Apuntes de Unity: Flappy Bird

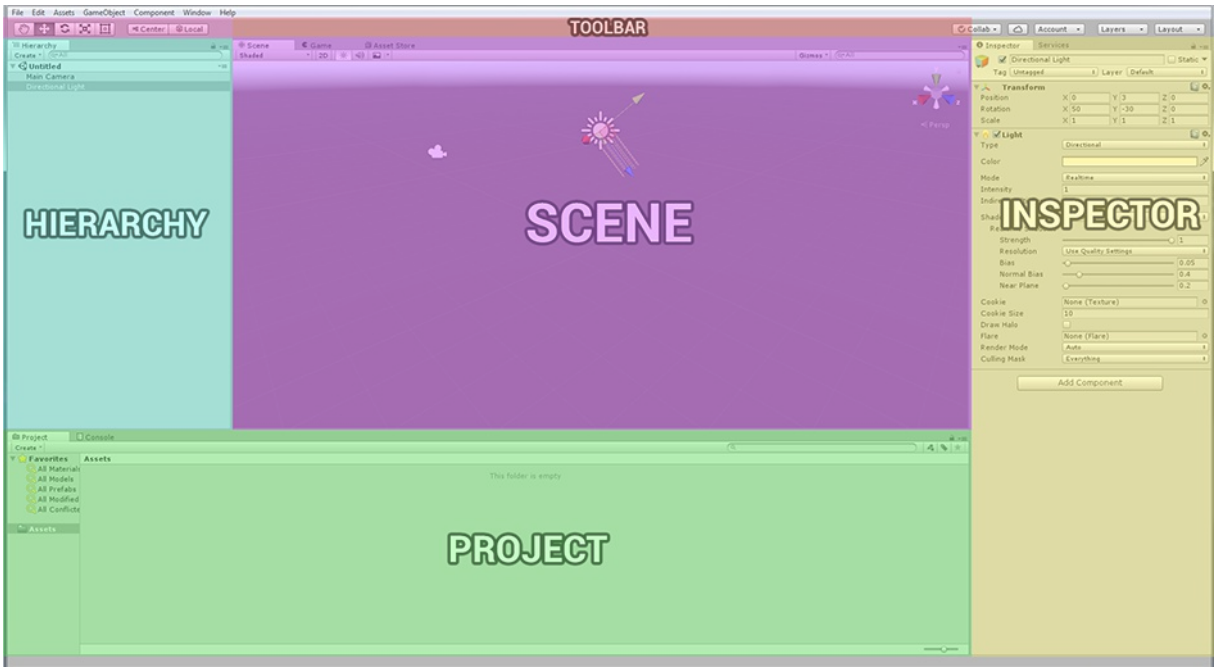
> Autor: Miguel Medina Ballesteros (Maximetinu)
> Contacto: maximetinu (at) gmail (dot) com

En esta guía veremos paso a paso como crear y programar nuestro primer videojuego en Unity: un clon del Flappy Bird.

Introducción

1. Descargar Unity. También te descarga Microsoft Visual Studio listo para funcionar con Unity
    - En caso de tener Microsoft Visual Studio previamente instalado, modificar su instalación para que soporte Unity. `En Windows: Menú Inicio > Buscar > Visual Studio Installer > Visual Studio Community 2017 > More > Modify > Game development with Unity > Modify`
2. Crear cuenta en Unity
3. Crear nuevo proyecto con el nombe `NombreApellido1Apellido2_FlappyBird`
    - Seleccionar Template 2D
    -

Ventanas principales de Unity

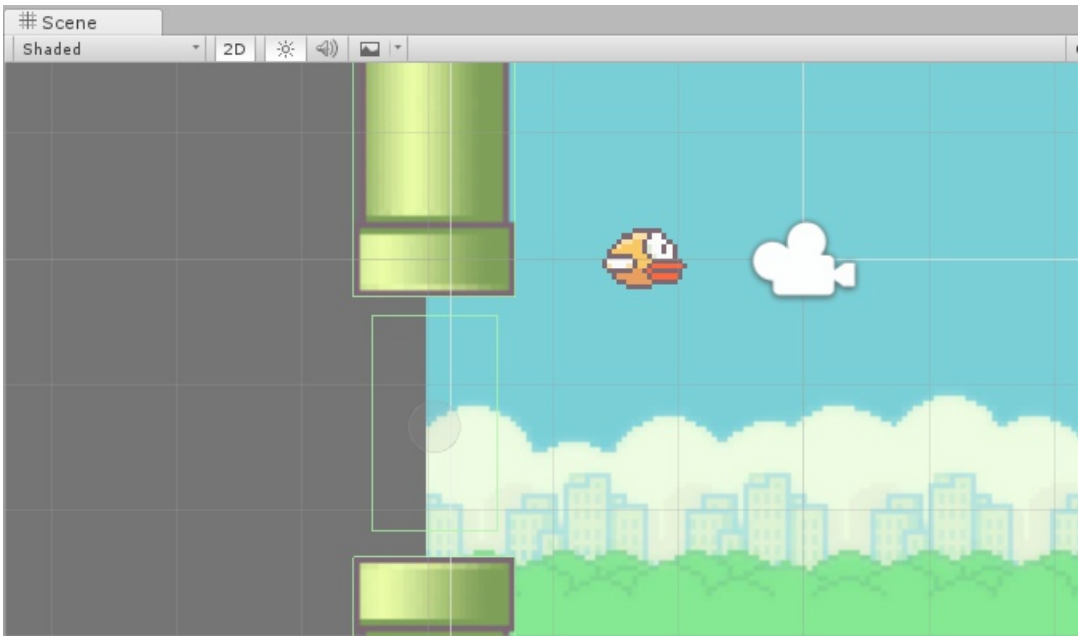


En la screenshot superior se muestran las 4 ventanas principales de Unity. En la lista de abajo se describe qué es y para qué sirve cada ventana.

En la screenshot salen solo 4 ventanas, pero faltarían 2 importantes: Console, que está minimizada junto con proyecto, y Game, que está minimizada junto con escena.

Si en algún momento pierdes alguna pantalla o te desconfiguras la interfaz, **puedes volver al layout por defecto pulsando sobre Layout > Default**. Ahí también puedes guardar tu propia configuración.

Scene



Te muestra visualmente la escena* actual que tienes abierta. Te permite volar por el nivel, seleccionar objetos al pulsarlos y mover, rotar o escalar el objeto seleccionado con las herramientas de la barra superior.

> *En Unity, una escena es como un nivel o una pantalla del juego, un pequeño mundo.

Puedes cambiar de modo 3D a 2D pulsando sobre el botón 2D, en la parte superior de la ventana de Scene.

En modo 3D, también puedes cambiar entre vista isométrica o perspectiva pulsando sobre el icono de arriba a la derecha.

Barra superior y herramientas mover, rotar y escalar

La barra de herramientas de arriba siempre está visible.

A su izquierda encontramos las herramientas de edición de la ventana Scene: Movert, Rotar y Escalar. Fíjate que, al modificar cualquiera de estos valores de un objeto mediante estas herramientas, estás modificando directamente su componente Transform (en el Inspector).

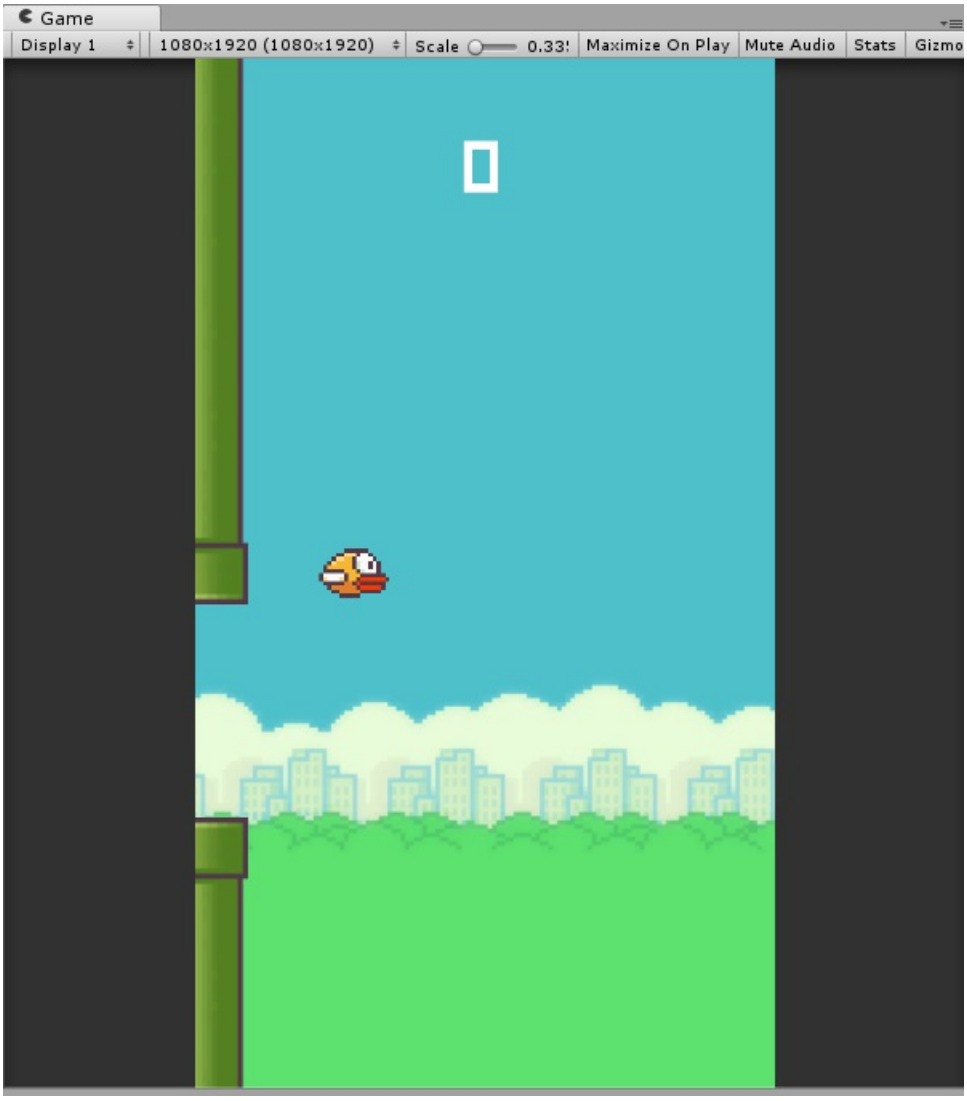
En el centro están los botones de control del juego. Mientras no le des al botón Play estarás en Modo Edición y no estarás probando el juego realmente.

- **Play**: entra en Modo Juego, así puedes probar las físicas, los controles, los movimientos, las animaciones... **Los cambios que hagas en el Modo Juego no se guardarán**.
- **Pause**: pausa el Modo Juego pero sin salir de él.
- **Skip frame**: el botón de la derecha funciona solo cuando el juego está pausado, y sirve para pasar al siguiente frame del juego sin tener que darle a Play. Es decir, para ejecutarlo muy lentamente.

A la derecha hay botones varios sobre organización:

- Collab: abre la ventana de Collab para subir los cambios a tu cuenta de Unity.
- Nube (Services): abre la ventana de los servicios integrados de Unity.
- Account: entra o sal de tu cuenta, para poder usar Collab.
- Layout: guarda o carga tu disposición de ventanas.

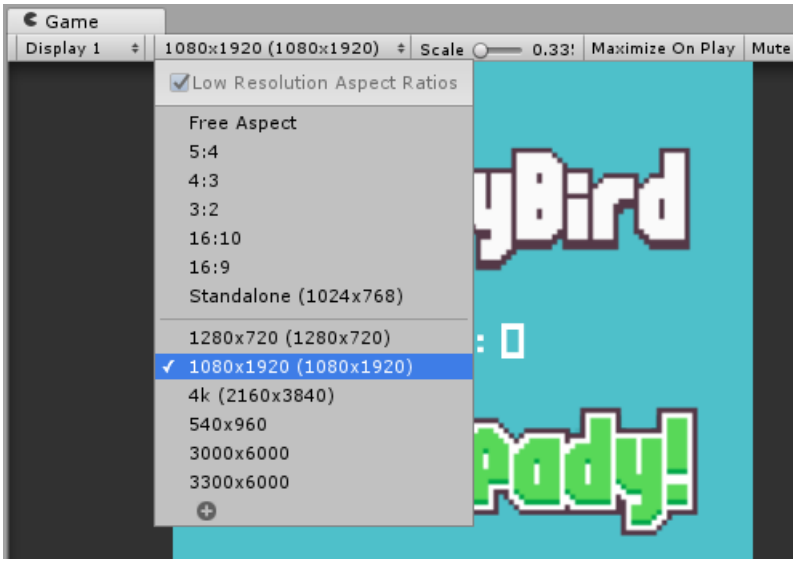
Game



La ventana de Game es muy importante porque **muestra lo que se verá en el juego final en realidad** a través de la Main Camera de la escena.

Desde Game no se puede girar la cámara ni seleccionar objetos.

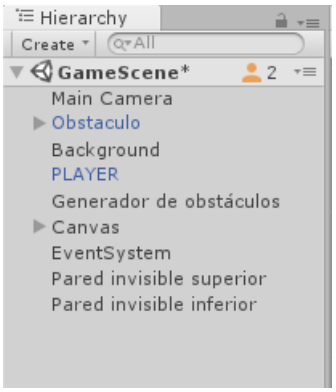
En las opciones de su barra superior, es posible simular que la pantalla tiene una resolución o una relación de aspecto determinados. Sirve para probar qué tal se verá en diferentes resoluciones.



Además, se le puede decir al programa que la maximice cada vez que pulsamos Play.

Es recomendable tenerla siempre mostrada, junto con Scene.

Hierarchy



Muestra la escena abierta en este momento, igual que la ventana Scene, solo que no de forma gráfica sino de forma textual.

Se llama jerarquía porque los Game Objects, al igual que los objetos en programas de 3D, pueden tener uno o varios hijos, de forma que la posición de los hijos siempre es relativa a la de su padre.

¡Nombra los Game Objects de forma descriptiva!

Al seleccionar un Game Object también te lo selecciona en la vista de la escena y te lo muestra en el inspector, igual que si lo seleccionaras visualmente en la vista de la escena.

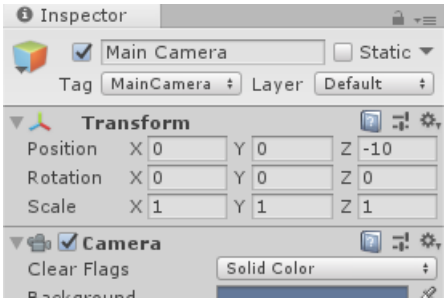
Inspector

Es la ventana más importante de Unity y la que más tiempo estarás mirando, probablemente.

Muestra el objeto seleccionado en detalle. Si es un Game Object, siempre mostrará las cosas comunes a todos los Game Objects arriba del todo, lo primero:

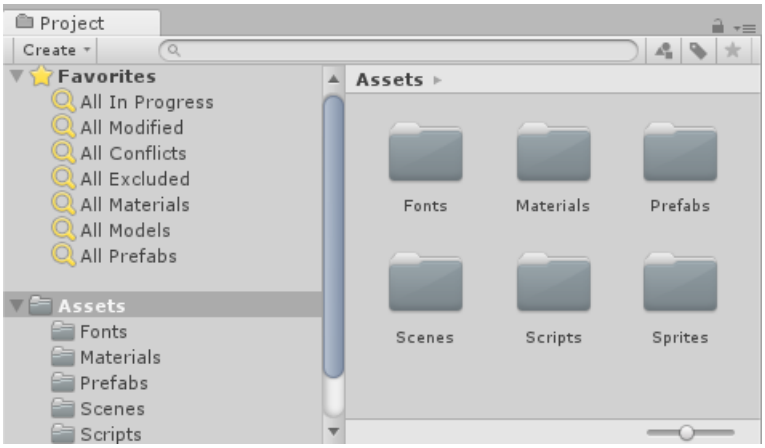
- Nombre del Game Object
- Tag
- Si está activo o desactivado
- Su componente Transform: posición, rotación y escala.

Desde aquí, modificando su Transform puedes cambiar la posición o la rotación del objeto de una forma mucho más precisa que con las herramientas de la vista de la escena.



Además, muestra la lista de componentes que tiene asociados el objeto, y permite modificarlos incluso en Modo Juego.

Project



Muestra la lista de archivos del juego, también llamados Assets (Recursos). **Es como un explorador de archivos**. Aquí estarán los archivos que compondrán el proyecto: imágenes, materiales, objetos 3D, scripts, escenas… Como puedes imaginar, es muy importante mantenerlo organizado y crear una carpeta para cada tipo de archivo.

Con botón derecho > Show in Explorer (o reveal in Finder en Mac) puedes abrir la carpeta en el explorador del sistema de archivos.

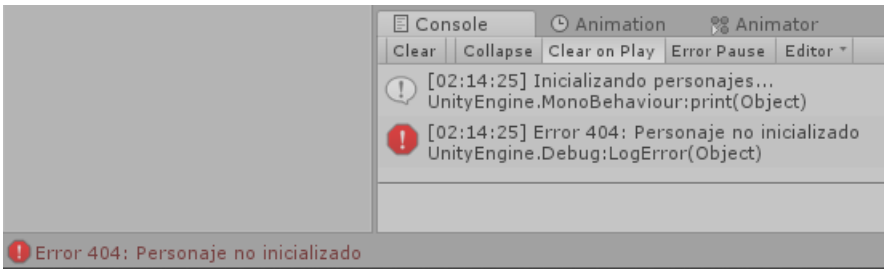
Console

Principalmente, **muestra los mensajes de error del código** que hemos escrito.

Siempre tiene que estar vacía de errores, si nos salge algo en rojo (error) o en amarillo (warning), hay que corregirlo.

También sirve para mostrar valores en consola para debuggear (es decir, para probar que algo está funcionando).

La barra inferior del todo también muestra siempre el último mensaje imprimido en la consola.



Otras ventanas

Unity tiene muchas ventanas más, pero son más secundarias (pero no menos importantes).

Además, nosotros mismos podemos crear nuevas ventanas que nos permitan modificar más fácilmente nuestro juego. O, cuando utilicemos algún asset externo de la Asset Store, es posible que tenga sus propias ventanas de configuración.

- **Lightning**: para modificar las opciones de iluminación de la escena y su skybox (sólo útil en 3D).
- **Asset Store**: para descargar códigos, 3D, 2D o incluso juegos completos hechos por otras personas para utilizarlos en tu juego.
- **Package Manager**: como la Asset Store pero de herramientas más avanzadas oficiales de Unity, como el constructor de escenarios Pro Builder o las fuentes TextMeshPro.
- **Animation & Animator**: la primera es para crear animaciones en cualquier Game Object, y la segunda para aplicar esas animaciones y definir la máquina de estados de las animaciones. Este sistema de máquina de estados se llama Mechanim.
- **Services**: aquí puedes configurar tu proyecto para que se guarde en la nube, en tu cuenta personal de Unity.

Conceptos clave de Unity

Unidades de medida

Siempre que hablemos de posiciones en Unity, las medimos en unidades que representan 1 metro. Por ejemplo, un cubo por defecto mide 1x1x1, y su escala es (1, 1, 1). Si movemos el dubo a la posición (3, 0, 0) lo estamos moviendo 3 metros hacia la derecha.

Las rotaciones se miden en ángulos y la escala es una proporción de tamaño (2, 2, 2) será dos veces más grande que el objeto original.

XYZ distintas de los programas de 3D

Lo primero a tener en cuenta es que, mientras que en los programas de edición 3D el plano XY es el del suelo y Z es la altura, en Unity esto no es así. El plano del suelo es XZ y la altura es Y. Z, en este caso, es la profundidad (más lejos, más cerca). En resumen:

- Eje X: derecha o izquierda
- Eje Y: arriba o abajo
- Eje Z: lejos o cerca*

> Cerca es negativo y lejos positivo. Es decir, un objeto en (0, 0, 50) estará centrado pero 50 metros más allá. Por eso la cámara por defecto está en Z = -10, para mirar hacia lo lejos. En cualquier caso, estas referencias dependen del punto de vista, claro.

Game Objects

Todos los objetos de las escenas de Unity son Game Objects, y son los objetos que se utilizan para representar a los del mundo real. Por ejemplo, en un juego de carreras tendríamos 8 GameObjects Coche1, Coche2, Coche3… que representarían los 8 participantes de la carrera.

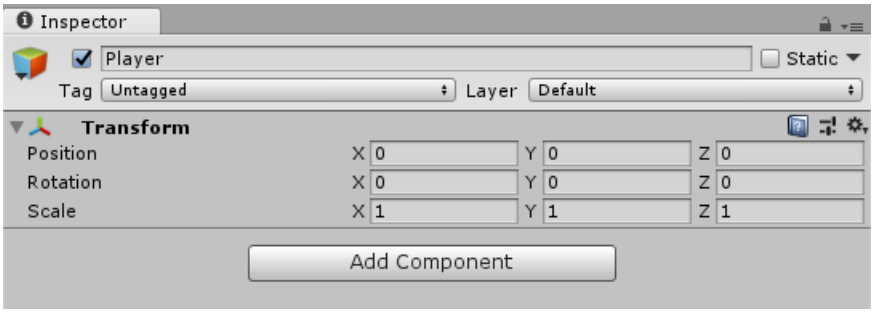
Todos los Game Objects están posicionados en el mundo gracias a que tienen un componente Transform obligatoriamente.

Un Game Object vacío por sí solo no sirve de nada, ya que al no tener ningún otro componente solo representa un punto en el espacio (su Transform) pero vacío e invisible.

Componentes

Los Game Objects están compuestos por componentes, que son los que de verdad le dan funcionalidad al juego. Cada componente tiene una función específica: renderizar el sprite del personaje, gestionar las colisiones, gestionar las físicas, hacer de cámara o directamente el comportamiento que nosotros programemos en un script.

Transform

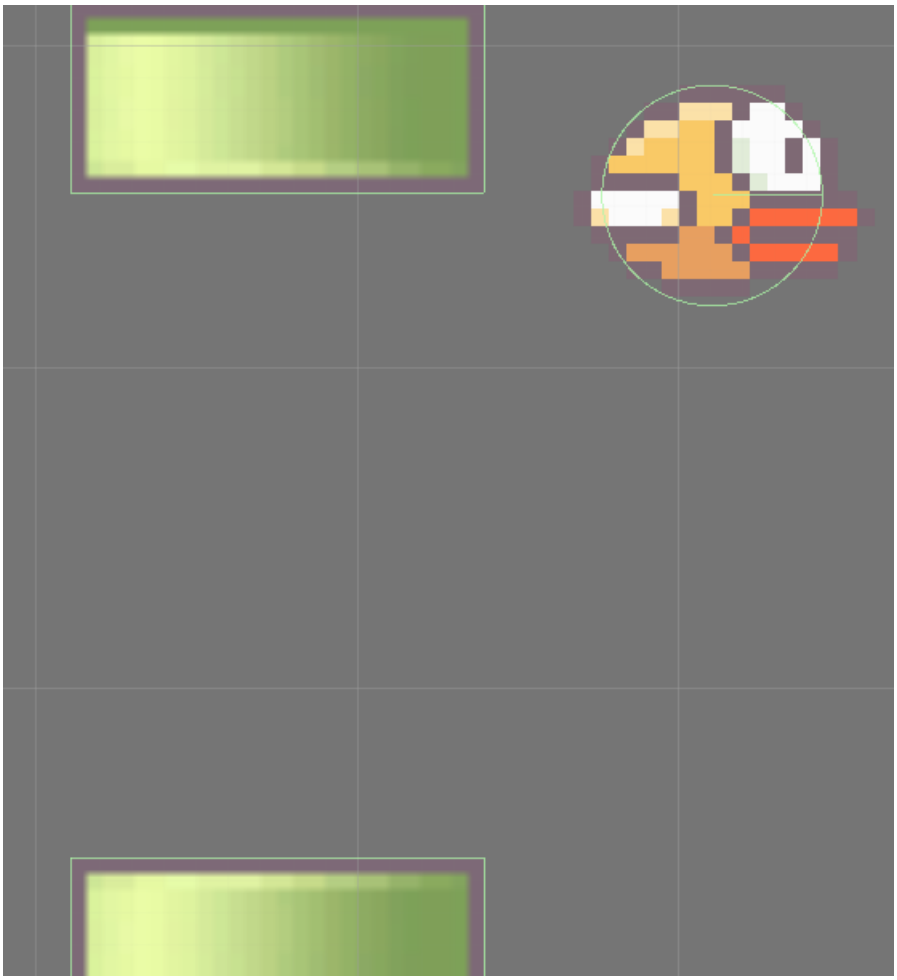


Componente especial que tienen que tener todos los Game Objects para tener una posición, una rotación y una escala en el mundo.

Además, es gracias a este componente que los Game Objects pueden tener hijos o padres en la jerarquía de objetos.

Componente encargado de posicionar los Game Objects en la escena

Rigidbody y Colliders



El Rigidbody hace que nuestro objeto se vea afectado por el motor de físicas de Unity (que caiga por gravedad, que choque y rebote contra paredes, que roce con el suelo y se frene...).

Para que pueda chocar es necesario que el objeto tenga un un Collider adaptado a su forma. Hay varios tipos:

- Sphere Collider
- Box Collider
- Capsule Collider
- Mesh Collider
- etc

Además, tanto los Rigidbody como los Colliders tienen sus respectivas versiones en 2D, ya que las físicas 2D se calculan con un motor de físicas diferente del de 3D.

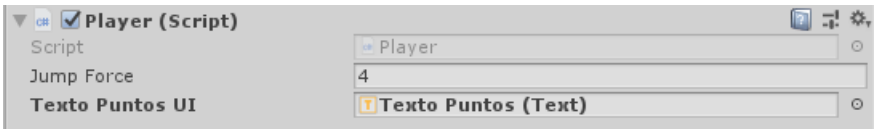
Animator

Cualquier animación que quieras ejecutar en Unity tendrá que pasar por un Animator. Un Animator es un componente que tienen que tener los Game Objects que van a tener animaciones. Además de las animaciones, cada Game Object animado necesitará un Animator Controller para funcionar. En la ventana Animator se muestra el Animator Controller seleccionado o el Animator Controller asociado al Animator del Game Object seleccionado.

No te preocupes si ahora no lo entiendes, es una parte compleja de Unity que es necesario utilizar en un caso real para entenderla bien.

Script (MonoBehaviour)

Es un componente programado por nosotros que define un comportamiento. Todo lo que no nos dé Unity ya hecho tendremos que crearlo nosotros a través de Scripts, y como Unity es un lienzo en blanco que se puede adaptar a cualquier videojuego, desde un First Pershon Shooter hasta un Plataformas 2D, es aquí donde tendremos que programar cualquier particularidad de nuestro juego, por básica que sea.



Por ejemplo:

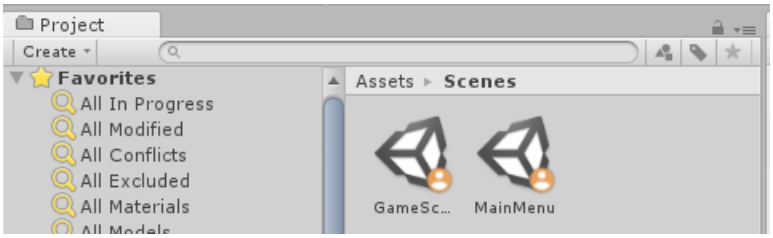
- Movimiento del jugador
- Movimiento de la cámara
- Cambio de nivel
- Muerte o vida del personaje
- Coger objetos
- Contar puntos
- Guardar highscore

Los MonoBehaviours podremos hacer que se comuniquen entre ellos. Por ejemplo, el Script del jugador, que

controla su movimiento y su muerte, tendrá que decirle a un Script central encargado de contar los puntos y mostrar la pantalla de muerte que has muerto, para que él se encargue de mostrar la muerte y los menús.

