Animación en Unity3D

UNIDAD 6: Programación de videojuegos 2d

Roberto Rodríguez Ortiz

Versión 1.1 Febrero 2017

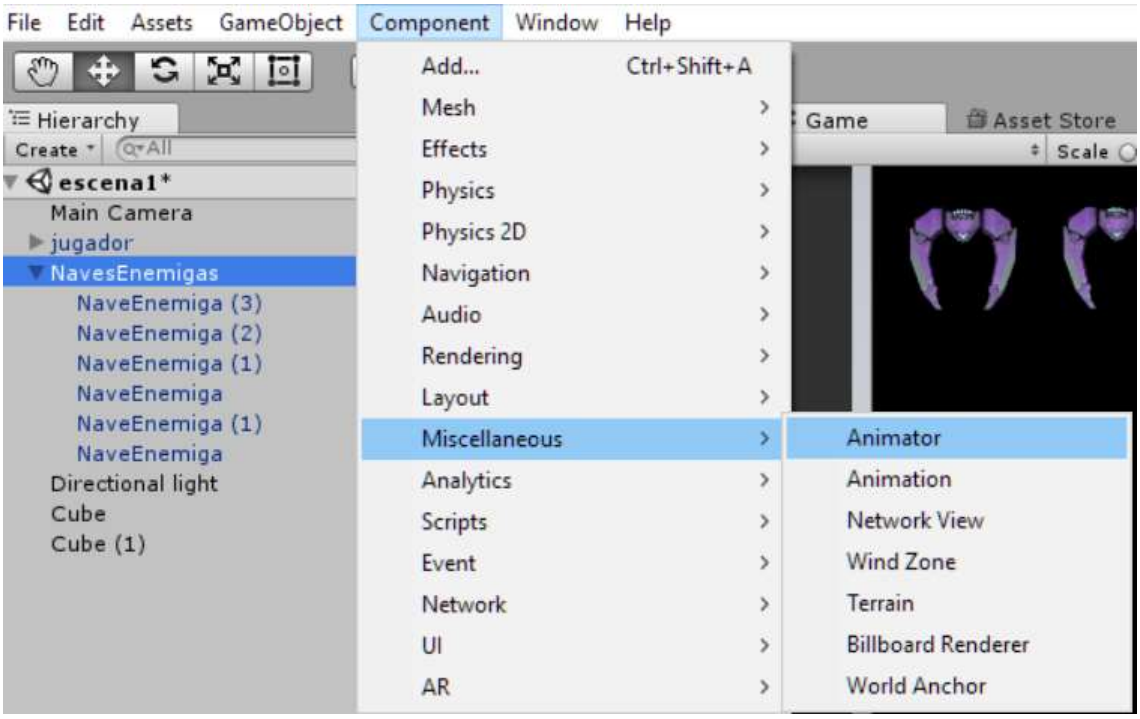
Contenido

1. Animando un objeto

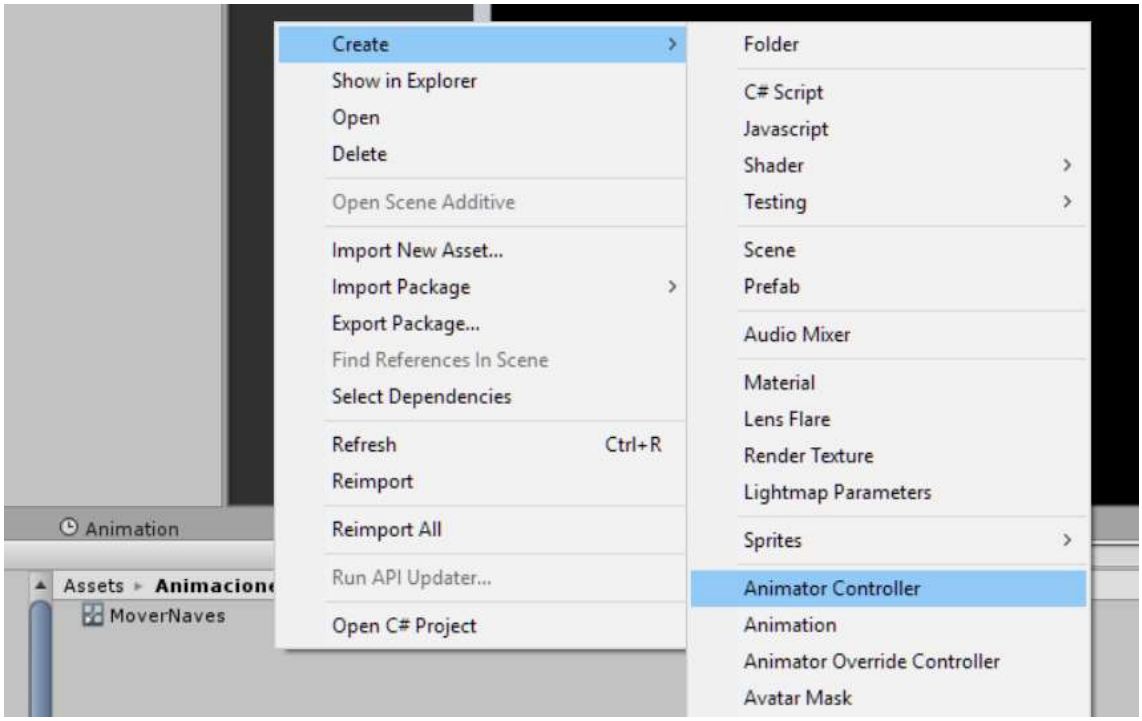
Incluir animaciones en objetos de la escena enriquece el apartado visual y genera mecánicas más divertidas, es un elemento esencial de cualquier videojuego. Unity proporciona un completo animador con una máquina de estados que nos permite realizar animaciones muy completas que pueden ser utilizadas desde los Scripts de nuestra escena.

Como ejemplo vamos a animar las naves del juego Space Invaders, vamos a conseguir que las naves realicen un movimiento horizontal como en el juego original.

