PMDM06.- Desarrollo de juegos 2D.

1.- INTRODUCCION.

Aunque muchas veces se piensa que Unity solo sirve para el desarrollo de juegos 3D, Unity ofrece la posibilidad de realizar juegos 2D. Para ello, Unity utiliza una vista ortográfica, es decir, una vista en la que todo se proyecto respetando su tamaño contra un plano. A pesar de ser un juego 2D se pueden utilizar diferentes capas para simular efectos como un fondo. Para conseguir que un juego 2D tenga un fondo se puede utilizar el efecto Parallax.

1.1.- GRÁFICOS 2D.

Los objetos gráficos en 2D se conocen como Sprites. Los sprites son esencialmente texturas estándar, pero existen técnicas especiales para combinar y administrar texturas de sprites para conseguir una mayor eficiencia y comodidad durante el desarrollo. Haciendo uso de Atlas se puede conseguir tener una cantidad de elevada de sprites y conseguir una mayor eficiencia. Unity proporciona un editor de Sprite incorporado que permitirle extraer los sprites de un Atlas. Esto le permite editar varias imágenes de componentes dentro de una única textura en su editor de imágenes. Podría usar esto, por ejemplo, para mantener los brazos, las piernas y el cuerpo de un personaje como elementos separados dentro de una imagen.

Los sprites se representan con un Sprite Renderer. En la jerarquía se puede crear directamente un Sprite o se puede crear un objeto vacío y añadirle un Sprite Renderer.

1.2.- COMPONENTES 2D.

Se debe prestar atención cuando se esté desarrollando en 2D ya que los componentes que nos ofrece Unity en muchos casos están diseñados para juegos 3D. Los componentes para 2D tienen un sufijo que indica que son los adecuados para 2D. Si queremos utilizar un Rigidbody tendremos que utilizar un Rigidbody2D. En otros casos los componentes para 2D y 3D tienen nombres diferentes.

2.- COMPONENTES PARA DESARROLLO DE JUEGOS 2D.

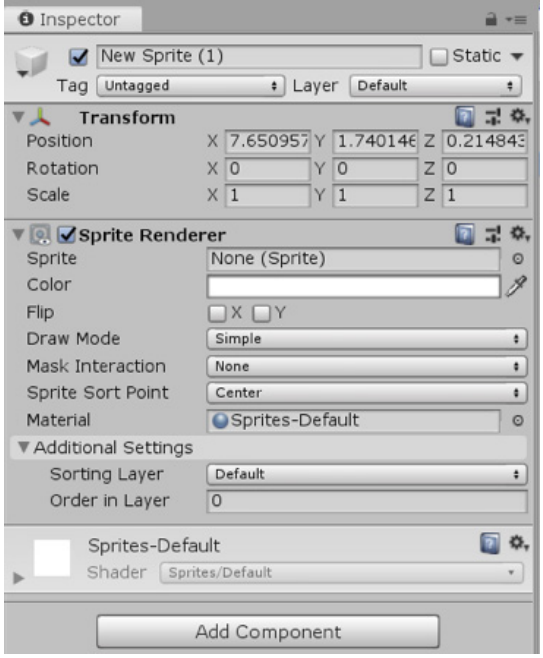
2.1.- SPRITES.

Tal y como se mencionó en la introducción los Sprites son objetos 2D. Gracias a los Sprites se puede añadir gráficos a un juego 2D. En el curso vamos a hacer uso Sprites que ya están creados y no vamos a tener que crear nuevos gráficos. Unity ofrece las siguientes herramientas para crear y editar Sprites:

- Sprite Creator. Permite crear gráficos. Gracias a esta herramienta no es necesario a que los artistas gráficos terminen su trabajo para poder iniciar el proyecto. Los programados pueden crear unos gráficos básicos que serán reemplazados por los definitivos.
- Sprite Editor. Permite editar gráficos. Una de las tareas más comunes es la extracción de gráficos que se encuentran en un Atlas.

Cuando creamos un Sprite en Unity veremos los siguiente en el inspector:

Panel inspector

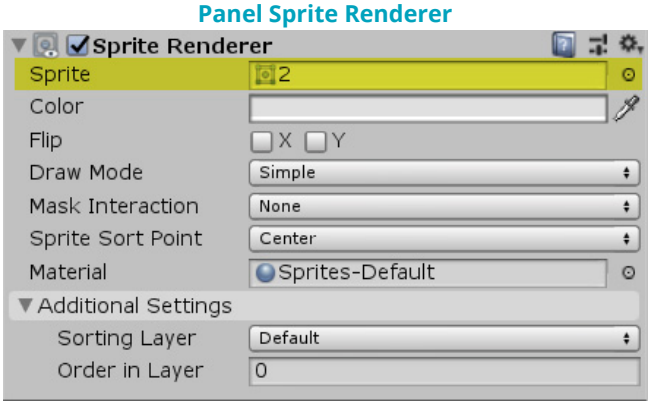


BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

El primero componente es el Transform del componente. Este componente nos permite fijar la posición, rotación y escala del Sprite. Las propiedades se pueden fijar directamente en el editor o se pueden modificar en tiempo de ejecución.

El segundo componte es el Sprite Renderer. Este componente es que se va a encargar de realizar el renderizado. Para asignar el gráfico 2D utilizaremos la propiedad Sprite.