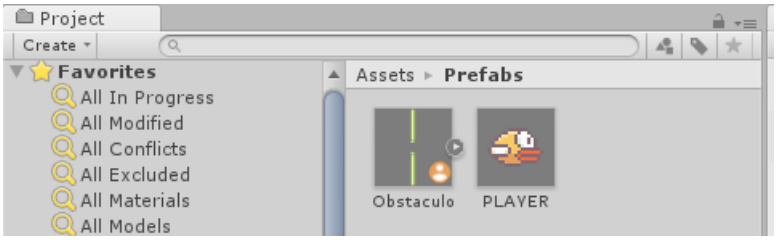
Scenes

Las escenas son los niveles del juego o las distintas pantallas (pantalla de menú, pantalla de selección de perosonaje, pantalla de juego e incluso pantalla de carga).



Se guardan como un archivo de tipo `.unity` en la ventana del proyecto. Arriba de la ventana de Hierarchy podemos ver la escena que tenemos abierta en este momento. Al pulsar sobre `Archivo > Guardar` o hacer `Ctrl + S` estamos guardando la escena actual (los objetos que tiene, la posición de los objetos y también la configuración de los objetos, que son los valores de las variables de sus componentes).

Prefabs

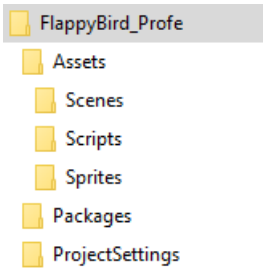


Un Prefab es un Game Object que guardas como un Asset en la ventana del proyecto para reutilizarlo en más de una escena. Suelen ser objetos centrales del gameplay que van a ser spawneados* en nuestro juego, como el jugador en sí, obstáculos, power ups, balas o incluso efectos de sistemas de partículas como polvo o explosiones que serán spawneados en algún momento del juego.

Para crear un prefab

> **Spawnear*, o **instanciar** en español, es cuando creamos un objeto nuevo en la escena mientras el juego está ejecutándose.

Estructura de directorios de un proyecto de Unity



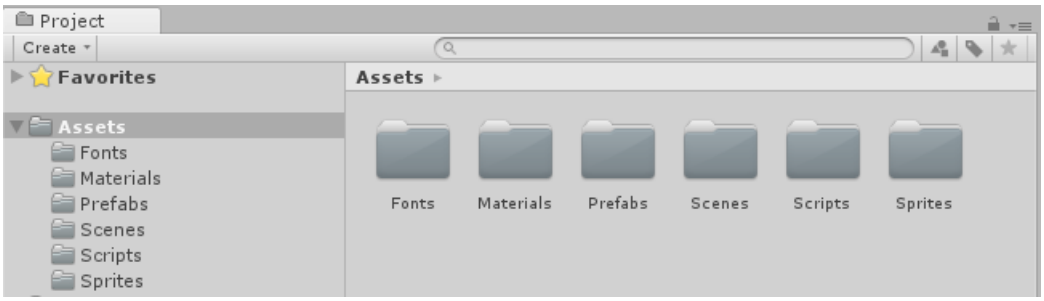
Un proyecto de Unity se guarda en realidad como una carpeta con una serie de subcarpetas y archivos dentro necesarios para que funcione. La subcarpeta más importante para nosotros es la de Assets, ya que el contenido de esa carpeta es lo que se muestra en la ventana de proyecto y es ahí donde guardaremos todos los recursos de nuestro juego.

Por ejemplo, un proyecto llamado FlappyFish, tendrá la siguiente estructura de directorios:

- FlappyFish
  - Project Settings
  - Temp
  - Assets
    - ... Aquí irían las subcarpetas que nosotros queramos crear para mantener nuestro proyecto organizado.

Para guardarnos nuestro proyecto, por ejemplo, en un pendrive, tendríamos que copiarnos la carpeta FlappyFish completa, y luego seleccionar esa carpeta par abrirla como un proyecto de Unity. Si tuviéramos que entregar el proyecto, bastaría con comprimir esa carpeta entera en .zip y subir el archivo comprimido.

Ordenar directorios



Como has visto, existen muchos tipos de archivos diferentes en Unity. Para mantener el proyecto ordenado, es importante que en la ventana Project creemos carpetas para guardar estos archivos y mantener el proyecto ordenado.

- Scenes
- Scripts
- Sprites
- Audio
  - Music
  - Sound effects

- Prefabs
- Materials
- Models3D
- Animations
    - Animation clips
    - Animator controllers

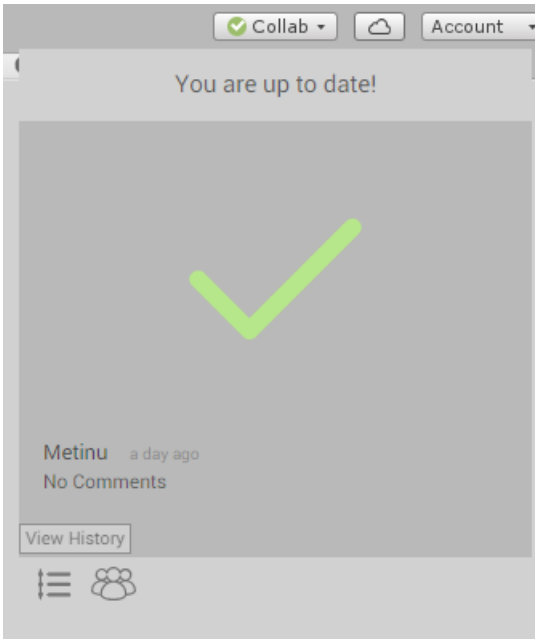
> Un consejo: Unity ordena los directorios por nombre. Como los directorios Scenes y Scripts son los que más utilizaremos, podemos nombrarlos _Scenes y _Scripts para que se muestren al principio del todo.

Cómo guardar el proyecto

Offline

Para guardarnos el proyecto en nuestro pendrive basta con copiar la carpeta raíz del proyecto, la que tiene el mismo nombre que el proyecto en sí.

Online (con Unity Collab)



En la barra de herramientas de Unity, arriba a la derecha, podemos desplegar el menú de Collab para subir nuestros cambios a la nube personal de nuestra cuenta de Unity.

¡Asegúrate de que estás logueado en tu cuenta y no en la de otro! Puedes hacerlo justo en el botón de al lado: Accounts

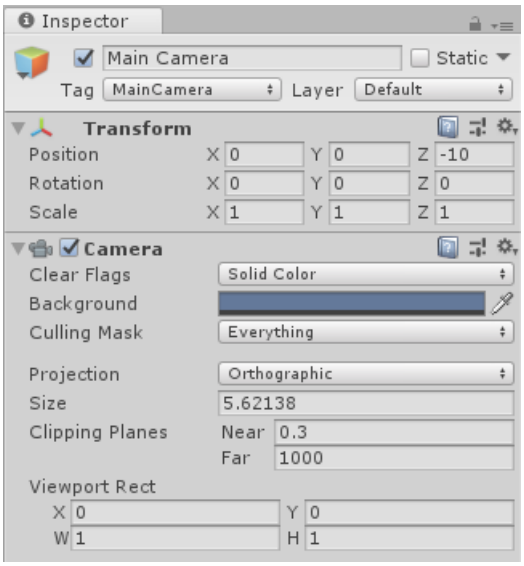
Online (con GitHub)

Si quieres subir tu proyecto a GitHub, tanto el código como el resto de archivos, tienes que configurar la carpeta del proyecto para que sea un repositorio. La forma más fácil de hacerlo es mediante GitHub Desktop.

Preparar la cámara

Vamos a ponernos manos a la obra con el juego. Lo primero es preparar la cámara. Por defecto, Unity en su plantilla para 2D nos trae creada una escena llamada “Sample Scene” que únicamente tiene un Game Object “Main Camera” que es la cámara del juego. Originalmente ya está bien configurado, pero hay si modificamos cualquier cosa tenemos que asegurarnos de que están bien configurados las dos cosas siguientes:

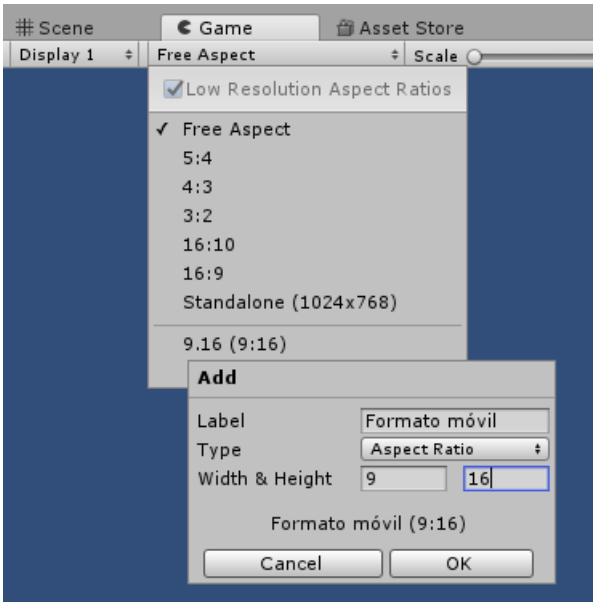
- Posición en Z = -10 y sin rotar, de forma que la cámara mire al centro del mundo (0, 0, 0) y al plano XY que es donde se desarrollará la acción (por eso lo llamamos el plano de acción).
- Proyección Ortográfica para que los objetos no se deformen con la distancia.



Más adelante, modificaremos también su variable “Size” para que englobe el background y toda la pantalla de móvil.

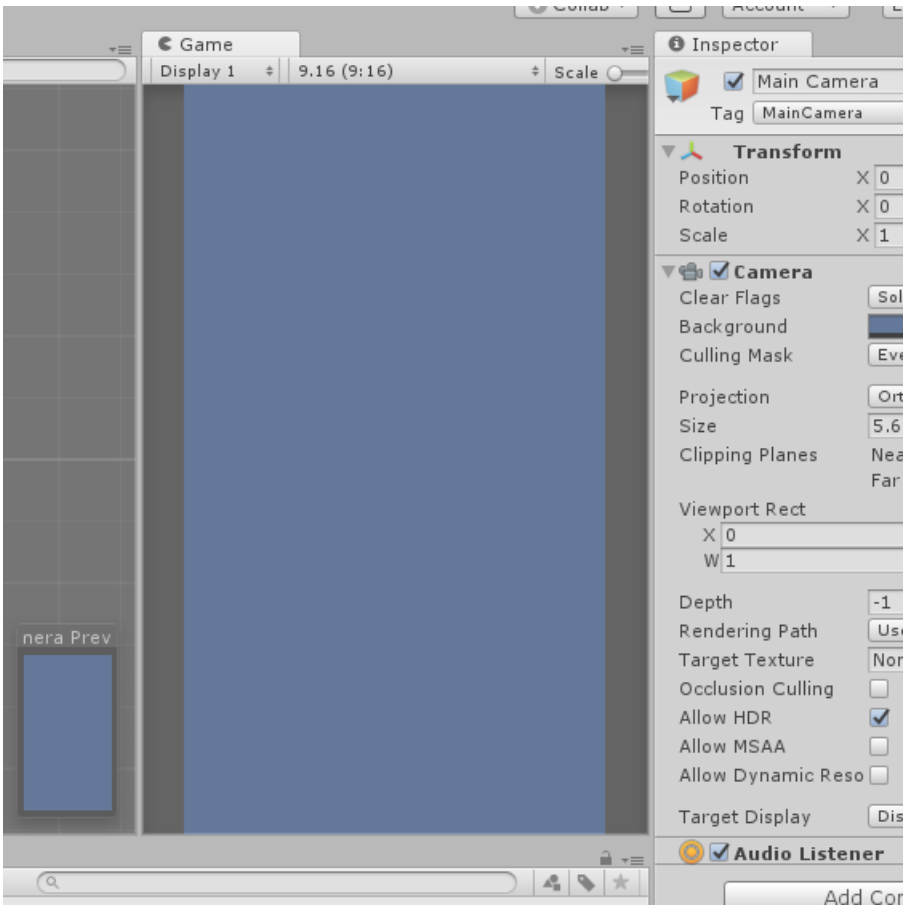
Aspect ratio

Como nuestro juego será jugado en móviles, configuraremos la ventana de Game para que se vea en formato móvil. Para ello, en la barra superior de la ventana de Game, desplegar “Free Aspect” y seleccionar el símbolo + para añadir un nuevo aspect ratio de 9:16. Ten cuidado de seleccionar “Aspect Ratio” en tipo.



Ventana Game siempre visible

Si arrastramos la ventana de Game, podemos acoplarla en cualquier otro sitio de la interfaz. En juegos móviles con pantallas sencillas es recomendable tenerla siempre visible.



Importar sprites

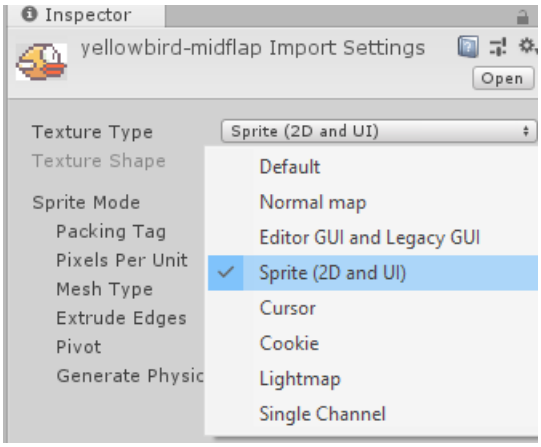
Importar sprite a Unity

Para importar un Sprite a Unity, arrastra la imagen PNG a la ventana de proyecto, o directamente guárdala dentro de la carpeta de Assets de Unity.

Tiene que ser PNG para que tenga transparencia, y en caso de ser una spritesheet de animaciones, recórtalo en photoshop para que solo sea un fotograma, de las animaciones nos encargaremos después.

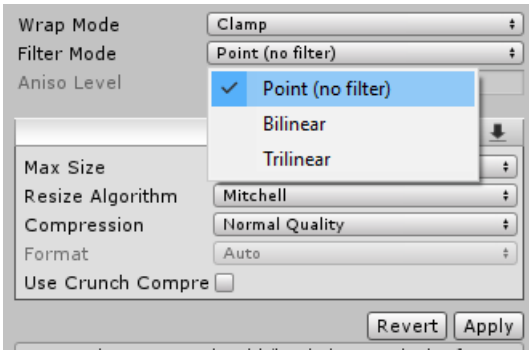
> Un spritesheet es una recopilación de sprites guardados en una misma imágen de forma cuadriculada, ocupando cada uno una celda, para que así los sprites se carguen en la memoria RAM todos de golpe reduciendo así los tiempos de caga.
>
> Normalmente, se agrupan los fotogramas de las animaciones de un personaje, o las distintas baldosas (tiles) que componen el mapa de un videojuego 2D.
>
> Busca Spritesheet en Google Imágenes para ver ejemplos.

Con el Sprite seleccionado, mira la ventana de Inspector para cambiar sus opciones de importación, y asegúrate de que dn Texture Type está seleccionado Sprite (2D and UI), como en la screenshot:



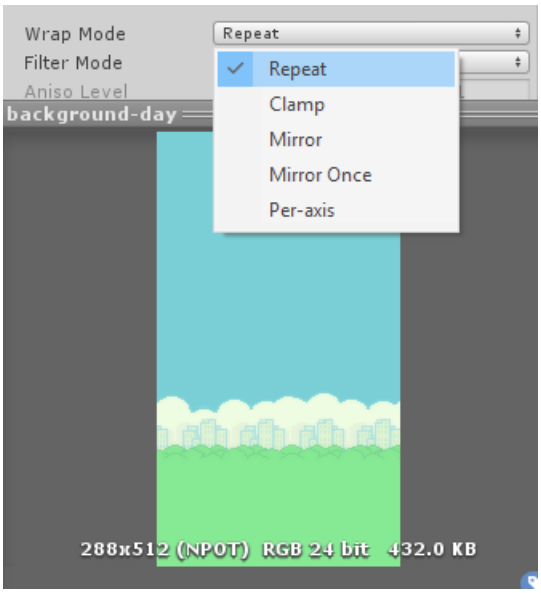
Reescalar sprite sin perder calidad

Si tu juego es pixel art, querrás evitar que los píxeles se difuminen al aumentar el tamaño del sprite. Para ello, selecciona el archivo del sprite y en las opciones de importación, en Filter Mode selecciona Point (no filter), como en la siguiente screenshot:



Hacer textura del background tileable

Si nuestro sprite es un background o una textura que queremos a aplicar a algún suelo o pared, y por tanto queremos que se repita para rellenar el objeto, tenemos que seleccionar “Repeat” en las opciones de importación, en el Wrap Mode.

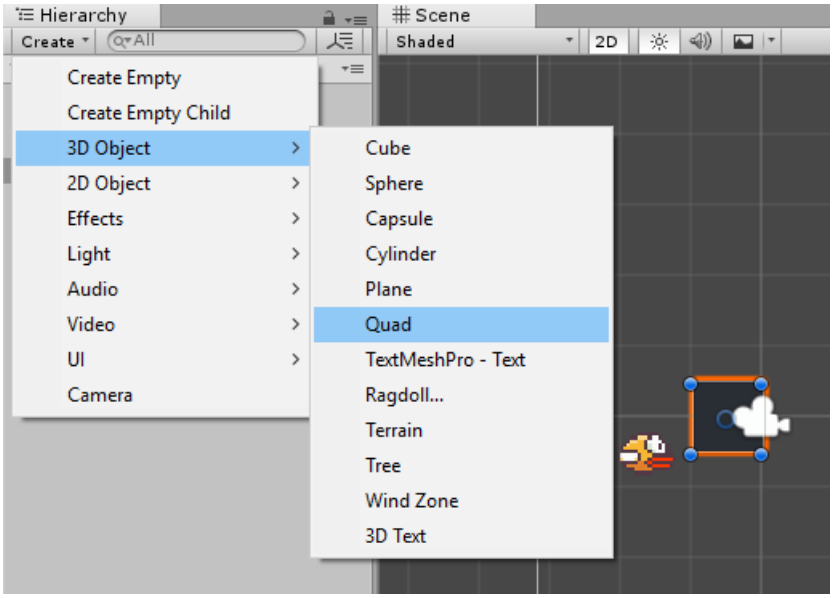


En nuestro caso, esto querremos hacerlo para nuestro sprite de Background.

> Una textura tileable es aquella cuyo borde encaja con su borde del lago contrario, de manera que no se note la repetición al repetirla. En nuestro caso, los backgrounds de Flappy Bird, son tileables solo horizontalmente.

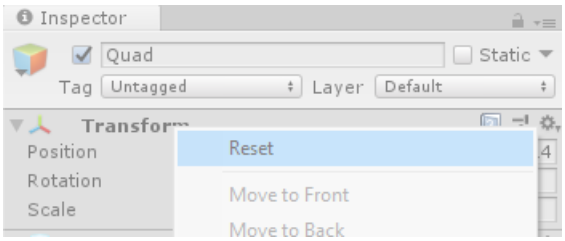
Preparar fondo

Primero tenemos que crear un plano de fondo donde poner nuestro background como textura. Para ello, en la ventana de jerarquía, selecciona Create > 3D object > Quad.

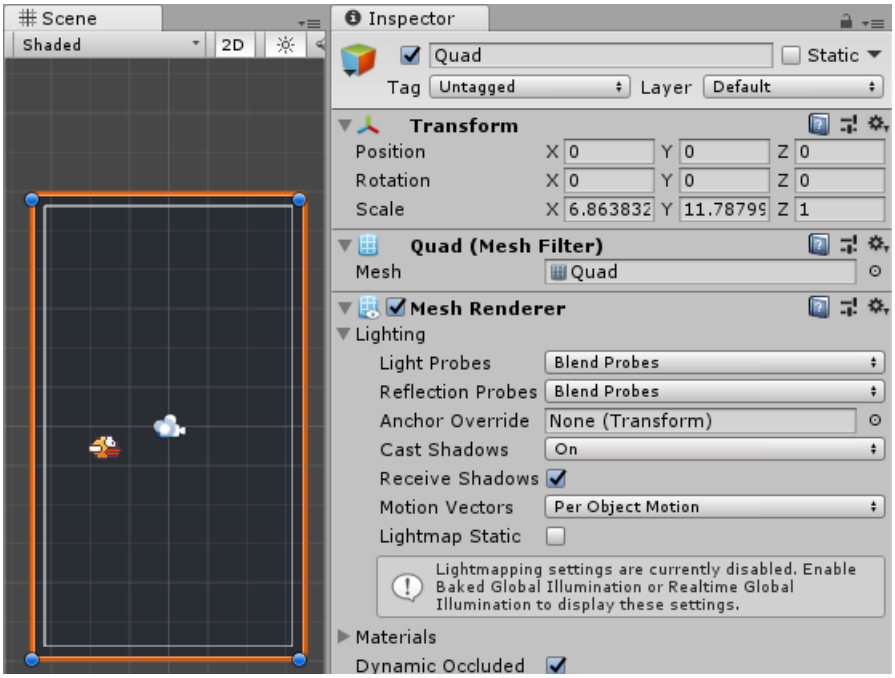


Fíjate que el Quad no nos lo ha creado en el centro de la pantalla (0, 0, 0). Primero, posiciónalo en el centro pulsando botón derecho en su Transform > Reset (como en la próxima screenshot):

> No te preocupes si en tu pantalla tú aún no tienes el personaje. Lo pondremos después.

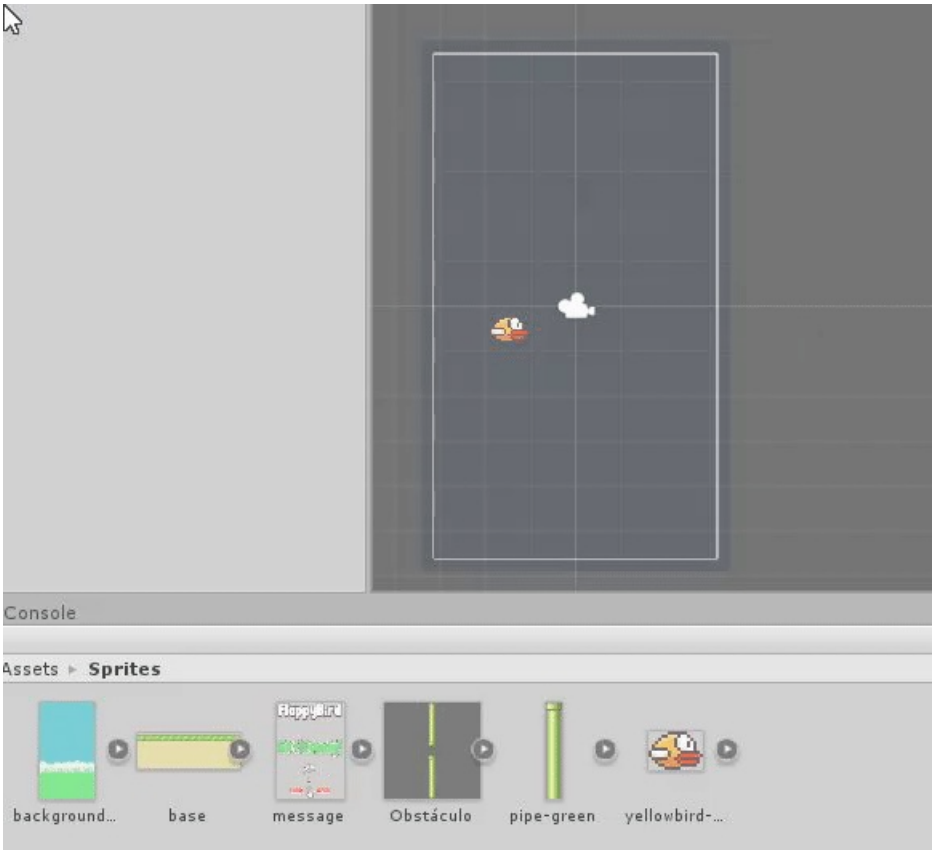


Y a continuación selecciona la herramienta “Rect Tool” en la barra de herramientas de arriba a la derecha, o pulsando T, y estíralo hasta que ocupe toda la pantalla:



Si se sale un poco por los bordes no pasa nada, es preferible eso a que se vea el fondo azul de Unity.

A continuación, arrastra el sprite del background desde la vista de proyecto hasta el Quad. Esto hará que automáticamente, en la ventana del proyecto, se cree un material con la textura del background aplicada. Se verá muy oscuro:



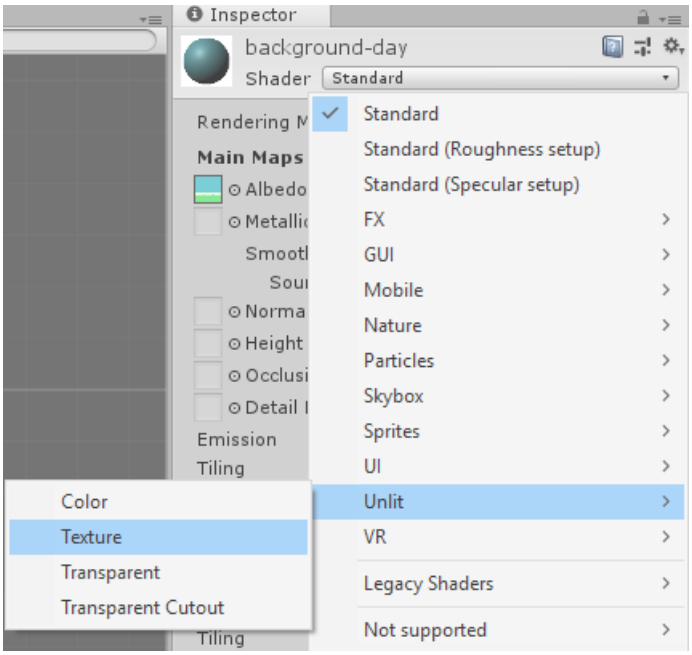
* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en

movimiento.

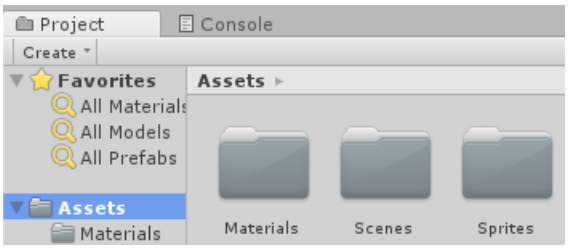
¡Fíjate que me ha creado automáticamente una carpeta "Materials"! Dentro de esa carpeta está el material con la textura del background aplicada.

> ¿Por qué se ve tan oscuro? Debido a que, como explicaré ahora después, al final de la sección, estamos haciendo una triquiñuela mezclando 3D con 2D (por si no te habías dado cuenta al crear el Quad pulsando sobre Create > 3D object) para posicionar el fondo. Al ser un objeto 3D y no tener ninguna luz en la escena, no está iluminado y se ve súper oscuro. En seguida explicaremos por qué hemos usado un objeto en 3D en lugar de un sprite 2D.

Para que no se vea tan oscuro, tenemos que seleccionar el material que nos ha creado automáticamente en la ventana del proyecto, y en el inspector cambiarle el shader a Unlit > Texture para que ignore las sombras y siempre se muestre iluminado.



A continuación, sé ordenado y mueve la carpeta que se ha creado automáticamente, "Materials", a la raíz del proyecto. A priori no volveremos a necesitar ningún otro material, ya que estamos trabajando en 2D, pero todos los materiales los guardaríamos ahí.



