

Tema 1. Motores de juegos 1: Contacto con Unity

1.1. Bibliotecas para juegos y motores de juegos

Existen multitud de bibliotecas y de motores que se pueden utilizar para crear juegos. Por lo general, se tiende a usar la palabra “**biblioteca**” cuando sólo proporciona soporte para tareas muy básicas, como mostrar imágenes, reproducir sonidos o comprobar el estado del teclado y otros dispositivos de entrada. Por el contrario, se suele usar la palabra “**motor**” cuando añade otras funcionalidades de nivel más alto, como soporte para “físicas” (gravedad, colisiones, sistemas de partículas, etc), animaciones, plantillas de juegos o de elementos (personajes con lógicas asociadas, menús, etc), e incluso en ocasiones tiene un entorno de desarrollo específico. Aun así, esta separación no siempre es clara, porque muchas de las bibliotecas básicas se han ido enriqueciendo con el tiempo para añadir más funcionalidades, o bien existen módulos de terceros que permiten ampliarlas y acelerar el desarrollo de programas más complejos.

Algunas de las bibliotecas para juegos más habituales son:

SDL es una biblioteca muy empleada para juegos en 2D, emuladores y software que necesite un acceso directo al hardware. Su primera versión es de 1998. Está creada en C, por lo que es directamente utilizable desde C y C++, pero existen “bindings” para otros lenguajes como C# y Python. **Pygame** es una biblioteca para Python que se apoya en SDL.

Allegro cubre el mismo segmento que SDL, es más antigua (de principios de los 90) pero se sigue manteniendo. Eso sí, su cuota de mercado es menor. También está creada en C y tiene “bindings” para otros lenguajes.

XNA, de Microsoft, era un conjunto de herramientas para facilitar la creación de juegos para Windows y XBox usando C#. Cuando Microsoft abandonó su desarrollo, surgieron alternativas open source para seguir empleando ese modelo, como **MonoGame** y FNA.

OpenGL es una biblioteca gráfica de muy bajo nivel, orientada principalmente a 3D (aunque también se puede usar en 2D), y relativamente poco amigable.

LibGDX es una biblioteca creada (principalmente) en Java, inicialmente para desarrollos en Android, pero que también permite crear juegos de escritorio. Comenzó en 2014.

Cocos2d es una biblioteca de código abierto, creada inicialmente para Python pero de la que luego se han hecho muchos “forks”, para ObjectiveC (iPhone), C++, Ruby, Java, C#, JavaScript...

Kivy es para Python, de código abierto, pensada principalmente para dispositivos móviles, aunque también permite desarrollar software de escritorio.

Y como ejemplos de motores y herramientas:

Unity3D es un motor de juegos multiplataforma 3D y 2D, propietario, creado en C++ pero que usa C# como lenguaje de script. Su origen es del año 2005 y se permite su uso libre para usuarios/empresas que generen menos de 100.000 dólares al año, debiéndose pagar a partir de ese punto. Será la herramienta en la que nos centraremos.

Unreal Engine es un motor de juegos creado inicialmente por Epic Games para el juego FPS Unreal (1998) y evolucionado posteriormente. Está creado en C++ y orientado sobre todo a la creación de juegos 3D, aunque también se puede emplear para 2D y, por supuesto, para otras temáticas de juego distinto de los “shooters” en primera persona. Permite generar ejecutables para multitud de plataformas. Su código libre es descargable y su modelo de negocio se basa en royalties generados por los programas que se creen usándolo.

Godot Engine es un motor de juegos multiplataforma 3D y 2D, de código abierto, creado en C++ pero que usa su propio lenguaje (GDScript) o bien, opcionalmente, C#. Incluye su propio entorno de desarrollo.

idTech es el nombre genérico que se da a una serie de motores creados por idSoftware para sus juegos Doom, Quake y secuelas posteriores. Las primeras versiones de estos motores son de código abierto (y se pueden descargar desde la cuenta de GitHub de idSoftware), pero las más recientes no.

Panda3D es un motor de código abierto, creado por Disney y la Carnegie Mellon University, desarrollado en C++ pero que usa Python como lenguaje de script. En general, resulta mucho menos amigable que otros como Unity.

CoronaSDK era otro producto propietario, de código cerrado, pero que se podía usar “gratis” en la mayoría de sus funcionalidades, si bien algunas avanzadas eran de pago. Recientemente (mayo de 2020) se disolvió la empresa que lo mantenía (Corona Labs Inc) y está en el proceso de convertirse a una aplicación totalmente de código abierto, bajo el nombre Solar2D. Usa LUA como lenguaje de script y permite generar aplicaciones de escritorio y móviles.

Construct permite crear juegos en HTML5 (lo que supone que también se pueden convertir a “aplicaciones híbridas” para hacerlos funcionar en Android e iOS, o incluso en escritorio), y usa una aproximación “visual”, que permite indicar los eventos a los que debe responder el programa, en vez de teclear órdenes. La primera versión era de código abierto, pero la actual es comercial.

GameMaker Studio también es un producto propietario, que usa su propio lenguaje de script (bastante parecido a LUA o BASIC) y que permite exportar para múltiples plataformas. La versión gratuita sólo funciona durante 30 días, y a partir de ese momento se debe pagar según las plataformas de destino (por ejemplo, actualmente son \$39 al año o \$99 de por vida para Windows y Mac, \$149 de por vida para HTML5, \$149 de por vida para Android...).

Pero hay muchísimos más, tanto de código abierto como comerciales... Puedes ver una lista mucho más exhaustiva en:

https://en.wikipedia.org/wiki/List_of_game_engines

y otra un poco más compacta en:

https://en.wikipedia.org/wiki/LibGDX#External_links

1.2. ¿Por qué Unity?

Unity es uno de los motores de juegos más empleados en la actualidad. Permite crear tanto juegos (o aplicaciones multimedia) 2D como 3D, amateur o profesionales, y su curva de aprendizaje es menos pronunciada que en otros motores. Se usa C# como lenguaje de scripting y se puede emplear el propio Visual Studio como entorno de desarrollo.

En este tema vamos a ver algunas de las funcionalidades básicas de Unity, y crearemos un juego básico de “matar marcianos” en 2D. Posteriormente crearemos un juego básico en 3D, uno más avanzado en 2D y otro más avanzado en 3D.

1.3. Instalación de Unity

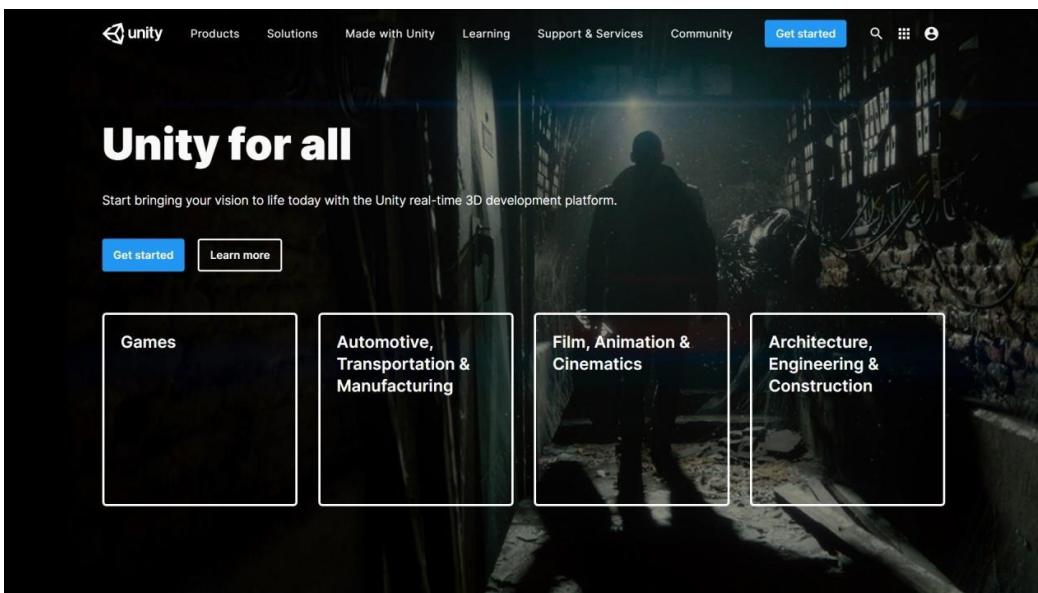
(Nota, vamos a ver los pasos que es necesario dar para instalar Unity a fecha agosto de 2020; pueden cambiar en fechas posteriores, pero es esperable que las diferencias no sean grandes).

Primero deberemos dirigirnos a su página oficial, unity.com, que debería mostrar una apariencia parecida a la siguiente:



<https://unity.com/es>

En ocasiones, o según la configuración regional, quizá la página no aparezca en español sino en inglés:



En ella tenemos aparece un botón “Comencemos” (o “Get Started”) que nos lleva a elegir entre tres versiones:

The image shows two side-by-side screenshots of the Unity Store's 'Plans and pricing' section. Both are titled 'Planes y precios' and 'Plans and pricing'. They both mention 'We offer a range of plans for all levels of expertise and industries. All plans are royalty-free.' Below this, there are two tabs: 'Individual' and 'Business'. The 'Individual' tab is selected in the left screenshot, while the 'Business' tab is selected in the right screenshot.

Left Screenshot (Individual Tab):

- Plus:** \$399 per year. Includes: Plan inicio, presupuesto.
- Pro (LO MÁS POPULAR):** \$1,800 per year. Includes: Plan inicio, presupuesto.
- Empresa:** \$200 per year. Includes: Plan inicio, presupuesto, intercambios.

Right Screenshot (Business Tab):

- Plus:** \$399 per year. Includes: Plan inicio, presupuesto.
- Pro (RECOMENDADO):** \$1,800 per year. Includes: Plan inicio, presupuesto.
- Enterprise:** \$200 per year. Includes: Plan inicio, presupuesto.

Both sections include a 'Subscribe' button and a 'Conocé más' link. The right screenshot also includes a note about royalties: 'Eligibility: If your revenue or funding is greater than \$200K in the last 12 months, you are required to use Pro or Enterprise.'

Aparentemente, esas tres versiones son de pago, pero en la pestaña “Individual” se puede observar que hay una versión “Personal”, que no tiene coste y cuya licencia nos permite usarla mientras nuestros ingresos no superen los 100.000 dólares anuales. Esa será la razonable cuando uno empieza. Desde hace poco, existe también una versión “Student”, que permite alguna característica adicional, como soporte de desarrollo en equipo, pero que sólo será utilizable mientras se sea estudiante (y habrá que acreditarlo):

The image shows two side-by-side screenshots of the Unity Store's 'Plans and pricing' section. The left screenshot is for 'Individual' users and the right is for 'Business' users.

Left Screenshot (Individual Tab):

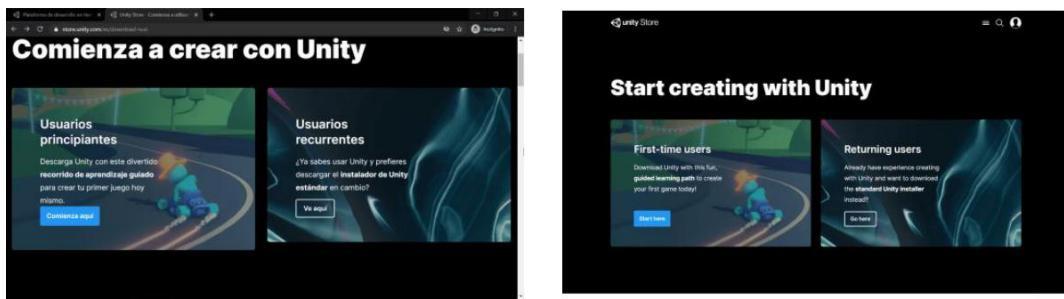
- Estudiante:** Gratis. Includes: Aprende a usar las herramientas y los flujos de trabajo que utilizan los profesionales.
- Personal:** Comienza a crear con la versión gratuita de Unity. Includes: Gratis.
- Learn Premium:** Dominá Unity con las sesiones en vivo dirigidas por expertos y con el aprendizaje bajo demanda. Includes: Comienza a aprender, Conoce más.

Right Screenshot (Business Tab):

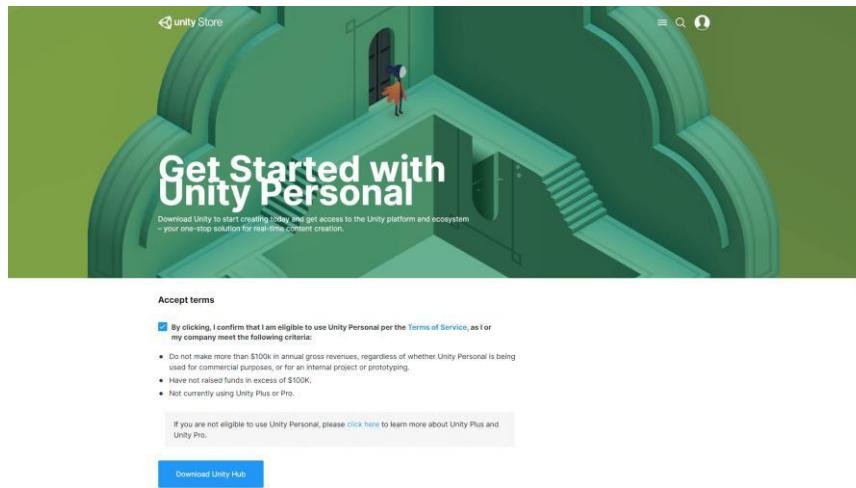
- Student:** Free. Includes: Learn the tools and workflows professionals use on the job.
- Personal:** Free. Includes: Start creating with the free version of Unity.
- Learn Premium:** Master Unity with expert-led live sessions and self-paced learning. Includes: Start learning, Learn more.

Both sections include a 'Sign up' button and a 'Get started' button. The right screenshot also includes a note about royalties: 'Eligibility: Must be a student at an accredited educational institution of legal age to consent to the conclusion and processing of personal data information e.g., age 13 in the US. If in the EU must pass the 13th birthday. Download Pro is not available.'

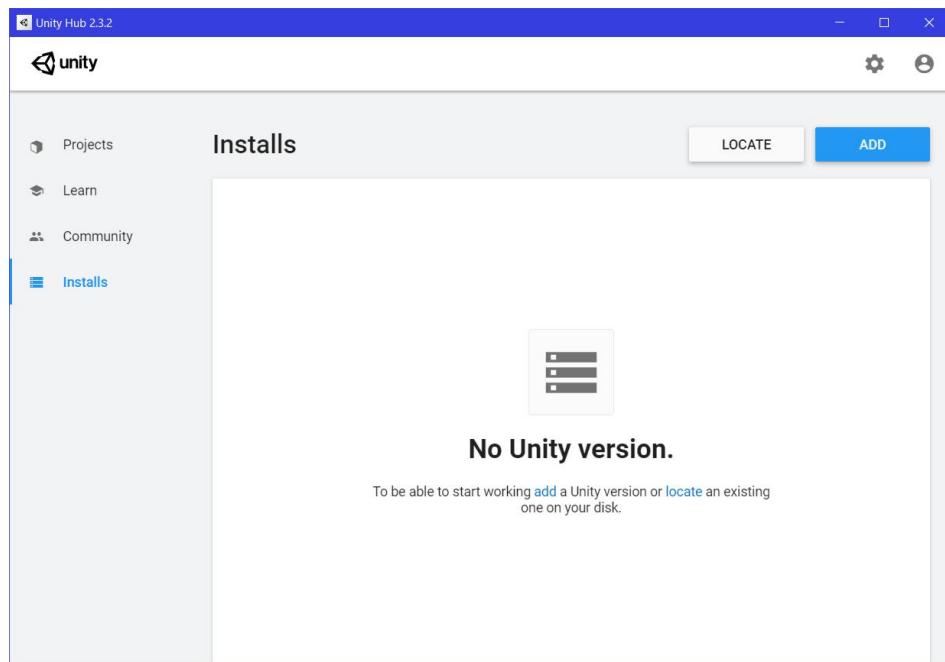
A continuación nos permitirá elegir entre un “camino guiado” para usuarios principiantes (“first-time users”), que incorpora varios mini juegos en los que hay que ir haciendo modificaciones para ayudar a entender los fundamentos básicos, u una “instalación estándar” para usuarios “recurrentes” (“returning users”). Yo optaré por esta última.



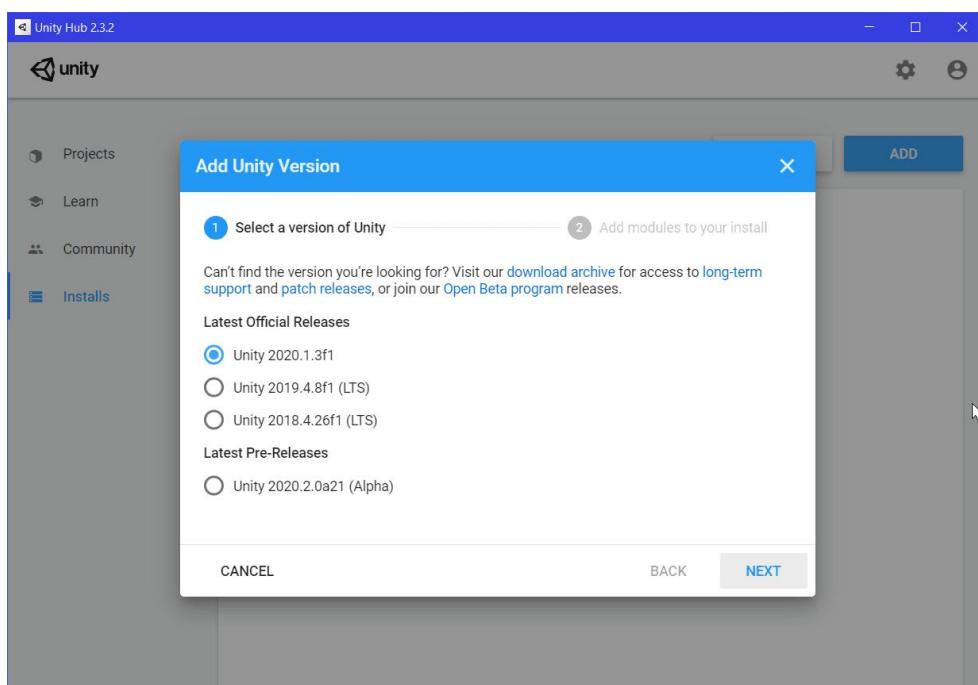
Tras confirmar que somos “elegibles” (básicamente, que nuestros ingresos no superan los 100.000 dólares), nos llevará a la descarga del “Unity Hub” para nuestro sistema operativo (en mi caso, Windows). Éste hace de “pantalla de bienvenida” y que además nos permitirá tener varias versiones del editor de Unity instaladas a la vez:



Tras descargarlo (deberían ser cerca de 50 MB) y lanzarlo, se nos mostrará la lista de versiones de Unity que tenemos instaladas (y que inicialmente no debería ser ninguna):



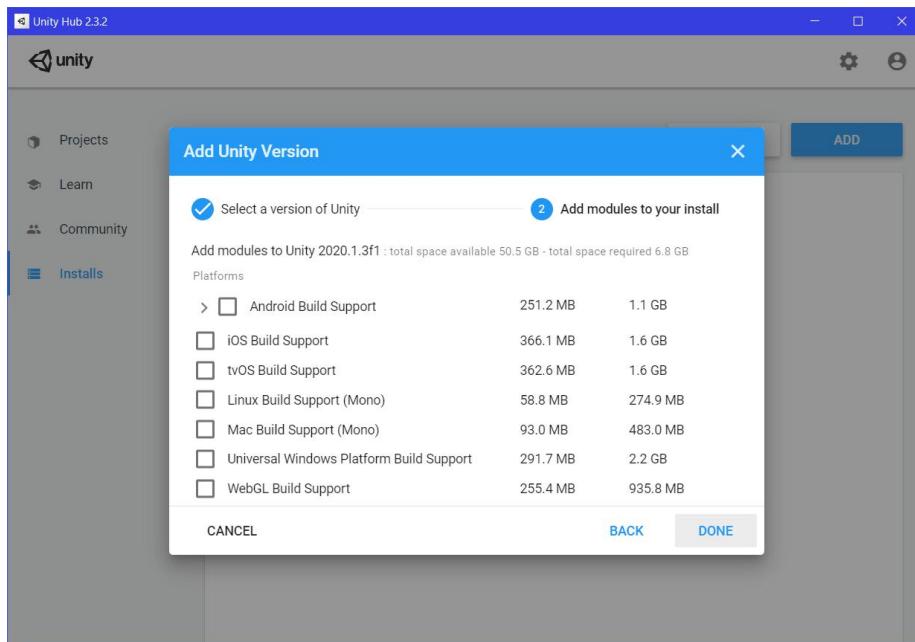
Si pulsamos el botón “Add”, se nos propondrá varias versiones de Unity a instalar. Por ejemplo, en este momento se puede escoger entre la más reciente estable (2020.1), otras ligeramente menos modernas pero con soporte de larga duración (2019.4 LTS y 2018.4 LTS), o bien una versión “pre-release”, ya sean en fase temprana de desarrollo (alpha, como la 2020.2 alpha) o en fase de pruebas finales (beta).



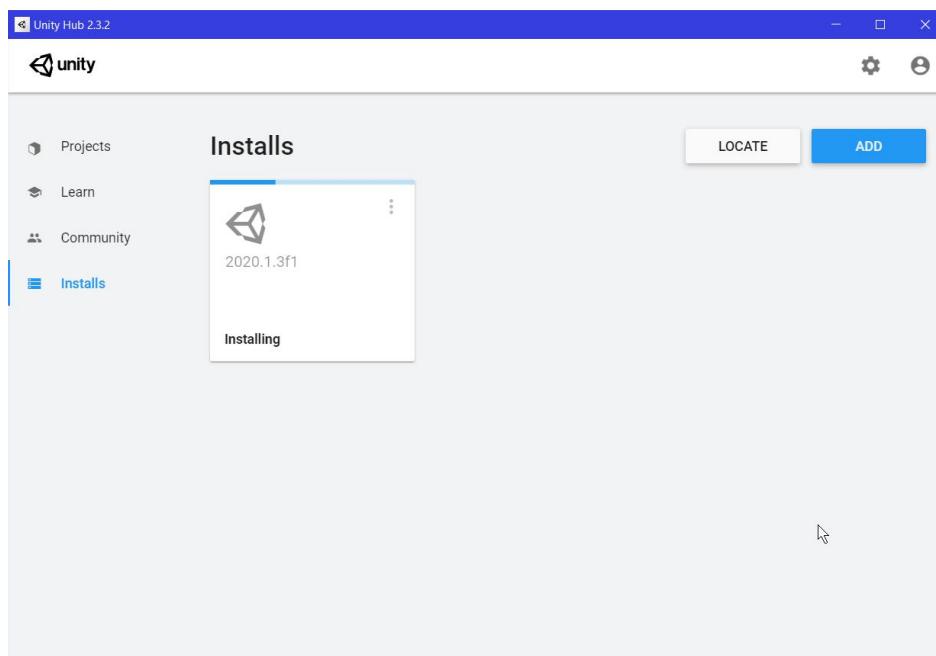
En principio, para aprender, lo más recomendable es esquivar las “pre-release”, que pueden contener algún “bug”, especialmente las “alpha”, e

ir a una versión reciente, para conocer las características que incorporarán las siguientes versiones. Por eso, yo escogeré la 2020.1.

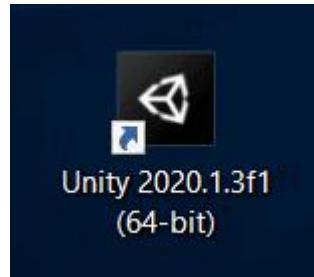
A continuación, se nos avisa de cuánto va a ocupar (6.6 GB) y se nos propone algunos componentes opcionales, como el soporte de destinos Android, Linux o Mac, junto con el tamaño de descarga y el tamaño de instalación que supondrán. Yo no instalaré ningún opcional.



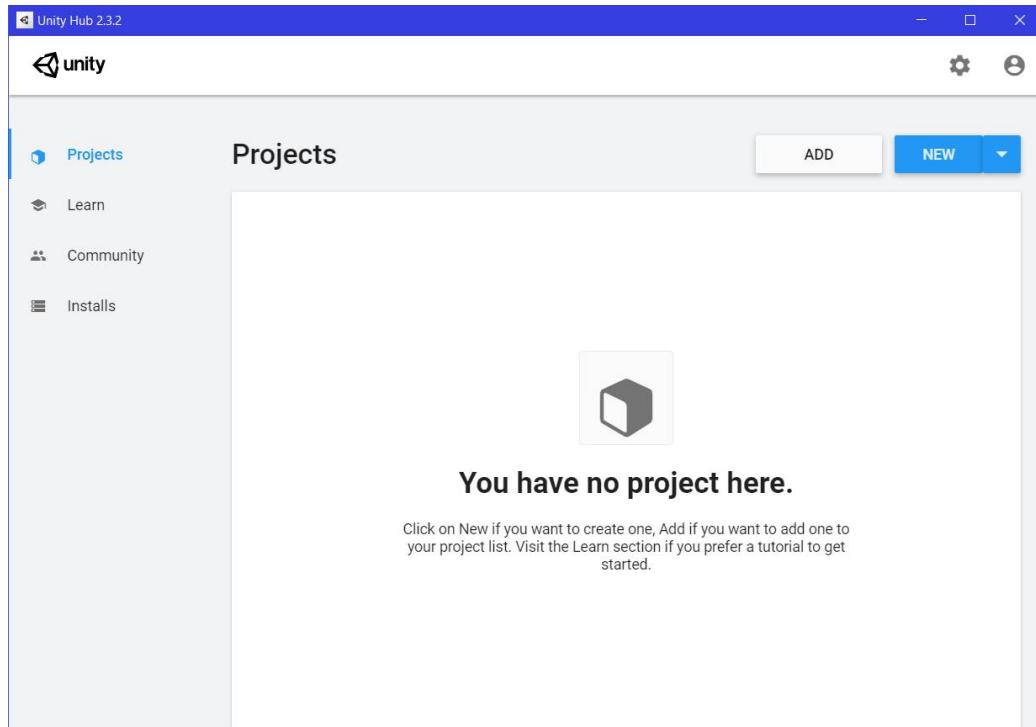
El tiempo que tarde la instalación dependerá de lo rápida que sea nuestra conexión a Internet y nuestro equipo.



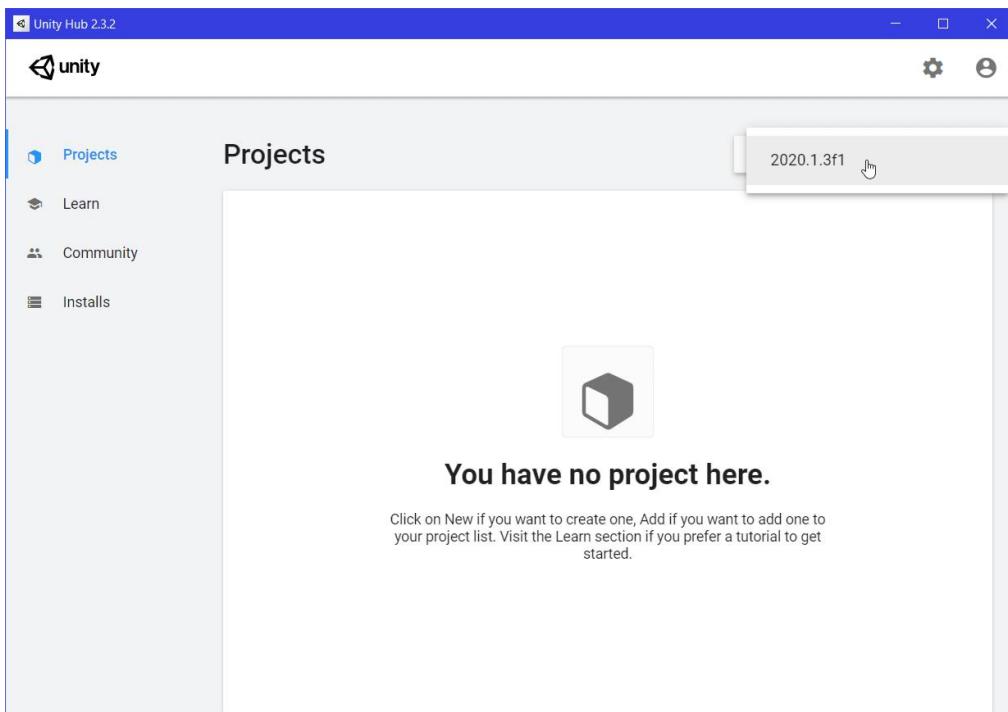
Cuando la instalación se complete, debería aparecer un enlace a Unity en nuestro escritorio:



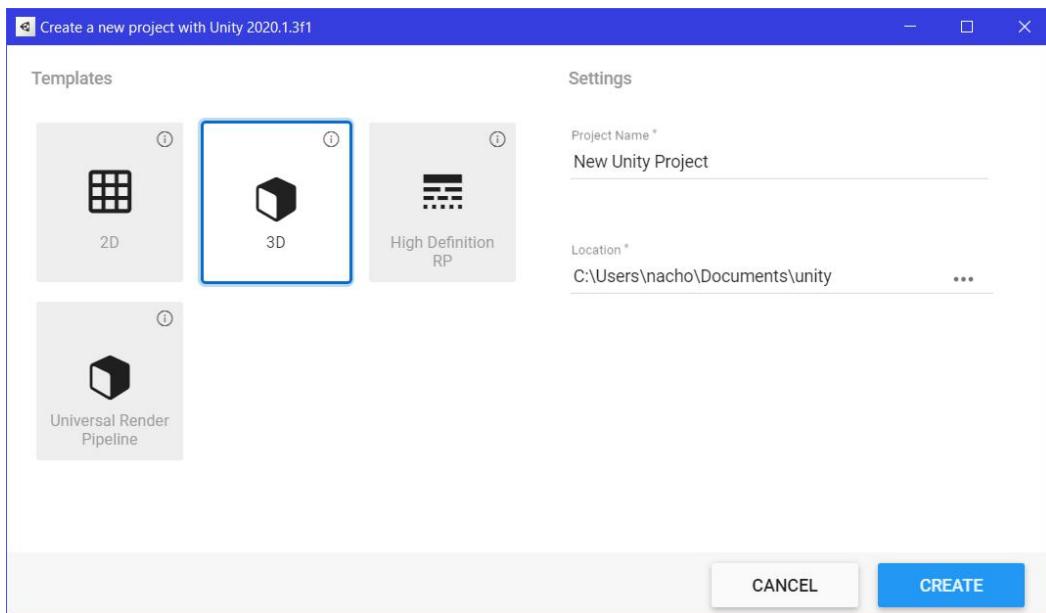
Pero, aun así, la forma más habitual de entrar será desde la pestaña “Projects” del Unity Hub, que nos permitirá relanzar nuestros proyectos anteriores de forma sencilla, así, como añadir un proyecto nuevo si pulsamos el botón “New”:



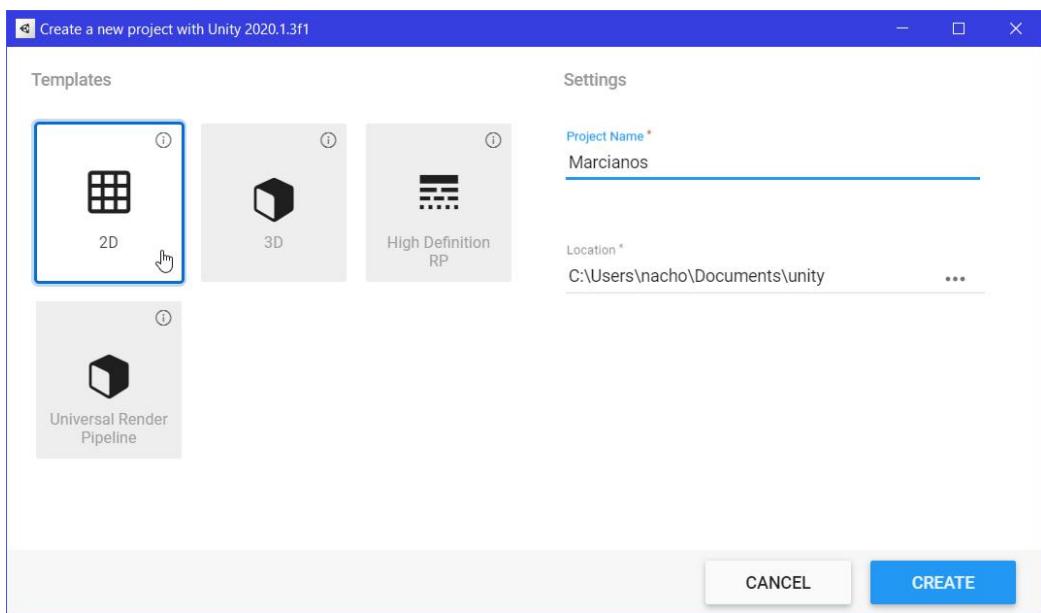
y se nos permitirá elegir cual de las versiones de Unity que tengamos instaladas (por ahora sólo una) queremos usar para ese proyecto:



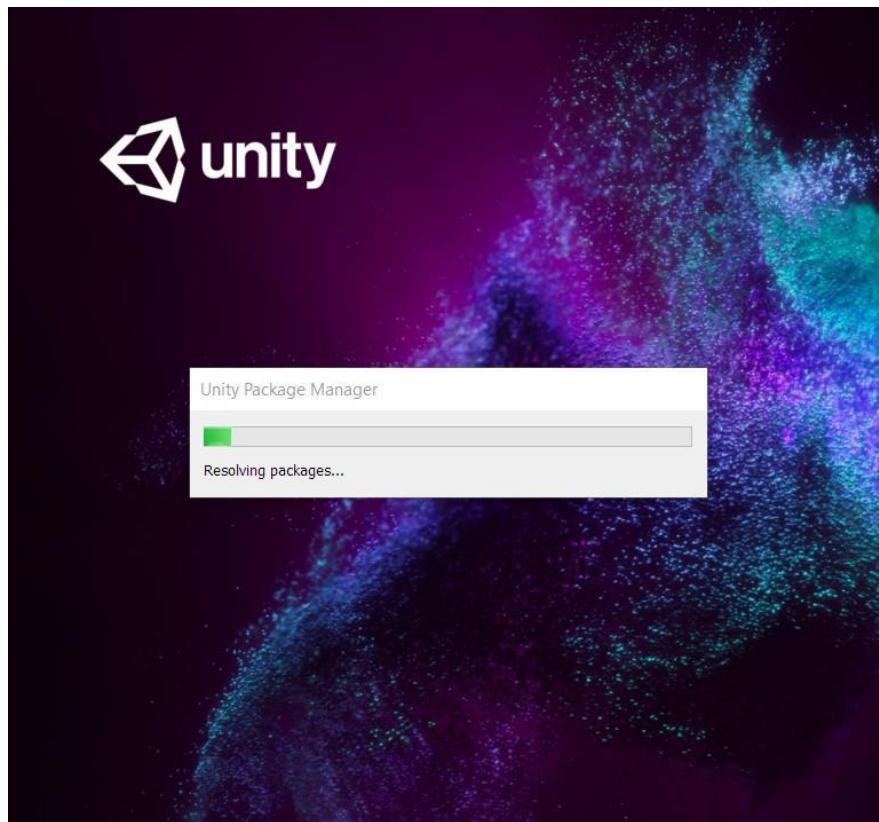
Aparecerá una nueva pantalla en la que se nos preguntará el tipo de proyecto y su nombre.