Panel Sprite Renderer



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

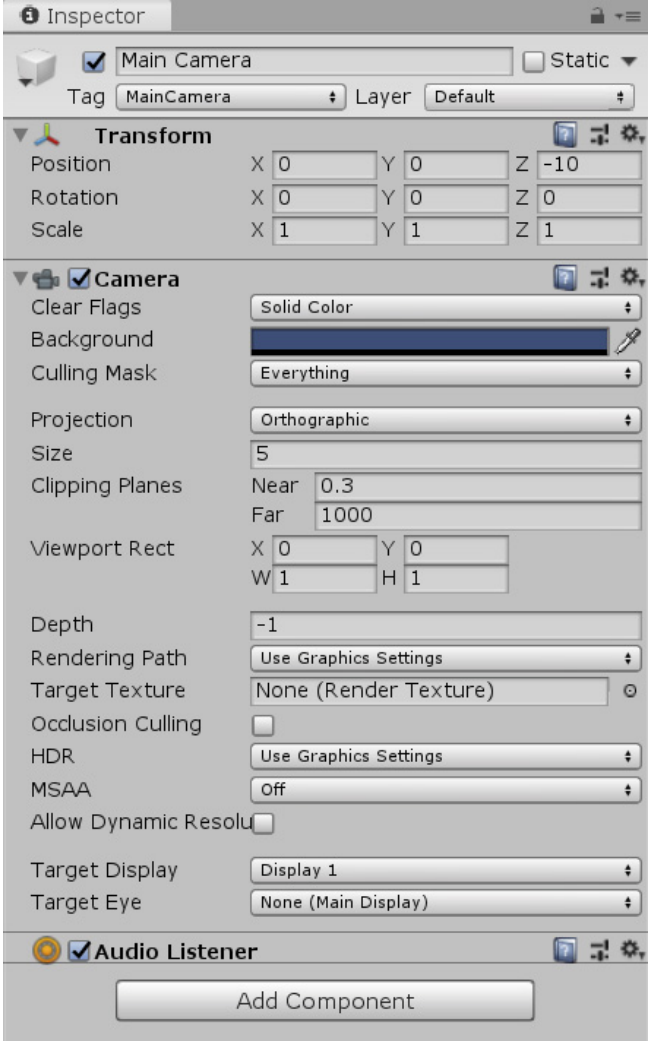
Si fuera necesario realizar alguna rotación se puede utilizar la propiedad Flip.

Si queremos añadir un script que se encargue de controlar el comportamiento del Sprite lo haremos añadiendo un componente.

2.2.- CÁMARA.

Las cámaras son los dispositivos que capturan y muestran el mundo al jugador. Al personalizar y manipular las cámaras, podemos hacer que la presentación de nuestro juego sea única. En un juego podemos tener una cantidad ilimitada de cámaras en una escena, aunque en nuestro caso solo utilizaremos una cámara.. Al solo tener una cámara esta será la main camera tal y como podemos ver en la imagen que nos muestra el inspector de Unity.

Panel inspector

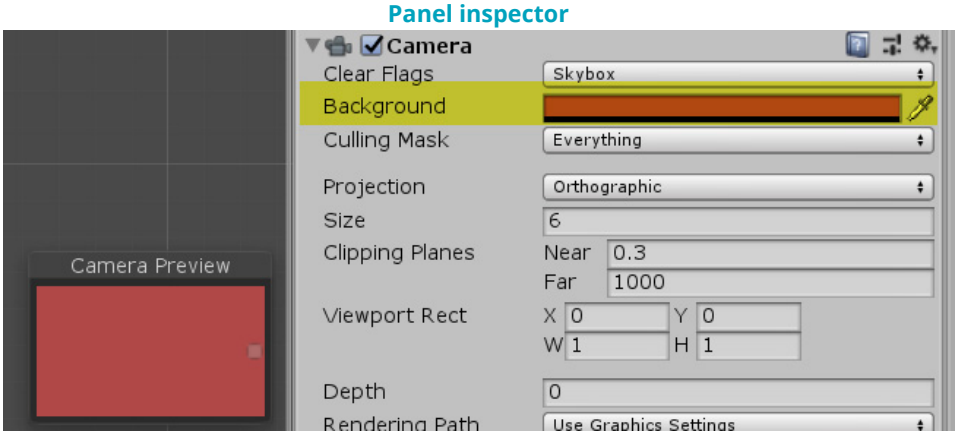


BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Ya se ha mencionado previamente que la cámara en un juego en dos dimensiones utilizar una proyección ortográfica. Esto se puede observar en la propiedad Projection. A continuación, vamos a explicar alguna de las propiedades de las cámaras:

- **Background.** Esta propiedad sirve para indicar el color que debe utilizarse a modo de fondo. Este color se aplica una a aquellos pixeles que lo necesiten una vez se han dibujado todos los objetos de la escena. Se debe tener en cuenta que si se utiliza un SkyBox no se utiliza. En un juego en 2D no suele ser común utilizar un SkyBox, por lo tanto, el fondo se suele aplicar si es necesario. En las siguientes imágenes se puede ver la diferencia entre que genera el uso de un color de fondo azul y uno rojo.

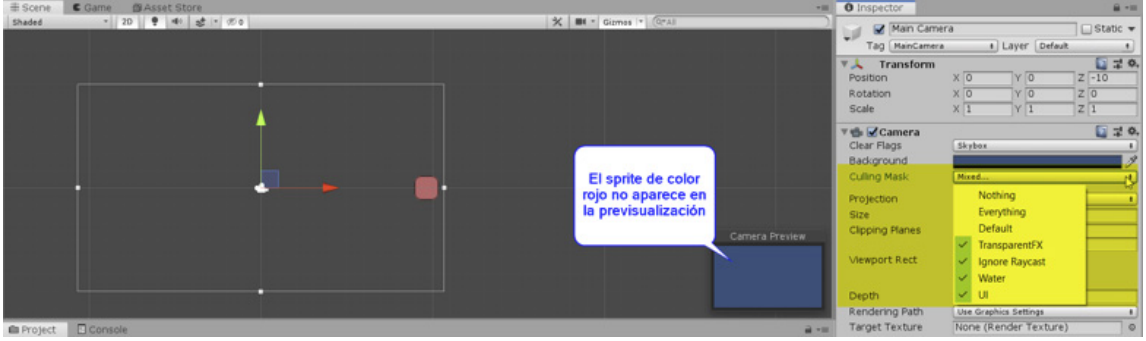
Panel inspector



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

- **Culling Mask.** Haciendo uso de la máscara de "desecho" se puede indicar que capas deben renderizarse. En las siguientes imágenes se puede observar la diferencia entre incluir una o no hacerlos dentro de la capa:

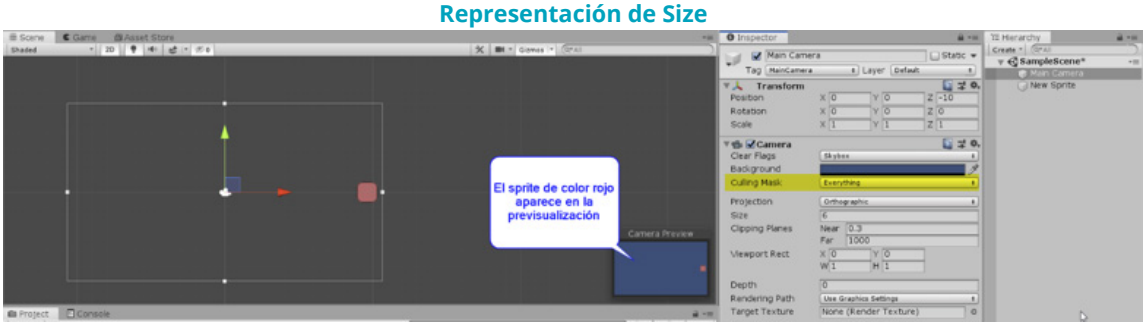
Representación de Culling Mask



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

- **Size.** Esta propiedad nos permite indicar el tamaño de viewport. En las siguientes imágenes se puede observar la diferencia entre un tamaño de 5 y seis unidades:

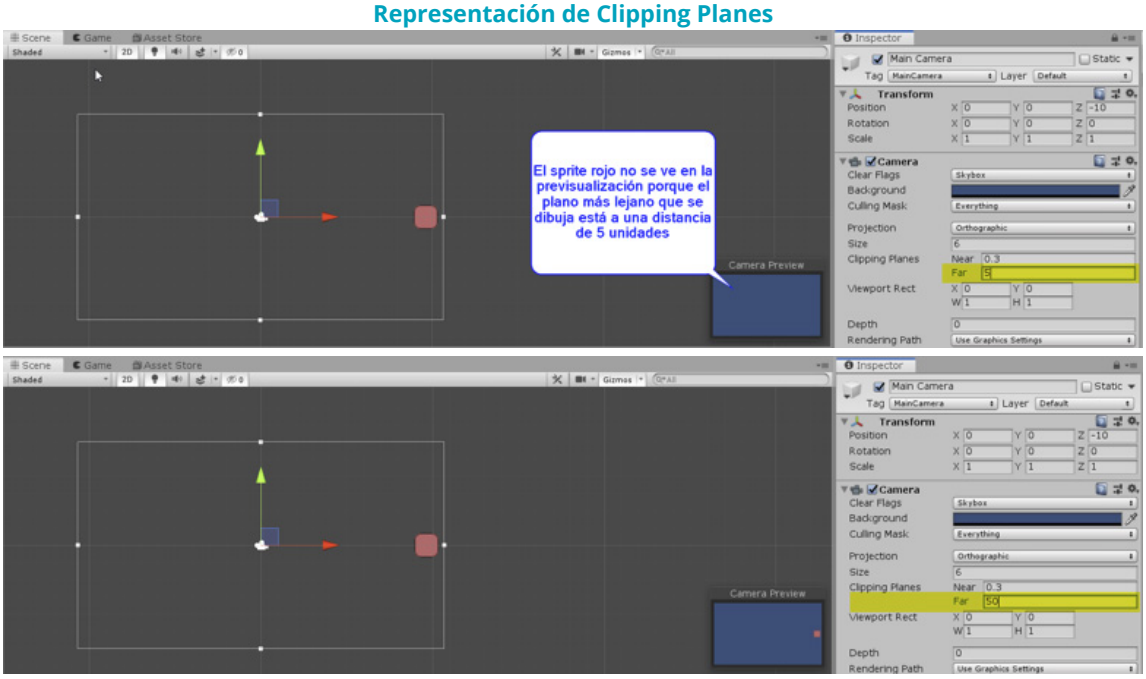
Representación de Size



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

- **Clipping Planes.** Haciendo uso de los planos de recorte se indica desde que distancia y hasta que distancia de la cámara se debe renderizar. Aunque sea un juego en dos dimensiones se pueden colocar objetos en diferentes planos de la cámara. En un juego en dos dimensiones no se permite a un jugador desplazarse en el eje Z, por lo tanto, los planos de recorte solo los tendremos en cuenta para asegurarnos de que todos los objetos están dentro de los planos que se muestran. En las siguientes imágenes se muestra lo que ocurre si se modifican los planos de recorte y se deja fuera un objeto:

Representación de Clipping Planes



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

Por último, recordar que en la unidad anterior vimos que las cámaras también nos permitían cambiar entre diferentes sistemas de coordenadas.

2.3.- ILUMINACIÓN.

En el apartado que hace referencia a los sprites ya se indicó que el material por defecto que utilizan los sprites no tienen en cuenta la iluminación. Es por esto que en muchos juegos en dos dimensiones no se utiliza iluminación. Solo incluiremos iluminación si algún material que utilizamos en juego necesita una fuente de luz.

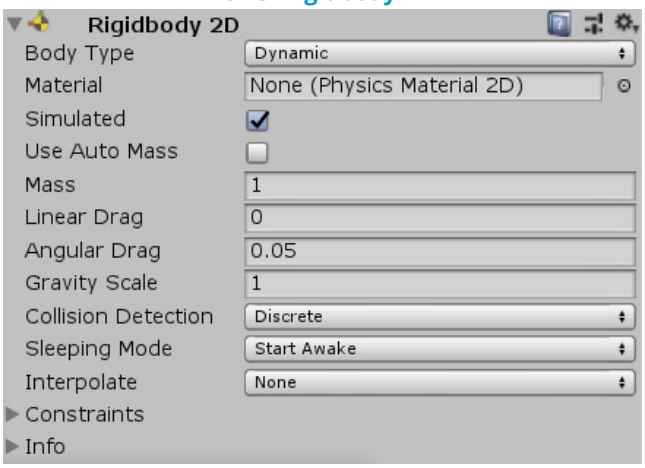
2.4.- FÍSICAS 2D.