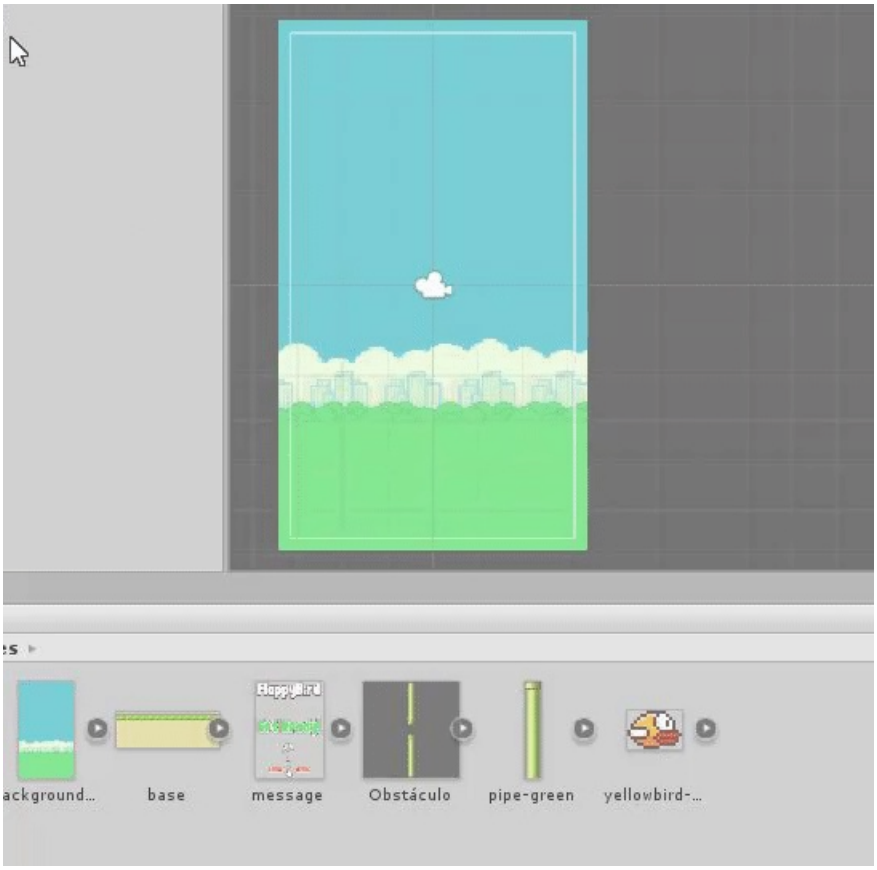
Por último, si tu fondo es más ancho que la pantalla del móvil, verás que al adaptarse al Quad se ha comprimido un montón. Si es así, estira el Quad a lo ancho hasta que las dimensiones se adapten bien.

También, cámbiale el nombre al Game Object y llámalo “Background”.

> ¿Por qué hemos usado un objeto 3D si estamos trabajando en 2D? Porque programar que el fondo scrollee usando Sprites es un pelín más complicado de programar, ya que se trata de duplicar el fondo a lo ancho y teletransportarlo en el momento exacto hacia atrás para que no se note el cambio, como hace en este tutorial.
>
> Nosotros conseguiremos el mismo efecto usando otro método algo más sencillo, que es el que se muestra en este otro tutorial. Consiste en animar el offset de la textura del material.

Preparar player

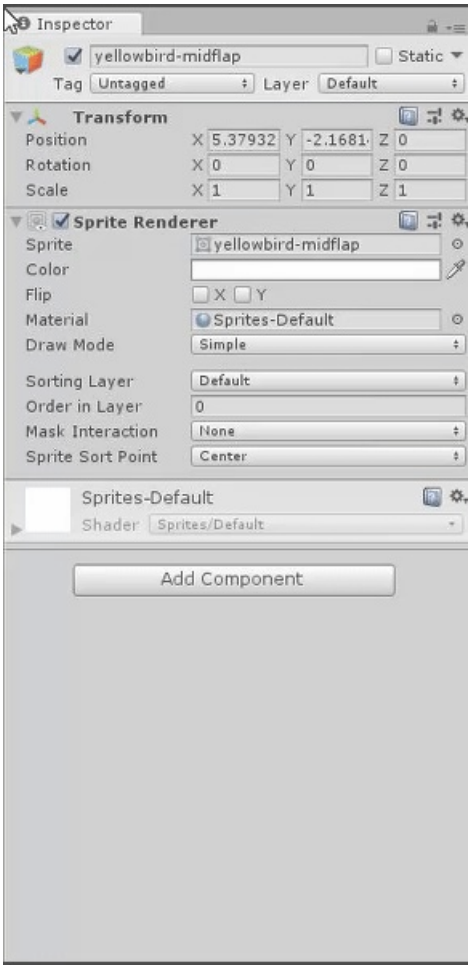
Para crear un Game Object que se vea en 2D a través de un sprite, necesitamos el componente Sprite Renderer en el Game Object. Podemos hacerlo manualmente, o podemos dejar a Unity que lo haga por nosotros simplemente arrastrando y soltando el sprite en la vista de la escena.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

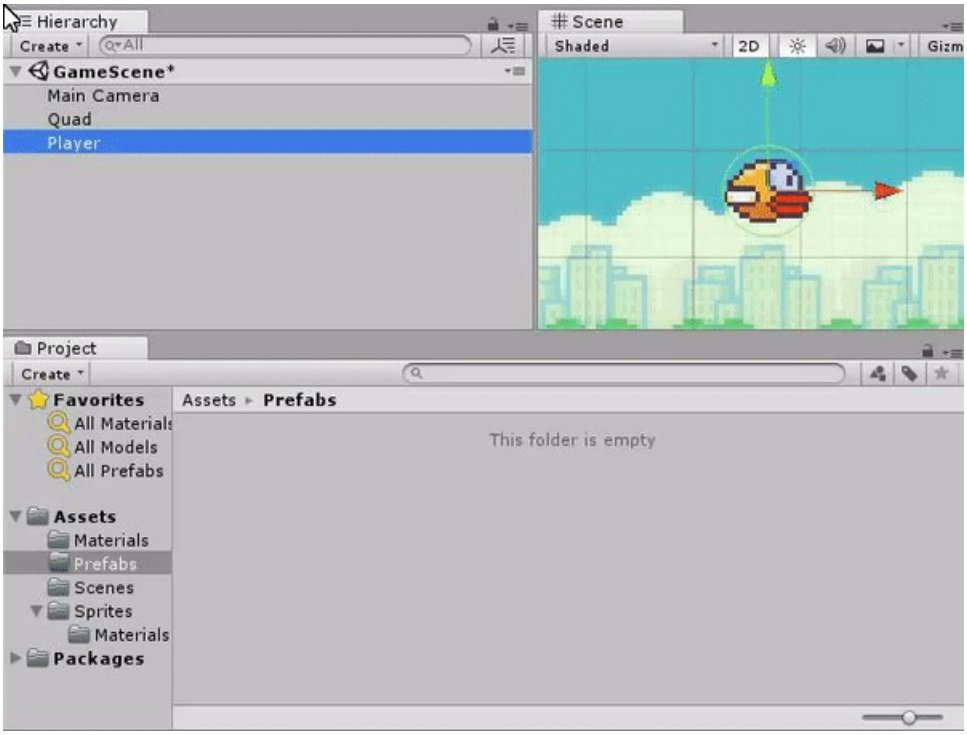
Cambiamos el nombre del Game Object a Player, y le añadimos los componentes que necesita para funcionar, que son un Rigidbody2D para que responda a físicas y un Collider para que colisione. Para ello,

clickar en “Add Component” en el inspector del Game Object player y añadir estos componentes.



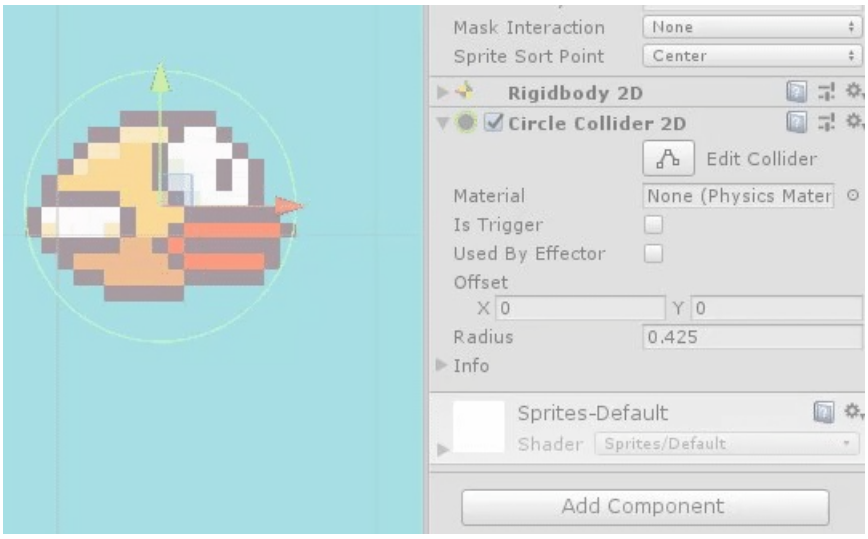
* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Ya tenemos listo a nuestro Player para caer y para chocar. Lo próximo será programarlo para que salte, pero eso lo dejaremos para después. A continuación, vamos a convertirlo en un Prefab. Para ello, basta con arrastrarlo desde la ventana de jerarquía hasta la ventana del proyecto. Crea una carpeta llamada Prefabs para la ocasión, y guárdalo ahí:



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

En principio, no necesitaremos spawnear a nuestro personaje, ya que solo tenemos un nivel. Pero convertir en Prefab al personaje es una buena práctica ya que nos será útil cuando nuestro juego tenga varios niveles.

> Si necesitar ajustar el hitbox del jugador, es decir, su collider, pulsa en “Edit Collider”, en el componente collider, y usa los gizmos para adaptarlo. Como en el este gif:
>
> 
>

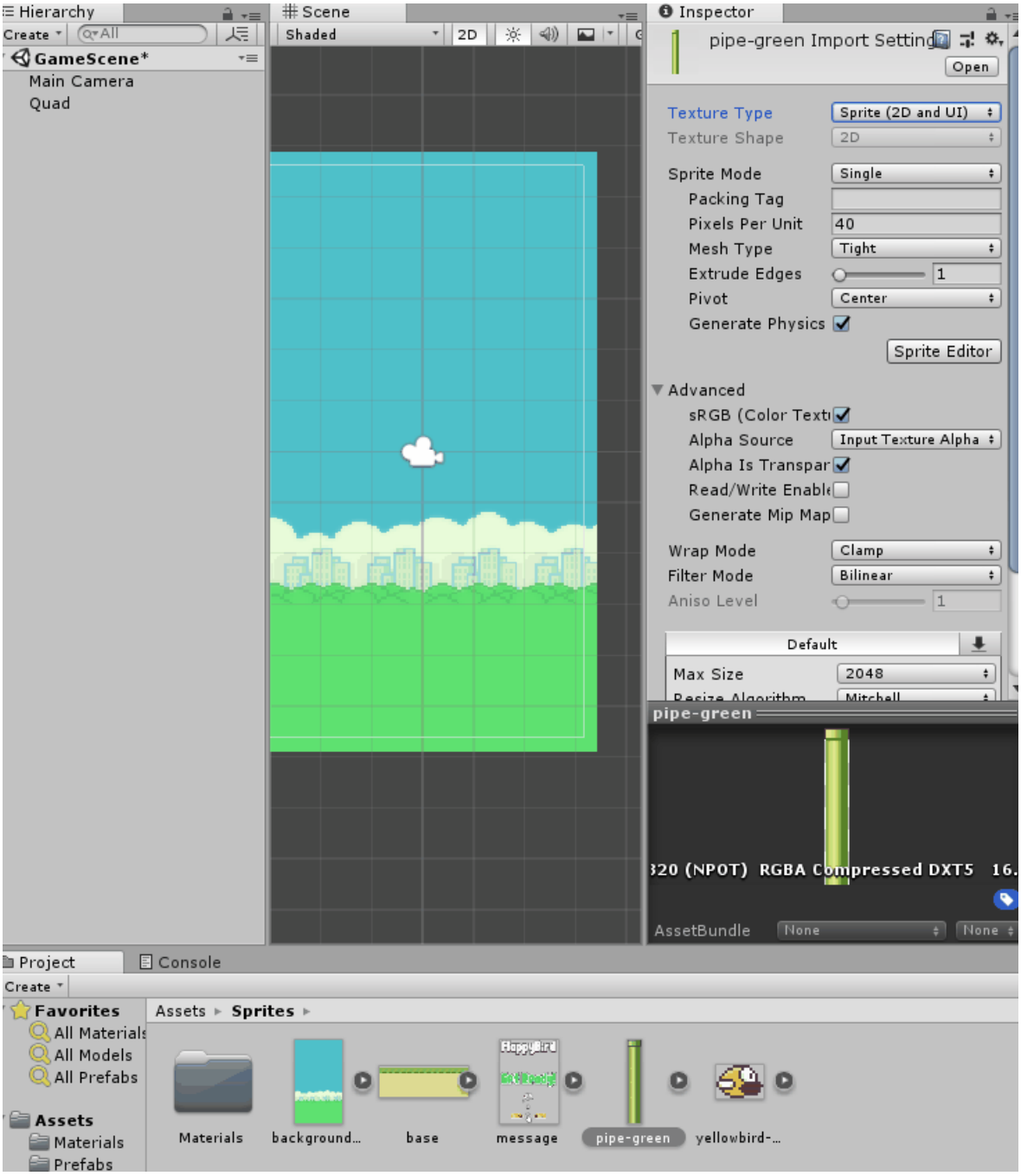
* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

> Además, puede ser que tu necesites un tipo de Collider diferente para adaptarlo a la forma de tu sprite. De ser así, no dudes en probar otro tipo de sprites, pero cuida siempre de que sean 2D.

Preparar obstáculo

A continuación, vamos a preparar el prefab del obstáculo. Un Obstáculo constará de 2 partes, la de arriba y la de abajo, dejando un hueco entre ellas. En este caso el Prefab que tenemos que crear es el prefab completo del obstáculo con sus dos partes, para así spawearlo todo de una vez. **Es muy importante que los objetos hijos estén centrados con respecto al padre, porque si están desplazados aparecerán desplazados al spawnear.** Para que esté centrado vamos a tener cuidado de que el padre esté en el (0, 0, 0) y que las partes de arriba y de abajo estén alineadas en X = 0.

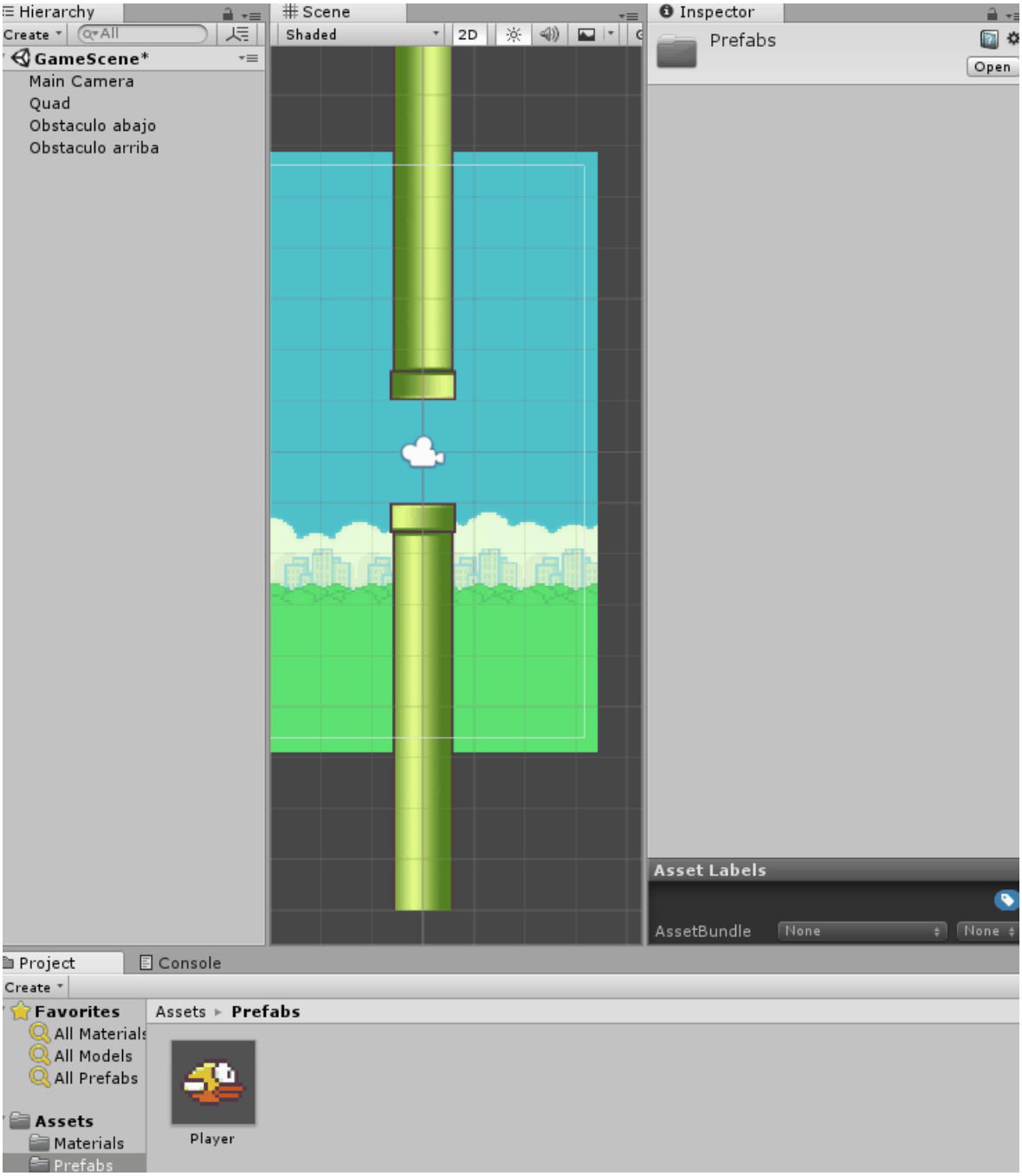
Hacemos como con el player, arrastramos y soltamos el sprite del obstáculo, y los colocamos los 2 en X = 0, solo que desplazados en la vertical. Para el obstáculo superior, hará falta girarlo 180º en el eje Z.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

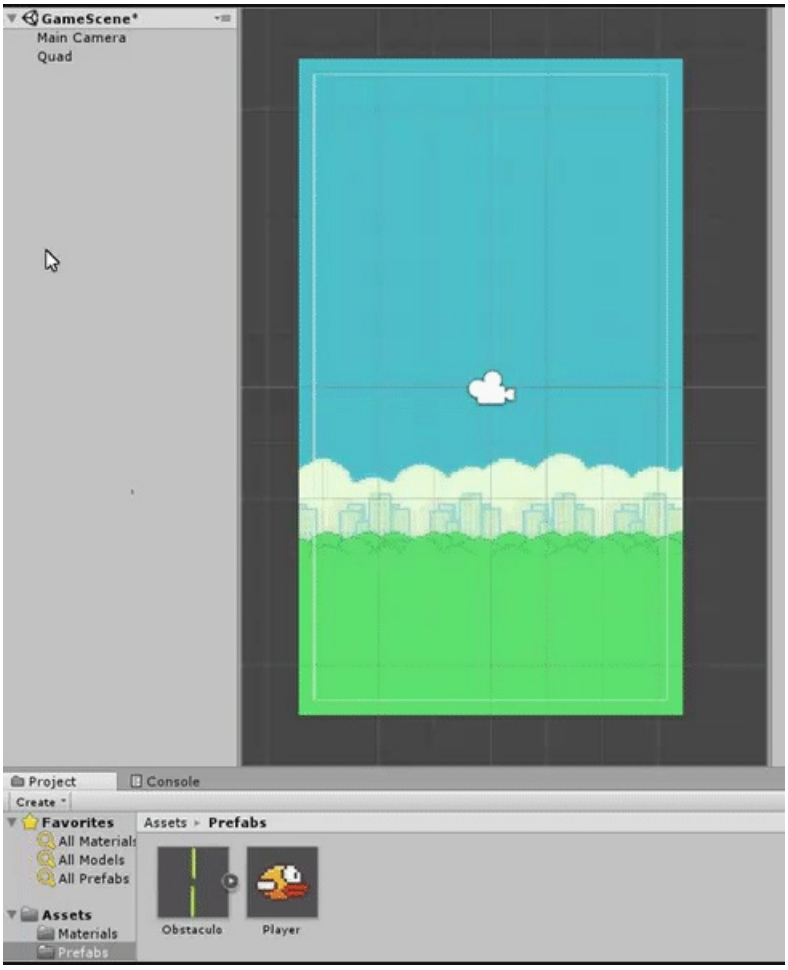
En mi caso, el obstáculo de abajo lo he puesto en -5 en la vertical, y el obstáculo de arriba en +5. **Asegúrate de que ambos están igual de desplazados del centro, para que cuando spawnees el obstáculo completo aleatoriamente, por ejemplo arriba, el hueco quede justo donde lo has spawneado**.

Una vez con las dos partes centradas y posicionadas, añade un collider a cada una, y hazlas hijas de un mismo objeto padre vacío y centrado en (0, 0, 0). **Ya solo falta hacer el prefab. Arrastra el Game Object padre Obstáculo a la ventana del proyecto y listo**. Hazlo como se muestra en el siguiente gif:



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

¡Ahora ya podemos spawnear obstáculos a nuestro antojo sin tener que reconstruirlos cada vez! Pruébalo tú mismo arrastrando los prefabs a la escena:



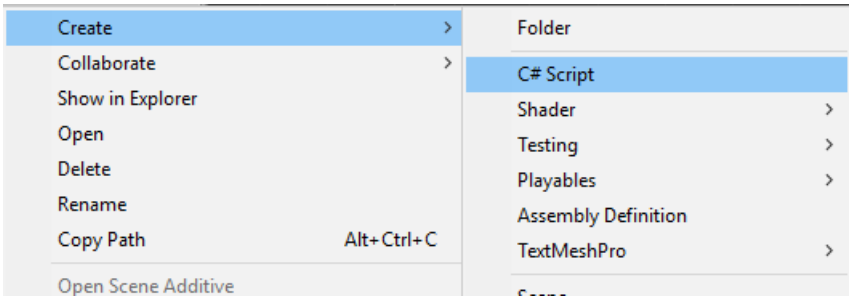
* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Esto mismo que estamos haciendo nosotros es lo que programaremos a nuestro script “Generador de obstáculos” que haga.

> **Mantén el proyecto ordenado desde el primer momento**. Guarda el prefab del obstáculo y el del player en la carpeta Prefabs.

Player.cs - Que el jugador salte

Es el momento de crear nuestro primer código. Hazlo mediante **botón derecho > Create > C# Script**.



Llama al nuevo script creado "**Player**", ya que será el código que asignaremos al Game Object del jugador y que será responsable de su control.

> El nombre del archivo .cs del script **tiene que ser el mismo que el nombre de la clase** MonoBehaviour del código. El archivo y la clase tienen que llamarse igual. Por eso, es importante que nombres el archivo comenzando con la P mayúscula, al igual que harías con una clase. **Si cambias el nombre de la clase en el código también tienes que cambiar el nombre del script, si no no funcionará**.

¿Qué necesitamos que haga el Player?

Gracias al Rigidbody2D que le pusimos al Game Object "Player", si pulsamos Play este debería caer. Ahora queremos que al pulsar espacio, deje de caer y salte hacia arriba.

Para ello, nuestro código estará en todo momento comprobando si el jugador ha pulsado espacio. Si se pulsa espacio, modificará la velocidad del rigidbody para que vaya hacia arriba.

¿Cómo le decimos que haga eso?

Necesitamos 3 ingredientes para programar esto:

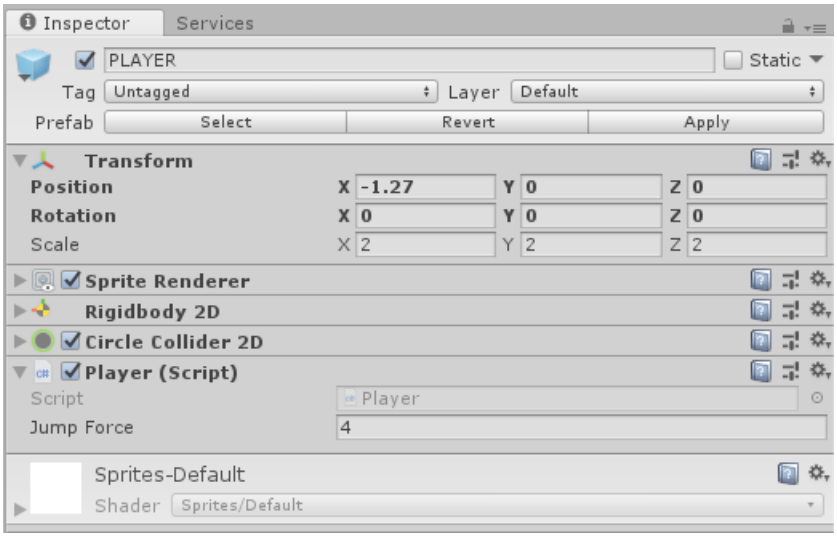
- `GetComponent<Rigidbody2D>()` para acceder al componente Rigidbody2D.
- `if (Input.GetKeyDown(KeyCode.Space)) { ... }` para comprobar si se ha pulsado espacio.
- `miRigidbody.velocity = Vector2.up * 5.0f` para cambiar su velocidad por 5 hacia arriba.

Necesitaremos usar `GetComponent<Rigidbody2D>()` para guardarnos la referencia a nuestro Rigidbody2D al inicio del juego (es decir, en `Start()`), ya que utilizar GetComponent cada frame es un desperdicio de recursos.

Lo que sí tendremos que hacer en todo momento, es decir, cada frame (es decir, en `Update()`) será la comprobación de si se ha pulsado espacio y, en caso de que sí, cambiar la velocidad del Rigidbody2D que hemos guardado en `Start()`.

Sería muy útil poder modificar la fuerza del salto sin tener que entrar a Visual Studio a modificar el código. Para ello, lo ideal es hacer la variable que multiplica la velociad (`* 5.0f` en el ejemplo anterior) pública, de forma que nos aparezca en el inspector y podamos modificarla desde ahí.