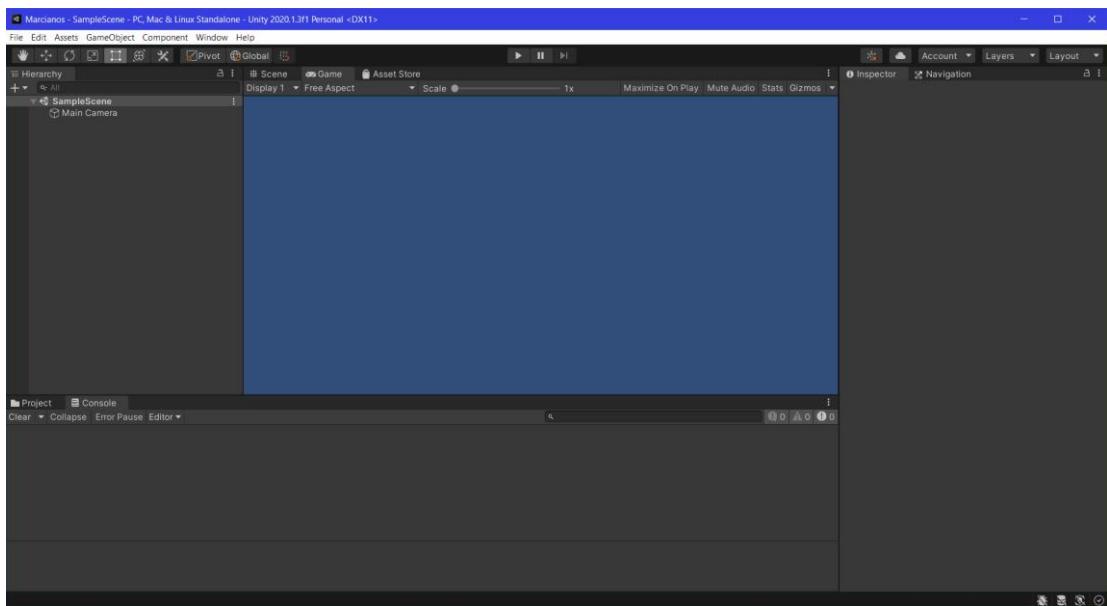
Nuestro primer proyecto será un juego de “matar marcianos” en 2D, así que cambiamos el tipo de proyecto y elegimos un nombre adecuado:



Se preparará nuestro proyecto y quizá el entorno inicial de Unity:



Y accederemos a la que será la vista normal del editor de Unity:

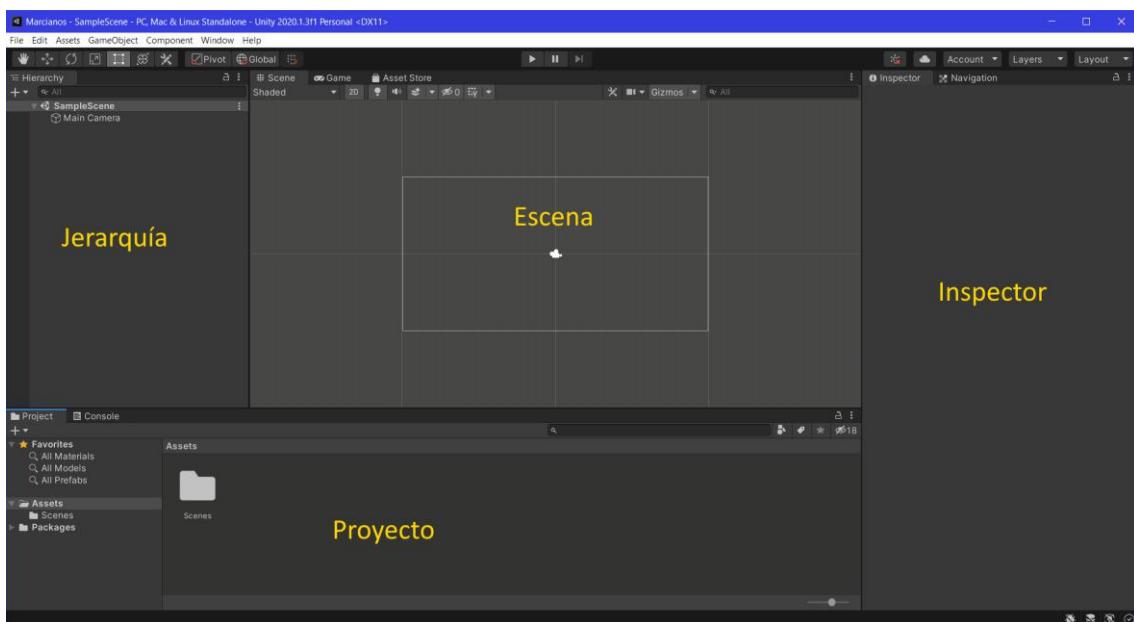


Ejercicio propuesto 1.3.1: Descarga e instala Unity.

1.4. El entorno de Unity

Vamos a comenzar por crear un proyecto 2D. Por eso, ciertas peculiaridades del entorno que están relacionadas con proyectos 3D las comentaremos más adelante.

Las principales zonas de la pantalla (al menos al nivel que nos interesa por ahora) son:



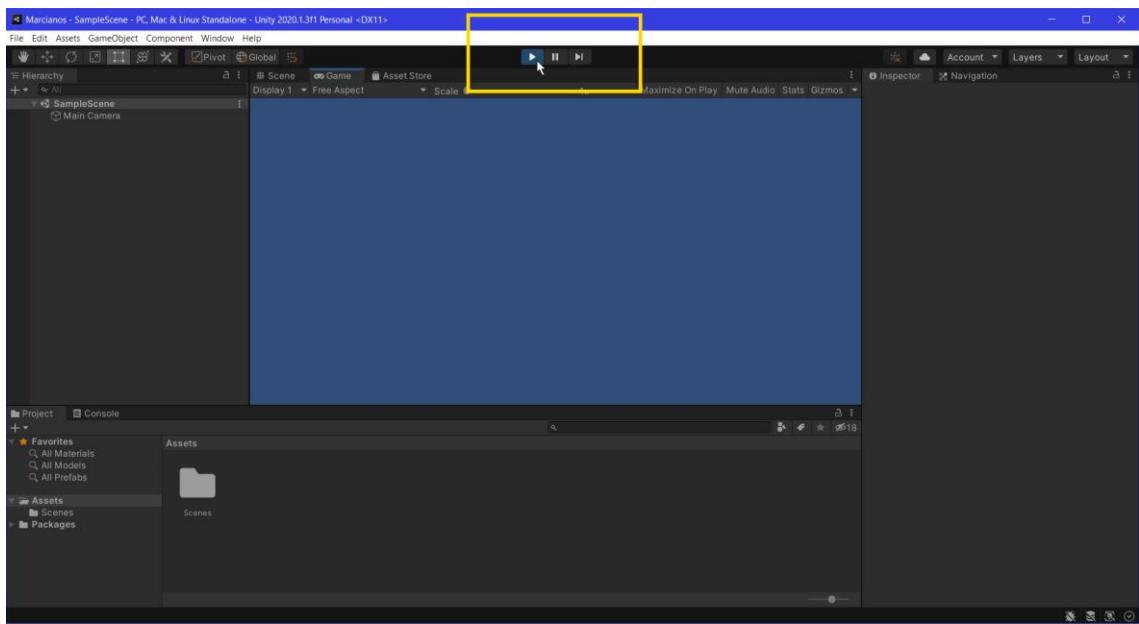
- La zona central superior de la pantalla es la vista de **Escena** (Scene), que muestra lo que será la parte visible de nuestro juego, y sobre la que nos podremos desplazar o hacer zoom para ver más detalles de elementos concretos. Si se nos muestra inicialmente la vista de juego (Game), nos interesará hacer clic en Scene para preparar los elementos desde la vista de Escena.

La parte izquierda es la “jerarquía” (“**Hierarchy**”), en la que aparecerán los distintos objetos que forman nuestro juego, y que por ahora sólo contiene una cámara, que pronto manipularemos.

La parte derecha es el “**Inspector**”, que usaremos para ajustar las propiedades del objeto que tengamos seleccionado.

La parte inferior mostrará los ficheros que forman parte del proyecto (“**Project**”), como imágenes, sonidos, scripts, etc. Esta parte inferior también nos permitirá acceder a la “consola”, que en ocasiones usaremos para depurar. Al igual que ocurría con la vista de Escena y la vista de juego, en este panel inferior tenemos una pestaña para la consola (Console) y otra para el proyecto (Project). En este primer momento, nos resultará más interesante ver el contenido del proyecto:

- La barra superior permite operaciones como mover, rotar y escalar objetos. La usaremos más adelante, a excepción de la zona central, que vamos a usar ya y que incluye un botón “Play” para poner el juego en funcionamiento y para detenerlo, y que por ahora sólo mostrará una pantalla azul:

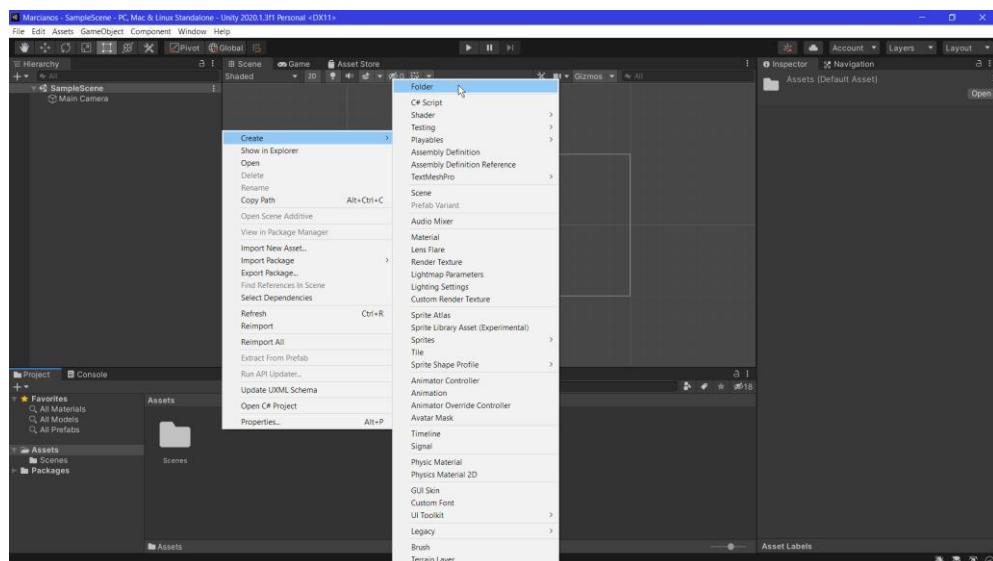


Para salir del modo de ejecución, habrá que volver a pulsar el botón “Play”.

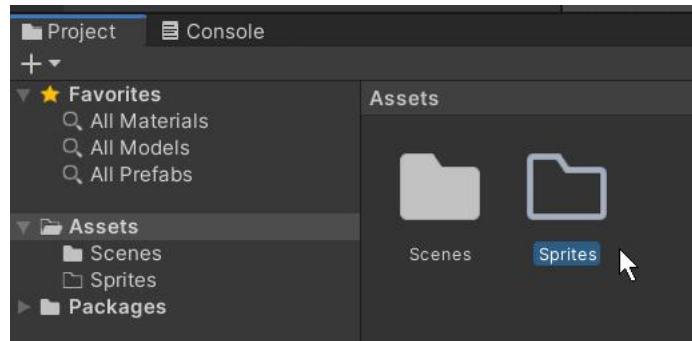
1.5. Mostrando una primera imagen: fondo y cámara

Como vamos a comenzar por imitar un juego retro 2D, el primer paso puede ser añadir una imagen que represente el fondo, y que nos ayudará a encuadrar la cámara.

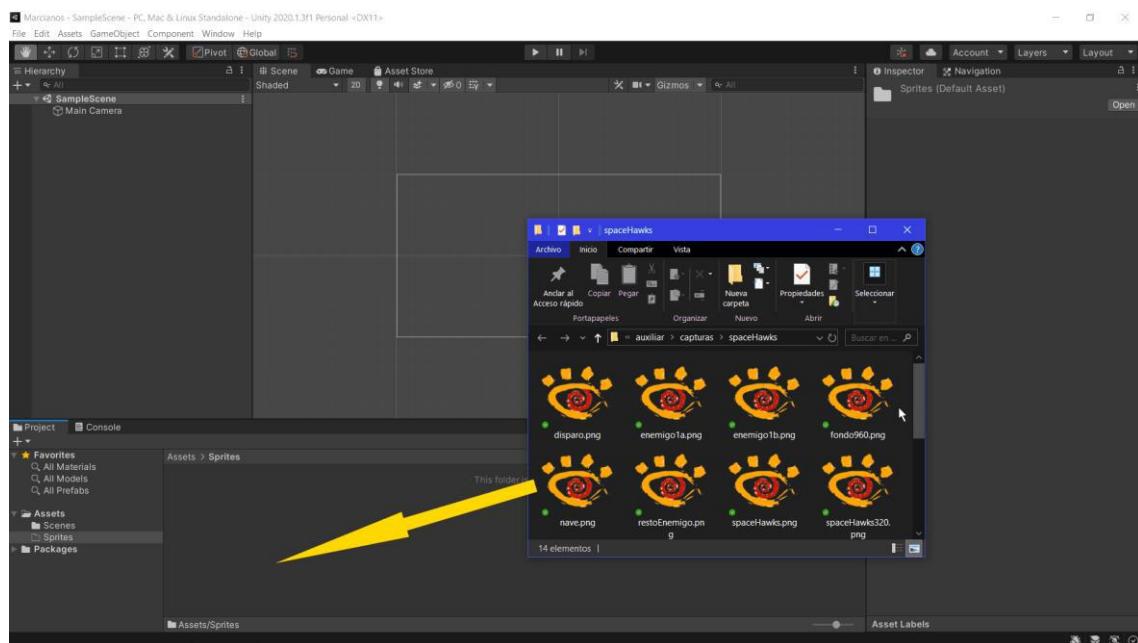
En primer lugar, en nuestros ficheros del proyecto crearemos una carpeta “Sprites”, a la que iremos añadiendo todas las imágenes que vayamos a utilizar en el juego. Lo haremos pulsando el botón derecho en el panel inferior, dentro de “Assets” y escogiendo al opción “Create / Folder”:



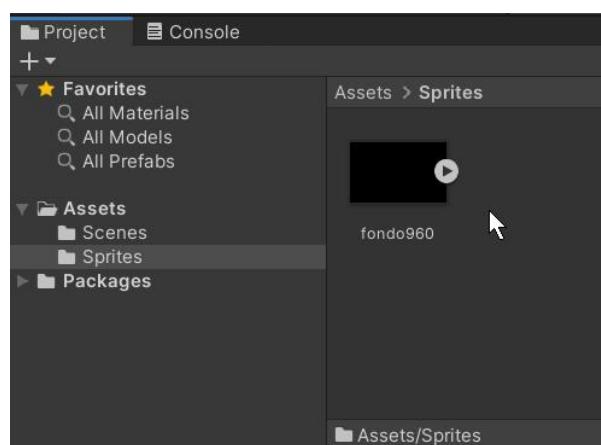
Si damos el nombre “Sprites” a esa carpeta, deberíamos ver algo como:



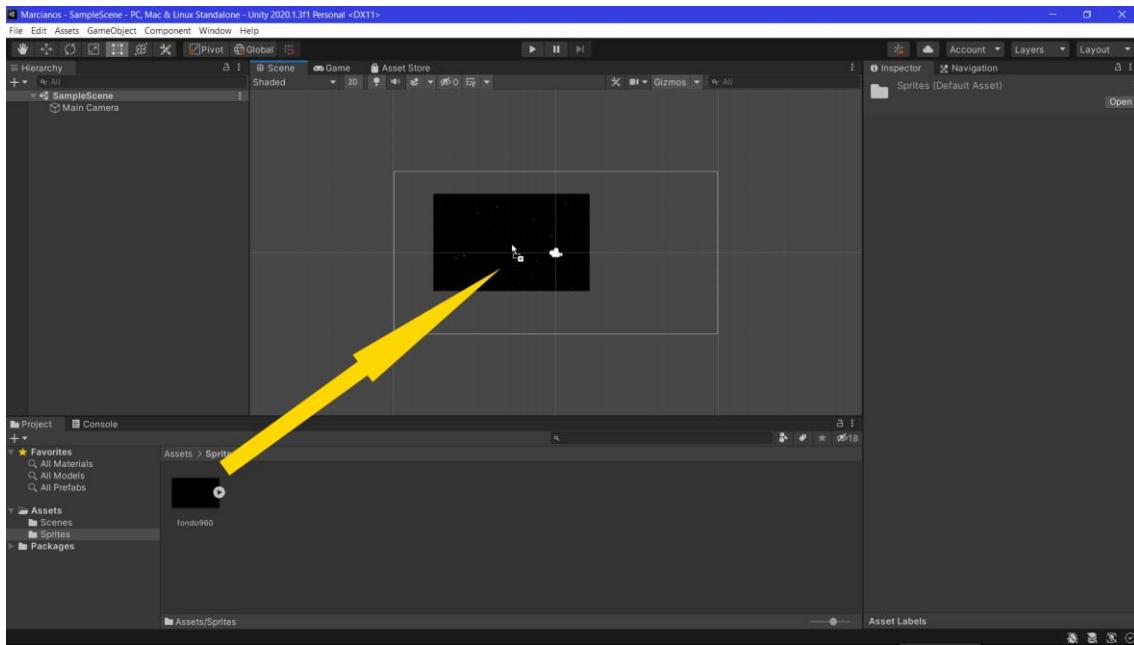
Y ahora sólo tenemos que arrastrar la imagen que nos interese (en mi caso, el “fondo” del juego) desde cualquier carpeta del ordenador hasta esa carpeta “Sprites”, usando, por ejemplo, el propio Explorador de Windows:



Y debería verse su miniatura dentro de dicha carpeta:

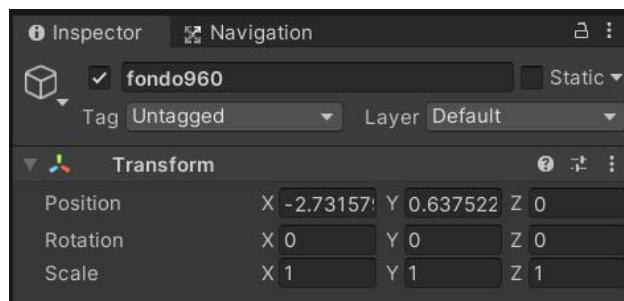


Hay varias formas de crear un “objeto del juego” (GameObject) a partir de esa imagen. La más sencilla es arrastrar la imagen desde el panel inferior a la “escena”. No te preocupes de momento si no queda centrado, lo solucionaremos a continuación:

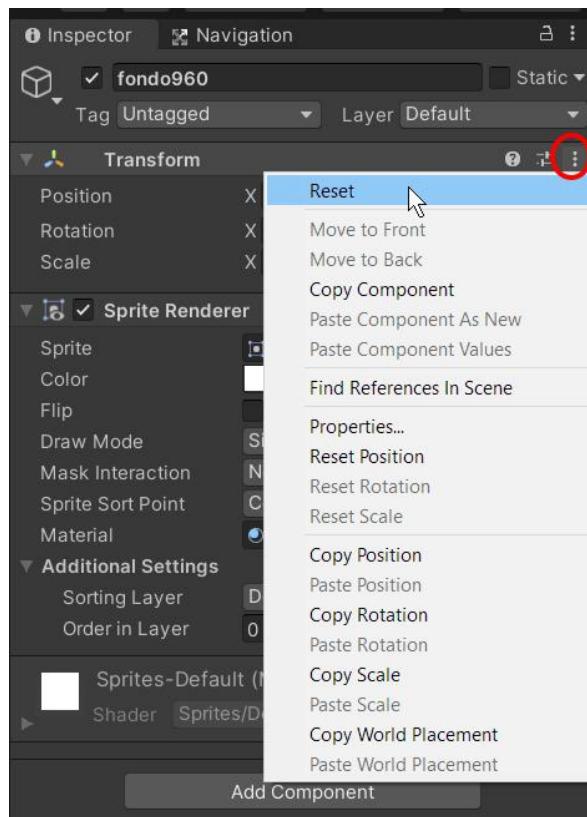


Lo esperable es que la imagen no quede totalmente centrada, pero es fácil de arreglar. Basta con, el “Inspector” (panel derecho) mirar su “transform”, que agrupa su posición, rotación y escala.

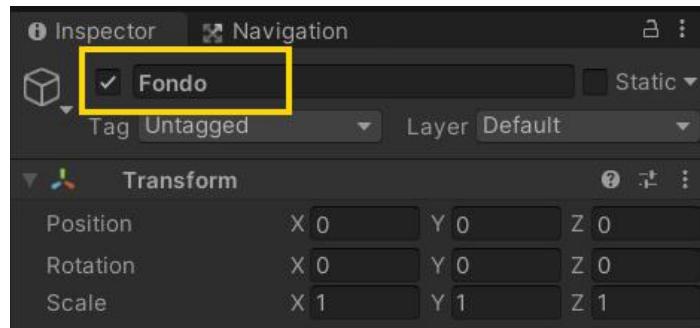
En Unity, el centro de la pantalla corresponde a las coordenadas (0,0), por lo que bastaría con cambiar el valor de X y el de Y en la posición, para hacer que ambos sean cero:



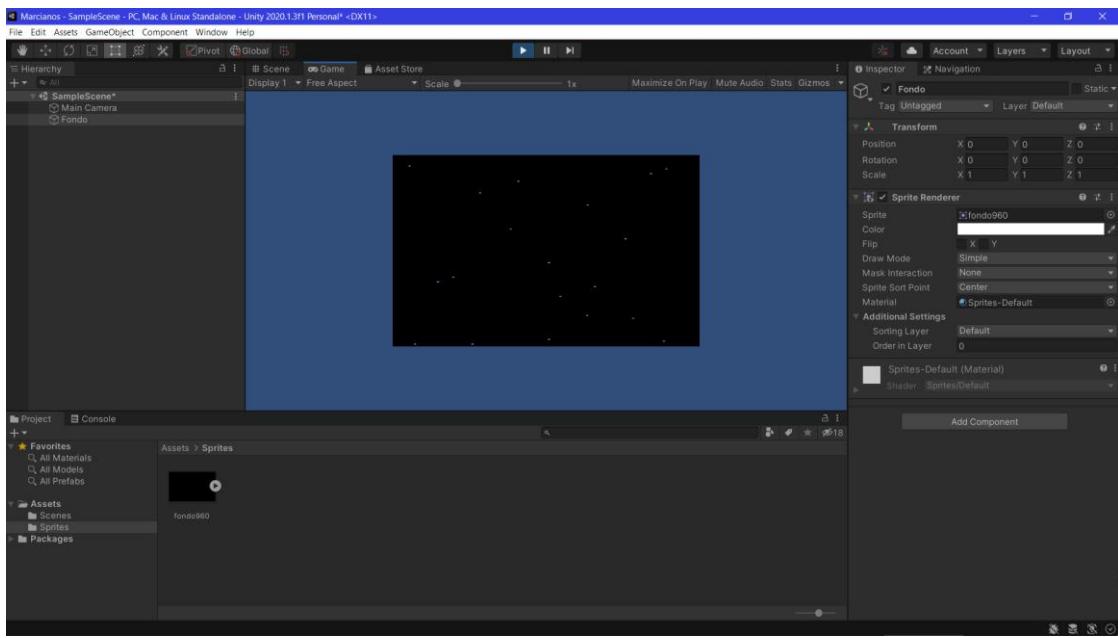
De hecho, existe una manera aún más rápida: junto al componente “Transform” aparece un engranaje, que despliega un menú contextual. La primera opción de ese menú es “Reset”, que deja con valores por defecto (o todos ellos) tanto la posición como la rotación y la escala:



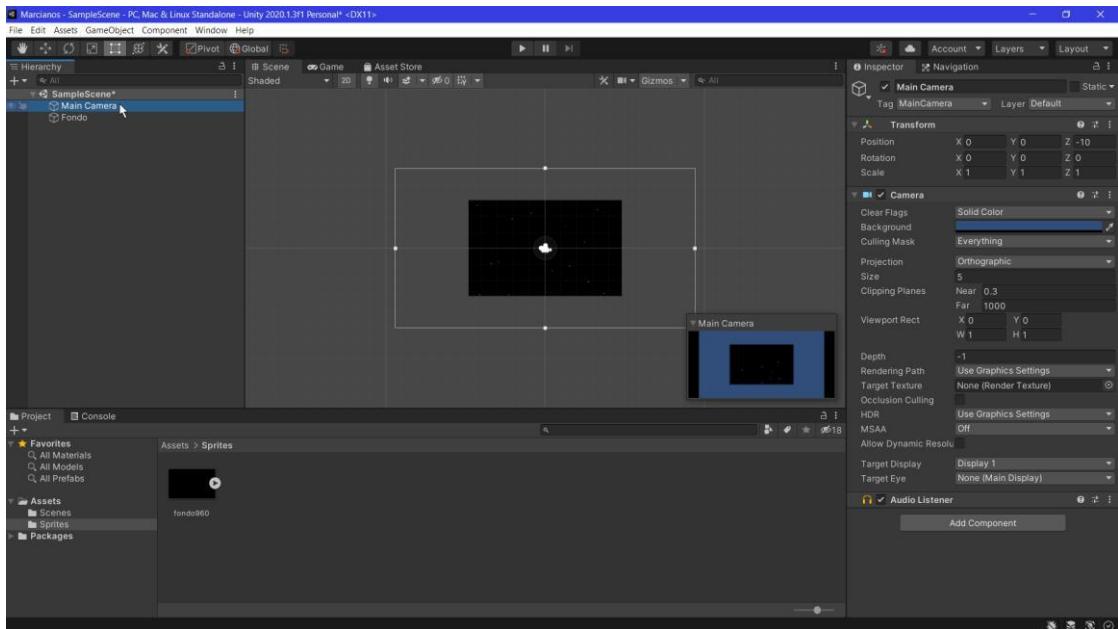
Antes de seguir adelante, suele ser recomendable usar nombres tan intuitivos como nos sea posible. En nuestro caso, podemos renombrar ese “fondo960” para que pase a llamarse “Fondo”. Lo conseguimos desde la primera casilla del “Inspector”:



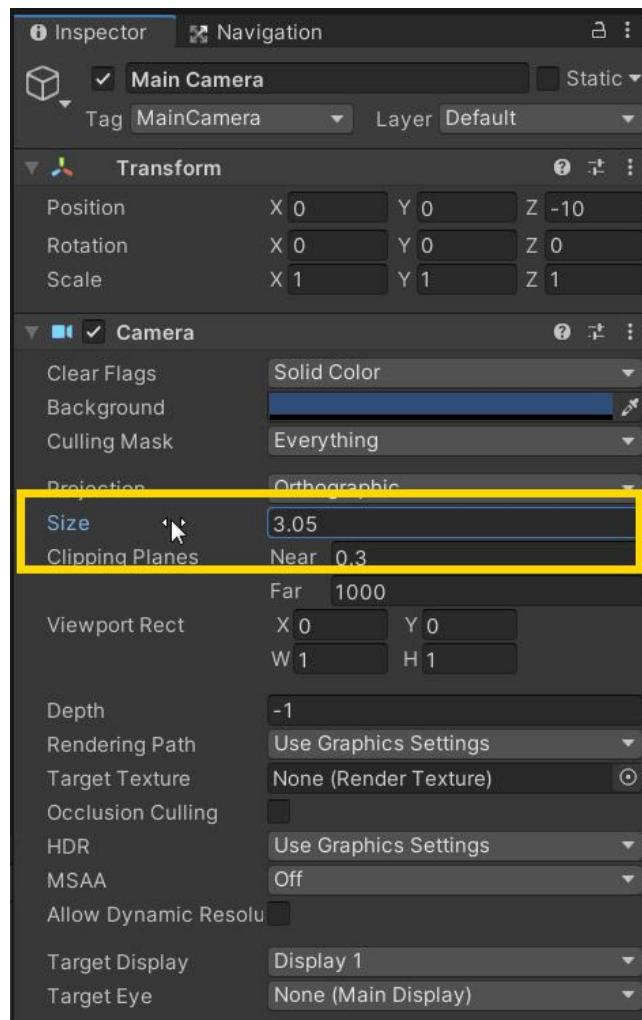
Si ahora volvemos a lanzar el juego, se verá nuestra imagen de fondo, pero todavía con demasiado borde azul alrededor:



Para cambiar ese comportamiento, hacemos clic en la Cámara (**“Main Camera”**), dentro de la jerarquía, y el Inspector nos mostrará sus propiedades:

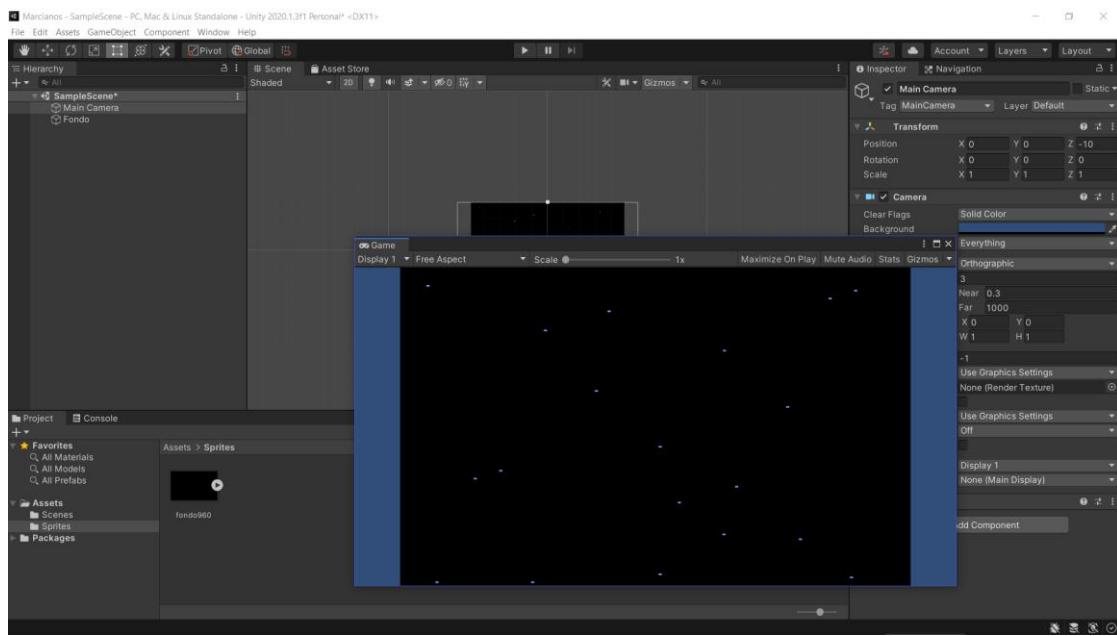


Para que la cámara encuadre sólo nuestro fondo, podemos cambiar su Tamaño (**“Size”**). No es necesario ir tecleando distintos valores, existe una forma más rápida: hacer clic con el ratón en la palabra “Size” y arrastrar hacia derecha o izquierda, con lo que el valor irá cambiando gradualmente:

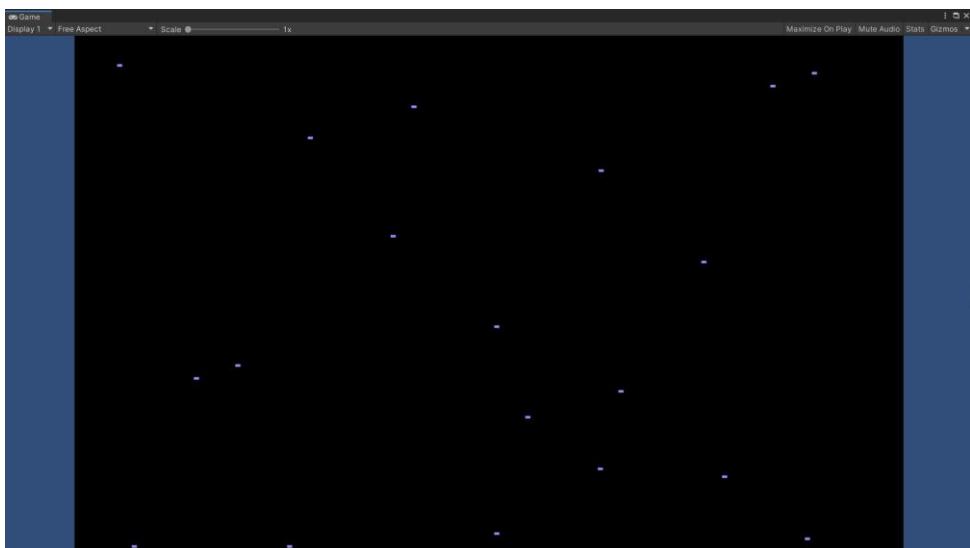


Ya sólo quedará margen por los lados, debido a que es un juego antiguo, creado para un formato de pantalla 4:3, cuando lo habitual actualmente son las pantallas en formato 16:9, más anchas.

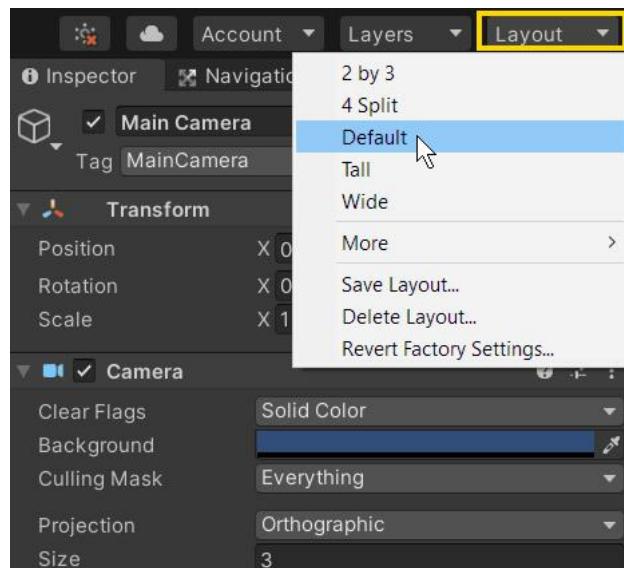
Al pulsar “Play” para ver cómo está quedando el juego, si no queremos que la ventana “**Game**” se vea tan pequeña, podemos arrastrar desde su pestaña para “desanclarla”:



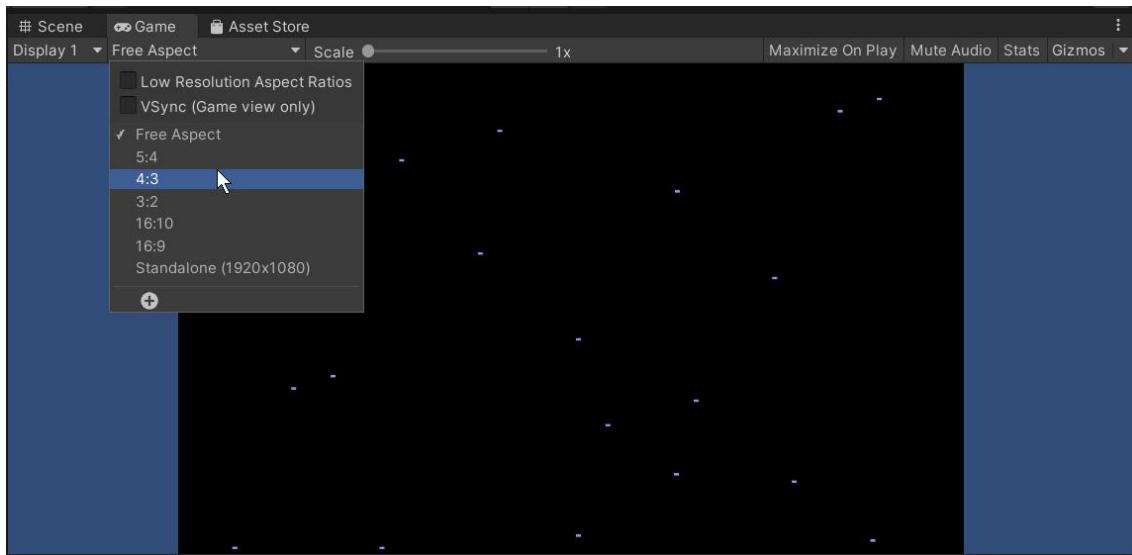
Y eso nos permitiría incluso maximizarla:



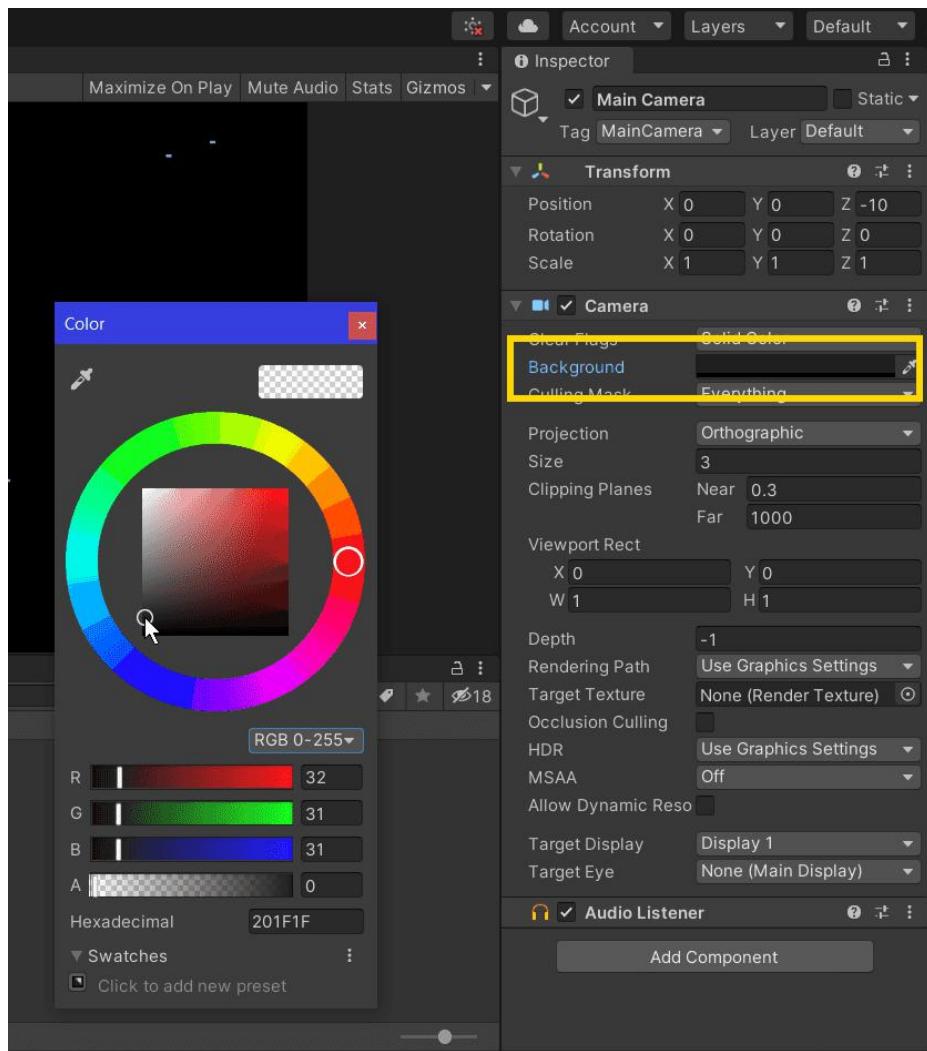
El “desanclar” ventanas puede acabar haciendo que no encontremos las cosas. En ese caso, siempre podemos volver a la **configuración de ventanas por defecto**, en el desplegable “Layout” (de la esquina superior derecha), escogiendo “Default”:



Si queremos ver cómo quedaría en la pantalla de ciertos grupos de usuarios, cuyas pantalla tuvieran distinta relación de aspecto, podemos hacerlo porque la ventana de juego tiene un desplegable para elegir el factor de forma, que podríamos cambiar de “**Free Aspect**” a “4:3” o cualquier otro formato:



Podemos cambiar el **color de fondo** para que no sea azul sino negro, o al menos un gris casi tan oscuro como el de nuestra imagen de fondo, de modo que se note que no es una zona jugable, pero que el borde tampoco resulte tan llamativo:



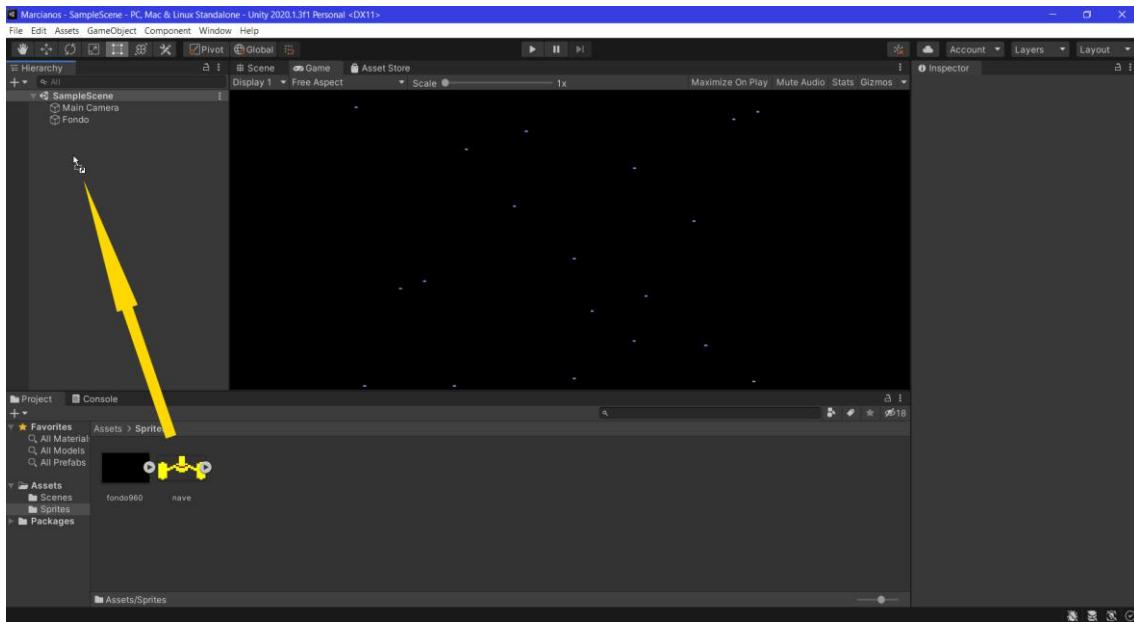
Cuidado: Unity permite hacer cambios durante el modo de ejecución, para probar sus efectos en el juego... pero esos cambios realizados cuando el juego está en marcha **se pierden** al detener la ejecución. Si estás ampliando o modificando el programa (por ejemplo, para cambiar esas coordenadas a 0,0), recuerda detener la ejecución antes, o puede ocurrir que pierdas parte del trabajo.