El uso del motor de físicas que nos ofrece Unity va a hacer que sea más sencilla la manipulación de objetos que queremos que se comporten como si estuvieran bajo el efecto de la gravedad, por ejemplo. Haciendo uso del motor de físicas podremos aplicar una fuerza sobre un objeto para provocar un salto. Si no hacemos uso del motor de físicas tendremos que utilizar ecuaciones para calcular y asignar una posición a nuestro objeto en función de su peso, la fuerza hayamos aplicado y gravedad en cada frame. Es importante recordar que siempre que se realicen acciones que estén relacionadas con el motor de físicas deben realizarse en el método FixedUpdate del componente.

Rigidbody 2D

Un componente Rigidbody 2D coloca un objeto bajo el control del motor de físicas. Por lo tanto, si queremos que nuestro Game Object este bajo el control del motor de físicas le añadiremos un Rigidbody 2D.

Panel Rigidbody 2D



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Entre las propiedades más comunes se encuentra Body Type ,Mass y Collision Detection. El tipo de cuerpo permite indicar si el cuerpo se va a mover (Dynamic o Kinematica) o si va a estar en una posición fija (static). Fijar la masa de un cuerpo es importante para que el motor de físicas pueda realizar los cálculos de forma adecuada. Si queremos mover un objeto con una masa elevada tendremos que aplicar una fuerza elevada. Con la detección de colisiones vamos a fijar el modo en el que el motor debe realizar la detección colisiones. Por defecto se hace de forma discreta que es como menos recursos consume. Si al realizar las pruebas observamos que después de las colisiones parece que se producen saltos, cambiaremos al modo de detección de colisiones continua.

Rigidbody 2D va a modificar el transform del componente. Es por esto que se debe tener cuidado al modificar el transform del componente para no hacer que el objeto se comporte de forma no deseada. Aunque todavía no hemos visto los colisionadores, es importante indicar que los Collider 2D se enlazan al Rigidbody 2D del objeto.

detección. Los colisionadores 2D unidos al mismo Rigidbody 2D no chocarán entre sí. Esto significa que puede crear un conjunto de colisionadores que actúen de manera efectiva como un colisionador compuesto único, todos moviéndose y girando en sincronía con el Rigidbody 2D. Gracias a los colisionadores vamos a poder evitar que un personaje cruce una pared sin tener que programarlo en un script. En el caso de una pared utilizaremos un Body Type estático.

Collider 2D

Los componentes Collider 2D definen la forma de un GameObject a efectos de colisiones. Un colisionador, que es invisible, no necesita tener exactamente la misma forma que la malla de GameObject; de hecho, una aproximación aproximada es a menudo más eficiente e

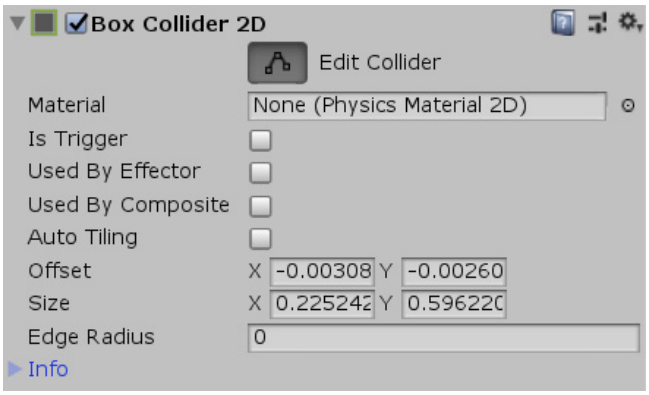
indistinguible en el juego. Los colisionadores para objetos 2D tienen nombres que terminan en "2D". Un Collider que no termina en 2D está diseñado para usarse en un GameObject 3D. Tenga en cuenta que no puede mezclar GameObjects 3D y Colliders 2D, ni GameObjects 2D y Colliders 3D. Los tipos Collider 2D que se pueden usar con Rigidbody 2D son:

- Circle Collider 2D
- Box Collider 2D
- Polygon Collider 2D
- Edge Collider 2D
- Capsule Collider 2D
- Composite Collider 2D

Siempre que sea posible y a la hora de realizar las pruebas no se observen comportamientos extraños será mejor utilizar Box Collider 2D ya que su uso consume menos recursos. De todas formas, habrá veces en las que un Box Collider 2D no nos sirva porque no se ajusta bien al objeto, por ejemplo: una colisión se detecta mucho antes de lo deseado porque el colisionador no se ajusta bien (es mucho mayor que el objeto) y antes de que dos GameObjects parezca que colisionan sus colisionadores colisionan haciendo que se realicen las acciones oportunas antes de tiempo.

A continuación, vamos a centrarnos en los Box Collider 2D, aunque la forma de trabajar con el resto de los colisionadores es análoga.

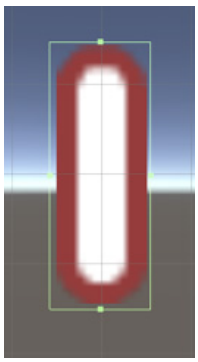
Panel Box Collider 2D



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Lo primero que debemos hacer con el colisionador es editar el colisionador y ajustarlo lo máximo posible al gráfico haciendo uso del botón Edit Collider.

Representación de edición el colisionador



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Tal y como se puede observar en las esquinas del colisionador hay un espacio hasta el gráfico que el usuario va a ver. En principio no debería ser un problema, pero si lo fuera tendríamos que utilizar un colisionador que se ajuste mejor al gráfico.

