

# Programación de videojuegos con Unity



## **UNIDAD 1: Programación de videojuegos 2D**

**Roberto Rodríguez Ortiz**

Versión 1.2 octubre 2019

Este documento se publica bajo licencia Creative Commons “Reconocimiento-CompartirIgual (by-sa)”



## Tabla de contenidos

1.	Los videojuegos .....	5
1.1.	Historia del videojuego .....	5
1.2.	Época actual .....	5
1.3.	Clasificación de los videojuegos .....	6
2.	Programación de videojuegos .....	7
2.1.	Librerías .....	8
2.2.	Motores/engines .....	8
3.	Conceptos previos videojuegos .....	10
3.1.	Sprite .....	10
3.2.	Coordenadas .....	11
4.	Unity .....	13
4.1.	Introducción .....	13
4.2.	Instalación de Unity .....	14
4.3.	La interfaz de usuario de Unity .....	15
4.4.	Vista de escena .....	17
4.4.1.	Configuración de la visualización .....	18
4.4.2.	Botones de Control .....	19
4.5.	Vista de Proyecto .....	19
4.6.	Vista de Jerarquía .....	20
4.7.	El Inspector .....	21
4.8.	Vista de Juego .....	21
5.	Scripting .....	22
6.	Primer juego - Pong .....	25
6.1.	Creando un proyecto .....	25
6.2.	La primera escena .....	27
6.3.	Añadiendo recursos .....	27
6.4.	Añadir el comportamiento físico .....	30
6.5.	Movimiento de la raqueta .....	31
6.6.	Añadiendo la pelota .....	36

7.	Prefabs.....	40
8.	Sonido.....	41
9.	Interfaz de usuario.....	44
9.1.	Gestión de la partida.....	47
9.1.1.	Crear nueva escena.....	48
9.1.2.	Crear una UI con un botón.....	49
10.	Generando el APK .....	50
11.	Fuentes.....	52

## 1. Los videojuegos

### 1.1. Historia del videojuego

En los años 70 [Ralph Baer](#), junto a su equipo había conseguido hacer funcionar un juego de ping pong para dos jugadores. Pero, como de costumbre, el problema estaba a la hora de comercializar el proyecto. Preguntó a muchas empresas y siempre obtuvo respuestas negativas por su parte, hasta que encontró a Magnavox (donde trabajaba Bill Enders, el cual les habló del proyecto de Baer). Finalmente esta empresa sí que acepta las condiciones y empieza a trabajar para comercializarla.

En 1972 se lanza la Magnavox Odyssey al mercado, con 12 juegos distintos. Esta consola solo podía funcionar en los televisores de la misma marca.

Por otra parte **Nolan Bushnell** y Ted Dabney juntan fondos para crear una empresa. El nombre de esta empresa sería [Atari](#). Cuando Bushnell asiste a la Magnavox Profit Caravan de ese mismo año y ve la consola de esa compañía, tiene la genial idea de crear un juego similar. Contratan a Allan Alcorn y le encargan el proyecto. Tres meses más tarde ya tiene un prototipo operativo y a Bushnell y Dabney les encanta. Lo bautizan como [Pong](#).

Introducen este nuevo juego en bares y en seguida causa una gran sensación, llegándose hasta "estropear" las máquinas debido a que las cajas de las monedas estaban a rebosar. [Pong](#) sería el mayor éxito jamás visto en la industria de los videojuegos hasta ese momento. Sin embargo Bushnell tendría problemas con Baer, ya que este último le acusaba de plagio de su juego. Finalmente consiguen un trato del que Bushnell sale completamente beneficiado.

### 1.2. Época actual

Finalmente llegamos a la era moderna, la de las consolas en HD, con disco duro, con gráficos impresionantes y sobre todo con muchos juegos que siguen haciendo historia y que en un futuro serán recordados.

Empieza el año 2000 con la aparición de la consola de sobremesa más vendida de toda la historia: **PlayStation 2**, la cual tuvo serios problemas para distribuirse en sus inicios. Al cabo de un año aparece la nueva consola de Nintendo: la GameCube. A ella se le suma la XBOX, el primer sistema de Microsoft. Estas dos consolas, pese a ser buenas máquinas, nunca

podieron con el éxito arrollador de PlayStation 2 y quedaron relegadas en un segundo lugar por la lucha en el mercado, compitiendo entre sí.

La última generación de consolas se compone de la Playstation 4, la XBOX One y la WII U, junto con el renacimiento del PC, se habla incluso de la PC Master Race, dada la dificultad que tienen las consolas en alcanzar el rendimiento gráfico de los PC, y sin olvidar la irrupción de los juegos para plataformas móviles (Smartphones, Tablets,...), nos dibujan un escenario muy prometedor para la industria del videojuego. **Los videojuegos han pasado a generar más dinero que la del cine (solo entradas vendidas) y la música juntas**, como en el caso de España; la industria de videojuegos [generó](#) 57.600 millones de euros durante 2009 en todo el mundo.

### 1.3. Clasificación de los videojuegos

**Arcade:** Los videojuegos tipo arcade se caracterizan por su jugabilidad simple, repetitiva y de acción rápida. Más que un género en sí, se trata de un calificativo bajo el que se engloban todos aquellos juegos típicos de las máquinas recreativas (beat 'em up, matamarcianos, plataformas). Tuvieron su época dorada en la década de 1980.

**Deportes:** Los videojuegos de deportes son aquellos que simulan deportes del mundo real. Entre ellos encontramos golf, tenis, fútbol, hockey, juegos olímpicos, etc. La mecánica del juego es la misma que en el deporte original, aunque a veces incorpora algunos añadidos.

**Rol:** Emparentados con los de aventura, los videojuegos de rol, o RPG, se caracterizan por la interacción con el personaje, una historia profunda y una evolución del personaje a medida que la historia avanza. Para lograr la evolución generalmente se hace que el jugador se enfrasque en una aventura donde irá conociendo nuevos personajes, explorando el mundo para ir juntando armas, experiencia, aliados e incluso magia.

**Estrategia:** Se caracterizan por la necesidad de manipular a un numeroso grupo de personajes, objetos o datos, haciendo uso de la inteligencia y la planificación, para lograr los objetivos. Aunque la mayoría de estos juegos son fundamentalmente de temática bélica, los hay también de estrategia económica, empresarial o social.

Dos grandes subgéneros son los juegos de estrategia en tiempo real (también llamados *RTS*, siglas en inglés de *real-time strategy*), y los por turnos (*TBS*, siglas también en inglés de *turn based strategy*).

**Simulación:** Este género se caracteriza por recrear situaciones o actividades del mundo real, dejando al jugador tomar el control de lo que ocurre. En ocasiones la simulación pretende un alto grado de verosimilitud, lo que le otorga una componente didáctica. Los tipos de simulación más populares son los de manejo de vehículos (pilotar un coche, un avión, un tren...), los de construcción (construir una ciudad, un parque de atracciones o un imperio), o los de vida (dirigir la vida de una persona o un animal virtual).

**Aventura Gráfica:** A comienzos de los 1990, el uso creciente del ratón dio pie a los juegos de aventura de tipo «Point and click», también llamados aventura gráfica, en los que ya no se hacía necesaria la introducción de comandos. El jugador puede, por ejemplo, hacer clic con el puntero sobre una cuerda para recogerla.

**FPS:** En los videojuegos de disparos en primera persona, conocidos también como *FPS* (*first person shooter*), se maneja al protagonista desde una perspectiva subjetiva, es decir, vemos en la pantalla lo que ve nuestro personaje, y por tanto no lo vemos a él. Sí vemos en cambio su arma, en primer plano, la cual deberemos usar para abatir a los diferentes enemigos que aparecerán frente a nosotros al avanzar. La perspectiva en primera persona, en un entorno 3D, tiene por meta dar al jugador la impresión de ser el personaje, buscando con ello una experiencia más realista de juego.

**Puzzles:** Los juegos de lógica o de agilidad mental, también llamados de puzles o rompecabezas, ponen a prueba la inteligencia del jugador para la resolución de problemas, que pueden ser de índole matemático, espacial o lógico.

## 2. Programación de videojuegos.

La programación de videojuegos siempre ha implicado una gran complejidad, el acceso a los recursos hardware, audio, red y sobre todo la tarjeta gráfica, donde a menudo es necesario programar a muy bajo nivel utilizando el lenguaje c ó ensamblador, suponía una gran barrera de entrada a programadores noveles.

A finales de los 90 y principios de este milenio, solo grandes estudios podían permitirse las herramientas necesarias para desarrollar videojuegos de gran calidad, llamados triple A.

La aparición de librerías específicas y posteriormente la popularización de motores de bajo coste o gratuitos han hecho posible que más programadores se animen a programar juegos. El crecimiento del número de estudio de

desarrollo “indies” (independientes) ha sido tan rápido que ha provocado una saturación de títulos que dificulta la visibilidad de los mismos.

## 2.1. Librerías

**OpenGL:** (Open Graphics Library) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por Silicon Graphics Inc. (SGI) en 1992 y se usa ampliamente en CAD, realidad virtual, representación científica, visualización de información y simulación de vuelo. También se usa en desarrollo de videojuegos, donde compite con Direct3D en plataformas Microsoft Windows.

**DirectX:** DirectX es una colección de API desarrolladas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo, en la plataforma Microsoft Windows.

**SDL:** Simple DirectMedia Layer (SDL) es un conjunto de bibliotecas desarrolladas en el lenguaje de programación C que proporcionan funciones básicas para realizar operaciones de dibujo en dos dimensiones, gestión de efectos de sonido y música, además de carga y gestión de imágenes. Fueron desarrolladas inicialmente por Sam Lantinga, un desarrollador de videojuegos para la plataforma GNU/Linux.

## 2.2. Motores/engines

Un motor de videojuego es un término que hace referencia a una serie de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Del mismo modo existen motores de juegos que operan tanto en consolas de videojuegos como en sistemas operativos. La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, scripting, animación, inteligencia artificial, redes, streaming, administración de memoria y un escenario gráfico.

[https://es.wikipedia.org/wiki/Motor\\_de\\_videojuego#Motores\\_de\\_videojuego](https://es.wikipedia.org/wiki/Motor_de_videojuego#Motores_de_videojuego)



Existen muchos motores de videojuegos con diferentes características, los más completos incluyen editores de niveles que integran todos los aspectos del desarrollo del juego.

Recientemente motores de gran potencia, cuyas licencias costaban miles o decenas de miles de euros han lanzado versiones gratuita o con costes muy reducidos (por ejemplo una parte de los beneficios).