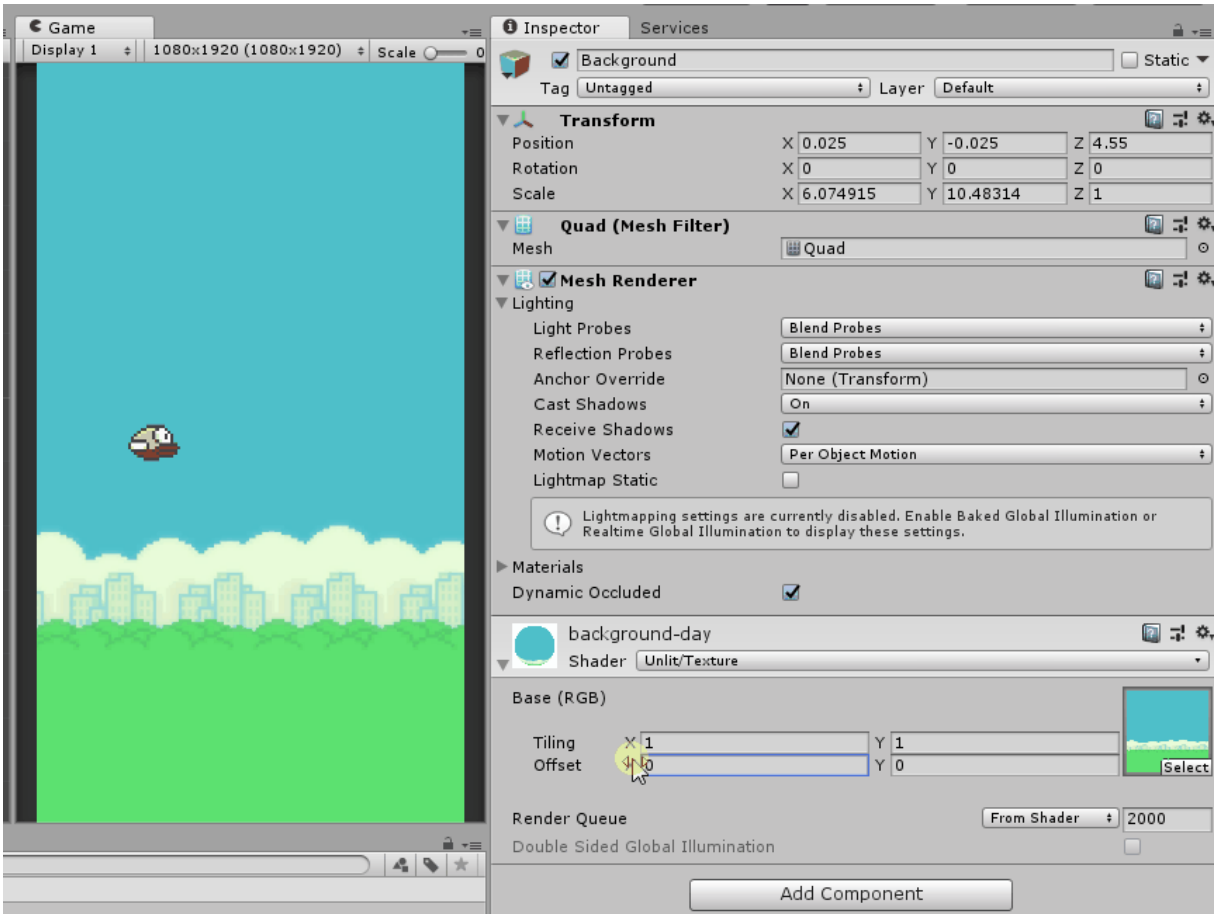
Al final, el script Player.cs tendrá que quedarnos así:



Background.cs - Que el fondo scrollee

¿Qué necesitamos que haga el Background?

Concretamente, queremos aprovechar el material que nos creó automáticamente Unity para acceder a él mediante un script y scrollear su textura. Le diremos al código que haga exactamente lo que hacemos en el siguiente gif:



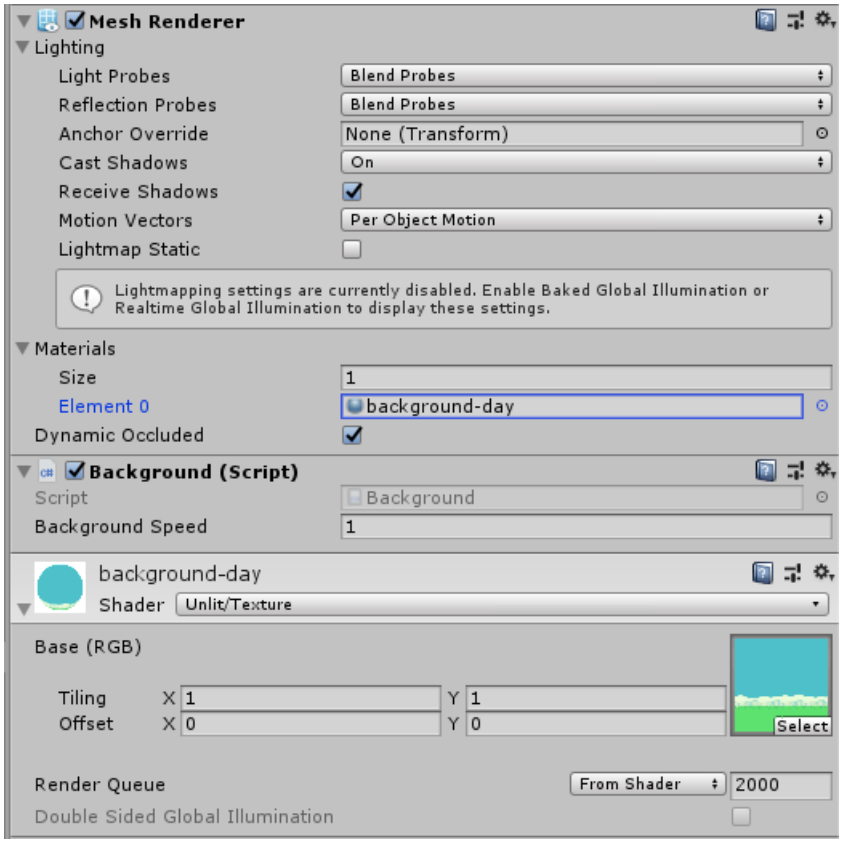
* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

¿Cómo le decimos que haga eso?

Lo primero, necesitaremos una variable pública que sea la velocidad de scroll del background para poder modificarla desde el inspector. Por ejemplo `public float backgroundSpeed`.

También necesitaremos una referencia al material del plano de fondo. Para ello, al inicio (en `Start()`), accederemos al componente Renderer (que es donde se halla el material), y al Renderer le pediremos su material, que lo guardaremos en una variable de tipo Material llamada miMaterial. Para ello:

```
miMaterial = GetComponent<Renderer>().material;
```



> En esta imagen puedes ver que, efectivamente, el componente Renderer (en este caso MeshRenderer) tiene un material asociado que es el mismo que el del fondo. ¿Ves como todo lo que hacemos en el código también puede hacerse desde Unity?

Hecho esto, en todo momento (en `Update()`), le diremos al código que ejecute lo siguiente:

```
miMaterial.mainTextureOffset = miMaterial.mainTextureOffset + Vector2.right * backgroundSpeed * Time.deltaTime;
```

De esta manera ya hemos conseguido justo lo que dijimos en el gif del principio que haríamos

> En la próxima sección, para mover al obstáculo, explicaremos qué significa Time.deltaTime.

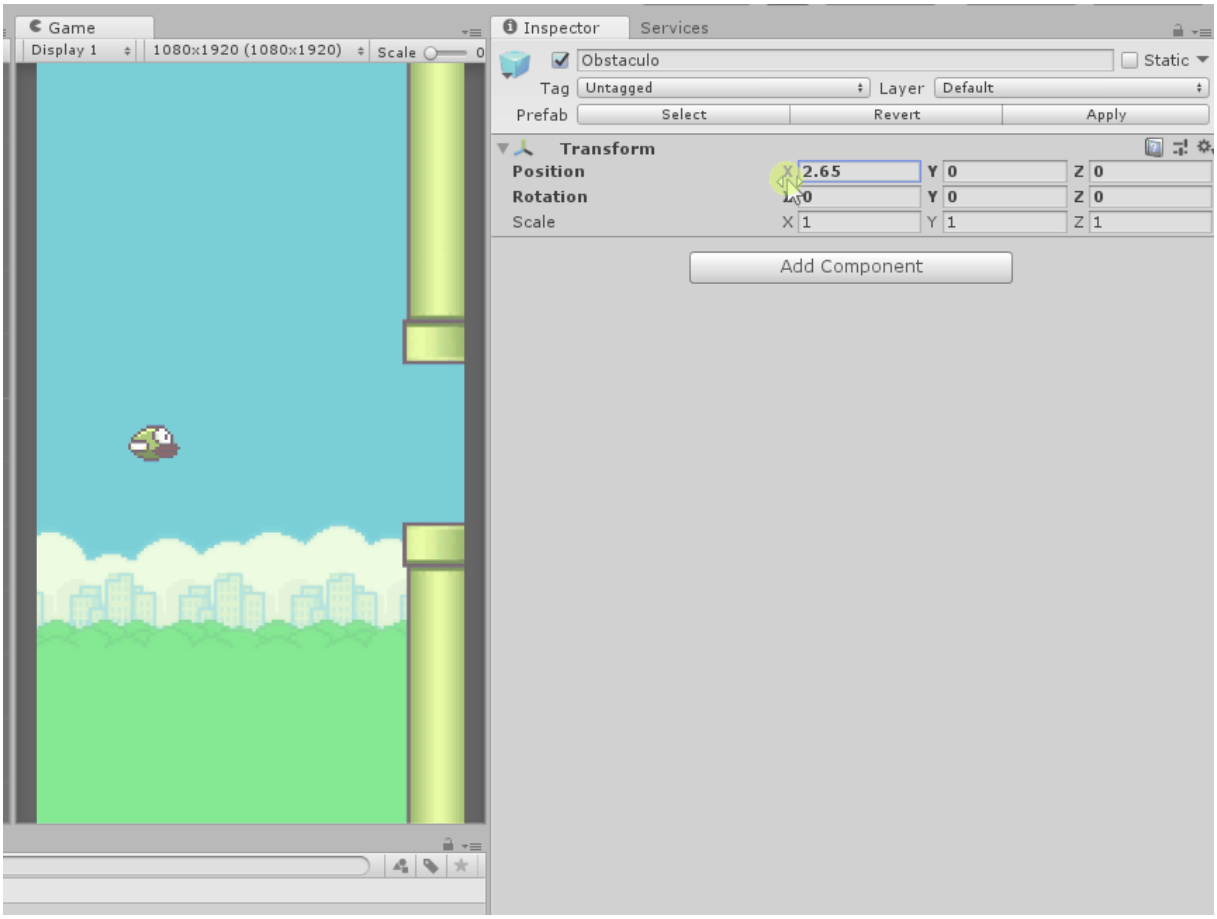
Obstacle.cs - Que el obstáculo se mueva

El movimiento del jugador lo hacemos mediante la velocidad del Rigidbody2D. Pero, ¿qué hacemos para mover las plataformas si queremos evitar que caigan? No es del todo imposible usar también un Rigidbody2D, ya que podemos quitarles la gravedad y bloquear su altura, pero entonces tendríamos al código constantemente calculando unas físicas que son inútiles.

¿Cuál es la alternativa al Rigidbody2D? No tener ningún Rigidbody2D y modificar directamente su posición en la horizontal, que es lo que queremos.

¿Qué necesitamos que haga el Obstáculo?

Para moverlo horizontalmente hacia la izquierda, le diremos al script que modifique su posición en la X hacia la izquierda, accediendo a su Transform. Exactamente igual que hacemos en el siguiente gif:



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

¿Cómo le decimos que haga eso?

Para acceder a su componente Transform no es necesario utilizar `GetComponent<Transform>()`. Ya que todos los Game Objects tienen obligatoriamente una transform, podemos acceder a la posición del Game Object asociado simplemente escribiendo `transform.position`.

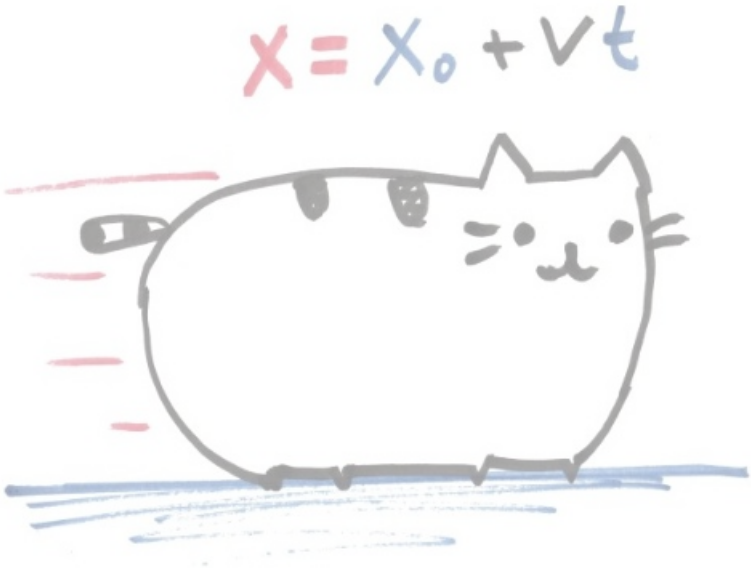
Necesitaremos también una variable pública velocidad, por ejemplo llamada `obstacleSpeed` que determinará la velocidad a la que se moverá el obstáculo.

¿Qué es, en esencia, el movimiento? El movimiento es un cambio en la posición. Nosotros no podemos decirle al código directamente que se mueva, pero podemos cambiar su posición ya que tenemos acceso a `transform.position`. **¿Cuánto queremos variar la posición?** Pues depende de dos factores: la velocidad de

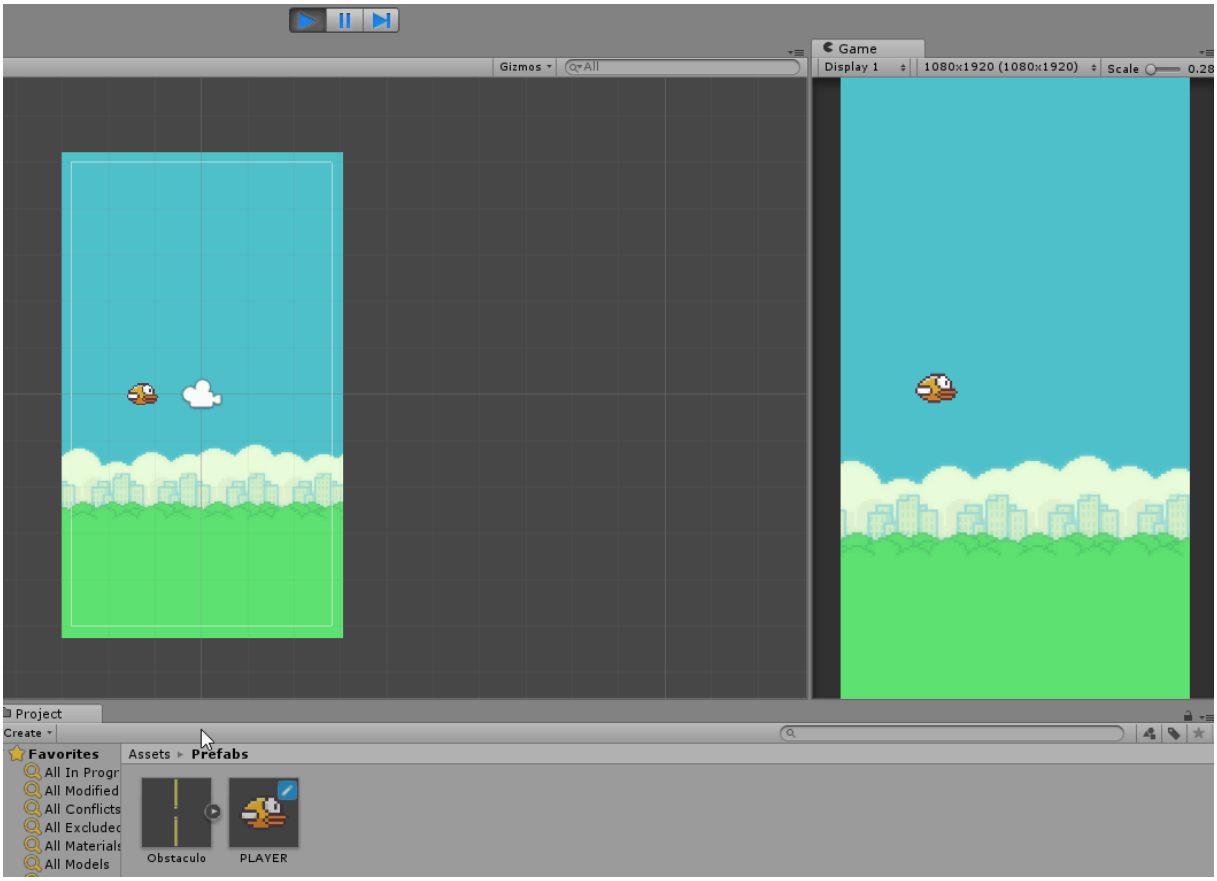
movimiento y el tiempo que se ha estado moviendo a dicha velocidad. En 1 hora a 100km/h nos movemos 100 km. Pero el código que hay dentro de `Update()` no se llama cada 1 hora, sino en todo momento, o más concretamente una vez en cada frame. **¿Cuánto tiempo ha pasado desde el último frame? La respuesta es** `Time.deltaTime`.

Por tanto, en cada frame (en `Update()`) tendremos que ejecutar el código:

```
transform.position = transform.position + Vector3.left * obstacleSpeed * Time.deltaTime;
```

> Más concretamente, el cálculo que acabamos de hacer responde a la fórmula matemática del **Movimiento Rectilíneo Uniforme**. ¿No recuerdas haberla estudiado en 3º o 4º de la ESO, en Física y Química?
>
> 
>

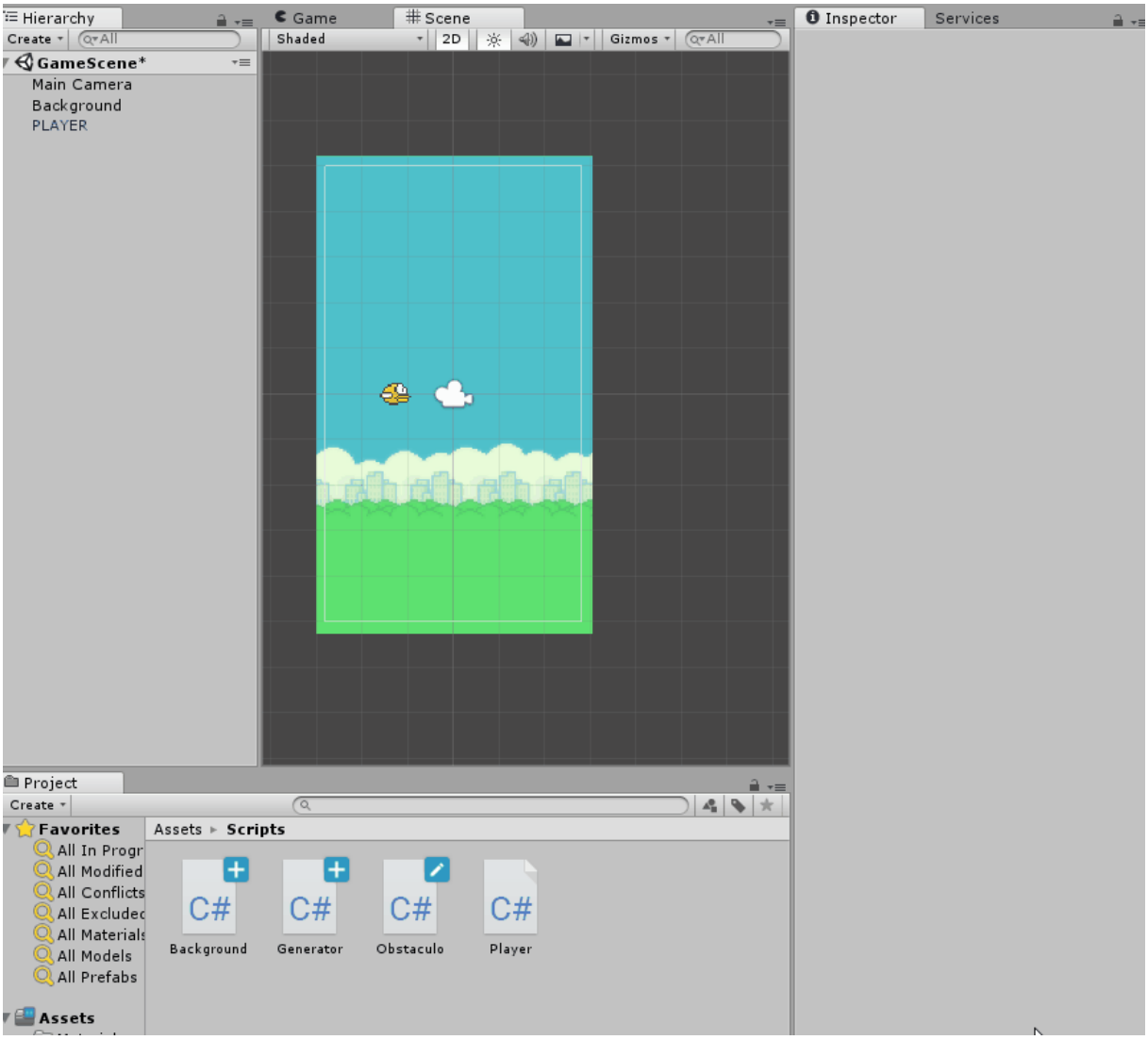
Ya tenemos a los obstáculos moviéndose hacia la izquierda. Recuerda aplicar los cambios al prefab después de agregarle el script y prueba que te funciona dándole a Play y arrastrando algunos obstáculos a la pantalla.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Spawner.cs - Generador que spawnea obstáculos cada X tiempo

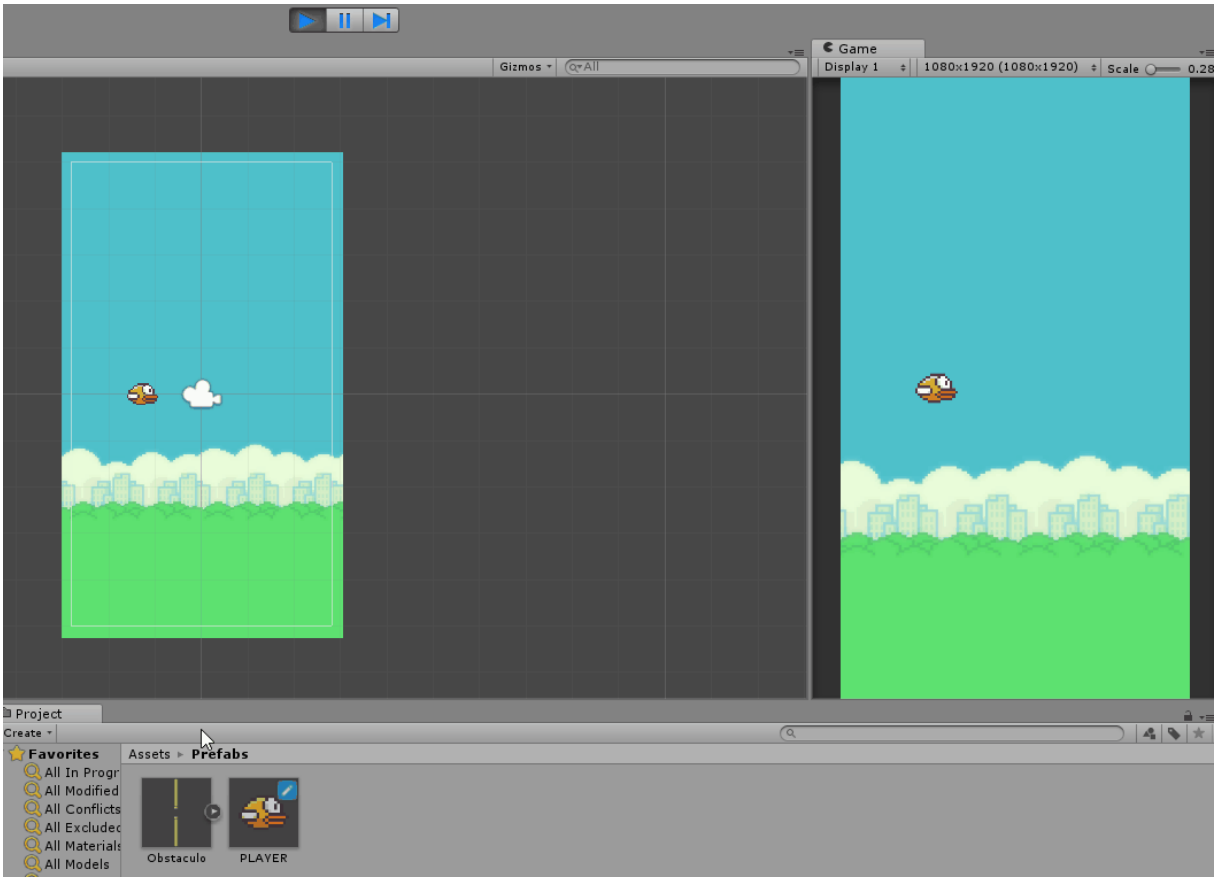
Vamos a crear un Game Object vacío llamado “Generador” y lo vamos a colocar fuera de la pantalla, a la derecha, alineado en el eje XY. Le añadiremos un script llamado Generador.cs o Spawner.cs, por ejemplo, que será el encargado de hacer aparecer los obstáculos en el nivel.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

¿Qué necesitamos que haga el Generador?

En esencia, necesitamos decirle al código que haga lo mismo que hacemos nosotros al arrastrar y soltar obstáculos en la escena:



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Pero teniendo cuidado de dos cosas:

1. Que los spawnee en la posición del generador, y no en cualquier parte, para que así aparezcan donde hayamos dejado el generador, que es a la derecha de la pantalla.
2. Que al spawnearlos, varíe la posición de spawn en un rango arriba o abajo aleatoriamente, para hacer el juego difícil.

¿Cómo le decimos que haga eso?

Necesitamos tres ingredientes para cocinar este código:

- `InvokeRepeating("nombre del método", tiempoParaEmpezar, tiempoDeRepetición)`, que es la forma de decirle al código que ejecute un método cada X segundos (tiempo de repetición) empezando con un delay de "tiempo para empezar" (que en nuestro caso será instantáneamente, `0.0f`).
- `Instantiate(objeto original para copiar, dónde instanciarlo, con qué rotación instanciarlo)`.
- `Random.Range(min, max)` nos da un número que está aleatoriamente entre `min` y `max`.

Como a InvokeRepeating no le podemos decir directamente que el método que tiene que llamar es

Instantiate, tendremos que crearnos un método, llamado Spawn() por ejemplo, que será el que instanciará el obstáculo.

- ¿Qué objeto instanciar? Necesitamos una referencia al prefab que queremos instanciar, que se la asignaremos al generador desde el inspector. Para ello, como atributo de la clase tendremos `public GameObject obstaclePrefab` y a instantiate le pasaremos `obstaclePrefab` como objeto original que copiar.
- ¿Dónde instanciar? En la posición del generador, es decir, la posición de la Transform del Game Object al que está el script asociado, es decir, `transform.position`.
- ¿Con qué rotación? Con ninguna, recto, es decir `Quaternion.identity`

De esta forma tendríamos:

```
void Spawn()
{
    Instantiate(prefabObstaculo, transform.position, Quaternion.identity);
    // Que significa:
    // Instancia el prefab del obstáculo,
    // en la posición de mi transform (la del generador)
    // y con rotación identidad (es decir, ninguna)
}
```

Pero todavía no estamos aleatorizando la posición en que aparece el obstáculo, ya que de momento aparecería siempre en `transform.position`, que es la posición del generador. ¿Cómo le metemos un desplazamiento aleatorio en el eje Y? **Cada vez que vayamos a spawnear** nos inventamos una posición aleatoria que le sumaremos a la posición del generador. Esa posición aleatoria variará entre un rango `spawnRandomRange` que le especificaremos desde el inspector.

```
void Spawn()
{
    Vector3 randomVertical;

    randomVertical.x = 0.0f;
    randomVertical.y = Random.Range(-spawnRandomRange, +spawnRandomRange);
    randomVertical.z = 0.0f;

    Instantiate(prefabObstaculo, transform.position + randomVertical, Quaternion.identity);
}
```

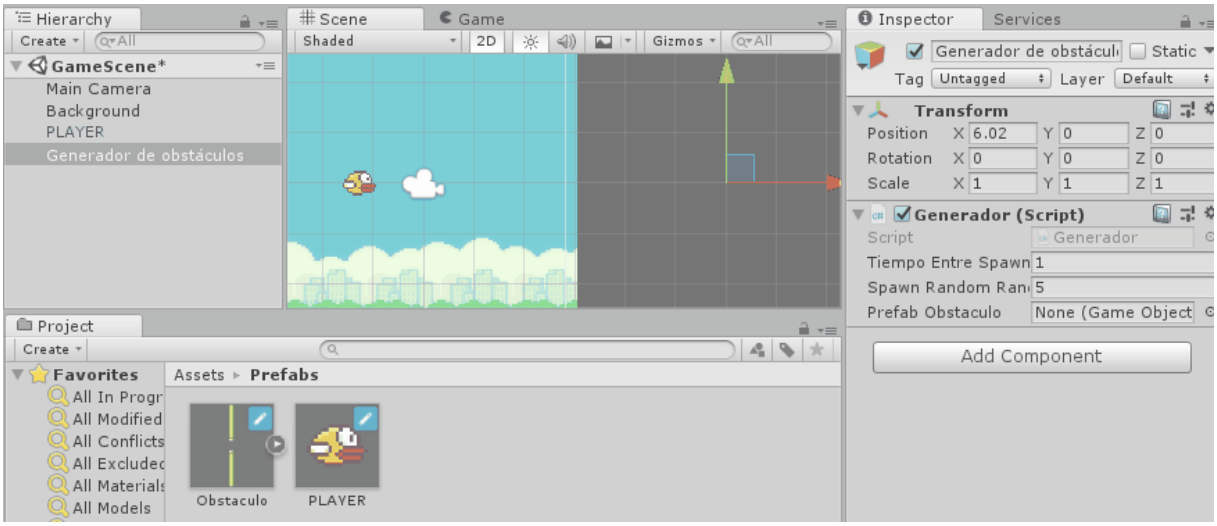
Ya solo nos falta decirle al código que llame repetidamente a nuestro método Spawn. Necesitamos también un atributo público en el inspector que será el tiempo entre spawns de un obstáculo a otro. Para ello, al inicio (en `Start()`) haremos:

```
InvokeRepeating("Spawn", 0.0f, tiempoEntreSpawns)
```

Y ya tenemos funcionando nuestro generador de obstáculos. Solo nos faltaría calibrarlo en el inspector para que los spawnee cada un tiempo que resulte divertido, y calibrar también su rango aleatorio para que no se salgan de la pantalla.