Para detectar la colisión entre dos colisionadores haremos uso de los mensajes que envía Unity. Para ello en el script que se encarga de controlar a uno de los objetos que han entrado en colisión utilizaremos uno de los siguientes métodos:

- OnCollisionEnter2D
- OnCollisionExit2D
- OnCollisionStay2D
- OnTriggerEnter2D
- OnTriggerExit2D
- OnTriggerStay2D

Ejemplo script

En el siguiente ejemplo se puede observar el script (componente) que se encarga de realizar las acciones necesarias una vez se detecta la colisión. El método recibe parámetro el colisionador que ha entrado en contacto con el colisionar del objeto que contiene el script. Todavía no hemos visto las etiquetas, pero se puede ver como haciendo uso de las etiquetas se puede saber el colisionador pertenece a un enemigo.

```
using UnityEngine;

public class Example : MonoBehaviour
{
    void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.tag == "Enemy")
        {
            //Acciones a realizar: quitar vida…
        }
    }
}
```

3.- SCRIPTS.

El comportamiento de GameObjects está controlado por los componentes que están unidos a ellos. Aunque los componentes integrados de Unity pueden ser muy versátiles, si se quiere activar eventos del juego, modificar las propiedades de los componentes a lo largo del tiempo y responder a la entrada del usuario de la forma que nosotros queremos tendremos que utilizar scripts. Unity admite el lenguaje de programación C# de forma nativa. Por lo tanto, los scripts los desarrollaremos utilizando C#. En la unidad anterior estudiamos el ciclo de vida de los componentes y vimos los principales métodos que se pueden utilizar la inicializar, modificar o deshabilitar … un componente. Estos son lo métodos que vamos a sobrescribir en nuestros scripts para que los objetos se comporten como nosotros queremos. A continuación, vamos a ver algunas de acciones que tendremos que realizar para poder controlar los objetos usando sus componentes, muchas de estas acciones se mostraron en la unidad anterior y únicamente vamos a hacer un breve repaso.

Acceder a los componentes

Si desde el script deseamos acceder a un componte del objeto utilizaremos la función GetComponent() tal y como se puede ver en el siguiente ejemplo:

```
void Start ()
{
    Rigidbody rb = GetComponent<Rigidbody>();

    // Change the mass of the object's Rigidbody.
    rb.mass = 10f;
}
```

A pesar de que se puede acceder a los componentes utilizando otras funciones que se vieron en la unidad anterior se recomienda utilizar la función GetComponent<T>() para evitar errores que se pueden producir si se realizan cambios en el código. Haciendo uso de los genéricos en caso de una clase cambie de nombre no existe riesgo ya que es el compilar el que indica que hay un error.

Acceder a otros objetos

La forma más sencilla para poder acceder a otros objetos es incluir un GameObject en el script y asignar la referencia a la variable. Esto se puede hacer en el editor directamente o si se conoce el nombre del objeto se puede asignar en alguno de los métodos que se encargan de

la inicialización. Una vez se dispone de la referencia al GameObject se puede utilizar su información para realizar las acciones pertinentes como en el siguiente ejemplo:

```
vpublic class Enemy : MonoBehaviour {
    public GameObject player;

    void Start() {
        // Colocar al enemigo 10 unidades más atras que el player.
        transform.position = player.transform.position - Vector3.forward * 10f;
    }
}
```

Acceder a objetos de la escena

En alguna ocasión puede que necesites obtener todos los objetos de la escena de un tipo. Para ello podremos utilizar la función FindObjectOfType<T> si solo buscamos un objeto o FindObjectsOfType si buscamos uno o más objetos. A continuación, se muestra un ejemplo:

```
void Start()
    {
        Enemy anEnemy = FindObjectOfType<Enemy>();

        if (anEnemy != null)
        {
            print("Encontrado un enemigo: " + anEnemy.name);
        }

        Enemy[] enemyList = FindObjectsOfType<Enemy>();
        foreach (Enemy enemy in enemyList)
        {
            print("Un nuevo enemigo: " + enemy.name);
        }
    }
```

