Tutorial Unity: El paseo del astronauta

Género: Juego de plataformas en 2.5D

Material disponible en http://gaia.fdi.ucm.es/files/people/guille/tallerUnity2015/

GitHub: https://github.com/GuilleUCM/TallerUnity

Autor: Guillermo Jiménez Díaz (gjimenez@fdi.ucm.es)

¿Qué es Unity?

En el proceso de creación de un videojuego se ven involucradas muchas personas con distintos roles. Por un lado, visualmente tenemos el **arte**. Objetos en 2 o 3 dimensiones que le dan una estética determinada a nuestro juego, y hacen que se distinga de los demás. Dentro del arte podemos distinguir desde los propios protagonistas del videojuego, hasta el mismo icono que determina la "vida" que nos queda. Todo cuenta a la hora de definir el estilo gráfico de un videojuego. Dentro del arte también incluimos el **sonido**. Desde las melodías más pegadizas (¿quién no conoce la del *Super Mario Bros*?) hasta el sonido de pulsar un botón del menú. La atmósfera sonora es algo que consigue involucrarnos en el juego sin que nos demos cuenta, y por eso es muy importante cuidarla.

Por otro lado tenemos el **diseño**. Es una parte esencial ya que consiste en explicar de qué va el juego: cuál es el objetivo del juego, cuántos niveles va a tener, qué elementos hay en cada nivel, cuáles serán las mecánicas del juego, qué tipos de enemigos hay en el juego, qué es lo que van a hacer...

Y por último hay que destacar la **programación**. Es posible tener un videojuego sin sonido, incluso con un arte muy básico a base de cuadrados (como el *Tetris*) pero es imposible tener un videojuego sin nada que esté "programado". La programación es una rama muy extensa y compleja de la informática y es la responsable de hacer que las acciones que se han hecho en el diseño se conviertan en realidad. De momento nos bastará con saber que gracias al código somos capaces de realizar acciones dentro de un juego. Para entender lo complejo que puede llegar a ser programar un videojuego, se puede empezar describiendo un ejemplo sencillo, ¿qué tiene que pasar para que mi personaje pueda saltar?:

1. El jugador pulsa el botón A (Saltar)
2. El juego recibe la pulsación de un botón
3. Una vez comprobado que es el botón de saltar, el juego avisa al personaje: ¡Tienes que saltar!
4. El personaje recibe la orden
5. Comienza a saltar. Mediante fórmulas físicas se determina cuánto tiene que subir, cuánto tiempo debe de estar en el aire, y cuando empieza a bajar.
6. Mientras sube o baja el personaje, hay que comprobar que no se choque con nada.
7. Una vez el personaje ha vuelto a tocar el suelo se deja de realizar la acción.

Visto de esta manera, algo tan sencillo como un salto, el cual podemos realizar miles de veces en el mismo videojuego, se complica bastante.

El **verdadero reto** de realizar un videojuego llega cuando hay que juntar todos los recursos: **gráficos + sonido + diseño + programación**, y hacer que de esa mezcla salga un juego divertido. Para eso existen los motores de juego.

Un motor de juego es una herramienta que proporciona facilidades al usuario a la hora de crear sus propios juegos. Facilita la integración de gráficos, así como su uso y modificación. Hace sencillo el hecho de introducir sonidos en el juego, y por supuesto, simplifica en cierta manera las laboriosas tareas de programación. Y precisamente eso es

Unity. Una herramienta capaz de integrar ***arte*** + ***sonido*** + ***diseño*** + ***programación*** bajo una misma interfaz, y hacer que podamos crear el **diseño** de juegos sencillos en cuestión de días, e incluso horas.

Unity proporciona entre otras cosas un motor gráfico, un editor de escenas, motor de físicas, motor de sonido, gestión de input entrada, scripting, etc. Además puede ser ampliado con plugins y librerías.

Otra gran ventaja de Unity3D es que es una herramienta multiplataforma de modo que puedes desarrollar en PC, Mac, Linux, Web, iOS, Android, Windows Phone e incluso para consola (Wii y WiiU, Xbox360, XboxOne, PS3 y PS4, entre otras).

A continuación iremos conociendo *Unity poco a poco*, y haremos unos ejercicios sencillos para aprender a utilizarlo.

¿Cómo es Unity?

Para comenzar, el primer paso es abrir la aplicación. Una vez haya terminado de cargarse, debería mostrarse la interfaz principal del programa. Antes de comenzar conviene que todos tengamos en pantalla algo similar a lo que se muestra en la siguiente captura, para ello, en la barra de herramientas superior deberemos elegir *Window – Layouts – 2 by 3*.

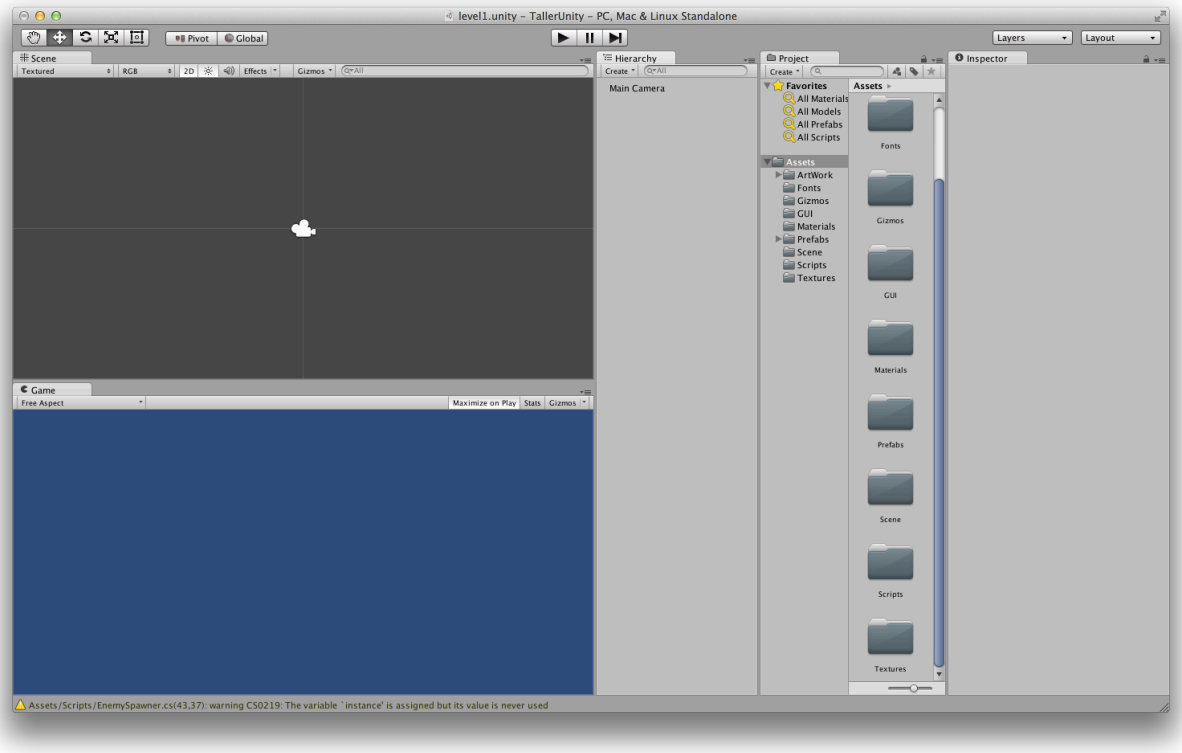


Figura 1. Interfaz básica de Unity

- **Barra de Menús**: Contiene los menús con las principales acciones que se pueden hacer en Unity. Iremos viendo algunas de estas acciones a medida que las vayamos necesitando.
- **Barra de herramientas**: En la barra superior podemos encontrar unos cuantos botones con algunas de las acciones que más usaremos dentro del motor. De momento, lo más interesante de esa barra son los cuatro botones de la izquierda , los cuales nos permitirán, una vez tengamos seleccionado un objeto, moverlo, rotarlo o escalarlo.

Figura 2. Controles de objeto

- **Ventana del Editor (Scene)**: En ella podremos colocar nuestros objetos como queramos y configurar sus características. **Dentro de esta ventana es donde vamos a hacer el juego**.

  Para poder cambiar la perspectiva se puede hacer click en los diferentes conos dibujados en el eje de coordenadas de arriba a la derecha:

Figura 3. Eje de coordenadas de la ventana del editor

  Para mover la vista tenemos las siguientes combinaciones:

  - Botón derecho: Rotar el escenario.
  - Botón izquierdo: Seleccionar objetos.
  - Alt + Botón izquierdo del ratón nos permite rotar la vista (movimiento llamado *ORBIT*).
  - Alt + Botón central del ratón nos permite desplazar la vista (movimiento llamado *PAN*).
  - Alt + Botón derecho o Rueda del ratón: Nos permite realizar *ZOOM*.

  También podemos mover la vista manteniendo pulsado el botón derecho del ratón y usando las teclas WASD + QE.

  Es bastante común en las escenas perder de vista algún objeto importante y, aunque en principio esto no es un problema, en escenas grandes con muchos objetos esto puede ser caótico. Si tenemos un objeto seleccionado en la escena podemos centrar la vista en el rápidamente pulsando "F" o haciendo doble click sobre él en la Jerarquía de la escena (que veremos más adelante).

- **Ventana de juego (Game)**: Sencillamente es la ventana donde vamos a ver nuestro juego en cuanto pulsemos el botón Play de la barra de herramientas.

Figura 4. Botón Play

  También podemos pararlo usando el mismo panel (o "**Ctrl + P**") además de pausar (o "**Ctrl + Shift + P**") y ejecutar paso a paso.