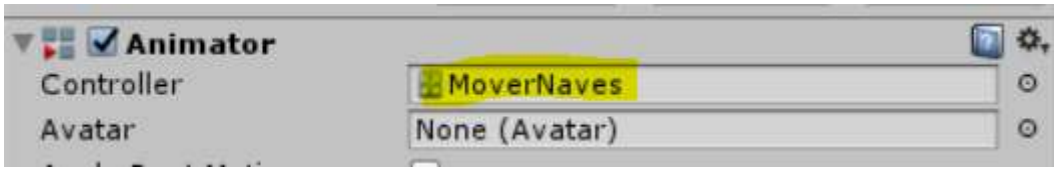
Las naves están creadas a través de un prefab y están agrupadas dentro de un objeto vacío al que hemos llamado NavesEnemigas, lo primero en añadir a este objeto un **Animator**:



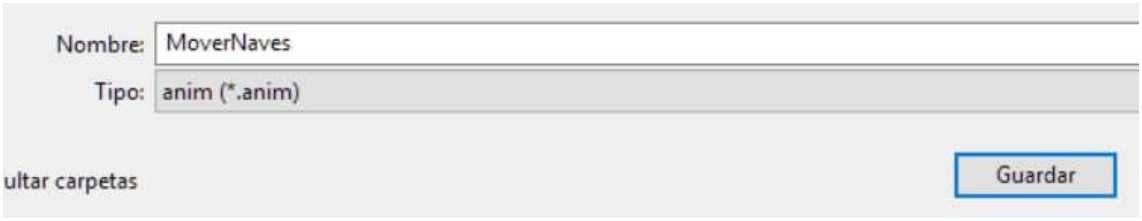
Debemos crear un **Animator Controler** en nuestros Assets, para ello he creado la carpeta Animaciones, siempre es mejor tenerlo organizado.



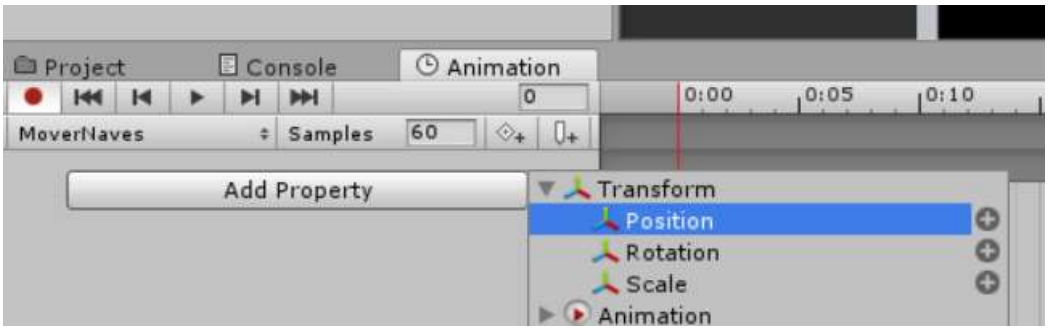
Lo he llamado MoverNaves, a continuación seleccionamos el objeto NavesEnemigas en la jerarquía y en el inspector asignamos el Animator Controller que hemos creado a su componente controller



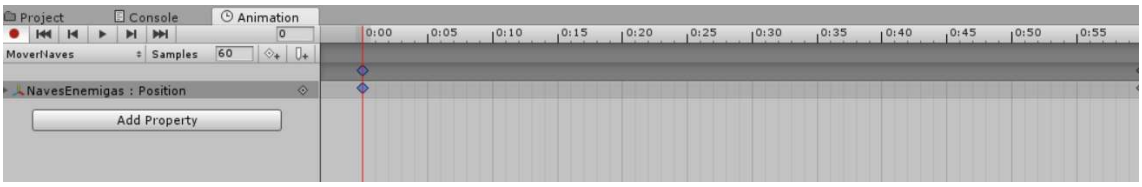
Ahora seleccionamos la pestaña Animation pulsamos sobre Create para crear un **AnimationClip** lo guardamos como MoverNaves.



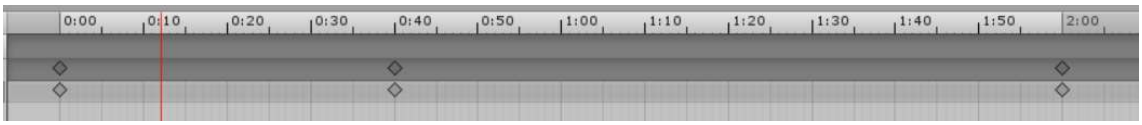
Pasamos a la creación de la animación y seleccionando el objeto NavesEnemigas en la escena pulsamos sobre Add Property y seleccionamos Transform->Position para aplicar una animación a través de la posición del objeto.



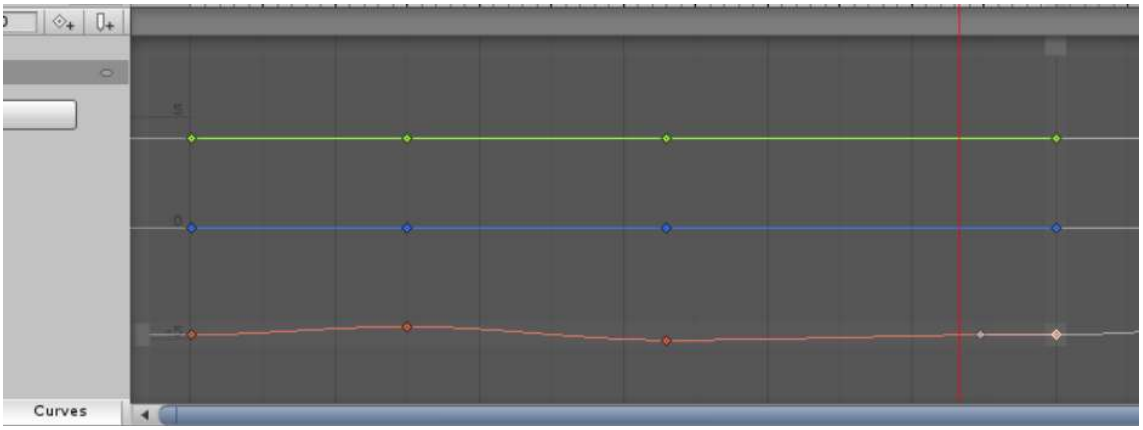
Veamos el Animation Clip Mover Naves, el botón rojo indica que estamos grabando la animación y tenemos unos controles para reproducirla y ver como queda en el editor. La animación consta de 60 Samples y el botón a la derecha nos permite introducir key frames que sería una nueva posición del objeto. Si pulsamos sobre la línea de tiempo aparecerá una línea roja y si movemos el objeto en la escena añadirá un keyframe, es la forma más sencilla