Puede que algún caso no estemos buscado objetos que contengan un script. Si os habéis fijado en el ejemplo estamos utilizando una Clase para indicar los objetos que estamos buscando. Para búsquedas como buscar todos los objetos que estén etiquetados podremos utilizar la función GameObject.FindWithTag(Etiqueta) si buscamos un único objeto o GameObject.FindGameObjectsWithTag(Etiqueta) si buscamos uno o más objetos. A continuación, se muestra un ejemplo:

```
GameObject player;
GameObject[] enemies;

void Start()
{
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

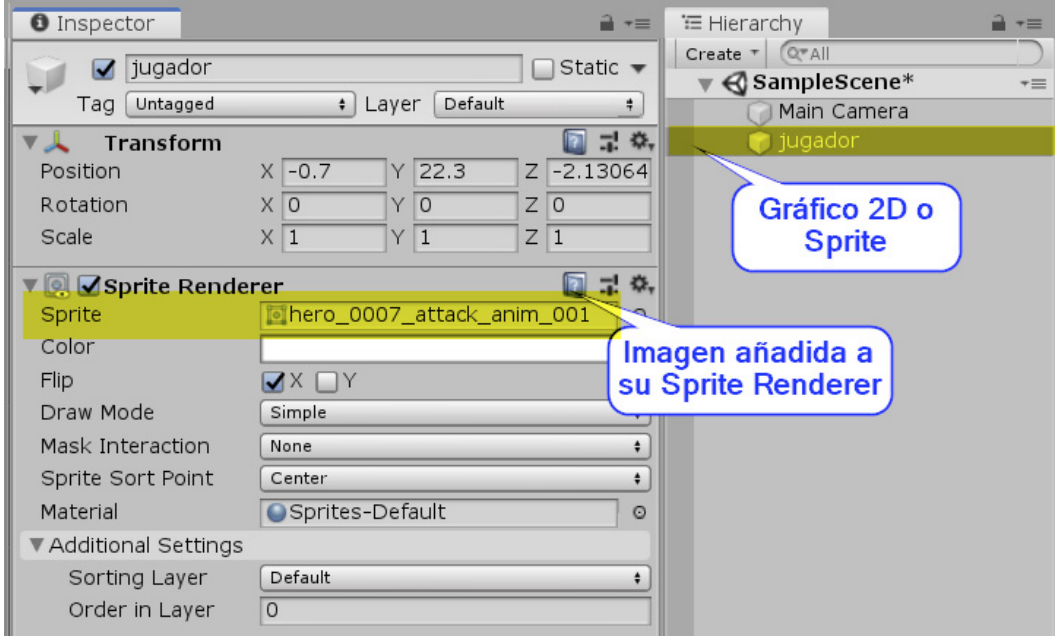
4.- ANIMACIÓN DE UN GRÁFICO 2D.

4.1.- CREAR UN CLIP.

Para poder utilizar una animación debemos crear un clip. Existen diferentes formas de crear los clips. En nuestro caso vamos a utilizar una secuencia de gráficos 2D. Para aquellos que hayáis utilizado animaciones en CSS3 os resultará familiar la forma de trabajar. Para aquellos que no hayáis utilizado animaciones de este tipo, el ejemplo que os resultará de ayuda son las animaciones que se crean en cuadernos. En cada folio se dibuja el personaje con una pequeña diferencia respecto a la hoja anterior. Para ver la animación solo hay que pasar las hojas rápidamente.

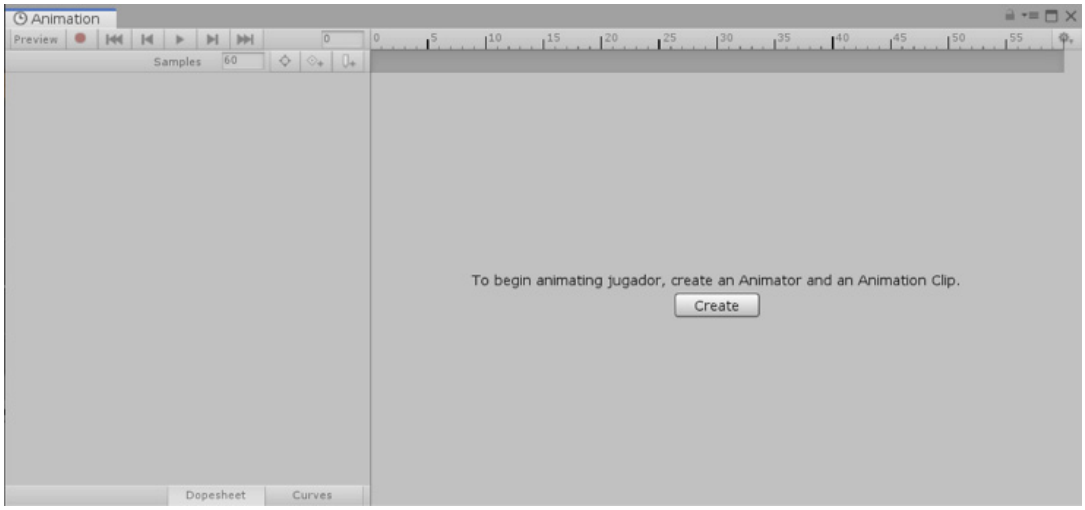
Los pasos que vamos a seguir para realizar el clip son los siguientes:

1. Crear un Sprite en la jerarquía y añadirle un gráfico 2D



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

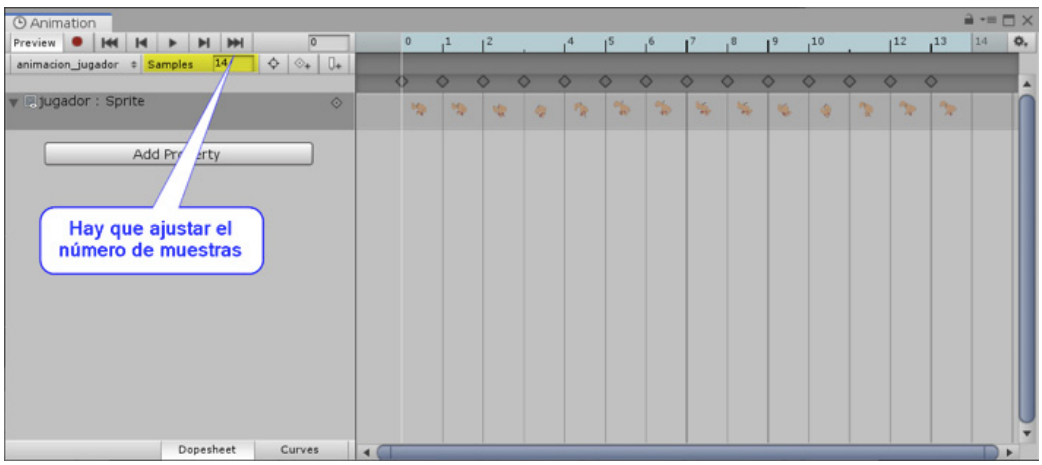
2. Crear la animación. Para ello con el GameObject seleccionado en la jerarquía Window->Animation->Animation (Se puede utilizar el atajo Ctrl+6).