**Unreal Engine:** creados por la compañía Epic Games. Implementado inicialmente en el shooter en primera persona Unreal en 1998, siendo la base de juegos como Unreal Tournament, Deus Ex, Turok, Tom Clancy's Rainbow Six: Vegas, America's Army, Red Steel, Gears of War, BioShock.... Está escrito en Unreal Script (Lenguaje propio similar Java o C# modificado), siendo compatible con varias plataformas como PC (Microsoft Windows, GNU/Linux), Apple Macintosh (Mac OS, Mac OS X) y la mayoría de consolas (Dreamcast, Gamecube, Wii, Xbox, Xbox 360, PlayStation 2, PlayStation 3, Xbox One, PlayStation 4). Unreal Engine también ofrece varias herramientas adicionales de gran ayuda para diseñadores y artistas. La **última versión** de este motor es el Unreal Engine 4, está diseñado para las APIs OpenGL y DirectX.

**CryEngine:** es un motor de juego creado por la empresa alemana desarrolladora de software Crytek, originalmente un motor de demostración para la empresa Nvidia, que al demostrar un gran potencial se implementa por primera vez en el videojuego Far Cry, desarrollado por la misma empresa creadora del motor. El 30 de marzo de 2006, la totalidad de los derechos de CryEngine son adquiridos por la distribuidora de videojuegos Ubisoft. Es compatible con la mayoría de las plataformas del mercado.

Recientemente Amazon ha lanzado su propio motor basado en CryEngine llamado **Lumberyard** de forma gratuita pero ligado fuertemente a los servicios en la nube de la plataforma de Amazon.

**Unity3D** es un motor de videojuego multiplataforma creado por Unity Technologies. Unity está disponible como plataforma de desarrollo para Microsoft Windows, OS X y Linux, y permite crear juegos para Windows, OS X, Linux, Xbox 360, PlayStation 3, Playstation Vita, Wii, Wii U, iPad, iPhone, Android y Windows Phone. Gracias al plugin web de Unity, también se pueden desarrollar videojuegos de navegador para Windows y Mac. **Se ha convertido en la referencia para los desarrolladores indies.**

<http://www.vidaextra.com/listas/si-quieres-hacer-tus-propios-juegos-estos-son-los-mejores-motores-que-vas-a-encontrar3>

**Godot:** un motor muy reciente, permite realizar juegos 2D y 3D, su licencia es MIT que es incluso más permisiva que la GPL.

### 3. Conceptos previos videojuegos

#### 3.1. Sprite

Un **sprite** (duendecillo o hada en inglés) es una imagen usada para representar un ente gráficamente (o parte de él) y poder posicionarlo en el lugar deseado de una escena mayor. Mediante este sistema además se pueden crear animaciones que representen dicho ente cambiando el sprite correspondiente. Un ente no tiene estar representado gráficamente por un único sprite sino que puede estar dividido en varios diferentes por:

- Limitaciones técnicas (tamaño máximo del sprite, número de colores, etc.)
- Optimización (especialmente si tiene formas irregulares o inconexas, o usar la técnica dirty rectangles)
- Crear representaciones mediante composición (por ejemplo, cambiar el arma o vestimenta de un personaje).

Los sprites pueden ser cualquier imagen o información que representa una imagen. Si bien teóricamente nada impide la usar imágenes vectoriales, las basta mayoría de las veces esta técnica es usada exclusivamente con **imágenes bitmap**.

Los sprites, en caso de ser un mapa de bits, para no tener que ser rectangulares o requerir un fondo con solo color, además suelen contener información sobre sus partes transparentes como puede ser utilizar un color de la paleta, uso de otra imagen con la máscara de transparencia, o directamente la cantidad de componente alpha de cada pixel.

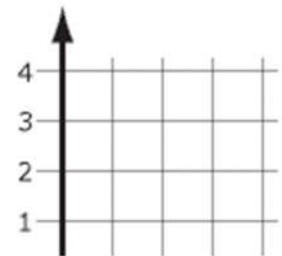
Una forma muy común para almacenar todos los sprites es el uso de **sprite sheets** o plantillas de sprites, que consisten en imágenes que contienen todos los sprites, normalmente relacionados que pertenecen a un ente, y que también en vez de almacenar internamente todos los sprites por separado lo que se hace es utilizar directamente tan solo la parte de la plantilla que corresponde a ese sprite.

### 3.2. Coordenadas

Uno de los más importantes conceptos dentro del desarrollo de videojuegos es el eje de coordenadas. Es tan importante debido a que nos permite recrear un plano o un espacio y posicionar en su interior los diferentes elementos que forman el videojuego.

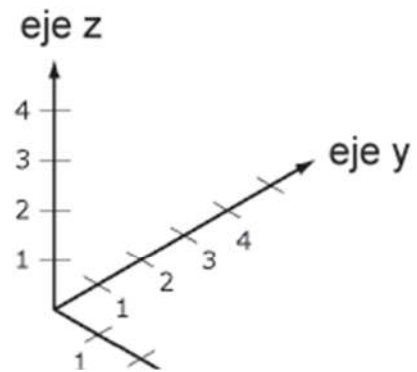
Los ejes de **coordenadas cartesianas** son una parte fundamental de las matemáticas y la física, ya que nos sirven para representar funciones y también posiciones. En ellos, cada eje representa una dimensión, por lo que si hablamos de videojuegos encontraremos dos ejes básicos.

**Eje de coordenadas en el plano:** representado por dos ejes X e Y que indican las dimensiones ancho y alto respectivamente. Se utiliza en juegos en **dos dimensiones**.

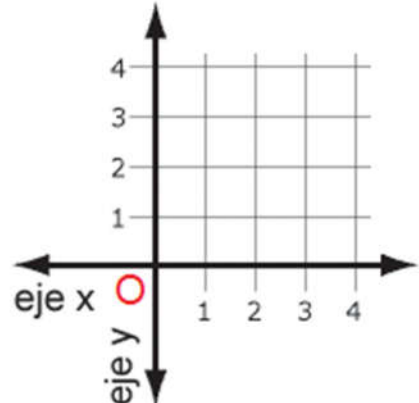


**Eje de coordenadas en el espacio:**

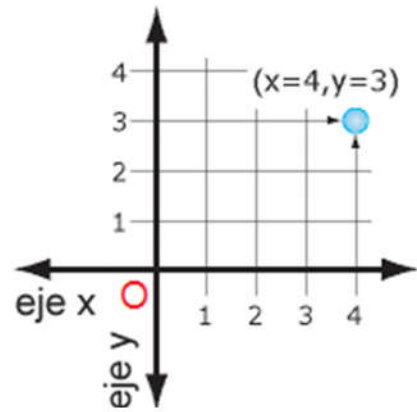
Representado por tres ejes, X, Y y Z que indican las dimensiones ancho, alto y profundidad respectivamente. Se utiliza en juegos en tres dimensiones, aunque también sirve para juegos en 2D cuando se utiliza el eje Z para indicar qué elementos se encuentran por encima de los otros en el plano, algo que normalmente se indica como una propiedad de los objetos que determina el orden de renderizado de abajo hacia arriba.



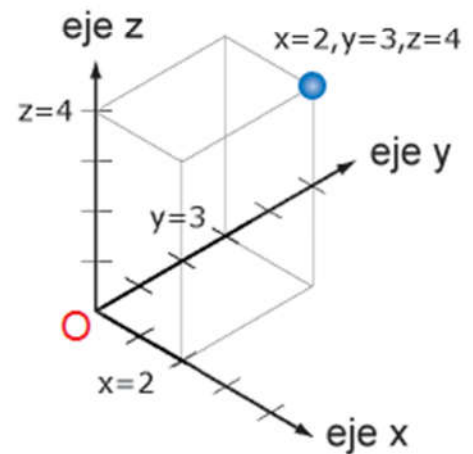
**Origen de coordenadas:** El término origen es muy utilizado cuando se habla del eje de coordenadas. Corresponde al lugar donde se cortan los ejes y tiene el valor cero. Es muy importante ya que gracias a él podemos tomar una referencia inicial a partir de la cual posicionar elementos sobre el eje de coordenadas, utilizando coordenadas, valga la redundancia.



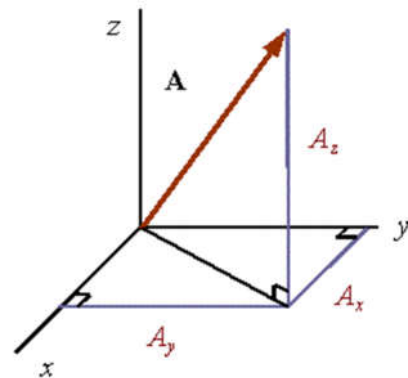
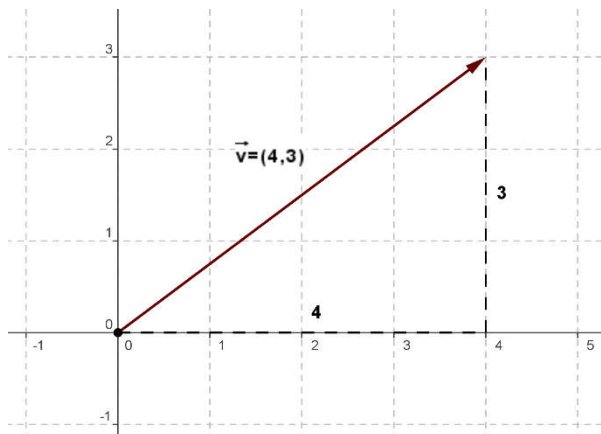
**Representación de elementos:** Tomando un punto sobre el eje de coordenadas (que es la mínima unidad que se puede representar), cada coordenada nos indica la distancia de cada dimensión partiendo del origen donde se encuentra el punto. Por ejemplo un punto A en el plano 2D estará formado por dos coordenadas  $A = (X,Y)$ :



Mientras que en el espacio 3D estará formado por tres  $A = (X,Y,Z)$ .



Otro elemento que se puede representar sobre el eje de coordenadas es un **vector**, que no es más que la distancia entre el origen y un punto, con dirección al punto:



## 4. Unity

### 4.1. Introducción

Unity es un motor gráfico 3D para PC y Mac que viene empaquetado como una herramienta para crear juegos, aplicaciones interactivas, visualizaciones y animaciones en 3D y tiempo real. Unity puede publicar contenido para múltiples plataformas como PC, Mac, Nintendo Wii y iPhone. El motor también puede publicar juegos basados en web usando el plugin Unity web player. Como motor gráfico, este es posiblemente el mejor motor por debajo de los 100.000€.

El editor de Unity es el centro de la línea de producción, ofreciendo un completo editor visual para crear juegos. **El contenido del juego es construido desde el editor** y el gameplay se programa usando un lenguaje de scripts. Esto significa que los desarrolladores no necesitan ser unos expertos en C++ para crear juegos con Unity, ya que las mecánicas de juego son compiladas usando una versión de JavaScript, **C#** o Boo, un dialecto de Python (el soporte a Boo se eliminó a partir de la versión 5).

