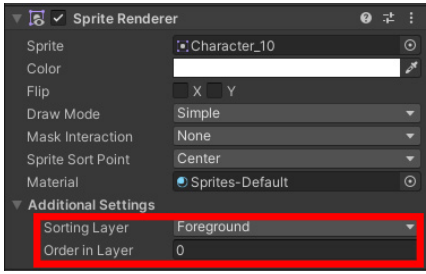
En el panel de capas (Edit > Project Settings > Tags and Layers) puedes añadir, quitar o modificar el orden de las capas que vas a usar (en este caso, hemos añadido 3 capas para distribuir los elementos de nuestro juego):



IMPORTANTE: como en la escena, el orden de renderizado de las capas es de arriba abajo, por lo que las capas que están encima aparecerán detrás de las que están debajo.

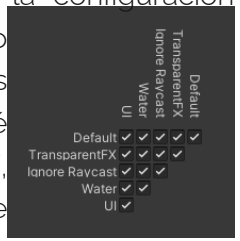
Al igual que los tags, es recomendable tener pensado desde el principio cuántas capas se van a usar en nuestro proyecto.

A continuación, en el componente Sprite Renderer de nuestro Sprite, podemos asignarle la capa a la que pertenece.



Como es habitual tener varios sprites en una misma capa, podemos incluso crear jerarquías mediante el parámetro "Order in Layer", donde los valores más bajos se colocarán detrás.

NOTA: Además de las Sorting Layers, están las capas clásicas, que permite organizar los elementos de nuestra escena para determinar la interacción entre ellas: en la configuración de físicas de nuestro proyecto (Edit>ProjectSettings>Physics 2D) podemos configurar qué capas colisionan entre sí, algo muy útil en un juego de plataformas.

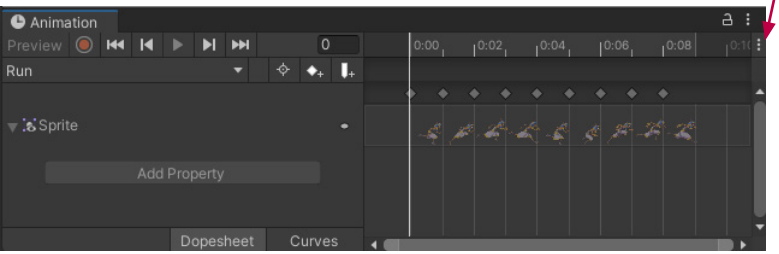


ANIMANDO EL SPRITE

Vamos a ver una forma rápida de aplicar lo que hemos visto anteriormente sobre la herramienta Animation aplicada a los sprites que acabamos de editar

Si arrastramos varias imágenes (de las obtenidas de recortar el sprite, o imágenes independientes) a la ventana de jerarquía o directamente a la escena, se creará automáticamente un Sprite con una animación vinculada (se abrirá la ventana del explorador de archivos para elegir la ubicación para guardarla).

Otro método es incorporar un sprite a la escena, añadirle un componente Animator, y arrastrar todos los sprites a la línea de tiempo del panel con la primera animación. El resultado es el mismo.

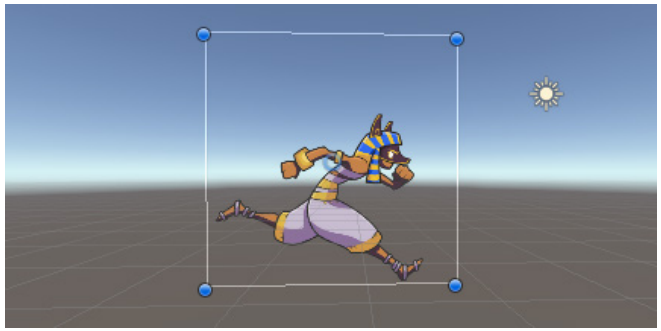


Como vimos en anteriores páginas, se ha creado un Animator Controller vinculado a este sprite de la escena, y una animación con el nombre que le hayamos dado. A partir de ahora, podemos añadir nuevas animaciones (Add new clip) y arrastrar nuevos sprites.

Se crea un fotograma con cada imagen, y auna velocidad de fotogramas por segundo predefinida, algo que rara vez es lo adecuado. Ajusta los tiempos moviendo todos los fotogramas de forma proporcional (seleccionándolos todos), o bien modificando el nº de fotogramas por segundo.

NOTA: para ajustar los fps manualmente en la animación, puedes activar la opción en el menú de amburguesa.