Ejercicio propuesto 1.5.1: Elige y/o prepara una imagen de fondo para un juego similar a éste (“matamarcianos” o parecido, sin 3D ni gravedad).

1.6. Añadiendo una segunda imagen

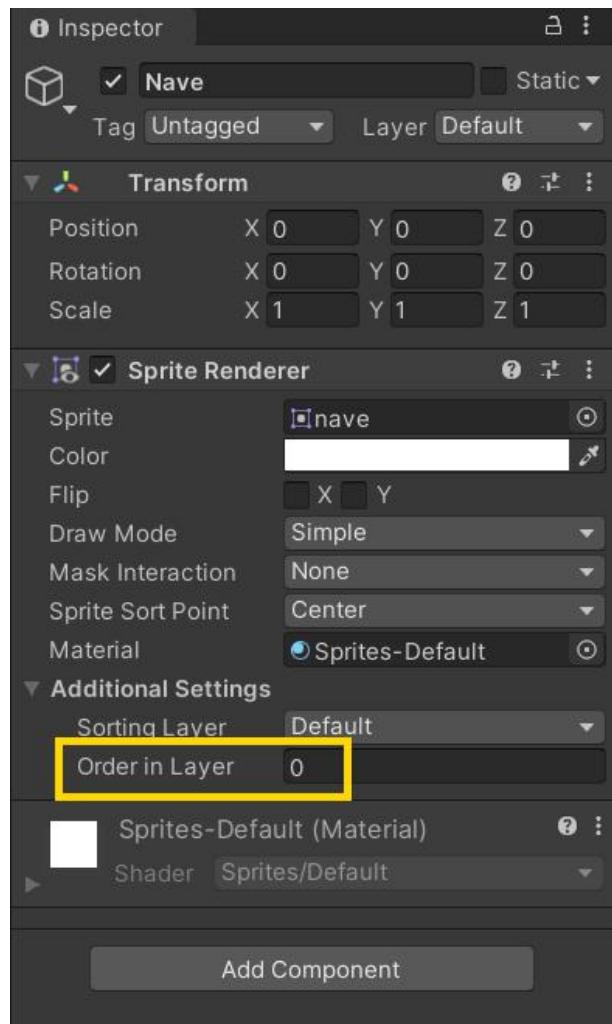
Para añadir una nave, al igual que hicimos con el fondo, podemos arrastrar su imagen a nuestra carpeta “Sprites” del proyecto, y luego de

ahí a la escena o, en este caso, por practicar una forma alternativa, arrastrarla a la jerarquía (panel izquierdo):

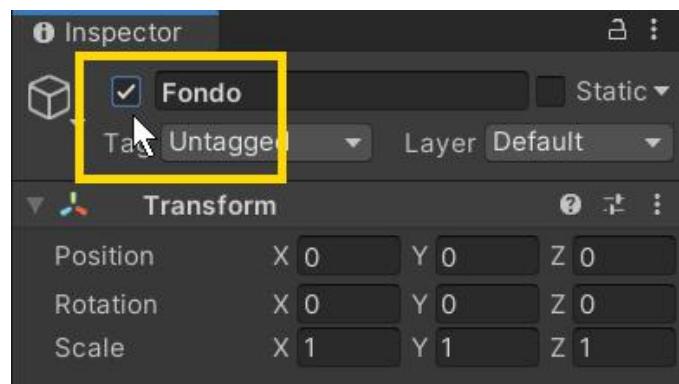


Esta alternativa tiene el inconveniente de que es “menos visual” si queremos que el elemento aparezca una posición concreta de la pantalla. Por el contrario, si queremos que esté centrado, resulta más útil, porque aparecerá directamente en las coordenadas (0,0).

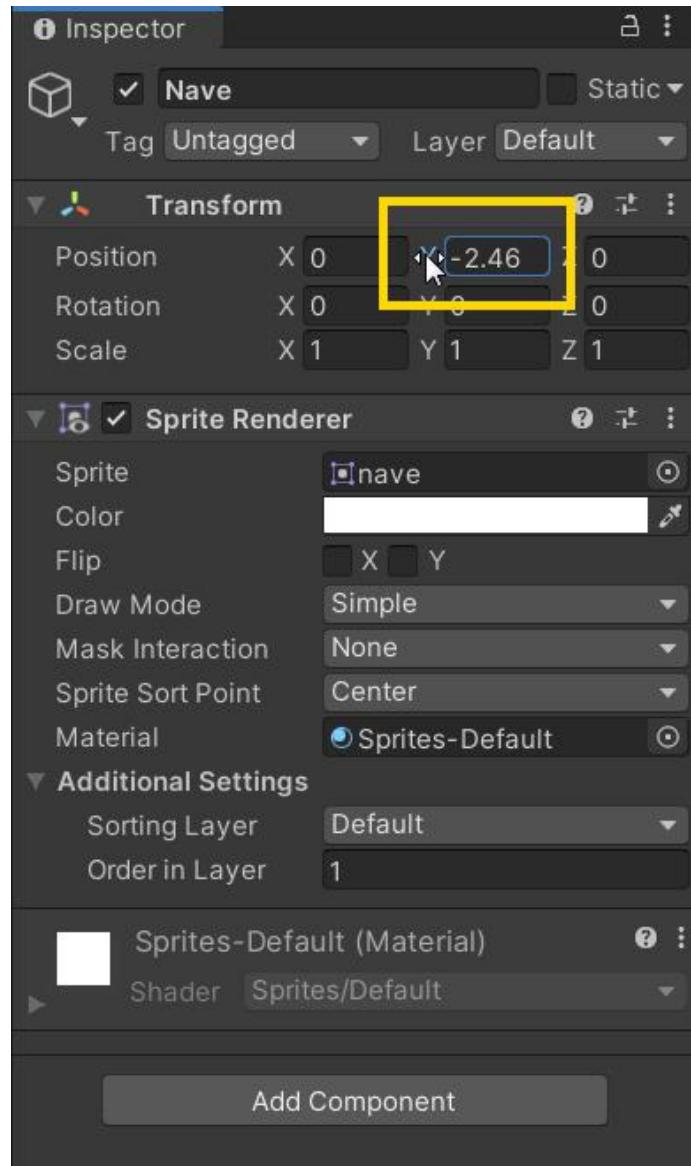
Ahora mismo, ambos objetos (fondo y nave) son visibles, pero se solapan. Hay dos formas de conseguir evitarlo. La más elegante, que veremos más adelante, sería crear varias “capas” (“Layers”). De momento, una solución muy rápida, y que es útil si tenemos pocos objetos, es indicar el “Orden dentro de la capa” (“Order in Layer”). Si a la nave le indicamos un orden de 1 pasará a mostrarse por delante del fondo, que tenía un orden de 0:



Si no ves la nave y dudas de si está ahí, una forma rápida de comprobarlo es **desactivar elementos** de la escena momentáneamente. Por ejemplo, podemos dejar de ver el fondo durante un instante si hacemos clic en el “checkbox” que hay junto a su nombre, en el inspector:

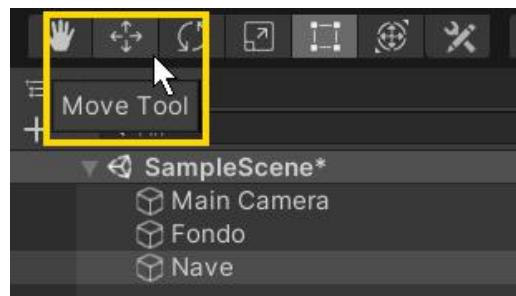


Ahora debemos mover la nave a la que será su posición real durante el juego, cambiando su posición Y en el Inspector. Nuevamente, no lo faremos tecleando valores al azar, sino arrastrando con el ratón sobre la palabra “Y”:

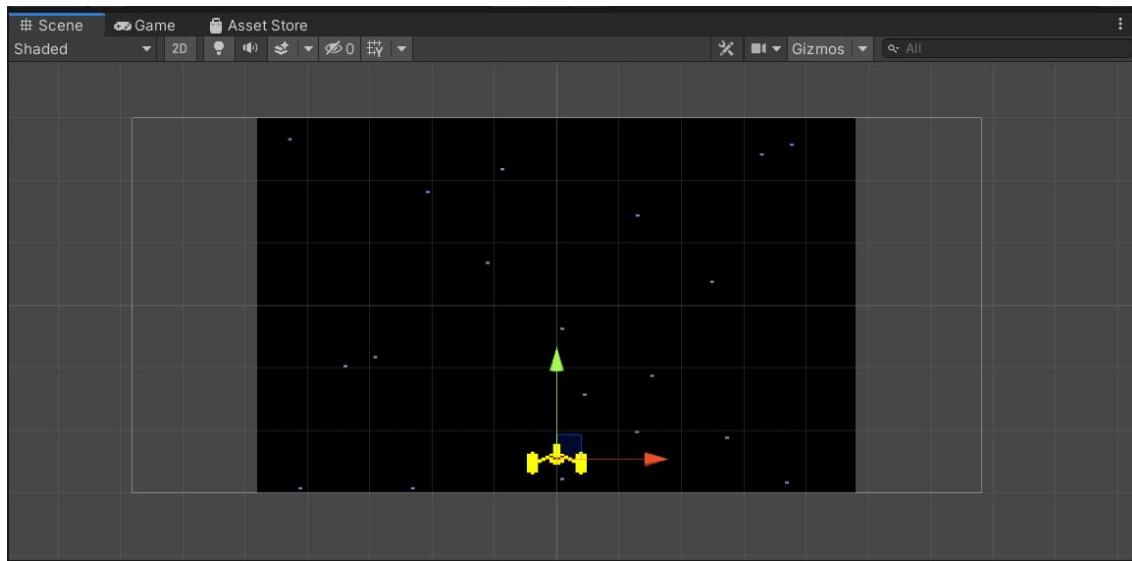


Verás que, como el tamaño de nuestra cámara es 3, el rango de valores de Y va de -3 (parte inferior de la pantalla) hasta 3 (parte superior), así que un valor aceptable para nuestra nave podría ser cercano a -2.5.

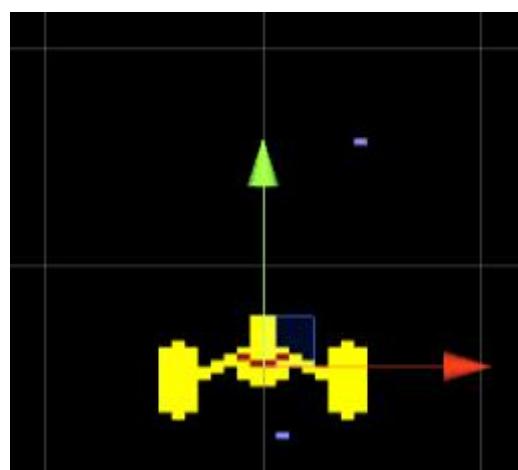
Como alternativa, podemos arrastrar el objeto en la escena para colocarlo “a ojo”, usando la herramienta “Mover”:



Y entonces aparecerán dos flechas sobre la imagen: una flecha roja para mover la nave hacia la derecha o izquierda, y una flecha verde para moverla arriba o abajo:

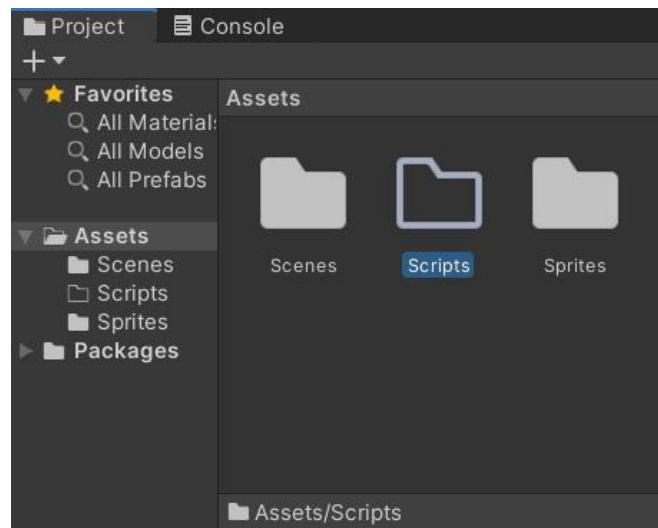


Si se desea mover tanto en horizontal como en vertical, se puede usar el recuadro azul que aparece en la intersección de ambas flechas:

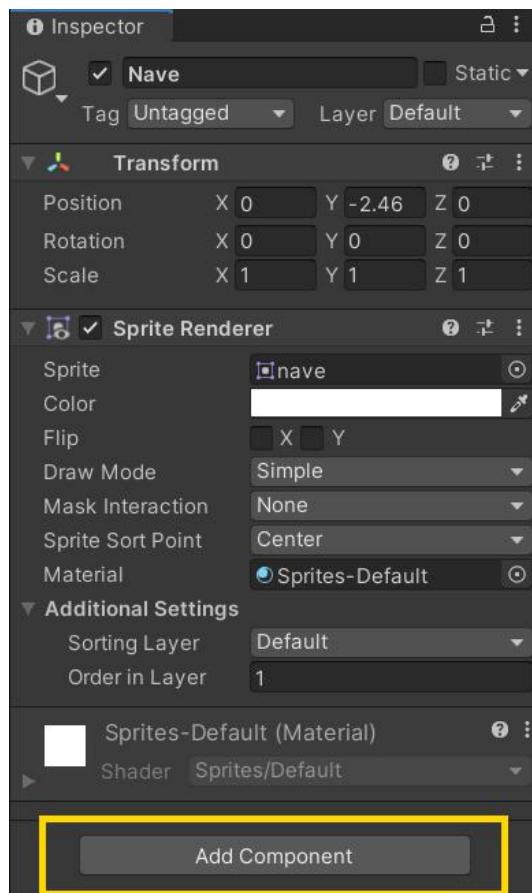


1.7. Moviendo la nave con el teclado

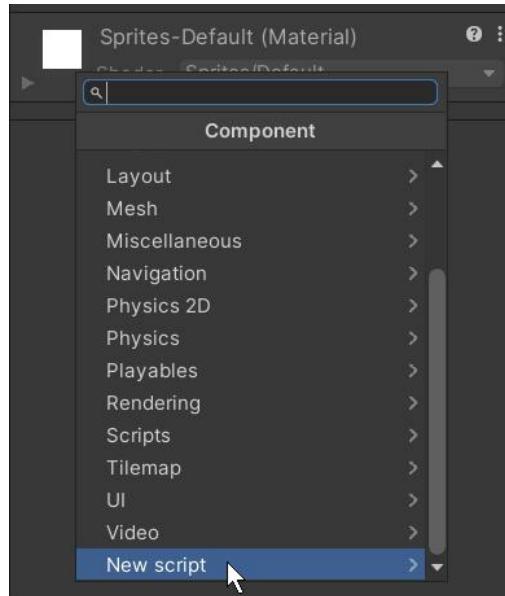
Para añadir “comportamientos” a la nave, deberemos crear un “Script” en C#. Comenzaremos por crear una carpeta “Scripts” dentro de Assets, que vaya a contener todos ellos:



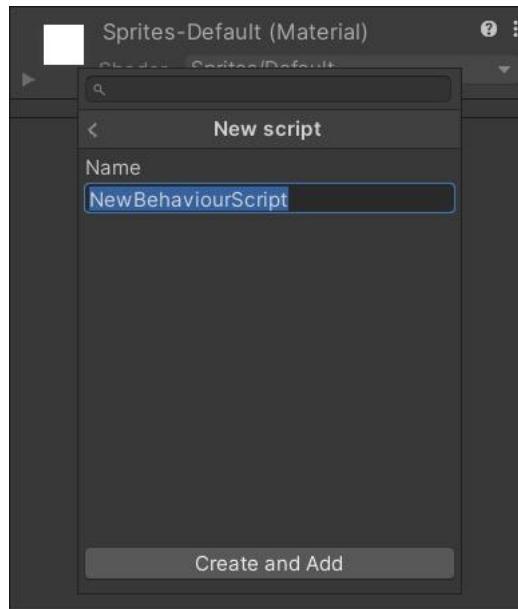
Ahora, seleccionamos la Nave, y encontraremos un botón para “Añadir componentes” (“Add component”) al final del inspector:



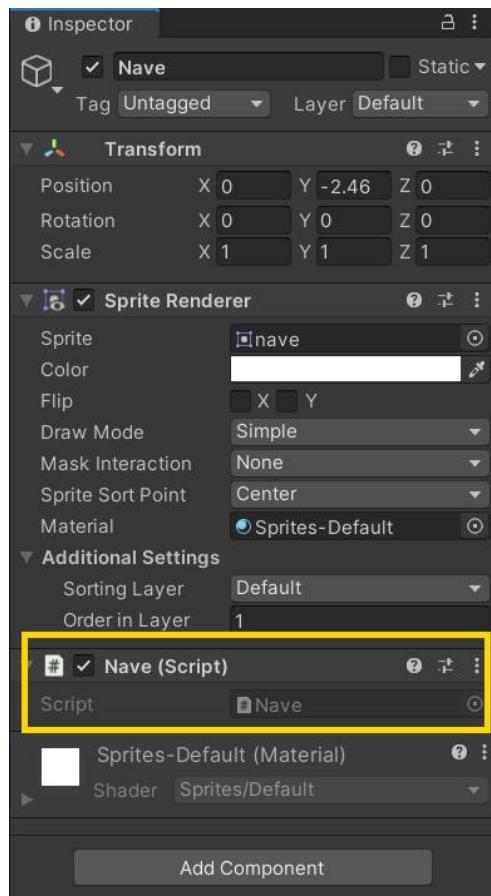
Al final de la lista de componentes que podemos añadir, aparecerá “New script”:



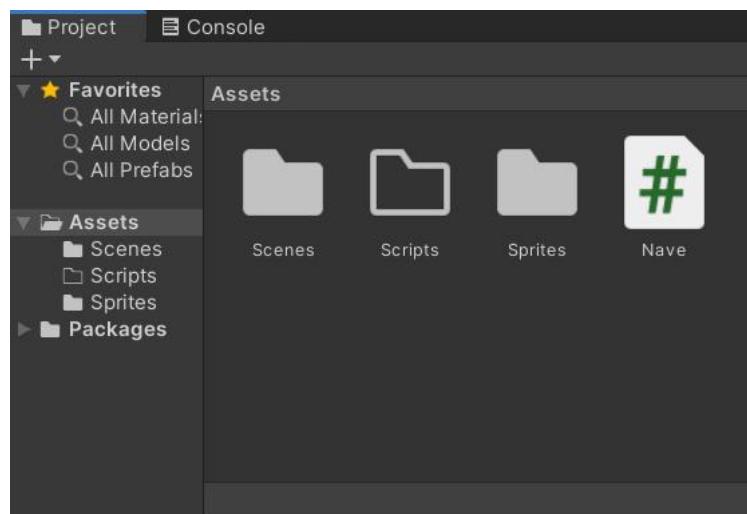
Y se nos propondrá como nombre “NewBehaviourScript”, que nosotros deberemos cambiar por el nombre de nuestro elemento (“Nave”):



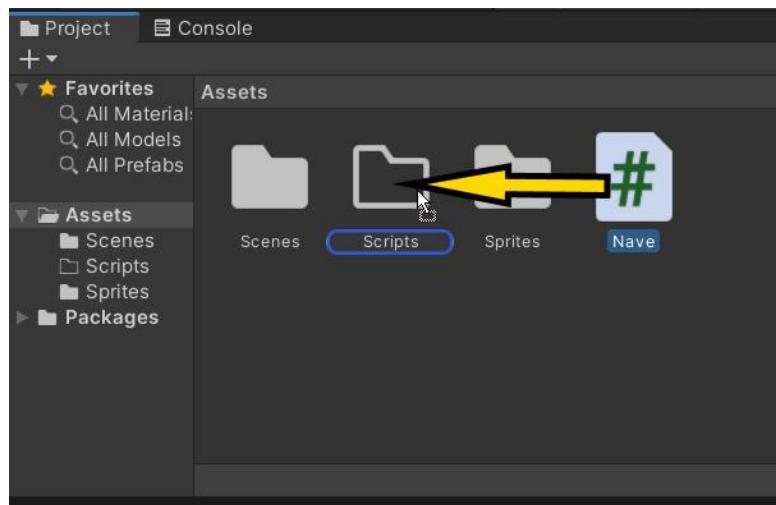
A partir de ese momento, en el inspector se mostrará que la Nave también contiene un Script:



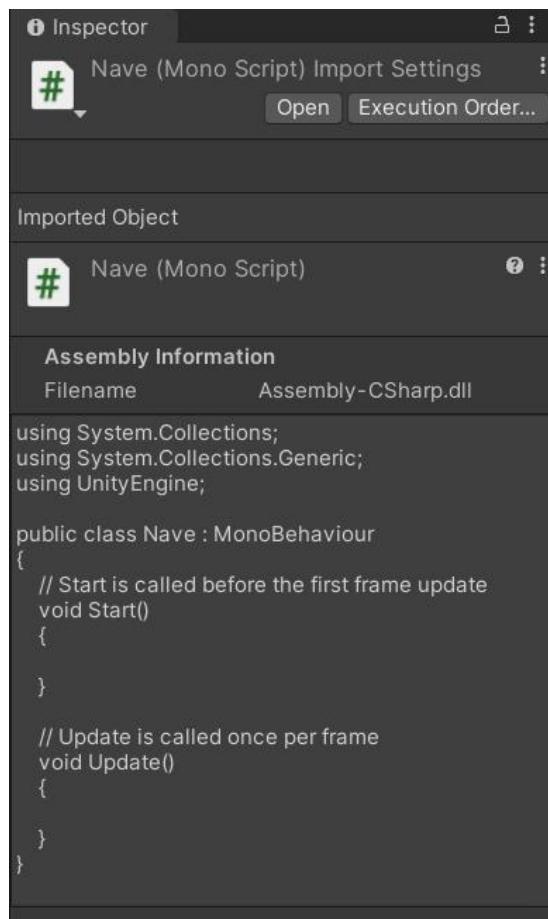
Y ese script aparecerá dentro de la carpeta de Assets de nuestro juego:



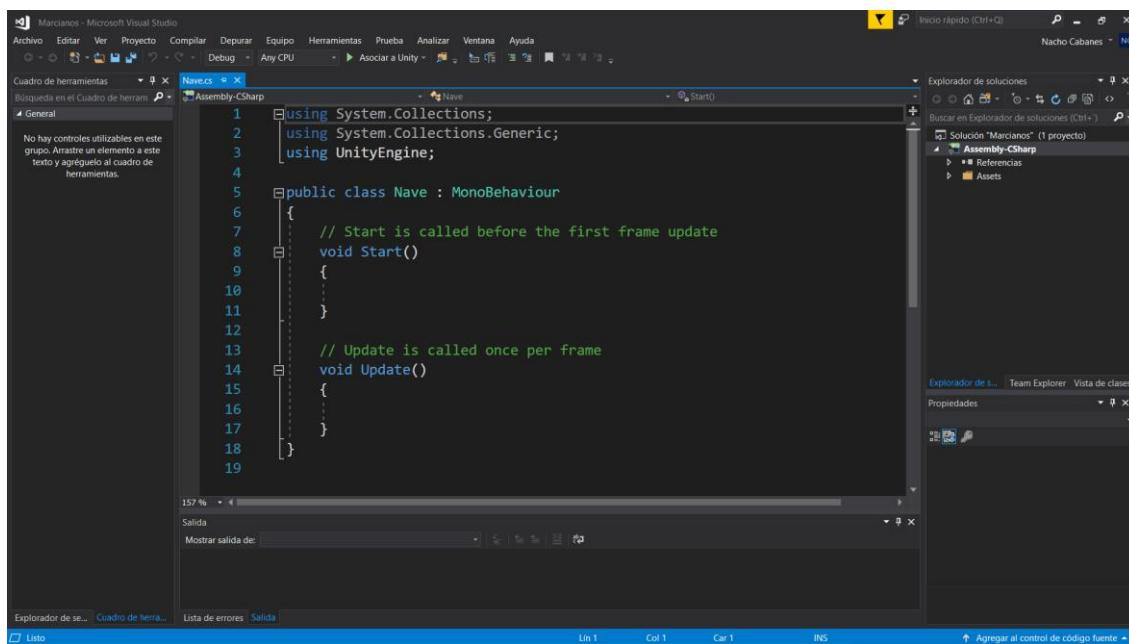
Para tener todo más organizado, podemos moverlo a la carpeta Scripts, usando el panel del proyecto:



Si entramos a esa carpeta y hacemos **un único clic** en el script “Nave” para seleccionarlo, se nos mostrará su contenido en el Inspector, que analizaremos en un instante.



Si hacemos **doble clic** en ese script, puede que se nos pregunte con qué editor lo queremos abrir (y sería aconsejable escoger Visual Studio), o quizás se tome directamente Visual Studio y se abra dentro de este entorno:



Visual Studio nos podrá marcar en tiempo real los posibles errores en el código, aunque quizás queden pequeños detalles de configuración por afinar, como veremos más adelante.

Aparece un esqueleto con dos métodos: “Start” (que se lanza al crear el objeto) y “Update” (que se lanza una vez para cada fotograma del juego):

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Nave : MonoBehaviour
{
    // Start is called before the first frame update void Start()
    {

    }

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

    }
}
```

(Si notas cierta “falta de comunicación” entre Unity y Visual Studio, no te preocupes todavía; lo solucionaremos un poco más adelante)

De cara a mover la nave, podremos mirar teclas individuales, pero una forma más versátil puede ser mirar el desplazamiento horizontal que ha indicado el usuario, comprobando el valor de “Input.GetAxis(

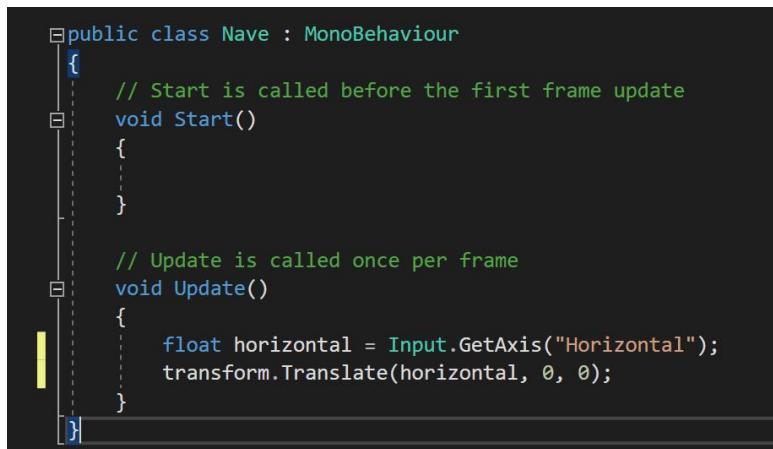
"Horizontal");", que permitirá comprobar con una sola orden tanto teclas como WASD, como flechas del teclado o gamepads (**atención** a las mayúsculas en "Horizontal").

```
float horizontal = Input.GetAxis("Horizontal");
```

Ese valor será negativo si el usuario ha indicado que desea moverse hacia la izquierda y positivo si ha indicado que desea moverse hacia la derecha. En el caso de el teclado, los valores serían -1 o +1 (o 0, si no se ha indicado movimiento), pero en un joystick o gamepad analógico, podría haber valores intermedios (por ejemplo, 0.35). Si no deseamos valores intermedios, sino sólo -1, 0, +1, podríamos usar "Input.GetAxisRaw".

Ahora sólo queda cambiar la posición de la nave (que era parte de su "**transform**", como podíamos ver en el "Inspector") a partir de ese desplazamiento horizontal que ha indicado el usuario. La forma más sencilla es usar "transform.Translate", indicándole el desplazamiento en X (horizontal), en Y (vertical, que no es el caso de este juego) y en Z (que no tendría sentido en un juego 2D):

```
transform.Translate(horizontal, 0, 0);
```



(No es necesario cerrar Visual Studio para probar el resultado: basta con guardar los cambios y volver a la ventana de Unity; en un par de segundos los cambios deberían estar accesibles)

Ese movimiento será demasiado rápido: si el juego se mueve a 60 fotogramas por segundo, y en cada fotograma la nave se mueve una unidad hacia la derecha, se estaría desplazando a 60 unidades por segundo, y nuestra pantalla tiene apenas 5 unidades de ancho, así que la nave abandonaría la pantalla en menos de una décima de segundo.

Se podría hacer que se muevan más lento, por ejemplo multiplicando ese valor “horizontal” por un número menor que uno:

```
transform.Translate(horizontal * 0.1f, 0, 0);
```

pero eso sigue sin ser una buena solución: es un movimiento que depende de la velocidad del equipo en el que se pruebe. Por eso, veremos un par de formas de hacer que ese movimiento sea estable.

Ejercicio propuesto 1.7.1: Elige y/o prepara una imagen de una nave para tu juego. Crea un elemento en tu jerarquía a partir de ella y permite que se mueva de lado a lado.

1.8. Movimiento independiente del equipo

La frecuencia con la que se llama a “Update” no será constante. En unos equipos más potentes se puede llamar con más frecuencia que en otros, e incluso en un mismo equipo puede ocurrir que baje la tasa de fotogramas por segundo en un momento puntual por saturación del sistema. Por eso, Unity ofrece formas de conseguir un movimiento estable.

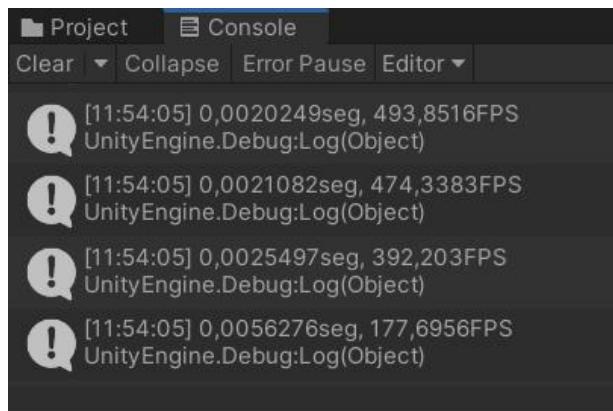
Una de ellas es mirar el valor de “Time.deltaTime”, que es la cantidad de segundos que ha transcurrido desde el anterior fotograma. Por ejemplo, si el juego se moviese a 60 fps, el valor de “Time.deltaTime” sería cercano a 0.01666 segundos.

De hecho, una prueba rápida puede ser mostrar en la consola de Unity el valor de Time.deltaTime (tiempo desde el último fotograma) y quizás también el de 1/Time.deltaTime, que sería la cantidad de fotogramas por segundo:

```
private float velocidad = 2;

void Update()
{
    Debug.Log(Time.deltaTime + "seg, " +
              (1.0f / Time.deltaTime) + "FPS");
}
```

y podemos encontrar cosas como éstas:



Vemos que, en este caso puntual, se llegan a alcanzar casi 500 fotogramas por segundo, pero en momentos puntuales ese valor es muy inferior.

Para utilizar ese valor en nuestros juegos como forma de lograr una velocidad estable, crearemos un atributo para la **velocidad**, medida en “unidades” de pantalla por segundo, y lo multiplicaremos por el tiempo que ha transcurrido en cada fotograma, de modo que si se pasa muchas veces por “Update”, se avanzará pocos píxeles, pero si hay un retardo y se tarda más tiempo en llamar a “Update”, se avanzará puntualmente más píxeles, y la velocidad resultante será la misma, aunque el movimiento en pantalla sea más brusco:

- Si el juego va a 100 fotogramas por segundo, pasarán 10 milisegundos entre cada par de fotogramas. Si el usuario ha pulsado hacia la izquierda, la nave se moverá: $-1 \text{ (valor del eje horizontal)} * 2 \text{ (velocidad)} * 0.010 \text{ (tiempo entre fotogramas)}$. El movimiento en cada fotograma sería -0.020 unidades de pantalla, y al cabo de un segundo se habría movido $-0.020 * 100 = -2$ unidades (hacia la izquierda).
- Si puntualmente el juego va a 20 fotogramas por segundo, pasarán 50 milisegundos entre cada par de fotogramas, de modo que la nave se moverá: $-1 \text{ (valor del eje horizontal)} * 2 \text{ (velocidad)} * 0.050 \text{ (tiempo entre fotogramas)} = -0.100$ unidades de pantalla. Después de un segundo, se habría movido $-0.100 * 20 = -2$ unidades, igual que en el caso anterior, pero el movimiento en pantalla sería más brusco.

Los cambios en el script serían estos:

```
private float velocidad = 2;

void Update()
{
    float horizontal = Input.GetAxis("Horizontal");
```

```
        transform.Translate(horizontal * velocidad * Time.deltaTime, 0, 0);
    }
```

Y el script completo quedaría así:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Nave : MonoBehaviour
{
    private float velocidad = 2;

    / Start is called before the first frame update void Start()
    {
    }

    / Update is called once per frame void Update()
    {
        float horizontal = Input.GetAxis("Horizontal");
        transform.Translate(horizontal * velocidad * Time.deltaTime,
            0, 0);
    }
}
```

No es la única forma de lograr una velocidad estable, más adelante veremos otra forma alternativa.

Ejercicio propuesto 1.8.1: Haz que tu nave se mueva a la misma velocidad en cualquier ordenador.

1.9. Un enemigo que se mueve solo

Los pasos para crear un enemigo que se mueva por sí mismo serán los que ya conocemos:

- Arrastrar su imagen a la carpeta de Sprites
- Crear un objeto a partir de esa imagen y, si es necesario, cambiar su nombre
- Elegir su “altura” dentro de la capa actual (“Order in Layer”), para que quede por encima o debajo de los demás elementos, según sea el efecto visual que deseemos en caso de que se solapen. Por lo general, deberá quedar por encima del fondo.
- Elegir su posición inicial en pantalla
- Crear un script que sume una cierta velocidad a su posición en cada fotograma y que cambie de signo la velocidad cuando se acerque a un borde de la pantalla

- Mover ese script a la carpeta correspondiente.