Los juegos creados en Unity son estructurados en **escenas** como el motor Gamebryo. En Unity una escena puede ser cualquier parte del juego, desde el menú de inicio como un nivel o área de tu juego; la elección es tuya ya que una escena es un lienzo en blanco sobre el que dibujar cada parte del juego usando las herramientas de Unity.

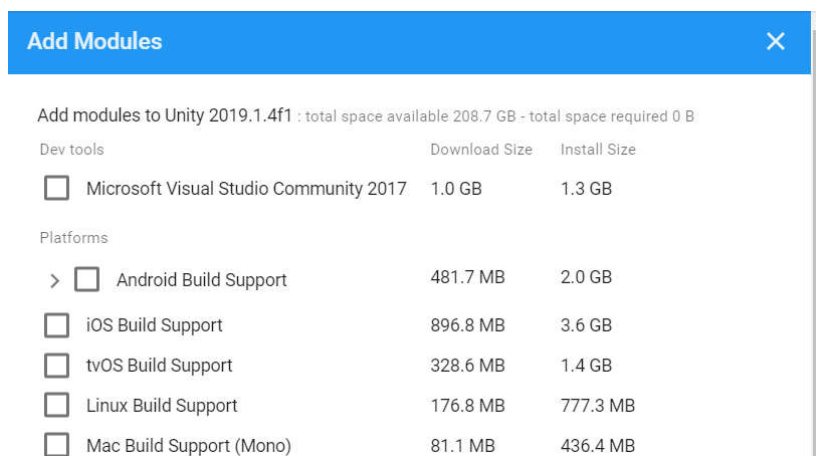
El motor también incluye un editor de terrenos, desde donde puedes crear un terreno (como una hoja en blanco), sobre la que los artistas podrán esculpir la geometría del terreno usando herramientas visuales, pintar o texturizar, cubrir de hierba o colocar árboles y otros elementos de terreno importados desde aplicaciones 3D como 3DS Max o Maya. Puede descargar gratuitamente desde <http://unity3d.com/es/get-unity/download?ref=personal>

## 4.2. Instalación de Unity

En versiones reciente de Unity es posible descargar el editor de forma independiente, o bien, a través de un instalador independiente llamado **Unity Hub** que permite instalar varias versiones diferentes del editor en el mismo equipo, de forma que abriremos cada proyecto con la versión deseada.

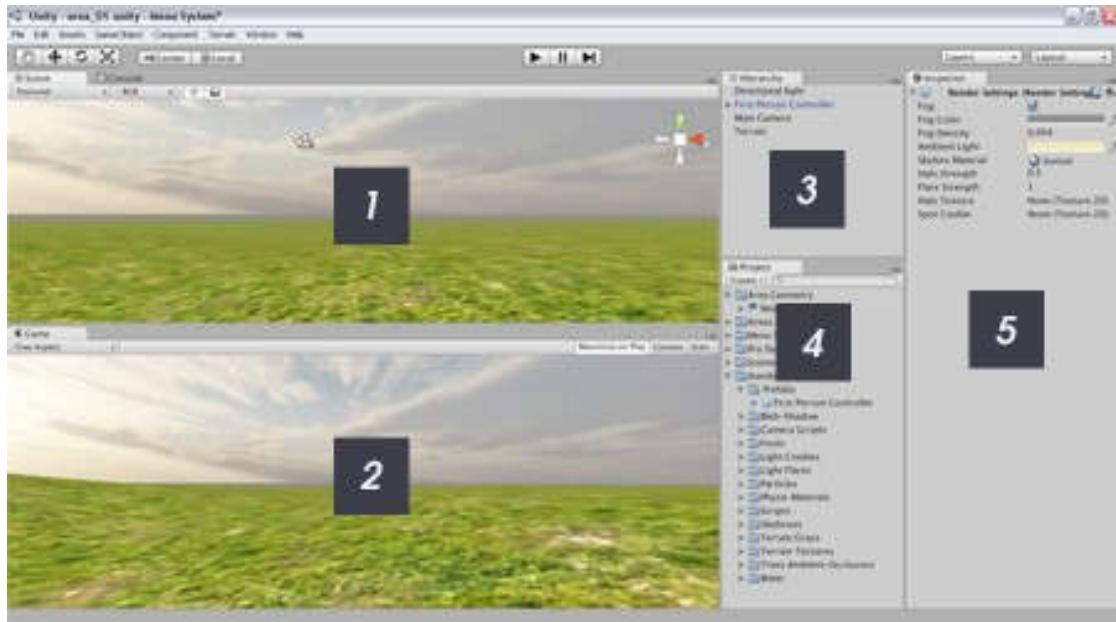


A través de Unity Hub también se pueden instalar los módulos de soporte para distintas plataformas (Android, Mac, etc...) y el editor de código para scripts (normalmente Visual Studio).



### 4.3. La interfaz de usuario de Unity

Existen principalmente 5 áreas de la interfaz de Unity, numeradas en la imagen de abajo.



#### 1 Vista de Escena

La escena es el área de construcción de Unity donde construimos visualmente cada escena de nuestro juego. Vista de Juego

En la vista de juego obtendremos una previsualización de nuestro juego. En cualquier momento podemos reproducir nuestro juego y jugarlo en esta vista.

#### 2 Vista de Juego

En esta vista vemos el juego en acción y podemos interactuar con el mismo.

#### 3 Vista de Jerarquía

La vista de jerarquía contiene todos los objetos en la escena actual.

#### 4 Vista de Proyecto

Esta es la librería de assets para nuestro juego, esto incluye modelos 2D/3D, código, sonidos, etc. Puedes importar objetos 3D de distintas aplicaciones a la librería, puedes importar texturas y crear otros objetos como Scripts o

Prefabs que se almacenaran aquí. Todos los assets que importes en tu juego se almacenaran aquí para que puedas usarlos en tu juego.

Ya que un juego normal contendrá varias escenas y una gran cantidad de assets es una buena idea estructurar la librería en diferentes carpetas que harán que nuestros assets se encuentren organizados y sea más fácil trabajar con ellos.

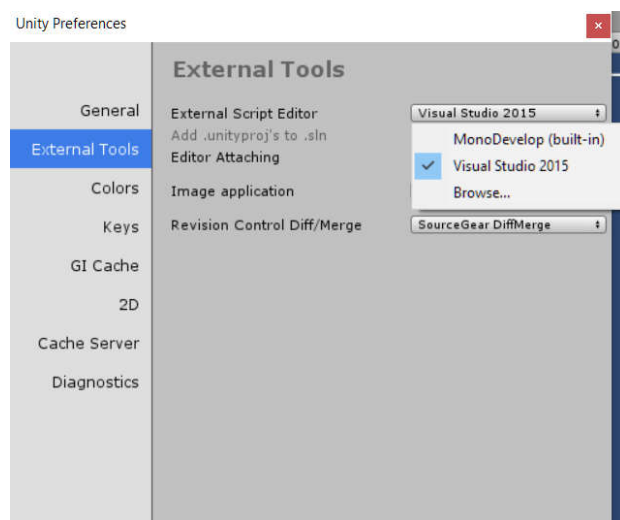


## 5 Vista de Inspector

La vista de inspector sirve para varias cosas. Si seleccionas objetos entonces mostrara las propiedades de ese objeto donde puedes personalizar varias características del objeto. También contiene la configuración para ciertas herramientas como la herramienta de terrenos si tenemos el terreno seleccionado.

**Los scripts son escritos usando editores** como MonoDevelop (incluido en la instalación) o Visual Studio. Puedes elegir que editor utilizar desde el menú *Edit -> Preferences*

Unity detectará que editores compatibles tienes instalados.



Cuando creamos un script, podemos saltar al editor de scripts desde Unity, guardar el script y usarlo en el juego.

Al igual que muchas aplicaciones **podemos modificar la interfaz por defecto de Unity**. Prueba a hacer clic sobre la pestaña de la vista de juego y arrastrarlo al lado de la pestaña de la vista de escena, veras que aparece un indicador de posición. Suelta el botón del ratón y la vista de juego no estará en la misma posición que la vista de escena, estará a un lado. Ahora podremos cambiar entre la vista de escena y la de juego al hacer click sobre la pestaña apropiada. Esto te dará mucho más espacio para la vista de escena, lo que resulta muy útil.



Cuando trabajemos con la vista de escena es útil también saber que pulsar el espacio agrandara la vista de escena hasta completar el editor por completo.

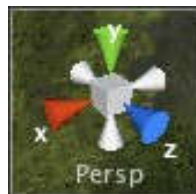
#### 4.4. Vista de escena

La vista de escena es un entorno 3D para crear cada escena. Trabajar con la vista de escena, en la forma más sencilla, sería arrastrar un objeto desde la vista de proyecto a la vista de escena que colocara el objeto en la escena; entonces podrás posicionarlo, escalarlo y rotarlo sin salir de la vista de escena.

La vista de escena es también el lugar donde editas los terrenos (esculpiéndolos, pintando texturas y colocando elementos), colocas luces y cámaras y otros objetos.

#### **Modos de visualización**

Por defecto la vista de escena tiene una perspectiva 3D de la escena. Podemos cambiar esto por un número de vistas Ortográficas: top down, side y front. En la parte derecha de la vista de escena veréis un "Gizmo" que parece una caja con conos que salen de ella.



Podemos usar este Gizmo para cambiar la perspectiva:

Hacer clic sobre la caja nos llevara al modo perspectiva.

Hacer clic en el cono verde (y) nos llevara al modo Top-Down.

Hacer clic sobre el cono rojo (x) nos llevara al modo Side (derecha).

Hacer clic sobre el icono azul (z) nos llegara al modo Front(frontal).

También tenemos 3 conos grises que nos llevaran a los siguientes modos: "Back, Left y Bottom", es castellano, Atrás, Izquierda y Abajo.

#### 4.4.1. Configuración de la visualización

En la esquina izquierda de la vista de escena encontraremos un conjunto de botones para cambiar la configuración general de la visualización. Vamos a ver cada uno de ellos, de izquierda a derecha.



**Render Mode:** La primera opción es Render mode o modo de renderización en castellano. Por defecto aparecerá en "Shaded". Si hacemos clic aparecerá una lista desplegable con un número de diferentes opciones de renderizado:

- Shaded: Las texturas se renderizan en la vista.
- Wireframe: Las superficies no se renderizan, solo vemos la malla.
- Shaded Wireframe: Las texturas se renderizan, pero también vemos la malla.

**2D:** La segunda opción es el modo de visualización, aparecerá por defecto en 2D si nuestro proyecto lo hemos creado para 2D. Podemos seleccionar el tipo de vista entre 2D y 3D.

**Interruptor de luces:** El siguiente botón enciende o apaga la iluminación del escenario. Apagar la iluminación resultara en una escena mostrada sin luces; lo que puede ser útil para el rendimiento y también si no hay luces en la escena.