- **Jerarquía de escena (Hierarchy)**: En Unity los juegos se dividen en escenas. La forma más sencilla de entender qué es una escena es pensar en ellas como "niveles" de un mismo juego. Los objetos que hay en cada aparecen ordenados en una **jerarquía**, que es un conjunto de elementos que tienen relaciones tipo padre-hijo en la que el elemento padre es el contenedor de elementos hijo. Generalmente la jerarquía se inicia con un único objeto "Main Camera" ya que es lo único que se incluye por defecto en cualquier escena. En la jerarquía de escena se nos muestra una lista de todos los elementos que hay actualmente cargados en el editor. Esta lista se refresca dinámicamente durante el juego por lo que si creamos o destruimos objetos durante la ejecución del juego se verán aquí reflejados de inmediato.

- **Proyecto (Project)**: Nuestro juego en realidad está formado por muchos ficheros diferentes que se encuentran guardados en alguna carpeta de nuestro ordenador (audio, texturas, escenas, scripts, prefabs…). En este apartado podemos ver qué ficheros tiene nuestro proyecto, y en qué carpeta se encuentran guardados. Si queremos utilizar algo en nuestro juego, lo primero de lo que tenemos que asegurarnos es de que esté en alguna de las carpetas que podemos ver en esta ventana.

- **Inspector**: Esta ventana nos mostrará las características de cualquier objeto que tengamos seleccionado. No es lo mismo seleccionar una carpeta, una imagen, o al protagonista de nuestro juego. El inspector será quién nos permita distinguir entre los diferentes objetos y cambiar sus atributos.

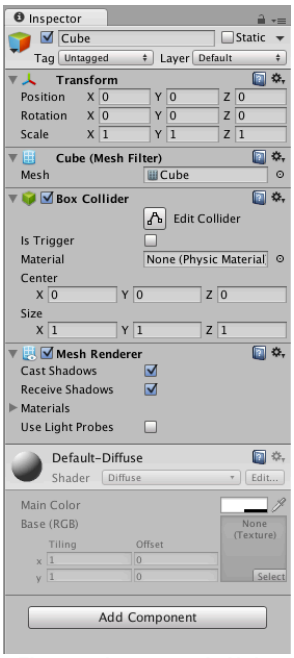


Figura 5. Inspector de Unity

- **Consola (console)**. En esta consola se registran todos los eventos y logs que se produzcan en el juego o el editor, incluidos los errores y los avisos. Se suele utilizar para lanzar mensajes de depuración de forma sencilla. (Windows >> Console (Ctrl+Shift+C)).

Conceptos básicos de Unity

Cada juego que creemos estará contenido en un **proyecto** de Unity. Cada proyecto se divide en **escenas**: la pantalla de títulos, cada uno de los niveles... Iremos construyendo distintas escenas dependiendo del número de niveles y pantallas diferentes que compongan nuestro proyecto. Cada escena se guarda en archivos independientes. Cuando cargamos una escena todos los objetos de la misma se cargan en el panel de la jerarquía.

Unity se basa en una arquitectura de componentes. ¿Qué significa realmente esto?

Cada escena se compone de **GameObjects**: La cámara, las luces, los enemigos, el jugador… El gameobject es la unidad básica dentro del motor. Es meramente un contenedor no son capaces de hacer nada por sí mismos. Se organizan en jerarquías.

Para que un GameObject tenga funcionalidad es necesario añadirle **componentes**. Cada componente le proporciona una funcionalidad distinta: física, movimiento, gráficos, daños…

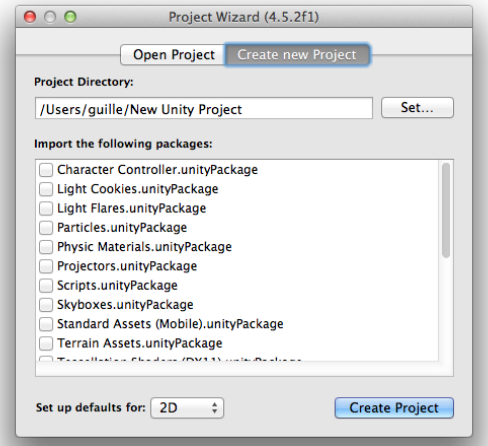
Todos los GameObject tienen un componente **Transform** que define su posición, tamaño y rotación del elemento en 3D. Este componente no puede ser eliminado.

Un GameObject tiene además la siguiente información:

- **Nombre**: Lo identifica de otros GameObject (aunque puede haber nombres repetidos).
- **Tag**: una etiqueta que lo diferencia de otros GameObject (útil para crear categorías de GameObjects)
- **Layer**: una capa a la que pertenece. Utilizado por el motor de físicas.

Preparando la escena

Vamos a comenzar de verdad con nuestro juego. Para ello creamos un proyecto de Unity: **File – New Project.** Nos aseguramos que en Scene tenemos el 2D pulsado para ver la escena completamente en 2D.



Crear un GameObject

Inicialmente vamos a crear un GameObject muy sencillo: un cubo. Para añadir un cubo a la escena bastará con desplegar la opción **GameObject – 3D Object – Cube** de la barra de herramientas.

Para poder ver los movimientos de cámara en 3D puedes pulsar sobre el botón **2D** que aparece en la ventana Scene. Podéis ir alternando las perspectivas haciendo click en las diferentes flechas del icono del eje de coordenadas. Aparte, podéis mover el punto de vista con **Click Central** y con **Alt + Click Izquierdo**

También podemos hacer pruebas para modificar este objeto. Para ello usaremos las tres herramientas básicas para modificar los GameObjects de la escena: mover (W), rotar (E) y escalar (R).

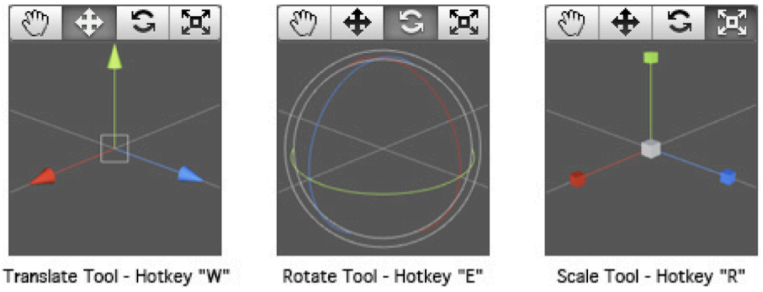


Figura 6. Herramientas de manipulación de GameObjects

Una vez que hayas terminado de probar esto puedes eliminarlo ya que no formará parte de la escena.

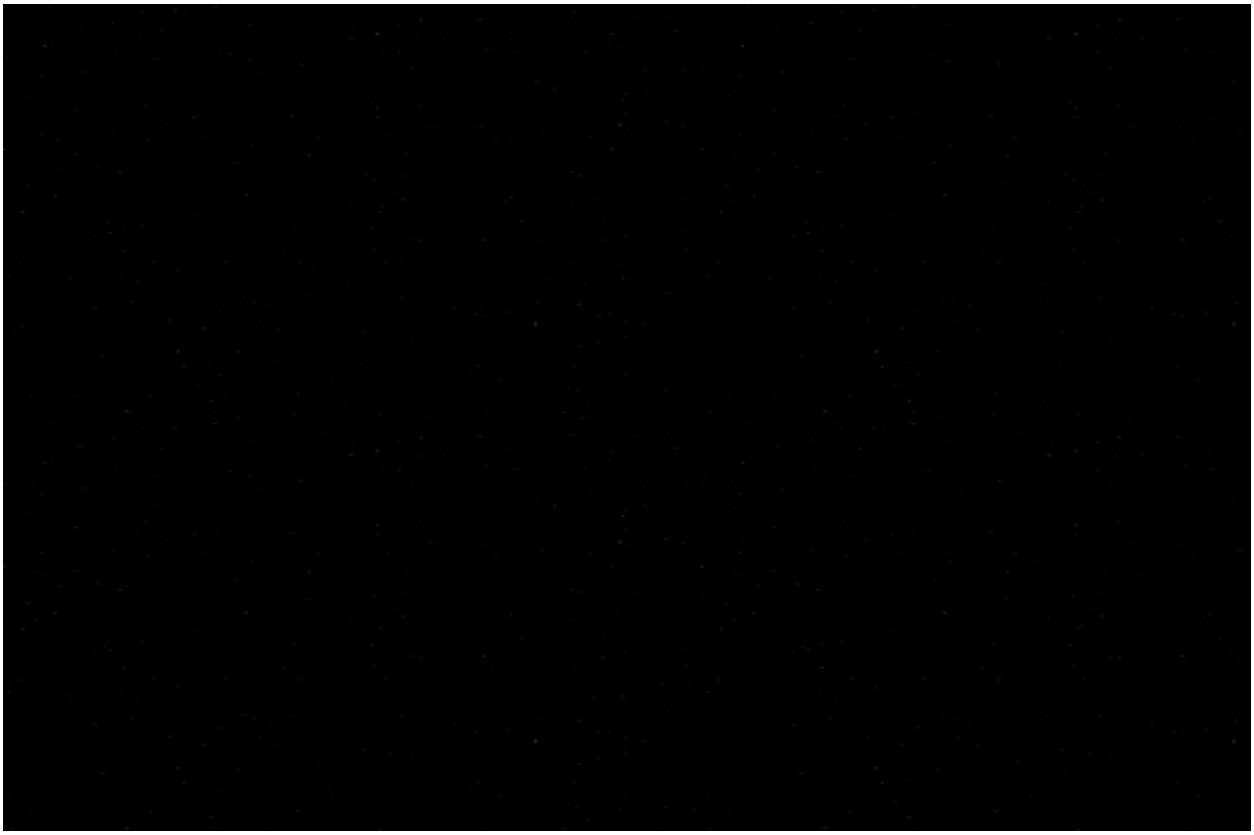
Para terminar con la preparación de la escena vamos a cargar un paquete (**packages** en Unity) que contiene todos los recursos que usaremos para hacer este tutorial. Para ello elegimos la opción del menú **Assets - Import Package - Custom Package**. En el explorador de archivos buscamos el fichero **taller-fdi-ucm-2015.unitypackage** que habremos descargado de alguno de los sitios indicados al principio de este tutorial. En la siguiente ventana pediremos que importe todo y pulsaremos el botón **Import**. Veremos que en el proyecto se han añadido un montón de carpetas. Esta suele ser una organización muy común de directorios en Unity por lo que es recomendable tenerla en cuenta.

