

Primer juego: Pong

Seguimos con nuestro proceso de aprendizaje de Unity. Ahora pasamos a algo más interesante, al trabajo real. Vamos a crear un nuevo proyecto en el que vamos a recrear el mítico juego Pong.

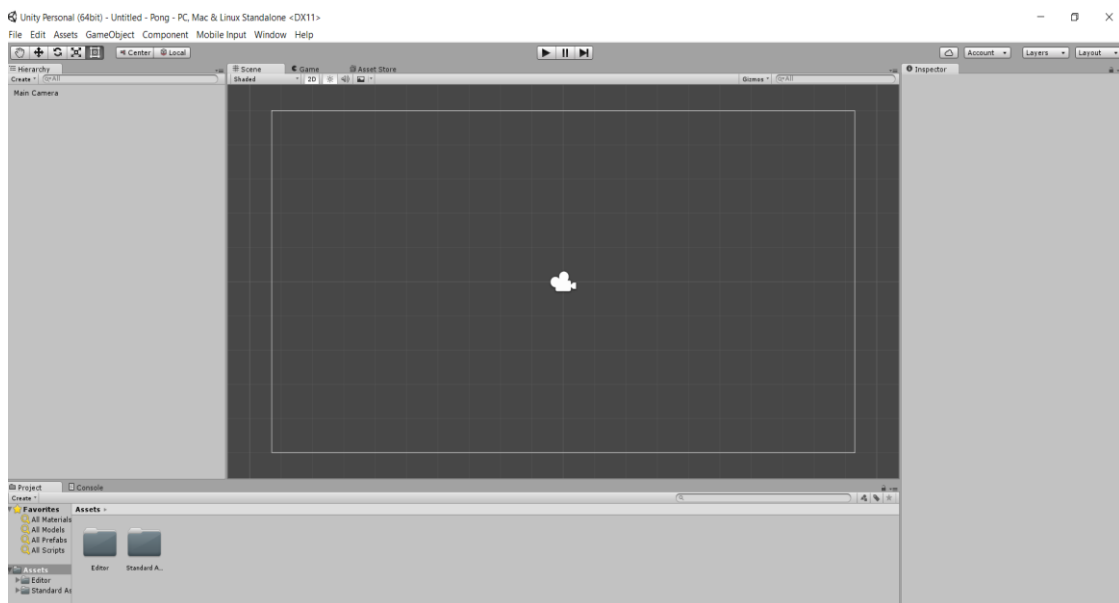
Creando un proyecto

Empecemos por crear un nuevo proyecto. En Windows Unity habrá creado por defecto la carpeta "Unity Project" en la carpeta de mis documentos. Puedes guardar ahí los tutoriales si quieres, pero te recomiendo que crees una carpeta específica y ahí guardes el material con el que trabajemos.

Para crear un nuevo proyecto vamos a File -> New Project, esto hará que se muestre el cuadro de dialogo "Create New Project".

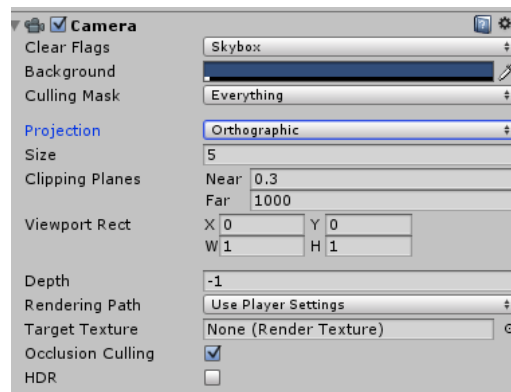
Cuando hagas esto Unity se reiniciará, creará la estructura del proyecto en la carpeta especificada e importará los paquetes seleccionados al proyecto. Una vez termine se mostrará una escena en blanco con una cámara.

Como verás en la vista de escena, tenemos una escena en blanco ya que no hay nada en ella, pero Unity nos ayuda y ha incluido una cámara inicial por nosotros, que podemos ver en la vista de escena y en la jerarquía.



Al elegir el proyecto en 2D la cámara inicialmente muestra una vista ortogonal, a diferencia del 3D donde la vista es en perspectiva.

Si seleccionamos Main Camera en la jerarquía Hierarchy, veremos sus propiedades:



Selecciona el background y cambia el color a negro, será el color de fondo del juego.

También verás los assets estándar en la vista de proyecto, listos para ser usados.

La primera escena

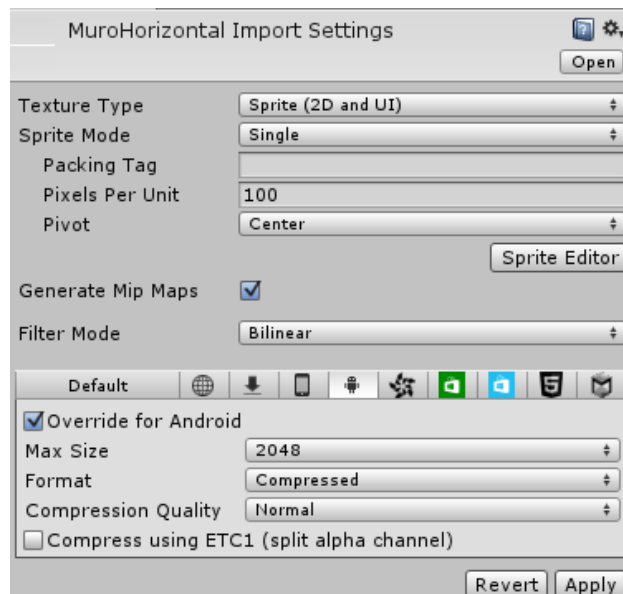
Unity ha creado nuestra primera escena automáticamente al crear el proyecto. Si quieres crear una escena nueva, sencillamente ve a "File -> New Scene".

Esta nueva escena es un espacio sin título, realmente necesitamos guardarlo. Para ello vamos a "File -> Save Scene" y guarda la escena en algún lugar en la carpeta de asset para este proyecto. La escena aparecerá en la vista de proyecto una vez hagas esto. Puedes llamar la escena "escena1". Con el proyecto creado y nuestra primera escena delante, estamos listos para empezar a construir nuestro primer nivel.

Añadiendo recursos


Vamos a añadir las imágenes necesarias para realizar el juego, para ello seleccionamos en el menú Assets -> Import New Assets, donde podremos buscar el recurso que deseamos incluir en nuestro proyecto. Una vez añadido aparecerá en Projects->Assets y lo podremos incluir en la escena.