**Interruptor de sonido:** permite encender y apagar el sonido.

**Interruptor de skybox, lense flare y niebla:** El último botón activa y desactiva estos tres efectos. Esta opción es útil para desactivar los efectos por razones de rendimiento o visibilidad al trabajar sobre una escena.

**Gizmos:** permite visualizar en pantalla elementos que **no** van a ser mostrados durante la ejecución en la plataforma de destino, por ejemplo un símbolo donde se reproduzca un sonido, el icono de la cámara, los colliders, etc...son elementos que nos sirven para depurar durante la ejecución.

**Buscador:** nos permite buscar un elemento dentro de la escena.

#### 4.4.2. Botones de Control

Debajo de las opciones de visualización veras una fila con 4 botones, como en la imagen de abajo.



Puedes usar Q, W, E, R para alternar entre cada uno de los controles, que detallamos debajo:

- *Hand Tool* (Q): Este control nos permite movernos alrededor en la vista de escena.

Mantener ALT nos permitirá rotar, COMMAND/CTRL nos permitirá hacer zoom y SHIFT incrementa la velocidad de movimiento mientras usas la herramienta.

- *Translate Tool* (W): Nos permite mover cualquier objeto seleccionado en la escena en los ejes X, Y y Z.
- *Rotate Tool* (E): Nos permite rotar cualquier objeto seleccionado en la escena.
- *Scale Tool* (R): Nos permite escalar cualquier objeto seleccionado en la escena.



Editor de vértices, nos permite deformar una malla a través del editor.

#### 4.5. Vista de Proyecto

La vista de proyecto es esencialmente una librería de assets para el proyecto de nuestro juego. Todos los componentes del juego que crees desde el editor y todos los objetos que importes como modelos 3D, texturas, efectos de sonido, música etc. se guardaran ahí.

Como este panel contiene todos los assets de tu juego, y no solo los que están en la escena actual, es importante mantener una buena estructura. Podemos crear carpetas y colocar los objetos dentro de esas carpetas para crear una jerarquía de carpetas.

## Notas de Uso

- No crees la estructura de tu librería o nuevas assets usando Windows Explorer o Finder ya que Unity puede perder enlaces y dependencias importantes. Como practica general debemos usar las herramientas de organización de la vista de proyecto para crear la estructura y organizar los assets puesto que Unity seguirá su localización.
- En la parte superior de la vista de proyecto veras un botón "Create", que mostrara una lista desplegable con varias opciones de creación. Podremos crear carpetas, scripts, shaders, animaciones y otros tipos de objetos usando este panel.
- Hacer dos clic sobre un ítem en la librería nos permitirá renombrarlo.
- Puedes hacer clic y arrastrar carpetas e ítems para organizar la estructura.
- Puedes importar y exportar paquetes (colecciones de assets) simplemente haciendo clic derecho sobre la vista de proyecto y seleccionando la opción apropiada.
- Puedes importar assets como archivos de audio, texturas modelos etc. con hacer clic derecho y seleccionar "Import Asset"

### 4.6. Vista de Jerarquía

La vista de jerarquía contiene una lista de todos los objetos usados en la escena actual. Cualquier objeto que coloques en la escena aparecerá como una entrada en la jerarquía.

La jerarquía también sirve como método rápido y fácil para seleccionar objetos en la escena. Si quieres por ejemplo, seleccionar un objeto de la escena puedes seleccionarlo desde la jerarquía en lugar de moverte por la escena, encontrarlo y seleccionarlo.

Cuando un objeto es seleccionado en la jerarquía también lo es en la vista de escena, donde puedes moverlo, escalarlo, rotarlo, borrarlo o editarlo. El inspector también mostrara las propiedades del objeto seleccionado; de esta forma la jerarquía sirve como una herramienta útil para seleccionar rápidamente objetos y editar sus propiedades.

#### 4.7. El Inspector

El inspector es esencialmente un panel de propiedades; si seleccionas un objeto en la escena todas las propiedades editables aparecerán en el inspector.

Por ejemplo, si seleccionas una luz o cámara, el inspector te permitirá editar varias propiedades de la luz o de la cámara. Adicionalmente el inspector sirve como panel de herramientas para ciertos tipos de objetos. Por ejemplo, si seleccionas un terreno el inspector mostrara las opciones de terreno y también el editor con herramientas como esculpir, texturizar, etc.

Como el inspector cambia según el objeto que seleccionemos la mejor forma de aprenderlo es probarlo.

#### 4.8. Vista de Juego

En Unity puedes ejecutar tu juego sin salir del editor, lo que es una bendición para los diseñadores que están construyendo niveles y los desarrolladores que están añadiendo nuevas mecánicas de juego.



La imagen sobre estas líneas muestra los controles de reproducción, que están localizados en la parte superior del editor. Puedes entrar en la previsualización del juego en cualquier momento pulsando el botón de reproducción (el primero por la izquierda), pausar usando el botón de pausa (central) o saltar adelante usando el botón derecho.

Puedes jugar desde la vista de juego o extenderla a pantalla completa.

El panel también tiene un menú contextual en la esquina izquierda que aparece como "Free Aspect" por defecto, de esta lista podremos seleccionar un número de proporciones para el juego, lo que es ideal para probar nuestro juego en distintas pantallas y plataformas.

## 5. Scripting

En este apartado veremos algunos de las operaciones más habituales a realizar desde los scripts C#.

- **C#**

**C# es un lenguaje muy similar a java, no te costará nada acostumbrarte.**

- Tipos básicos: int, double, bool, string...
- Arrays: `int[] enteros = new int[7] {0,0,0,0,0,0,0};`
- Cada Script genera una clase con el nombre del fichero, que hereda de `MonoBehaviour` :

```
public class nombreClase : MonoBehaviour {
```

- Las funciones son muy similares a java :

```
void Start () {
```

- En vez de import utilizamos using : `using System;`

- Obtener la referencia a un objeto de la escena :

```
GameObject miObjeto = GameObject.Find("objetoEnEscena");
```

- Mover un objeto

Utilizamos el componente transform del objeto y se utiliza un vector de 3 coordenadas (x,y,z). Por ejemplo lo podemos situar en la punto central de la escena :

```
miobjeto.gameObject.transform.position = new Vector3(0, 0, 0);
```

- Acceso a características de un objeto

A través de `GetComponent` podemos acceder a los componentes del objeto (se pueden consultar en el inspector) por ejemplo el `Rigidbody`

```
miObjeto.GetComponent<Rigidbody>().collisionDetectionMode =
```

```
CollisionDetectionMode.ContinuousDynamic;
```

- Instanciar un objeto en tiempo de ejecución

Para ello necesitamos la referencia al objeto, la posición donde queremos que aparezca y la rotación.

```
//primero cargamos el objeto desde la carpeta Resource
```

```
GameObject miObjeto = (GameObject)Resources.Load("objeto");
```

```
Instantiate(miObjeto, new Vector3(0,0,0), Quaternion.identity);
```

- Destruir un objeto

```
DestroyObject(miObjeto.gameObject);
```

- Control del tiempo

Podemos realizar distintas operaciones con el objeto Time, por ejemplo pausar la ejecución :

```
Time.timeScale = 0;
```

Más información en:

<http://docs.unity3d.com/ScriptReference/Time.html>

- Carga de una escena

```
Application.LoadLevel("nombreEscena");
```

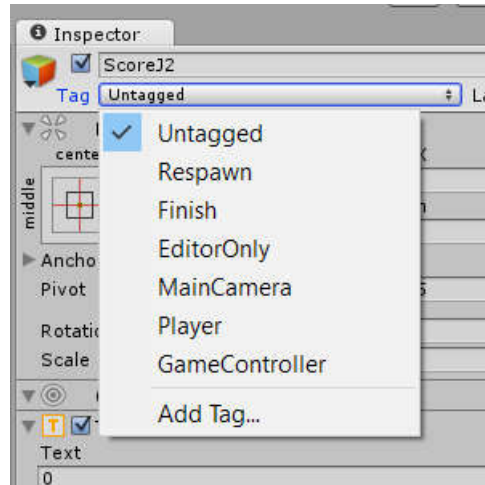
Este método está en desuso (deprecated) mejor utilizad

```
SceneManager.LoadScene("nombreEscena");
```

- Acceso a todos los objetos con un tag determinado

```
GameObject contenedor_objetos =  
GameObject.FindGameObjectWithTag("miAsset");  
  
foreach (Transform objeto in contenedor_objetos.transform){  
  
    ...  
  
}
```

Los tags pueden asignarse a los objetos dentro de la escena de forma que un grupo de objetos compartan el mismo tag.



- Preferencias de usuario PlayerPrefs

Podemos guardar valores **entre ejecuciones** del juego, por ejemplo volumen del sonido, puntuaciones, etc... para ello se utiliza la clase PlayerPrefs:

```
PlayerPrefs.SetString("nombreJugador", "pepe");  
...
```

```
String miJugador = PlayerPrefs.GetString("nombreJugador");
```

Existen funciones para guardar/recuperar cadenas y tipos numéricos, más información en:

<http://docs.unity3d.com/ScriptReference/PlayerPrefs.html>

- Acceso al script de otro objeto de la escena:

Puede ser necesario acceder al script asociado a otro objeto en cuyo caso :

```
private Script_OtroObjeto script_otroObjeto;  
  
//obtenemos la referencia al objeto dentro de la escena  
GameObject otroObjeto = GameObject.Find("otroObjeto");  
  
//obtenemos la referencia al script dentro del objeto  
  
script_otroObjeto = otroObjeto.GetComponent<Script_OtroObjeto>();
```



## 6. Primer juego - Pong

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

### 6.1. Creando un proyecto

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

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



Projects Getting started

Project name\*

Pong

Location\*

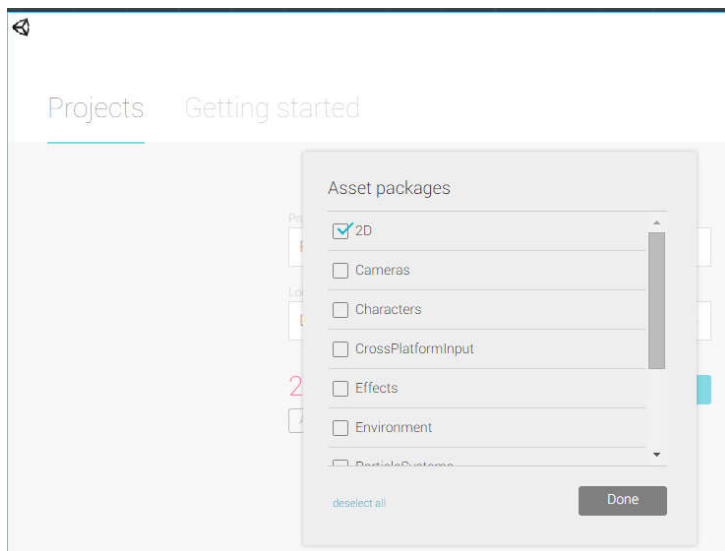
D:\UnityProjects

2D 3D

Asset packages...