> Si nuestros obstáculos no están bien centrados, puede que necesitemos especificar dos valores diferentes para generar el obstáculo, uno superior y otro inferior.

¿Ya está? ¿Ya tenemos al juego spawneando obstáculos? No, aún no. Tenemos el script preaparado con esa funcionalidad, pero **aún tenemos que pasarle por inspector la referencia al prefab que tiene que spawnear**. Si no hacemos esto el juego fallará porque no sabrá qué es lo que tiene que spawnear.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

> ¡Cuidado! ¡Un fallo muy típico es asignarle al generador un obstáculo que ya esté en la escena! Si haces esto, cuando ese obstáculo ya no esté porque lo hayamos borrado, se quedará sin objeto original que copiar y el programa fallará. **Asegúrate de arrastrar el prefab del proyecto, no un obstáculo de la escena.**

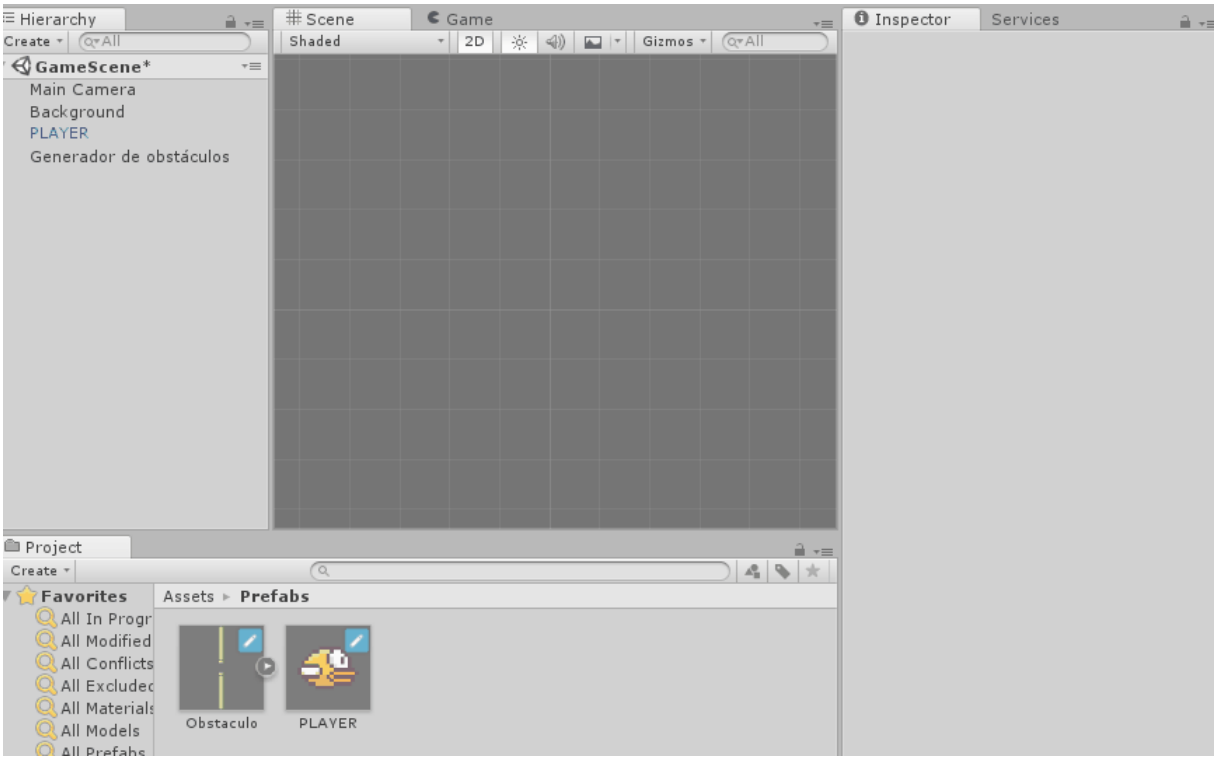
Player.cs - Que el jugador cuente puntos

Para contar los puntos necesitamos detectar cuando pasamos entre los dos obstáculos, y en ese momento hacer `puntos = puntos + 1`.

Para detectar cuando pasa el jugador por un sitio se hace mediante **un collider con el que realmente no colisionas físicamente, sino que lo atraviesas. Es decir, un trigger.** Esto es aplicable a objetos del suelo que atraviesas (las armas del Call of Duty o la selva del League of Legends, por ejemplo...), a los puntos de control en cualquier videojuego, a sitios por los que al pasar se ejecuta una cinemática o se activa una trampa, etc.

Cambiar prefab del obstáculo: trigger invisible para contar

Por tanto necesitamos que nuestro obstáculo tenga un 3er Game Object hijo: un trigger invisible en el centro de los dos colliders. Para ello abre el prefab en la escena, añádele un Game Object vacío, ponle un collider, márcalo como trigger y aplica los cambios al resto de prefabs.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

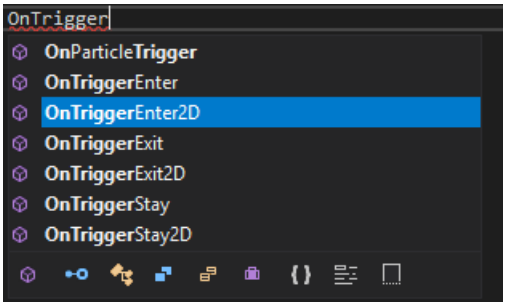
Mejorar código del Player.cs: sumar puntos al pasar por el trigger

Para decirle a un script que ejecute un código al entrar en un trigger, hay que poner el código que queramos ejecutar dentro de un método especial (como Start y como Update) llamado **OnTriggerEnter**.

```
void OnTriggerEnter2D(Collider2D other)
{
    // Your code here
}
```

Este método recibe como argumento el collider del trigger en el que estamos entrando, pero esa información no necesitamos utilizarla ahora.

Estos métodos especiales no necesitamos recordarlos, ya que los podemos encontrar fácilmente en la documentación oficial de Unity. Además, Visual Studio también nos los recomendará al comenzar a escribirlos.



Sabiendo esto, ya solo tenemos que declarar una variable puntuación como atributo, que esté inicializada a 0, y programar que OnTriggerEnter sume 1 punto a la puntuación actual.

Para poder verlo, tendremos que mostrar la puntuación por algún sitio, ¿no? De momento, hazlo por consola. **Para mostrar un texto por consola se hace mediante el método** `print ("texto a mostrar)"`.

```
void OnTriggerEnter2D(Collider2D other)   // Al pasar por un trigger
{
    puntos = puntos + 1;      // sumar 1 punto
    print ("Puntos actuales: " + puntos);  // e imprimir puntos totales en consola
}
```

Como ya nuestro generador está spawneando obstáculos con el trigger en medio, que es lo que hemos hecho en la sección anterior, este código será suficiente para que al pasar por ellos se impriman por consola los puntos actuales del jugador.

Player.cs - Mostrar puntos del jugador en UI

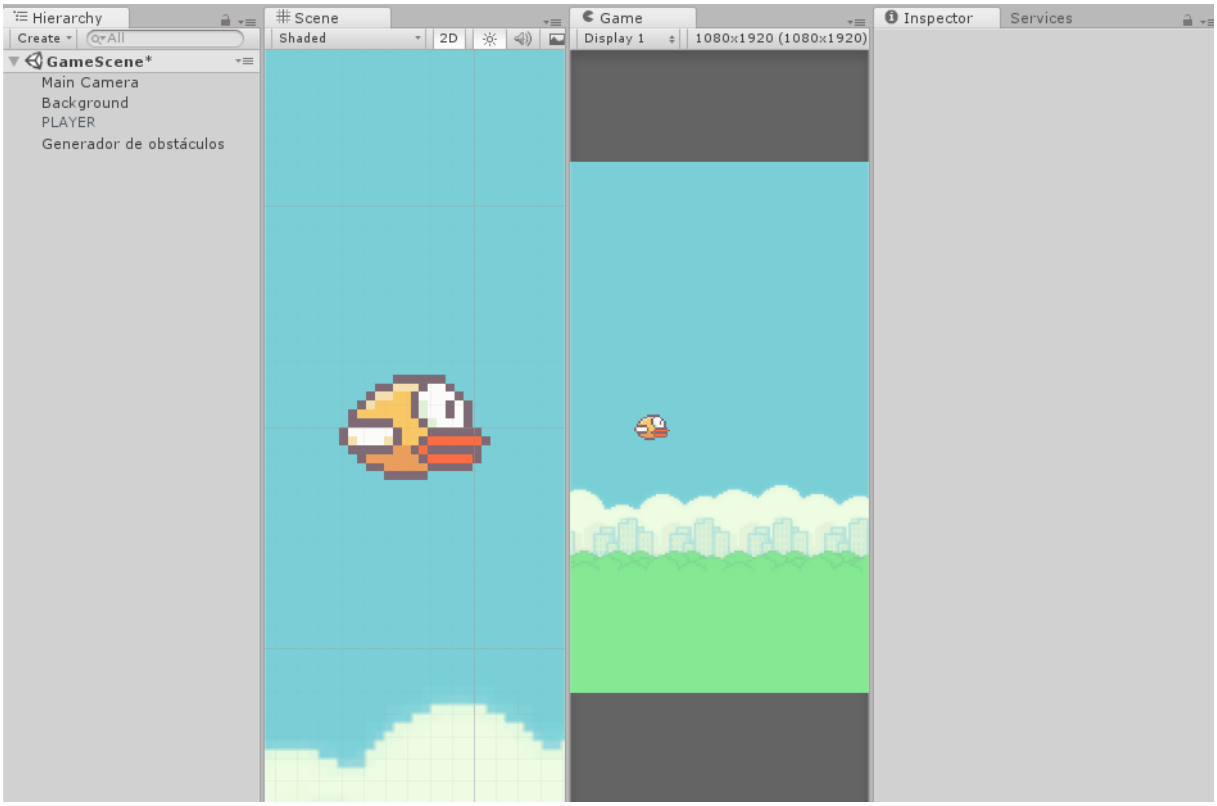
Mostrar los puntos en la consola no es lo más ideal. **Los mensajes en consola sólo deben utilizarse para hacer pruebas, ya que solo se ven en el editor de Unity y no en el juego final.** Vamos a poner un texto en pantalla y después le diremos al jugador que muestre sus puntos en ese texto.

Preparar UI

Para crear un texto en pantalla se hace mediante **Create > UI > Text**. Esto nos crea tres Game Objects nuevos:

- Un Game Object “Canvas”, que es una cámara especial que solo renderiza las cosas de la interfaz que son hijas de este Game Object.
- Un Game Object “Text”, hijo de Canvas, que es el texto que hemos creado.
- Un Game Object “Event System” responsable de que funcionen algunas cosas automáticamente, como detectar clicks en botones.

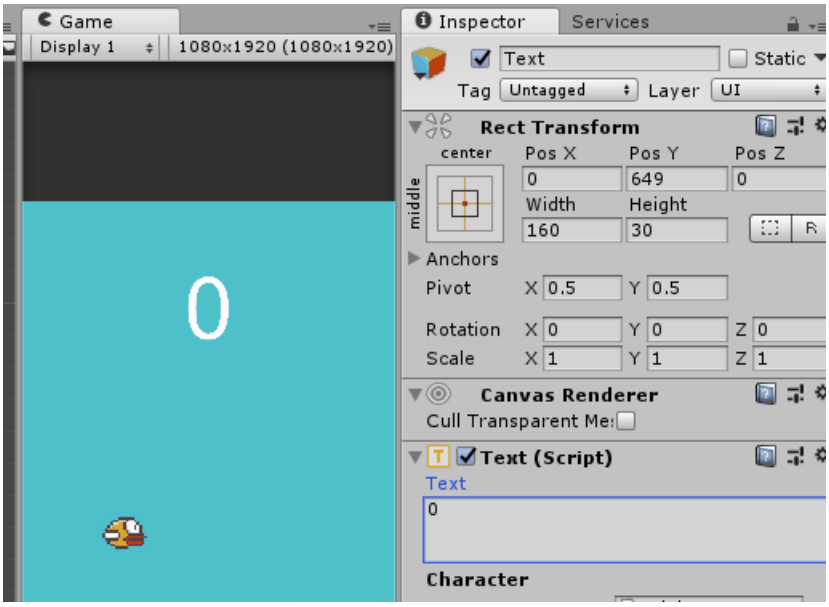
Coloca el texto de tus puntos en el lugar de pantalla que quieras, dale un tamaño, una alineación de párrafo y un color adecuados, inicializa el texto a “0” (ya que será lo que aparecerá al principio) y setea los overflow horizontal y vertical en “overflow” para que pueda ser tan grande como quieras.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

¿Qué necesitamos que haga el programa?

Con el texto ya preparado, tan solo tendremos que darle acceso al Player.cs a ese texto para que lo modifique él mismo, al igual que podemos hacer nosotros desde el inspector.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

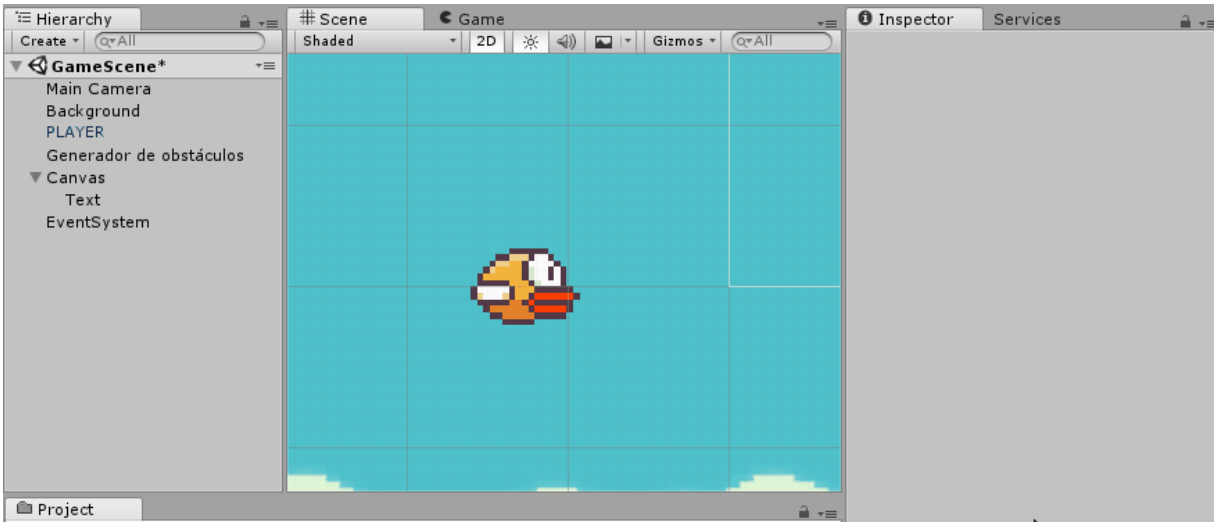
Para ello el Player.cs necesitará saber cuál es el texto que tiene que modificar.

¿Cómo le decimos que haga eso?

Al igual que el script del generador necesita una referencia pública al Game Object del prefab que tiene que spawnear, a partir de ahora nuestro script Player.cs necesitará también una referencia pública al componente Texto que hemos modificado nosotros directamente desde el inspector.

Pero primero, para que el script del jugador sea capaz de comunicarse con la interfaz, es necesario decirle al script, al principio del todo, que vamos a utilizar esa parte del motor gráfico, añadiendo la línea `using UnityEngine.UI;` al resto que nos trae por defecto.

Tras eso, ya seremos capaces de declarar la variable Text pública y asignársela por el inspector. **Vamos a hacerlo ahora, antes de que se nos olvide.**



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

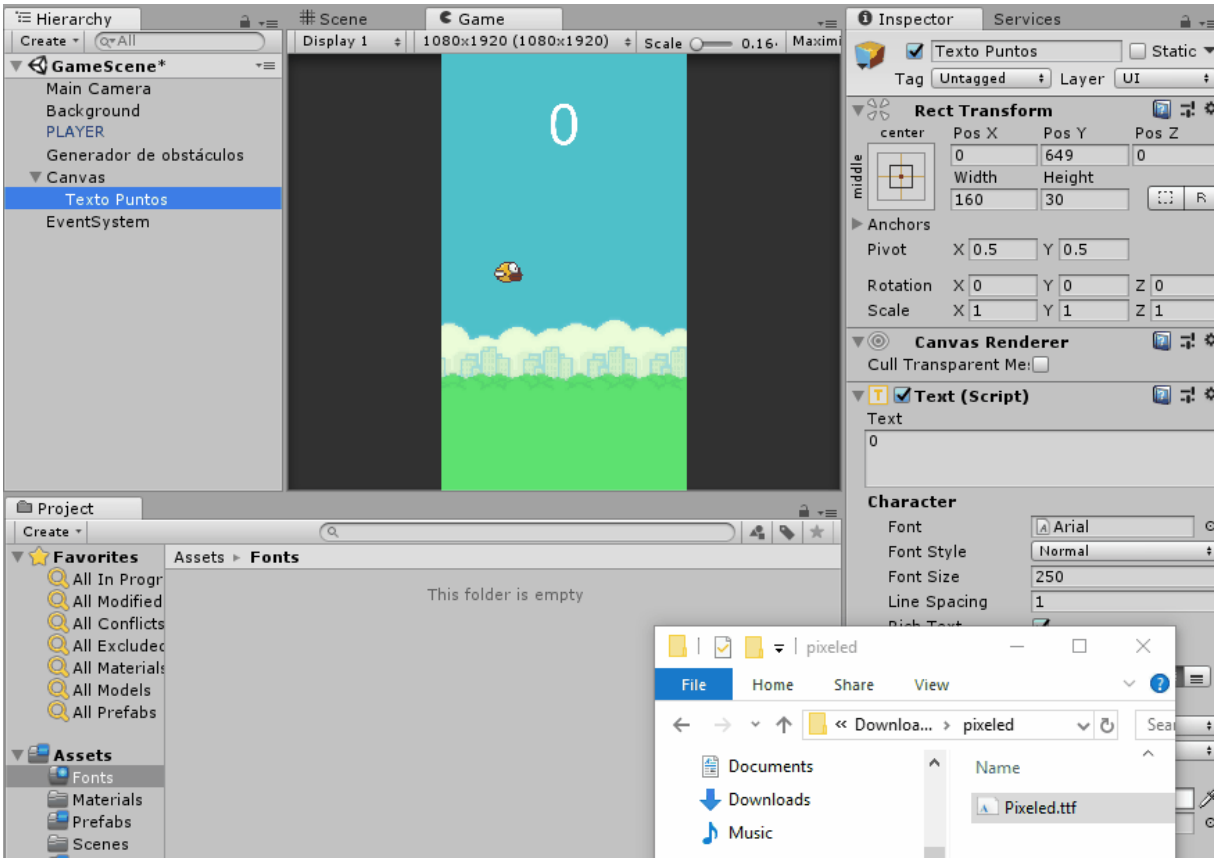
Hecho esto, **aún no funciona como queremos**, porque aún no le hemos dicho que muestre los puntos en ese texto en lugar de en la consola. Para ello, vamos a modificar lo que ocurre OnTriggerEnter para que imprima los puntos en ese texto en lugar de por pantalla. Para ello:

```
void OnTriggerEnter2D(Collider2D other)   // Al pasar por un trigger
{
    puntos = puntos + 1;      // sumar 1 punto

 miTextoPuntosUI.text = puntos.ToString(); // y escribir los puntos en el texto de la UI

    // print ("Puntos actuales: " + puntos);
}
```

El método ToString() convierte nuestra variable puntos a string de forma sencilla.

Cambiar fuente

Cambiar la fuente del texto de los puntos es muy fácil. Solo necesitas importar al proyecto el archivo .ttf de tu fuente y asignarlo en el inspector.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Obstacle.cs - Que el obstáculo desaparezca pasado un tiempo

Ahora mismo, los obstáculos siguen moviéndose hacia la izquierda hasta el infinito. Podemos decirle al Script del obstáculo que se autodestruya pasado un tiempo con el método `Destroy(objeto que destruir, cuándo destruirlo)`.

Para referirnos al Game Object del objeto en el que estamos, no necesitamos más que escribir en el código `gameObject`. Tan fácil como acceder a la posición mediante la `transform.position`, ¿verdad?

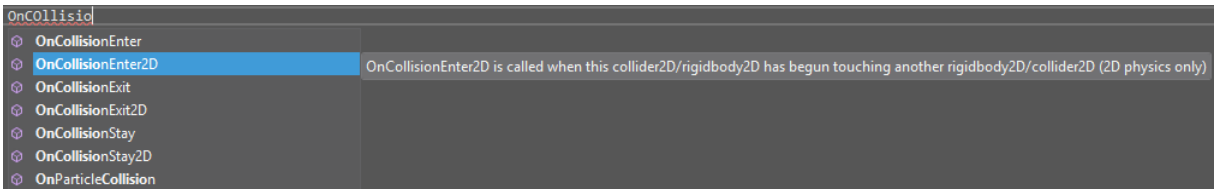
Modifica el script del obstáculo para que al inicio (en `Start()`) se programe para autodestruirse a sí mismo pasado un número de segundos configurable desde el inspector (tendrás que darle tiempo suficiente para que cruce la pantalla y se esconda por la izquierda).

```
Destroy(gameObject, timeToDestroy);
```

Player.cs - Que el jugador muera

¿Cuándo queremos que muera el jugador? Cuando colisione. ¿Dónde escribimos el código para que se ejecute sólo cuando el jugador ha chocado con algo? **Dentro del método OnCollisionEnter.**

Al igual que con OnTriggerEnter, no tenemos que recordar el nombre de este método, ya que Visual Studio nos lo recomendará automáticamente:



Aprovechando que hemos aprendido a destruir los obstáculos pasado un tiempo, vamos a destruir al jugador cuando choque, inmediatamente, mediante `Destroy(gameObject, 0.0f)` o, lo que es lo mismo: `Destroy(gameObject)`.

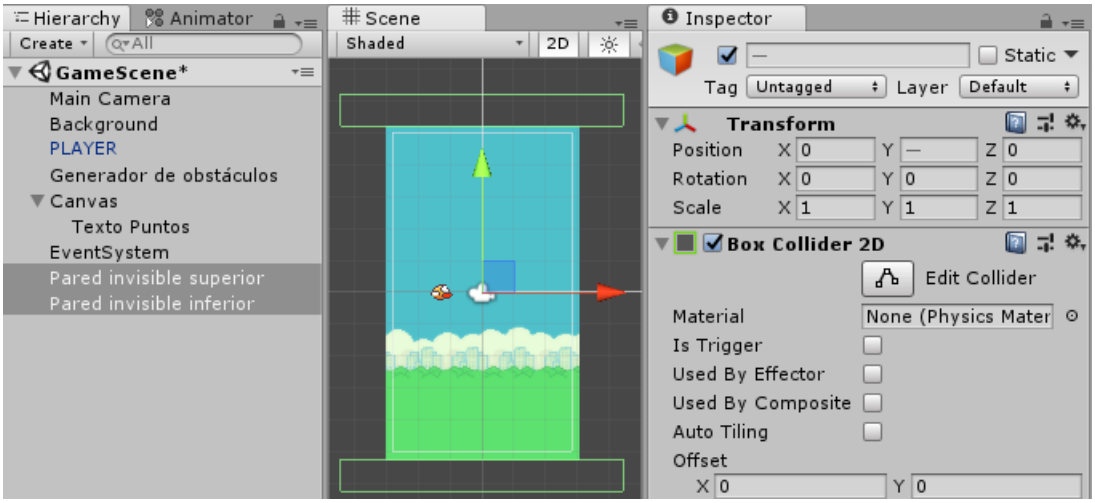
Al final, nuestro método OnCollisionEnter nos quedaría así:

```
void OnCollisionEnter2D(Collision2D collision)  // Al chocar con otro collider
{
    Destroy(gameObject);       // destruir el gameObject del jugador
}
```

De momento **no es nada bonito**, ya que simplemente desaparece el jugador al chocar y no hay ningún tipo de feedback visual o sonoro de que has muerto. Más adelante mejoraremos este punto, incluyendo alguna animación o explosión y mostrando la pantalla de Game Over.

Paredes invisibles

Ahora que el Player muere cuando choca con algo, es el momento de poner paredes invisibles alrededor del escenario, para evitar que se salga por arriba o que caiga hasta el infinito. Para ello, simplemente añade 2 Game Objects con colliders arriba y abajo, y listo.



Player.cs - Reiniciar escena al morir el jugador

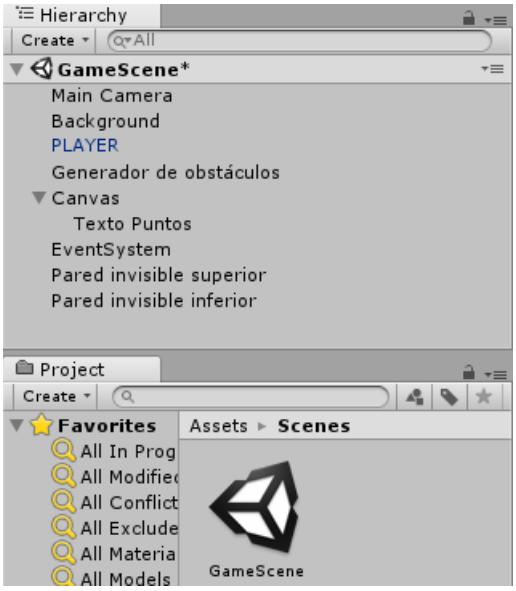
Ahora mismo nuestro jugador desaparece cuando choca con algo y las tuberías siguen yendo hacia la izquierda infinitamente, y se deja de poder jugar.

Vamos a arreglar esto. En el momento en que el jugador choque, vamos a reiniciar la escena, directamente.

Al igual que con la interfaz, tenemos que decirle al script que va a utilizar la parte del motor encargada de cargar los niveles. Para ello hay que añadir al principio del todo una nueva línea: `using UnityEngine.SceneManagement;`.