Si queremos que se mueva en diagonal, de modo que recorra una buena parte de la superficie de la pantalla y que pueda llegar a tocar a nuestra nave (lo que nos permitirá aprender a comprobar colisiones), podríamos preparar una velocidad en X y otra velocidad en Y, sumarlas a la posición del enemigo y cambiar su signo si alcanza algún extremo de la pantalla, así:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

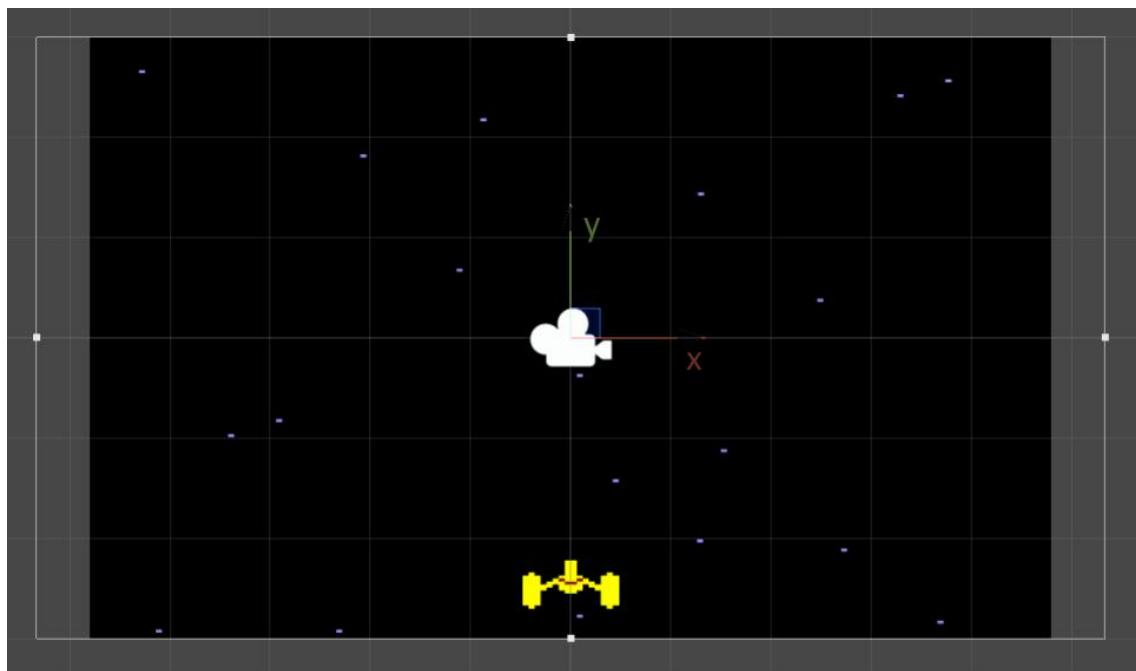
public class Enemigo : MonoBehaviour
{
    private float velocidadX = 2;
    private float velocidadY = -1.1f;

    / Start is called before the first frame update void Start()
    {
    }

    / Update is called once per frame void Update()
    {
        transform.Translate(velocidadX *
            Time.deltaTime, velocidadY *
            Time.deltaTime,
            0);
        if ((transform.position.x < -4) || (transform.position.x
            > 4)) velocidadX = -velocidadX;
        if ((transform.position.y < -2.5) || (transform.position.y >
            2.5)) velocidadY = -velocidadY;
    }
}
```

Para las **coordenadas** máximas y mínimas en las que puede moverse, se puede comprobar la “rejilla” que aparece superpuesta en la escena, o bien directamente mover algún elemento, como nuestra nave, con el ratón (tras escoger la herramienta de “Mover”) a distintos puntos de la pantalla y ver qué coordenadas aparecen en el Inspector.

El centro de la pantalla coincide inicialmente con la posición (0,0), las coordenadas X crecen hacia la derecha y las Y crecen hacia arriba. Por ejemplo, en el caso de nuestro juego, tras ajustar el tamaño de la cámara, la Y de la zona visible en la cámara va desde -3 hasta +3, mientras que la X va aproximadamente desde -5 hasta +5.



Ejercicio propuesto 1.9.1: Elige y/o prepara una imagen de un enemigo para tu juego. Crea un objeto a partir de ella y haz que se mueva por sí solo.

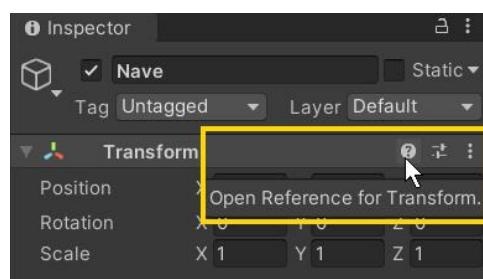
1.10. Cómo obtener ayuda

En ocasiones será interesante poder contar con la referencia oficial, para poder consultar los detalles sobre un cierto elemento de Unity o alguno de sus métodos o propiedades.

Por ejemplo, para saber más sobre lo que nos permite el “Transform” de un objeto, será interesante consultar la página:

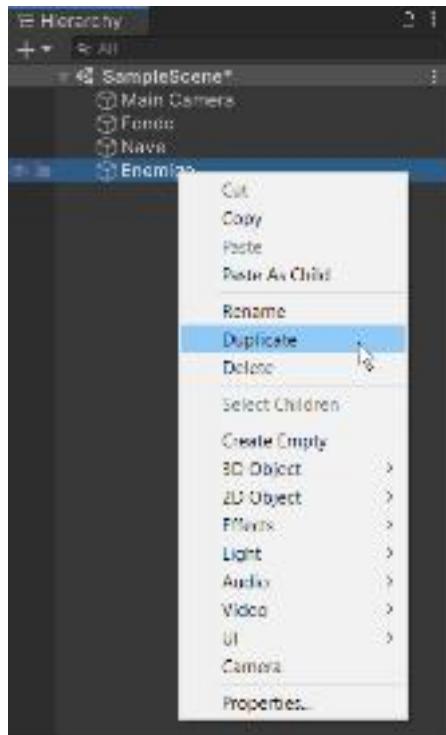
<https://docs.unity3d.com/ScriptReference/Transform.html>

Aun así, en general no es necesario memorizar estas direcciones ni tenerlas en marcadores, porque existe un botón que permite acceder a la ayuda sobre cada elemento desde el propio inspector, haciendo clic en el ícono que muestra una interrogación y que aparece en la parte superior de cada componente de un objeto:

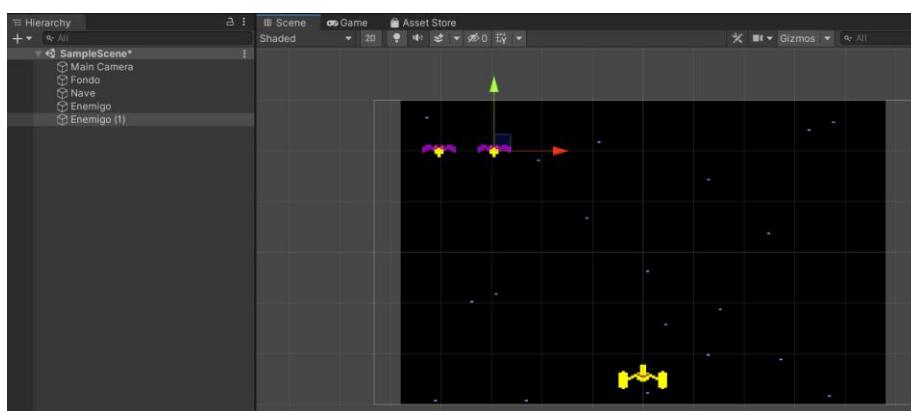


1.11. Múltiples enemigos. Jerarquías de objetos

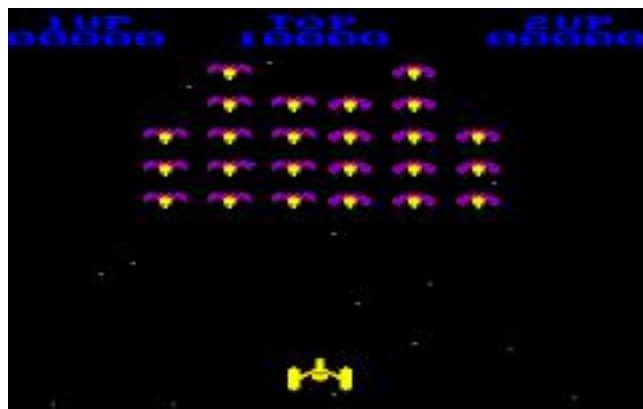
La forma más sencilla de crear varios enemigos, una vez que tenemos afinado el comportamiento y apariencia de uno de ellos, es duplicar a partir del primero, usando el botón derecho del ratón (o las teclas Ctrl+D), en el panel de la jerarquía:



La copia quedará en la misma posición que el objeto original, pero la podemos mover “a ojo” con la correspondiente herramienta

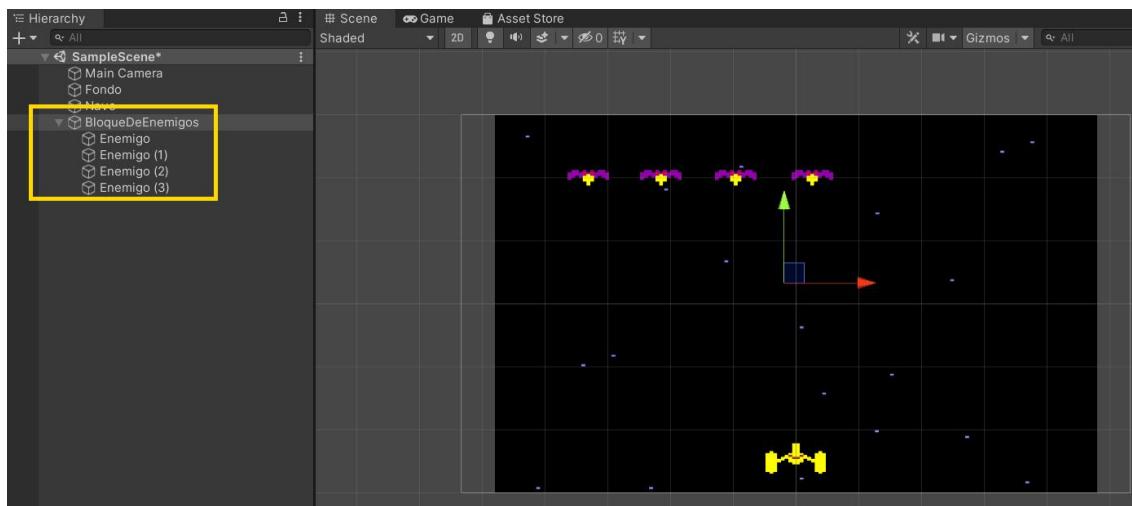


Por ejemplo, el Space Hawks original tenía 24 enemigos ($2+4+6+6+6$):



Pero en nuestro caso vamos a añadir sólo 3 enemigos adicionales (hasta un total de 4), porque veremos que esta forma no es la más adecuada y dará lugar a problemas posteriores.

Si añadimos muchos elementos, la “jerarquía” (el panel izquierdo) empezará a quedar cada vez más saturada, por lo que podemos crear un objeto vacío (botón derecho, “Create Empty”), que se podría llamar “BloqueDeEnemigos” y que actuará como contenedor de todos los enemigos. Arrastramos los enemigos a él, y así podremos “plegar”/“desplegar” todo ese bloque cuando nos interese:



Ejercicio propuesto 1.11.1: Añade varios enemigos a tu juego, duplicando uno existente. Haz que formen parte de un único objeto contenedor.

1.12. Contacto con los “prefabs”

Duplicar objetos como hemos hecho tiene una carencia grave: si descubrimos que hemos olvidado algún detalle de los enemigos, ahora

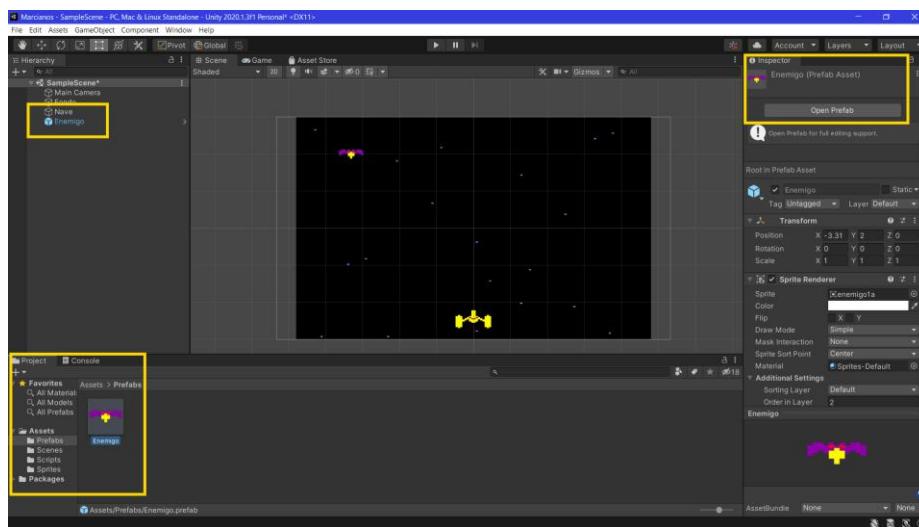
deberíamos cambiarlo en cada uno de ellos, uno por uno, lo que resulta poco eficiente y propenso a errores, especialmente si tenemos muchos objetos.

Por ejemplo, imaginemos que no hubiéramos añadido el “script” todavía al primer enemigo. Ahora deberíamos añadirlo a los 4, uno por uno. Lo mismo ocurrirá si queremos hacer algún cambio, como podría ser incluir una animación.

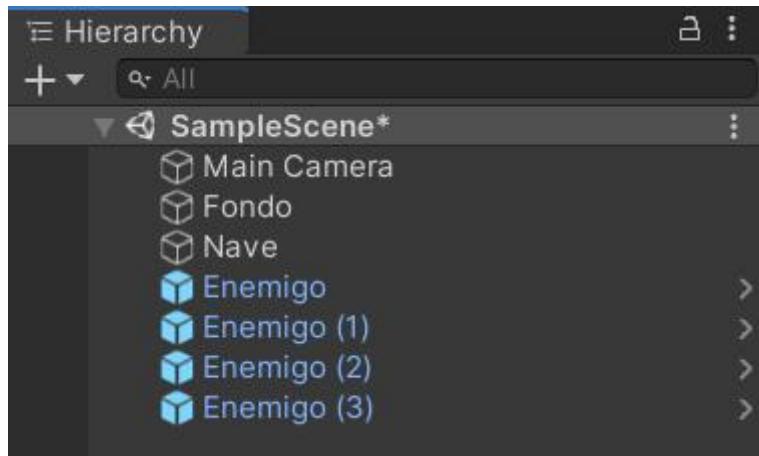
La alternativa es crear “**prefabs**”, que serán algo así como “objetos prefabricados reutilizables”. Crear (y usar) un prefab apenas lleva 3 pasos:

- El primer paso, no necesario pero recomendado, es crear una nueva carpeta dentro de nuestro proyecto. El nombre puede ser cualquiera, pero lo ideal es llamarla “Prefabs” para encontrar los distintos componentes con más facilidad. En nuestro caso, que ya habíamos creado varios enemigos repetitivos, también deberíamos borrarlos, desde Enemigo(1) hasta Enemigo(3).
- El siguiente paso será arrastrar el objeto “Enemigo” desde el panel de la jerarquía (izquierda) al panel de proyecto (inferior). Veremos que el objeto de la jerarquía cambia a color azul, para indicarnos que se basa en un “prefab”.





- Si ahora queremos añadir objetos basados en ese “prefab”, basta con arrastrar en orden inverso, desde el proyecto a la jerarquía. Veremos que los nuevos objetos también aparecen en color azul.



En el próximo apartado cómo modificar a la vez todos los elementos que se basan en un “prefab”, y más adelante veremos cómo crear nuevos objetos desde programa a partir de “prefabs”, en vez de hacerlo desde el diseñador visual.

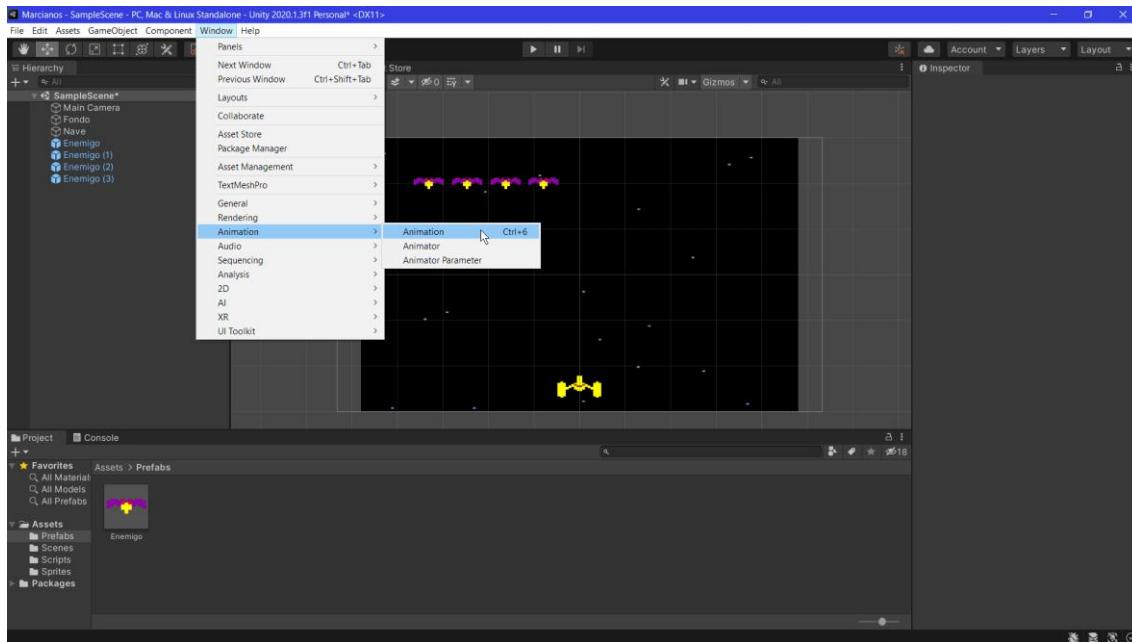
Ejercicio propuesto 1.12.1: Crea un “prefab” a partir de tu enemigo. Tu juego debe tener varios enemigos iguales, pero estarán basados en ese “prefab”, en vez de ser duplicados.

1.13. Un sprite animado

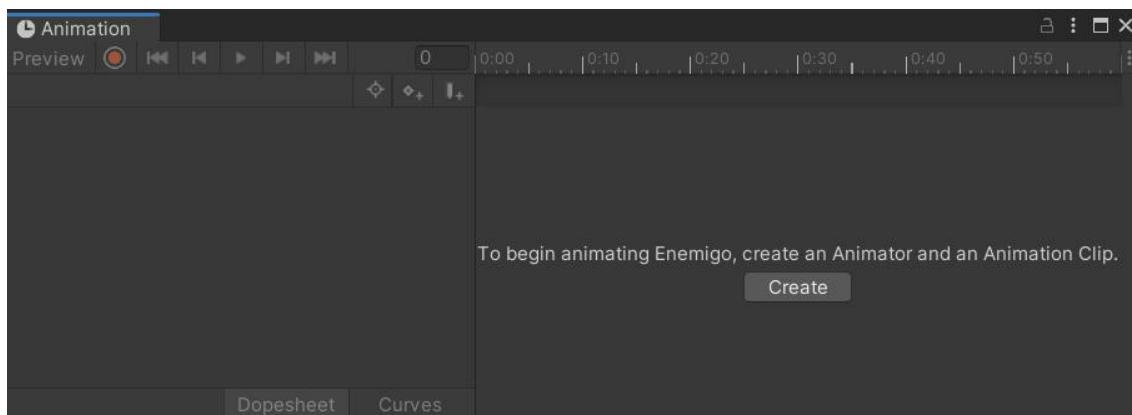
Es muy habitual que los personajes de un juego tengan varios “fotogramas”, que, reproducidos en secuencia, den una apariencia de

movimiento. Unity tiene soporte para animaciones de personajes 2D, tanto si partimos de imágenes individuales como si empleamos una única imagen que contenga todos los sprites (lo que llamaremos una “spritesheet”).

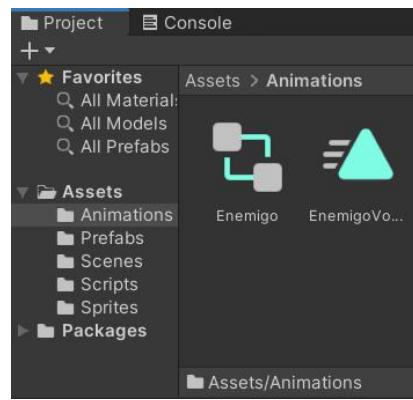
Deberemos comenzar por abrir la ventana de animación, desde el menú Window, opción Animation y subopción **Animation** (no “Animator”, que veremos más adelante):



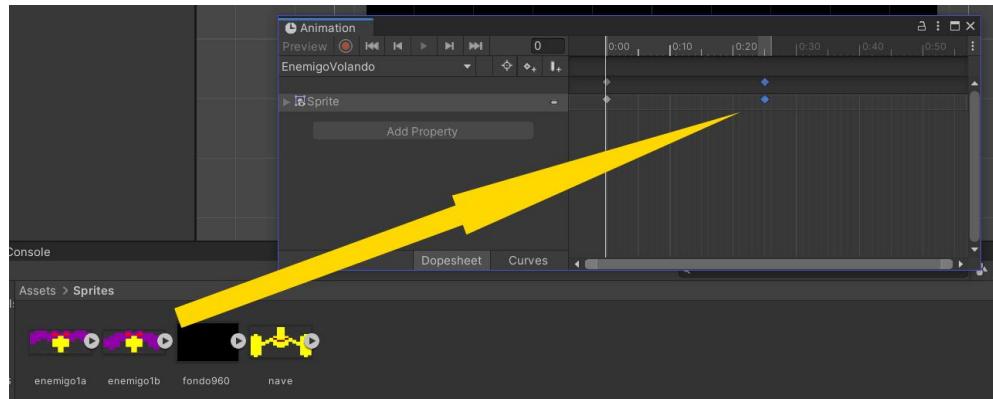
Aparecerá una ventana. Si tenemos seleccionado algún objeto, como nuestro enemigo, o hacemos clic en él ahora, aparecerá un botón para crear una animación:



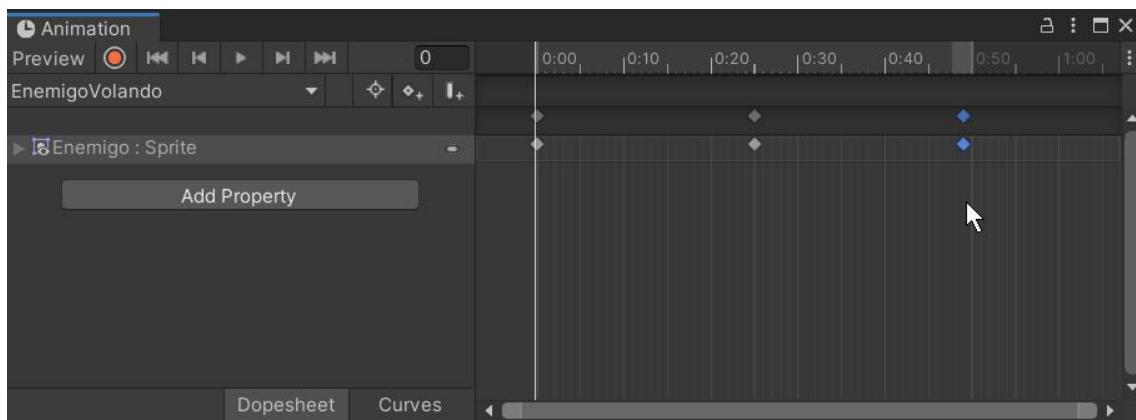
Y deberemos elegir un nombre, que puede ser “enemigoVolando” (y podemos aprovechar para crear una carpeta específica para guardar todas las animaciones):



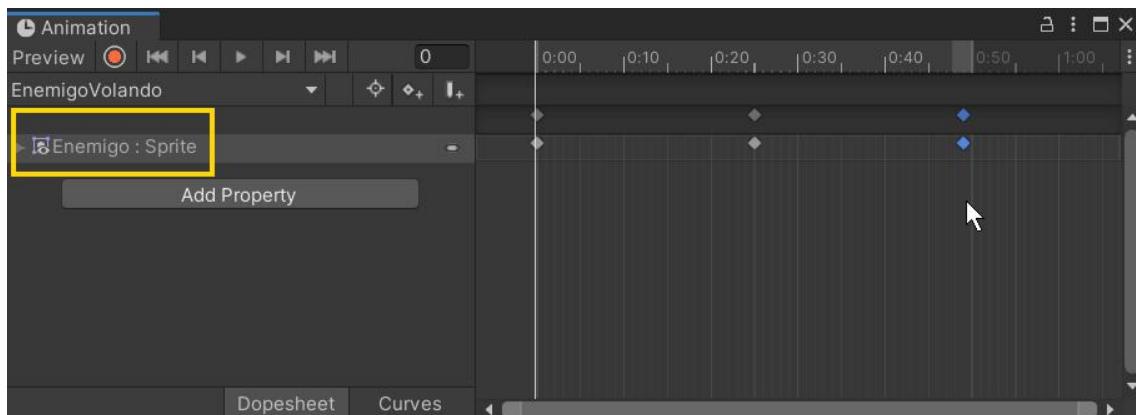
Y bastará con seleccionar las dos (o más) imágenes que tenemos preparadas para nuestro enemigo y arrastrarlas a la parte superior derecha de la ventana de animación, debajo de la línea de tiempo:



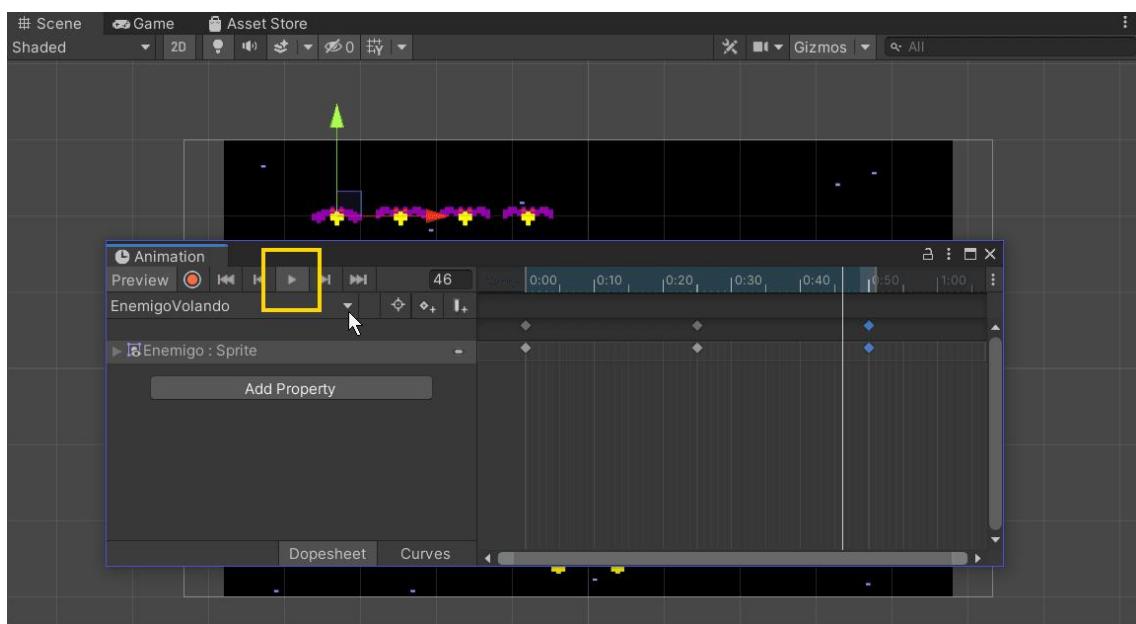
Y entonces aparecerán dos rombos azules, que representan los “fotogramas clave”. Podremos separarlos o juntarlos, según si queremos que la animación sea más lenta o más rápida:



Veremos que en la parte izquierda de la ventana se nos avisa de que estamos cambiando la propiedad “sprite”, pero hay muchos más detalles que podríamos animar, como veremos más adelante.



Además tenemos un botón “play” que nos permite ver cómo está quedando la animación, para asegurarnos de que no sea demasiado rápida ni demasiado lenta.



Si la velocidad no nos pareciese adecuada, podríamos acercar o separar los fotogramas clave hasta encontrar un efecto que nos guste.

El problema es que sólo tendrá animación el objeto que hemos escogido, uno de los enemigos, pero no todos ellos. Aplicaremos los cambios a todos los objetos en el siguiente apartado.

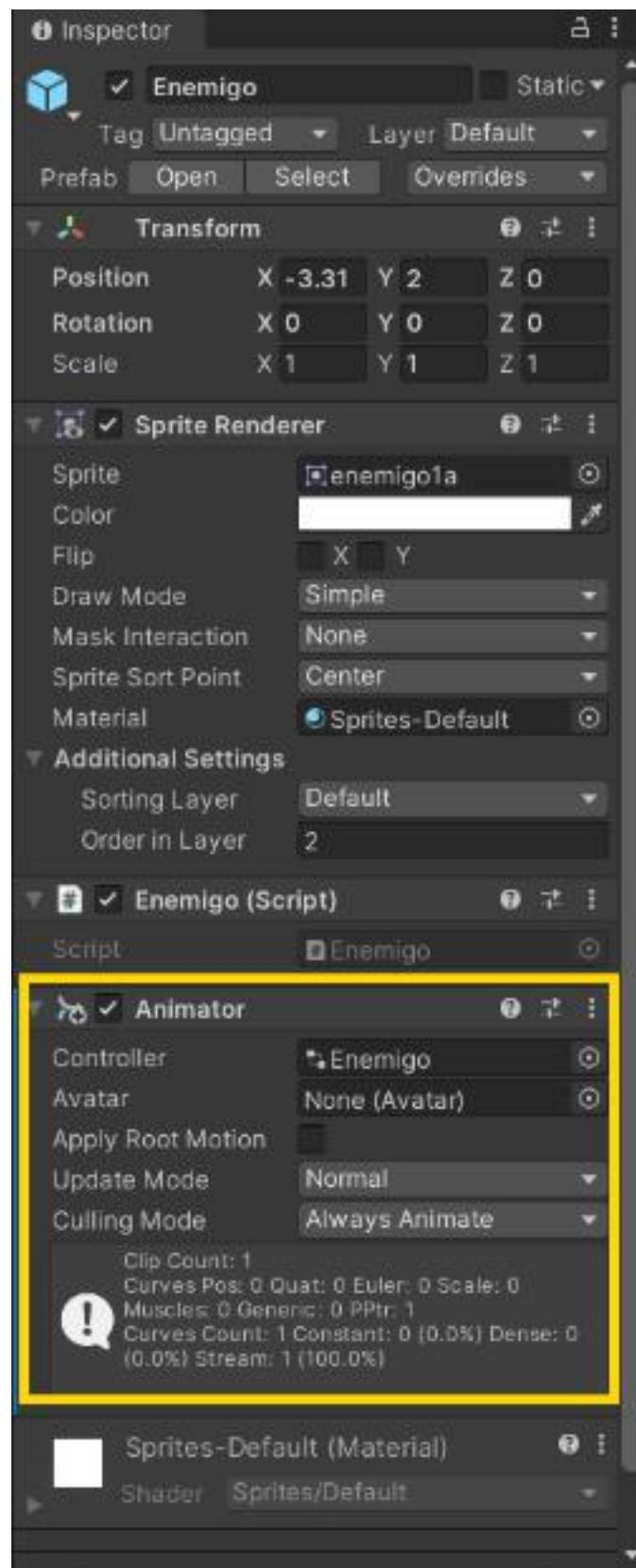
Más adelante, cuando hagamos un “juego de plataformas”, veremos cómo tener varias animaciones distintas (por ejemplo, una para el movimiento hacia la derecha y otra hacia la izquierda, o bien una -o varias- para el salto). También veremos cómo encadenar unas animaciones entre otras, o incluso lanzar “eventos” en un cierto punto de la animación.

Ejercicio propuesto 1.13.1: Añade una animación a uno de tus enemigos.

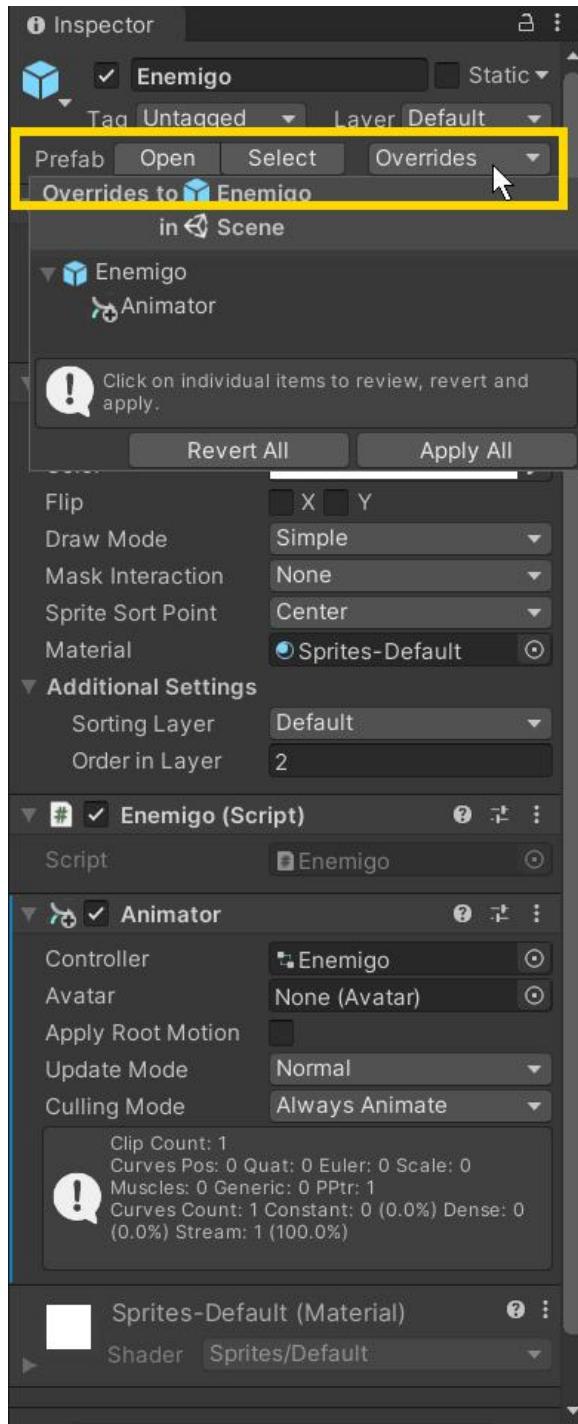
1.14. Aplicando los cambios al prefab

Hemos añadido una animación a uno de nuestros enemigos, pero los demás no han cambiado. Si se trata, como en este caso, de un cambio que nos gustaría que se afectara a todos ellos, basta con decir que esos cambios se apliquen al prefab.

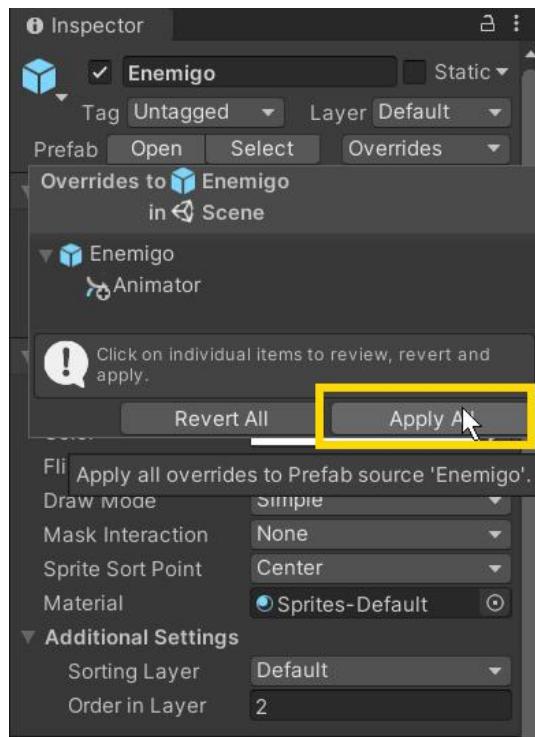
Vamos a comenzar por seleccionar el enemigo que contiene la animación y mirar en el inspector: veremos que tiene un componente “Animator”, que los demás enemigos no tienen:



Pero en la parte superior del inspector se nos recuerda que este objeto se basa en un prefab



Si desplegamos “Overrides”, se nos permite aplicar los cambios (“Apply all”) al prefab del que procede (también podríamos usar “Revert all” para deshacer los cambios):



Y ahora no sólo los cuatro enemigos existentes tienen animación: si añadimos un quinto enemigo a partir del prefab, también la tendrá:



Ejercicio propuesto 1.14.1: Añade una animación a tu enemigo o tu personaje.

1.15. Propiedades editables desde el inspector: [SerializeField]

Unity da la posibilidad de que alguna de las propiedades de nuestros objetos se pueda modificar no sólo desde programa, sino también desde el editor, lo que hace que sea más fácil hacer pruebas rápidas. Hay dos formas de conseguirlo:

- Declarar la propiedad como “public”, lo que tiene el efecto secundario (habitualmente negativo) de que se podrá modificar desde otros elementos del juego.
- Etiquetar la propiedad como “[SerializeField]”, lo que no tiene ese efecto secundario, y será la alternativa que más utilicemos.

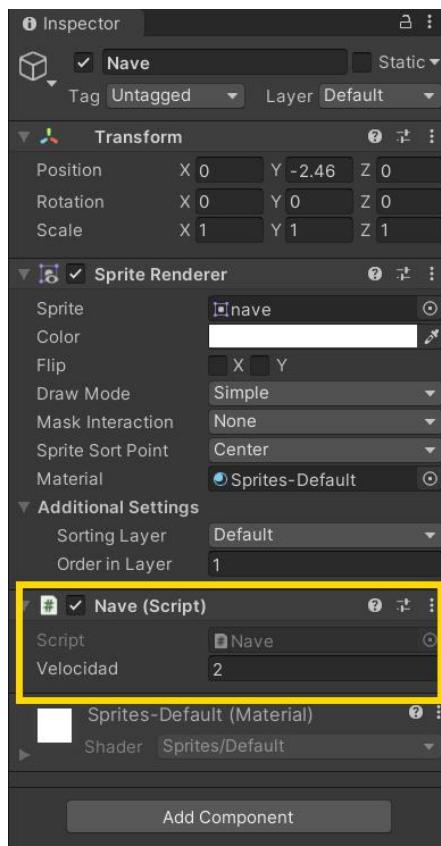
Así, en el script de nuestra nave podríamos cambiar