Create project

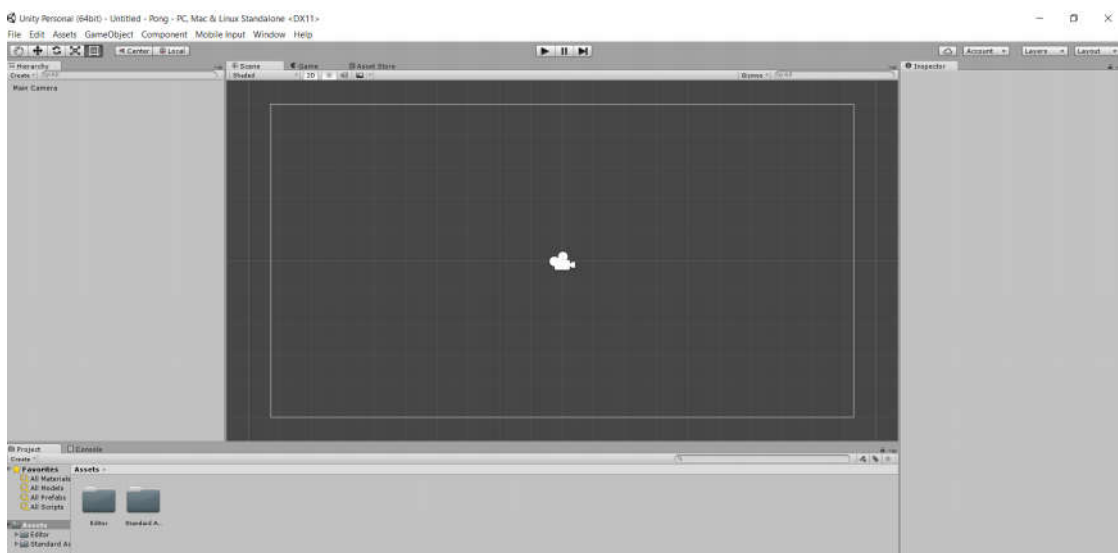
Podemos incluir una serie de assets (componentes) que nos ayuden a la realización de nuestro proyecto, estos pueden ser añadidos posteriormente. Por ejemplo:



Como en la imagen superior selecciona los paquetes solo los paquetes estándar y luego haz clic en "Create Project".

Cuando hagas esto Unity se reiniciara, creara la estructura del proyecto en la carpeta especificada e importara los paquetes seleccionados al proyecto. Una vez termine se mostrara una escena en blanco con una cámara.

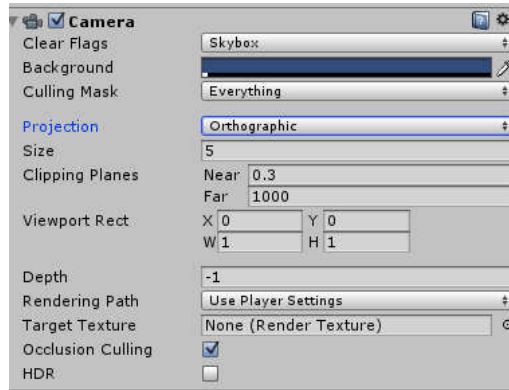
Como veras en la vista de escena, tenemos una escena en blanco ya que no hay nada en ella, pero Unity nos ayuda y ha incluido una cámara inicial por nosotros, que podemos ver en la vista de escena y en la jerarquía.



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

<http://www.desarrollolibre.net/blog/tema/84/blender/diferencias-entre-proyeccion-perspectiva-y-ortogonal-ortografica#.VptwiRXhChd>

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



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

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

## 6.2. La primera escena

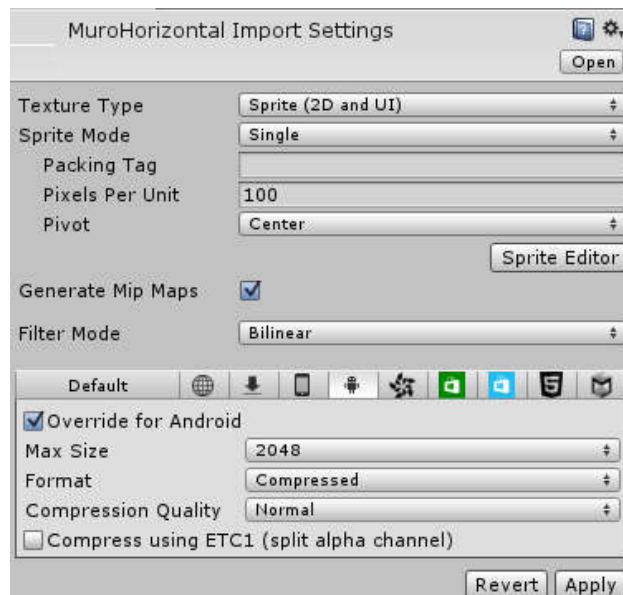
Unity ha creado nuestra primera escena automáticamente al crear el proyecto. Si quieres crear una escena nueva, sencillamente ve a "File -> New Scene".

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

## 6.3. Añadiendo recursos


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

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



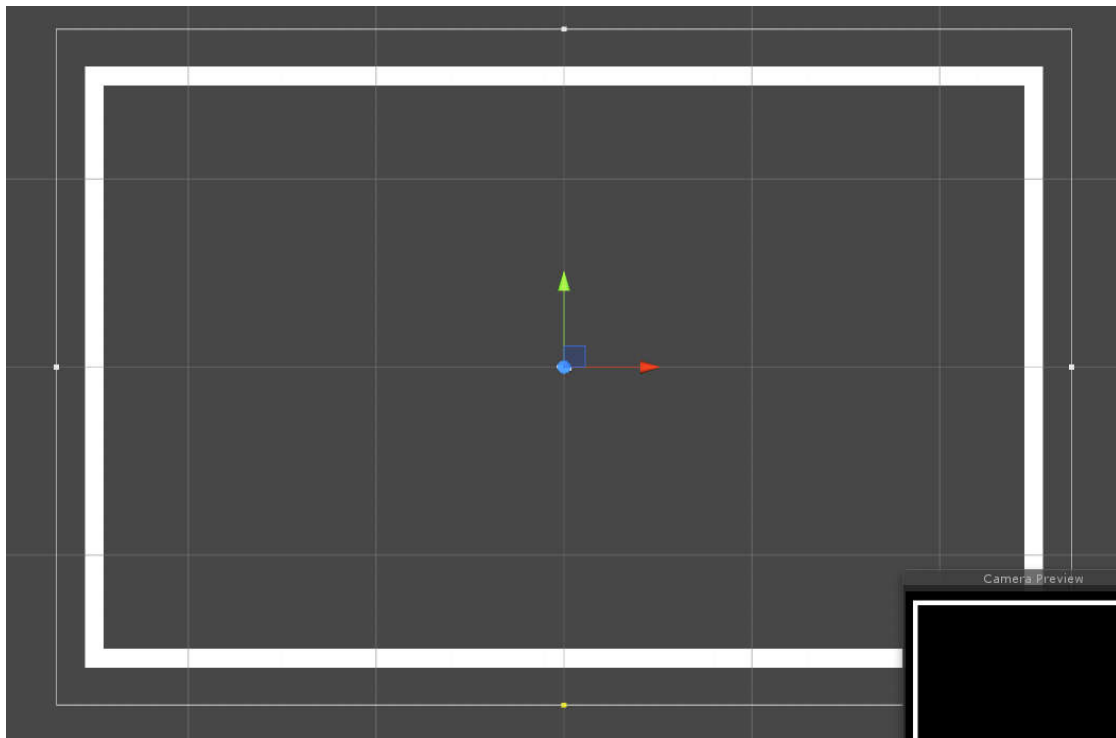
Por ahora no cambiaremos ninguno de los valores, en la parte inferior puede modificarse la calidad de la imagen dependiendo de la plataforma para la que estemos desarrollando.

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

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

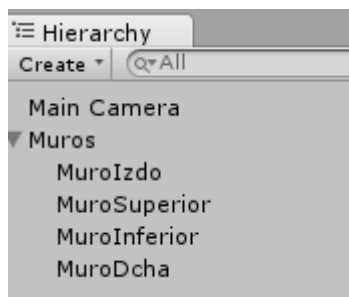


El aspecto una vez introducidos los muros (2 verticales y 2 horizontales) sería similar a este:



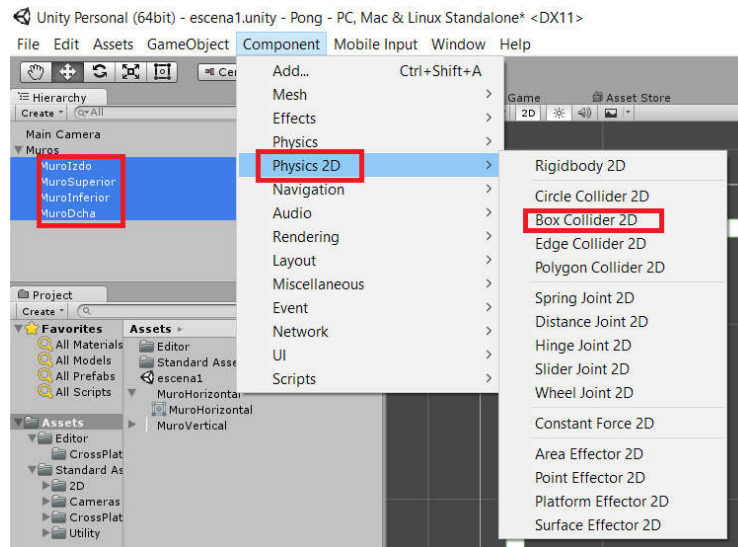
Desde la vista de jerarquía pulsando con el botón derecho podemos renombrar los elementos de la escena e incluso organizarlos en carpetas.