Tras esto, cuando detectemos la colisión, seremos capaces de usar el siguiente método: `SceneManager.LoadScene("nombre de la escena")`.

Como podemos ver tanto en la ventana de jerarquía como en la vista de proyecto, en la carpeta de Scenes, la escena de nuestro juego se llama “GameScene”:



Por tanto, bastaría con programar: `SceneManager.LoadScene("GameScene")`. Pero entonces, ¿qué pasa si le cambiamos el nombre al archivo de la escena? Que no funcionaría.

Podemos encadenar el código de cargar escena con `SceneManager.GetActiveScene().name`, que nos devuelve el nombre de la escena activa en ese momento.

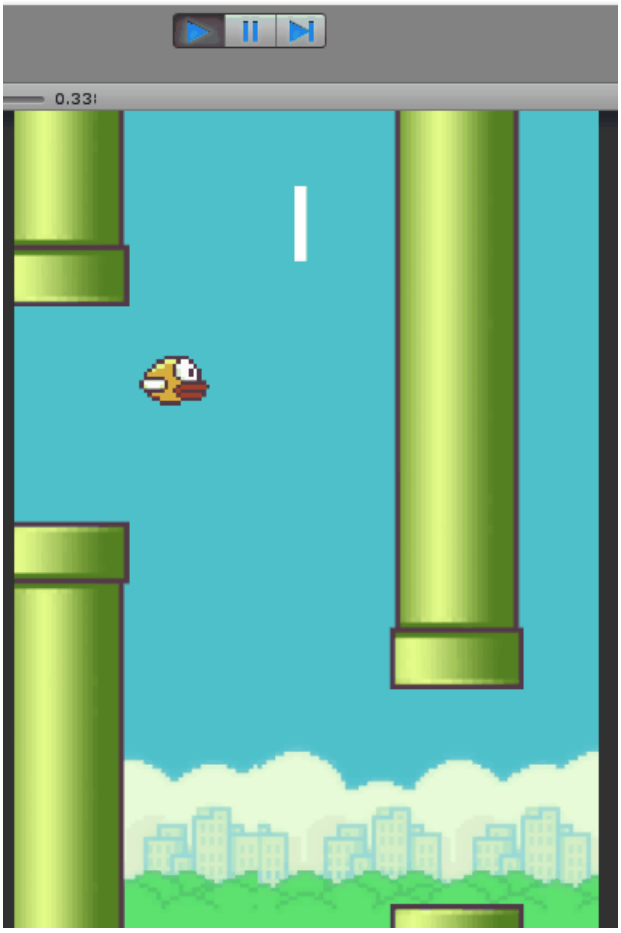
De esta forma, nos quedaría `SceneManager.LoadScene(SceneManager.GetActiveScene().name)`, que carga de nuevo la escena abierta en ese momento.

Al final, el código de OnCollisionEnter nos queda como:

```
void OnCollisionEnter2D(Collision2D collision)
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);

    //Destroy(gameObject);
}
```

Hemos comentado el código que hace que el jugador se destruya ya que la escena se reinicia instantáneamente y no da tiempo a destruirlo. Como no se llega a ver, para qué ponerlo, ahorramos recursos.

De esta manera, ya al menos podemos jugar continuadamente:



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Guardar highscore

De qué sirve un juego difícil si no podemos picar a los jugadores. Vamos a aprender a guardar una variable highscore que se almacene en la memoria del teléfono o del ordenador y que perdure entre sesiones de juego, de forma que si una noche al irte a dormir te haces 25 puntazos, al día siguiente y aunque hayas apagado el teléfono, al abrir la aplicación de nuevo sigas teniendo ahí delante tus 25 puntos para presumir de ellos.

Para guardar datos en memoria normalmente hacen falta complejos sistemas de guardado y autoguardado que representen completamente el estado de la partida de un jugador, desde el inventario en un juego de rol, o la posición en el mundo en el momento de guardar, hasta las skins desbloqueadas por cada jugador en un juego online cualquiera.

¿Qué datos queremos guardar en nuestro caso? Simplemente un número entero, representando el highscore. ¿Y cómo lo vamos a hacer? No nos complicaremos la vida: utilizaremos un sistema que Unity nos facilita para guardar datos en el registro del sistema operativo: las PlayerPrefs. No es lo más seguro del

mundo, pero sí lo más fácil de aprender y útil a bajo nivel. Cuando tengáis el juego listo, os reto a que intentéis hackearos a vosotros mismos.

Antes de continuar, recomiendo ver este tutorial de Brackeys, que explica justo lo que vamos a ver ahora de forma visual y detallada. Brackeys es un youtuber que se dedica a hacer videotutoriales cortos y sencillos sobre Unity. Como él hay muchos, pero su contenido es de garantía de calidad gracias a que está patrocinado oficialmente por Unity.

PlayerPrefs: cómo guardar datos permanentes entre sesiones de juego

Al igual que **Input**, **Time** o **SceneManager**, **PlayerPrefs** es una clase estática a la que podemos acceder desde cualquier parte del código. En nuestro caso concreto vamos a ver cómo guardar y cargar un número entero en las PlayerPrefs. El código para ello sería el siguiente:

Guardar dato de tipo entero

```
PlayerPrefs.SetInt("NombreEnteroPersistente", enteroDeMiCódigoQueQuieroGuardar);
```

La variable que pasemos como segundo argumento al método de SetInt de PlayerPrefs, tiene que ser la variable de tipo entero de nuestro código que queremos almacenar en memoria. En nuestro caso, nosotros querremos almacenar la puntuación del jugador.

Cargar dato de tipo entero

```
int enteroDeMiCódigo = PlayerPrefs.GetInt("NombreEnteroPersistente");
```

El método GetInt de PlayerPrefs nos devuelve el entero que guardamos anteriormente llamado “NombreEnteroPersistente”, o 0 en caso de no haber guardado nada anteriormente. Para que el número que nos devuelve no se quede en el aire, tendremos que asignarlo a alguna variable o hacer algo con él. En nuestro caso, lo mostraremos primero por consola, y más adelante en un texto en la interfaz.

Para cargar otro tipo de datos

- `SetFloat / GetFloat`
- `SetString / GetString`

Ambos con la misma nomenclatura que la explicada para guardar y cargar datos enteros.

Para borrar los datos anteriormente guardados

- `DeleteAll()`
- `DeleteKey("NombreVariableQueQuieroBorrar")`

Player.cs - Guardar highscore en las PlayerPrefs

Por tanto, ¿en qué parte de nuestro código tendremos que programar que se guarden los puntos del jugador **si y solo si hemos superado el récord anterior**? ¿En qué parte tiene sentido hacer esto? ¿Está claro no? **Guardaremos el highscore (en caso de haber superado la marca anterior) en el momento en que el jugador muere**.

```
void OnCollisionEnter2D(Collision2D collision)
{
 // Antes de guardar el highscore comprobamos si hemos superado el récord anterior
    if (PlayerPrefs.GetInt("highscore") < puntos)
    {
        PlayerPrefs.SetInt("highscore", puntos);
    }

    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    //Destroy(gameObject);
}
```

Con este código ya lo tienes guardado, pero no deberías confiar en mí. Compruébalo tú mismo. Imprime en la consola el highscore actual. Puedes imprimir ese valor cuando quieras, pero tendría sentido sobre todo al inicio del juego o al morir.

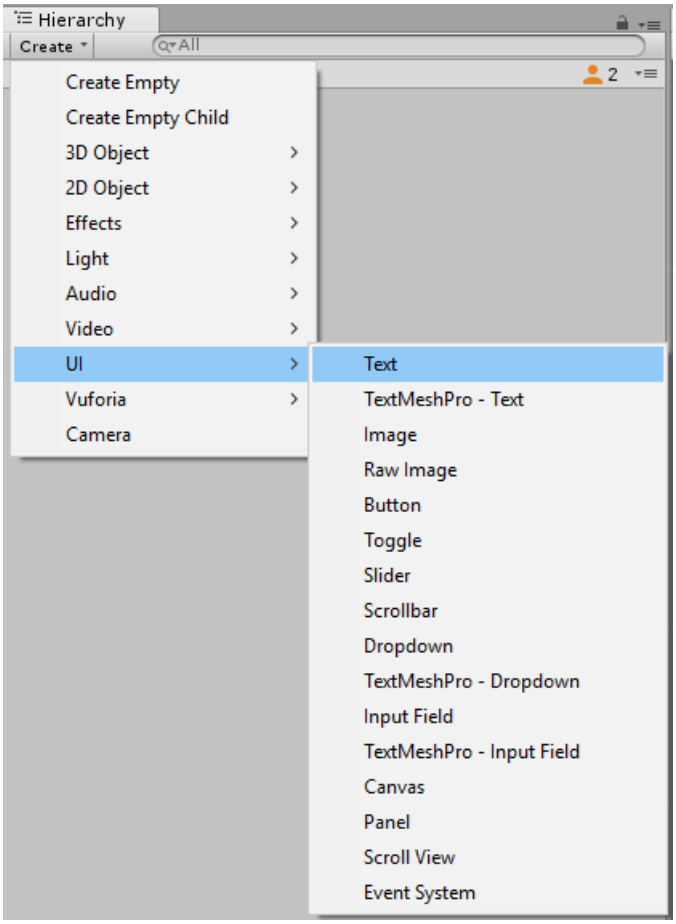
```
// Imprime el highscore al morir o al comenzar
// con este código para comprobar que está funcionando
print (PlayerPrefs.GetInt("highscore"));
```

UI en Unity: Crear menú principal

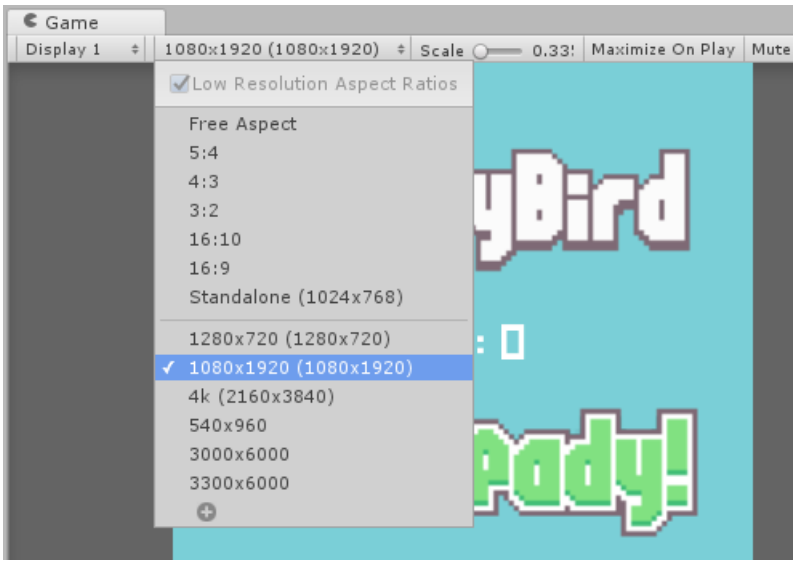
Las pantallas de interfaz como el menú principal o los menús de pausa suelen ir en escenas diferentes, a no ser que, por ejemplo, tu juego sea un juego online y por tanto quieres que la ventana de pausa no congele el juego en realidad.

Entonces, crea una nueva escena y ponle un nombre apropiado (por ejemplo MainMenu).

Para crear elementos de interfaz se hace bajo el menú de Create > UI. Al crear cualquier elemento de UI, si tienes un Canvas te lo creará dentro de él, y si no lo tienes te creará automáticamente un Canvas y un EventSystem.



¡Muy importante! Antes de ponerte a diseñar la interfaz, vuelve a asegurarte de que en la ventana de juego tienes configurada una resolución de aspect ratio 9:16 que es común a la mayoría de los teléfonos móviles (por ejemplo, 1080x1920 píxeles).



Desde el principio estamos diseñando el juego pensando en este tipo de forma de pantalla o de ventana vertical. Con las interfaces no será diferente. **Si haces una interfaz preciosa pero luego cambias de resolución, se te descuadrará todo**. Desgraciadamente no basta con tener buen gusto, también hay que saber hacer interfaces responsivas, es decir, que se adapten a distintas resoluciones.

En el caso de nuestro juego nos curaremos en salud forzando a que se juegue con la pantalla del móvil en vertical o a que la ventana de Windows no sea redimensionable.

Un par de secciones después explicaremos cómo hacer una interfaz responsiva. Aunque, si tienes pensado pararte un rato a conseguir la interfaz de tus sueños, entonces adelántate a esa sección para hacerla desde el principio adaptable a distintas resoluciones.

Elementos básicos de interfaz

Create > UI >

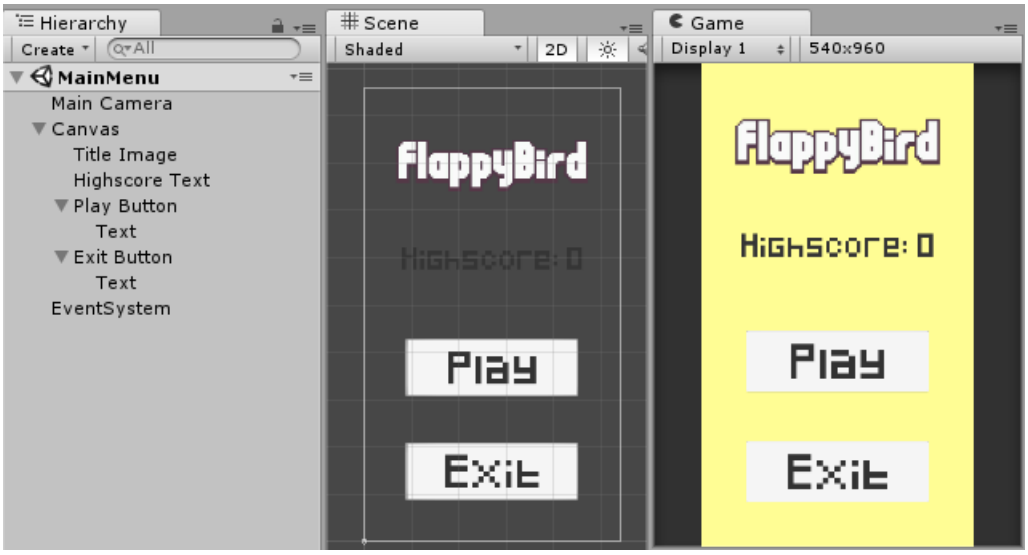
Button: lo usaremos para empezar a jugar y para salir del juego.
Text: lo usaremos para mostrar el highscore (y como texto dentro de los botones).
Image: lo usaremos para mostrar el logo en el menú principal.
Panel: equivalente a una ventana, en la que dentro puedes meter cosas, como más botones y textos.
Canvas: creado automáticamente, representa toda la interfaz. Todos los objetos de interfaz tienen que estar dentro de él. Es como una cámara especial que solo renderiza elementos de interfaz.
EventSystem: creado automáticamente, se encarga de que funcionen los clicks y el táctil en móviles.

Dos botones, un texto y una imagen

Como personalices tu interfaz lo dejo a tu elección, y a tus ganas de pelearte con el (al principio) anti-intuitivo sistema de interfaces de Unity. Lo que sí necesitaremos, al menos, será:

- Imagen de título
- Texto para el highscore
- Botón de jugar
- Botón de salir

En mi caso, mi interfaz me ha quedado así:



Juega con los diferentes componentes de los GameObjects que ves en la jerarquía para cambiar las opciones de cada uno. Por ejemplo:

- A la imagen de título le he dicho que preserve su aspect ratio original para que no se deforme.
- A los textos les he cambiado la font, los he alineado al centro (y por geometría) y me he asegurado de que no se cortan configurando su propiedad “Overflow”.
- A la cámara le he dicho que renderice el fondo de la escena de color amarillo.

Dale al play y comprueba como los botones responden a los clicks aunque todavía no hagan nada. Bonito, ¿verdad? Ahora solo falta que funcione.

MainMenu.cs - Añadir funcionalidad al menú

¡¡Script nuevo!! Necesitamos que nuestro menú principal haga cosas. Para ello, creamos un script MainMenu.cs y se lo añadimos al Canvas, que es un buen punto central que engloba todo el menú.

¿Qué cosas necesita hacer el menú?

Cambiar a la escena de juego al darle a Play
Salir de la aplicación (o cerrar la ventana en Windows) al darle a Exit.
Escribir en el texto del highscore el highscore real, en lugar de 0.
¡Bonus level! También pasar a la escena de juego al pulsar espacio.

MainMenu.cs - Programar la funcionalidad

Primero vamos a programar la funcionalidad, y después le diremos a los botones que ejecuten la funcionalidad que hemos programado. Para poder decirle a un botón que haga algo, necesitamos encapsular lo que queremos que el botón haga dentro de un método. Es decir, que tendremos que crear dos

métodos en MainMenu.cs, uno para el botón Play y otro para el botón Exit, pero hasta que no configuremos los botones de la interfaz para llamar a estos métodos realmente no estarán haciendo nada. **El código estará ahí, solo falta ejecutarlo**.

Necesitaremos una referencia al texto del highscore `Text highscoreText`, que tendremos que arrastrársela por inspector tal y como hicimos con el texto de los puntos del jugador en la escena de juego. Para ello también tendremos que decirle al script que vamos a usar la interfaz, `using UnityEngine.UI`.

Al inicio (en el `Start()`) leeremos el highscore guardado en memoria y lo asignaremos al texto.

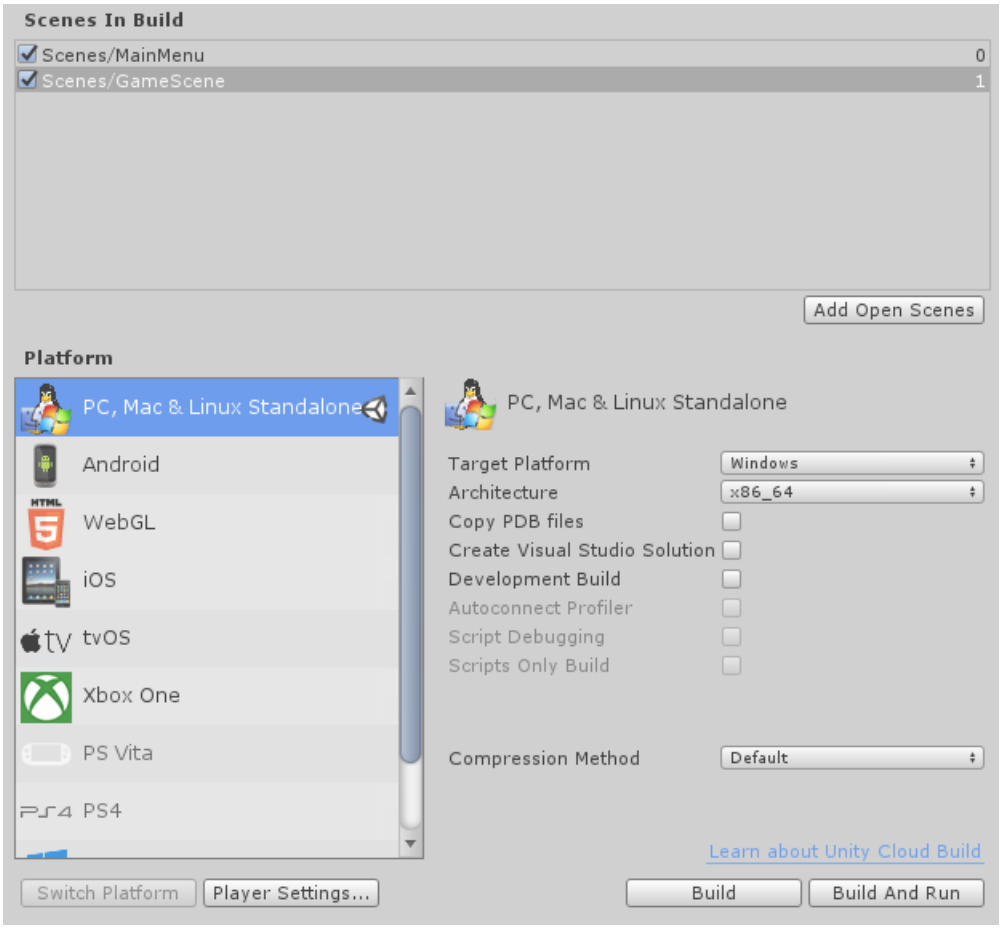
Necesitaremos un método `Play()` que llamaremos después al clickar el botón Play. Este método cargará la escena de juego tal y como hacía el jugador al morir, solo que en lugar de pasar como argumento al `SceneManager.LoadScene(...)` el nombre de la escena activa, le pasaremos el nombre de nuestra escena de juego.

Necesitaremos un método `CloseApp()` que llamaremos después al clickar el botón Exit. En este punto del proyecto ya conocías todos los ingredientes para hacer este script excepto este: el código necesario para salir de la aplicación. Para cerrar la aplicación (en caso de Android) o cerrar la ventana en Windows, tenemos que llamar al código `Application.Quit();`.

El script completo nos debería quedar así:

```
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class MainMenu : MonoBehaviour
{
    public Text highscoreText;

    void Start() // Actualizar texto interfaz al inicio
    {
        highscoreText.text = "Highscore: " + PlayerPrefs.GetInt("highscore");
    }

 // Bonus Level: crear aquí un método Update() y, fijándote en como lo hicimos en Player.cs,
 // programa aquí que también comience el juego si pulsamos la tecla espacio,
 // para que sea más ágil volver a jugar de nuevo

    public void Play()
    {
        SceneManager.LoadScene("GameScene");
    }

    public void CloseApp()
    {
        Application.Quit();
    }
}
```

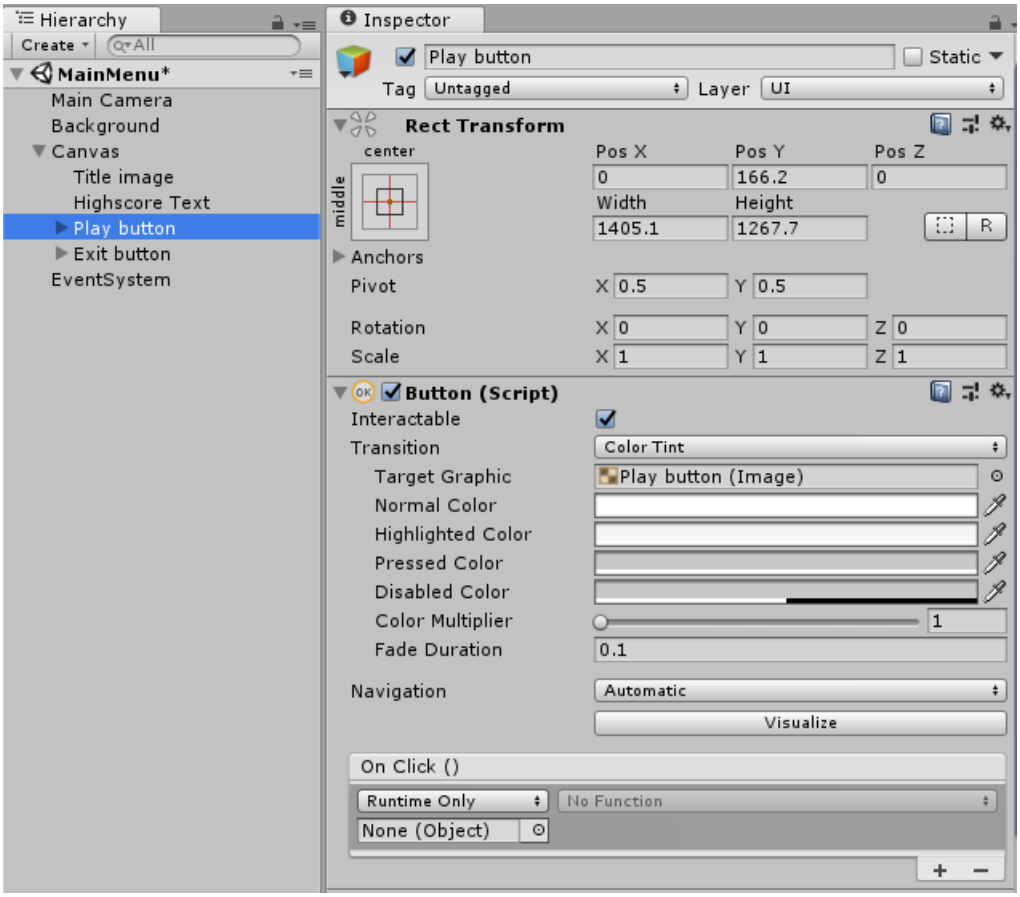
Queda una última cosa para que este script funcione: para que el SceneManager pueda cargar la escena de juego, es necesario decirle qué escenas van a estar incluidas en el proyecto. Esto se hace desde el menú de **File > Build Settings**, arrastrando al cuadro de "Scenes in build" las dos escenas de nuestro juego.



No es necesario tocar nada más en esta pantalla, basta con cerrarla. Volveremos aquí más adelante para hacer la build final del juego (es decir, exportar nuestro proyecto de Unity como un juego final de verdad).

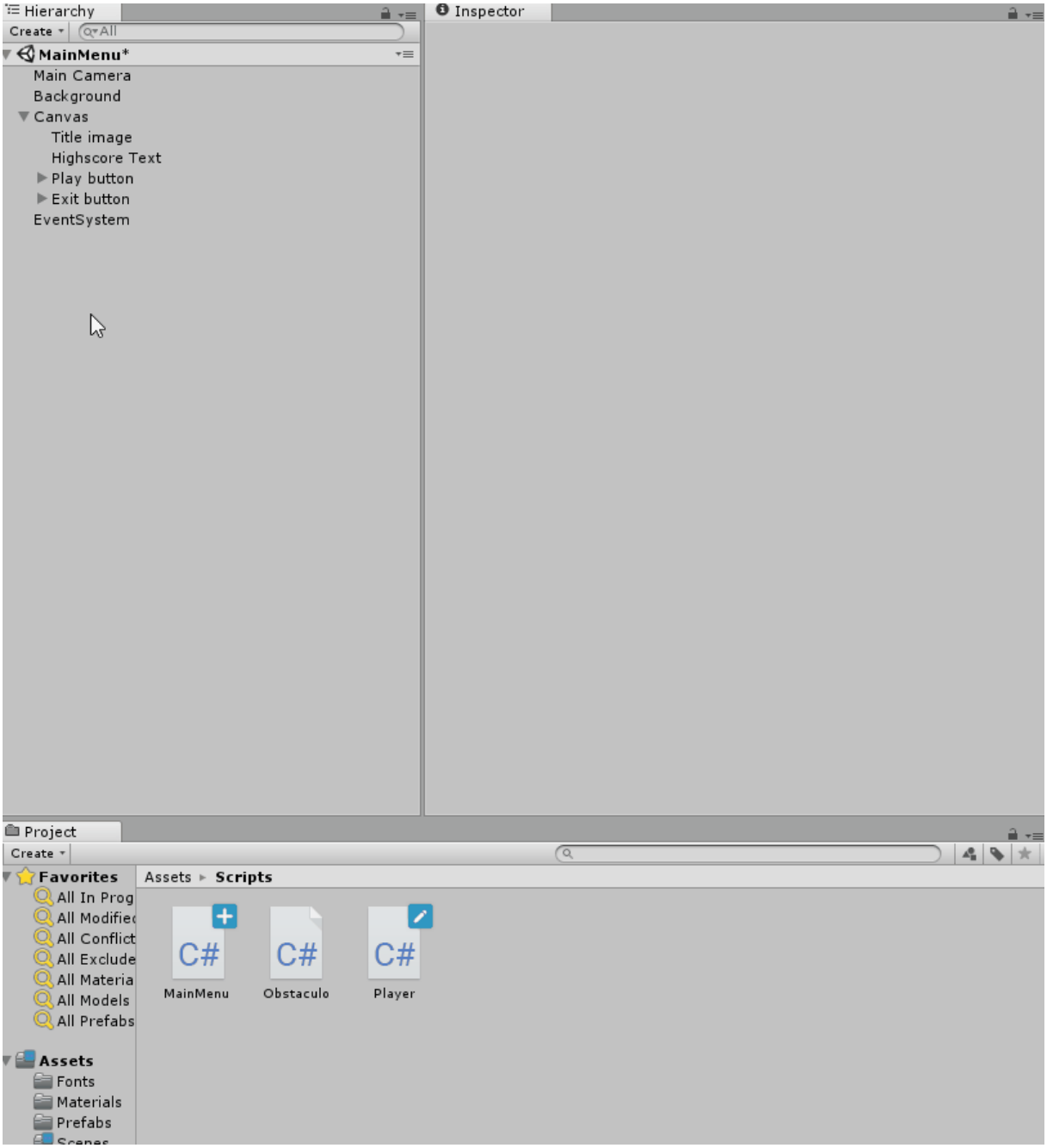
MainMenu.cs - Conectar script con botones

Como hemos dicho, `Play()` y `CloseApp()` todavía no están siendo llamados por nadie, por lo que ese código no se está ejecutando. Para que se ejecute, tendremos que decirle a los botones que lo hagan ellos cuando reciban un click.