A medida que añadimos nuevas animaciones, veremos que ahora aparece el nuevo estado en la ventana de Animator, y si damos al play a la animación con el Sprite seleccionado en la escena veremos el resultado:



PRACTICA

Crea un Sprite animado con el personaje del material importado del Endless Runner. Verás que su hoja de sprites incluye fotogramas tanto para correr como para saltar.

Elige los fotogramas adecuados y crea esas animaciones. Ajusta los tiempos para que sea creíble la animación. Recuerda: la de correr va en bucle, pero la de saltar no.

TRUCOS (parte 1)

A continuación veremos una serie de trucos y ejemplos que nos pueden venir muy bien creando un juego 2D.

Mover al personaje mediante físicas

Puedes usar el parámetro velocity perteneciente al RigidBody2D para desplazar al personaje, aplicando un movimiento lateral, determinado por el valor del joystick (desplX) y una velocidad (maxSpeed).

Asegúrate de que las fuerzas verticales permanecen intactas, para no interrumpir un salto:

```
RigidBody2D rb = getComponent<RigidBody2D>();
rb.velocity = new Vector2(desplX * maxSpeed, rb.velocity.y);
```

Plataformas móviles

Cuando un objeto cae por físicas sobre otro que está animado, a diferencia del mundo real no queda "agarrado" a él ni a su inercia. Una forma de resolver eso es hacer que el robot al caer sobre una plataforma que se mueve pase a ser "hijo" suya, y si deja de tocarla que deje de ser hijo. Eso se logra con el atributo Transform.parent. Ejemplo para poner en un OnCollisionEnter():

```
if(collision.gameObject.tag == "PlataformaMovil")
{
     transform.parent = collision.gameObject.transform;
}
```

Y si deja de tocarlas, deberá pasar a no tener padre:

```
transform.parent = null;
```

Detectar si estamos tocando suelo

Aunque tiene algunos defectos, usar el detector de colisiones para comprobar si estamos tocando el suelo, y de esa forma actuar sobre los parámetros del Animator, o incluso para evitar doble saltos.

En el ejemplo que se muestra, detecta si ha tocado o ha dejado de tocar un objeto con tag "Plataforma", y en base a eso pone el valor a la booleana "isGrounded".

```
//Detectamos si hay suelo
Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.tag == "Plataforma")
    {
        isGrounded = true;
    }

}

Mensaje de Unity | 0 referencias
private void OnCollisionExit2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Plataforma")
    {
        isGrounded = false;
    }

}
```

Otro método es usar el RayCast. Esto nos permite incluso detectar si hay suelo a una distancia mínima desde varios puntos del sprite, no en todo el colisionador como el ejemplo anterior. En el siguiente ejemplo lanza un "rayo" de 20 cmts. hacia abajo desde el Vector3 de nuestra posición, pero podemos desplazarlo:

```
float alcance = 0.2f;
Vector3 origen = transform.position;
RaycastHit2D hit = Physics2D.Raycast(origen, Vector2.down,
alcance);
if (hit.collider != null) {
        isGrounded = true;
}
```

TRUCOS (parte 2)

"Voltear" un personaje 2D

Algo muy habitual es conseguir que el sprite mire hacia un lado o hacia otro. Cómo hacerlo:

Tendremos una booleana que determina si estamos mirando a la derecha.

Utilizando el eje horizontal, comprobamos hacia dónde nos movemos y hacia dónde estamos girando, y si hay que girarse, ejecutamos el método que hemos llamado "flip"

En el método Flip, cambiamos el valor de la booleana (así evitamos que se ejecute más veces) y modificamos la escala de nuestro Sprite para que en el eje X valga menos 1 (es decir, que se voltee)

Para ello usamos el método "localScale" de Transform (fíjate que no se limita a poner un valor de -1 en la escala horizontal, sino que comprueba primero a qué escala estamos para evitar redimensionamientos).