```
private float velocidad = 2;
```

por

```
[SerializeField] float velocidad = 2;
```

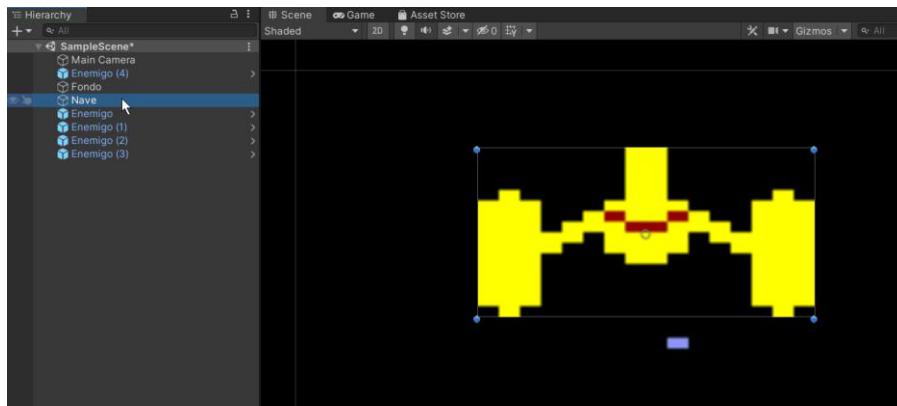
y se vería así en el inspector:



Ejercicio propuesto 1.15.1: Haz que la velocidad de tu nave esté accesible desde el inspector.

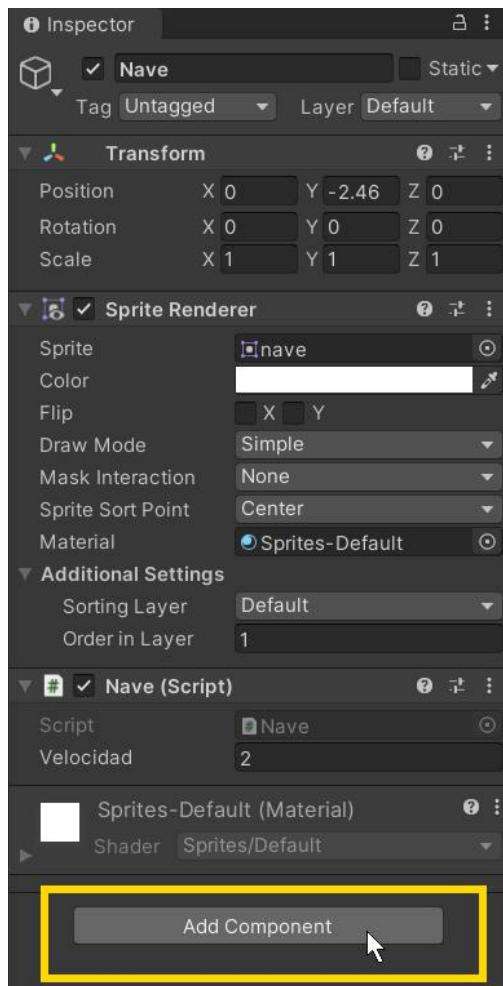
1.16. Cuerpos rígidos (Rigidbody2D)

Vamos a ver cómo detectar que nuestra nave choca con algún enemigo. Como primer paso, nos puede interesar verla centrada en pantalla, haciendo doble clic en su nombre, en la jerarquía.

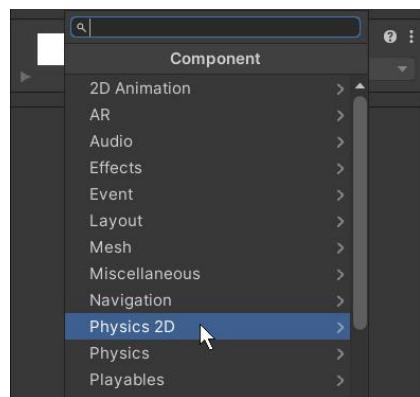


(Una forma alternativa, quizá un poco más incómoda, de “enmarcar” un objeto en la escena es seleccionarlo con un clic, luego desplazar el ratón hasta la escena y entonces pulsar la tecla F - abreviatura de “frame”, marco-).

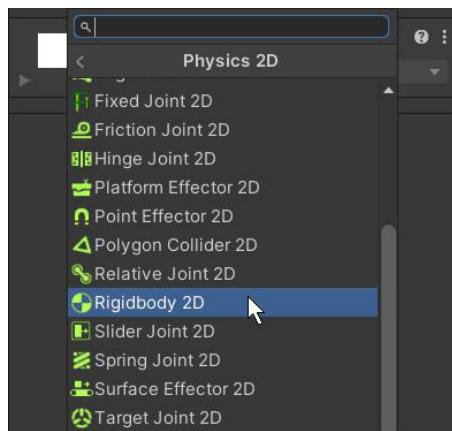
Para que Unity compruebe por nosotros las colisiones entre dos objetos, al menos uno de ellos debe tener un “cuerpo rígido” (Rigidbody2D) asociado. La forma de hacerlo, con la nave seleccionada, será pulsar el botón “AddComponent” en el Inspector



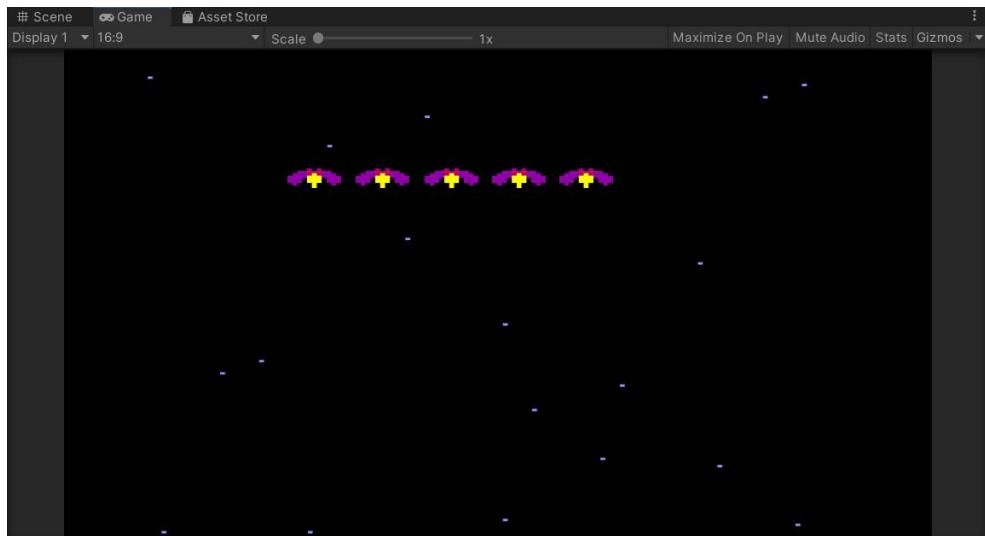
A continuación, deberemos dirigirnos a la categoría Physics 2D:



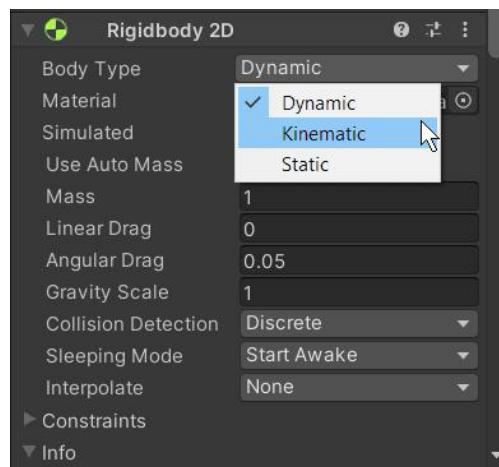
Y, dentro de la categoría Physics 2D, escoger un RigidBody 2D:



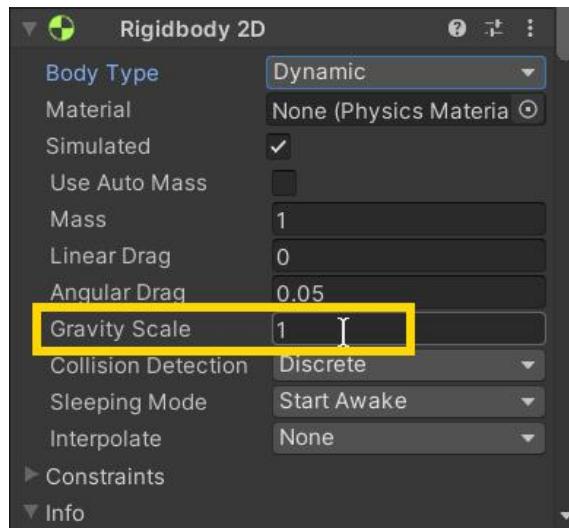
Pero eso tendrá un efecto no deseado: cuando lancemos el juego, la nave “pesará” y “se caerá” fuera de la pantalla:



Hay varias formas de solucionarlo. Por ejemplo, podríamos cambiar el tipo de cuerpo, para que en vez de ser “dinámico” (dynamic) pase a ser “cinemático” (kinematic) y no le afecten las fuerzas:



Eso es algo que usaremos (por ejemplo) en las plataformas de juegos como el que haremos dentro de poco. Otra alternativa, la que emplearemos en esta ocasión, es que sí sea un cuerpo “dinámico” pero que no le afecte la gravedad, indicando un valor 0 para la “escala de la gravedad”:

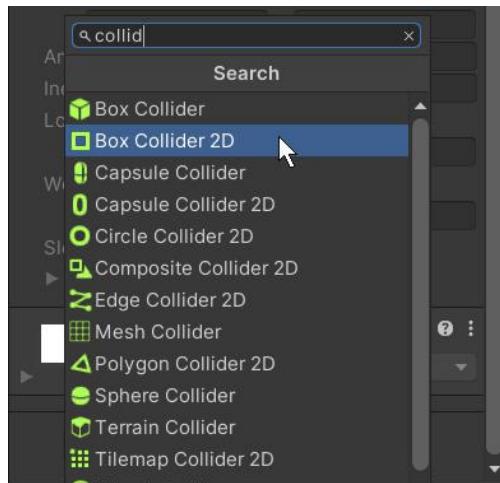


Cuidado: Un fallo frecuente cuando se empieza con Unity es añadir un Rigidbody en vez de un Rigidbody2D: las “físicas 3D” y las “físicas 2D” no se podrán mezclar.

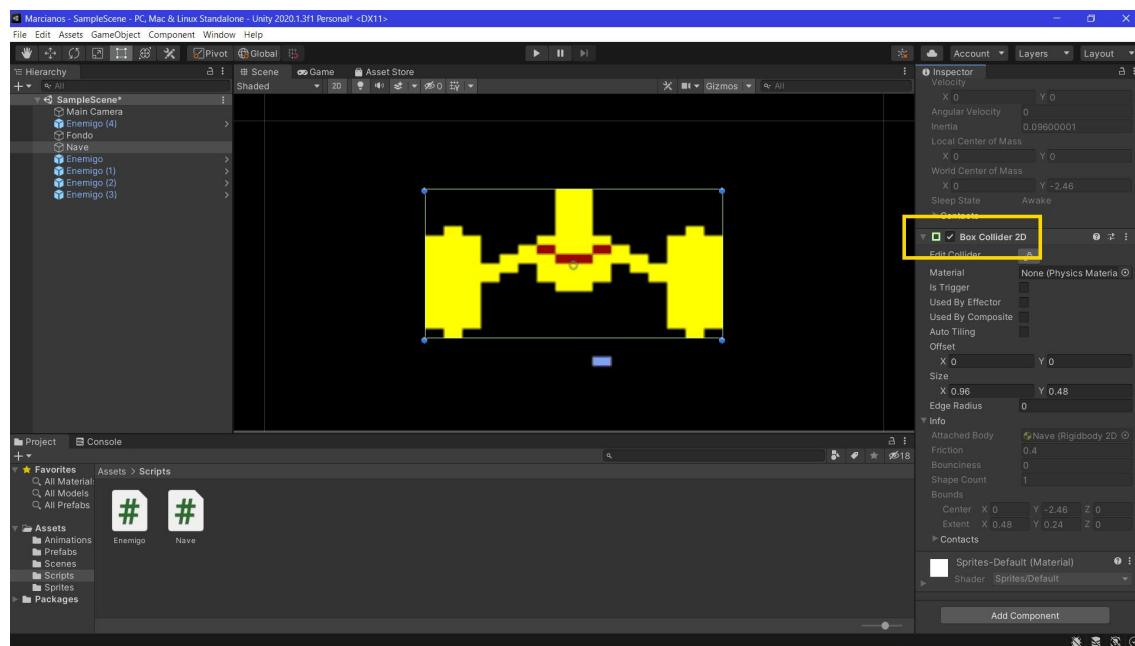
Ejercicio propuesto 1.16.1: Añade un Rigidbody2D a tu nave, pero no permitas que le afecte la gravedad.

1.17. Comprobación básica de colisiones (BoxCollider2D)

Ahora que ya tenemos un “cuerpo rígido”, también deberemos añadir un “comprobador de colisiones” (o “collider”). El más sencillo es el “BoxCollider2D”, que se encargará de comprobar colisiones empleando rectángulos, ya que tanto nuestra nave como nuestro “enemigo” son “básicamente rectangulares”:

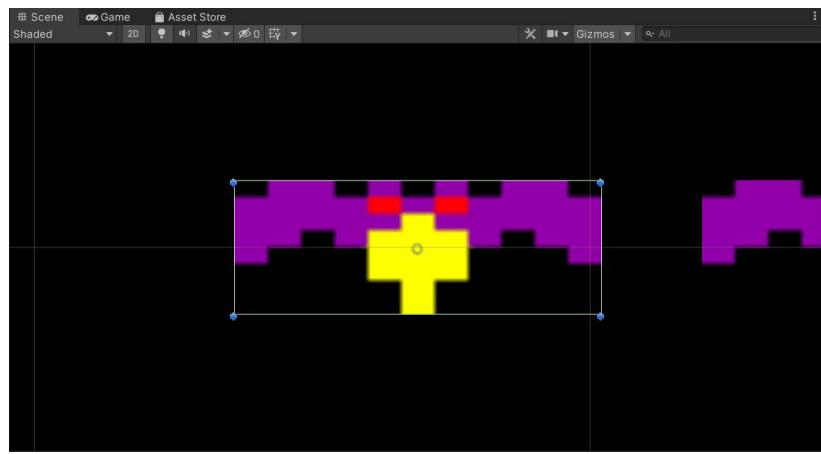


Alrededor de la imagen aparecerá un recuadro verde, que indica el “collider”. Podemos afinar su posición y tamaño para encuadrar la imagen de la forma más precisa posible, aunque es habitual que automáticamente se ajuste bastante bien al sprite que hayamos utilizado:

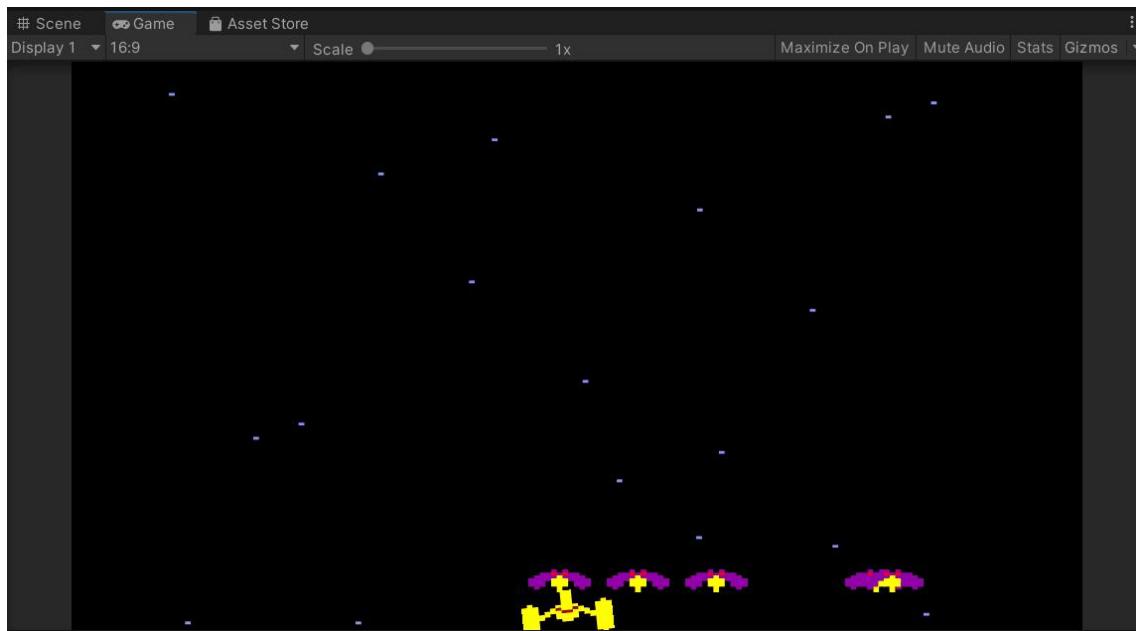


Nota: También hay “colliders” con otras formas. Los más habituales, junto con los rectangulares, serán los circulares y los poligonales, que emplearemos más adelante.

Sólo uno de los elementos que participa en la colisión necesita tener un Rigidbody2D, así que añadiremos al “enemigo” (seleccionando el “prefab”, no un objeto individual) únicamente un BoxCollider2D (y, nuevamente, comprobaremos su posición y su tamaño):



Si ahora lanzamos el juego, veremos que la nave podrá chocar con el enemigo. El comportamiento esperable cuando dos objetos “físicos” choquen es que reboten y/o giren, que es lo que ocurrirá en este momento, pero no es algo deseable en un juego de este tipo, así que lo cambiaremos en el siguiente apartado:



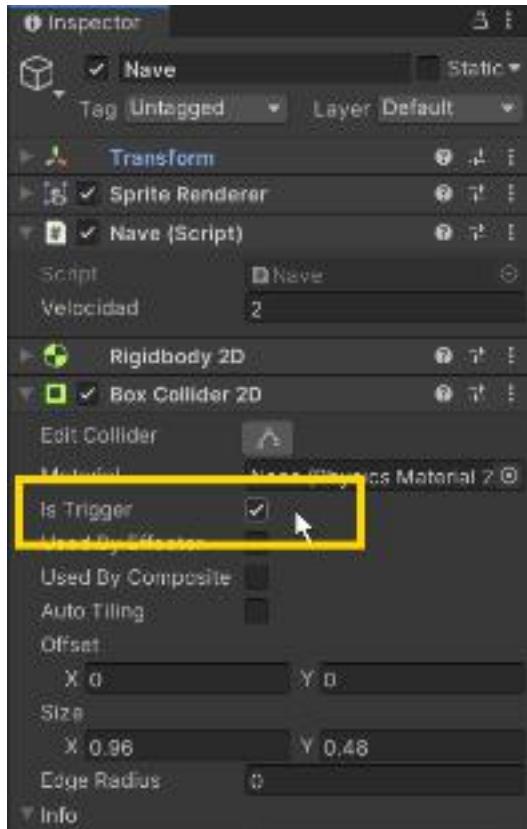
Ejercicio propuesto 1.17.1: Añade colliders a tu nave y a tu enemigo y ajusta sus posiciones y su zona de movimiento para que puedan chocar entre ellos.

1.18. Aviso de colisiones

En un juego como éste no queremos que los disparos reboten en los enemigos o viceversa. Nos bastará con saber que se han tocado, para que ambos desaparezcan de la pantalla.

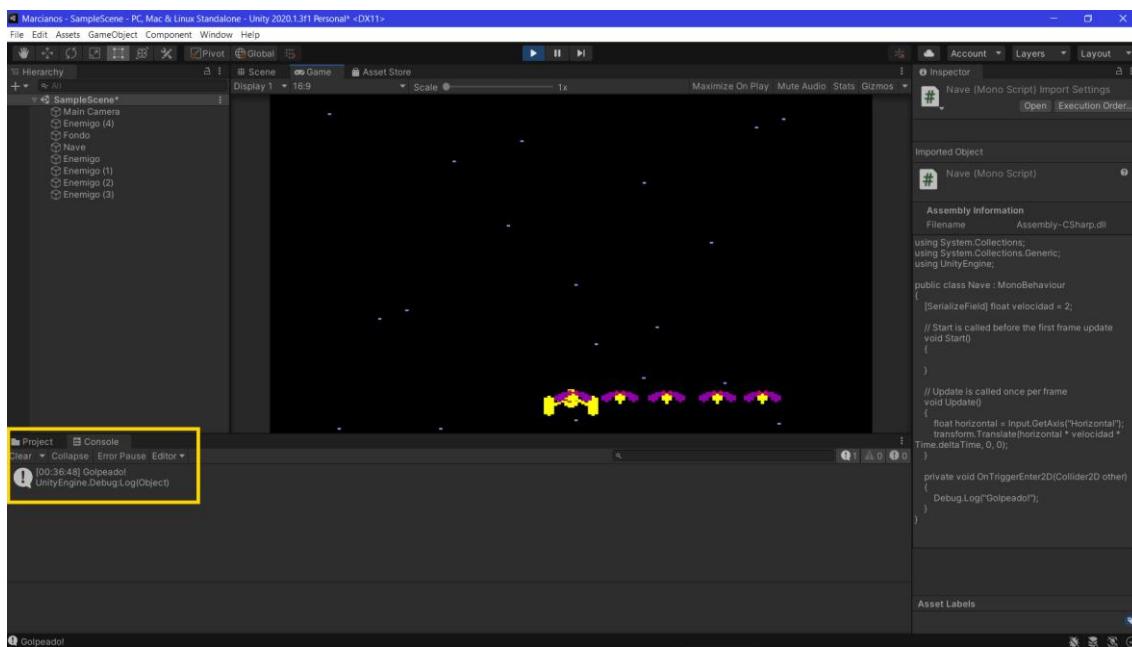
Para conseguir eso, haremos que no se procese la colisión de forma normal, desencadenando “reacciones físicas” en los objetos, sino que simplemente se nos avise de que han colisionado.

Para eso, activaremos el checkbox “**is trigger**” en el “Collider” del elemento que queremos que nos avise, que en nuestro caso podría ser nuestra nave:



Y para indicar qué se debe hacer cuando se detecte la colisión, deberíamos implementar el método “`OnTriggerEnter2D`”. De momento, nos podríamos limitar a escribir el texto “Hola” en la consola, y más adelante haremos “cosas más reales”:

```
private void OnTriggerEnter2D(Collider2D other)
{
    Debug.Log("Golpeado!");
}
```



Ejercicio propuesto 1.18.1: Comprueba colisiones con el enemigo y muestra un texto cuando se dé una colisión.

1.19. Control de versiones y copias de seguridad

Nuestro proyecto va creciendo, y es recomendable hacer una copia de seguridad cada cierto tiempo, para no perder todo en caso de error grave o de avería del equipo. Es más, es altamente recomendable no (sólo) hacer copias de seguridad de todo el proyecto en bloque, sino usar un sistema de control de versiones, como Git, que nos permita retroceder de forma sencilla a puntos anteriores en caso de problemas.

Pero Unity crea varias carpetas dentro de un proyecto que contienen información “cacheada”, que ocupan mucho espacio y que no son necesarias en la copia de seguridad, porque se regenerarían si llevamos el proyecto a otro ordenador. Por ejemplo, es el caso de las carpetas “Library” y “Temp”. Deberías excluir esas dos carpetas (y quizás otras auxiliares que cree Visual Studio, como “obj”) de tu copia de seguridad y de los datos que se vuelcan a tu repositorio Git.

Un proyecto tan sencillo como este, con Unity 2020 ocupa más de 150 MB. Incluso comprimido en formato ZIP sigue siendo de más de 50 MB. Borrando esas carpetas del contenido del fichero comprimido, éste baja hasta un tamaño inferior a 50 KB.

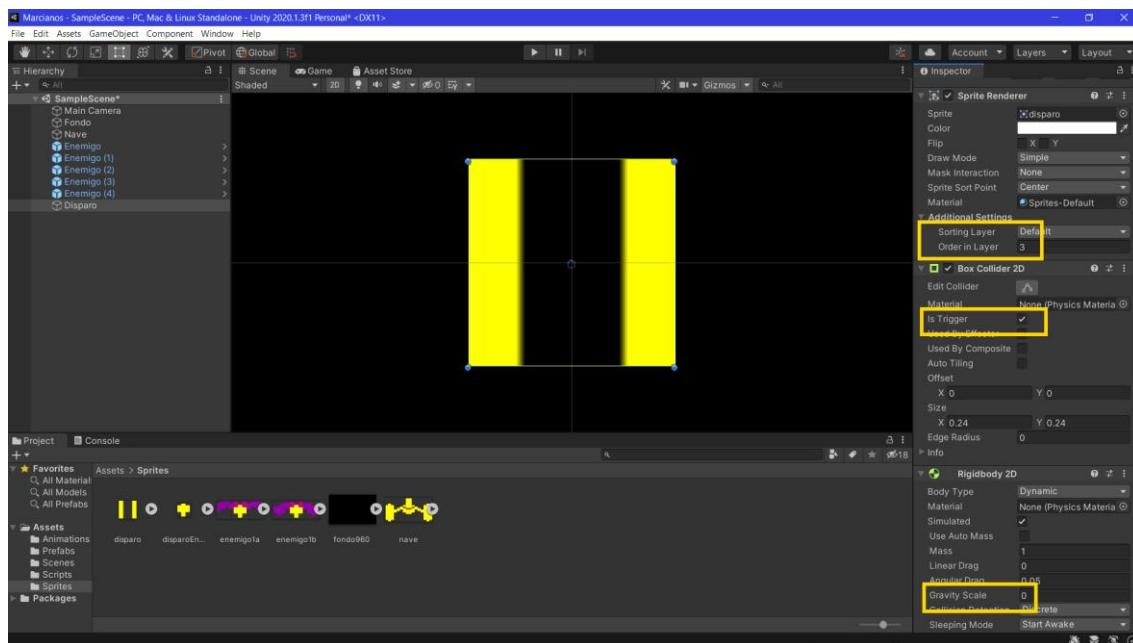
Ejercicio propuesto 1.19.1: Crea una copia de seguridad de tu juego. Es deseable (pero no imprescindible) que, además, crees un repositorio Git para él.

1.20. Añadiendo un disparo: Instantiate

Para crear el disparo del personaje hacia los enemigos seguiremos los siguientes pasos:

- Añadir el sprite a la carpeta de Sprites del proyecto.
- Arrastrarlo a la escena para crear un objeto (Game object) a partir de esa imagen.
- Ajustar su altura dentro de la capa visual (o su capa), para que quede por encima del fondo.
- Añadirle un BoxCollider2D, para que se puedan comprobar colisiones con él. Como no queremos rebotes, sino detectar la colisión, lo marcaremos como “trigger”.
- Al menos uno de cada dos elementos que participen en una colisión debe tener un “RigidBody”. Como queremos que el disparo impacte con los enemigos, que no lo tienen, deberemos añadir un “RigidBody2D” al disparo. Para que no le afecte la gravedad, daremos un valor 0 a su “gravity scale”.

El resultado será algo como:



A continuación, arrastramos el disparo desde la jerarquía al panel inferior, para **crear un “prefab”**, y lo eliminamos de la jerarquía, para que no exista ningún disparo activo en el juego.

Sólo falta crear un objeto en tiempo real al pulsar una cierta tecla. Lo podemos hacer **desde el script que gestiona la “Nave”**, ya que aparecerá en la misma posición que ésta. Lo conseguiremos con la orden “Instantiate”, que, en su formato más cómodo, recibe tres parámetros:

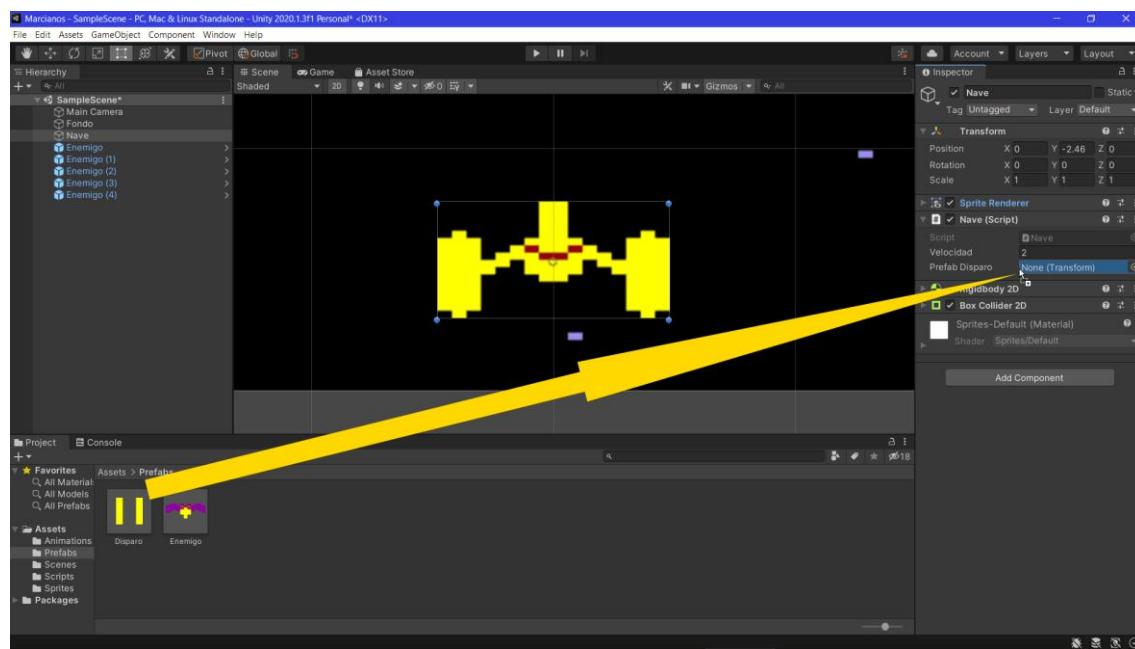
- El tipo de objeto a crear
- La posición en la que debe aparecer
- Su rotación

Esto supone varios pasos, que en nuestro caso serían:

- La forma más habitual de crear el tipo de objeto es con un atributo que esté accesible desde el editor, con [SerializeField]:

```
[SerializeField] Transform prefabDisparo;
```

Y daríamos valor a ese atributo arrastrando el “prefab” hasta esa casilla:



- La posición en que aparezca será la misma que la de la nave, por lo que escribiremos “transform.position”.
- La rotación será “ninguna”, lo que se expresa como “Quaternion.identity”.

Con todo ello, el código que deberíamos escribir sería algo como:

```
Instantiate(prefabDisparo, transform.position, Quaternion.identity);
```

Y entonces el método Update de la clase Nave podría quedar así:

```
void Update()
{
    float horizontal = Input.GetAxis("Horizontal");
    transform.Translate(horizontal * velocidad * Time.deltaTime,
    0, 0);

    if (Input.GetButtonDown("Fire1"))
    {
        Instantiate(prefabDisparo, transform.position, Quaternion.identity);
    }
}
```

Al pulsar la tecla Ctrl (o el primer botón de disparo del Gamepad), el disparo aparecerá, pero todavía no se moverá. De hecho, como es de color amarillo, igual que la nave, y aparece sobre ésta, apenas se distingue, a no ser que movamos la nave:





Ejercicio propuesto 1.20.1: Haz que aparezca el disparo cuando pulses la tecla (o botón) correspondiente.

1.21. Un disparo que se mueve: GetComponent

Estábamos creando un disparo pero luego nos desentendíamos de él. Si queremos cambiar alguna de sus propiedades, como por ejemplo su velocidad, deberemos guardar en una variable el objeto que genera “Instantiate”, que es de tipo “Transform”:

```
Transform disparo =
    Instantiate(prefabDisparo,
    transform.position,
    Quaternion.identity);
```

Ahora tendremos que buscar la velocidad asociada al componente “Rigidbody2D” del GameObject asociado a ese disparo, y decir que sea una velocidad vertical, así:

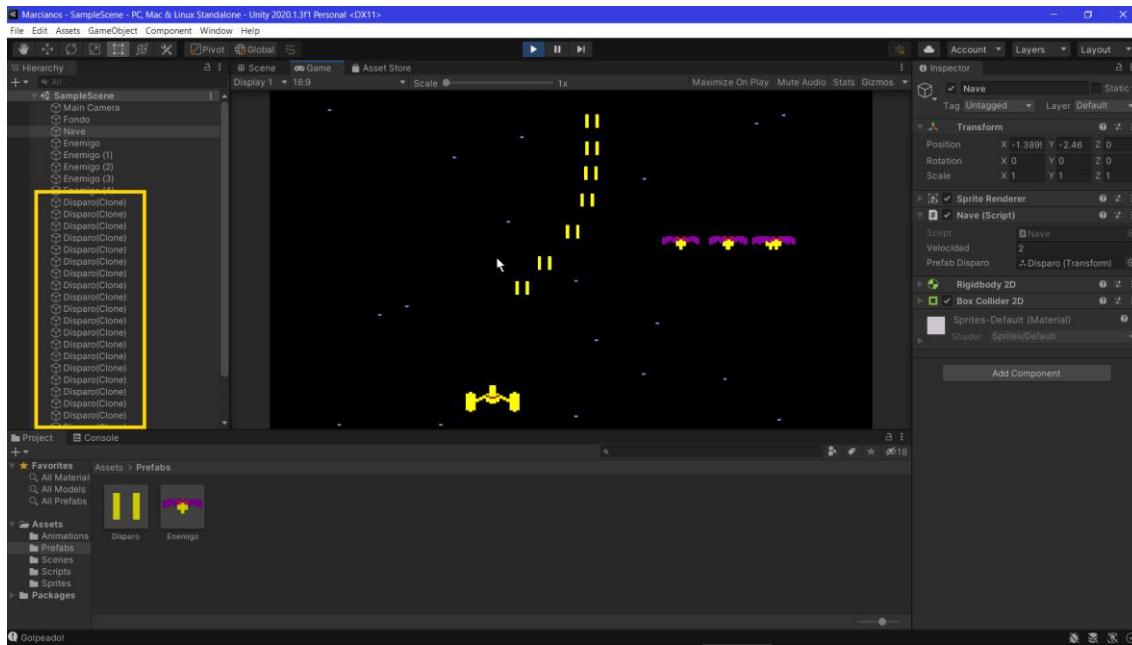
```
disparo.gameObject.GetComponent< Rigidbody2D >().velocity =
    new Vector3(0, velocidadDisparo, 0);
```

Donde ese “velocidadDisparo” será un nuevo atributo:

```
private float velocidadDisparo = 2;
```

Nota: Ese “GetComponent” para acceder a componentes de un objeto es algo que usaremos con frecuencia en programas un poco más avanzados. Como ya veremos, también usaremos construcciones parecidas para acceder a otros objetos de nuestro juego.

Pero el problema es que el disparo aparece, se mueve... y cuando sale de la pantalla sigue subiendo eternamente, sin destruirse. Y lo mismo ocurre con los nuevos disparos que van apareciendo cuando se vuelve a pulsar la tecla:



Una forma sencilla de solucionarlo puede ser asociar un script a la clase Disparo, para que sea el propio disparo el que compruebe si ha pasado de una cierta altura, y se autodestruya en ese caso.

Asociar un script a un “prefab” es igual de sencillo que hacerlo para un GameObject: hacemos un clic en el “prefab” para seleccionarlo y pulsamos el botón “Add Component” del inspector.

En ese script, bastará con comprobar su coordenada Y en el método “Update” y pedir que destruya el objeto cuando se supere una cierta posición, así:

```
public class Disparo : MonoBehaviour
{
    void Update()
    {
        if (transform.position.y > 5)
            Destroy(gameObject);
    }
}
```

Nota: En general, los números mágicos, como ese “5”, es algo que deberemos evitar. Una solución más elegante y más “al estilo Unity” sería añadir un “collider” por encima de la zona visible de la pantalla, y desactivar el disparo cuando choque con ese “collider”.

Ejercicio propuesto 1.21.1: Haz que tu disparo se mueva... y desaparezca en un momento dado.

1.22. Un disparo que destruye enemigos: Tags

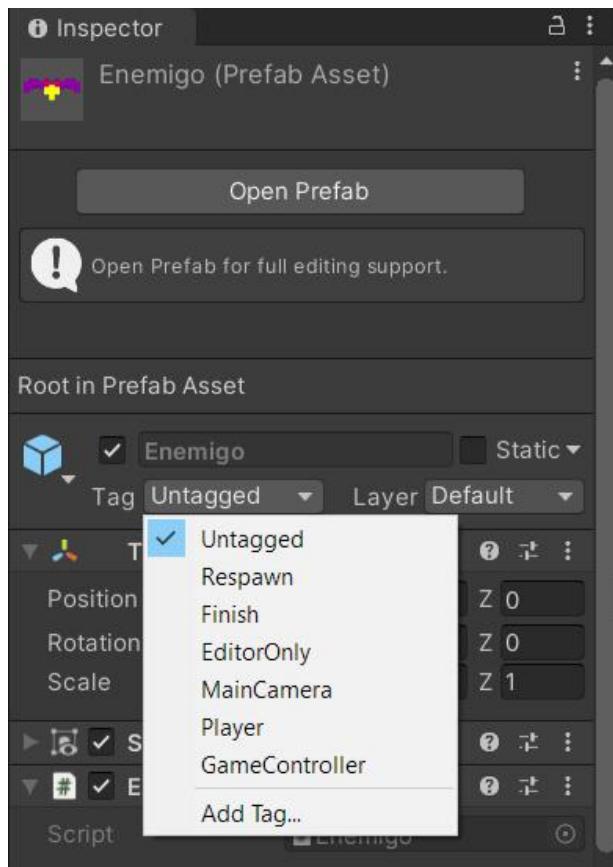
Con lo que hemos hecho hasta ahora, es fácil conseguir que el disparo destruya un enemigo. En principio, basta con ver si el disparo colisiona con algo, usando “OnTriggerEnter2D” en el script del disparo y, en ese caso, destruir el objeto correspondiente:

```
private void OnTriggerEnter2D(Collider2D other)
{
    Destroy(other.gameObject);
}
```

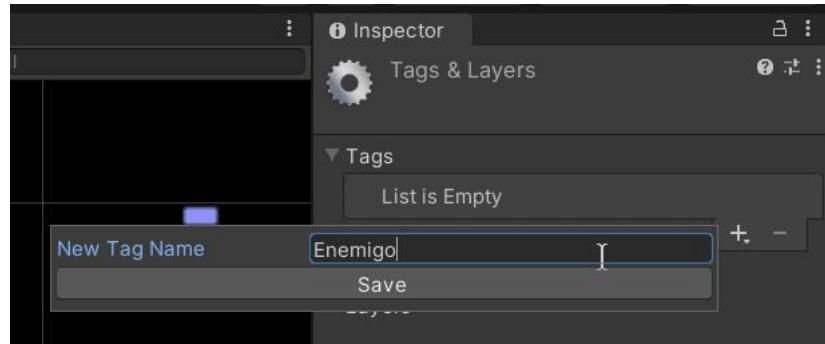
El problema es que el disparo sale desde la misma posición de la nave, por lo que también colisiona con ésta y la destruye. Una forma de solucionarlo es añadir una etiqueta (“tag”) a los enemigos, y comprobar la etiqueta del objeto con el que se colisiona. Además, también deberíamos destruir el propio disparo:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Enemigo")
    {
        Destroy(other.gameObject);
        Destroy(gameObject);
    }
}
```

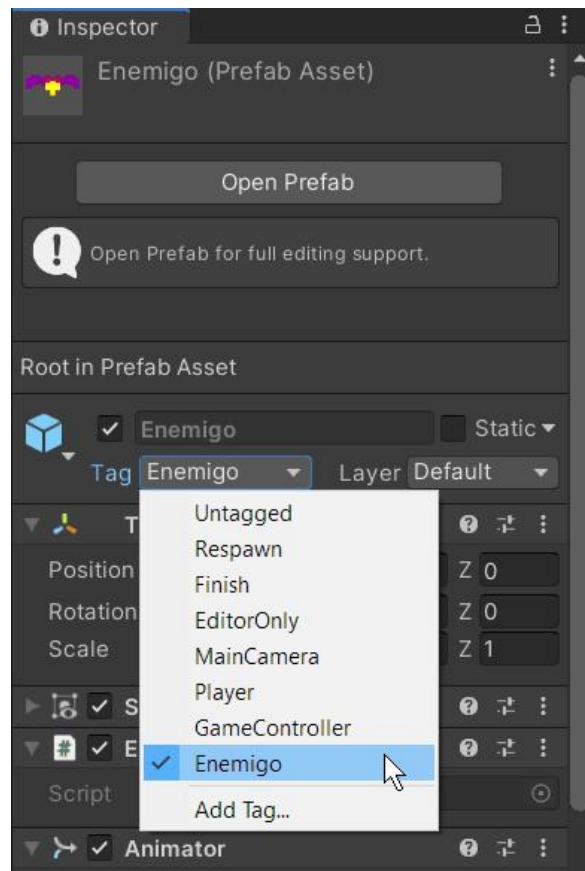
Esa etiqueta (“tag”) para distinguir a los enemigos deberíamos aplicarla al “prefab”, mejor que a un enemigo individual. Las etiquetas aparecen en el inspector, justo debajo del nombre del objeto. Si hacemos clic en el enemigo, veremos que existen algunas etiquetas predefinidas, pero ninguna que parezca adecuada para nuestro enemigo:



Para crear una nueva etiqueta, escogeremos la opción “Add Tag...” y aparecerá un nuevo panel, que muestra una lista vacía y un botón “+” para añadir elementos a esa lista:



En cuanto creamos una nueva etiqueta, ésta aparecerá en la lista de etiquetas que podemos seleccionar:



Si no hemos seleccionado el prefab, sino un objeto individual, la etiqueta se mostraría en negrita, para indicar que es diferente del resto de objetos que se basan en el “prefab”.