Ya estamos preparados para comenzar con el tutorial.

Fondo de la escena

Vamos a comenzar creando el fondo de la escena: un fondo de estrellas.

1. Es conveniente poner un elemento que sea el padre de todos los elementos de la escena. Creamos una entidad vacía donde meteremos todos los elementos de la escena: **Game Object - Create Empty**. Lo llamamos **Root** y lo situamos en la posición (0,0,0) desde el inspector.
2. Arrastra el GameObject **Main Camera** para que sea hijo de Root. Coloca la cámara en el (0, 0, -10).
3. A este elemento le pondremos la etiqueta **Root**. Como seguramente no exista la tendremos que crear. En el inspector pulsa sobre la **lista desplegable Tag** y selecciona la opción **Add Tag**. En el hueco **Element 0** escribe la palabra **Root**. Si volvemos a seleccionar el GameObject Root veremos que ahora, en la lista desplegable, podemos seleccionar la etiqueta Root.
4. Creamos un plano: **GameObject – 3D Object - Plane**
5. Lo rotamos en X=-90. Esto se hace desde el Inspector, en el grupo **Transform**, donde aparecen los atributos de la entidad Position, Rotation y Scale. Veremos un plano en gris en el editor
6. Lo hacemos colgar de la cámara (arrastramos en la Jerarquía el objeto creado llamado Plane sobre la MainCamera) y lo situamos en el (0,0,100). Ahora será un fondo fijo.
7. Añadimos el material **backgroundMaterial** al plano. Para ello buscamos el material en la carpeta **Assets - Materials** y lo arrastramos sobre la entidad Plane de la jerarquía. Veremos que ahora el plano tien un nuevo componente llamado Background Material
8. Le cambiamos el nombre al plano (Background) y el tamaño para que ocupe toda la pantalla en la ventana de **Game** y ya tenemos el fondo.
9. Guardamos la escena en la carpeta **Assets – Scenes** con el nombre **level1.** Si esta carpeta no está creada entonces pulsa con el botón derecho en la carpeta Assets y selecciona la opción **Create – Folder**.



¡Perfecto! Ya tenemos una escena con un fondo algo menos soso que un espacio gris y vacío.

La luz

Uno de los aspectos que más ambientación da a los videojuegos es la iluminación. En nuestra vida diaria no nos damos cuenta, pero la iluminación está presente absolutamente en todos los rincones y lugares a los que vamos.

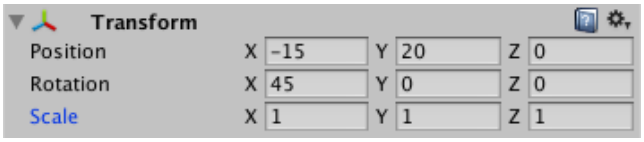
Desde el propio sol, con el día y la noche, hasta nuestra sencilla lámpara de noche, que ilumina un "círculo" encima de nuestra mesa.

En la mayoría de videojuegos se trabaja con tres luces

- **Direccional (Directional)**: Correspondiente al sol. Ilumina toda la escena de igual forma, y tiene un color determinado
- **Punto (Point):** Un punto sería como una bombilla, ilumina en todas direcciones pero con un rango limitado
- **Foco (Spotlight):** El ejemplo más directo es comparar este tipo de luz con un foco de cine. Tiene un alcance determinado, y además, ilumina únicamente en un "cono"

Para iluminar nuestra escena crearemos una luz direccional. Este tipo de luces iluminan en una dirección dada de forma uniforme. De esta forma, no importa en qué posición la pongamos.

1. Creamos la luz direccional desde el menú **GameObject – Light – Directional Light**
2. La renombramos como `Foco` y la hacemos que cuelgue de Root
3. Configuramos los valores (aconsejados) que se muestran a continuación

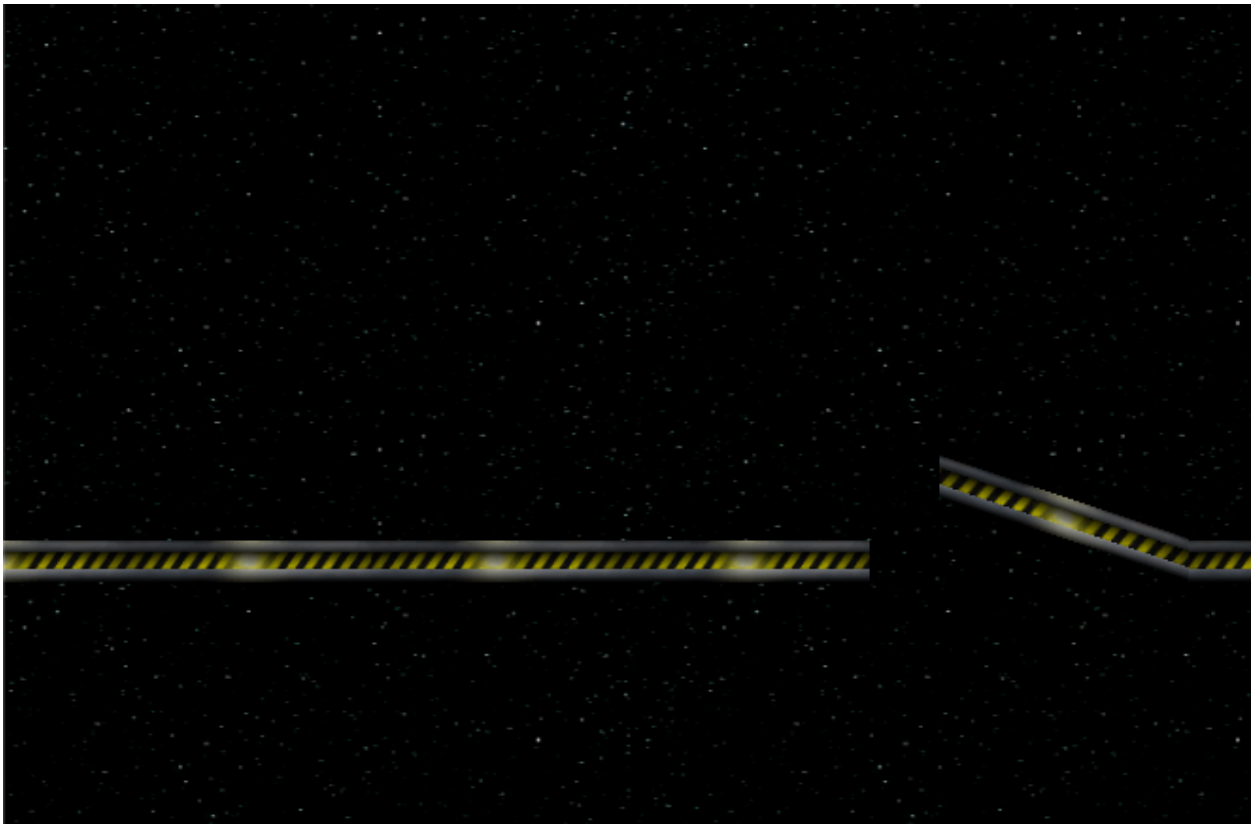


El resultado final será algo parecido a esto:

Plataformas

Vamos a empezar por lo fácil, que es poner objetos estáticos en la escena. Para ello vamos a hacer uso de otro elemento de Unity llamado **Prefab**. Un Prefab es un objeto pre-fabricado que nos permite guardar en el proyecto diferentes tipos de GameObjects configurados como nosotros queramos. Un prefab se podría definir como un "molde" a partir del cual podemos crear copias de GameObjects.

1. Vamos al Proyecto, a la carpeta Prefab - Platforms, y arrastramos el prefab **LevelPlatform** a la escena. Nos aseguramos de que la posición en Z=0.
2. Movemos la plataforma a donde nos apetezca.
3. Seguimos añadiendo otras plataformas (**LevelPlatform** y **AngledPlatform(** arrastrando los distintos prefab de plataformas que tenemos en la carpeta Prefab->Platforms. Hacemos que unas casen con otras para dar un continuo a la escena. Para ayudarnos podemos mantener pulsada la tecla "V". De esta forma se ajustan automáticamente los vértices de los elementos.
4. Puedes rotar los objetos si es necesario. También puedes duplicar los objetos que ya tengas en las escena con "Ctrl + D". Recuerda que todas las plataformas que añadas han de estar en el Z=0.



Cada una de estas plataformas es lo que se conoce como un objeto tileable, que es fundamental en la creación de videojuegos, ya que nos permite crear escenario (contenido) de forma rápida y barata, es decir, sin mucho esfuerzo. Consiste en tener pequeños objetos a modo de ladrillos, que nos permitan crear objetos más complejos dando sensación de variedad al juego. Con ellos podemos construir objetos complejos que se comportan como uno solo.

1. Creamos una entidad vacía (**GameObject - Create Empty**), y lo llamamos **Platform**. Ojo con el eje Z.
2. Metemos cada uno de los tiles o plataformas que creamos anteriormente dentro de Platform. Así podremos moverlos todos como un único elemento.