Como podemos ver en el inspector de mi “Play button”, al final del componente “Button” hay un evento “On Click ()” que de momento no está haciendo nada (No Function, None…).

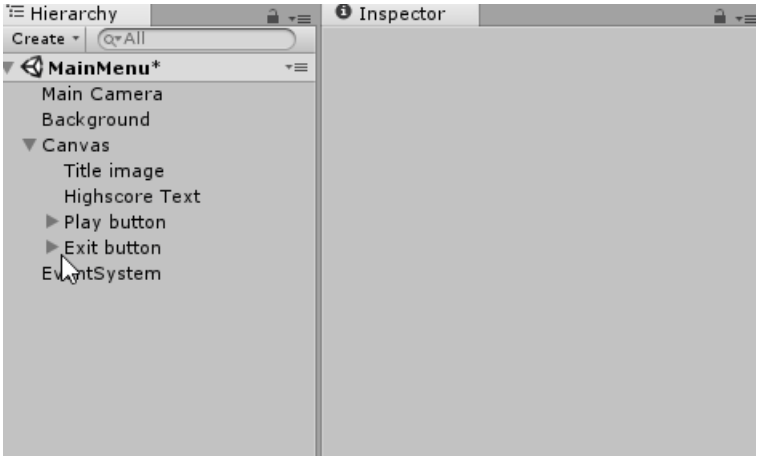
Vamos a añadirle al Canvas el script MainMenu.cs que hemos programado, que como dijimos es un buen punto central, y después le vamos a decir al “Play button” que llame al script que hemos añadido. Esto se hace arrastrando el objeto que tiene el script al evento y seleccionando en la lista desplegable el método del script que queremos ejecutar.



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Ya solo nos faltaría hacer lo mismo con el botón de Exit, pero conectándolo a nuestro otro método.

¡Y también recuerda asignarle al script el texto del highscore para que sea capaz de actualizarlo!



* La imagen de arriba es un gif, no podrás verla en PDF. Ve a http://rpmi.metinu.com/ para verla en movimiento.

Player.cs - Volver al menú principal al morir

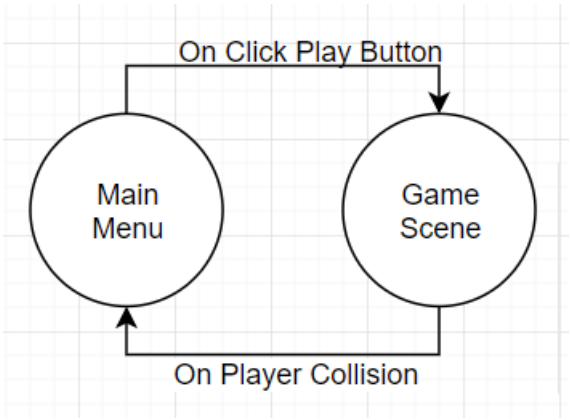
En este punto, nuestro juego comenzaría en la escena del menú, y al darle al botón de Play comenzaríamos el juego, pero... ¿Qué pasa cuando morimos? Ahora mismo reiniciamos la escena cuando chocamos, por lo que nunca podremos volver al menú principal...

```
// Así deberíamos tener el código del Player.cs en este momento:
void OnCollisionEnter2D(Collision2D collision)
{
    if (PlayerPrefs.GetInt("highscore") < puntos)
    {
        PlayerPrefs.SetInt("highscore", puntos);
    }

    SceneManager.LoadScene(SceneManager.GetActiveScene().name);  // Reiniciar escena actual
    //Destroy(gameObject);
}
```

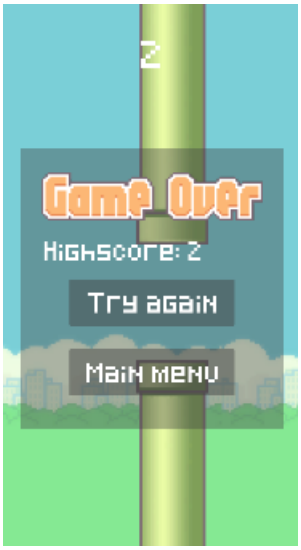
Para volver al menú principal en lugar de reiniciar el juego, bastaría con cambiar la escena que cargamos por la del menú principal, en lugar de que sea la escena activa.

```
[...]
SceneManager.LoadScene("MainMenu");
[...]
```

En este punto, ya tenemos un flujo de juego simplísimo y un mínimo de navegabilidad entre pantallas de juego:



Bonus Level: ventana de Game Over



Si quisieras crear una **ventana de Game Over que se abriera al morir y que tuviera dos botones, uno para reiniciar y otro para volver al menú**, ¿cómo lo harías? Con lo aprendido hasta ahora te será fácil. Aquí te dejo una pequeña guía:

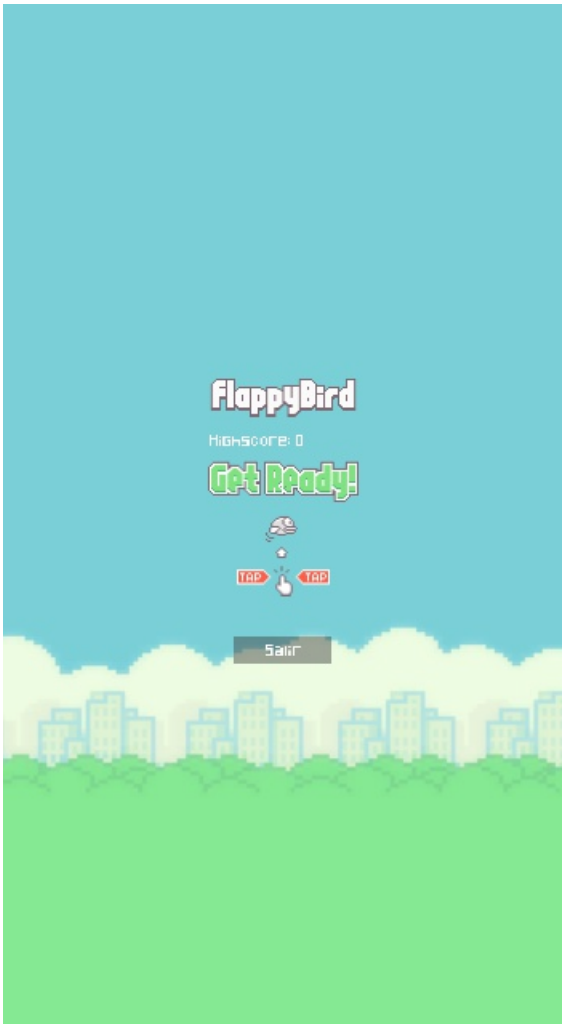
- Necesitarás un **elemento de interfaz "Panel"** para meter dentro de él los botones, el highscore, la imagen del jugador muerto... Por supuesto, al principio dejarás todo el panel desactivado, para que sea invisible.
- Necesitarás un **nuevo script GameOver.cs** muy parecido al MainMenu.cs, con dos métodos, uno para cada botón, que reinicien la escena de juego y que vuelvan al menú principal, respectivamente. Además, también tendrá que actualizar su propio texto de highscore. ¡Y no olvides conectarle los botones!
- Cuando el jugador choque y muera, **ya no querrás volver al menú, sino mostrar este panel**. Para ello te será útil saber que un GameObject cualquiera puedes activarlo utilizando el método `miReferenciaAUnGameobject.SetActive(true)`.
- Cuando tengas todo funcionando, te darás cuenta de que **tu jugador sigue saltando si pulsas espacio aunque esté muerto** y la pantalla de muerte se haya mostrado. Necesitarás comprobar antes de saltar,

no solo si ha pulsado espacio sino también si está vivo o muerto. Para esto sirven las variables booleanas true/false.

Interfaz responsiva (adaptable a distintas resoluciones)

Ya tenemos hecha la interfaz, pero probablemente nos encontraremos con un problema: cuando cambiamos a resoluciones más grandes, como la de un Samsung Galaxy S por ejemplo (1,440x2,960), puede que no se nos descuadre todo pero se nos verá mucho más pequeño y no nos cabrá el dedo en el botón.

Si no nos damos cuenta de esto durante el desarrollo nos daremos cuenta después, al abrir la aplicación por primera vez en nuestro móvil. Probablemente se nos verá algo así:



(Sí, este es otro ejemplo de menú diferente al que enseñé antes. Y qué.)

Eso es debido a que el tamaño de la interfaz se mide en píxeles a menos que le expecifiquemos lo contrario.

Y si un botón mide 500 píxeles de ancho, cuando nuestra pantalla tenga de ancho 2000 píxeles en lugar de 1000, notaremos la diferencia (se verá mucho más pequeño).

Unity tiene documentación sobre los pasos a seguir para conseguir una interfaz todoterreno. Léete su artículo oficial para comprender más a fondo lo que se explica aquí: Unity: Diseñando UI para Múltiples Resoluciones.

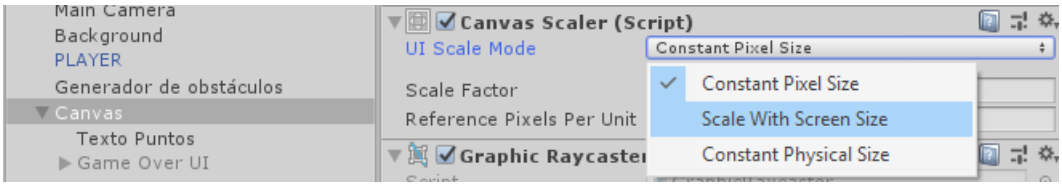
Modificando las opciones del Canvas Scaler

En nuestro caso adaptar la interfaz será muy sencillo gracias a que sólo nos preocuparemos de móviles en vertical, y no de móviles en horizontal, tablets en horizontal y vertical, monitores anchos, cuadrados y antiguos, 4k…

Principalmente **tendremos que editar las opciones del componente "Canvas Scaler"** asignado automáticamente a nuestro Game Object "Canvas", que es donde reside la interfaz.

Constant pixel size VS Scale with screen size

Lo primero y más importante, d**ecirle a Unity que escale la interfaz con el tamaño de la pantalla en lugar de con el tamaño de los píxeles**. Una vez marcada esta opción, todas las resoluciones con el mismo aspect ratio serán prácticamente idénticas.



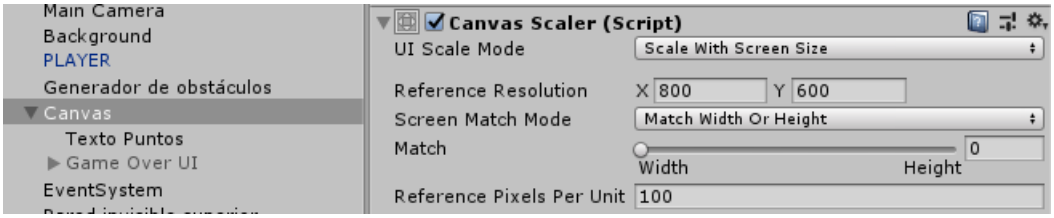
Pruébalo tú mismo cambiando las resoluciones de la ventana de Game. Si entre resoluciones con el mismo factor de forma la interfaz no se agranda y achica, es que lo has hecho bien.

En realidad lo ideal hubiera sido diseñar la interfaz desde el principio con esta opción ya configurada, pero si no te enfrentas a un problema nunca conocerás su solución.

Match width VS match height

Si con cambiar el UI Scale Mode todavía se te ve la interfaz demasiado pequeña o demasiado grande, juega con las propiedades Reference Resolution y Match.

En la resolución de referencia lo ideal es configurar el mismo valor de nuestra ventana de Game, 1080x1920, y diseñar la interfaz en base a esa resolución. Pero como no lo hemos hecho así desde el principio, cambia los valores hasta que la interfaz que tenías se adapte correctamente.

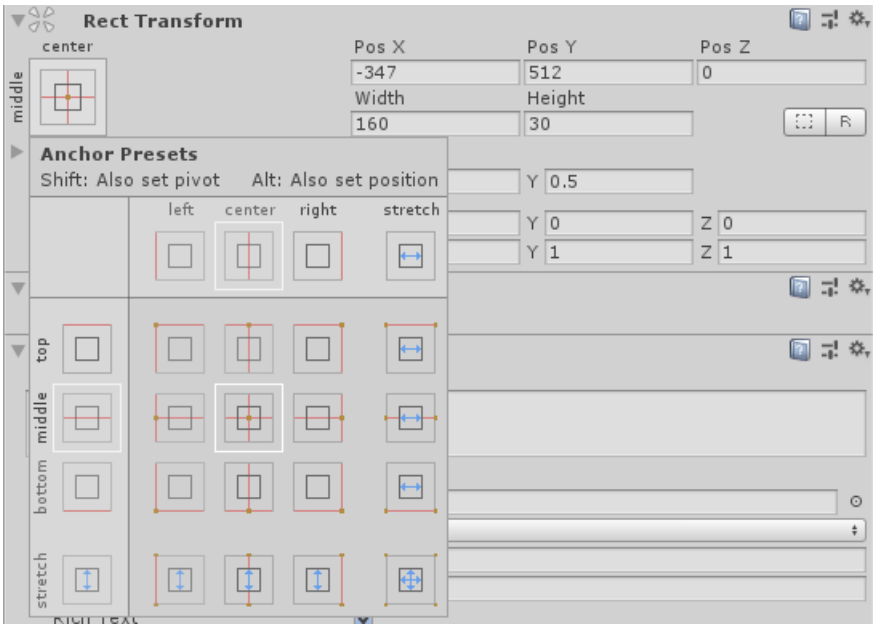


¿Te has fijado en que si cambias el valor de la Y realmente no está cambiando nada? Eso es porque la propiedad Match está al 100% inclinada hacia escalar en base a la anchura, es decir, en base a la X. Lee la documentación oficial de Unity antes mencionada para entenderlo mejor.

¡Por cierto! Estas opciones tendrás que modificarlas en cada uno de tus canvas, tanto en el del Main Menú como en el de la escena de juego, aunque solo tengas un número para la puntuación. ¿No querrás que ese número se te vea enano en los móviles de última generación, no?

Posicionar UI mediante anclas

Nos falta solo una opción más que no nos ha hecho falta tocar hacer la interfaz responsiva, pero que también es muy útil. De nuevo, en la documentación oficial de Unity lo explican mucho mejor y más visualmente. Lo explicaremos brevemente: se trata de posicionar los elementos de interfaz en base a anclas:



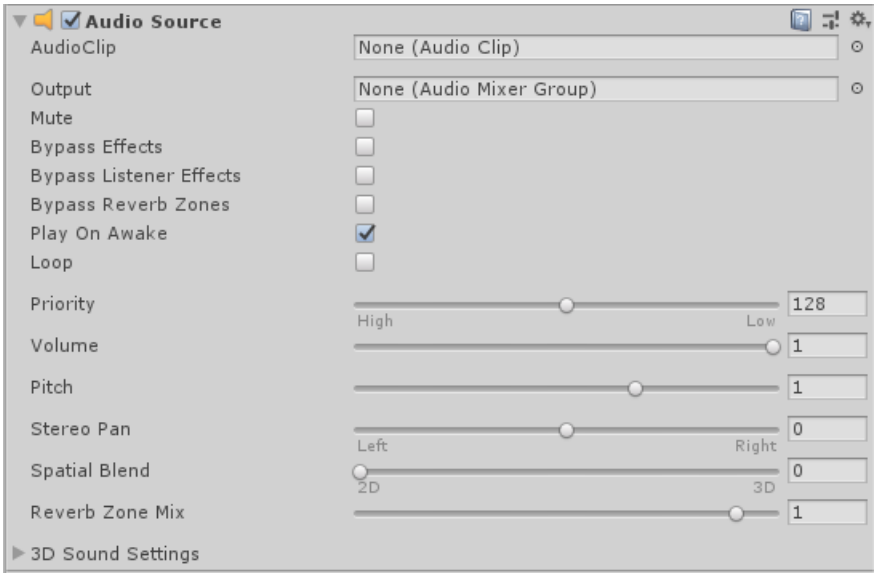
De momento, todos los elementos de la interfaz los hemos posicionado en base al centro de la pantalla, que es la opción por defecto. ¿Pero y si nosotros colocamos una imagen desplazada hacia abajo y a la izquierda, como un minimapa? Ocurrirá que en resoluciones más grandes, se quedará alejada de la esquina, que es donde queremos que esté. Por tanto, lo ideal sería colocarla en base a la esquina inferior izquierda de la pantalla, y no en base al centro.

En nuestro caso nos salvamos de preocuparnos por esto porque nuestro juego solo es apto para resoluciones móviles y verticales, pero si quisiéramos adaptarlo a más pantallas tendríamos que utilizar estas opciones.

Background music

> Me ahorraré la explicación sobre cómo importar música y sonidos al proyecto. Es tan sencillo como arrastrar y soltar a la ventana de proyecto cualquier sonido que queramos importar. Unity soporta diversos formatos, aunque los más recomendables son, por orden de preferencia *.wav*, *.ogg* y *.mp3*.
>
> Crea una nueva carpeta "Audio" para guardar en ella los sonidos.

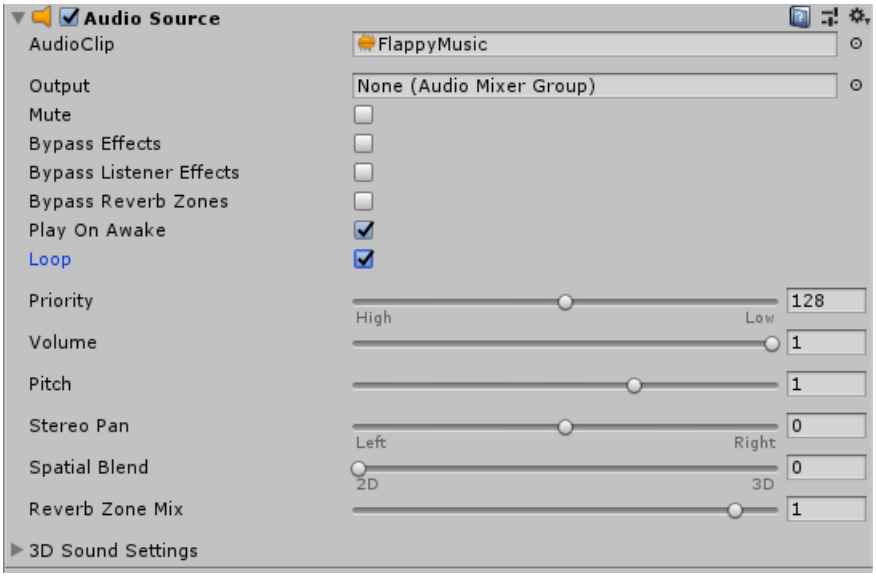
En Unity, para reproducir cualquier sonido necesitamos un componente llamado **Audio Source**, que es el responsable de reproducir audio.



Las opciones más interesantes del Audio Source, o al menos las que nosotros utilizaremos, son las siguientes:

- **AudioClip**: aquí puedes arrastrar cualquier archivo de sonido que hayas importado previamente al proyecto
- **Play On Awake**: reproduce el sonido desde que el juego comienza. Querremos tildar esta casilla para la música.
- **Loop**: vuelve a reproducir el sonido inmediatamente tras acabar. También lo necesitaremos para la música.
- **Volume**: aquí podremos ajustar el volumen de este emisor (por tanto, de la música).

Por tanto, para reproducir la música del fondo, necesitaremos algo así:



¿A qué Game Object le añadimos el Audio Source? La ubicación realmente no importa, ya que al ser sonido 2D se escuchará por igual esté donde esté. Pero para mantener el proyecto ordenado, posiciónalo en un Game Object central y relevante, que sepas que siempre estará ahí como el Canvas, el Background o la MainCamera. **Otra buena opción es crear un nuevo Game Object para la música**.

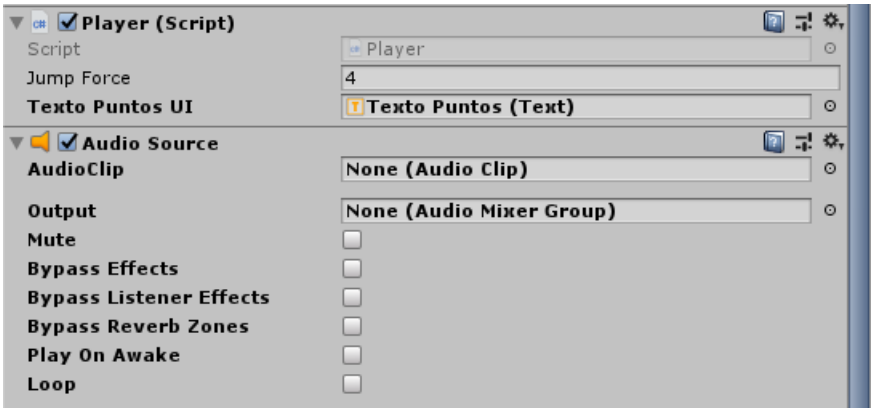
¡Listo! ¡Y sin escribir una sola línea de código! Ya tenemos música en la escena de juego. Si quieres, puedes hacer lo mismo para el menú principal pero con otra canción diferente, más tranquila.

Player.cs - Reproducir sonidos al saltar, morir, ganar punto...

Ahora sí que nos tocará programar...

Antes de seguir, te recomendaré este otro tutorial de Brackeys, donde explica un sistema para reproducir sonidos mucho más versátil, útil, fácil de usar y reutilizable que el que explicaremos en esta sección. Yo lo explicaré de forma más sencilla, para empezar.

Antes de nada, necesitaremos un nuevo componente Audio Source emisor de audio, que lo tendrá equipado el game object del jugador, que será él el que reproducirá los sonidos (**aunque tendremos que decirle que lo haga mediante código**).



Esta vez no queremos asignarle ningún sonido en concreto como Audio Clip, ya que en unos momentos querremos reproducir uno y en otros, otro.

Tampoco queremos que lo reproduzca al comenzar el juego, por lo que dejamos la casilla "Play On Awake" desmarcada.

Para poder utilizar el Audio Source en el código, tendremos que inicializarlo al comenzar el juego (es decir, en `Start()`), al igual que hacíamos con el Rigidbody para poder saltar después.

```
[...]
Rigidbody2D miRigidbody;
AudioSource emisorAudio;

// Use this for initialization
void Start()          // al inicio
{
    miRigidbody = GetComponent<Rigidbody2D>();  // dame mi Rigidbody asociado
    emisorAudio = GetComponent<AudioSource>();  // y dame mi AudioSource asociado
}
[...]
```

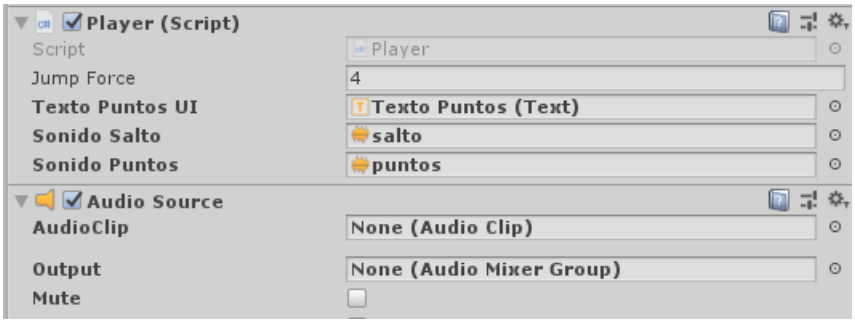
A partir de aquí, en cualquier parte del código podremos decirle al emisor de audio que emita un sonido con el método `emisorAudio.PlayOneShot(miAudioClip)`.

Sonidos al saltar y ganar puntos

Como el método `emisor.PlayOneShot(audioClip)` recibe como parámetro una variable de tipo Audio Clip (al igual que el Audio Source por inspector, tendremos que declararle al jugador tantas variables públicas de tipo Audio Clip como sonidos queramos ser capaces de reproducir después.

```
public class Player : MonoBehaviour
{
    public float jumpForce = 10.0f;
    public Text textoPuntosUI;
    public AudioClip sonidoSalto;
    public AudioClip sonidoPuntos;

    int puntos = 0;
    Rigidbody2D miRigidbody;
    AudioSource emisorAudio;
 [...]
}
```

Hecho esto, **ahora podemos (y debemos) asignarle al jugador sus sonidos desde el inspector**.



Ya estamos listos para decirle al código que reproduzca los sonidos con `emisor.PlayOneShot(..)`. ¿Cuándo querremos reproducir cada uno? ¿En qué momento del código?

El sonido del salto lo reproduciremos en el momento en que salta (si pulsamos espacio):

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))     // si pulsamos espacio
    {
        miRigidbody.velocity = Vector2.up * jumpForce;  // saltamos
        emisorAudio.PlayOneShot(sonidoSalto);    // y reproducimos sonido del salto
    }
}
```

Y el sonido de los puntos lo reproduciremos si pasamos por un trigger (OnTriggerEnter2D), en el mismo momento en que sumamos los puntos:

```
void OnTriggerEnter2D(Collider2D other)
{
    puntos = puntos + 1;      // sumamos puntos
    emisorAudio.PlayOneShot(sonidoPuntos);  // reproducimos sonido de puntos
    textoPuntosUI.text = puntos.ToString();  // actualizamos puntos en la interfaz
    //print("Mi puntuación es: " + puntos);
}
```

Sonido al morir

Hasta ahora, estamos volviendo al menú principal en el momento de chocar, a no ser que hayas hecho también el bonus level de la interfaz y le hayas incluido una ventana de Game Over.

Como estamos reiniciando la escena inmediatamente, no da tiempo a reproducir un sonido de muerte. Esto se puede arreglar de dos formas:

1. En lugar de reiniciar la escena directamente, introducir un delay (retraso) con el método `Invoke("nombreMétodo", delay)`.
2. Hacer el bonus level de la ventana de game over

¡Cuidado! Tampoco podemos destruir el Game Object del jugador como hacíamos antes, ya que si lo hacemos destruiremos con él el Audio Source y dejaremos de ser capaces de emitir sonidos.

Player.cs - Saltar también con controles táctiles

Sonidos, interfaces, muerte... ¡Estamos listos para exportar a *.apk* (para Android) o a *.exe* (para Windows) y ya tendremos el juego listo! Falta pulir un par de detalles, pero, en cuanto a gameplay...

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))     // si pulsamos espacio
    {
        miRigidbody.velocity = Vector2.up * jumpForce;  // saltamos
        emisorAudio.PlayOneShot(sonidoSalto);    // y reproducimos sonido del salto
    }
}
```

¿Cómo tenías pensado que el jugador pulsara la tecla espacio en una pantalla de móvil?