Como ya habíamos adelantado (apartado 1.14), en ese caso, para hacer que los demás objetos compartan también esa nueva propiedad, miraríamos la línea siguiente a las etiquetas. En ella aparece “prefab” junto con varias opciones. Si escogemos la última opción “Overrides” (reemplaza), podremos decir que queremos aplicar esos cambios a todos los objetos basados en el “prefab” (“Apply All”). A partir de ese momento, la etiqueta ya no aparecerá en negrita (porque es compartida por todos los enemigos).

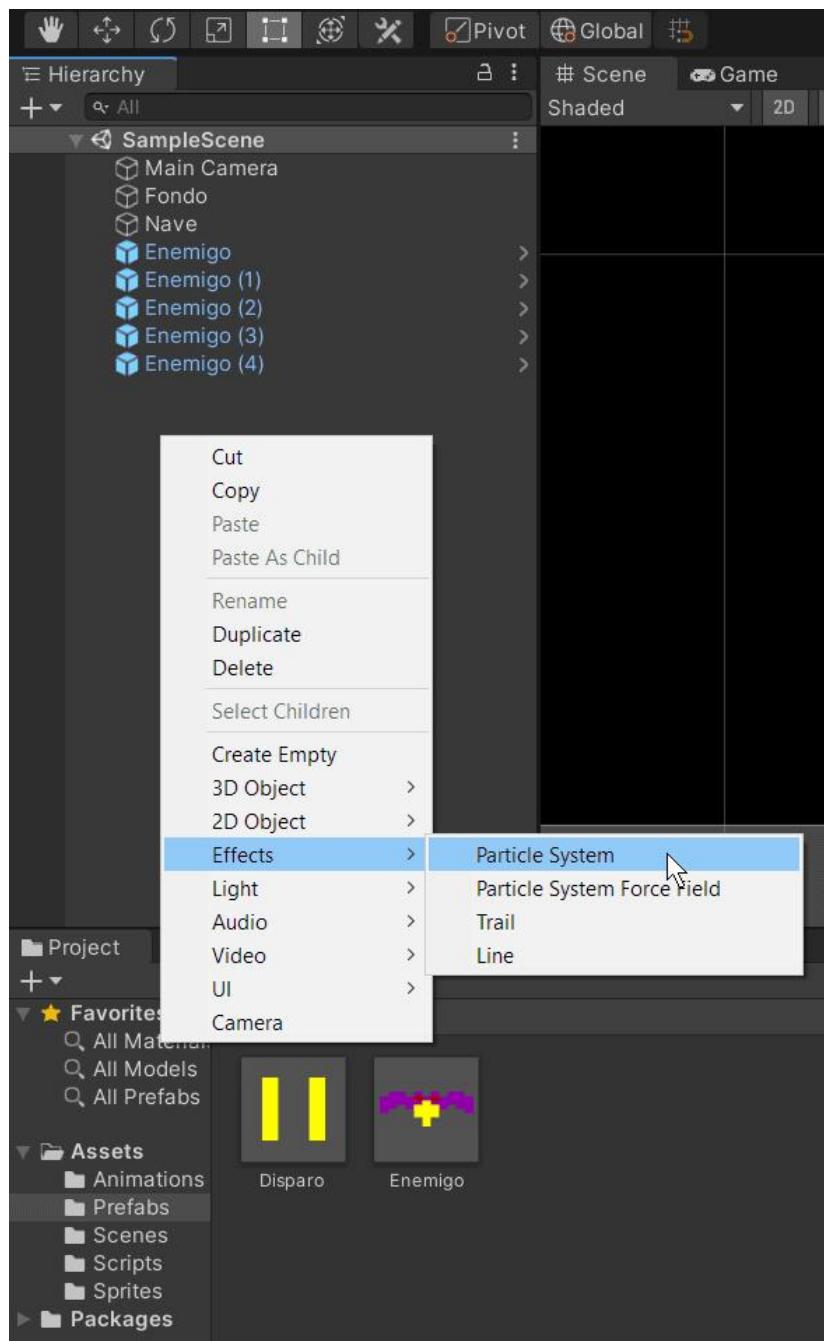
Ejercicio propuesto 1.22.1: Haz que tu disparo pueda destruir enemigos.

1.23. Una explosión usando sistemas de partículas

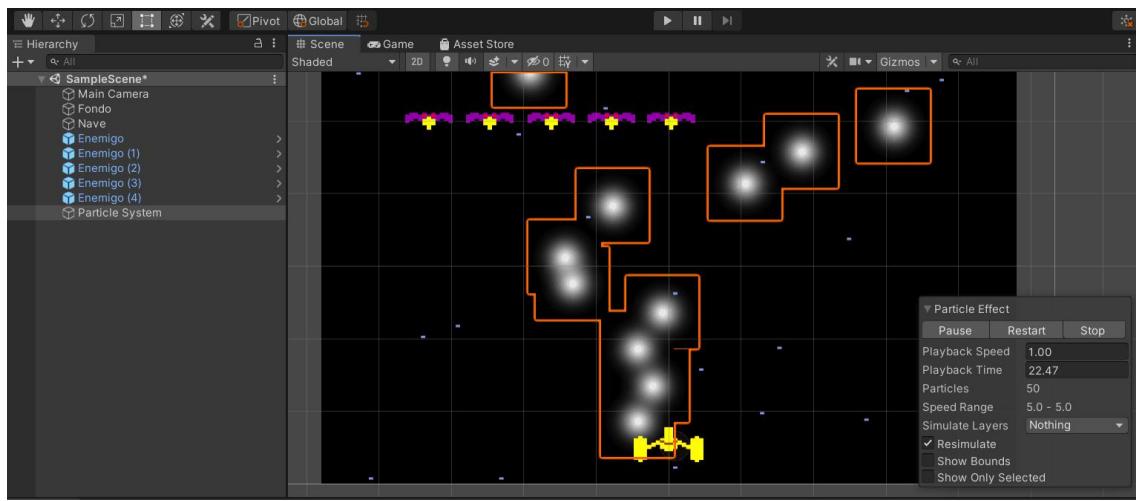
La forma “tradicional” de hacer una explosión sería cambiar el “sprite” del enemigo por otro (realmente, por una secuencia, formando una

animación) y no comprobar colisiones durante el tiempo que dure esa animación o la visualización de ese sprite alternativo.

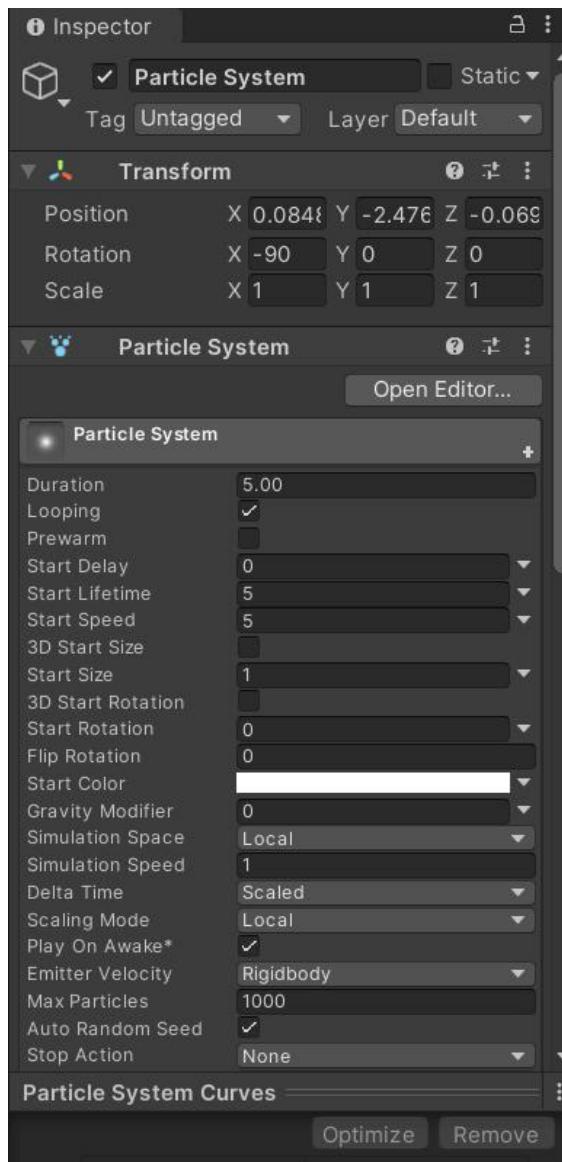
En Unity existe una alternativa interesante: crear un efecto visual empleando un “sistema de partículas”. Para ello deberemos pulsar el botón derecho en la jerarquía y escoger la opción Effects / Particle System:



En el centro de la pantalla aparecerán unas partículas circulares difuminadas, que se desplazan hacia arriba y desaparecen al cabo de un instante:

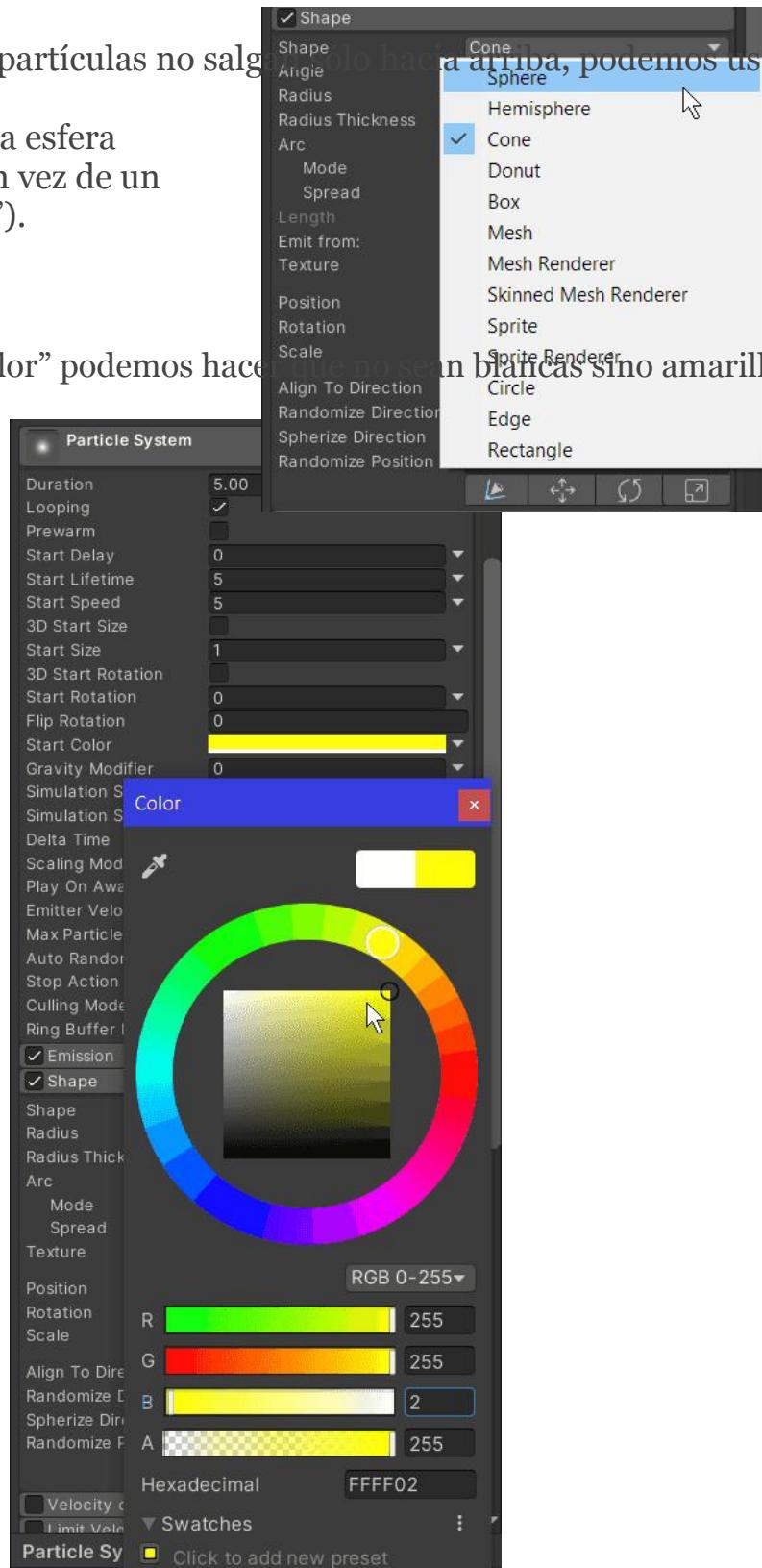


Como es habitual en Unity, podemos personalizar la apariencia y el comportamiento desde el panel derecho (Inspector), que muestra muchas opciones para los sistemas de partículas:

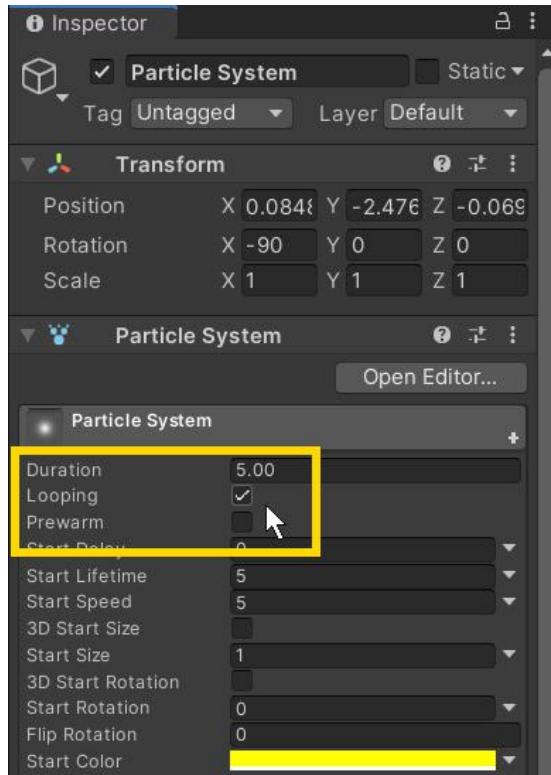


Podemos retocarlas para acercarlo al efecto que deseemos, por ejemplo:

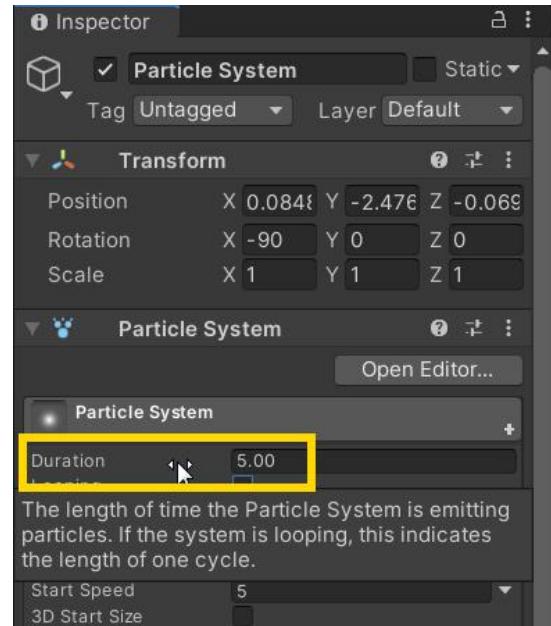
- Para que las partículas no salgan solo hacia arriba, podemos usar como forma (“Shape”) una esfera (“Sphere”) en vez de un cono (“Cone”).
- Con “Start color” podemos hacer que no sean blancas sino amarillas.



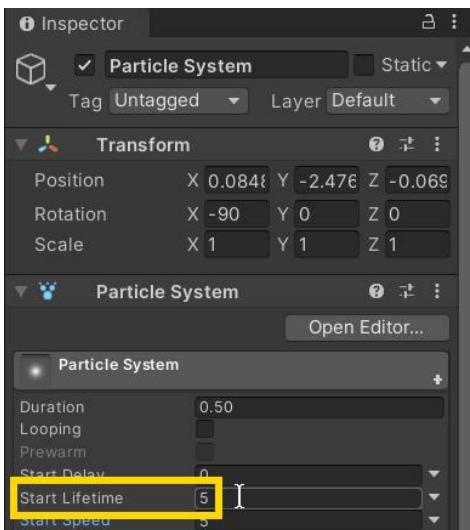
- Podemos hacer que no se repita (desactivar la casilla “Looping”).



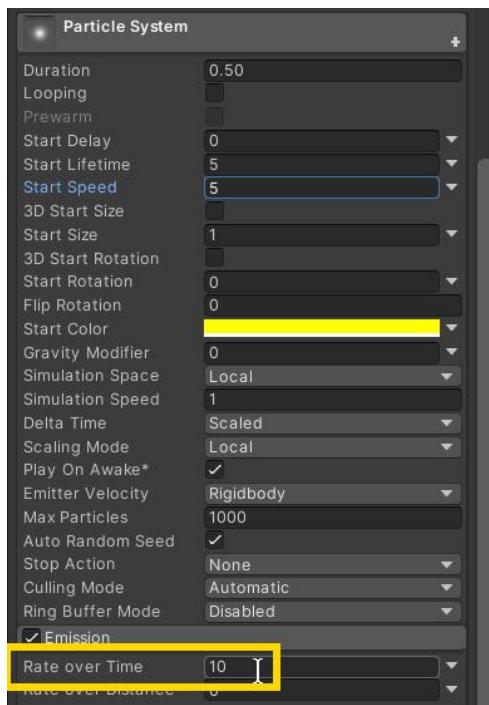
- La duración que se nos propone es muy larga: podemos rebajar el tiempo durante el que se crean las partículas (“Duration”) de 5 segundos a 1 segundo o incluso 0.5 segundos.



- Para que las partículas avancen menos, también podemos cambiar su tiempo de vida (“Start Lifetime”) de 5 segundos a apenas 0.2 segundos o menos.



- Para que salgan más partículas durante ese breve período de tiempo podemos ampliar, en el apartado “Emission”, la propiedad “Rate over time” para que aumente de 10 a un valor como 30.

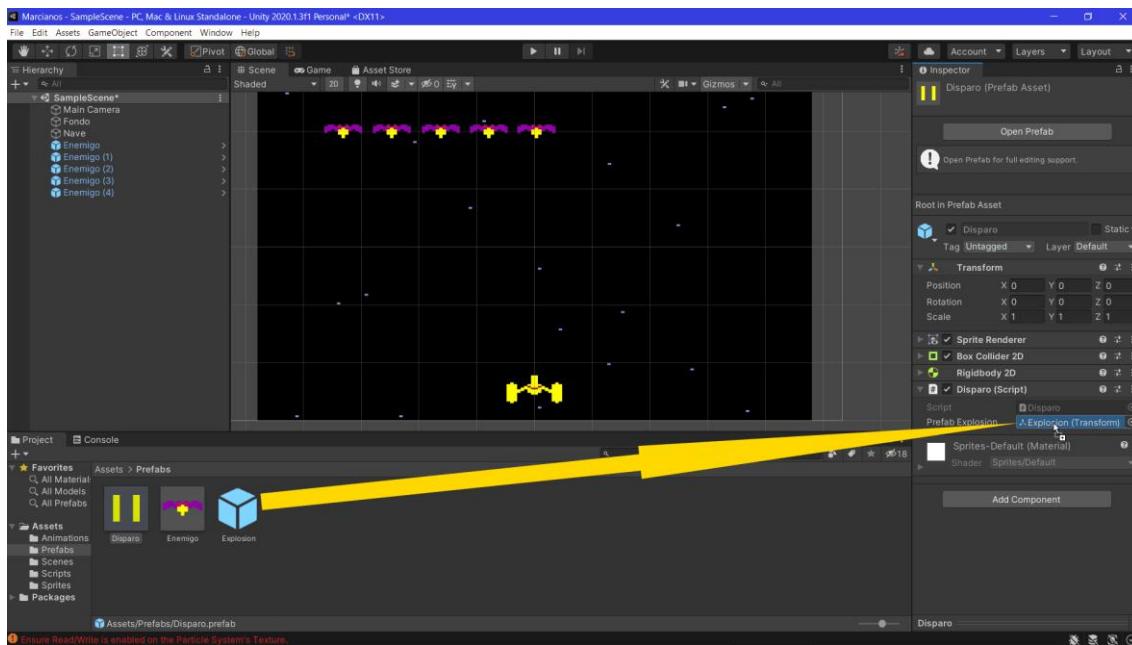


Sólo queda que se muestre ese sistema de partículas cuando deseemos. Para ello, haremos algo parecido a lo que habíamos preparado para el disparo: primero lo convertiremos en un “prefab” y luego **instanciaremos ese “prefab”** cuando el disparo impacte con un enemigo.

Podemos usar (por ejemplo) la propia clase Disparo, que es la que estaba destruyendo al enemigo mediante código, y añadirle un nuevo atributo accesible desde el editor:

```
[SerializeField] Transform prefabExplosion;
```

Arrastraríamos desde el “prefab” hasta la casilla que se acaba de crear para vincularlo



Y actualizaremos el evento “OnTriggerEnter2D” para que lance la explosión en la posición en que estaba el enemigo:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Enemigo")
    {
        Instantiate(prefabExplosion,
                    other.transform.position,
                    Quaternion.identity);
        Destroy(other.gameObject);
        Destroy(gameObject);
    }
}
```

De hecho, deberíamos también destruir el objeto de la explosión. Hay una variante de “Destroy” a la que se le puede indicar tras cuánto tiempo debe ocurrir, y en nuestro caso bastaría con un segundo, así:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.tag == "Enemigo")
    {
        Transform explosion =
            Instantiate(prefabExplosion,
                        other.transform.position,
                        Quaternion.identity);
```



```

        Destroy(other.gameObject);
        Destroy(explosion.gameObject, 1f);
        Destroy(gameObject);
    }
}

```

Ejercicio propuesto 1.23.1: Crea un efecto de explosión cuando el disparo toque a un enemigo.

1.24. Reproducir un sonido

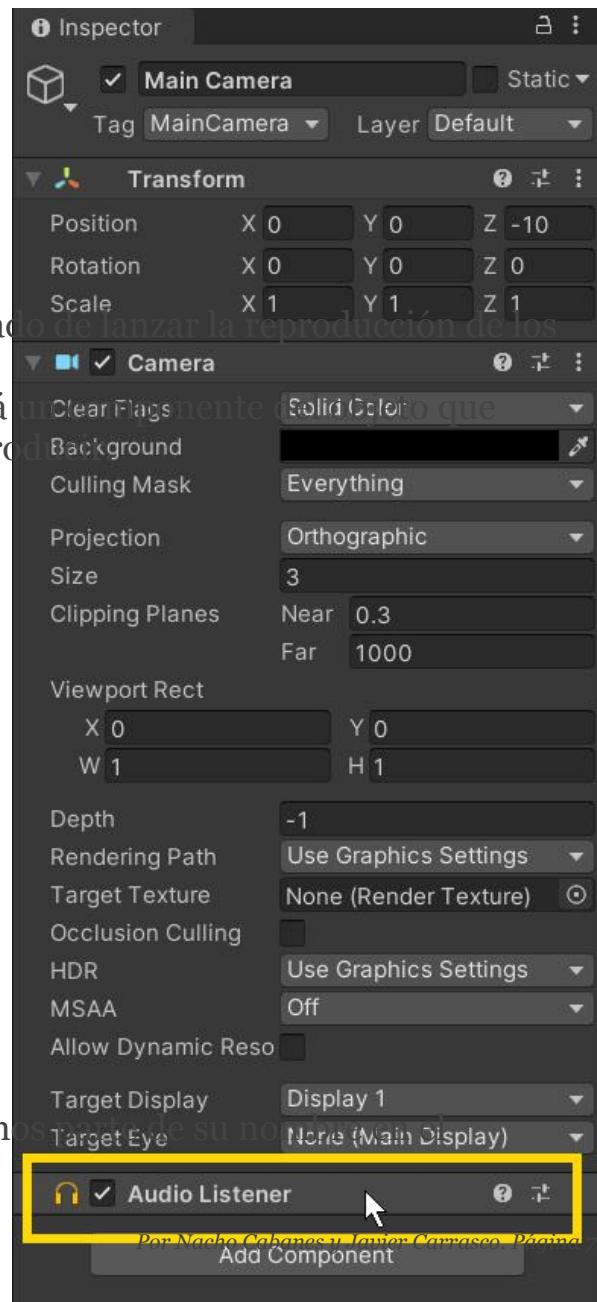
Hay varias formas de reproducir sonidos en Unity. Una de las más “naturales” es reproducir un sonido asociado a un objeto (Game object), y para ello necesitamos 3 cosas:

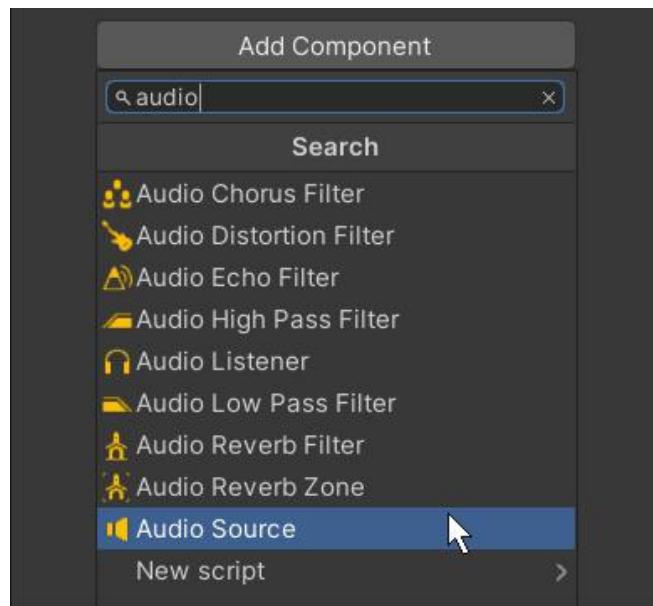
- Un “audio listener” encargado de lanzar la reproducción de los sonidos.
- Un “audio source”, que será un componente del objeto que reproduce el sonido.
- Uno o varios sonidos a reproducir.

El “audio listener” ya viene incorporado: hay uno en la cámara principal.

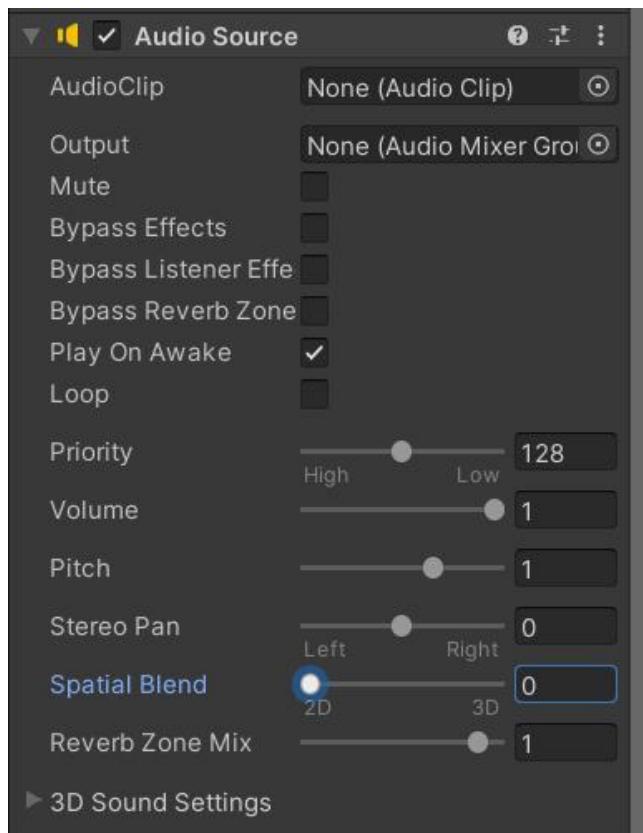
El “audio source” se añade a cada elemento del juego que nos interese. Por ejemplo, vamos a hacer que nuestra nave haga un sonido cuando dispare.

Usaremos el botón “Add component” del “Inspector” y buscaremos en la categoría correspondiente, o bien teclearemos en el buscador:

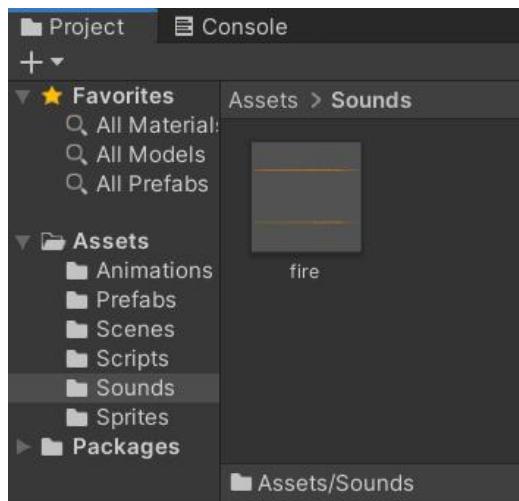




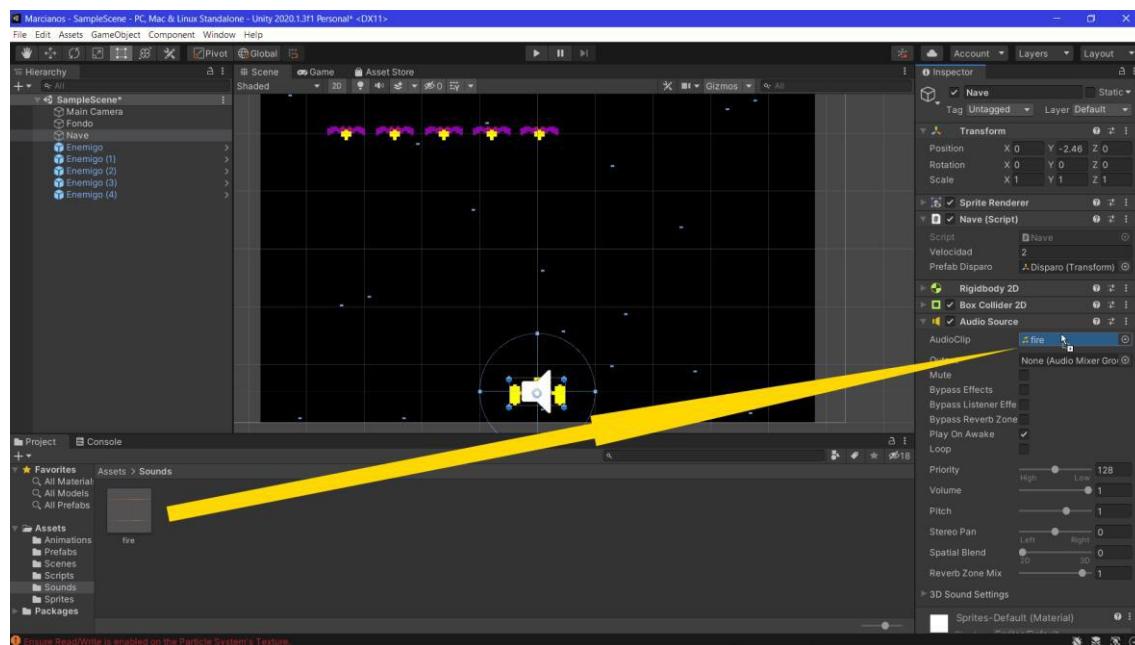
Y ese nuevo componente tendrá detalles como su “audio clip”, su volumen o si queremos reproducir la música en bucle (“loop”), silenciarla (“mute”):



Añadir el sonido es fácil: de forma similar a como hicimos con los “sprites”, creamos una carpeta “Sounds” (o “Sonidos”) dentro de “Assets” y arrastramos a ella los fragmentos de sonido que hayamos preparado o capturado.



Y luego arrastramos el sonido que nos interese desde esa carpeta “Sounds” al componente “AudioClip” de ese objeto:



Ya sólo queda añadir el fragmento de programa que hace que se reproduzca el sonido, que puede ser tan sencillo como llamar al método Play de ese “audio source” cuando se pulse el botón de disparo:

```
GetComponent<

```

De modo que el método Update de la Nave quedaría así:

```
void Update()
{
    float horizontal = Input.GetAxis("Horizontal");
    transform.Translate(horizontal * velocidad * Time.deltaTime,
    0, 0);

    if (Input.GetButtonDown("Fire1"))
    {
```

```

        GetComponent< AudioSource >().Play();
        Transform disparo =
            Instantiate(prefabDisparo,
            transform.position,
            Quaternion.identity);
        disparo.gameObject.GetComponent< Rigidbody2D >().velocity =
            new Vector3(0, velocidadDisparo, 0);
    }
}

```

Como curiosidad, asociar un sonido a un objeto dará problemas si se destruye ese objeto mientras se escucha el sonido, porque éste se interrumpirá bruscamente. Más adelante veremos otra forma de reproducir sonidos que no tiene ese problema, y que será más adecuada en algunos efectos de sonido como explosiones.

Ejercicio propuesto 1.24.1: Añade un sonido a algún suceso de tu juego, como cuando se mueva la nave, se dispare o cuando el enemigo rebote.

1.25. Enemigos que disparan: corrutinas

Sabemos la mayoría de los pasos necesarios para que los enemigos disparen:

- Crear o buscar un sprite que represente el disparo
- Usarlo para crear un GameObject a partir de él
- Ajustar su “altura” o su capa, para que quede por encima del fondo
- Añadirle un Collider y, según el caso, un RigidBody
- Convertirlo en un “prefab”
- Instanciar ese “prefab” cuando queramos que aparezca el disparo
- Añadir velocidad a ese disparo después de instanciarlo
- Comprobar si colisiona con nuestra nave

Pero hay un par de detalles que no sabemos:

- Cómo llevar cuenta de las vidas restantes cuando un disparo del enemigo nos impacte, así como los puntos obtenidos al disparar nosotros a un enemigo, etc. Bastaría con crear una clase adicional que represente la lógica del juego, pero eso es algo que aplazaremos para el próximo tema.
- Cómo hacer que disparen cada cierto tiempo al azar, y esa va a ser la novedad de este apartado.