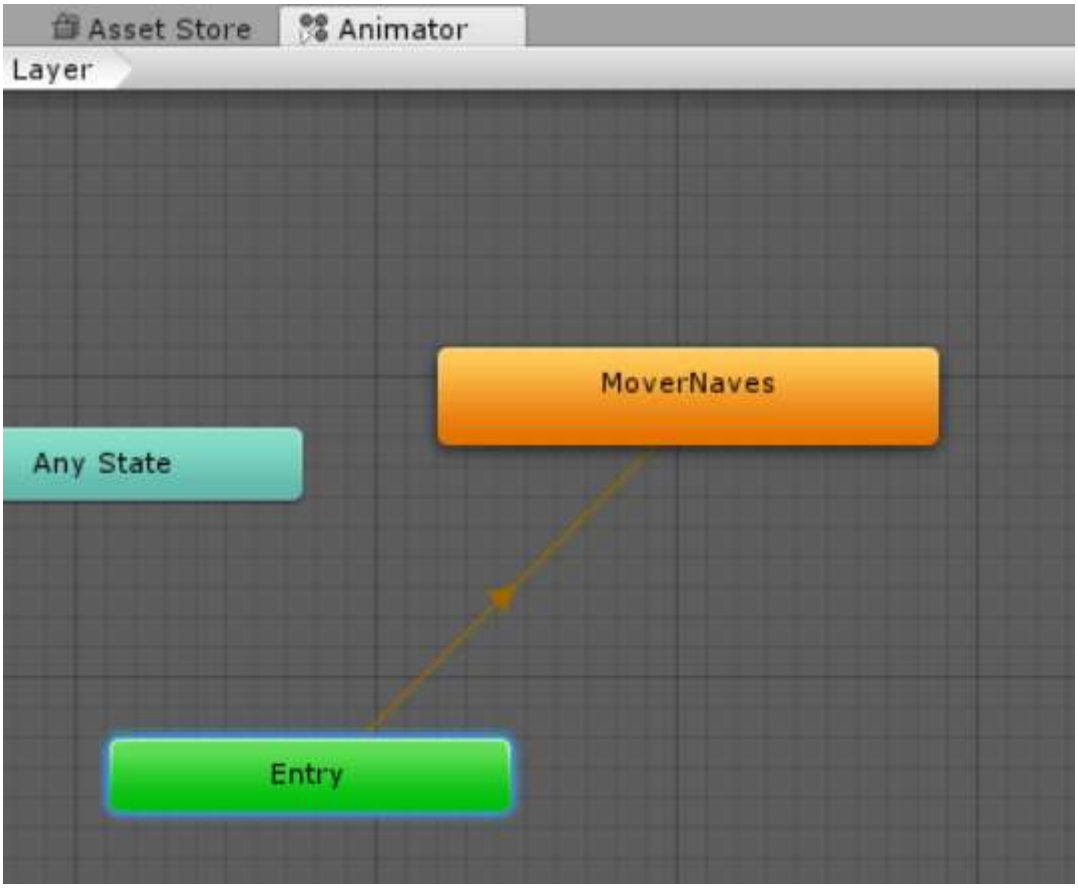
Si añadimos un **keyframe** donde el objeto NavesEnemigas esté más a la derecha lograremos un efecto de movimiento dentro del eje y, el keyframe puede moverse dentro de la línea de tiempo para buscar el mejor efecto o pueden añadirse más keyframes para realizar movimientos más complejos. La duración de la animación también puede configurarse en la parte superior donde aparecen las marcas de los segundos, lo mejor es probar con distintos valores y visualizar el efecto generado.



Se puede editar también a través de curvas (pestaña Curves), cada una de ellas representa un eje (en rojo el eje x), es un editor más visual, elige el que te sea más cómodo:



Una vez que tenemos creada la animación tenemos que establecer cuando se ejecuta, para ello abrimos el Animator Controller MoverNaves, por defecto la primera acción será ejecutar la animación que hemos creado:



Desde este editor podemos añadir más animaciones y transiciones entre las mismas (lo veremos en el siguiente punto donde animaremos sprites).

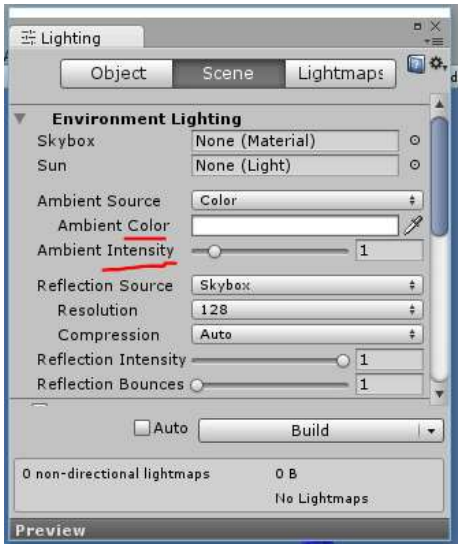
2. Animando un Sprite

Si deseáis realizar la siguiente actividad os podéis descargar los assets en forma de paquete desde la página de los tutoriales (magníficos aunque en este apartado un tanto anticuados) de Walker Boys http://www.walkerboystudio.com/wbstudio/project-2d-mario/.

2.1. Configuración

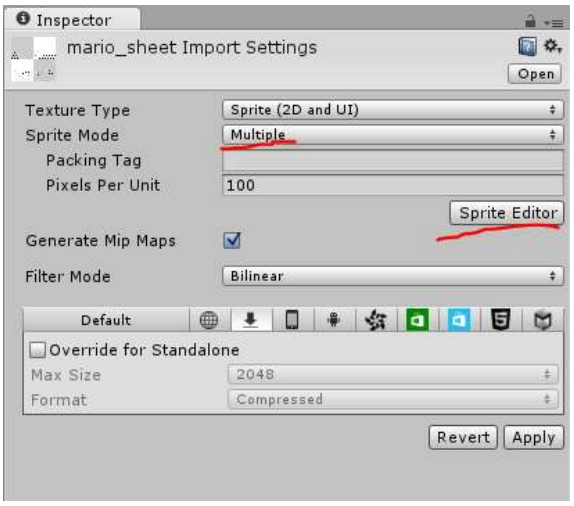
2.1.1. Iluminar la escena

Antes de insertar los sprites en la escena vamos a iluminarla de forma que se vean con claridad. En el menú Windows -> Lighting, vamos a seleccionar un color para la luz ambiente que sea de color claro :