Jugador y la cámara

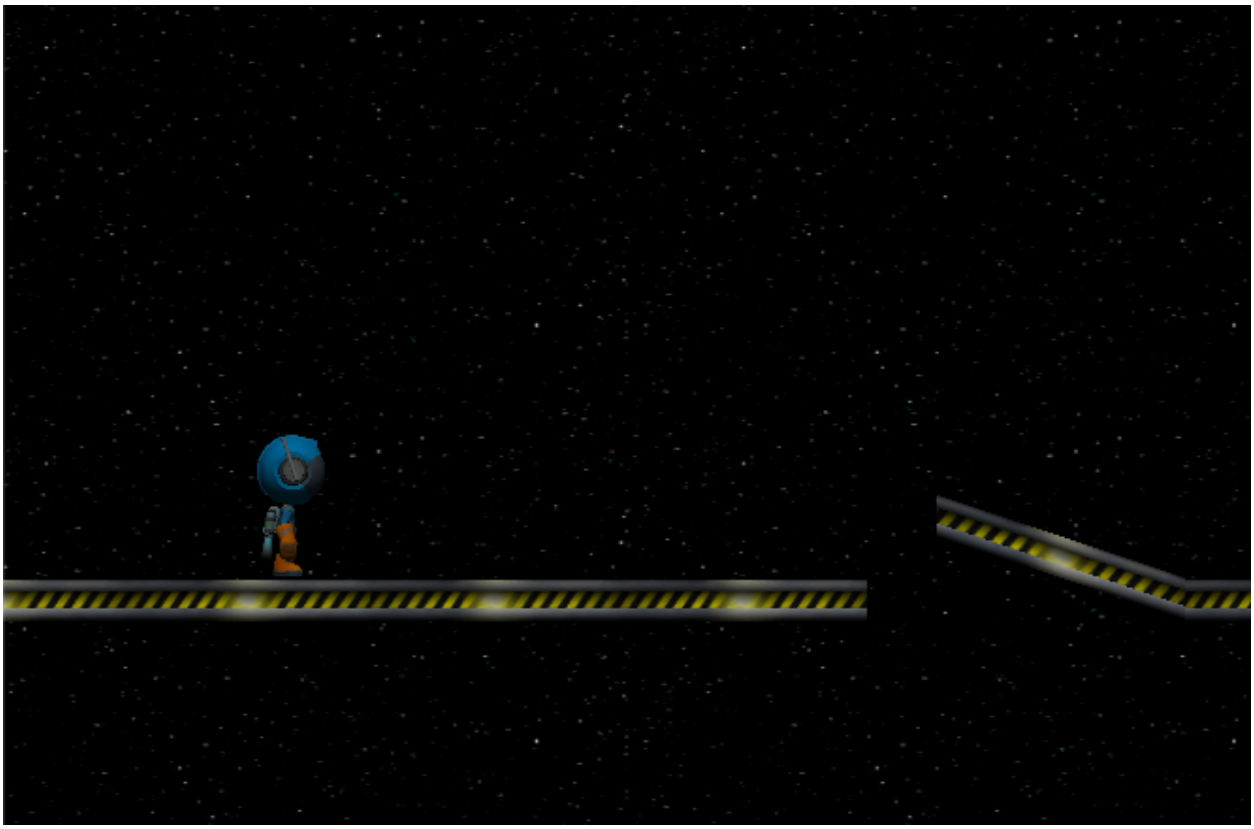
Ahora vamos a insertar al personaje que representa al jugador y vamos a hacer algunas modificaciones sobre la cámara. Si quieres puedes partir de la **release b0-jugador** que hay en el repositorio de GitHub.

Personaje del jugador

Todo juego necesita un personaje. En nuestro caso usaremos un divertido astronauta. En primer lugar tenemos que añadir al personaje en la escena, para ello:

1. Seleccionamos al **Player** de la lista de recursos disponibles de nuestro proyecto (carpeta Prefabs).
2. Lo arrastramos a la escena y lo colocamos en un sitio que nos guste (más adelante veremos que hay que tener en cuenta que esta no va a ser su posición inicial)
3. Colócalo más o menos sobre las plataformas para evitar que se caiga infinitamente.

Con esto ya tendremos nuestro astronauta, el cual podremos manejar con las teclas de flechas (la tecla espacio sirve para saltar).



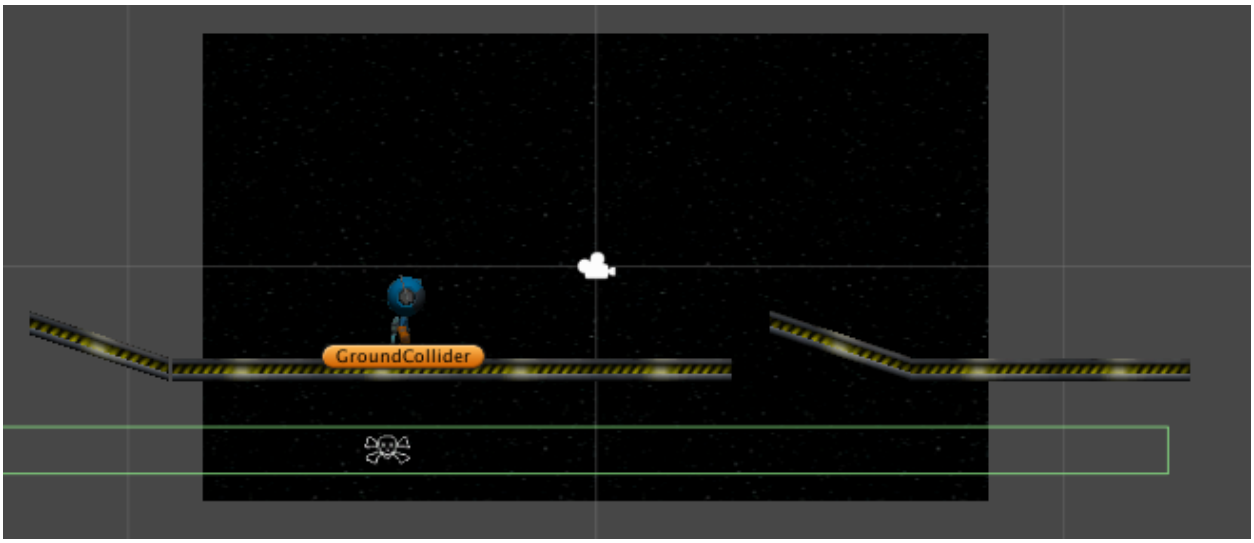
Zona de Muerte y Respawn

Ahora que ya tenemos algo que podemos usar de nivel, vamos a hablar de la parte más divertida de un desarrollador de videojuegos: matar al jugador.

Primeramente vamos a añadir una zona de muerte a la escena: una zona en la que si cae el jugador, morirá. Entre los Prefabs veréis que hay uno llamado **DeathZone**. Al seleccionarlo, podemos ver en sus opciones que cuenta con un script que avisa a quien entre de que toca morirse y un Box Collider 2D que permitirá colisionar al objeto para poder lanzar este aviso.

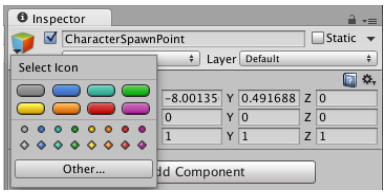
Lo podemos añadir a la escena, por debajo de la plataforma. Veremos que se dibuja como un rectángulo verde con

una calavera en medio. Podemos hacerlo tan grande como queramos (cambiar la escala X) para que ocupe toda la parte inferior de la plataforma.



Obviamente, para que esto funcione, necesitaremos hacer algo una vez que hayamos matado al jugador. Vamos a crear puntos un punto de re-spawn, un punto en el que el jugador reaparecerá cada vez que empieza el juego o muere.

1. Creamos un GameObject vacío: **GameObject - Create Empty**. Lo renombramos como **Character Spawn Point**. Cambiamos el Gizmo a un rectángulo rojo para que lo veamos claramente en la escena.



2. Posicionamos el punto de respawn en la escena, típicamente al principio y un poco por encima de las plataformas.
3. Para hacer que la entidad Player sepa dónde tiene que volver a aparecer cuando muere seleccionamos la entidad **Player** y arrastramos la entidad **Character Spawn Point** desde la jerarquía hasta el hueco llamado **Spawn Point** que hay en el componente Respawn On Death que aparece en el Inspector.

Ya podemos mover a nuestro personaje por la escena y, además, puede morir. Haz la prueba: si te caes de la plataforma y chocas con el **Death Trigger** verás que el Player vuelve al punto de Respawn.

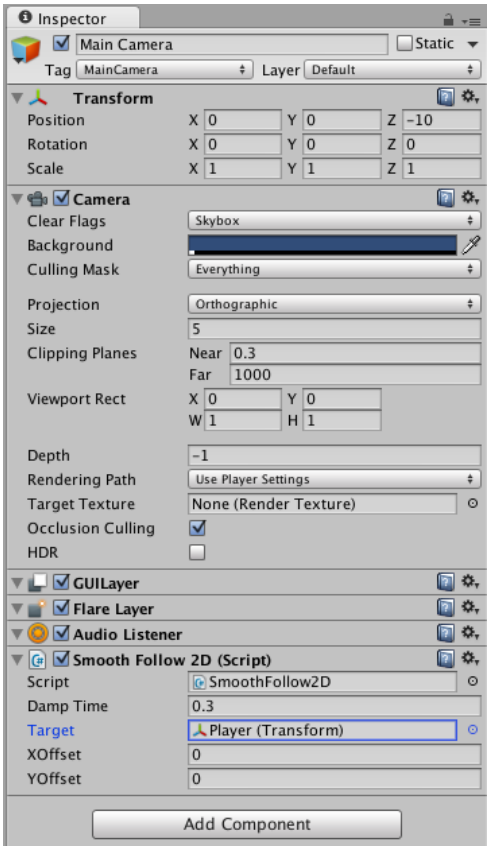
La cámara

Uno de los aspectos que a la gente le suele pasar más desapercibido en un videojuego es la cámara con la que se "graba" la escena que ve el jugador. Para poder ver cualquier cosa, hace falta que el juego tenga un punto de vista desde el que mirar. Dicho de forma un poco más elegante, en todo juego hace falta tener una cámara.