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

3. Unity nos indica que tenemos que crear un Animator y Animation Clip. Pulsado en Create

4. Añadimos las diferentes imágenes que vamos a utilizar en la animación y ajustamos en número de muestras del clip.



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

4.2.- ANIMATOR.

Una vez que tenemos el clip, solo nos falta definir el Animator para que el Sprite se comporte como nosotros queremos. Dentro del Animator tendremos que configurar los siguientes aspectos:

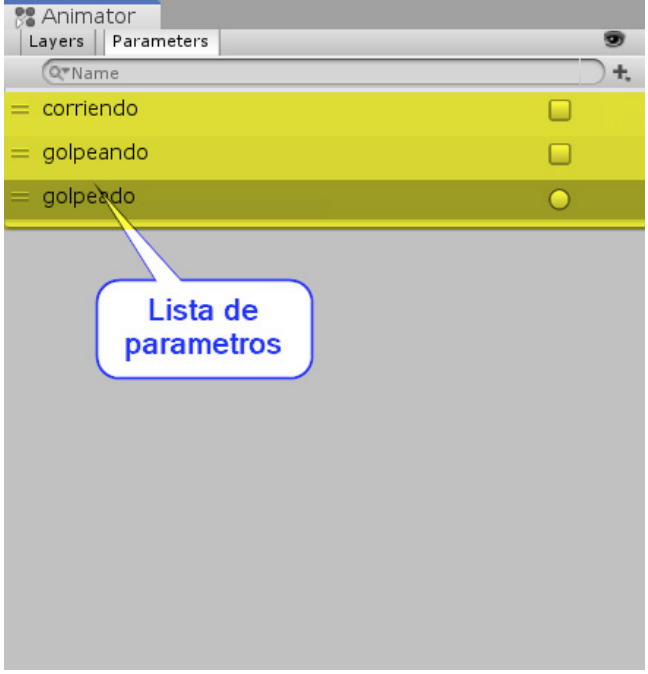
- **Estados**. En los estados indicaremos la animación que debe reproducirse y podremos ver las transiciones desde el estado.

Representación del panel

BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

- **Parámetros**. Los parámetros del Animator nos van a permitir cambiar entre los diferentes estados.

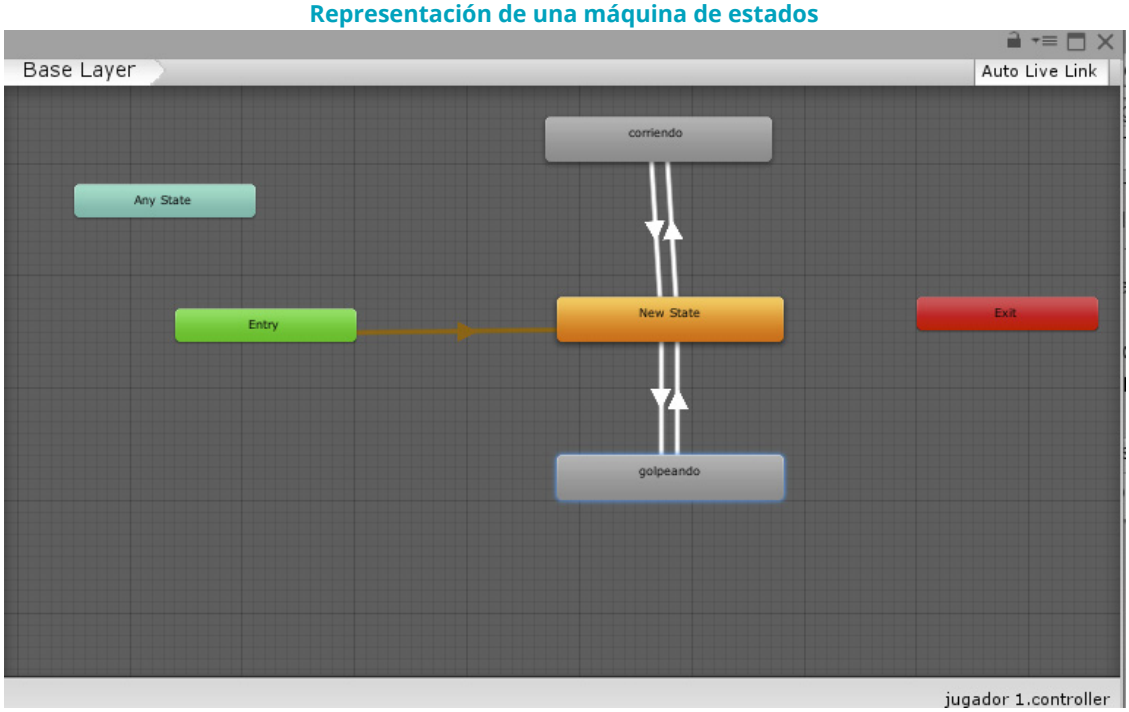
Representación del panel



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unit Technologies

A continuación, podemos ver **una máquina de estados** que incluye tres estados y sus transiciones.

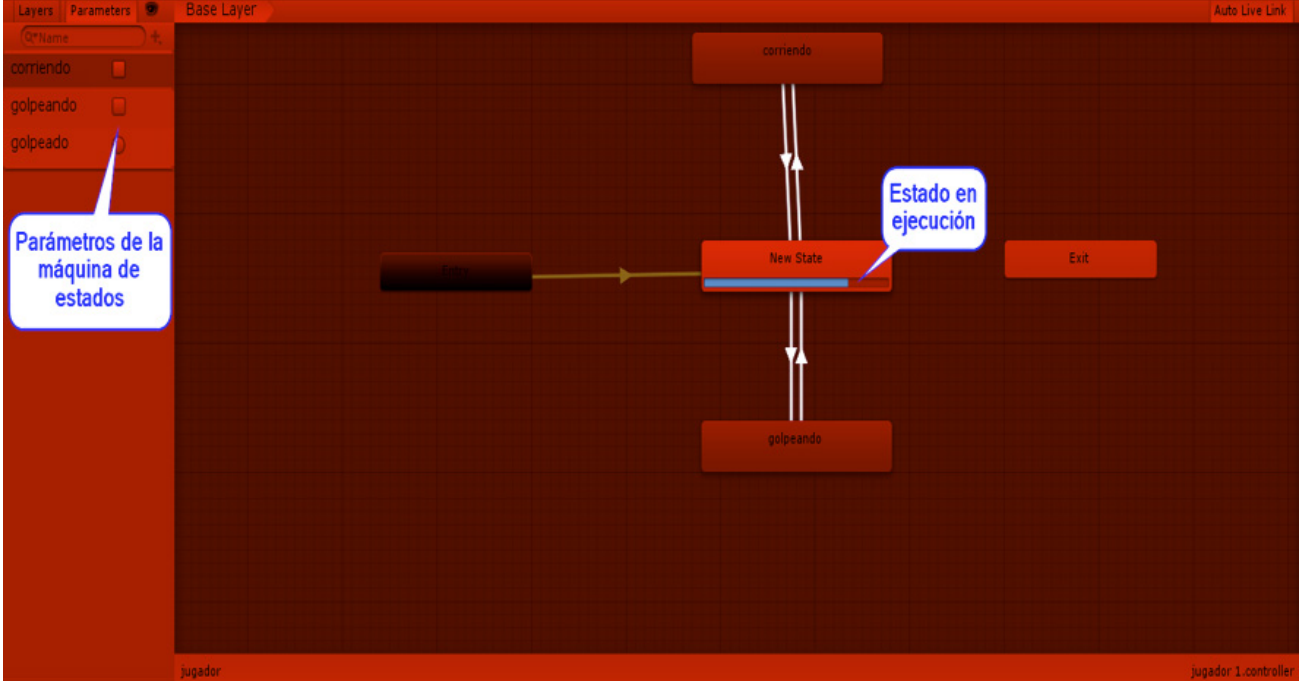
Representación de una máquina de estados



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Una vez arranquemos el reproductor podremos ver en que estado se encuentra el Animator y las transiciones entre estados.

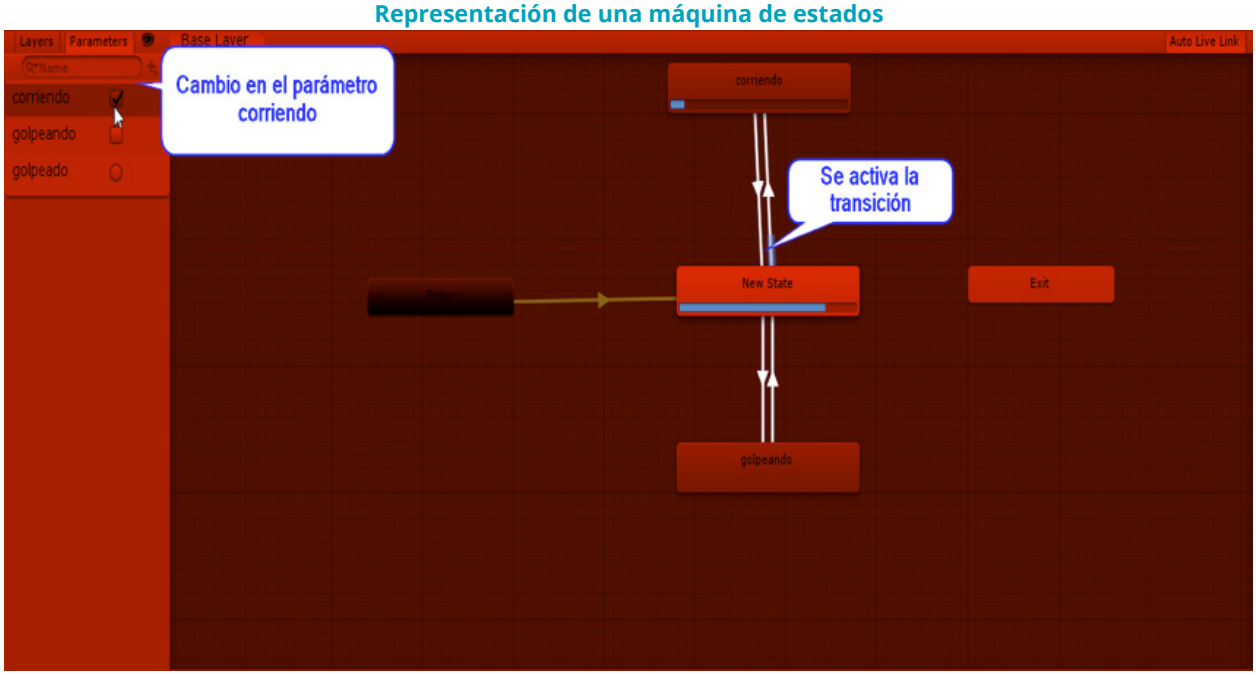
Representación de una máquina de estados



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

Al cambiar el estado de alguno de los parámetros se produce una transición:

Representación de una máquina de estados