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

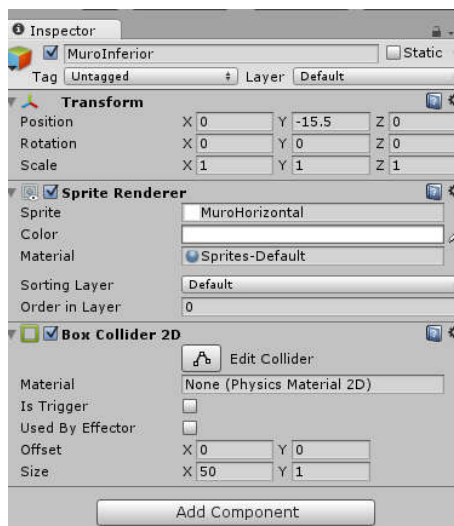


#### 6.4. Añadir el comportamiento físico

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



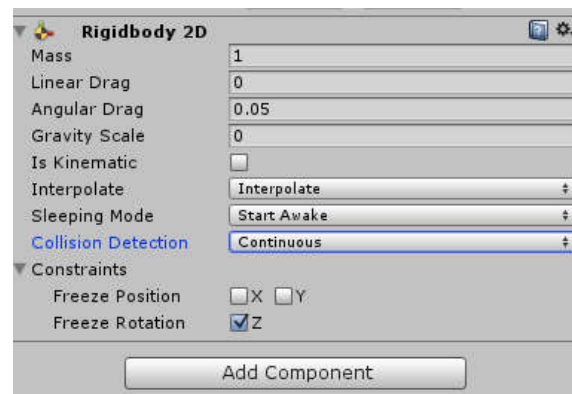
Si seleccionamos un componente en la jerarquía en la ventana del inspector aparecerá el Box Collider:



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

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

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



## 6.5. Movimiento de la raqueta

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

Nota: C # es uno de los lenguajes de programación que se pueden utilizar para secuencias de comandos. Javascript y Boo se pueden utilizar también.

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

Así es nuestro script actualmente:

```
using UnityEngine;
using System.Collections;

public class MoverRaqueta : MonoBehaviour {

    // Use this for initialization
    void Start () {

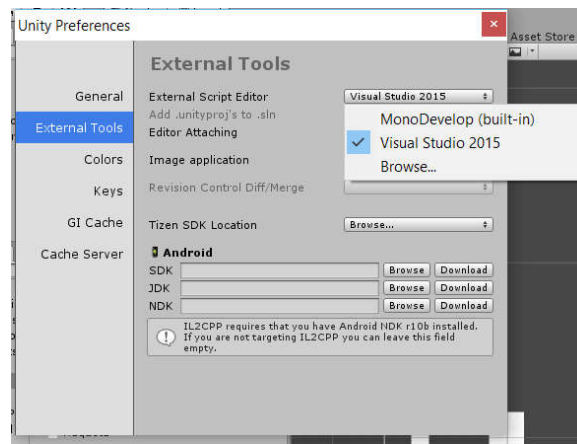
    }

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

    }

}
```

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



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

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

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

```
using UnityEngine;
using System.Collections;

public class MoverRaqueta : MonoBehaviour {
```



```

    void FixedUpdate () {

    }
}

```

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

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

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

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

```

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

```

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

```

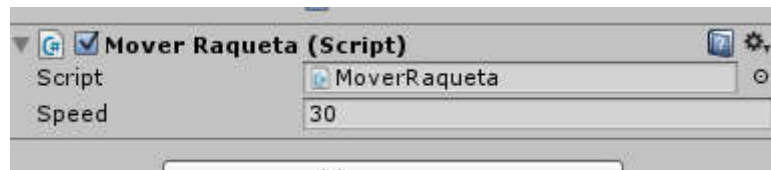
public class MoverRaqueta : MonoBehaviour {
    public float speed = 30;

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

```

Si usáis **Visual Studio**, cuidado con el fin de línea:  
<https://www.youtube.com/watch?v=DTP9AHY23sw>

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



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

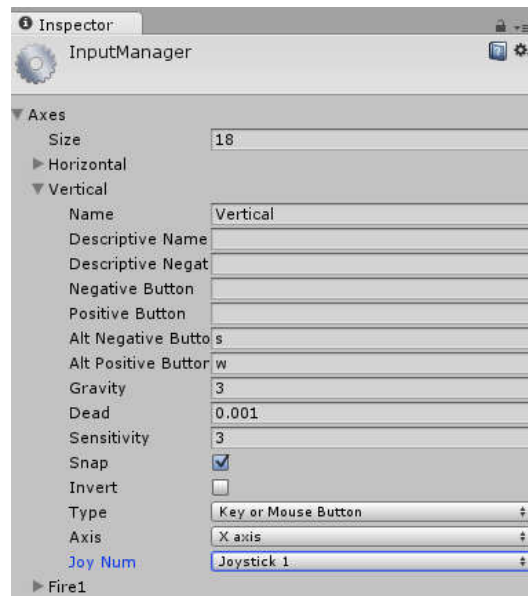
```
void FixedUpdate () {  
    float v = Input.GetAxisRaw("Vertical");  
    GetComponent<Rigidbody2D>().velocity = new Vector2(0,  
v) * speed;  
}
```

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

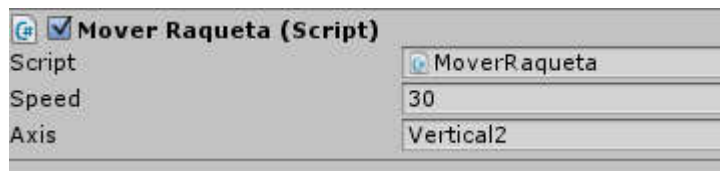
Sólo hay un problema, no podemos movemos las raquetas por separado. Vamos a crear una nueva variable de eje para que podamos cambiar el eje de entrada en el Inspector:

```
public float speed = 30;  
public string axis = "Vertical";  
  
void FixedUpdate()  
{  
    float v = Input.GetAxisRaw(axis);  
    GetComponent<Rigidbody2D>().velocity = new Vector2(0, v) * speed;  
}
```

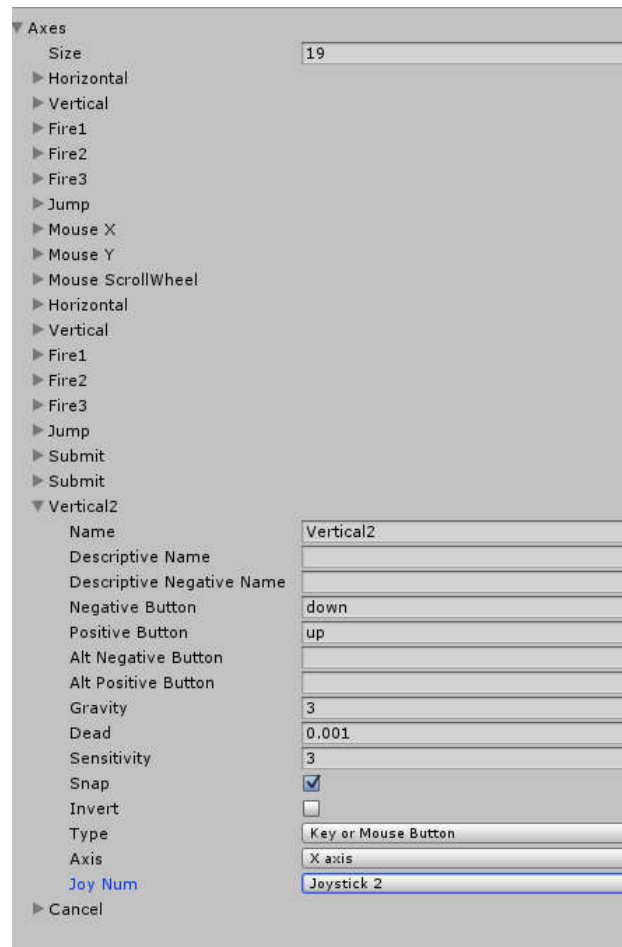
Seleccionamos Edit -> Project Settings -> Input, aquí podemos modificar el eje vertical actual de modo que sólo utilice las teclas W y S. También vamos a hacer que use solamente Joystick 1:



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



Ahora las dos raquetas se moverán de forma independiente.



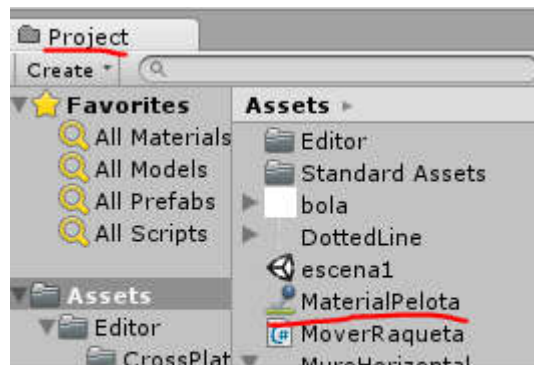
## 6.6. Añadiendo la pelota

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

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

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

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



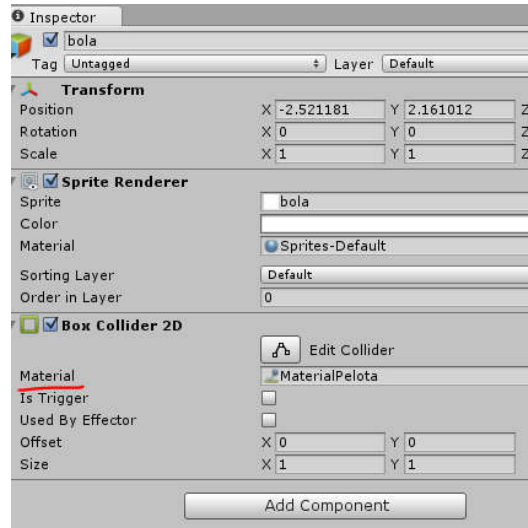
Ajustamos sus valores para que rebote:



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

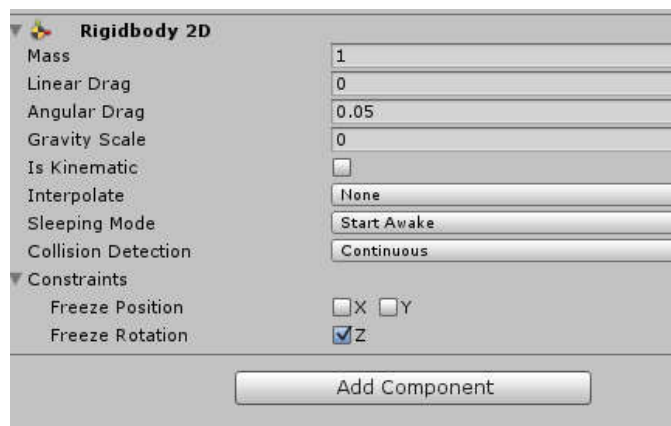
<http://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html>

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



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

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



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

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

```

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

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

```

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

```

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

```

```

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

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

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

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

        // en este caso se mueve hacia la izda (x=-1)
    }
}

```

```

    Vector2 dir = new Vector2(-1, y).normalized;

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

```

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

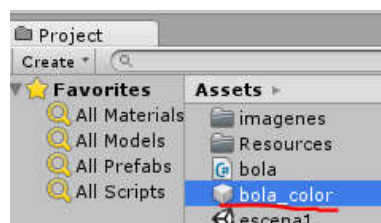
- Incorporar el sonido de la pelota en el choque
- Acelerar la pelota a lo largo del tiempo
- Añadir puntuaciones y que se inicie el juego cada vez que un jugador pierda
- Establecer unas condiciones de victoria y reiniciar la partida
- Incorporar un menú de juego
- ...