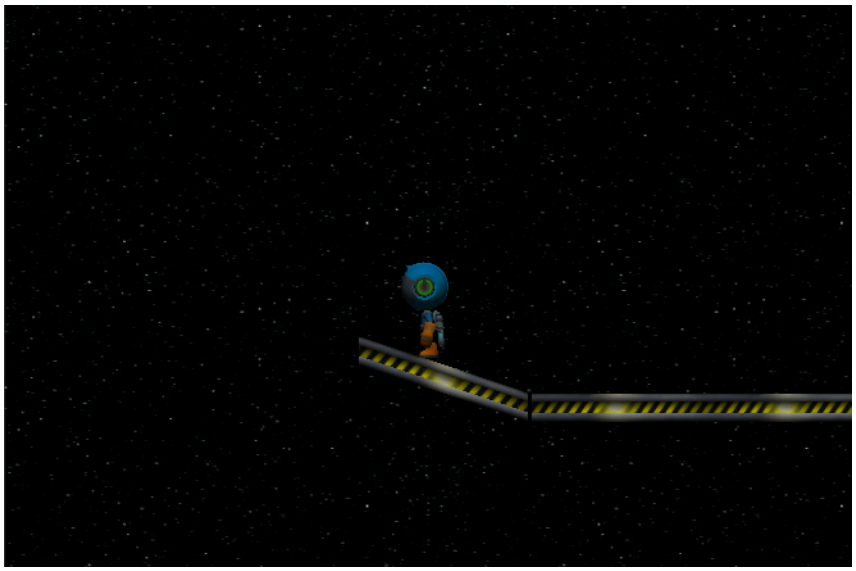
Las cámaras pueden ser *en primera persona*, como la del Call of Duty, que sólo permite ver las manos del jugador; *en tercera persona*, como es el ejemplo del Super Mario 64, donde la cámara está en la espalda del jugador, y le puede ver de cuerpo entero o por ejemplo. En nuestro caso, tendremos una cámara en vista lateral en la cual se enfoca al jugador de perfil y nunca cambia su rotación, únicamente su dirección.

Para poder añadir una cámara a nuestra escena, y ver a nuestro personaje debemos seguir los siguientes pasos:

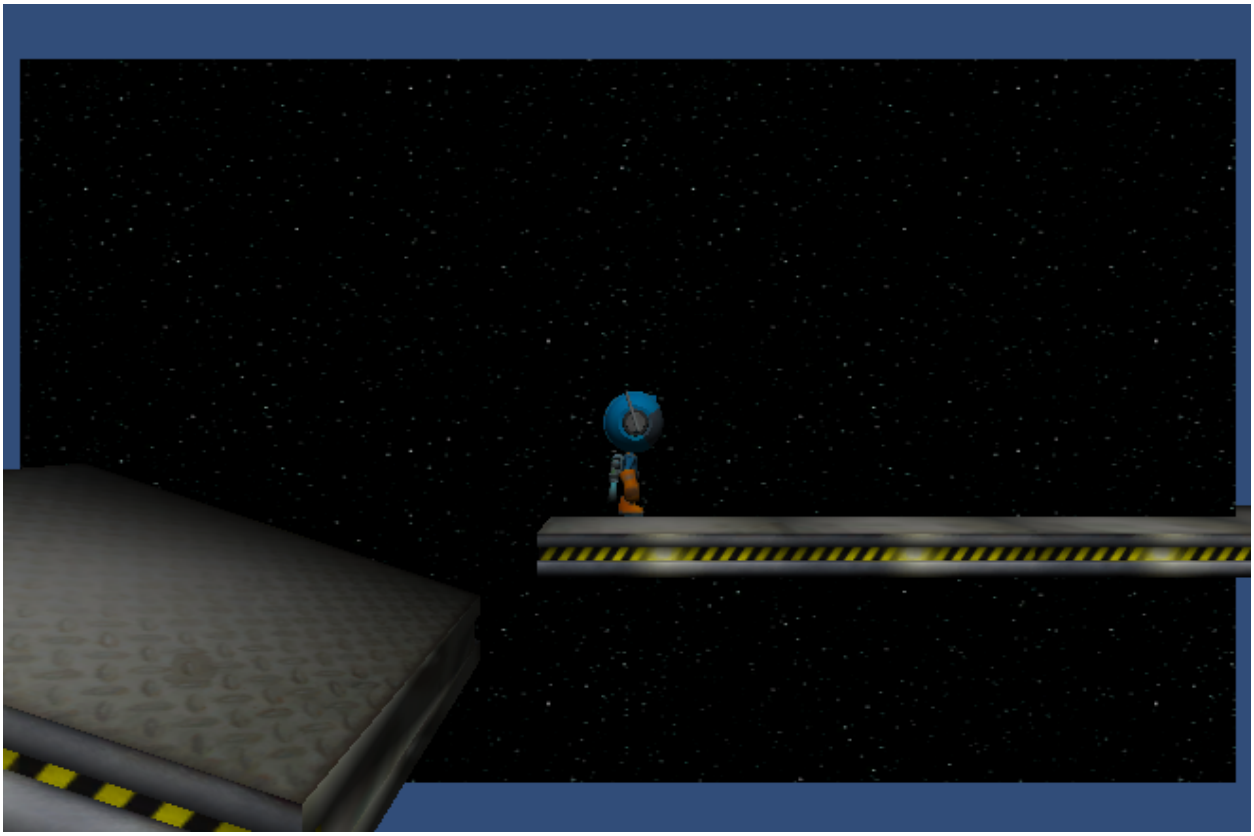
1. Busca en el **Project** la carpeta **Assets - Scripts**. Arrastra el componente **SmoothFollow2D** a la entidad de la cámara (llamado **Main Camera**)
2. Este Script hace que la cámara siga por defecto al GameObject cuya etiqueta sea Player. Sin embargo, para asegurarnos podemos seleccionar en la Propiedad Target la entidad llamada Player. Juega con los otros parámetros que tiene el componente para modificar el comportamiento de la cámara.
   - **Damp Time** es el tiempo que tarda la cámara en posicionarse enfocando al jugador.
   - **XOffset** e **YOffset** sirven para que la cámara no coloque al jugador en el centro exacto de la escena sino que lo sitúe a un lado.



Ahora veremos cómo la cámara sigue en todo momento al jugador.



El tipo de cámara que hay actualmente es lo que se conoce como cámara *Ortográfica* y es la que se usa en juegos 2D ya que no proporciona profundidad a la escena. Podemos hacer la prueba y cambiarla a cámara en *Perspectiva* desde el Inspector con la entidad **MainCamera** seleccionada, de modo que sí podamos ver la profundidad y así conseguir el efecto 2.5D que queremos (juego en 2D pero con profundidad). Si no has colocado las plataformas o al jugador teniendo cuidado que Z=0 con la cámara en perspectiva verás los fallos



Coloca las plataformas que no estuviesen bien colocadas (así como el jugador) y modifica el fondo para que se vea correctamente.

Objetos inanimados

Vamos a incluir algunos objetos inanimados en la escena. Algunos los podremos mover, otros quedarán fijos y a algunos los podremos atravesar. Para conseguir todos estos comportamientos físicos necesitamos saber algo más sobre la física de Unity y los cuerpos rígidos. Si quieres puedes partir de la **release b1-objetos** que hay en el repositorio de GitHub.

Cuerpos Rígidos

Un cuerpo rígido o **RigidBody**, define una característica física de un objeto para que éste se comporte dentro de la física del juego como un cuerpo sólido. Ejemplos de cuerpos sólidos pueden ser las estructuras, cajas, barriles, etc. Su comportamiento es el siguiente:

- Los cuerpos rígidos son atraídos por la fuerza de la gravedad.
- Los cuerpos rígidos chocan con otros cuerpos rígidos. Al ser rígidos no se pueden traspasar y si se les aplica una fuerza mayor a la fuerza de la gravedad y su masa, estos podrán ser desplazados.

Unity incluye dos tipos de física: la física 2D y la física 3D. Los componentes que se usan en física 2D tienen el mismo nombre que los de 3D (RigidBody, BoxCollider...) pero añaden el sufijo “2D” al elemento. Es importante que sepamos que no podemos hacer que interactúen elementos de física en 2D con elementos de física en 3D por lo que una vez se decida por un tipo de física tenemos que ser consecuentes con el resto. Como estamos haciendo un juego en 2.5D vamos a aplicar la física 2D, de manera que se ignore el eje Z en los movimientos de los cuerpos rígidos.

Cajas y obstáculos

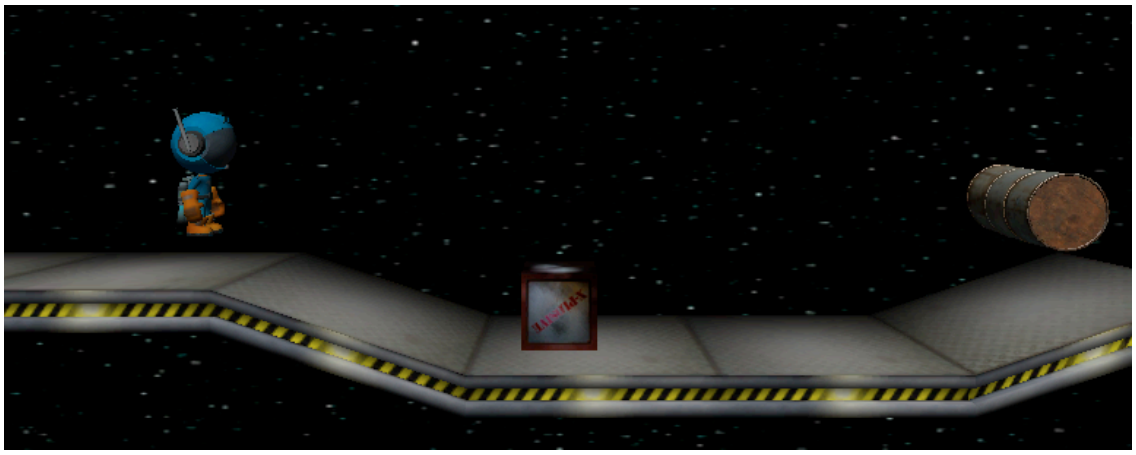
En el proyecto de ejemplo disponemos de una caja previamente construida y almacenada en un Prefab. La podemos encontrar en el panel Project, en la carpeta **Prefabs**, con el nombre de **Crate**.