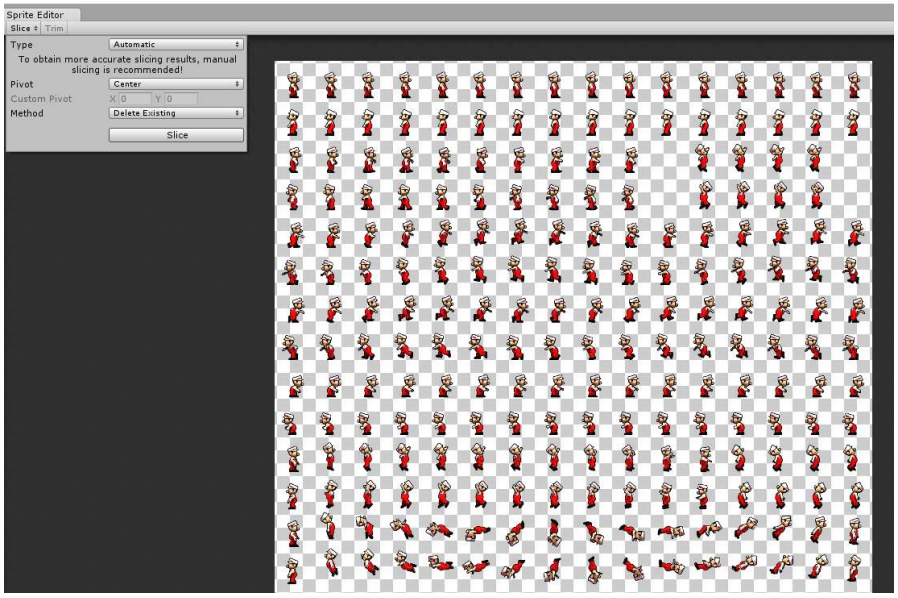


2.1.2. Importar un nuevo sprite sheet

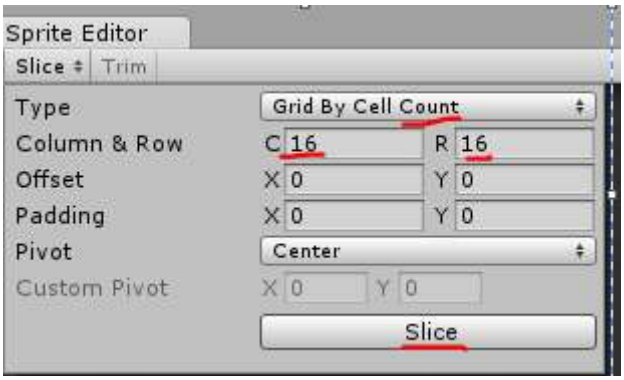
Los sprite sheet son imágenes que contienen varias modelos de un mismo personaje en distintas posiciones que utilizaremos para realizar las animaciones. Seleccionaremos Sprite mode -> Multiple y posteriormente abriremos el sprite editor



Dentro del Sprite Editor, podemos separar la imagen en unidades de forma manual o automática, elegiremos la forma manual que puede realizarse por tamaño en píxeles o por unidades. En este caso elegimos uno de los sprite sheets de Mario :

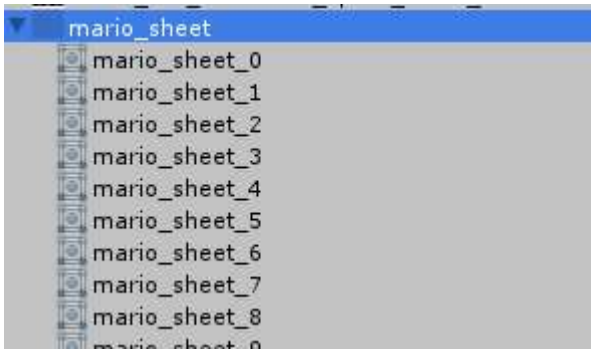


En este caso vamos a utilizar unidades (Grid By Cell Count), dado que tenemos una matriz de 16 x 16 imágenes, también podríamos utilizar By Cell Size (en nuestro caso 256x256 píxeles):



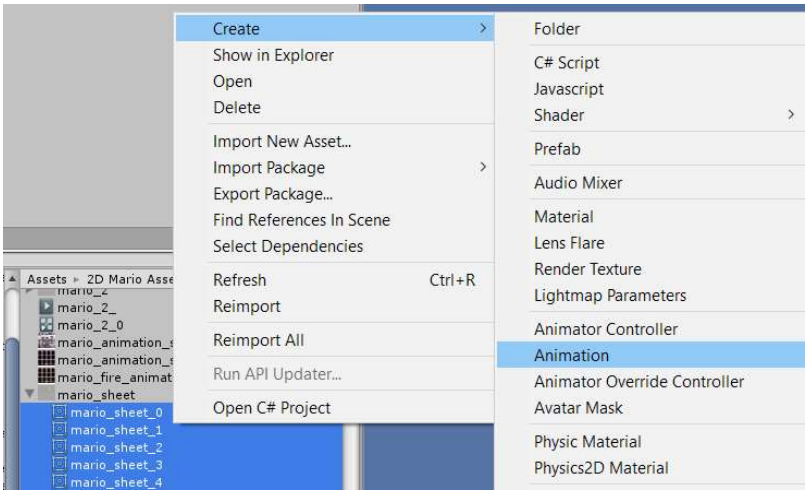
Una vez pulsado slice veremos como se muestra cada cuadro dentro de la imagen

Dentro de los assets del proyecto nuestro sprite se habrá dividido en imágenes :



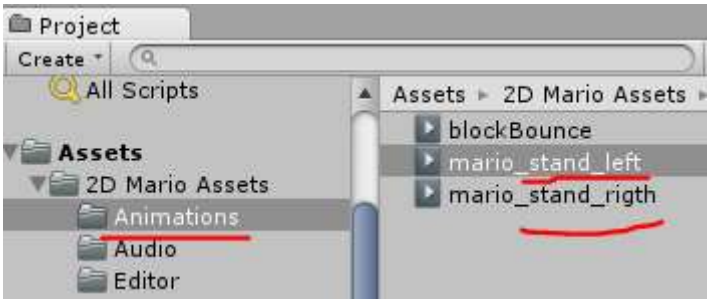
2.2. Creación de las animaciones

Podemos seleccionar una serie de imágenes y crear una animación con las mismas a través del botón derecho del ratón (en este caso de la imagen 0 a la 15)



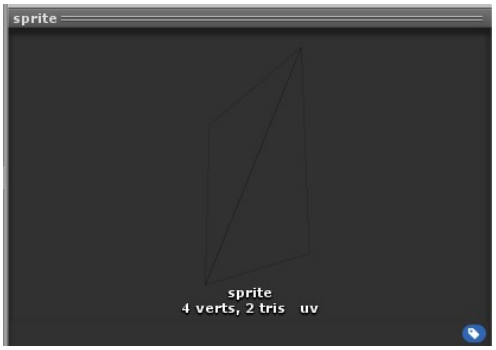
Le asignamos el nombre mario_stand_rigth , pues la animación muestra a mario parado mirando hacia la derecha. Lo guardamos en la carpeta animations.