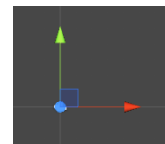
Al insertar una imagen (Sprite) podemos elegir algunos parámetros:



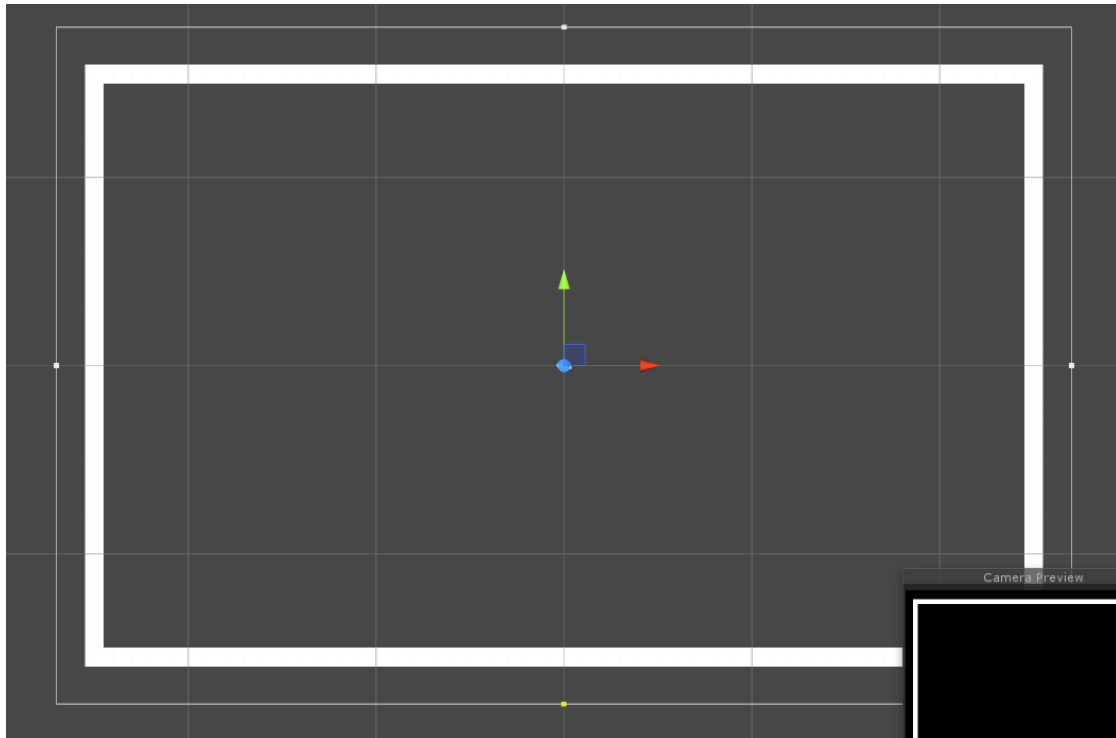
Por ahora no cambiaremos ninguno de los valores, en la parte inferior puede modificarse la calidad de la imagen dependiendo de la plataforma para la que estemos desarrollando.

El modo de filtro y de formato se puede utilizar para decidir entre la calidad y el rendimiento. Cambiaremos los **píxeles por unidad** a 1 significa que 1 x 1 píxeles caben en una unidad en el mundo del juego. Vamos a utilizar este valor para todas nuestras texturas, si al cambiar este valor los sprites aparecen muy grandes podemos cambiar el Size de la Main Camera, indicando un valor mayor.

Podemos arrastrar los assets desde la ventana de Project hacia la ventana de la escena y entonces aparecerán en la ventana Hierachy (jerarquía), vamos a incorporar dos muros verticales y 2 horizontales) y los coloremos en forma de cuadrado, para ello podemos utilizar la herramienta Translate Tools  y al seleccionar el asset dentro de la escena podremos moverlo con las flechas:



El aspecto, una vez introducidos los muros (2 verticales y 2 horizontales) sería similar a este:



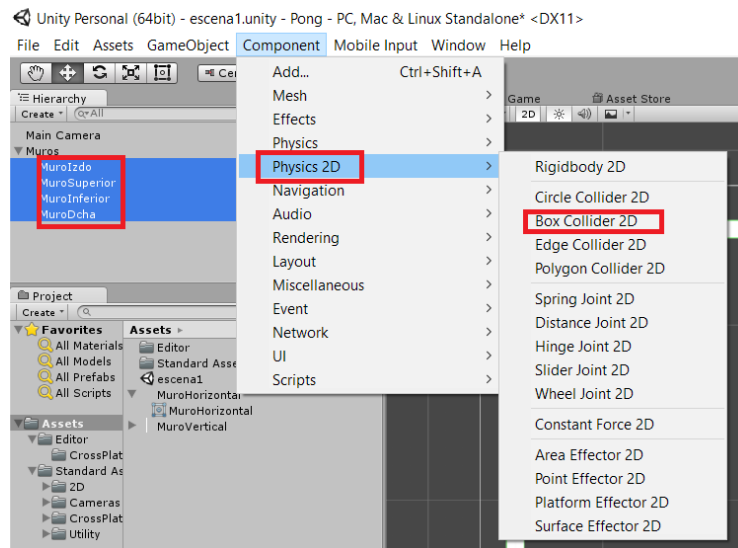
Desde la vista de jerarquía pulsando con el botón derecho podemos renombrar los elementos de la escena e incluso organizarlos en carpetas.

Nota: realmente no pueden crearse carpetas, pero si un GameObject vacío donde introduciremos (arrastrando) los objetos que deseemos.

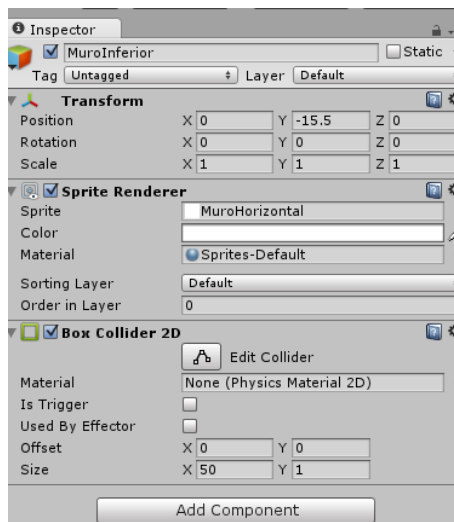


Añadiendo comportamiento físico

Los muros en este momento son solo imágenes dentro de la escena, vamos a incorporarles un “comportamiento” físico de forma que reaccionen ante colisiones. Seleccionamos los muros y añadimos un Box Collider 2D.



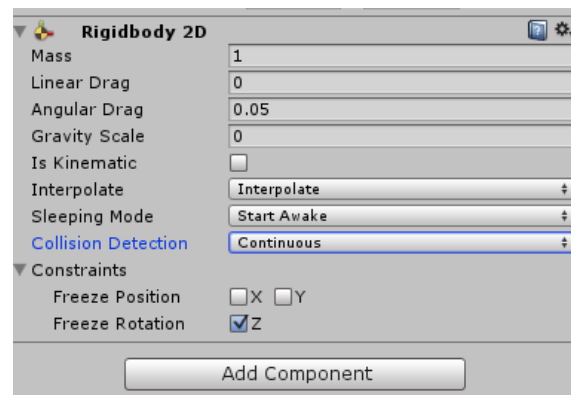
Si seleccionamos un componente en la jerarquía en la ventana del inspector aparecerá el Box Collider:



Si seleccionamos un muro en la vista de escena veremos un rectángulo verde que los rodea, ese es el box collider (no será visible durante el juego).

Insertamos las dos raquetas, les añadimos igual que a los muros el Box Collider 2D. El jugador también debe ser capaz de mover la raqueta arriba y hacia abajo, pero las raquetas deben dejar de moverse cuando chocan con un muro. Lo que requeriría una complicada solución matemática se resuelve fácilmente con un **Rigidbody**, el cual ajusta la posición de un objeto a lo que es físicamente correcto (lo que implica colisiones, gravedad...). Como regla general, todo elemento físico que se mueve a través del mundo del juego se necesita un Rigidbody.