Crate tiene los siguientes componentes:

- **BoxCollider2D**: Una caja de colisión que permite la interacción física con el resto del mundo.
- **RigidBody2D**: Permite dotar a un objeto de las propiedades de cuerpos rígidos descritas anteriormente. Está definido con los valores por defecto, pero se pueden ajustar para por ejemplo, hacerla inamovible (marcando isKinematic), para ignorar la gravedad (gravityScale) o para hacerla más pesada (con más masa).
- **MeshRenderer** para dibujar la caja usando el material llamado CrateMat.

Para poner en práctica estos conocimientos haz lo siguiente:

1. Añade a la escena un puñado de cajas, asegurándote que la posición en el eje z sea igual a 0. ¿Puedes empujarlas?
2. Cambia la masa de la caja y prueba a empujarlas de nuevo. ¿Te cuesta más moverlas?
3. Pon una marca en la propiedad IsKinematic de la caja. ¿Puedes moverla de alguna forma?¿Cómo se comporta cuando el jugador “choca” con ella?
4. Añade un barril al escenario. Lo podrás encontrar en la carpeta Prefabs. ¿Por qué no cae y por qué podemos atravesarlo? Fíjate en los compontes del barril. Realmente no tiene ni Collider ni Rigidbody. Cambia su rotación sobre el eje Y para que parezca que va a echar a rodar.



5. Vamos a añadir el Collider al barril: selecciónalo y pulsa el botón **Add Component** que aparece en el Inspector. Selecciona la opción Physics 2D – Circle Collider 2D. Verás cómo ahora el barril queda volando pero no lo puedes atravesar.
6. Ahora añade al barril el componente **RigidBody2D** (de nuevo con el botón **Add Component** que aparece en el Inspector). Ahora deberías ver cómo el barril no solo cae a la plataforma sino que, además, rueda.

Recogida de objetos

Vamos a hacer que nuestro personaje vaya recogiendo unos cristales marcianos que colocaremos en la escena. Más adelante veremos cómo conseguir puntos por estos cristales.

1. En la carpeta Prefabs busca el prefab llamado **Crystal** y añádelo a la escena. Asegúrate de que los cristales en la escena tienen la propiedad IsTrigger activada (componente Box Collider 2D). Esto hace que podamos colisionar con ella y ejecutar acciones (aunque el efecto sea que la atravesamos)
2. Ahora vamos a hacer que el cristal desaparezca al ser tocado por el jugador. Vamos a crear nuestro primer componente. Pulsa con el botón derecho en la carpeta **Scripts** y selecciona la opción **Create - C# Script**. Llámalo **PickItem**[1]. Haz doble click sobre el componente y verás que se abre el editor para los scripts. Depende de cómo esté configurado se abrirá el editor de Unity, Mono Develop, u otro editor como Visual Studio. Al abrirlo aparecerá lo siguiente:

```
using UnityEngine;
using System.Collections;

public class PickItem : MonoBehaviour {

  // Use this for initialization
  void Start () {

  }

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

  }
}
```

Un componente de Unity se caracteriza por ser una clase que hereda de MonoBehaviour. Además, tiene una serie de métodos (como el Start y el Update que aparecen en el código) que se ejecutan en un orden

[1] Si tienes algún problema creando el componente usa en su lugar el componente ScoreItem que aparece en la carpeta Script

concreto en cada vuelta del bucle de juego. Para saber más sobre los métodos y el orden en el que se ejecutan os recomendamos echar un vistazo a esta página del manual de Unity.

Vamos a cambiar el código anterior por este otro:

```
using UnityEngine;
using System.Collections;

public class PickItem : MonoBehaviour {

  public void OnTriggerEnter2D(Collider2D col)
  {
        if (col.transform.CompareTag("Player")) {
              Destroy(gameObject);
        }
  }
}
```

Este componente hace que el gameobject que lo tiene sea destruido si colisiona con el jugador (otro objeto cuyo tag sea Player). Además, detecta la colisión solo en caso de que el gameobject tenga un trigger (un collider con la propiedad *Is Trigger* activada). Una vez hecho esto vuelve a Unity y añade este nuevo componente al cristal. Verás que cuando el jugador pasa por encima de él, el cristal desaparece.

Puntuaciones

Todo juego que se precie debería tener un sistema de puntos. Vamos a hacer que el jugador vaya consiguiendo puntos a medida que coja los cristales. Para saber cuántos puntos tiene vamos a crear un HUD (Head Up Display), una pantalla sobre impresa donde irán apareciendo los puntos del jugador. Para este tutorial vamos a utilizar el sistema de GUI antiguo de Unity (conocido como Legacy). El nuevo sistema es más versátil y podéis saber más sobre él en esta página del manual de Unity.

1. Busca en la carpeta Scripts el componente **Score** y arrástralo sobre el jugador.
2. Una vez añadido indica cuál es el **skin** que vamos a usar para que se vea la puntuación en pantalla: un skin son las propiedades con las que vamos a dibujar el la puntuación sobreimpresa en pantalla (tipo, tamaño y color de fuente, colores, etc.) Selecciona la que está en la carpeta **GUI** y modifícala a tu antojo para poner distinto tipo de letra, color, etc.
3. Cambia las propiedades **X** e **Y** de **Size** del componente **Score**. El valor ha de estar entre 0 y 1 e indica el porcentaje del ancho y del alto de la pantalla que va a ocupar la puntuación.
4. Cambia las propiedades **X** e **Y** de **Offset** del componente **Score**. El valor ha de estar entre 0 y 1 e indica el porcentaje que vamos a desplazar la etiqueta en la que se muestra la puntuación con respecto a la esquina superior izquierda de la pantalla.
5. Ahora haz la prueba. Mueve el jugador a donde has ido poniendo los cristales. Verás como éstos desaparecen al tocarlos el jugador… pero la puntuación no cambia. Para ello es necesario modificar el componente PickItem que creamos anteriormente. Vuelve a abrirlo en el editor y haz las siguientes modificaciones:

```
using UnityEngine;
using System.Collections;

public class PickItem : MonoBehaviour {

  public float m_score = 100f;

  public void OnTriggerEnter2D(Collider2D col)
  {
        if (col.transform.CompareTag("Player")) {
           col.SendMessage("AddScore", m_score, SendMessageOptions.DontRequireReceiver);
           Destroy(gameObject);
```

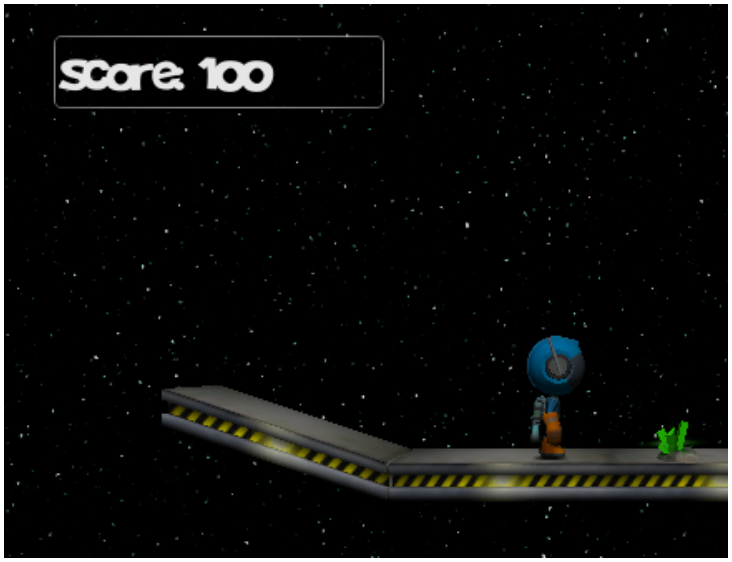
```
            }
    }
}
```

Primero hemos creado un atributo **m_score** que representa la puntuación que conseguirá el jugador al cogerlo. El atributo es público (sic) para que aparezca en el editor y pueda ser modificado por el diseñador del nivel sin tener que modificar el código.

Posteriormente, antes de destruir el cristal, enviamos un mensaje al otro objeto implicado en la colisión (en nuestro caso el jugador), indicándole que suba su puntuación. **SendMessage** es la forma que usa Unity para enviar un mensaje a uno (o varios) de los componentes del otro objeto implicado. Es una forma cómoda de comunicar eventos sin necesidad de conocer exactamente el componente que va a responder al mensaje. Sin embargo, es algo ineficiente (usa introspección) por lo que no es recomendable abusar de él.

En nuestro caso, el componente recientemente añadido al jugador (Score) es el que dispone del método addScore, que servirá para cambiar el HUD.

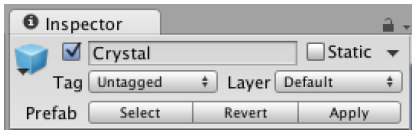
6. Ahora haz la prueba. Mueve el jugador a donde has ido poniendo los cristales. Verás como éstos desaparecen al tocarlos el jugador y cómo la puntuación aumenta.



Prefab de los cristales

Si añades un nuevo cristal a la escena verás que tendrás que volver a añadir el componente PickItem. Si queremos añadir un montón de cristales a la escena esto sería muy tedioso. Por este motivo es recomendable modificar el Prefab con los nuevos cambios realizados, de modo que ahora el Prefab de los cristales disponga ya del nuevo Componente:

1. Selecciona el cristal de la escena (el que contiene el componente PickItem). Pulsa sobre el botón **Apply** que aparece en la parte superior del Inspector.