Realizamos el mismo proceso para el resto de animaciones (por ejemplo de la imagen 16 a 31 sería mario detenido mirando a la izquierda mario_stand_left)

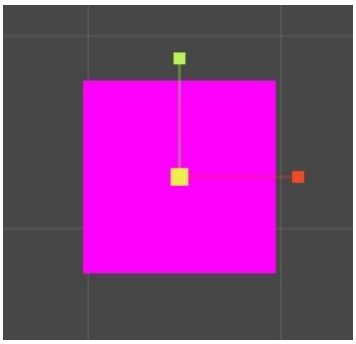


2.2.1. Insertar el sprite en la escena

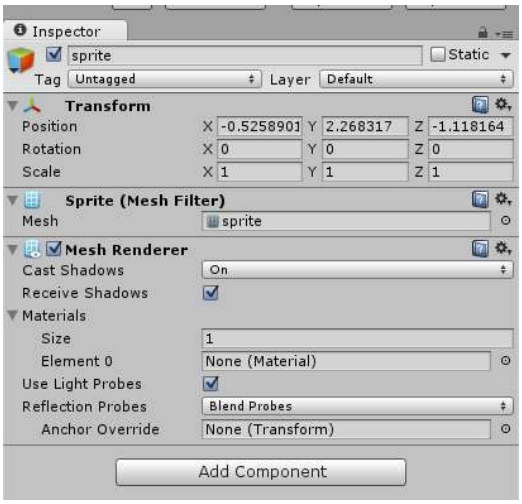
Insertamos en la escena un plano que va a contener el sprite :



En la escena:



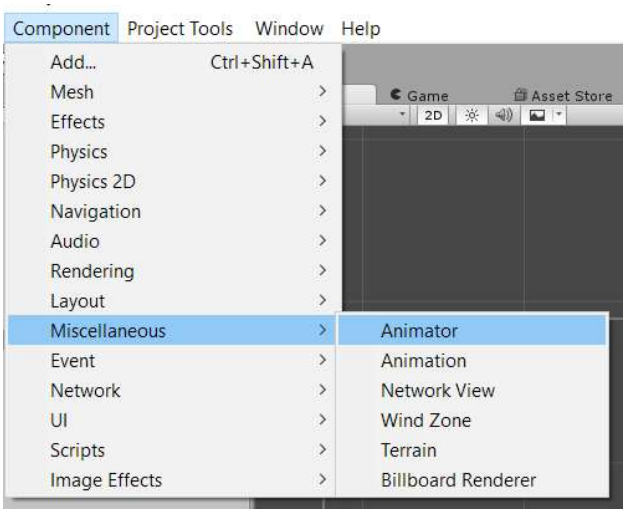
En el inspector:



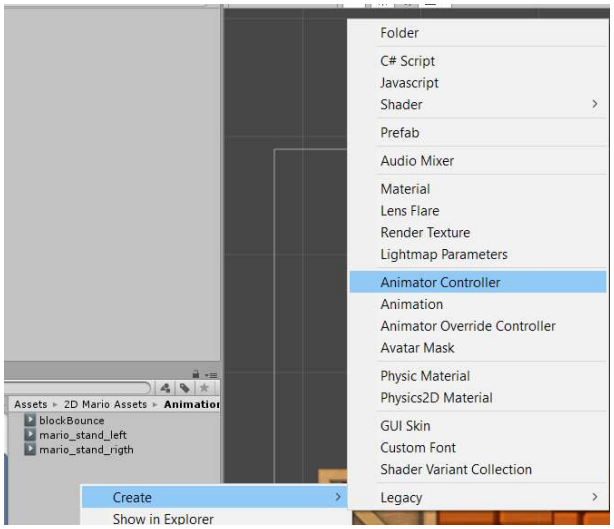
O bien, arrastro una de la imágenes de Mario hacia la escena (le cambio la escala a 0,5 por ser muy grande) (Vistas Assets -> Scene -> Inspector)

2.2.2. Animar el sprite

Le añadimos un Animator a nuestro sprite :

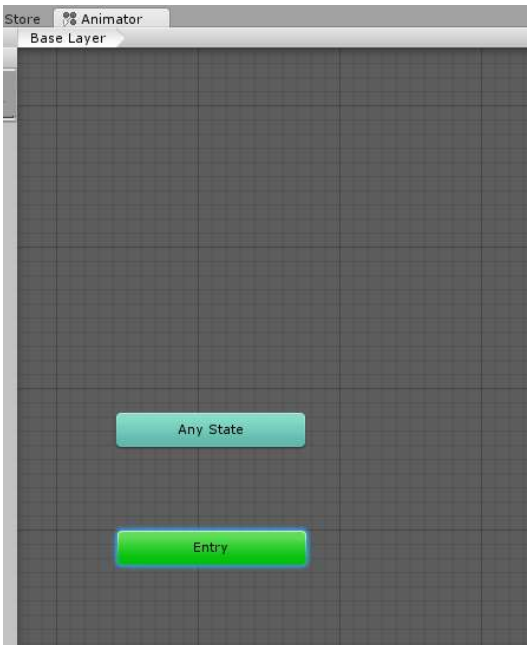


Añadimos un Animator Controler dentro de los Assets :

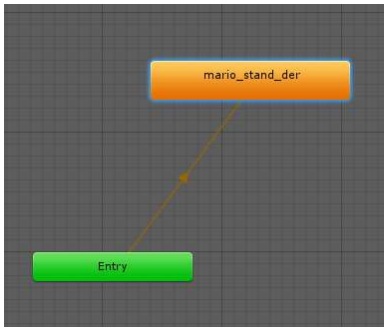


Lo añadimos al animator del objeto :

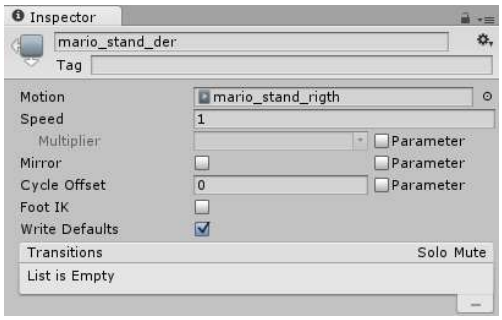
Doble click sobre el Animator Controller y abrimos el editor de Animaciones :



Creamos un nuevo estado (parado mirando hacia la derecha):



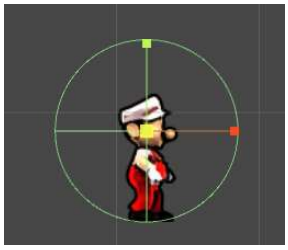
Y le añadimos la animación que ya teníamos:

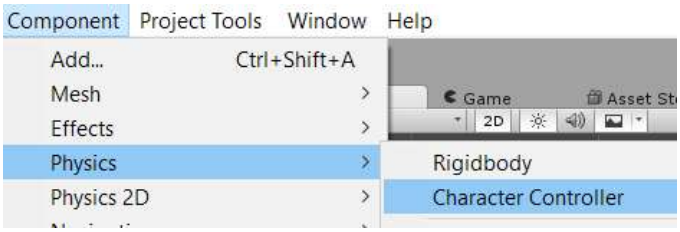


Si ejecutamos el juego Mario debería mostrar la animación…

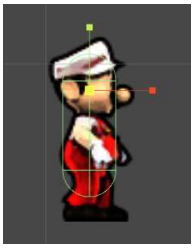
2.3. Character Controller

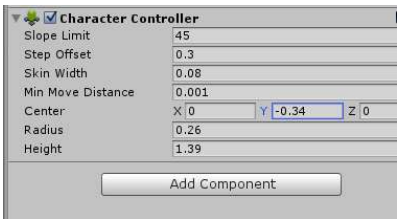
Para controlar el movimiento y las animaciones del personaje, vamos a añadir un Character Controller al personaje Mario:





Vamos a adaptar su tamaño a la forma del personaje, tendrá la forma de una cápsula y servirá tanto de Collider como de RigidBody:





Ahora vamos con el código, le asignamos un script, Script_PlayerControl. En el inicio vamos a obtener las referencias tanto al Animator como al Character Controller :

```
private Animator animator;
CharacterController controller;

void Start()
{
    animator = this.GetComponent<Animator>();
    controller = GetComponent<CharacterController>();
}
```

Vamos a aplicarle la gravedad (¡hay que obedecer la leyes!).

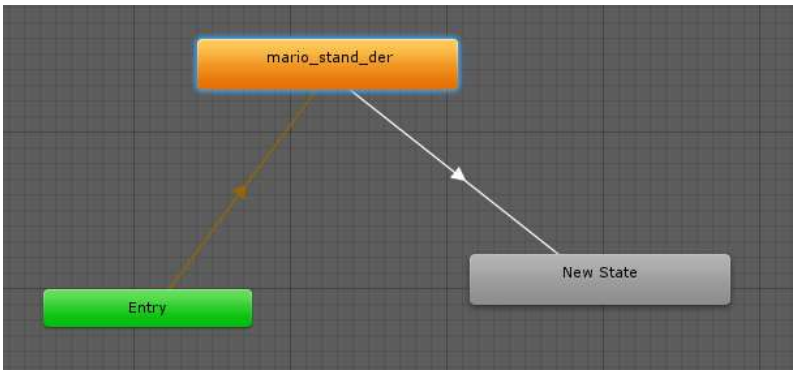
```
public float gravity = 20.0F;
private Vector3 moveDirection = Vector3.zero;

void Update()
{
    moveDirection.y -= gravity * Time.deltaTime;
    controller.Move(moveDirection * Time.deltaTime);
```

}

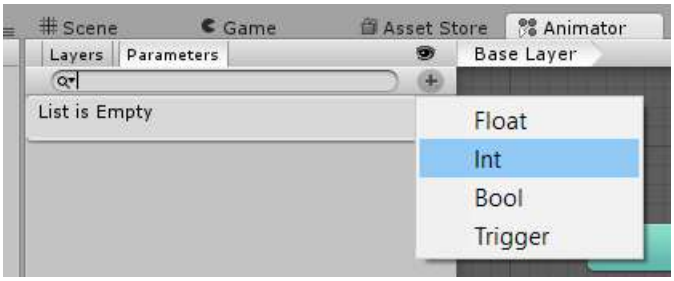
Time.deltaTime : delTime es el tiempo desde que se renderizó el último frame, añadiéndolo como factor de multiplicación en el movimiento conseguimos la independencia del movimiento respecto a la velocidad de renderizado del dispositivo.

Ahora queremos que responda a la entrada de teclado (o el control que deseemos) y que las animaciones cambien en función de esa entrada. Para ello debemos volver al editor de animaciones .

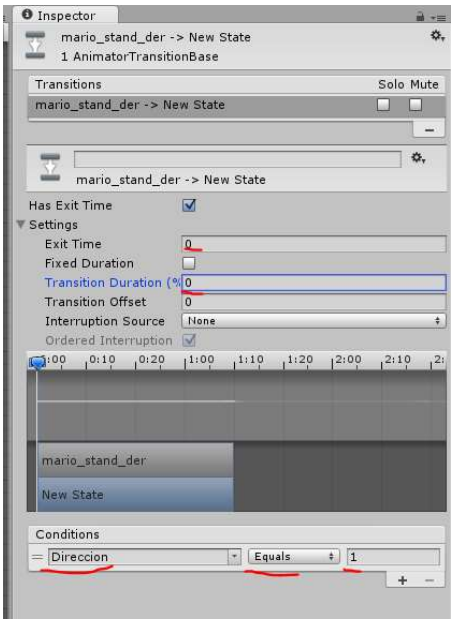
Creamos un nuevo estado :



Y le añadimos una transición desde el anterior. Sobre la transición añadimos un parámetro dirección, de forma que un cambio en dicho valor lance la transición:

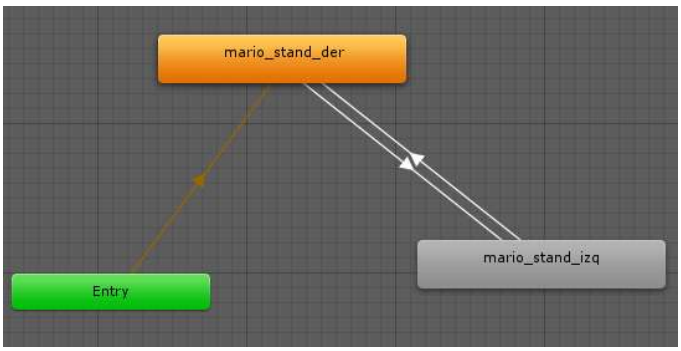


Puede ser de tipo entero, una vez creada seleccionamos la transición y añadimos como condición para que se lance que dicha variable tome el valor -1.



También quitamos los tiempos de transición para que el cambio sea inmediato.

Añadimos otra transición para que vuelva de nuevo al estado anterior (dirección valor 1)

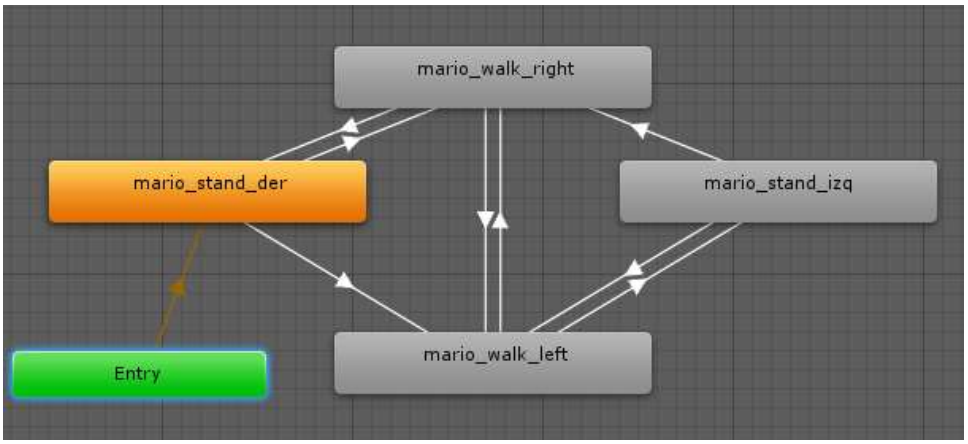


Es decir la variable dirección tomará el valor 1 cuando Mario vaya hacia la derecha y -1 cuando vaya hacia la izquierda. Ahora tendremos que modificar el código para que dicha variable se modifique según la entrada de teclado.