Una primera aproximación podría ser: dentro del método Update, generamos un número al azar. Si ese número es mayor que un cierto valor, creamos el disparo. El problema de este planteamiento es que si tenemos 20 enemigos y el juego va a 50 fotogramas por segundo, estaríamos generando 1000 números al azar cada segundo, incluso si queremos que se dispare apenas una vez cada 3 o 5 segundos.

Un planteamiento menos costoso sería generar un número al azar que represente el tiempo que hay que esperar hasta que salga el (siguiente) disparo, y en cada llamada a Update sumar el valor de Time.deltaTime para ir contando tiempo hasta alcanzar ese momento previsto, en el que lanzaremos un cierto método.

Una alternativa similar pero más elegante es generar ese número al azar para el tiempo que hay que esperar, y entonces poner en marcha un temporizador que lance un cierto método cuando ese tiempo transcurra.

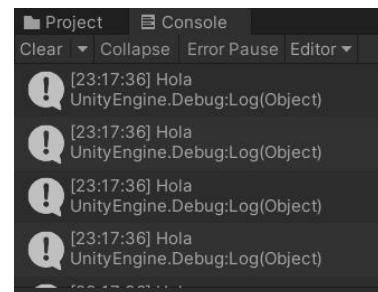
Para que Unity ponga en marcha una función al cabo de un cierto tiempo, deberemos seguir ciertos pasos:

- Llamar a la función con “StartCoroutine”
- La función no debe ser de tipo “void” sino de tipo “IEnumerator”.
- La función debe usar “yield return new WaitForSeconds(n);” para hacer las pausas.

Como es un tanto enrevesado, vamos a hacer una primera aproximación: que aparezca el mensaje “Hola” en la consola de depuración, tres segundos después de que se cree un enemigo. Habrá que hacer los siguientes cambios en la clase Enemigo:

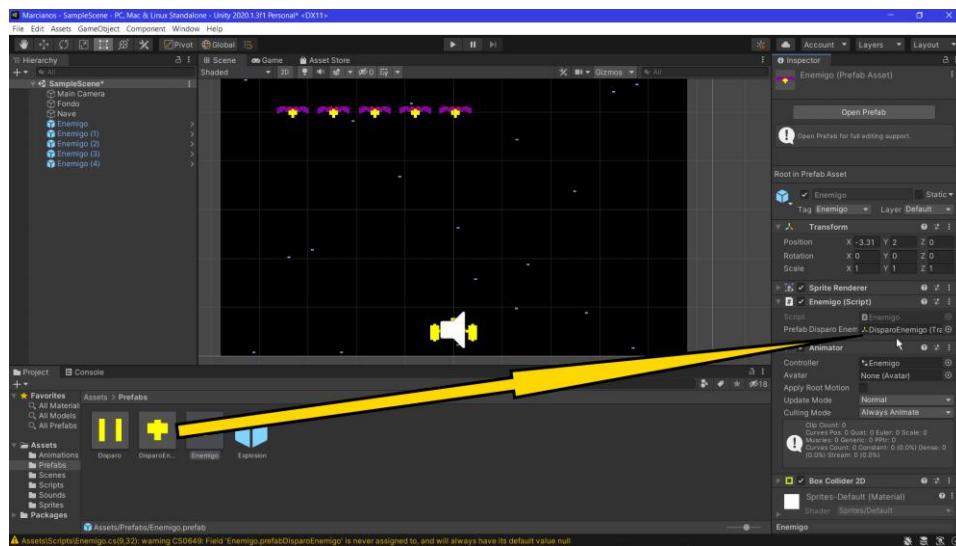
```
void Start()
{
    StartCoroutine( Disparar() );
}

IEnumerator Disparar()
{
    yield return new
    WaitForSeconds(3);
    Debug.Log("Hola");
}
```



Hacer que aparezca un disparo cada cierto tiempo no es mucho más difícil que eso:

- La parte visual ya la conocemos, porque es similar al disparo de nuestra nave: deberemos crear un objeto a partir del sprite, asignarle un RigidBody2D (con escala de gravedad 0) para poder aplicarle velocidad, crear el prefab para dicho disparo, añadirle un script que haga que desaparezca cuando salga de la pantalla...



- A nivel de programa, dentro del script del enemigo, tendremos generar un número al azar entre 2 valores, esperar esa cantidad de segundos e instanciar el “prefab” del disparo del enemigo (y volver a llamar a la misma función para el siguiente disparo):

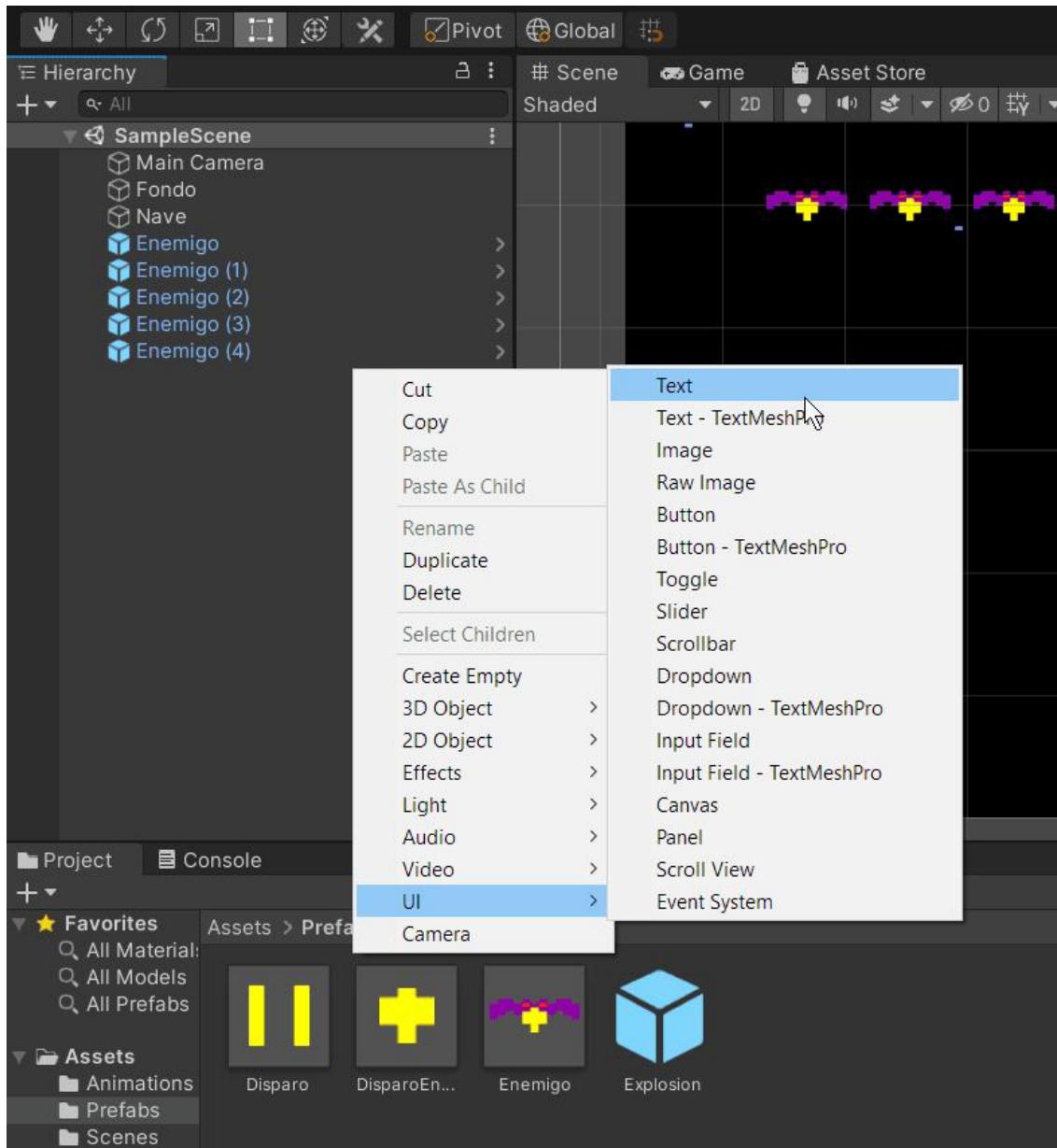
```
IEnumerator Disparar()
{
    float pausa = Random.Range(5.0f, 11.0f);
    yield return new WaitForSeconds(pausa);
    Transform disparo =
        Instantiate(prefabDisparoEnemigo,
        transform.position, Quaternion.identity);
    disparo.gameObject.GetComponent<Rigidbody2D>().veloc
        ity = new Vector3(0, velocidadDisparo, 0);
    StartCoroutine( Disparar() );
}
```

Ejercicio propuesto 1.25.1: Completa ese esqueleto para que se compruebe colisiones entre el disparo enemigo y nuestra nave. Haz que los disparos desaparezcan al cabo de un cierto tiempo o al llegar a una cierta posición.

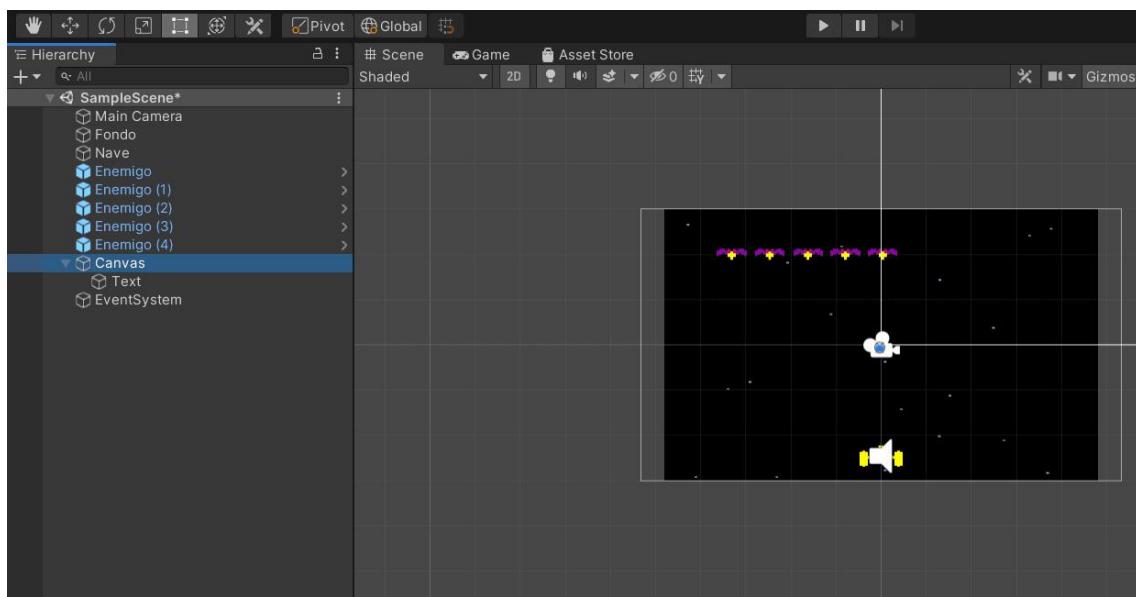
1.26. Un interfaz de usuario con texto

Para añadir un texto (contador de puntos, de vidas, información sobre el nivel, etc.), tenemos a nuestra disposición las herramientas de

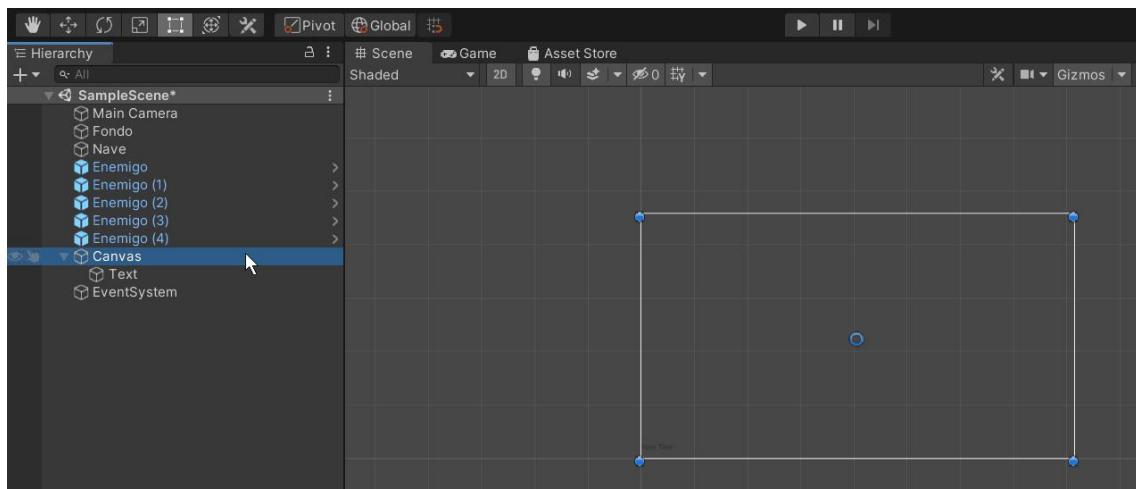
creación de interfaces de usuario (**UI**), dentro de la jerarquía. Entre ellas, por ahora nos interesará crear un texto:



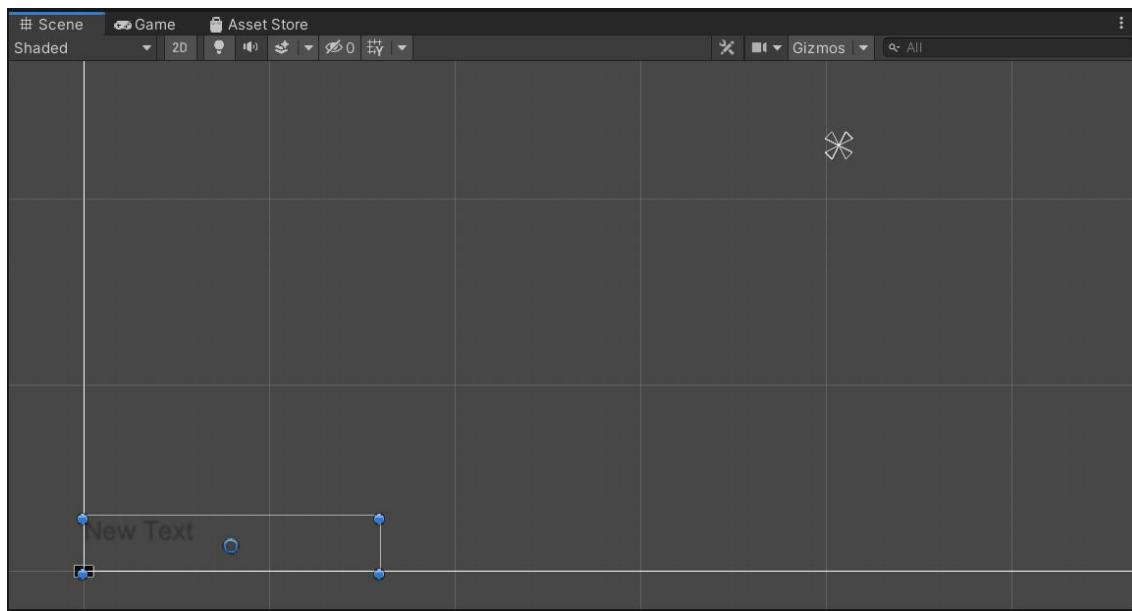
Veremos que al hacerlo aparece también un “Canvas” y un “EventSystem”. Este Canvas actuará como contenedor de todo el interfaz de usuario, y habitualmente aparecerá “descolocado” con relación a los sprites en la fase de diseño (pero sí se vera centrado en el momento de lanzar el juego). Típicamente será un rectángulo mucho mayor que nuestra escena, y que saldrá hacia la derecha y hacia arriba desde el centro de la escena:



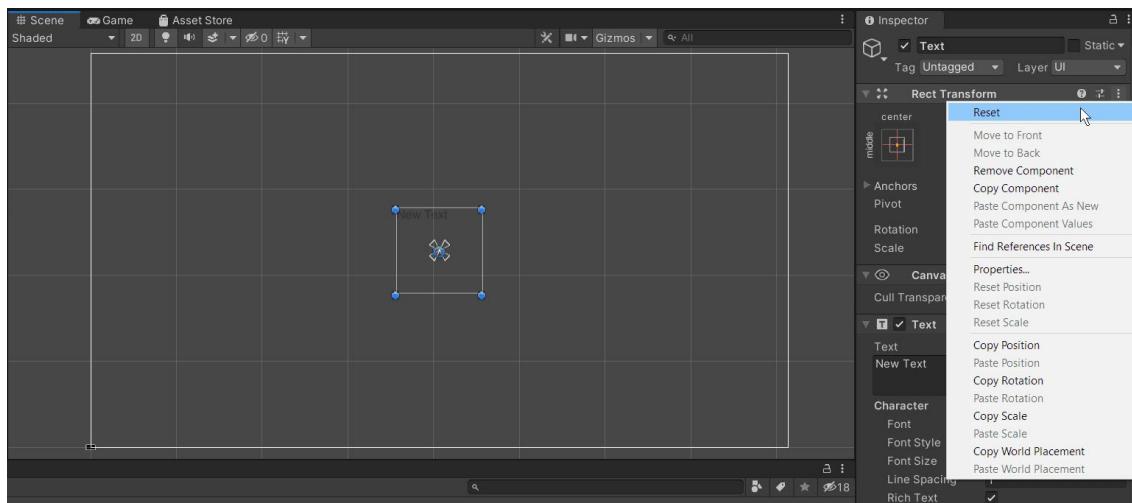
Para poder ver todo el “Canvas”, podemos hacer doble clic sobre él en la jerarquía (y entonces será la “escena” del juego la que se verá muy pequeña en una esquina):



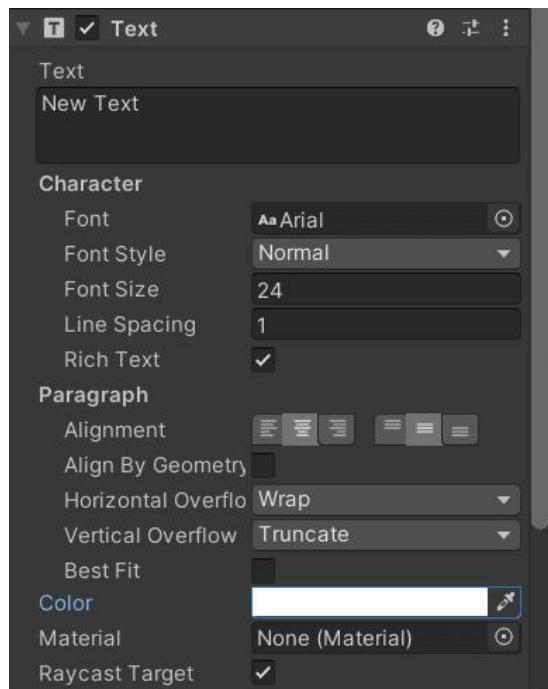
Es posible que el texto no aparezca centrado y que tenga un color oscuro, que no se vea bien sobre nuestro fondo:



Para que quede centrado en el Canvas, como ya sabemos, podemos “resetear” su componente “Transform:”



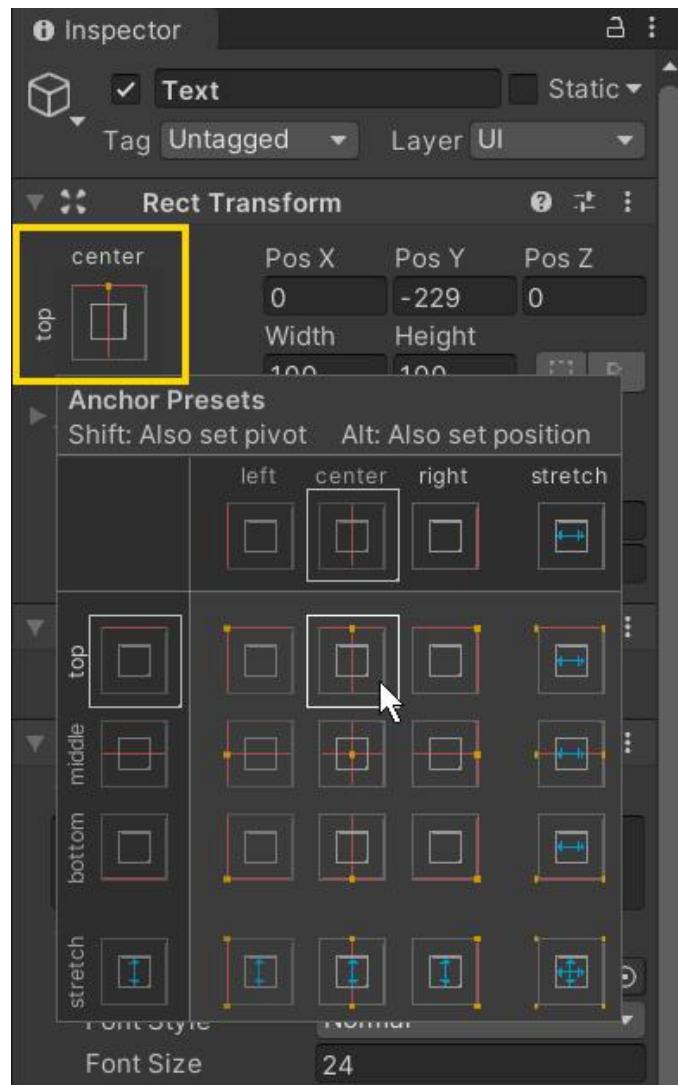
Y a continuación podemos cambiar su tamaño (por ejemplo, 24), su alineación dentro de su recuadro (centrado) y su color (blanco, al ser oscuro nuestro fondo):



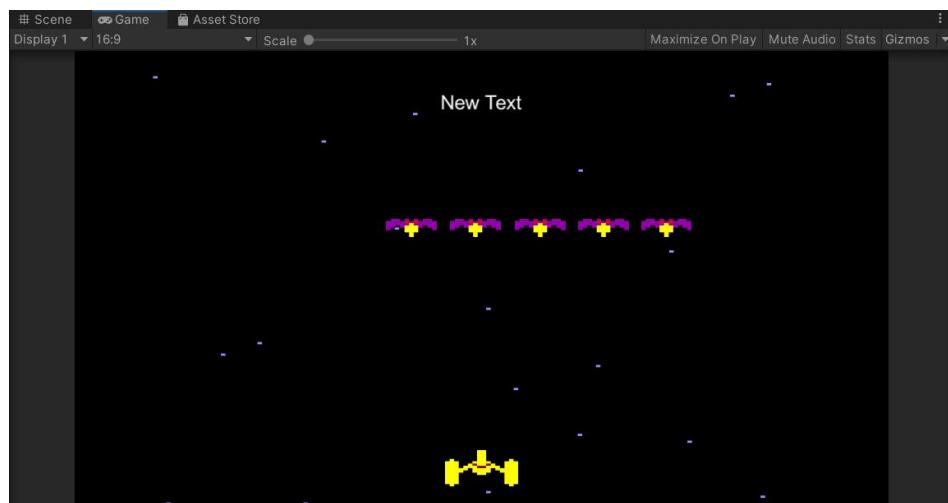
Si queremos cambiar su posición, podemos usar la herramienta de Mover, y desplazarlo con la flecha vertical:



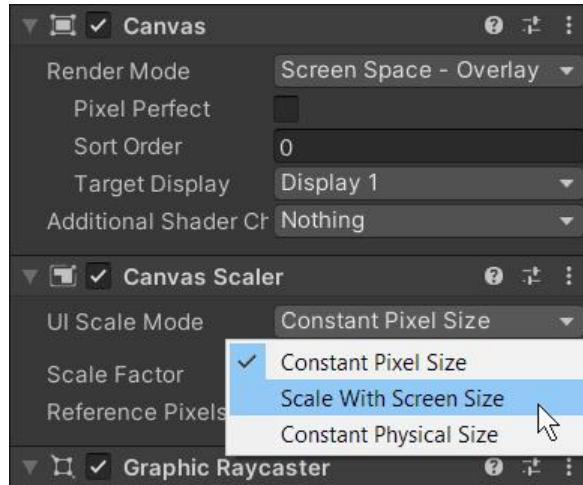
Y, en general, será interesante pensar dónde querremos que “se ancle” ese texto, para prever el comportamiento en pantallas con una proporción distinta a la que estamos usando para diseñar el juego. Lo haremos cambiando su “anchor”, en el apartado de “Rect Transforms”. Por ejemplo, en nuestro caso, podría estar anclado a la parte superior central de la pantalla:



Aun así, podemos encontrar un detalle que quizá no nos guste con los valores por defecto del texto de un Canvas: es probable que el texto se considere “de tamaño constante”, por lo que puede verse comparativamente más pequeño si la pantalla es más grande:



Como alternativa, podemos pedir que “el Canvas” se “escale con el tamaño de pantalla”, con lo que se verá igual (salvo los márgenes laterales) en cualquier pantalla:



Para **modificar ese texto desde código** (por ejemplo, si necesitáramos mostrar una puntuación), deberemos **asociarlo** a un objeto (game object). Como no hemos creado un objeto que represente a todo el juego y que tenga un script que centralice la lógica que es común al juego, podríamos hacerlo desde el objeto Nave.

Comenzaremos por crear en ese objeto un atributo que sea de tipo “UnityEngine.UI.Text”. Si este atributo es **público**, lo podremos modificar también desde el editor de Unity y desde otras clases. Si no queremos modificarlo desde otras clases, bastaría con [SerializeField]:

```
public UnityEngine.UI.Text textoSaludo;
```

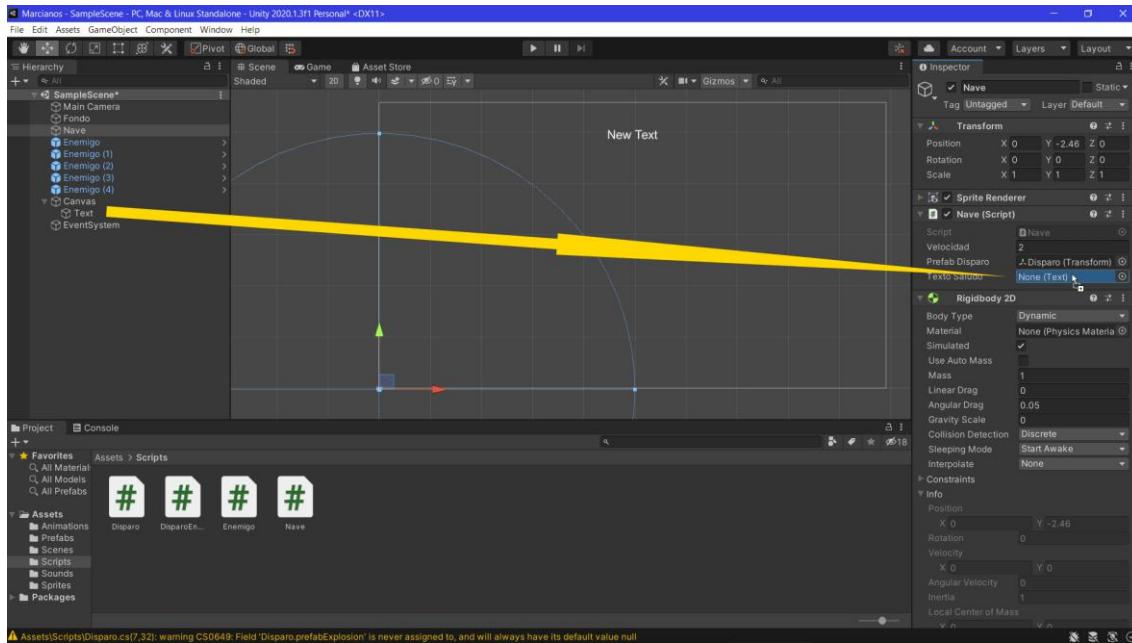
Como alternativa, podemos añadir la directiva “using UnityEngine.UI;” y entonces declarar ese atributo como de tipo “Text” (lo que no necesariamente resultará más legible):

```
using UnityEngine.UI;  
// ...  
public Text textoSaludo;
```

Para cambiar el texto que se muestre en ese elemento del interfaz (dentro de poco, porque aún no está enlazado con ningún elemento real del interfaz) usaríamos su propiedad “text”, así:

```
if (Input.GetButtonDown("Fire1"))  
{  
    textoSaludo.text = "Hola";  
    // ...  
}
```

Sólo nos queda enlazar el elemento de texto que habíamos creado en el interfaz de usuario con ese nuevo atributo público que tenemos en el script Nave, arrastrándolo hasta la correspondiente casilla del inspector:



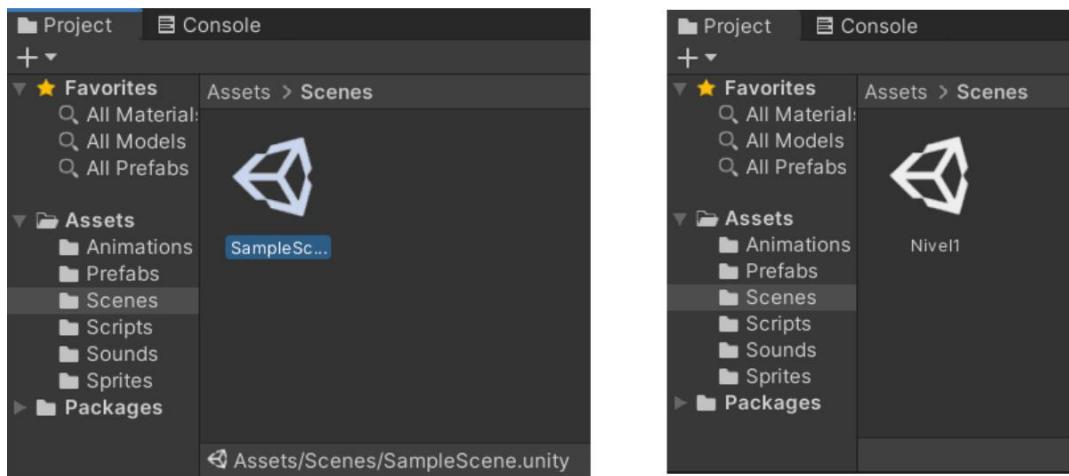
Ahora nuestro juego contiene un texto “New text” que se convierte en un “Hola” cuando pulsamos el botón de disparo. Obviamente, en un juego real, se trataría de un marcador con los puntos que hemos obtenido, las vidas restantes, etc., pero esas mejoras quedarán propuestas como ejercicio.

Ejercicio propuesto 1.26.1: Añade un texto estático a tu esqueleto de juego. Elige su tamaño y color. Asócialo a un elemento del juego. Cambia el texto desde el programa.

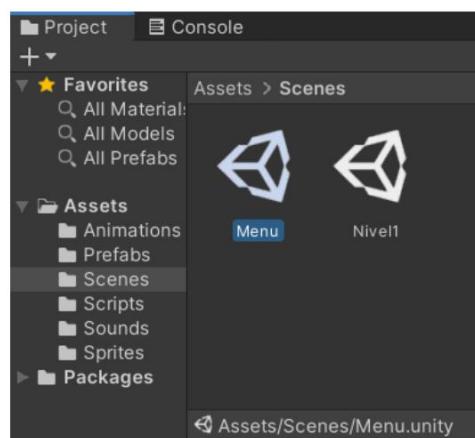
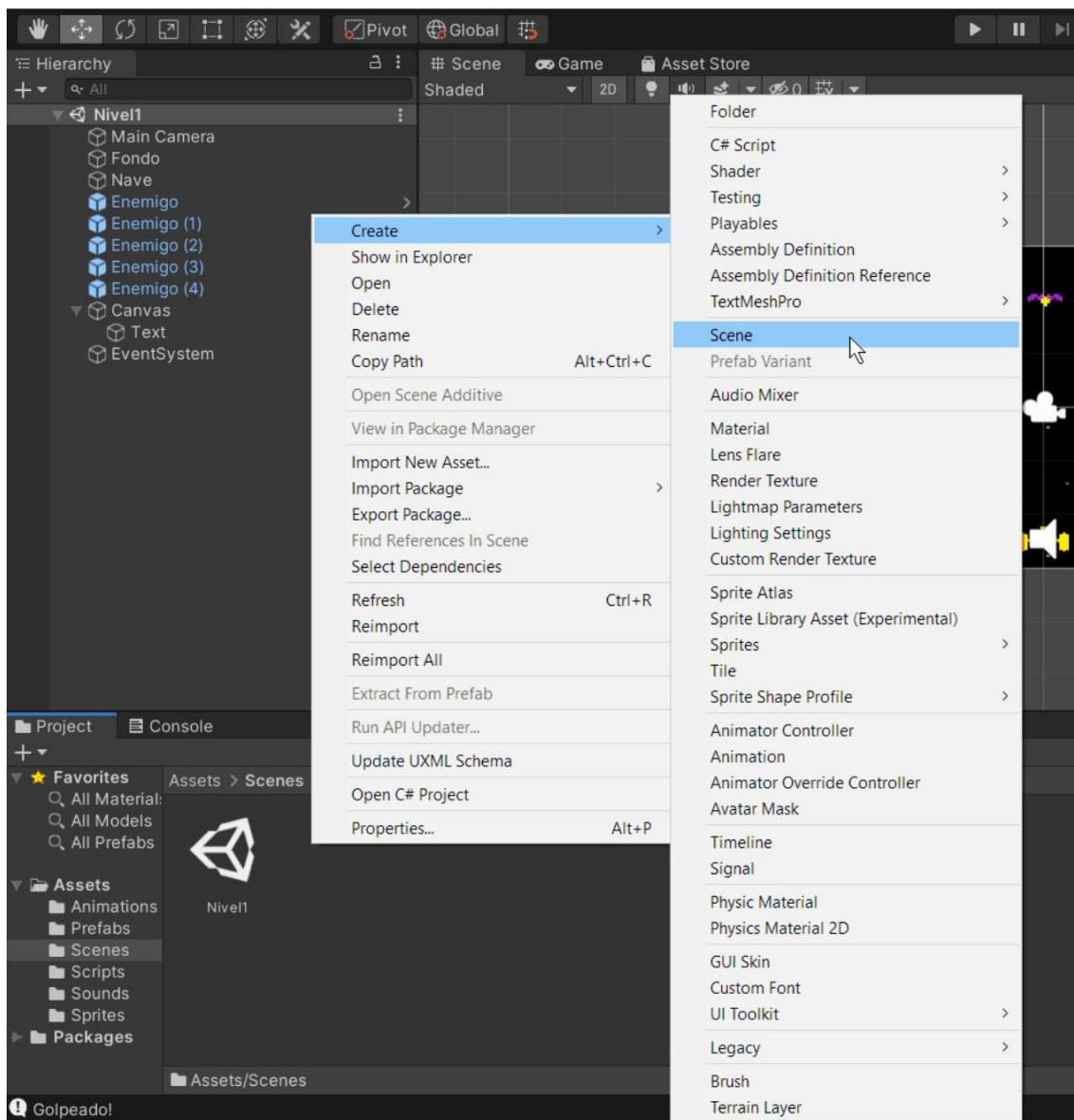
1.27. Una pantalla de bienvenida

Unity permite que un juego esté formado por múltiples “escenas”, cada una de las cuales puede representar un nivel del juego, o bien pantallas adicionales, como una de menú, o de créditos, o de ayuda.