Ya está. Con este sencillo cambio todos tus cristales (tanto el Prefab como todos los que hayas añadido a la escena) tendrán ya el componente PickItem.

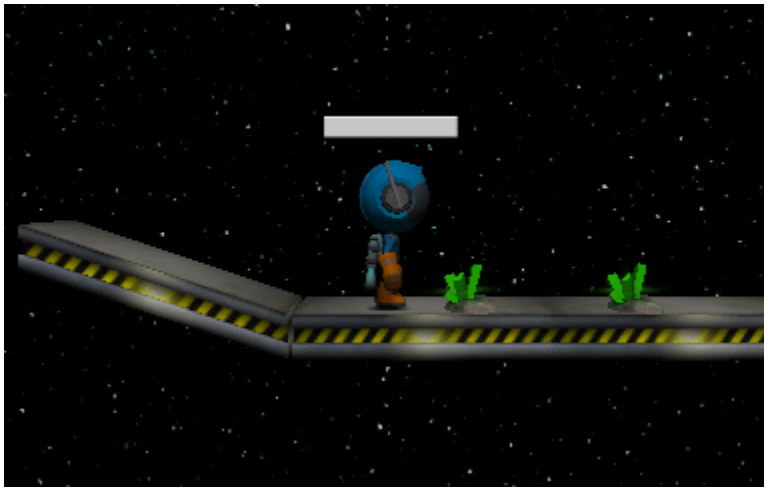
Salud y lluvia de meteoros

Ahora el jugador solo muere si se cae de la plataforma. Vamos a hacer el juego algo más interesante creando una lluvia de meteoros que haga que el jugador pierda vida y pueda morir. Si quieres puedes partir de la **release b2-salud** que hay en el repositorio de GitHub.

Barra de salud

Ahora añadiremos la salud a nuestro jugador. Vamos a añadirle una barra de vida simple.

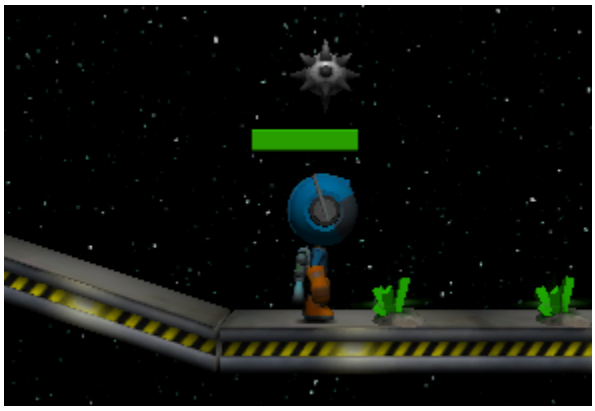
1. Busca el prefab llamado **HealthBar** en la carpeta de **Prefabs** y añádelo a la escena. Da igual dónde lo añadas ya que, si te fijas en los componentes que tiene verás que tiene un componente llamado **FollowPlayer** que sirve para colocarlo sobre el jugador. Si das a Play verás que la barra aparece sobre el jugador.
2. Usa las propiedades X e Y de Offset del componente FollowPlayer para hacer que se coloque más o menos como en la imagen.



Meteoros

Ahora añadiremos a la escena los meteoros que caerán del cielo y que dañarán al jugador cuando colisionen con él. Lo primero es que tendremos que crear un Prefab de cada uno de los objetos de tipo Meteoro que caerán del cielo:

1. Crea un gameobject vacío y llámalo **Meteor**.
2. Busca en la carpeta **ArtWork** un objeto llamado **SpikeBall** y añádelo a la escena como hijo del objeto anteriormente creado. Si das al Play verás que el objeto queda estático. ¿Recuerdas qué le falta?
3. Añade los componentes que hagan que el meteoro caiga y que pueda colisionar con el jugador: **RigidBody2D** y **CircleCollider2D**. Ahora verás que cae… pero que se queda sobre la plataforma. Si activas la opción Is Trigger entonces este problema quedará resuelto.
4. Añade el componente **DamageCollider**, que sirve para hacer daño al jugador. La propiedad **Damage** indica la cantidad de daño que hace al jugador. **Destroy on Collision** sirve para indicar si el meteoro se destruirá o no al chocar contra el jugador.
5. Coloca el meteoro más o menos sobre el punto de spawn del jugador. Verás que al dar al Play el meteoro cae sobre el jugador y se destruye (pero no causa daño alguno al jugador). Si lo pones un poco más adelante del jugador verás que los meteoros atraviesan la plataforma.
6. Para que el meteoro haga daño al jugador solo falta hacer una cosa: busca el componente **HealthBar** en la carpeta **Scripts** y añádelo al jugador. Puedes cambiar la propiedad **MaxHealth** del jugador para indicar cuál es su máximo nivel de salud. Una vez hecho esto verás que la barra de vida se reduce y cambia de color a medida que los meteoros chocan con el jugador. Si bajan por debajo de un mínimo verás que el jugador vuelve a aparecer en el punto de inicio con la barra de vida completa.



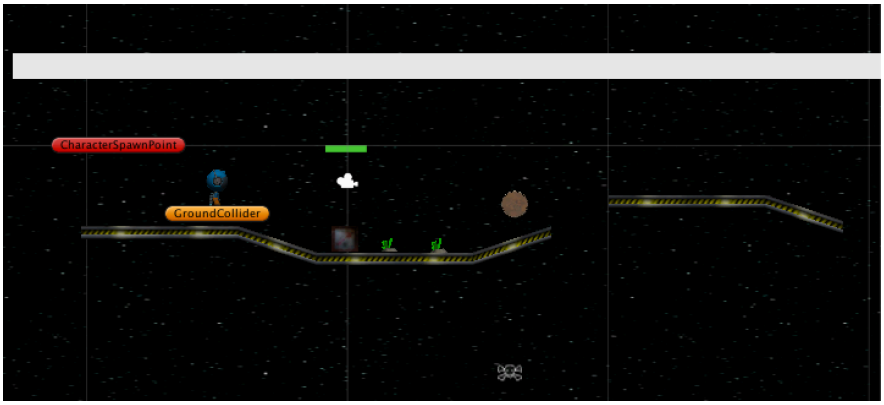
Lluvia de meteoros

Para terminar vamos a crear una lluvia de meteoros por todo el escenario. De este modo, el jugador ha de ir esquivando los meteoros para no quedarse sin vida.

1. En la escena tenemos un único meteoro. Sin embargo, vamos a querer crear muchos como él. Para ello tendremos que crear un Prefab. Ve a la carpeta Prefabs. Pulsa con el botón derecho y selecciona **Create – Prefab**. Llámalo **Meteor**.
2. Ahora arrastra desde la jerarquía el objeto Meteor sobre el Prefab recién creado. Ya está, ya hemos creado nuestro Prefab. Elimina de la escena el meteoro ya que no lo necesitaremos más.

Ahora vamos a crear el objeto responsable de provocar la lluvia de meteoros. Los objetos que crean enemigos, ítems, etc. En un videojuego se suelen conocer como Spawners.

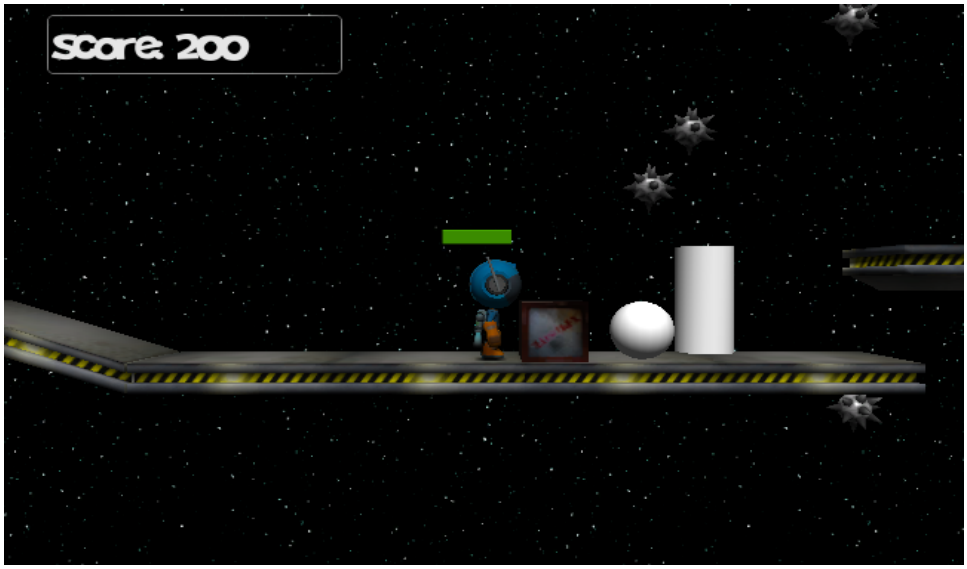
1. Crea un cubo (Menú **GameObject – 3D Object - Cube**) y colócalo por encima del escenario. Llámalo **MeteorRain**. Cambia la escala X del cubo para que ocupe casi todo el escenario ya que los meteoros irán saliendo de posiciones aleatorias de este cubo.



2. Arrastra sobre este cubo el componente **EnemySpawner** (está en la carpeta **Scripts**) y configúralo para que cree la lluvia de meteoros:
   - Arrastra sobre la propiedad **Entity** el prefab **Meteor** de la carpeta **Prefabs**. Éste será el tipo de "enemigos" que generará nuestro cubo llamado **MeteorRain**
   - Configura cada cuánto tiempo quieres que se generen nuevos enemigos. Para ello completa las propiedades **Min Spawn Time** y **Max Spawn Time** del componente (la cantidad se pone en segundos) Si pones la misma cantidad en ambos entonces los meteoros se irán generando de manera periódica. Si pones dos cantidades distintas entonces los enemigos se irán generando cada cierto tiempo aleatorio entre las dos cantidades de tiempo escritas en el componente.