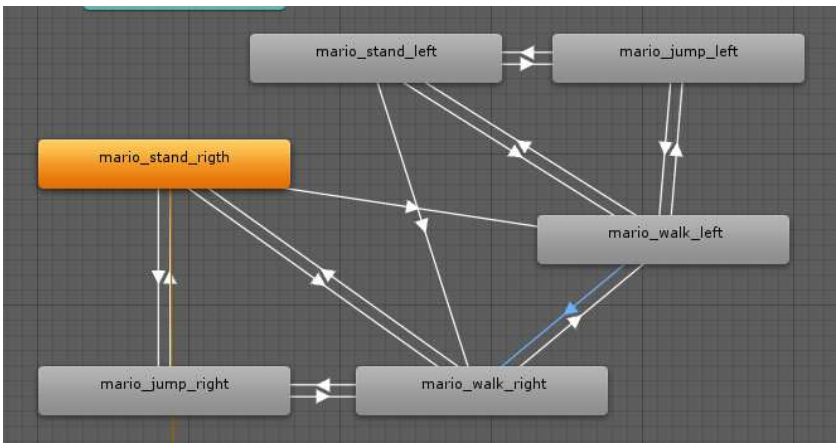
```
void Update()
{
    var horizontal = Input.GetAxis("Horizontal");
    if (controller.isGrounded)
    {
        if (horizontal > 0)//movimiento hacia la derecha
        {
            animator.SetInteger("Direccion", 1);
        }
        else if (horizontal < 0)//movimiento hacia la izquierda
        {
```

```
            animator.SetInteger("Direccion", -1);
        }
        else if (horizontal == 0)
        {
            animator.SetInteger("Direccion", 0);
        }

        moveDirection = new Vector3(Input.GetAxis("Horizontal"), 0, 0);
        //de coordenadas locales a globales
        moveDirection = transform.TransformDirection(moveDirection);
        moveDirection *= speed;
        if (Input.GetButton("Jump"))//si se pulsa la tecla salto
            moveDirection.y = jumpSpeed;
    }
    moveDirection.y -= gravity * Time.deltaTime;
    controller.Move(moveDirection * Time.deltaTime);

}
```

Las transiciones completas, considerando solo el movimiento izquierda y derecha, serían



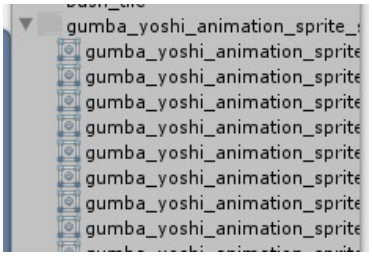
Si añadimos el salto quedaría así (he utilizado una variable salto de tipo entero, 1 salta, 0 no salta).



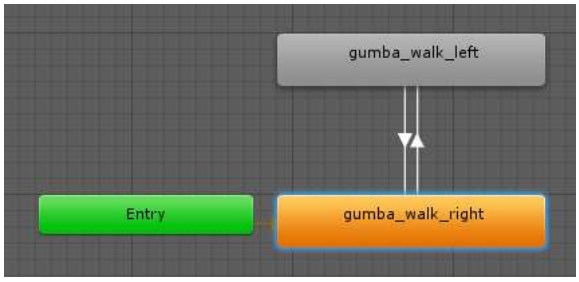
2.4. Añadiendo a Gumba

Gumba es una seta que se mueve de derecha a izquierda (cuando choca contra un obstáculo cambia de dirección) , si toca a Mario este pierde una vida y si Mario le salta encima desaparece.

Para crear los sprites seguimos el mismo proceso que con Mario:



Nos bastaría con las animaciones de andar a izquierda y a derecha.

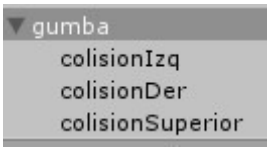


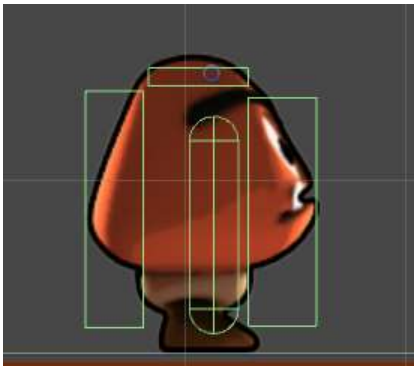
Codigo del script que controlará a gumba :

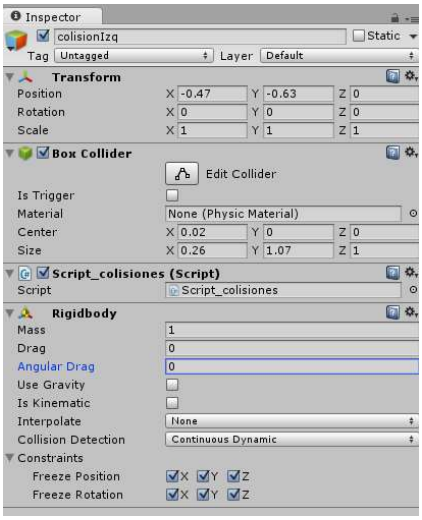
```
CharacterController controller;
private Vector3 moveDirection = Vector3.zero;
public int speed = 20;
private int direccion;
void Start()
{
   // animator = this.GetComponent<Animator>();
    controller = GetComponent<CharacterController>();
    direccion = 1;//inicialmente se mueve hacia la derecha
}

void Update () {
     //se mueve en la dirección indicada (negativo =>
                hacia la izda, positivo => hacia la derecha)
    moveDirection.x = direccion;
    moveDirection = transform.TransformDirection(moveDirection);
    //le aplicamos la velocidad, cuanto mayor sea más deprise se moverá
    moveDirection.x += speed * Time.deltaTime; //aplicamos la gravedad.
    moveDirection.y -= 10* Time.deltaTime;
    controller.Move(moveDirection * Time.deltaTime);
}
//marca la direccion en la que moverse, se llama desde las colisiones
public void setDireccion ( int dir)
{
```

```
        direccion = dir;
    }
```

A gumba le añadiremos un Character Controller y varios objetos hijos vacíos que contendrán un Box collider y un Rigidbody para detectar las colisiones izquierda, derecha y superior :