Para agregar un Rigidbody a nuestros Raquetas sólo seleccionamos a ambas en la Jerarquía, a continuación, pulsamos *Component -> Add -> Physics 2D -> Rigidbody 2D*. Luego modificamos el Rigidbody 2D para desactivar la gravedad (porque no hay gravedad en un juego de mesa), seleccionamos Freeze Rotación Z (las raquetas nunca deben girar) y establecer la detección de colisiones en Continuo y habilitar la interpolación de forma que la física sea tan exacta como sea posible:



Movimiento de la raqueta

Vamos a asegurarnos de que los jugadores pueden mover sus raquetas. Ese tipo de comportamiento personalizado por lo general requiere de **Scripts**. Con todavía seleccionan ambas raquetas, haremos clic en *Component -> Add -> New script* (es posible que tengas que hacer scroll), el nombre MoverRaqueta y seleccione CSharp como lenguaje.

Después podemos hacer doble clic en la secuencia de comandos en el Área del Proyecto a fin de abrirlo.

Así es nuestro script actualmente:

```
using UnityEngine;
using System.Collections;

public class MoverRaqueta : MonoBehaviour {

    // Use this for initialization
    void Start () {

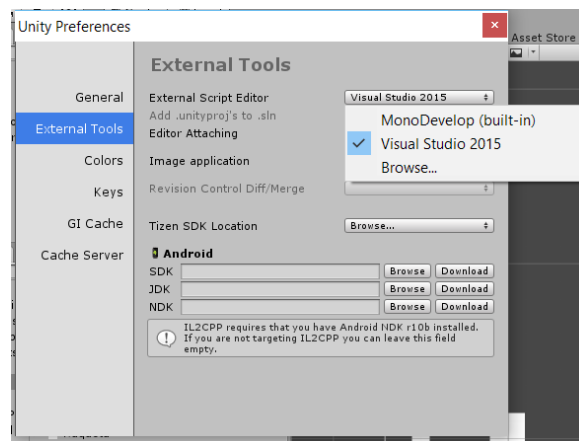
    }

    // Update is called once per frame
    void Update () {

    }

}
```

En Edit -> Preferences podemos seleccionar el IDE que por defecto abrirá nuestros Scripts



La función de Start se llama automáticamente al iniciar el juego. La función de Update se llama automáticamente una y otra vez, aproximadamente 60 veces por segundo.

Pero hay otra función de actualización, se llama FixedUpdate, la cual también se llama una y otra vez, pero en un intervalo de tiempo fijo. Las físicas de Unity se calculan en el mismo intervalo de tiempo exacto, por lo que por lo general es una buena idea usar FixedUpdate al hacer cosas físicas para conseguir más precisión.

Bueno por lo que vamos a quitar las funciones de Start y Update y crear una función FixedUpdate en su lugar:

```
using UnityEngine;
using System.Collections;

public class MoverRaqueta : MonoBehaviour {

    void FixedUpdate () {

    }

}
```

Las raquetas tienen un componente Rigidbody y vamos a utilizar la propiedad de velocidad Rigidbody para el movimiento. La velocidad es siempre la dirección de movimiento multiplicado por la velocidad.

La dirección será un Vector2 con el componente x (dirección horizontal) y el componente y (dirección vertical).

Las raquetas sólo deben moverse hacia arriba y hacia abajo, lo que significa que el componente x siempre será 0 y el componente y será 1 para arriba, -1 hacia abajo o 0 para no avanzar.

El valor de y depende de la entrada del usuario. Podríamos comprobar todo tipo de pulsaciones de teclas (WSAD, flechas, palos gamepad, etc.), o podríamos simplemente utilizar la función de la Unity.GetAxisRaw:

```
void FixedUpdate () {
    float v = Input.GetAxisRaw("Vertical");
}
```

Ahora podemos usar GetComponent para acceder al componente Rigidbody de la raqueta y establecer su velocidad. También vamos a añadir una variable de la velocidad en nuestro Script, para controlar la velocidad de movimiento de la raqueta:

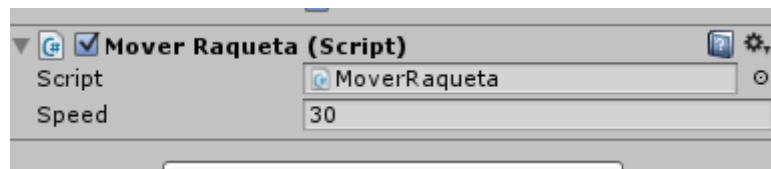

```

public class MoverRaqueta : MonoBehaviour {
    public float speed = 30;

    void FixedUpdate () {
        float v = Input.GetAxisRaw("Vertical");
        GetComponent<Rigidbody2D>().velocity = new Vector2(0, v);
    }
}

```

La variable que acabamos de crear es accesible desde el inspector:



Hacemos uso de la variable, la velocidad de la raqueta se modifica en función del valor de la variable speed:

```

void FixedUpdate () {
    float v = Input.GetAxisRaw("Vertical");
    GetComponent<Rigidbody2D>().velocity = new Vector2(0, v) * speed;
}

```

Hemos establecido velocidad como la dirección multiplicado por la velocidad, que es exactamente la definición de velocidad. Si salvamos la secuencia de comandos y pulsamos Reproducir ahora podemos mover las raquetas (con los cursores o las teclas w y s).

Sólo hay un problema, no podemos mover las raquetas por separado. Vamos a crear una nueva variable de eje para que podamos cambiar el eje de entrada en el Inspector:

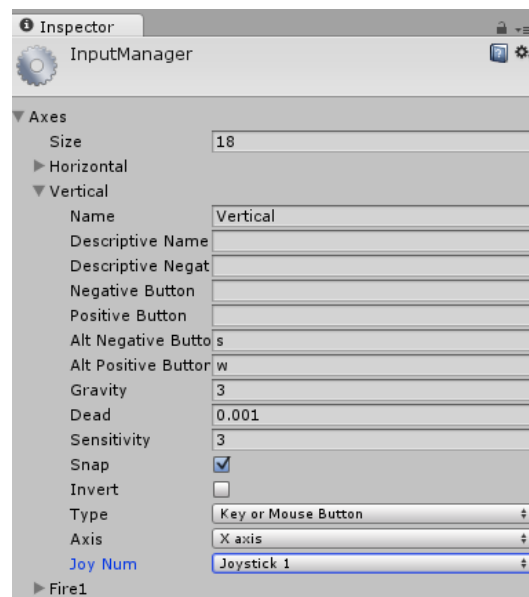
```

public float speed = 30;
public string axis = "Vertical";

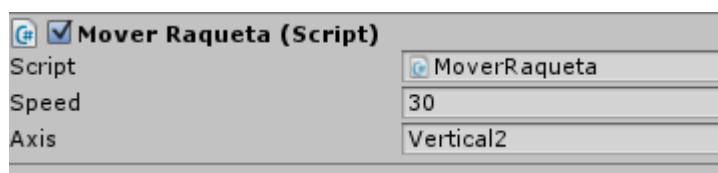
void FixedUpdate()
{
    float v = Input.GetAxisRaw(axis);
    GetComponent<Rigidbody2D>().velocity = new Vector2(0, v) * speed;
}

```

Seleccionamos Edit -> Project Settings -> Input, aquí podemos modificar el eje vertical actual de modo que sólo utilice las teclas W y S. También vamos a hacer que use solamente Joystick 1:



Ahora incrementamos Size en uno (19) para introducir un nuevo Eje (Axis), a continuación seleccionaremos una de las raquetas en la jerarquía y cambiaremos el eje en el script :



Ahora las dos raquetas se moverán de forma independiente.