## 7. Prefabs

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

<http://docs.unity3d.com/es/current/Manual/Prefabs.html>

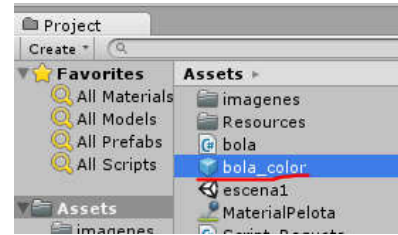
Creamos prefab vacío, botón derecho del ratón Create -> Prefab



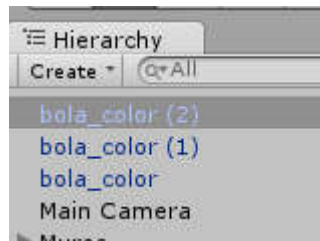


Arrastramos la bola desde la vista de jerarquía sobre el prefab :

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



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



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

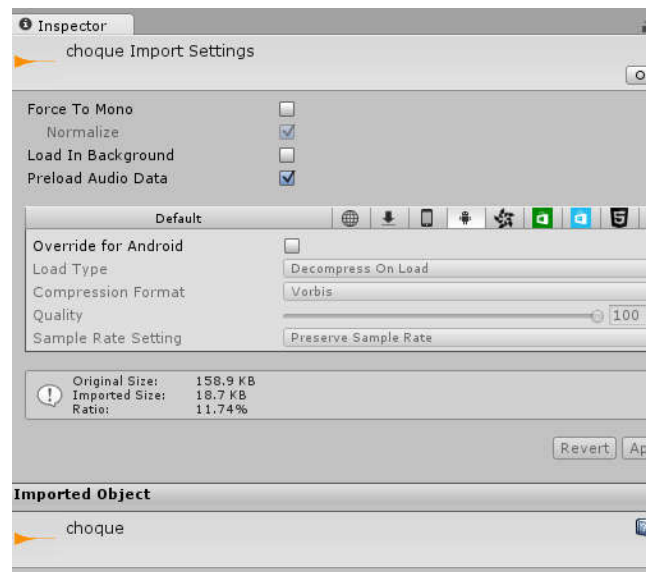
## 8. Sonido

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

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

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

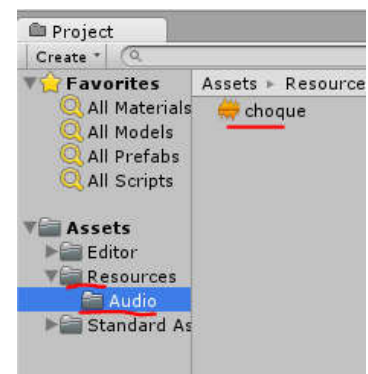
Primero debemos importar los archivos de sonido dentro del editor, igual que con las imágenes será a través de Assets -> Import New Asset.



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



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



Para reproducir el sonido realizamos una llamada a la función:

**AudioSource.PlayClipAtPoint(**

Referencia\_objeto\_sonido, //sonido a reproducir, AudioClip

Vector3\_position, //posición de la que parte el audio

Volumen));

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

```
public float speed = 30;
public AudioClip choque; //declaramos el AudioClip como atributo del script
// Use this for initialization
void Start () {
    GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;

    //Cargo el sonido desde la jerarquía de mi escena
    //(la ruta depende de tu organización (no lleva extensión))
    choque = (AudioClip)Resources.Load("Audio/choque");
}
```

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

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

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

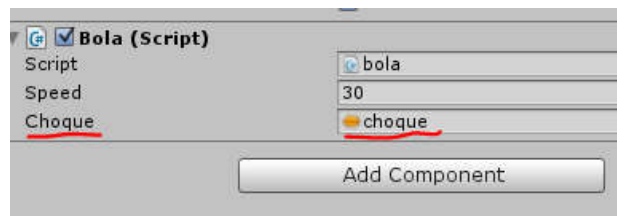
```
AudioSource audio = GetComponent<AudioSource>(); //tenemos que obtener la referencia a un componente de tipo AudioSource
```

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

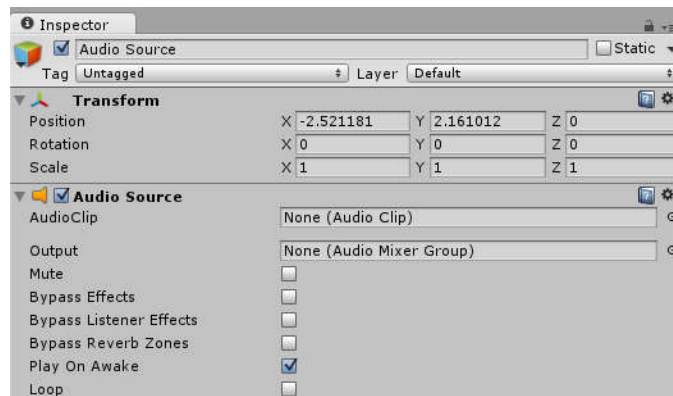
Si no queremos cargar el sonido desde el script mediante :

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

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



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

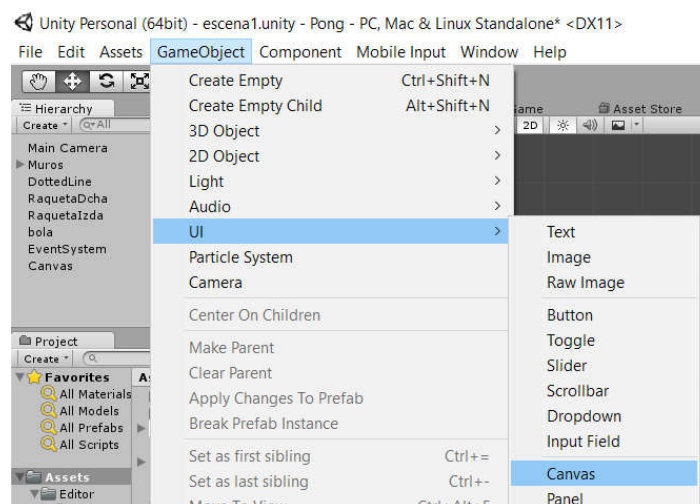


Entre sus propiedades están:

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

## 9. Interfaz de usuario

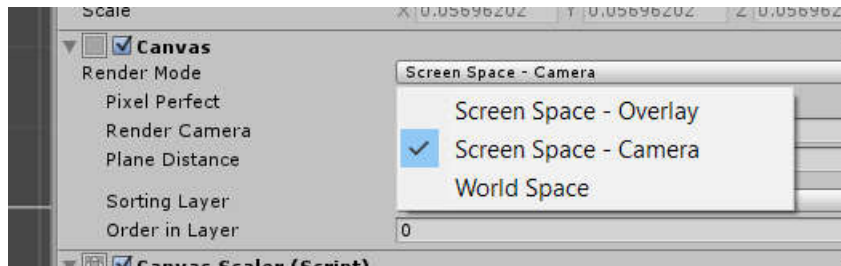
Lo primero que tenemos que crear es un objeto Canvas.



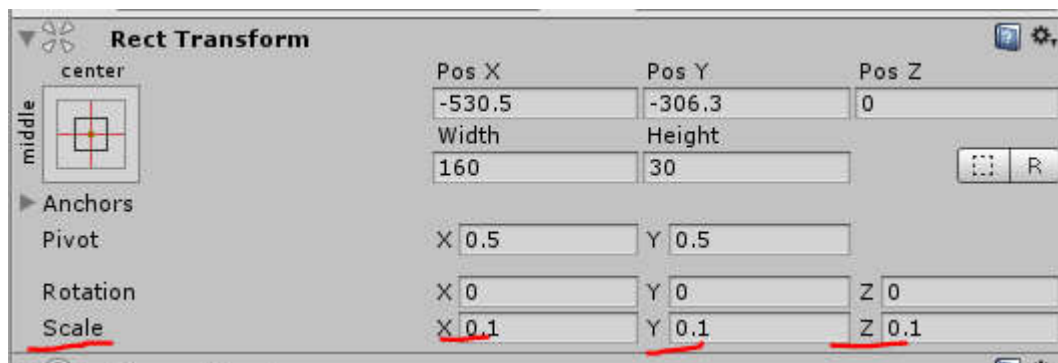
Con el objeto canvas seleccionado podemos insertar nuevos objetos de tipo UI, texto, imagen, botones....que serán hijos del objeto canvas.

Existen 3 formas de mostrar el canvas, aparecen en Render Mode, para más información <http://docs.unity3d.com/Manual/UICanvas.html>,

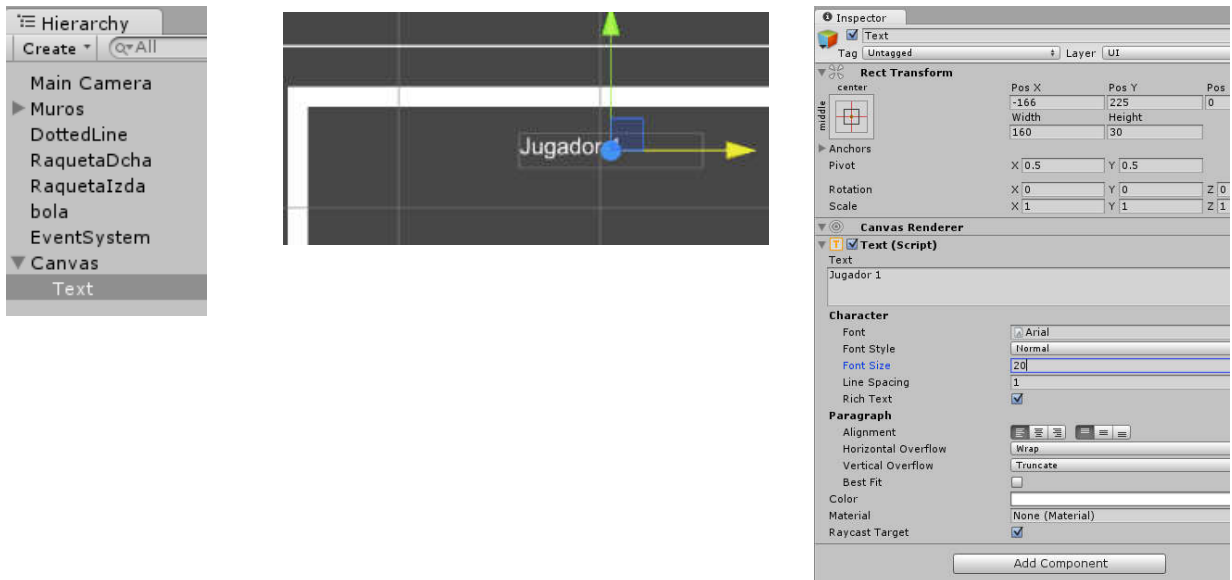
Vamos a elegir **Screen Space- Camera** y seleccionaremos Main Camera en el apartado Render Camera.