Vamos a modificar el código para que compruebe si hemos pulsado la pantalla táctil. Es un código algo complejo, ya que utiliza arrays y enumerados, que son cosas que todavía no hemos visto. El código se quedaría así:

```
void Update()
{
 // si pulsamos espacio
 // ó tocamos la pantalla con un dedo
 // y además ese dedo acaba de tocar la pantalla ahora
 // (para que no salte infinito si mantienes)
    if (Input.GetKeyDown(KeyCode.Space)|| Input.touchCount == 1 && Input.touches[0].phase == T
    {
        miRigidbody.velocity = Vector2.up * jumpForce;  // saltamos
        emisorAudio.PlayOneShot(sonidoSalto);    // y reproducimos sonido del salto
    }
}
```

Qué línea de código más larga y fea ese *if*, ¿verdad? Vamos a dejarlo más bonito encapsulándolo en su propia función:

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space) || PantallaTocada())
    {
        miRigidbody.velocity = Vector2.up * jumpForce;  // saltamos
        emisorAudio.PlayOneShot(sonidoSalto);    // y reproducimos sonido del salto
    }
}

bool PantallaTocada()
{
 if(Input.touchCount == 1 && Input.touches[0].phase == TouchPhase.Began)
 {
  return true;
 }
 else
 {
  return false;
 }
}
```

Ahora se lee mejor, ¿eh? **Si pulso espacio o he tocado la pantalla, entonces salta**.

Player.cs - Y ya que estamos, saltar también con click izquierdo

¡Venga! ¿Por qué no? El código necesario para ver si hemos pulsado el click izquierdo es `Input.GetMouseButtonDown(0)`. Añádelo al if para que también tenga en cuenta el click:

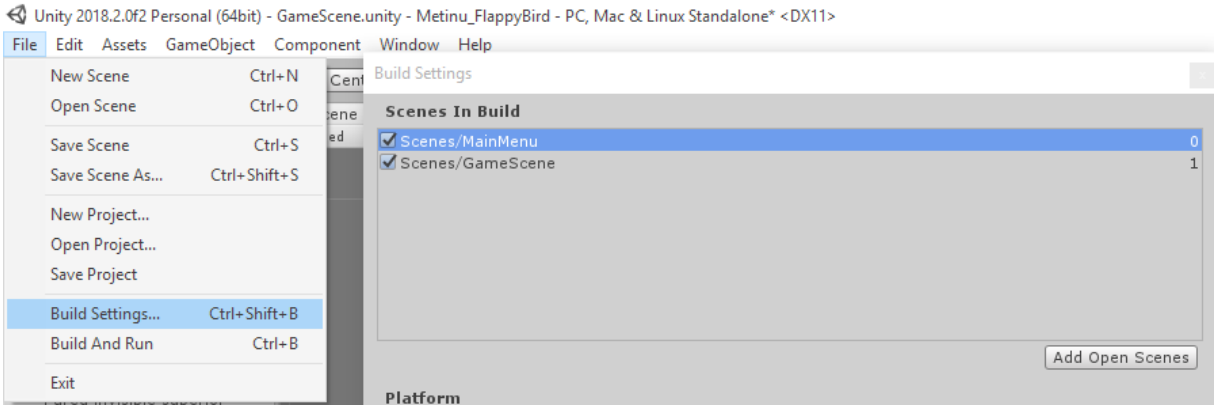
```
// si pulso espacio, toco la pantalla o hago click
if  (Input.GetKeyDown(KeyCode.Space)  ||  PantallaTocada() || Input.GetMouseButtonDown(0))
{
 // saltar
}
```

Hacer la build

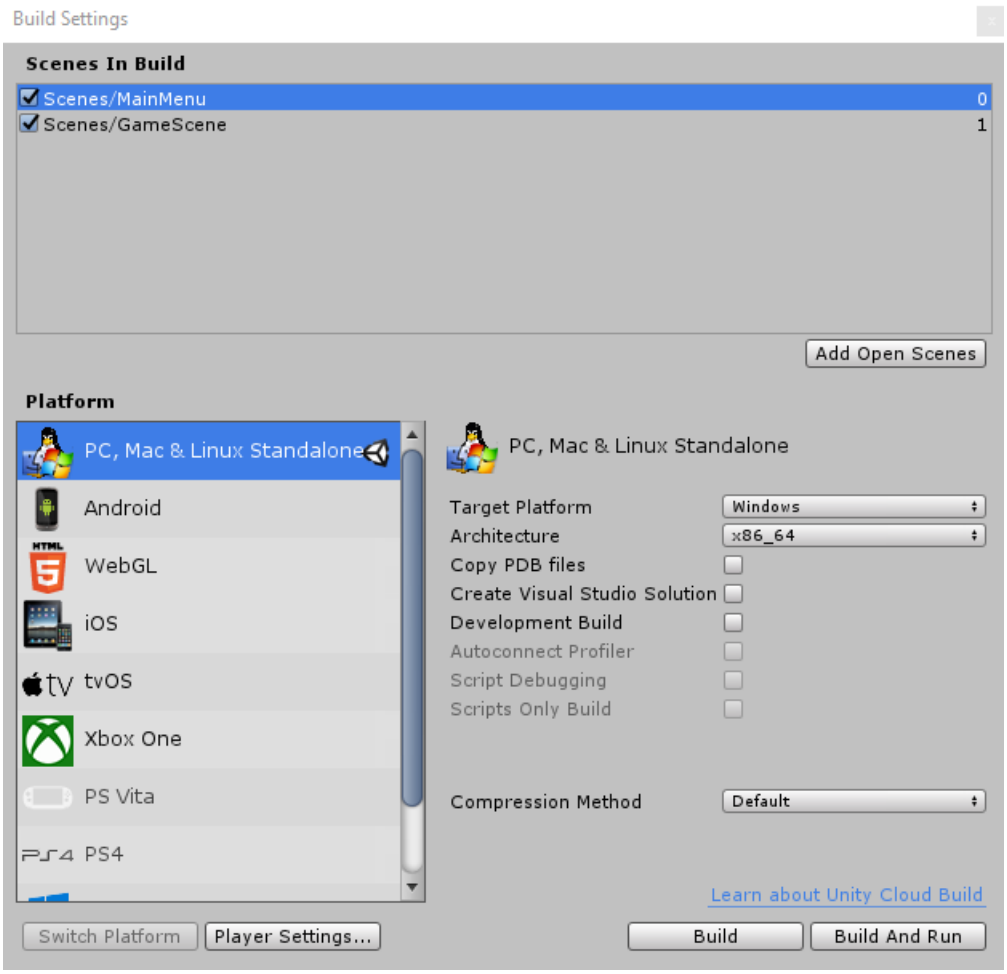
Ahora sí. Se acabó el código. Tenemos el juego 100% funcionando, es el momento de hacer la build.

> Una "build" de un juego es el juego en su versión final, listo para ser jugado por cualquiera sin necesidad de tener Unity instalado ni conocimientos técnicos. En Windows es un archivo *.exe*, en Android es un archivo *.apk*, y en la Play Station 4 es todo lo que haya grabado dentro del blu-ray.

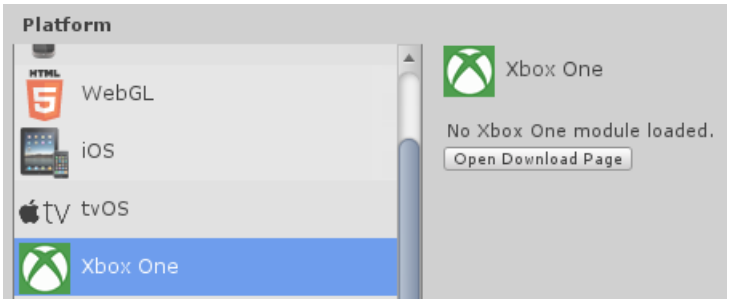
Abre de nuevo la ventana que abriste para incluir la escena de Main Menú en el proyecto, **en File > Build settings...**



En esta ventana, **seleccionas la plataforma a la izquierda y clickas en "Build"**, y listo. Pero no tan rápido. Hay un montón de cosas que tenemos que tener en cuenta primero, algunas de ellas imprescindibles.



Para que te deje elegir una plataforma, necesitas tener esa parte de Unity instalada. Si no la tienes, sigue los pasos de Unity para descargar ese módulo.



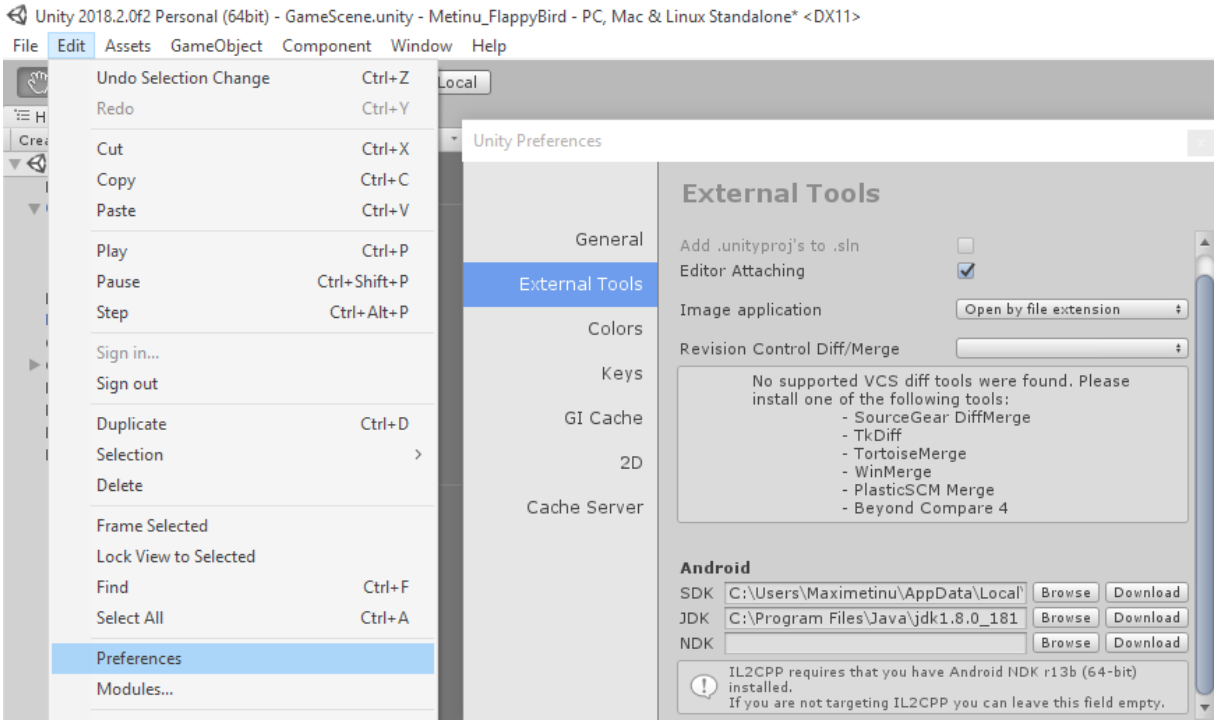
A continuación, tendrás que configurar algunas cosas clickando en "**Player Settings...**". Pero antes, probablemente tengas que instalar un par de cosas necesarias para poder exportar a Android, ya que no basta solo con el módulo de Android para Unity, sino que también necesitas los kits de desarrollador.

Kit de desarrollor de Java y Android (JDK y SDK)

El kit de desarrollador de Java se llama JDK (Java Development Kit), y el de Android SDK (Software Development Kit). Son librerías de código que necesita tener instaladas tu sistema operativo para poder

exportar a la plataforma Android.

Tendrás que instalarlas y especificarle a Unity su ubicación. Para abrir la página de descarga de cada uno, abre la ventana de preferencias de Unity y clicka en “Download” al lado de cada uno.



Para curarnos en salud, instálalos en este orden: **JDK Primero y SDK después**.

Mientras que Java JDK no notarás ni que lo tienes instalado, ya que es útil para ejecutar otros programas que usan Java pero por sí solo no es una aplicación, Android SDK viene junto con la instalación de Android Studio, un programa editor de código equivalente a Visual Studio, pero para escribir código para Android. Tú no necesitarás usar este programa para nada, pero es necesario instalarlo para tener el SDK.

* En realidad sí que puedes descargar el SDK independientemente, pero al no tener un instalador como el de un programa, no lo tendrás instalado en la ruta por defecto y Unity no sabrá encontrarlo si no le dices donde buscar.

Las rutas por defecto de cada uno son

JDK: `C:\Program Files\Java\jdk1.8.0_181`
SDK: `C:\Users\<tu_usuario>\AppData\Local\Android\sdk`
(sustituye <tu_usuario> por tu nombre de usuario en Windows)

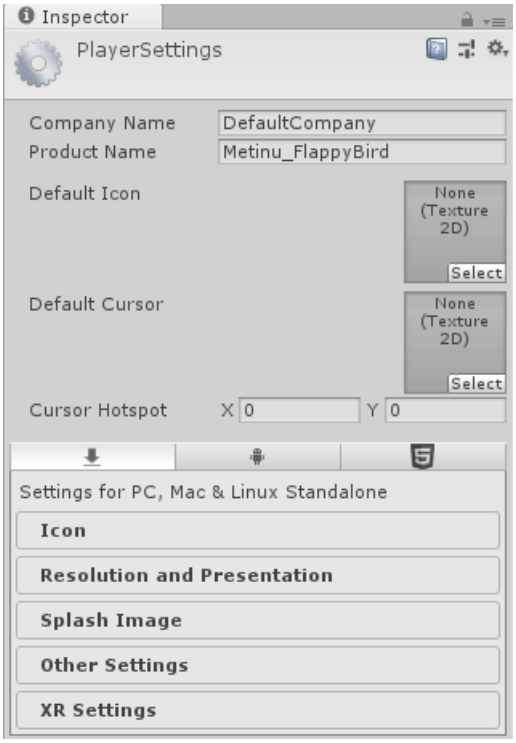
Igualmente no necesitas especificárselas a Unity, ya que las encontrará él solo en el momento de hacer la build.

Player settings comunes de Android y Windows

Hay que modificar una serie de opciones antes de exportar. Abre de nuevo la ventana de File > Build

settings… y clica en “Player settings…”. En la ventana del Inspector, se te abrirán las opciones de la Build.

Algunas cosas son comunes, pero abajo tienes las pestañas de las plataformas incluidas en tu proyecto (en mi caso son Windows, Android y HTML5, pero tú probablemente no tendrás esta última). Clicando en cada plataforma podrás especificar opciones distintas para cada una.



Como puedes ver, arriba del todo puedes indicar el nombre de la compañía (puedes inventar uno), el nombre del juego, el icono, incluso puedes cambiar el típico cursor blanco de Windows por cualquier otro.

Las secciones “Icon”, “Resolution and Presentation” y “Splash Image” son user-friendly y fáciles de entender, así que juega con cada una de ellas para personalizar tu build. **Pero no te adentres sin saber en la tierra de las “Other settings”**. Son opciones más avanzadas y podrías romper algo sin saberlo.

Secciones de las Player Settings

Icon: te permite personalizar el icono mucho más allá de la personalización que ofrece el “Default Icon” de arriba del todo (diferentes resoluciones y formas, etc).
Resolution and Presentation: detalles de la ventana de Windows o de la orientación del teléfono en caso de Android. Después modificaremos estos valores.
Splash Image: las splash images son los logos de las compañías que aparecen al principio de los juegos. Aquí puedes personalizar el tuyo propio.
Other settings: cajón de sastre para el resto de opciones avanzadas y más raras.
XR Settings: XR (Extended Reality), engloba las opciones para utilizar realidad aumentada y realidad virtual en el proyecto.

Player settings específicas de Windows

Resolution and Presentation

Vamos a configurar varias cosas específicas para la build de Windows:

- No permitiremos Fullscreen, se jugará en modo ventana.
- La ventana no será redimensionable.
- Y tendrá las mismas dimensiones que un teléfono móvil (360x640 para que no ocupe demasiado).

De esta forma **nos ahorramos adaptar la interfaz y el gameplay de nuestro juego a resoluciones más grandes**.

Estas opciones se editan en la sección **Resolution and Presentation**. Configura las opciones como las de la imagen:

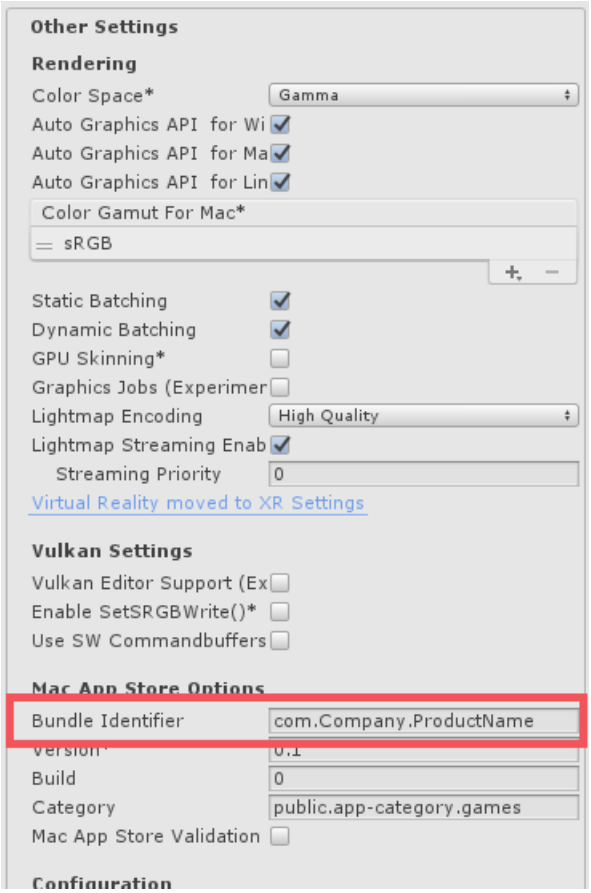
Posible bug: la resolución no funciona

¿Hiciste una Build antes con la resolución sin configurar, verdad? Al ejecutar el juego, se guardaron en tu sistema operativo las preferencias de resolución. **No te preocupes, no le pasará a otra gente que pruebe tu juego**. Aún así, si quieres disfrutarlo tú también, tienes 2 opciones:

- Eliminar del registro de Windows las preferencias del juego bajo el registro `HKEY_CURRENT_USER > Software > [Publisher Name] > [App Name] >`. Podrás modificarlo con el programa "regedit", aunque eso te lo dejo a ti.
- Cambiar el Bundle identifier que vas a setear en el siguiente apartado a otro distinto del anterior, para que no reconozca los valores que guardaste anteriormente.

Other settings > Bundle identifier

El Bundle Identifier es un código que identifica la aplicación, como un nombre de usuario. Lo encontrarás bajo la sección “Other settings”.

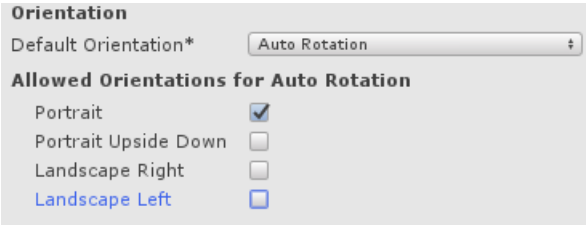


Tiene la forma de `com.Company.ProductName`. Siguiendo esa misma nomenclatura, cambia “Company” por el nombre de tu compañía (que te puedes inventar), y “ProductName” por el nombre de tu juego. **No se admiten espacios**.

Player settings específicas de Android

Resolution and Presentation

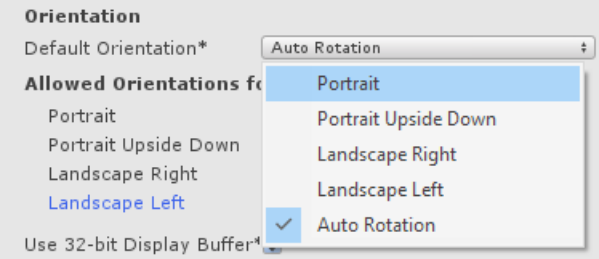
En la sección “Resolution and Presentation”, primero deshabilita todas las orientaciones menos la vertical normal, que se llama “Portrait”:



- Si no te aparecen las 4 opciones asegúrate que en “Default Orientation” ahora mismo esté en “Auto

Rotation". Después modificaremos esta opción, pero para desactivar las orientaciones permitidas tenemos que tenerla seleccionada ahora mismo.

Y ahora sí despliega "Default Orientation" y cámbiala a "Portrait":



Con esto evitarás que el jugador pueda cambiar la orientación del móvil mientras esté dentro de nuestra aplicación. Otra forma más de ahorrarnos adaptar el juego a diferentes resolucione.

Other settings > Bundle identifier

Exactamente igual que en Windows, y funciona de la misma manera, pero con una peculiaridad: **¡En Android es todavía más importante!** Si no cambias el Bundle Identifier no te permitirá hacer la Build, te dará fallo.

Al igual que en Windows, sustituye "Company" por el nombre de tu compañía y "ProductName" por el nombre de tu juego. **No se admiten espacios**.

Ahora sí: hacer la build

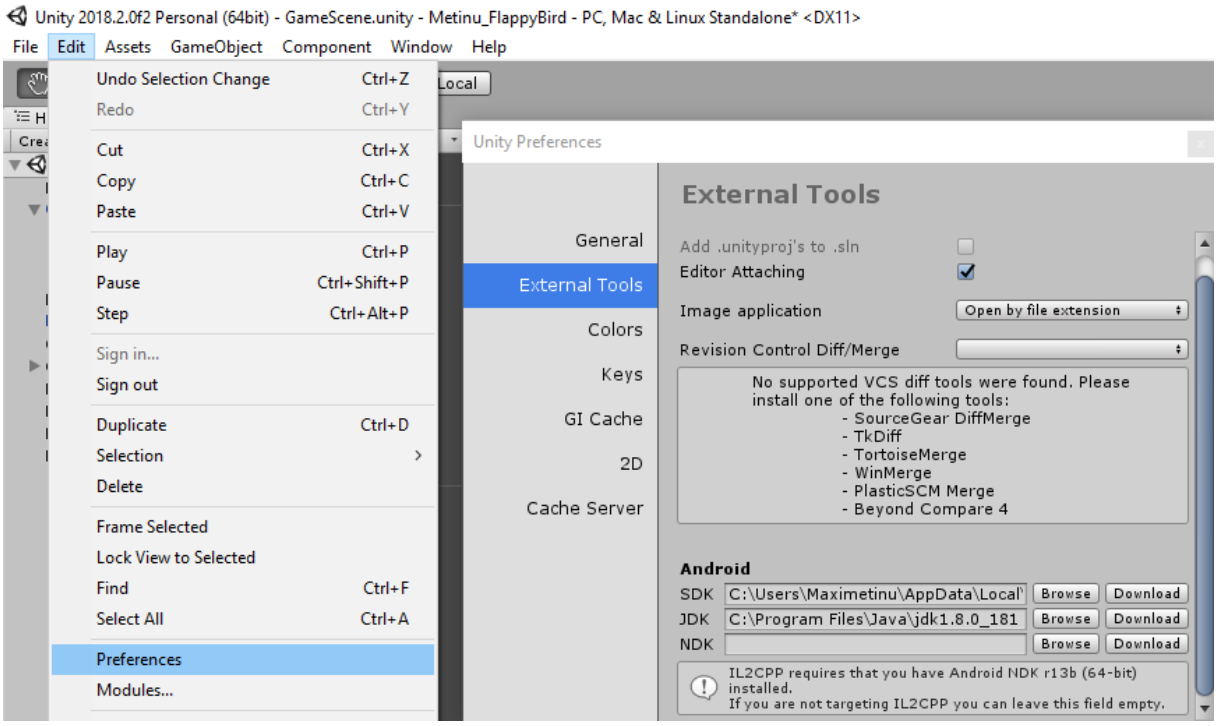
¡Ya está todo listo! Ahora sí que sí.

Vuelve a la ventana de File > Build Settings... A la izquierda, selecciona la plataforma para la que quieras hacer la build (Windows o Android) y pulsa en "**Build**". Te preguntará que dónde quieres guardarla. Hazlo donde quieras, aunque lo ideal es que la de Windows esté en una carpeta ella sola, ya que son varios archivos que no quieres tener desperdigados por el escritorio. Lo más recomendable es tener en la carpeta raíz del proyecto una subcarpeta llamada "Builds" donde guardes todas las build que vayas haciendo a lo largo del desarrollo.

F.A.Q. y posibles bugs

Posible bug con Android: no encuentra el SDK o el JDK

¿Solución? Asegúrate de que los tienes instalados y bien configurados en la ventana de preferencias de Unity, como explicamos en secciones anteriores.



Posible bug con Android: Bundle identifier no válido

Asegúrate de que has cambiado el Bundle identifier de Android a uno diferente al que te dan por defecto: `com.Company.ProductName`.

¡No me deja instalar la aplicación en el móvil!

Asegúrate de que tienes habilitada la instalación de aplicaciones de orígenes desconocidos. Aquí un tutorial de cómo hacerlo.

Otros posibles bugs que impiden que la build se haga:

Si tu proyecto tiene cualquier error de programación (es decir, si la consola muestra cualquier cosa en rojo), entonces no podrás hacer la build. Tendrás que solucionar los problemas que te da la consola primero.

Publicación

Esta parte la dejo a elección de cada uno. Tenéis infinitas opciones:

- Pagar la licencia de desarrollador de Android (25€) y subirla a Google Play: aquí te dejo un tutorial
- Subir la *.apk* o el *.exe* a Drive y compartir el archivo por url para que se lo descarguen tus colegas.
- Publicarlo en la plataforma de videojuegos indie itch.io.
- Publicar builds y el proyecto en sí en tu cuenta de GitHub, a modo de portfolio.
- Bonus level: hacer una build para WebGL y lo subo a esta misma web, para que sea jugable desde el navegador (hará falta adaptarlo a resolución ancha).

Postmortem

Un *postmortem* es un escrito que redactan los desarrolladores indie después de finalizar el desarrollo y la publicación de un videojuego, reflexionando sobre todo el camino recorrido, todo lo aprendido, qué ha salido bien y qué mal, etc. (Aquí tienes un ejemplo de uno real, de los creadores de The Fall Of Lazarus).

Pues adelante compañero. Es tu momento de escribir tu propio *postmortem*.

Keep learning

El camino no acaba aquí. Flappy Bird es el ejemplo más sencillo de jugabilidad efectiva pero sencilla de programar. Es un videojuego factible para empezar, pero queda mucho por aprender a hacer en Unity. Mucho de programación, sí, pero también saber hacer animaciones, cinemáticas, diálogos, editar mapas y crear niveles en 3D y en 2D por tilemaps, combates, disparos, utilizar assets externos que nos solucionan la vida y ahorran infinitas horas de programación, etc.

Lo bueno de Unity es que es un motor gráfico con una comunidad online enorme, y cualquier cosa que busques en Google encontrarás como hacerla. Y si buscas en inglés mucho mejor:

- projectile unity
- save highscore unity
- differences between collider and trigger unity
- jump unity
- touch input unity
- dialog unity
- first person shooter unity
- pixel art unity
- online game unity
- mmorpg unity
- etc

Prueba a buscar en Google por cualquiera de estos términos y encontrarás información. Además, también puedes escribir alguna frase característica de los errores que te da la consola si no los entiendes, y seguro que encontrarás la respuesta, ya sea en los foros oficiales de Unity o en su documentación.

Para terminar te dejo 3 links geniales para seguir aprendiendo. Si tienes paciencia y ganas de hacer videojuegos, aprenderás un montón.

- Canal Brackeys
  - Mover personaje en 2D
  - Animar personaje en 2D
  - Playlist: hacer juego de plataformas en 2D
  - 2D rigging animations
- Tutoriales oficiales de Unity
  - Bola que rueda 3D
  - Space shooter 2.5D
  - Survival shooter topdown 2.5D
- Canal Hagamos videojuegos (tutoriales en español, tienen un montón de cursos y playlists)

Happy coding! :)