▼ Axes

Size

19

► Horizontal

► Vertical

► Fire1

► Fire2

► Fire3

► Jump

► Mouse X

► Mouse Y

► Mouse ScrollWheel

► Horizontal

► Vertical

► Fire1

► Fire2

► Fire3

► Jump

► Submit

► Submit

▼ Vertical2

Name

Vertical2

Descriptive Name

Descriptive Negative Name

Negative Button

down

Positive Button

up

Alt Negative Button

Alt Positive Button

Gravity

3

Dead

0.001

Sensitivity

3

Snap

☒

Invert

☐

Type

Key or Mouse Button

Axis

X axis

Joy Num

Joystick 2

► Cancel

Alternativa

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MoverRaqueta : MonoBehaviour
6  {
7
8      public float speed = 30;
9      public bool raqueta1;
10
11     void FixedUpdate()
12     {
13         float v;
14         if (raqueta1)
15         {
16             v = Input.GetAxisRaw("Vertical");
17         }
18         else
19         {
20             v = Input.GetAxisRaw("Vertical2");
21         }
22         GetComponent<Rigidbody2D>().velocity = new Vector2(0, v) * speed;
23     }
24 }
25
26
27
```

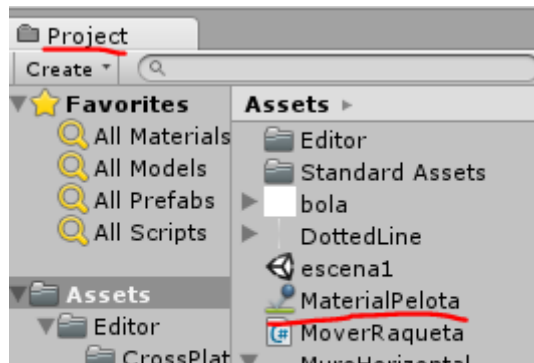
Añadiendo la pelota

Añadimos el fichero bola.png igual que hicimos con los muros (recordar el Pixel per Unit a 1), lo incorporamos en el centro de la escena y le asignamos un Box Collider 2D.

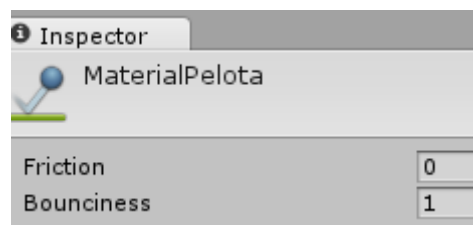
Nuestra bola se sabe que va a rebotar en las paredes. Por ejemplo, cuando se dirige directamente hacia una pared debe rebotar en la dirección exactamente opuesta. Cuando se choca en un ángulo de 45° con una pared, debe rebotar en un ángulo de 45° y así sucesivamente.

Parece necesaria una complicada codificación matemática a través de scripts. Pero dejaremos que Unity se ocupe de ello asignando un material físico al Collider de la pelota que hará que rebote.

Hacemos clic derecho en la ventana de Proyecto y seleccionamos Create -> Physics2D Material que vamos a nombrar MaterialPelota :

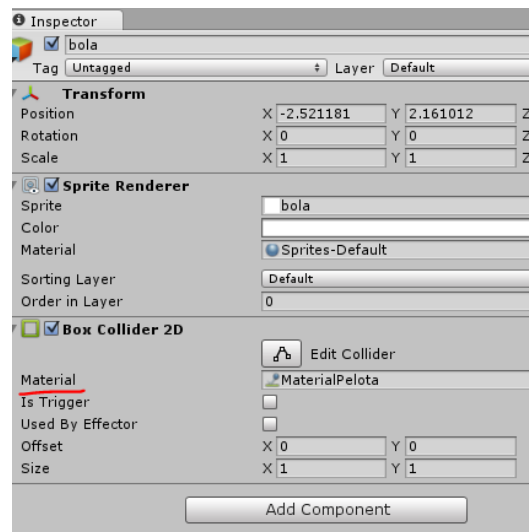


Ajustamos sus valores para que rebote:



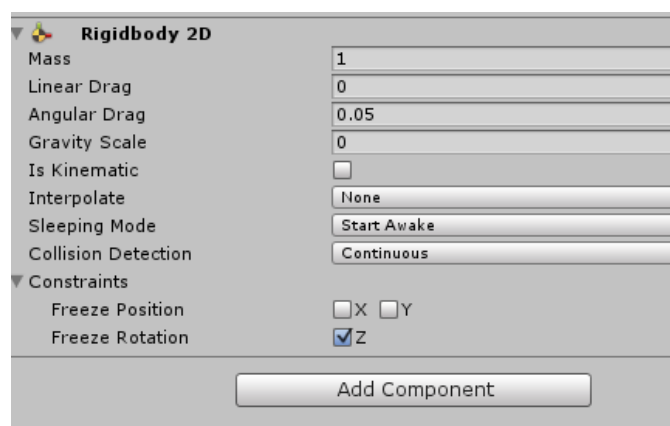
- **Bounciness** indica la fuerza con la que rebota (valores entre 0 y 1).
- **Friction** indica la fricción que recibe al moverse, cuanto más fricción tenderá a pararse más rápidamente.

Arrastraremos el material desde la ventana de proyecto hasta la bola en la ventana de jerarquía, de esta forma queda asignado el material a la pelota.



Para que la bola se mueva debemos añadirle un Rigidbody, la seleccionamos en la vista de jerarquía y pulsamos Component->Physics 2D->Rigidbody 2D.

Modificamos algunos de sus valores, ya que no nos interesa usar la gravedad, la detección de colisiones la ajustamos a **continua** (para mejorar precisión y le impedimos el giro en el eje z.



- **Is kinematic** : si marcamos esta opción el motor de físicas ignorará el objeto, deberemos ser nosotros a través de un script y el componente transform los que movamos el objeto.

Le asignamos un script a la pelota igual que hicimos con los muros. En este caso solo codificamos la función Start, en el Update no hacemos nada,

```
public class Ball : MonoBehaviour {
    public float speed = 30;

    void Start() {
        // Impulso inicial
        GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
    }
}
```

En el inicio se le aplica una velocidad hacia la derecha. Con esto la pelota ya se mueve pero el rebote siempre será en horizontal, tenemos que conseguir aplica un **rebote más realista** dependiendo de la zona de la raqueta que golpee.