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



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



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

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

```
using UnityEngine.UI;

using System;
```

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

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

## 9.1. Gestión de la partida

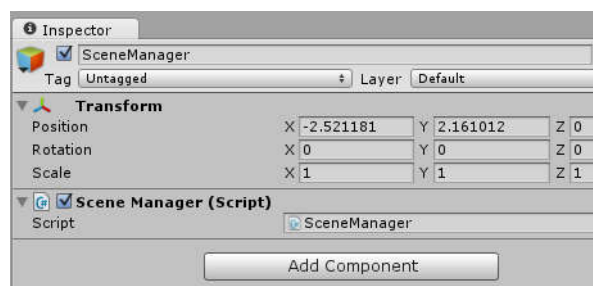
Hasta ahora hemos utilizado una única escena donde desarrollamos la partida, pero el jugador espera un interfaz mínimo y una lógica dentro del juego, es decir, que haya una condición/es de victoria o derrota, que pueda reiniciar la partida o salir, todo aquello que hace que un juego sea agradable de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

### Inicio

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

### Juego

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



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

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

### Fin

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

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

#### 9.1.1. Crear nueva escena

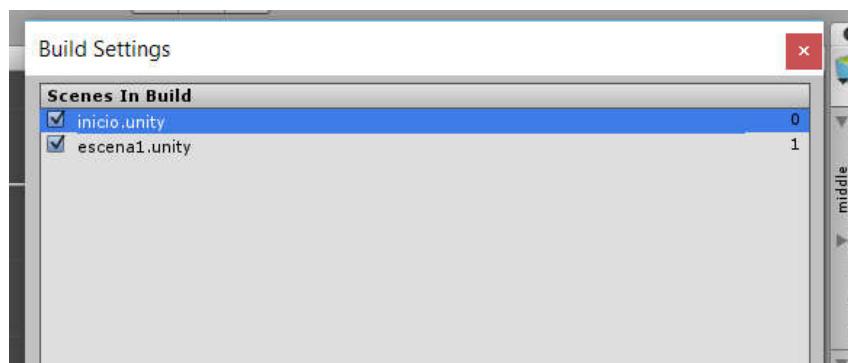
Para crear una nueva escena pulsamos File -> New Scene y nos aparecerá una escena vacía, la guardamos pulsando File -> Save Scene le damos un nombre descriptivo (inicio por ejemplo).

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

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

```
SceneManager.LoadScene("nombre_escena");
```

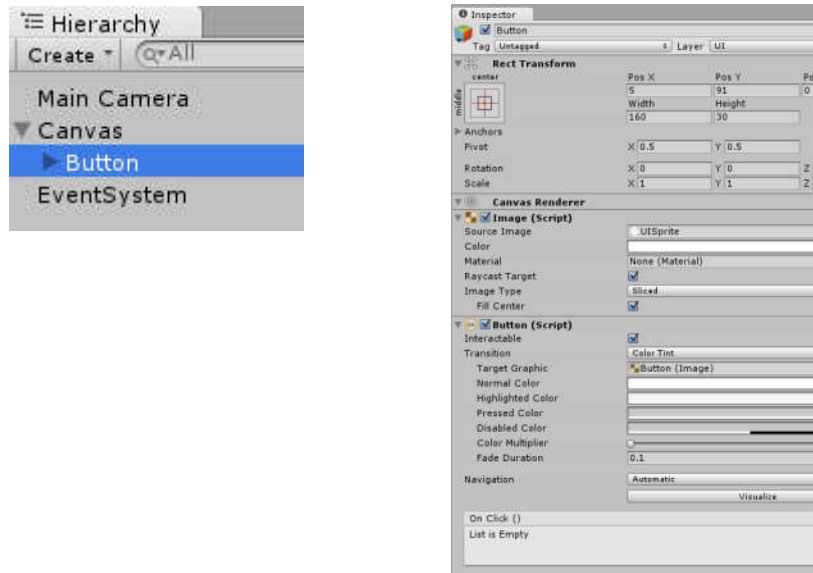
Para que la escena pueda ser cargada debe incluirse en el Build, pulsamos **File -> Build Settings**, se pueden arrastrar desde la vista de Proyecto:





### 9.1.2. Crear una UI con un botón

Creamos un canvas como hemos visto en puntos anteriores, le introducimos un botón a través de GameObject -> UI -> Button :



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



El botón es configurable:

<http://unity3d.com/es/learn/tutorials/modules/beginner/ui/ui-button>

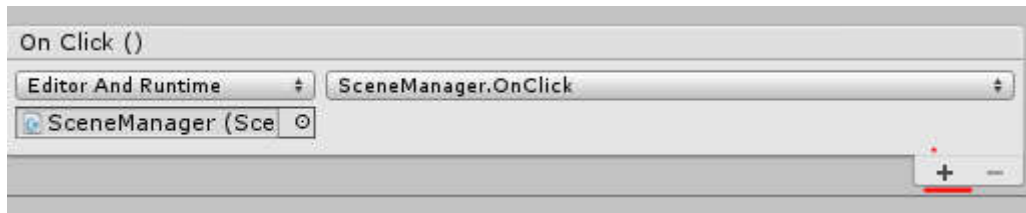
Vamos a utilizar un objeto con un script para recoger el click del ratón, creamos un objeto vacío llamado SceneManager al que asignamos un script llama SceneManager como se vio anteriormente. Dentro del script declaramos un **método público que devuelve void** y que será el encargado de carga la escena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

```

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

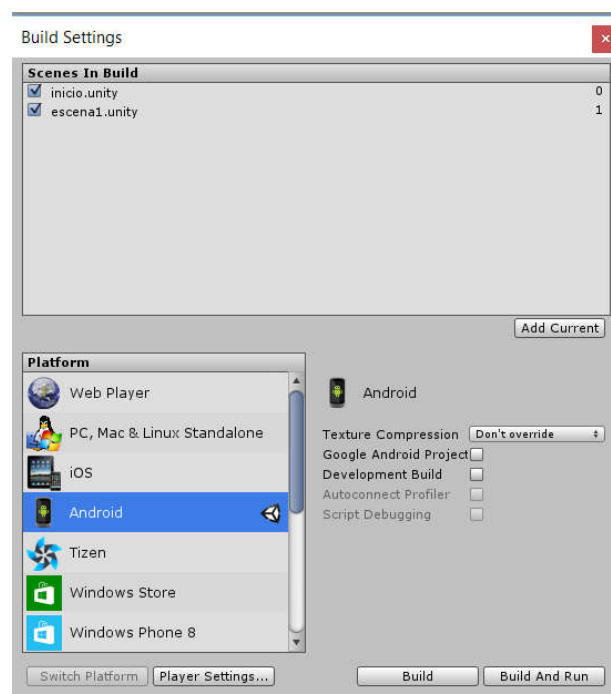
```

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



## 10. Generando el APK

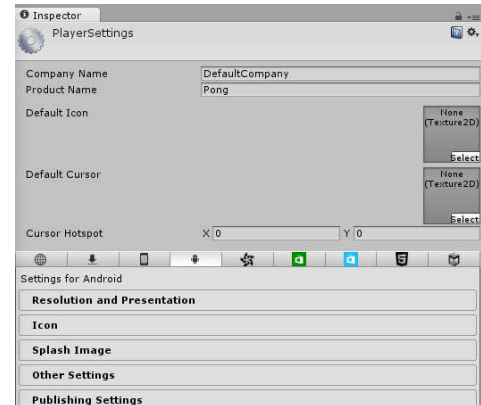
Para compilar todo el proyecto y generar el APK ya sea para probarlo en un dispositivo Android o subirlo a Google Play, pulsamos **File -> Build Settings** :



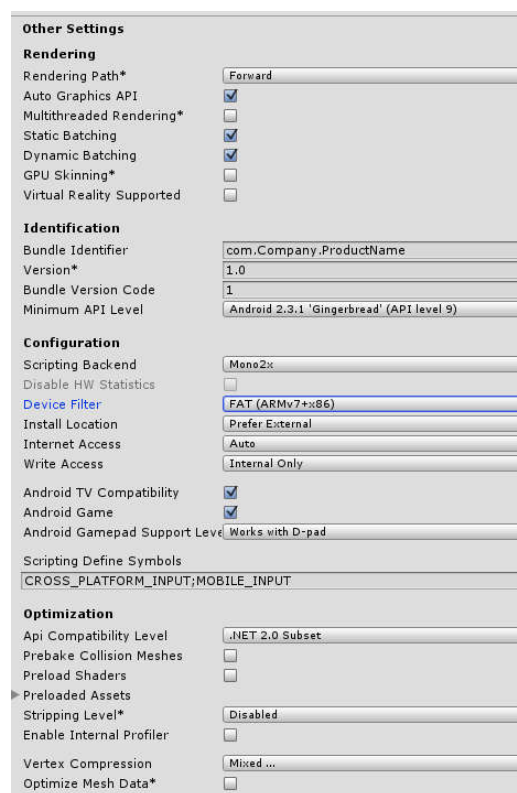
Elegimos Android y pulsamos sobre Player Settings

En esta pantalla debemos configurar los datos de nuestra aplicación

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

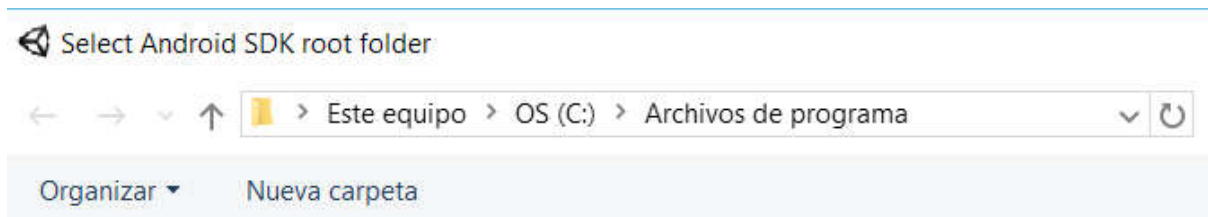


Muy importantes son los datos necesarios en el apartado **Other Settings**:



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

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



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

## 11.Fuentes

- [http://www.elotrolado.net/wiki/Historia\\_de\\_los\\_videojuegos](http://www.elotrolado.net/wiki/Historia_de_los_videojuegos)
- <https://es.wikipedia.org>
- Empezando en Unity3d, UnitySpain