BIRT LH (Copyright (cita)) *(link: https://unity.com/)* Obra derivada de captura de pantalla del programa Unity, propiedad de Unity Technologies

4.3.- CAMBIO DE LOS PARÁMETROS DESDE UN SCRIPT.

Ya hemos visto como cambiando el valor de los parámetros se puede provocar cambios en la máquina de estados. Pero, una vez se genere el ejecutable no se va a poder acceder al editor.

Por lo tanto, será necesario poder cambiar el valor de los parámetros desde un script. Para poder acceder a los parámetros realizaremos las siguientes acciones:

1. Definir una variable de tipo Animator en el script.
2. En uno de los métodos que se encargan de la inicialización obtener una referencia al componente.
3. Haciendo uso de los métodos SetFloat, SetInteger, SetBool, SetTrigger o ResetTrigge establecer los valores de los parámetros.

```
using UnityEngine;
using System.Collections;

public class SimplePlayer : MonoBehaviour {

    // definimos la variable (1)
    Animator animator;

    // Use this for initialization
    void Start () {
        // obtenemos la referencia al componente (2)
        animator = GetComponent();
    }

    // Update is called once per frame
    void Update () {
        float h = Input.GetAxis("Horizontal");
        float v = Input.GetAxis("Vertical");
        bool fire = Input.GetButtonDown("Fire1");
        // establecemos el valor de los parámetros (3)
        animator.SetFloat("Forward",v);
        animator.SetFloat("Strafe",h);
        animator.SetBool("Fire", fire);
    }

    void OnCollisionEnter(Collision col) {
        if (col.gameObject.CompareTag("Enemy"))
        {
        // establecemos el valor de los parámetros(3)
            animator.SetTrigger("Die");
        }
    }
}
```