```
float reboteY(Vector2 bolaPos, Vector2 raquetaPos,
              float alturaRaqueta)
{
    // || 1 <- parte superior de la raqueta
    // ||
    // || 0 <- parte media de la raqueta
    // ||
    // || -1 <- parte inferior de la raqueta
    return (bolaPos.y - raquetaPos.y) / alturaRaqueta;
}
```

```
void OnCollisionEnter2D(Collision2D col)
{
    // col es el objeto que recibe la colisión de la pelota
    if (col.gameObject.name == "RaquetaIzda")
    {
        // Calculamos la dirección de rebote
        float y = reboteY(
            transform.position, // posición de la pelota
            col.transform.position, // posición de la raqueta
            col.collider.bounds.size.y); // tamaño de la raqueta
    }
}
```

```

// Calculamos la dirección, lo normalizamos para que el tamaño
//del vector sea 1 al chocar con la raqueta izda
//la dirección x será 1 (hacia la derecha)
Vector2 dir = new Vector2(1, y).normalized;

GetComponent<Rigidbody2D>().velocity = dir * speed;
}

// golpea la raqueta derecha
if (col.gameObject.name == "RaquetaDcha")
{
    // Calculate hit Factor
    float y = reboteY(transform.position,
                      col.transform.position,
                      col.collider.bounds.size.y);

    // en este caso se mueve hacia la izda (x=-1)
    Vector2 dir = new Vector2(-1, y).normalized;

    // se aplica la velocidad
    GetComponent<Rigidbody2D>().velocity = dir * speed;
}
}

```

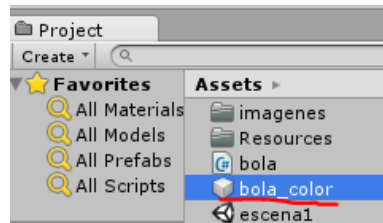
¿Qué mejoras podemos introducir al juego? Es cuestión de imaginación...

- Incorporar el sonido de la pelota en el choque
- Acelerar la pelota a lo largo del tiempo
- Añadir puntuaciones y que se inicie el juego cada vez que un jugador pierda
- Establecer unas condiciones de victoria y reiniciar la partida
- Incorporar un menú de juego
- ...

Prefabs

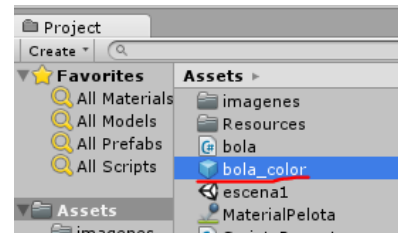
Los prefabs o prefabricados son assets que podemos usar como si fuera un molde para un objeto para ser utilizado en distintas escenas o de forma repetida, la ventaja de los prefabs es que cualquier cambio en el mismo se replica en todas sus instancias.

Creamos prefab vacío, botón derecho del ratón Create -> Prefab

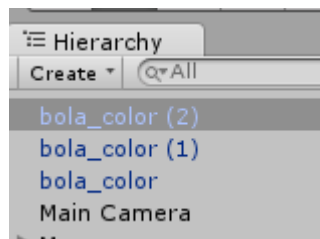


Arrastramos la bola desde la vista de jerarquía sobre el prefab :

Como vemos el icono ha cambiado y si pulsamos sobre el prefab vemos todos sus componentes en el inspector.



Ahora podemos arrastrar tantas veces como queramos el prefab desde la ventana de proyecto hacia la escena. Aparecerán en la jerarquía con un color azulado (indicando que son prefabs)



Podemos renombrarlos a nuestro gusto, y cualquier cambio en las propiedades del prefab dentro de la ventana proyecto se replicarán en los objetos de la escena.

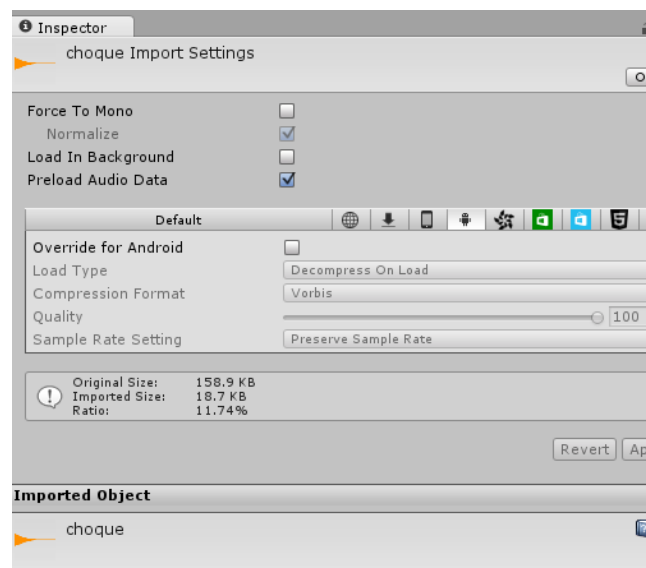
Sonido

El sonido y la música son una parte muy importante del conjunto del videojuego, donde estamos mezclando acción, imagen, sonido e interacción y donde el conjunto debe ser armonioso.

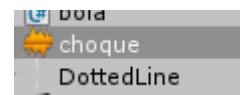
En juegos de cierta envergadura la importancia de la banda sonora se asemeja a la de las producciones cinematográficas, unos efectos realistas harán que nuestro juego sea mucho más atrayente e inmersivo.

Unity soporta varios formatos de audio como son el wav, ogg y mp3, para ello vamos a necesitar una fuente de sonido **AudioSource** y alguien que escuche un **AudioListener** (por defecto nuestra Main Camera ya tiene un AudioListener).

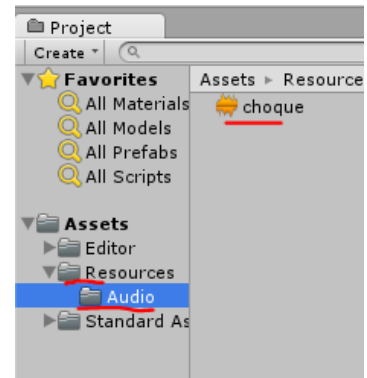
Primero debemos importar los archivos de sonido dentro del editor, igual que con las imágenes será a través de Assets -> Import New Asset.



Ya la tenemos disponible en nuestros Assets quizás sería buena idea crear una carpeta para organizar, por un lado, el audio, los sprites, etc...



Si queremos cargar los sonidos desde el código es necesario guardarlos en una carpeta llamada Resources dentro de los Assets:



Para reproducir el sonido realizamos una llamada a la función:

AudioSource.PlayClipAtPoint(

```
Referencia_objeto_sonido, //sonido a reproducir, AudioClip  
Vector3_position, //posición de la que parte el audio  
Volumen));
```

Para ello puedo declarar en mi Script una variable como la siguiente :

```
public float speed = 30;  
public AudioClip choque;//declaramos el AudioClip como atributo del  
script  
// Use this for initialization  
void Start () {  
    GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;  
  
    //Cargo el sonido desde la jerarquía de mi escena  
    //(la ruta depende de tu organización (no lleva extensión)  
    choque = (AudioClip)Resources.Load("Audio/choque");  
}
```

Lo reproducimos cada vez que haya una colisión, utilizamos la posición del objeto con el que chocamos.