3. Para finalizar haz que no se vea el cubo blanco de MeteorRain. Selecciónalo y quita el *tick* que aparece junto a la palabra **Mesh Renderer**. El cubo será invisible durante el juego (aunque seguirá produciendo el mismo efecto.

Una vez hecho esto deberías tener una lluvia de meteoros. Esquívalos como puedas antes de que te maten.



Un problema que tenemos es que nuestro juego crea muchos objetos (meteoros) que caen indefinidamente si no colisionan con el jugador. Esto haría que, si tenemos muchos objetos en la escena, el rendimiento decayese rápidamente. Para ello vamos a crear un nuevo componente que haga que los meteoros se destruyan por debajo de una determinada altura.

1. Crea un componente llamado **DestroyMeteor**[2] en la carpeta Scripts (Botón derecho – **Create – C# Scripts**).
2. Ábrelo en el editor y escribe el siguiente código:

```
using UnityEngine;
using System.Collections;

public class DestroyMeteor : MonoBehaviour {

  public float minHeight = 0.0f;

  // Update is called once per frame
  void Update () {
         if (transform.position.y <= minHeight)
                Destroy (gameObject);
  }
}
```

Ponemos un atributo público (para que lo podamos modificar desde el editor) llamado minHeight que indica la altura a la cual el meteoro se destruirá. Luego añadimos el método Update(), que es el método que se ejecutará en cada vuelta de bucle. Este método comprueba si la posición y del meteoro está por debajo de la altura indicada, en cuyo caso destruirá el objeto.

3. Selecciona el prefab Meteor, pulsa en el botón **Add Component** del inspector y busca el componente que acabas de crear para añadirlo. Ajusta la altura y el daño que hacen los meteoros. También puedes cambiar la

[2] Si tienes algún problema con la creación del componente puedes usar en su lugar el componente DestroyBelowHeight que aparece en la carpeta scripts.

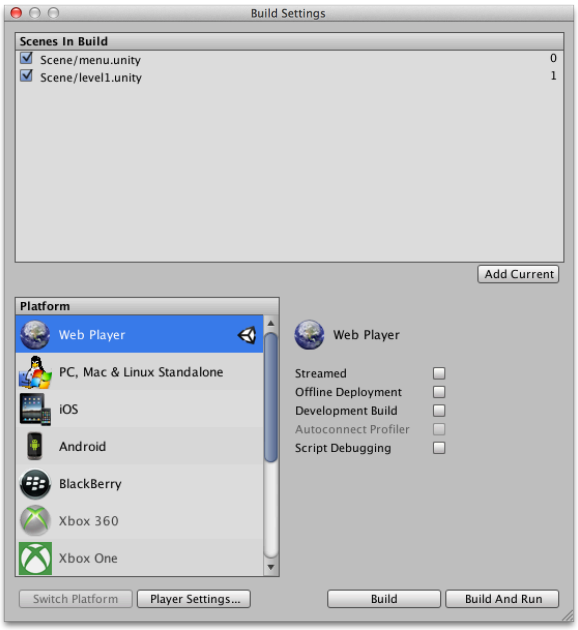
masa y la gravedad del mismo. De esta forma podrás hacer que caigan mucho más rápido (como auténticos meteoros)

Menú principal y actualización de puntuaciones

Para terminar vamos a hacer que el juego tenga un menú en el que se muestren las puntuaciones máximas logradas. Haremos que cuando el jugador muera vaya al menú principal y se actualice puntuación máxima conseguida. Si quieres puedes partir de la **release b3-menu** que hay en el repositorio de GitHub.

Como ya vimos anteriormente, cada menú, nivel... del juego se corresponde con una escena de Unity. Si vamos a la carpeta **Scene** veremos que tenemos una escena llamada **menu** que contiene el menú de la aplicación. Este menú es una escena de Unity como la que hemos estado construyendo hasta ahora y que usa el nuevo sistema de UI de Unity. Por falta de tiempo no vamos a ver cómo construirlo. Lo que sí vamos a ver es cómo hacer que al pulsar el botón de Play Game comience el juego.

1. Abre el menú **File – Build Settings**. Arrastra sobre el cuadro llamado **Scenes in Build** las dos escenas de la carpeta Scene. Asegúrate de que la primera (la escena 0) sea el menú.



2. Ahora crea un GameObject vacío y llámalo **Manager**.
3. Crea un componente llamado **ChangeScene**[3] en la carpeta Scripts. Ábrelo en el editor y escribe el siguiente código:

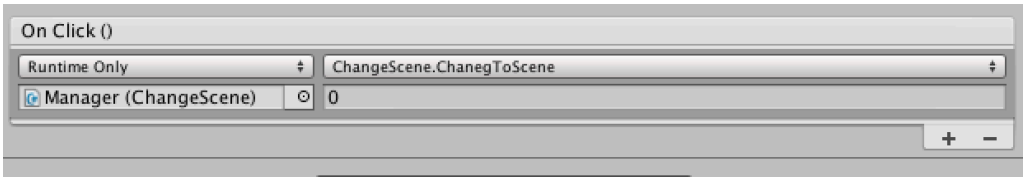
```
using UnityEngine;
using System.Collections;

public class ChangeScene : MonoBehaviour {

  public void ChangeToScene(int sceneToChangeTo) {
        Application.LoadLevel(sceneToChangeTo);
  }
}
```

La línea de código en negrita es la que nos va a permitir cambiar de escena. En este caso, le vamos a pasar un número de escena y Unity se encargará de cargar la nueva escena.

4. Añade este componente al GameObject Manager recién creado.
5. Por último, haremos que el botón ejecute este método del componente al ser pulsado. Busca el GameObject **PlayButton** que hay bajo el GameObject **Canvas**. En el inspector busca un cuadro llamado OnClick. Pulsa sobre el botón +, selecciona (o arrastra) el GameObject Manager al hueco que ha aparecido y busca en la lista desplegable **No Function** el componente **ChangeScene** y, dentro de él, el método **ChangeToScene**. Para terminar, por un **1** en el hueco que tienes para el parámetro.



6. Ahora haz la prueba. Al pulsar sobre el botón de Play Game debería cargarse el nivel.

[3] Si tienes algún problema con la creación del componente entonces usa el componente LoadLevel que aparece en la carpeta Scripts.

Para terminar vamos a guardar las puntuaciones del jugador. Unity tiene una clase en la que podemos guardar ciertas estadísticas "globales" del jugador. Estas estadísticas se guardan en un diccionario, de modo que a cada estadística se puede acceder y manipular si conocemos su nombre. El texto del menú usa una estadística llamada **HighScore** para escribir la puntuación máxima del jugador. Vamos a actualizar esta estadística cuando el jugador muera. Abre la escena **Level1** y haz lo siguiente:

1. Modifica el componente **RespawnOnDeath**. Abre el script en el editor y sustituye el antiguo método **onDeath** por el siguiente código:

```
public void OnDeath() {
        gameObject.SendMessage("SaveScore", SendMessageOptions.DontRequireReceiver);
        Application.LoadLevel(0);
}
```

   Este método va a pedir que se envíe un mensaje de **SaveScore** para que se guarde la puntuación y, en lugar de reaparecer en un punto de respawn, haremos que se cargue el menú. ¿Y dónde está el método SaveScore? Ahora mismo en ningún sitio, pero haremos que esté en el sitio donde debe estar: en el componente **Score** del jugador.

2. Busca el componente Score del jugador y ábrelo en el editor y añade el siguiente método:

```
void SaveScore() {
        int highscore = PlayerPrefs.GetInt("HighScore",0);
        if (m_score > highscore) {
                PlayerPrefs.SetInt("HighScore", m_score);

        }
}
```

Como puedes ver, primero accedemos a la estadística global HighScore y, si la puntuación que tenemos actualmente es mayor que la que teníamos, la actualizamos. Con esto ya veremos cómo nuestra puntuación se actualizará a medida que superemos nuestro record.

Fin de juego…y del taller

Con esto hemos completado el tutorial de Unity… y solo hemos visto una mínima parte del motor. La mejor forma de aprender sobre Unity es leer material y realizar alguno de los muchos tutoriales que hay en Internet. Aquí os dejamos algunas referencias que os pueden servir para iniciaros en Unity (aunque en Internet podrás encontrar muchísimas más):

- Jeremy Gibson. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. Addison-Wesley, 2014
- Sue Blackman. Beginning 3D Game Development with Unity 4: All-In-One, Multi-Platform Game Development (2nd edition). Apress, 2013
- Dave Calabrese. Unity 2D Game Development. Packt Publishing, 2014
- http://unity3d.com/learn: Sección oficial para aprender a usar Unity.
- http://www.emanueleferonato.com/category/unity3d/
- http://gamedevelopment.tutsplus.com/
- Tutoriales en Youtube:
  - Canal Unity: tiene algún tutorial pero sobre todo tiene charlas sobre Unity. https://www.youtube.com/user/Unity3D
  - Brackeys: https://www.youtube.com/user/Brackeys
  - Pushy Pixels: https://www.youtube.com/user/PushyPixels/videos
  - Hagamos Videojuegos: https://www.youtube.com/channel/UCBhkLrsmV9PVQMpT3qe-toA
- **Aprender a desarrollar videojuegos en la Facultad de Informática**
  - Vacaciones de videojuegos (11 a 17 años): https://vacacionesvideojuegos.wordpress.com/
  - Grado en Desarrollo de Videojuegos (a partir de 18 años): http://videojuegos.ucm.es/
  - Master en Desarrollo de Videojuegos (licenciados): http://www.videojuegos-ucm.es/