A los objetos vacíos que deben detectar las colisiones los llamamos colisionIzq, colisionDer y colisionSuperior y les añadimos un script que se encarga de cambiar la dirección de gumba cuando se choca lateralmente:

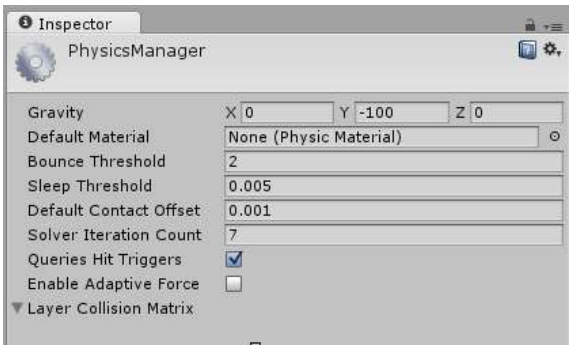
```
private Animator animator;
    Script_gumba script_gumba;
    void Start()
    {   //accedemos a los componentes del objeto padre (gumba)
        animator = gameObject.GetComponentInParent<Animator>();
        script_gumba = gameObject.GetComponentInParent<Script_gumba>();
    }

    void OnCollisionEnter(Collision col)
    {

        if ( gameObject.name=="colisionDer")
        {
            //si chocamos con algo a la derecha nos movemos hacia la izquierda
            animator.SetInteger("Direccion", -1);
            script_gumba.setDireccion(-1);
        }
        if (gameObject.name == "colisionIzq")
        {
            animator.SetInteger("Direccion", 1);
            script_gumba.setDireccion(1);
        }
    }
```

Para destruir a Gumba si Mario le salta encima, añadimos el siguiente código al script de Mario, al tener un Character Controler utilizamos la función OnControllerColliderHit :

```
void OnControllerColliderHit(ControllerColliderHit hit)
{
    Collider colider = hit.collider;
     //si Mario choca con gumba recibe un impulso en dirección contraria
    if (colider.transform.parent.name == "gumba" )
    {
        controller.Move(Vector3.Reflect(hit.moveDirection, hit.moveDirection)
                * 20 * Time.deltaTime);
    }
     //si saltamos sobre gumba este desaparece
    if (colider.gameObject.name == "colisionSuperior")
    {
        Destroy(colider.transform.parent.gameObject);
    }

}
```

Utilizando un Character Controler el control se realiza de una forma muy sencilla, pero las colisiones pueden resultar un tanto imprecisas si los objetos se mueven a mucha velocidad, por ejemplo nuestro Mario puede llevar a superponerse con Gumba de forma que llegue a chocar con el objeto colisiónSuperior desde un lateral, cosa que no debería suceder dado que la cápsula del Character Controler no debería penetrar los colisionadores laterales de Gumba.

Podemos modificar las propiedades físicas en un intento de mejorar la detección de colisiones. Edit->Project Settings ->Physics



Para mejorar la detección de la colisión podemos, aumentar el Conctact Offset para que la colisión se detecte antes, aumentar el valor de Solver Iteration Count, también podemos modificar los valores del Character controler Skin Width , disminuyéndolo para que no se superponga con otro colisionador, aunque estos cambios **no solucionan del todo el problema**. Si queremos mejorar en la detección de la colisión deberemos utilizar un RigibBody en vez del Character Controller.

2.5. Utilizando un RigidBody

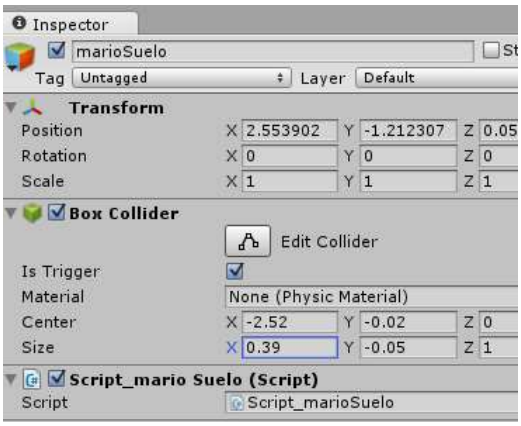
Vamos a ver los cambios a realizar si utilizamos un RigidBody y un Box Collider :

Por un lado la gravedad que anteriormente aplicábamos desde código pasa a ser controlada por el RigidBody, aunque puede que debemos modificar su velocidad, podemos hacerlo desde la ventana de físicas del proyecto como vimos antes (yo he puesto un valor de -100).

El **movimiento** pasa a realizarse con la velocidad del RigidBody :

```
GetComponent<Rigidbody>().velocity = moveDirection * Time.deltaTime;
```

Pero tenemos un problema y es que **no** tenemos la propiedad isGrounded, debemos comprobar si estamos tocado algo con la parte inferior del personaje, para ello añadiremos un objeto en los pies de Mario con un BoxCollider y un Script :



Como podéis ver el Collider está marcado como Is Trigger luego llamará a OnTriggerEnter cuando se produzca una colisión:

```
Script_playerControlRigidBody playerControl;
void Start () {
    playerControl =
         gameObject.GetComponentInParent<Script_playerControlRigidBody>();
  }

void OnTriggerEnter(Collider other)
{
    playerControl.setGrounded(true);//cuando toca algo mario está "grounded"
    if (other.gameObject.name == "colisionSuperior")
    {
        Destroy(other.transform.parent.gameObject);
    }
}
```

En el Script de Mario se ha añadido una variable y una función para controlar cuando está posado (grounded) :

```
private bool isGrounded = true;

public void setGrounded(bool b)
{
    this.isGrounded = b;
    animator.SetInteger("Salto", 0);
}
```

Y se ha modificado ligeramente el código del movimiento en el Script de Mario para reflejar los cambios :

```
void FixedUpdate()
    {
        var horizontal = Input.GetAxis("Horizontal");
        if (this.isGrounded)
        {
            if (horizontal > 0)
            {
                animator.SetInteger("Direccion", 1);
            }
            else if (horizontal < 0)
            {
                animator.SetInteger("Direccion", -1);
            }
            else if (horizontal == 0)
            {
                animator.SetInteger("Direccion", 0);
            }

            moveDirection = new Vector3(Input.GetAxis("Horizontal"), 0, 0);
            moveDirection *= speed;

            if (Input.GetButton("Jump"))
            {
                moveDirection.y = jumpSpeed;
                animator.SetInteger("Salto", 1);
                this.isGrounded = false;
            }
```

```
    //de coordenadas locales a globales
    moveDirection = transform.TransformDirection(moveDirection);
    GetComponent<Rigidbody>().velocity = moveDirection * Time.deltaTime;
    }
}
```