```
void OnCollisionEnter2D(Collision2D col)
{
    AudioSource.PlayClipAtPoint(choque,col.transform.position,5);
    ...
}
```

También se puede reproducir utilizando la variable de tipo AudioClip y pasándosela a un objeto de tipo AudioSource. Por ejemplo:

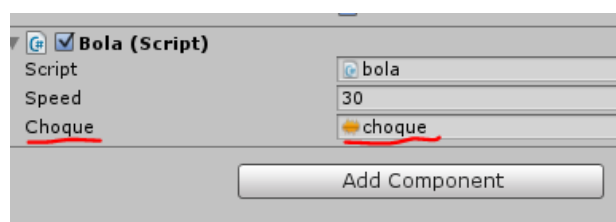
```
AudioSource audio = GetComponent(); //tenemos que obtener la referencia a un componente de tipo AudioSource
```

```
audio.PlayOneShot(choque);
```

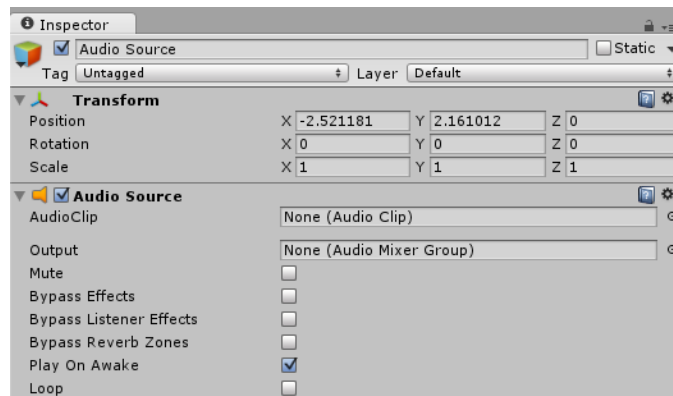
Si no queremos cargar el sonido desde el script mediante :

```
choque = (AudioClip)Resources.Load("Audio/choque");
```

Podemos arrastrar el archivo de sonido a la variable del script, el efecto sería el mismo.



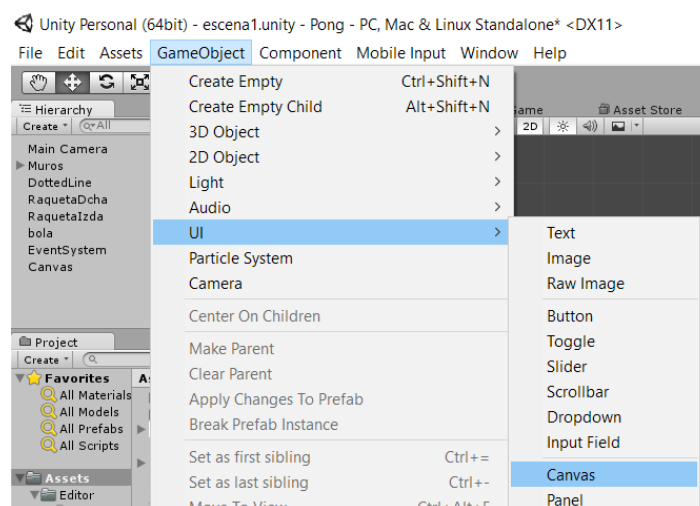
Para la música, por ejemplo, un tema que suena de fondo durante la escena, se puede utilizar el objeto AudioSource, GameObject -> Audio -> AudioSource



Entre sus propiedades están:

- AudioClip : fichero de audio a reproducir
- Play On Awake : se reproduce nada más instanciarse
- Loop: se reproduce en bucle.

Interfaz de usuario

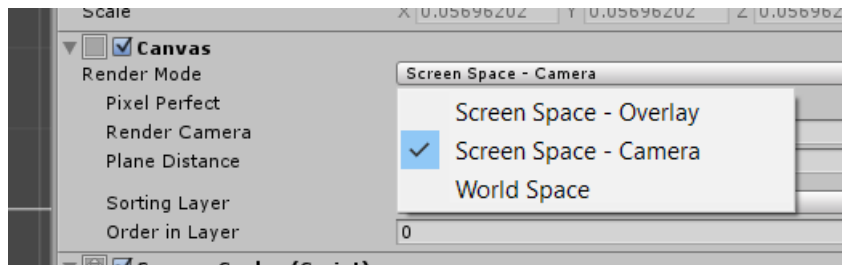


Lo primero que tenemos que crear es un objeto Canvas.

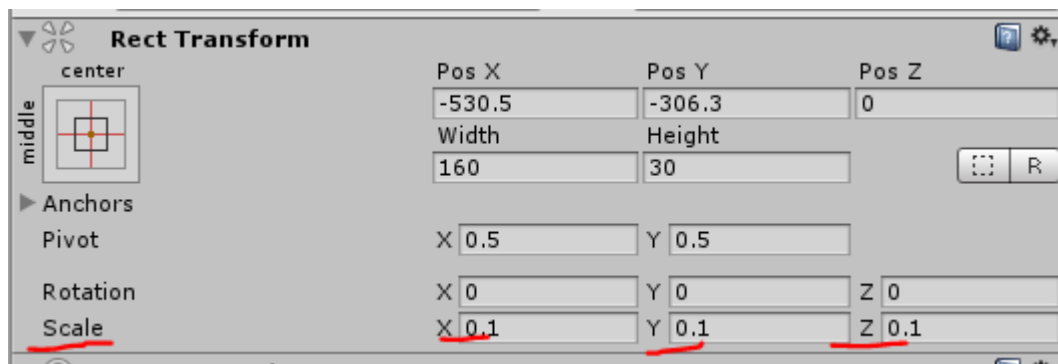
Con el objeto canvas seleccionado podemos insertar nuevos objetos de tipo UI, texto, imagen, botones.... que serán hijos del objeto canvas.

Existen 3 formas de mostrar el canvas, aparecen en Render Mode, para más información.

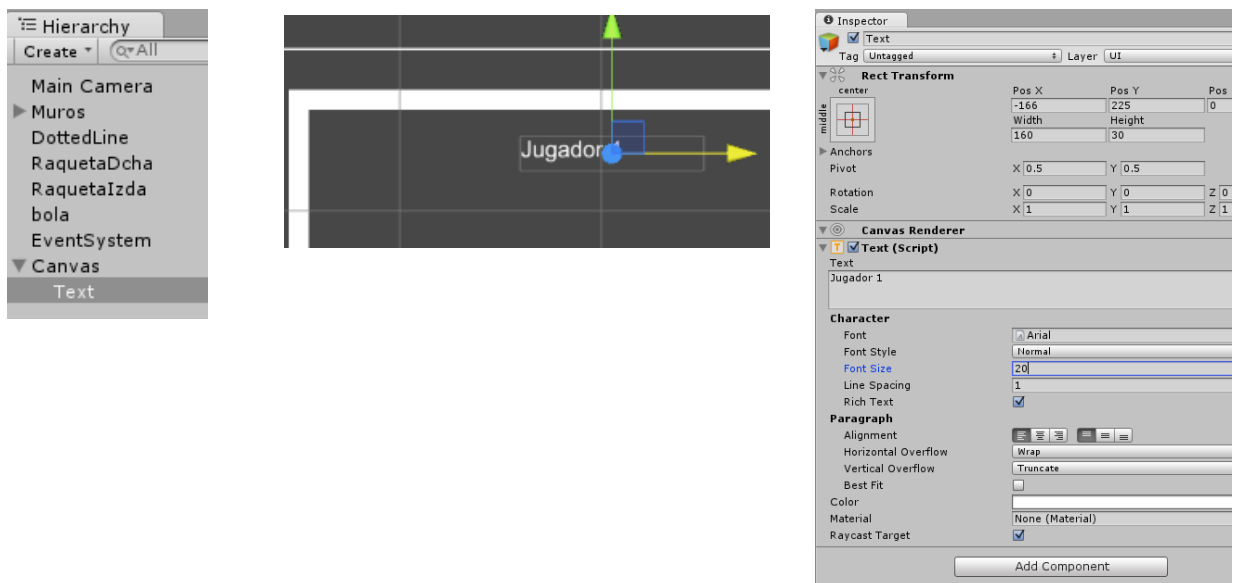
Vamos a elegir **Screen Space- Camera** y seleccionaremos Main Camera en el apartado Render Camera.



Si un objeto aparece (texto, botón...) con unas proporciones desmesuradas es recomendable ajustar los valores de la escala, antes que el tamaño en anchura y altura del objeto:



Insertamos un objeto de tipo texto dentro del canvas (GameObject -> UI -> Text), así quedaría en la Jerarquía, la escena y el inspector:



Vamos a insertar cuatro textos uno para el jugador 1, otro para el jugador 2 y dos más que indiquen la puntuación de cada uno.

Para acceder al texto desde el script (por ejemplo, para cambiar la puntuación) debemos incluir:

```
using UnityEngine.UI;
```

```
using System;
```

Cuando uno de los muros verticales reciba un impacto, el jugador del lado contrario incrementará su puntuación:

```
//col es el objeto que recibe la colisión de la pelota
if (col.gameObject.name == "MuroIzdo")
{
    GameObject scoreJ1 = GameObject.Find("ScoreJ2");
    Text scoreText = scoreJ1.GetComponent<Text>();
    int newScore = Int32.Parse(scoreText.text) + 1;
    scoreText.text = newScore.ToString();
}
```

Gestión de la partida

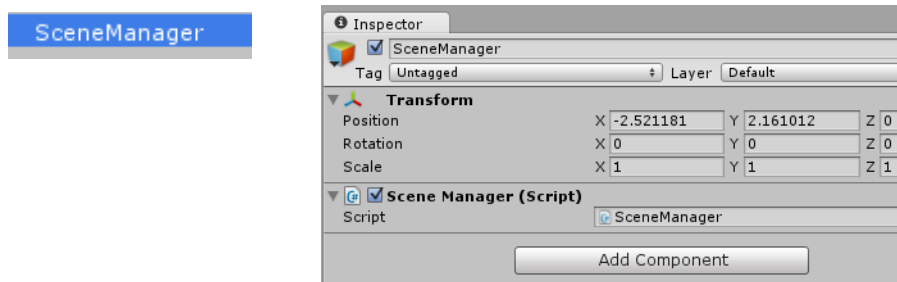
Hasta ahora hemos utilizado una única escena donde desarrollamos la partida, pero el jugador espera un interfaz mínimo y una lógica dentro del juego, es decir, que haya una condición/es de victoria o derrota, que pueda reiniciar la partida o salir, todo aquello que hace que un juego sea agradable de usar y a poder ser adictivo.

Inicio

Normalmente se muestra un menú de inicio donde se permite elegir entre varias opciones, iniciar el juego, continuar una partida, opciones del juego, salir, etc..

Juego

Durante el juego deberemos controlar, distintos aspectos del mismo, por ejemplo si este está pausado, acabado, etc.. es conveniente utilizar un GameObject vacío, le podemos llamar **Scene Manager** o **GameManager**, al cual le asignamos un script que se encargue de esta tarea. Aquí vemos su vista de jerarquía y de inspector:



Dentro de este script podemos llevar el recuento de puntos, creación de nuevos componentes vía código y todo lo relacionado con la lógica del juego.

Actualización: Unity introdujo una clase SceneManager, por lo que nuestro script entraría en conflicto con dicha clase, para ello podemos llamar a nuestro objeto **GameController** y crear un script llamado GameControllerScript asignado al mismo.

Fin

Al finalizar la partida se muestra un menú que nos pregunta si deseamos salir o volver a jugar. Por supuesto las opciones son infinitas puedes mostrar las opciones que tú quieras, una característica de los videojuegos es su gran libertad.

Vamos a realizar un menú de inicio para el cual crearemos una nueva escena llamada **inicio** y la que mostraremos una UI con un botón que al ser pulsado cargará la escena del juego.

Crear nueva escena

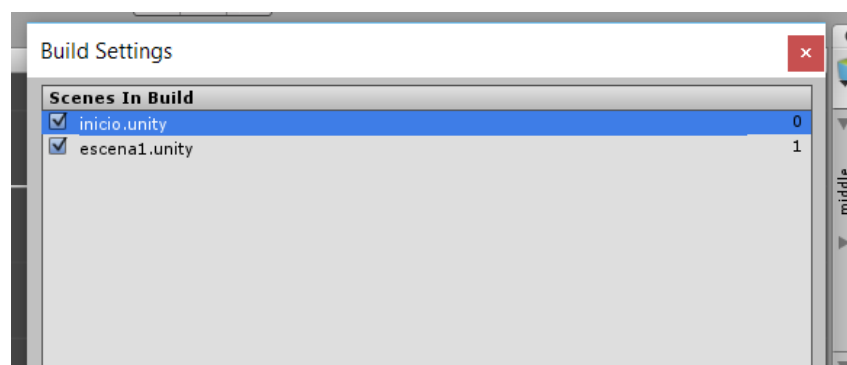
Para crear una nueva escena pulsamos File -> New Scene y nos aparecerá una escena vacía, la guardamos pulsando File -> Save Scene le damos un nombre descriptivo (inicio por ejemplo).

Para cargar una nueva escena desde el código simplemente llamamos a la función:

~~Application.LoadLevel~~("nombre_escena");//en desuso

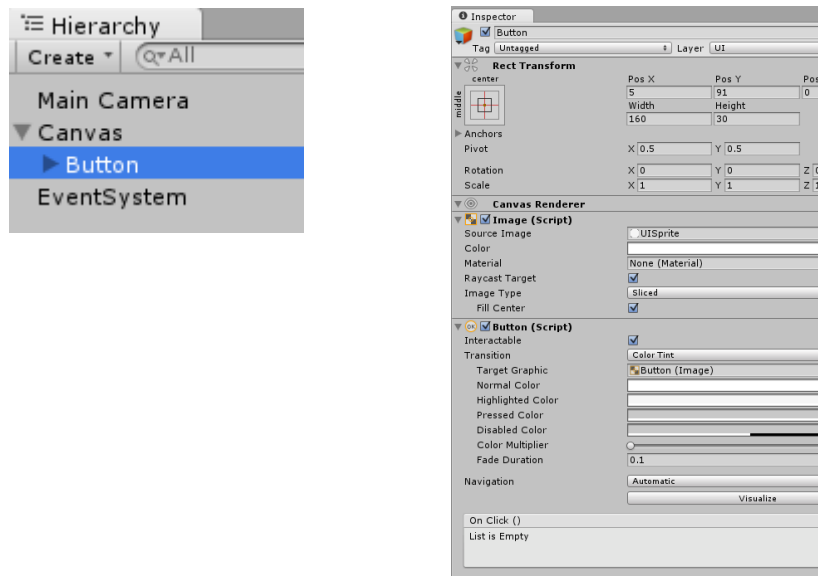
SceneManager.LoadScene("nombre_escena");

Para que la escena pueda ser cargada debe incluirse en el Build, pulsamos **File -> Build Settings**, se pueden arrastrar desde la vista de Proyecto:



Crear una UI con botón

Creamos un canvas como hemos visto en puntos anteriores, le introducimos



un botón a través de GameObject -> UI -> Button :

Lo colocamos en el centro de la escena y le cambiamos el texto (desplegamos en la jerarquía hasta encontrar el componente text del botón).



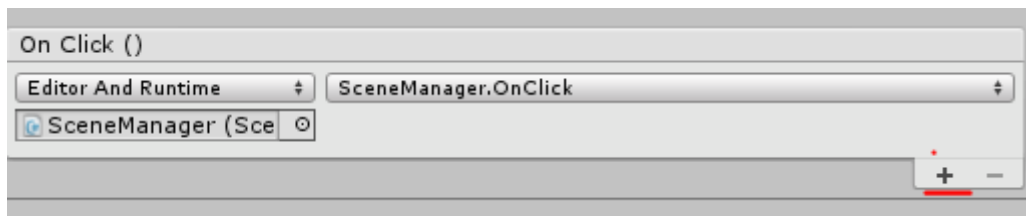
Vamos a utilizar un objeto con un script para recoger el click del ratón, creamos un objeto vacío llamado SceneManager al que asignamos un script llama SceneManager como se vio anteriormente. Dentro del script declaramos un **método público que devuelve void** y que será el encargado de carga la escena del juego:

```

public void OnClick()
{
    Application.LoadLevel("escena1");
}

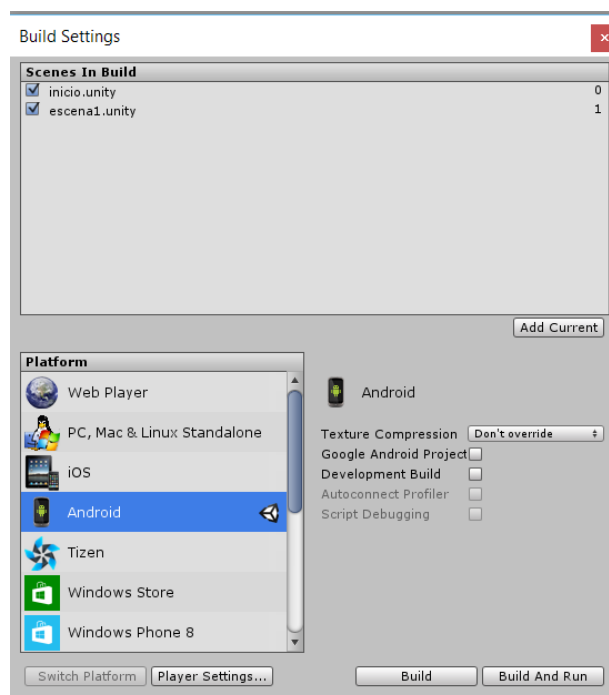
```

Seleccionamos el botón y le asignamos el objeto SceneManager y la función OnClick de este modo la función será llamada cada vez que hagamos click en el botón.



Generando el APK

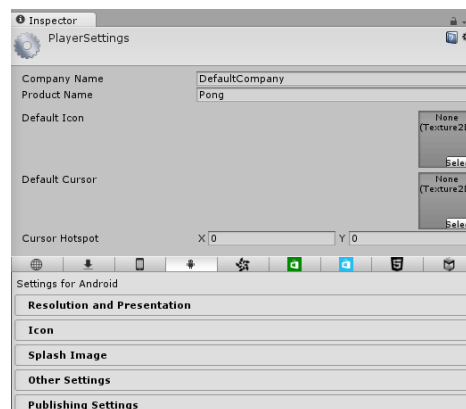
Para compilar todo el proyecto y generar el APK ya sea para probarlo en un dispositivo Android o subirlo a Google Play, pulsamos **File -> Build Settings** :



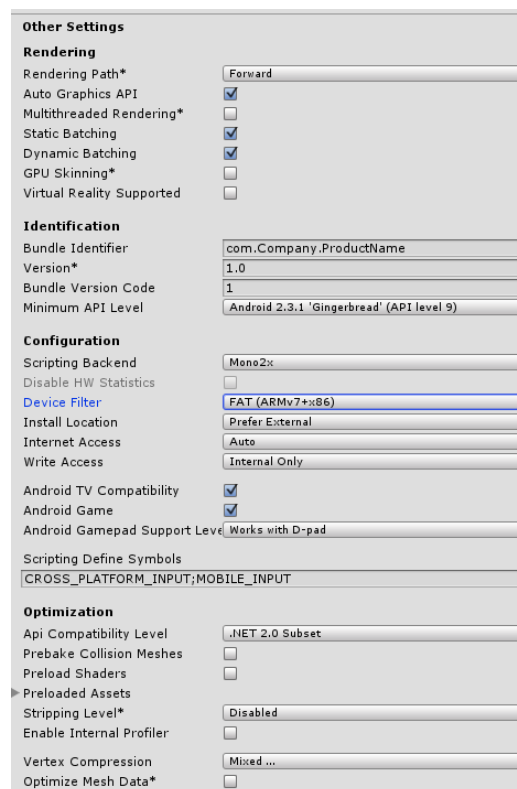
Elegimos Android y pulsamos sobre Player Settings

En esta pantalla debemos configurar los datos de nuestra aplicación

- Company Name : nombre de nuestra compañía de desarrollo
- Product Name : nombre de nuestro juego.
- Default Icon : icono de nuestro juego.
- Default Cursor : cursor por defecto

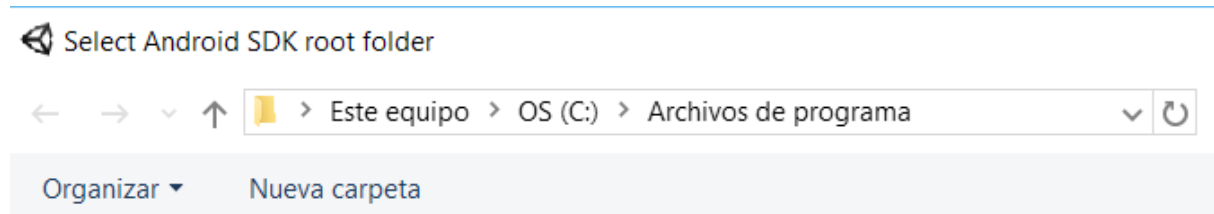


Muy importantes son los datos necesarios en el apartado **Other Settings**:



Si os fijáis son algunos de los datos que se incluían en el Android Manifest, como el identificador Bundle identifier que deberemos rellenar obligatoriamente (no vale con el que muestra por defecto), número de versión, nivel de API.

Una vez pulsamos en Build, empezará a compilar el proyecto y nos pedirá la ubicación del SDK de Android:



Y finalizará con la creación de nuestro APK que podemos probar en el teléfono o en un emulador.