Deberemos llamar al método"directionDetect" en el método Update, que comprobará si estamos mirando a donde debemos, y si no, ejecuta el método "Flip()" que nos voltea. Para esto crearemos una booleana llamada "mirandoDerecha" que al empezar el juego es true.

```
//Detectar si estamos mirando al lado correcto
0 referencias
void directionDetect()
{
    //Si nos movemos a la derecha y estamos mirando a la izquierda, volteamos
    if (desplH > 0 && !mirandoDerecha)
    {
        Flip();
    }
    // En caso contrario, si nos movemos a la izquierda y miramos a la derecha
    else if (desplH < 0 && mirandoDerecha)
    {
        // giramos
        Flip();
    }
}

2 referencias
void Flip()
{
    //Cambiamos el valor de la booleana, poniendo su valor contrario
    mirandoDerecha = !mirandoDerecha;

    //Creamos un Vector3 que es igual al de nuestra escala actual
    Vector3 theScale = transform.localScale;
    theScale.x *= -1;
    transform.localScale = theScale;
}
```

Apuntar un Sprite hacia otro

Aunque parezca sencillo, ya que en un mundo 3D con el método LookAt() funciona bastante bien, hacer que un Sprite se oriente hacia otro en todo momento es complicado. Por ejemplo, para las torretas.

Podemos usar este código (donde "player" es una variable Transform que contendrá a nuestro personaje, y que podemos modificar levemente para que "apunte a la cabeza"):

```
Vector3 dir = player.position - transform.position;
float angle = Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg;
transform.rotation = Quaternion.AngleAxis(angle, Vector3.
forward);
```

Detectar caídas

A menudo es muy práctico dividir un salto en dos animaciones: una que marca el ascenso, y otra con la caída. Eso nos permite pasar de "subir" a "bajar" no cuando lo dicte la animación, sino cuando realmente está cayendo el personaje. Esto nos permite también hacer que pase de caminar a caer directamente, por ejemplo si caeo por un precipicio.

Cuando estamos trabajando con físicas y RigidBody ("rb"), podemos saber si estamos cayendo por la velocidad de nuestro personaje en el eje Y, y en ese caso actuar sobre el parámetro que hace saltar esa animación:

```
if (rb.velocity.y < -0.25f)
{
    animator.SetBool("Falling", true);
}
```

TRUCOS (parte 3)

Cambiar el colisionador dependiendo de nuestro estado

Eso nos permitiría, por ejemplo, hacer más pequeño la caja del BoxCollider2D cuando el personaje está agachado, o estrecharlo si está saltando para que entre por ciertas rendijas.

Para hacerlo, tenemos que acceder al componente BoxCollider2D, y mediante una función que consulta el estado de la booleana que hace que se agache, y comprobando otra booleana (isCrouched) que evita que se ejecute constantemente, modificamos los parámetros "size" y "offset" del Box Collider.

Debemos crear las variables:

```
BoxCollider2D bc;
bool isCrouched;
```

En el método Start darle ssu valor:

```
bc = GetComponent<BoxCollider2D>();
isCrouched = false;
```

Ahora ya podemos crear un método al que llamaremos en el Update:

```
void CambiarCollider()
{
    //Si hemos puesto el parámetro agachado y no lo estamos
    if (anim.GetBool("Crouch") && !isCrouched)
    {
        bc.offset = new Vector2(0.35f, 0.9f);

        bc.size = new Vector2(1.7f, 1.5f);
        //Decimos que está agachado
        isCrouched = true;
    }

    //Si hemos puesto que no estamos agachado y lo estamos
    if (!anim.GetBool("Crouch") && isCrouched)
    {
        bc.offset = new Vector2(0f, 1.2f);
        bc.size = new Vector2(1f, 2.1f);
        isCrouched = false;
    }
}
```

Saltar en una dirección específica

En un ejemplo anterior hemos visto cómo hacer saltar al personaje en vertical, ahora vermos cómo hacerlo saltar hacia un lado en función de lo que digamos con el mando.

NOTA: si queremos que siempre salte en la misma dirección, el valor de desplazamiento horizontal debería ser fijo.

Para ello, obtendremos en dos variables float los ejes del mando ("desplX" y "desplV"), y también configuraremos la fuerza del salto ("jumpForce") así como la fuerza de desplazamiento ("desplForce").

Este método lo ejecutaremos al pulsar el botón de salto, y solo se ejecutará si estamos tocando el suelo (es importante cambiar esa variable a false una vez dejemos de tocar el suelo):

```
0 referencias
void SaltoDirigido()
{
    //Creamos un Vector que indica hacia donde se salta, con la fuerza de la variable impulso
    Vector2 direction = new Vector2(desplH * desplForce, desplV * jumpForce);
    //Saltamos solo si el sprite está tocando el suelo
    if (isGrounded == true)
    {
        //Aplicamos la fuerza según el Vector2 creado
        //Le tenemos que indicar que es un tipo de fuerza de impulso en el 2º parámetro
        rb.AddForce(direction, ForceMode2D.Impulse);
    }
}
```