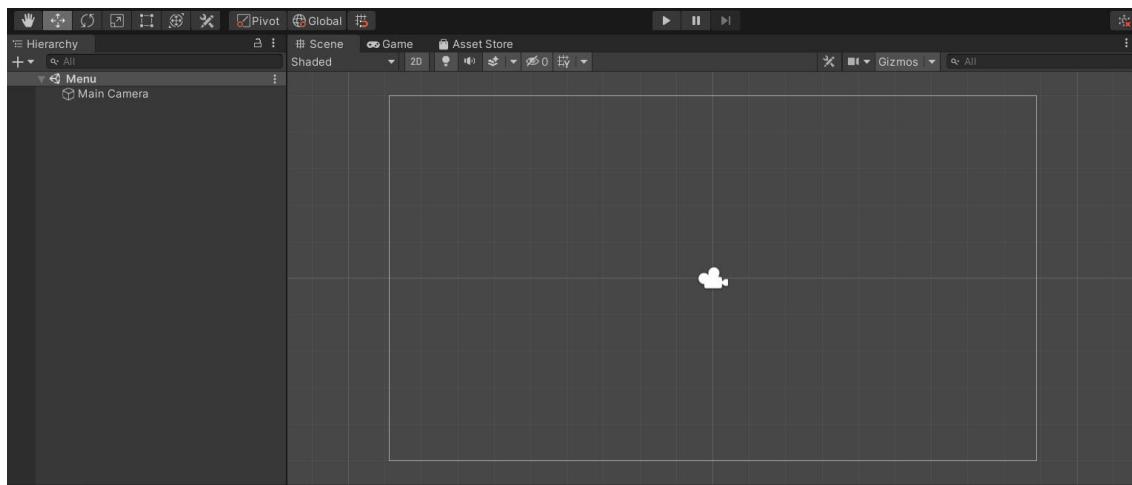
Por eso, en nuestro caso, comenzaremos por cambiar el nombre de la que hasta ahora era nuestra única escena, para que no se llame “SampleScene” sino “Nivel1”, en el panel del proyecto (**cuidado: asegúrate de haber guardado los cambios antes**):



Y después creamos una nueva pulsando el botón derecho del ratón en el panel inferior y eligiendo “Create / Scene” y un nombre, como “Menu”:

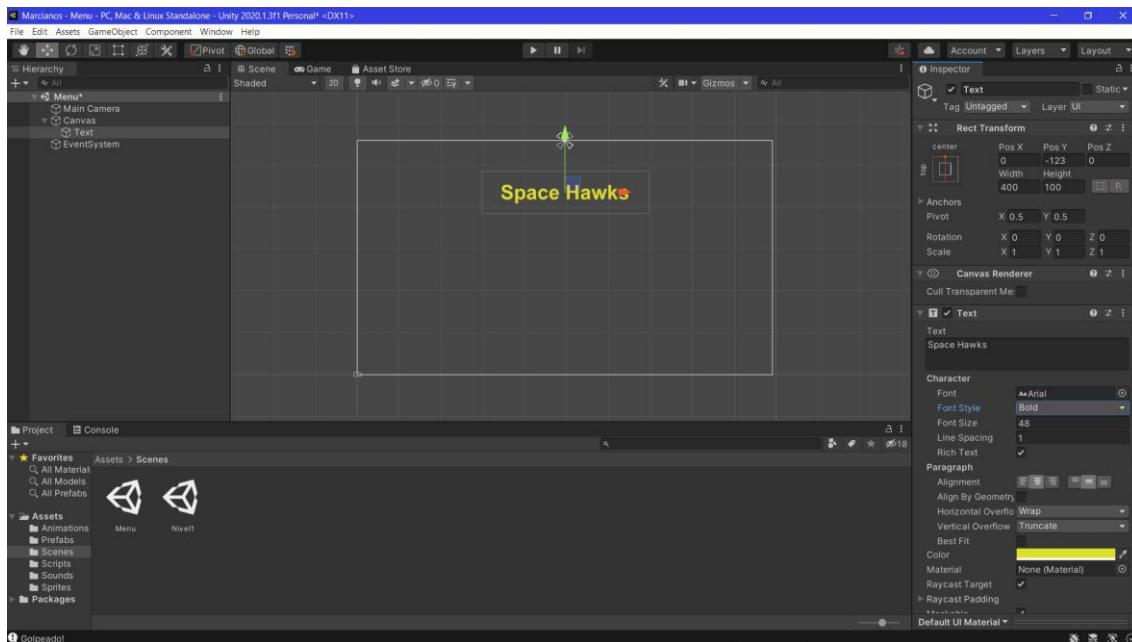


Al hacer doble clic en ella, veremos que ahora la jerarquía está vacía:



Vamos a añadir un texto y un botón. El texto es algo que ya conocemos:

- Pulsamos el botón derecho en la jerarquía, UI / Text, y aparecerá un “canvas” vacío y descolocado con relación a lo que era hasta ahora la escena (vacía). Centramos el texto, elegimos su color, tamaño y alineación y fuente, cambiamos su Text para que sea “Space Hawks” y podemos cambiar también el tamaño del recuadro en el apartado “Rect transform”:

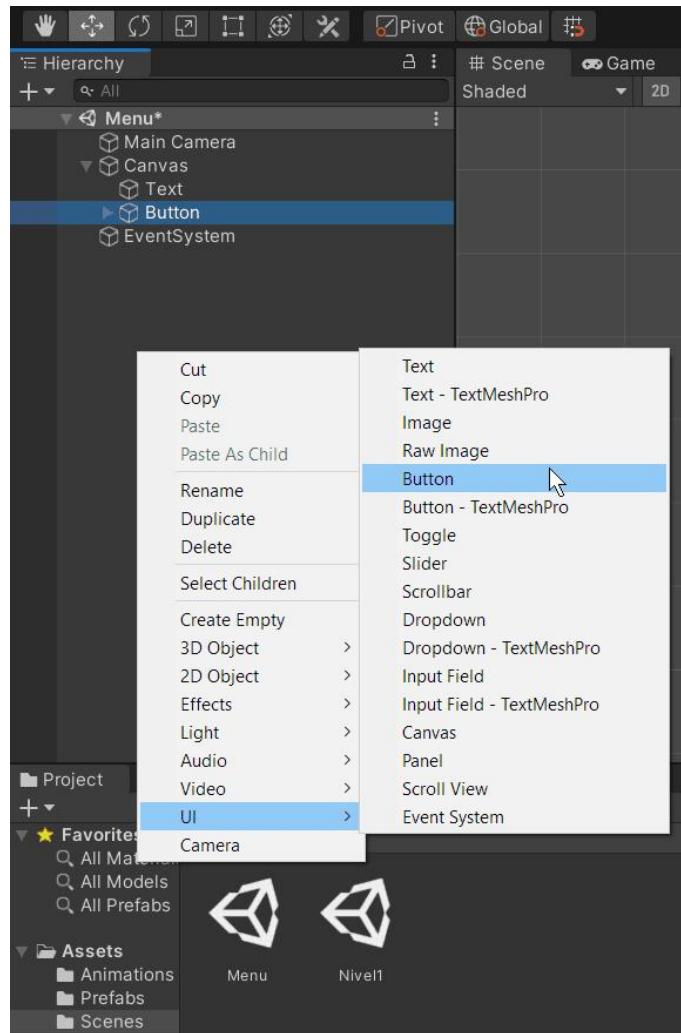


(Más adelante veremos cómo cambiar el tipo de letra).

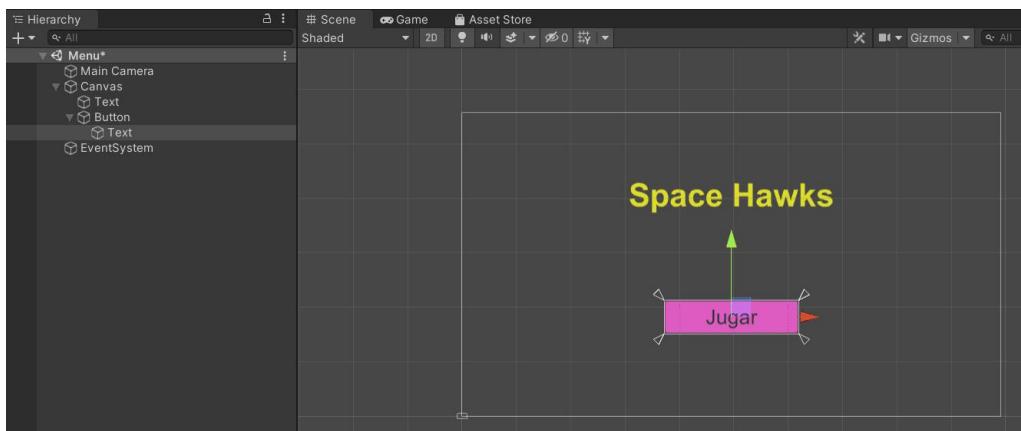
- Según nuestros gustos, indicamos en el Canvas que los píxeles sean de tamaño constante o que se escalen.

Para añadir el botón, los primeros pasos serán básicamente los mismos:

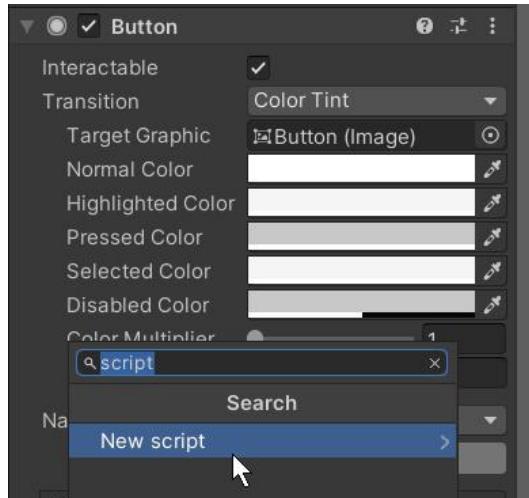
- Pulsar el botón derecho en la jerarquía, UI / Button, y aparecerá un botón con la apariencia por defecto:



- Ajustamos su posición, tamaño y color. El texto que muestra el botón aparece como “objeto hijo” del éste, y deberemos afinar también sus propiedades:



Ya sólo queda asociar el evento “click” de ese botón. Como primer paso, podemos crear un script asociado al Botón, llamado (por ejemplo) Menu:



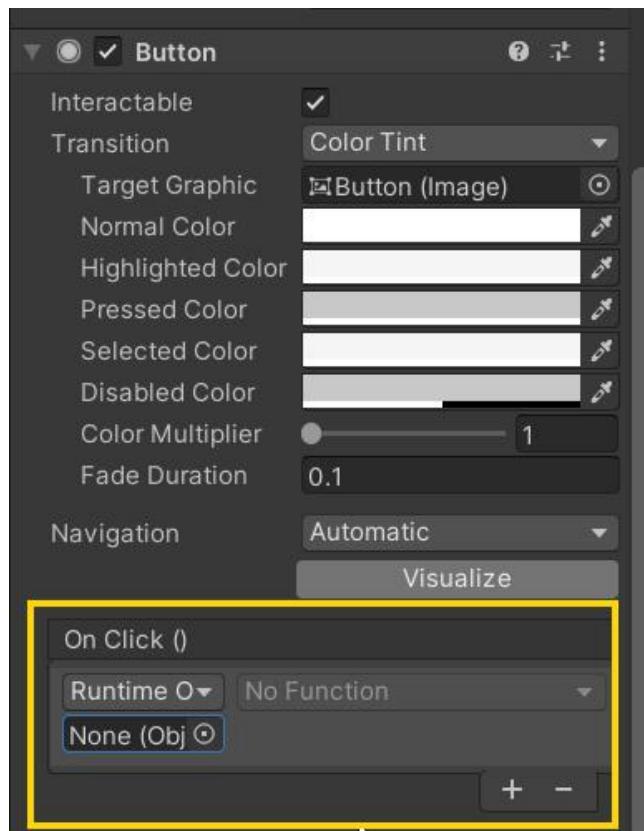
E incluir en él un método como éste:

```
public void LanzarJuego()
{
    SceneManager.LoadScene("Nivel1");
}
```

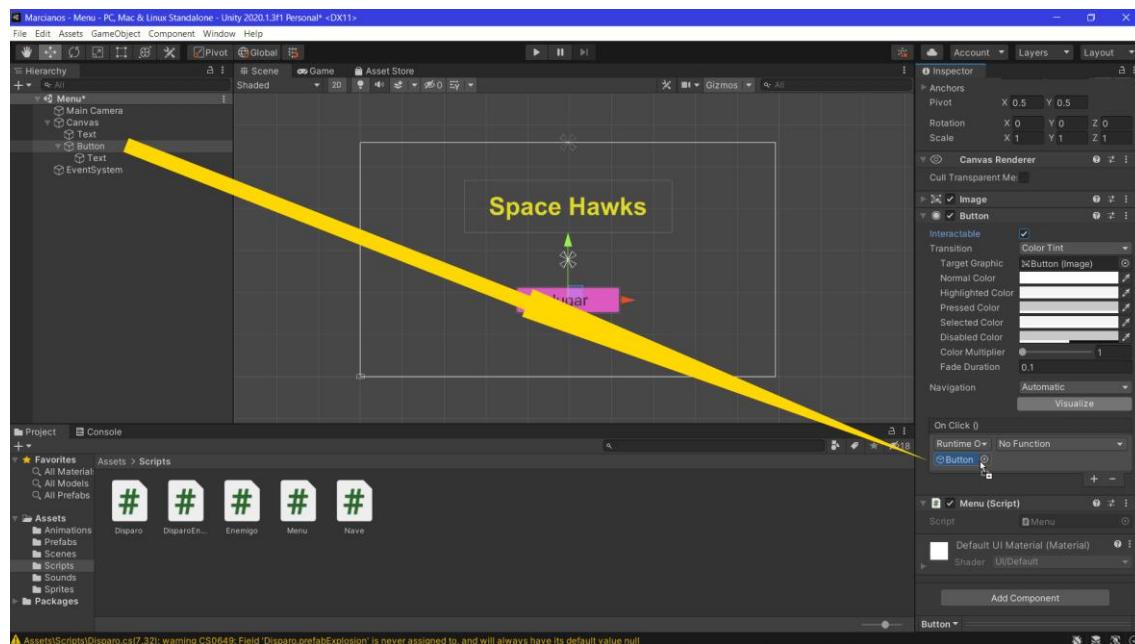
Y necesitaremos añadir un “using”:

```
using UnityEngine.SceneManagement;
```

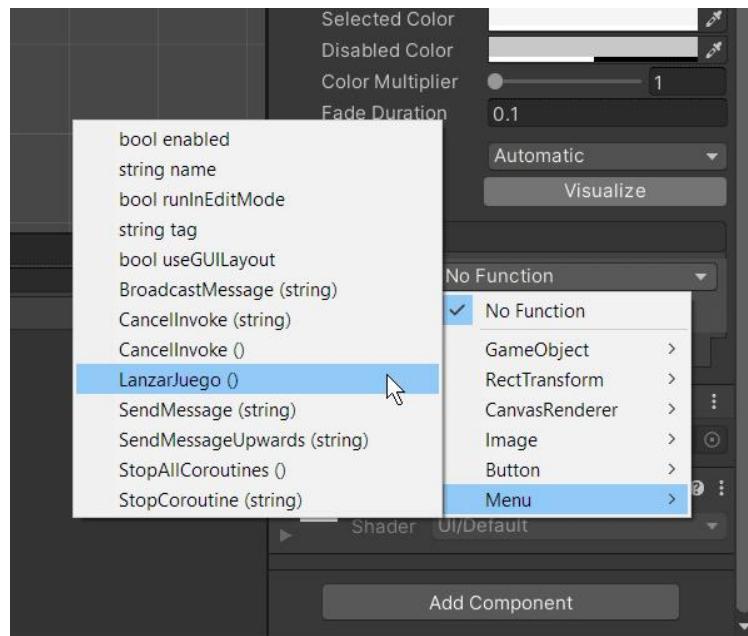
Como segundo paso, al seleccionar el botón, aparecerá en el Inspector un apartado llamado “On Click”, que contiene un símbolo “+” para añadir comportamientos:



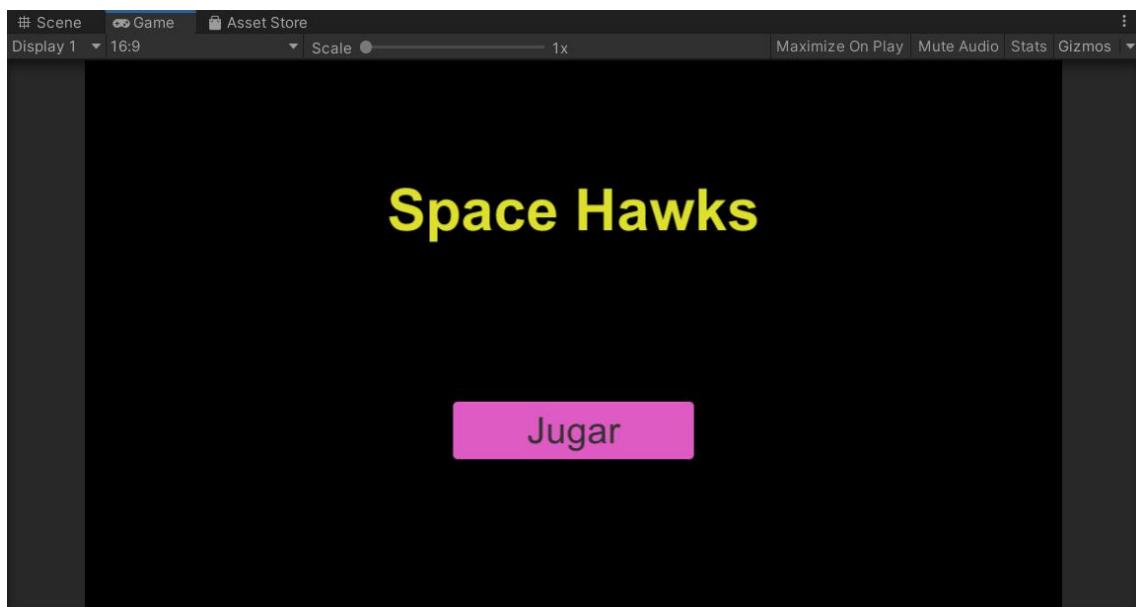
Aparece un apartado “Runtime object” en el que deberemos indicarle en qué objeto tiene que buscar las funciones que se puedan asociar a ese evento OnClick. Arrastraremos el botón desde la jerarquía:



Y ya podremos recorrer la lista de funciones disponibles, entre las que debería aparecer nuestra “LanzarJuego”:



y ya podremos probar el menú

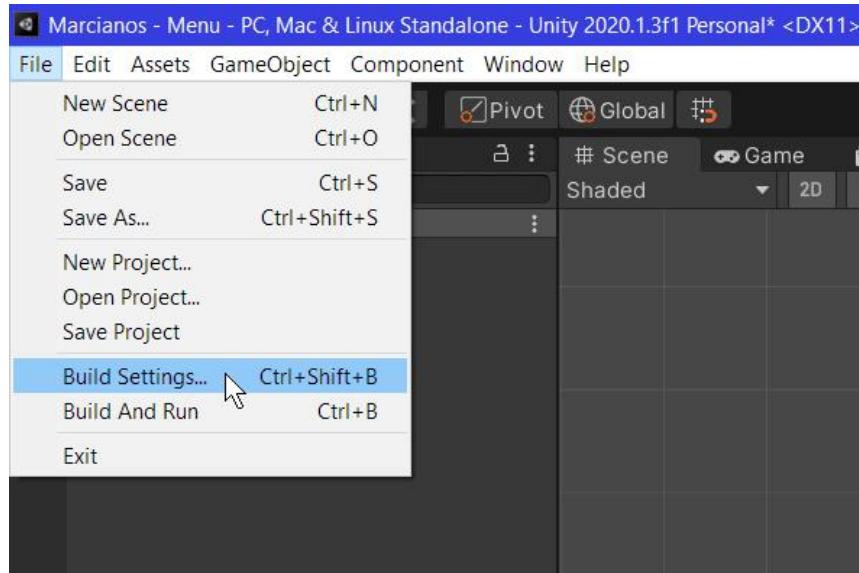


Ejercicio propuesto 1.27.1: Añade un menú a tu juego, que aparezca antes que el juego en sí.

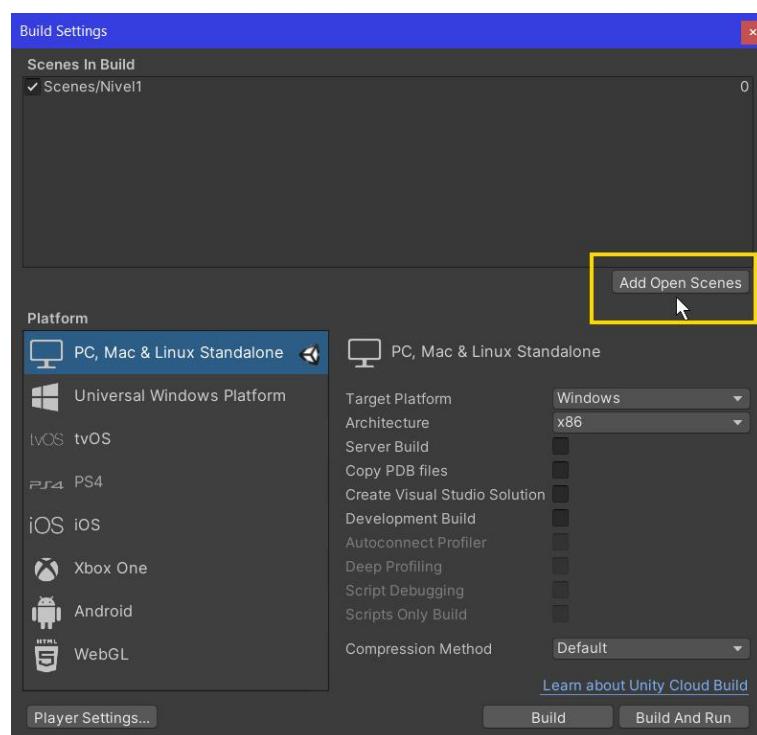
1.28. Creando un ejecutable

A medida que avanzamos el juego, va habiendo cada vez más ocasiones en las que querremos (o incluso necesitaremos) que lo pruebe otra persona. Si esa persona no tiene Unity instalado, la alternativa será crear un ejecutable que le podamos dar. El primer paso es configurar la

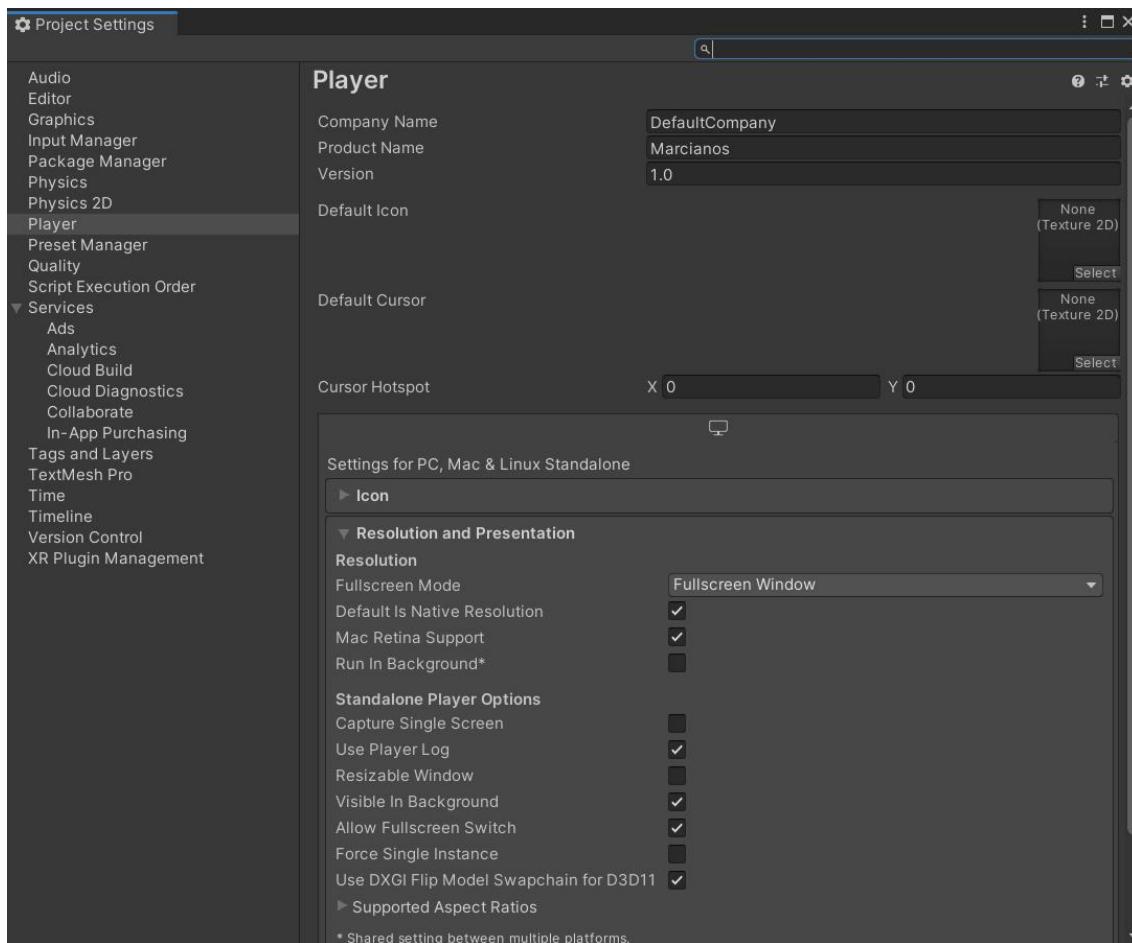
construcción de ese ejecutable, desde “Build Settings”, en el menú “File”:



Desde ahí podremos indicar para qué plataforma queremos generarlo (nosotros usaremos Windows por ahora), y otro detalle importante: las “escenas” que queremos añadir. Es habitual que se nos proponga la primera que habíamos creado, pero no todas ellas si, como en nuestro caso hemos creado varias. Para incluir todas ellas de una forma rápida, usaremos el botón “Add open scenes”:



En el botón “Player Settings” tenemos accesibles otros detalles más avanzados como el ícono del juego, si se debe iniciar a ventana completa o si se permite redimensionar, en caso de de lanzarse en ventana:



En la misma ventana de “Build Settings” tenemos un botón “Build and Run” para construir y lanzar el juego. En momentos posteriores, lo podremos hacer desde el propio menú File.

Nota: al construir se te preguntará dónde lo quieres guardar. Es habitual crear (por ejemplo) una carpeta “Build” dentro del proyecto. Si lo haces así, generalmente será deseable no incluir tampoco esa carpeta en las copias de seguridad ni en el control de versiones.

Ejercicio propuesto 1.28.1: Genera el ejecutable de tu juego y comprueba que funciona correctamente.

1.29. Los avisos de error: sincronizar Unity con Visual Studio

Por una parte, podemos mostrar información de depuración para comprobar que una cierta variable tiene el valor que esperamos o que se pasa por una cierta zona del código. Lo que escribamos con

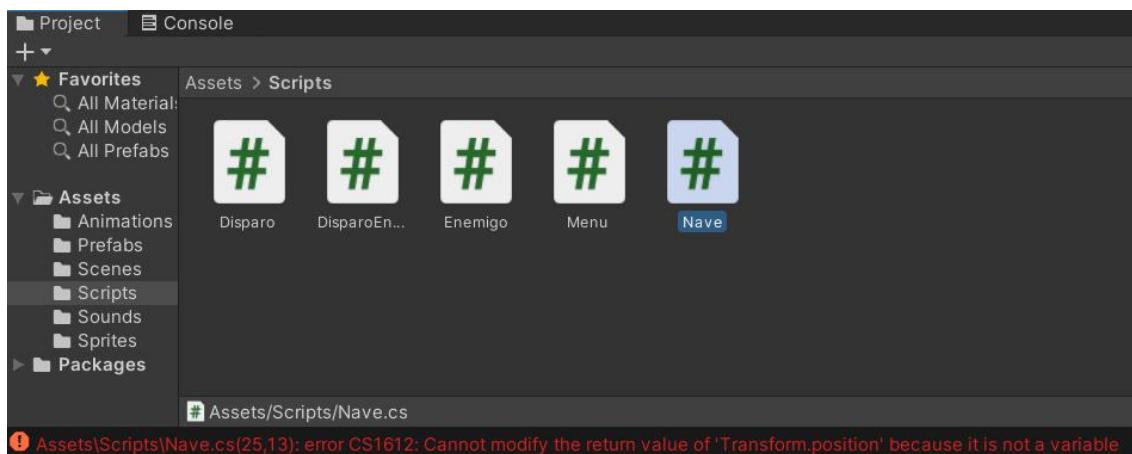
“Debug.Log” irá al panel inferior “Console”, como ya habíamos adelantado:

```
Debug.Log("Disparado");
```

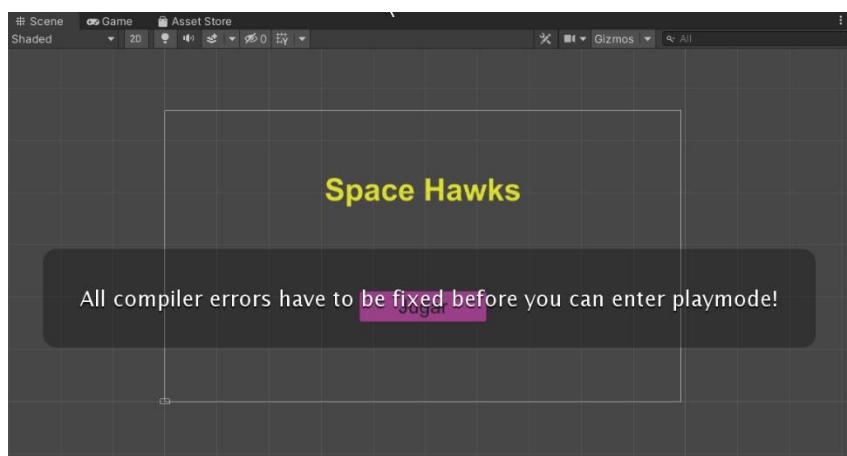
Además, en ocasiones nos sentiremos tentados de hacer cosas que “nos parezcan naturales” pero que no sean aceptables para Unity. Por ejemplo, podríamos intentar limitar la posición horizontal de nuestra nave forzando a que su posición en X no pueda pasar de ciertos valores límite:

```
if (transform.position.x < -3)
    transform.position.x = -3;
```

Si lo que tecleamos no es aceptable para Unity, se nos mostrará en la línea inferior de la pantalla (y en la “consola”, si la desplegamos).



Pero si no nos damos cuenta de ese detalle, en las versiones anteriores de Unity el programa “funcionará”... ¡¡sin incorporar los últimos cambios!!!. En Unity 2020 sí se nos avisará de que debemos corregir antes esos errores:



Por ello, lo ideal sería poder detectar los errores en el momento de editar con Visual Studio:

```
// Update is called once per frame
void Update()
{
    float horizontal = Input.GetAxis("Horizontal");
    transform.Translate(horizontal * velocidad * Time.deltaTime, 0, 0);

    if (transform.position.x < -3)
        transform.position.x = -3;

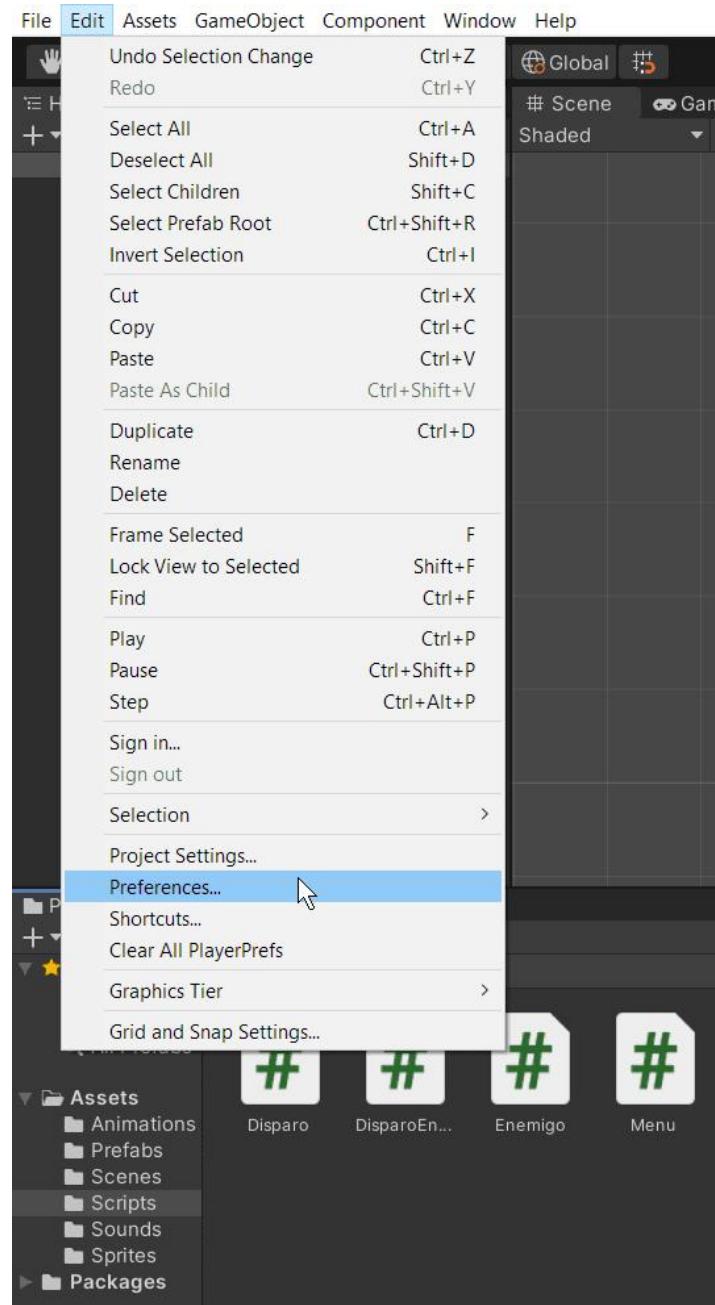
    if (Input.GetButtonDown("Fire1"))
    {
        textoSaludo.text = "Hola";
        GetComponent< AudioSource >().Play();
        Transform disparo = Instantiate(prefabDisparo,
            transform.position, Quaternion.identity);
        disparo.gameObject.GetComponent< Rigidbody2D >().velocity =
            new Vector3(0, velocidadDisparo, 0);
    }
}
```

Pero quizá, con una instalación estándar, no se destaqueen errores como el anterior:

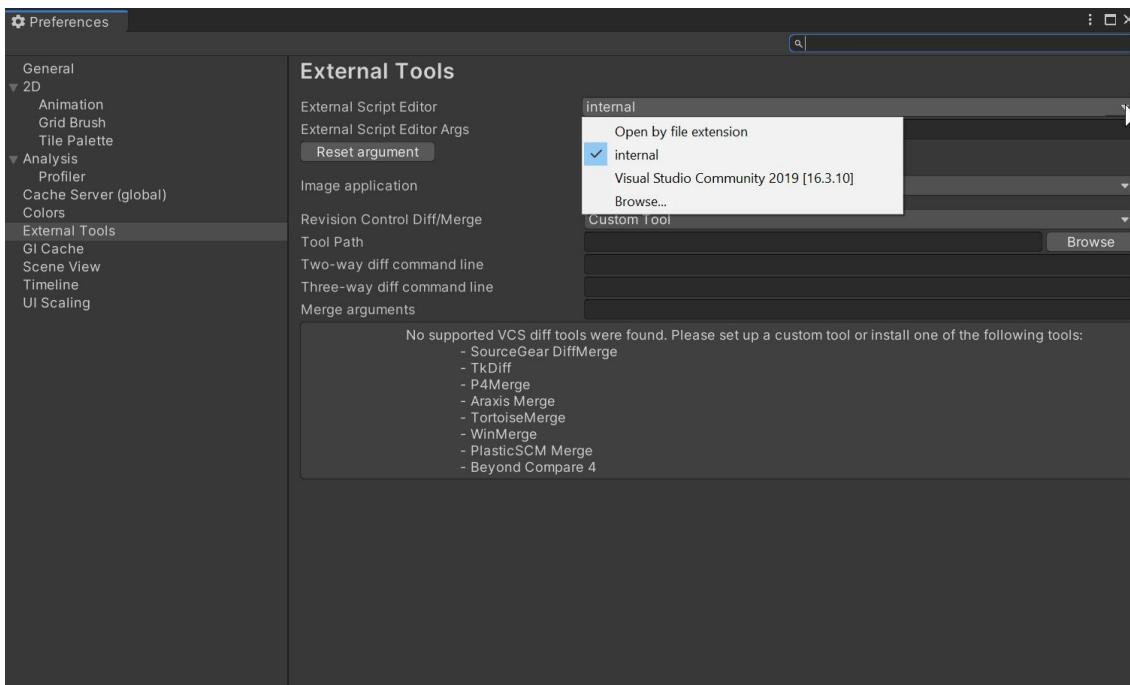
```
float horizontal = Input.GetAxis("Horizontal");
transform.Translate(horizontal * velocidad * Time.deltaTime, 0, 0);

if (transform.position.x < -3)
    transform.position.x = -3;
```

El primer paso para solucionarlo sería entrar a las preferencias (“Preferences”), en el menú “Edit”:



Y en la pestaña “External Tools” se nos indicará qué editor se está usando. Si aparece “Internal editor”, nos interesaría cambiarlo por Visual Studio:



Si Visual Studio no aparece en la lista de “External Script Editor”, nos interesaría saber dónde está instalado. Es un detalle que podemos ver desde el escritorio de Windows o el menú de Inicio, buscando el acceso directo a Visual Studio, pulsando el botón derecho y escogiendo “Propiedades”, para posteriormente “copiar y pegar” esa ruta.

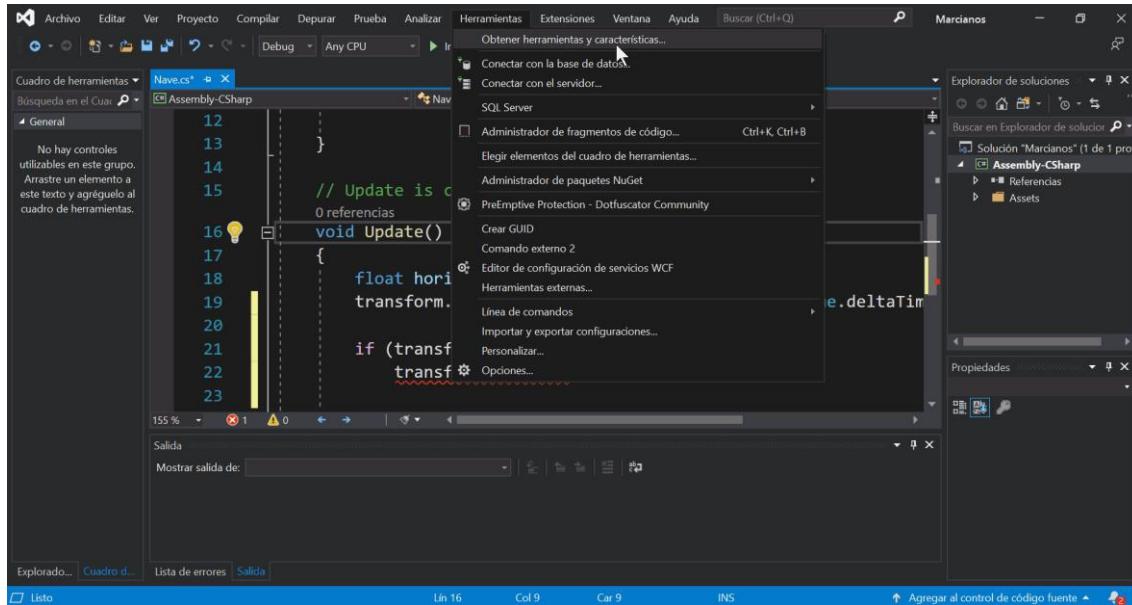
Si Visual Studio es nuestro editor reconocido, será más fácil saber cosas como que no podemos modificar los componentes de “transform.position”, pero sí asignarle un nuevo Vector3 formado por nuevos valores de X, Y, Z, por lo que una forma aceptable de limitar el movimiento podría ser:

```
if (transform.position.x < -3)
    transform.position = new Vector3(-3, transform.position.y, 0.0f);
```

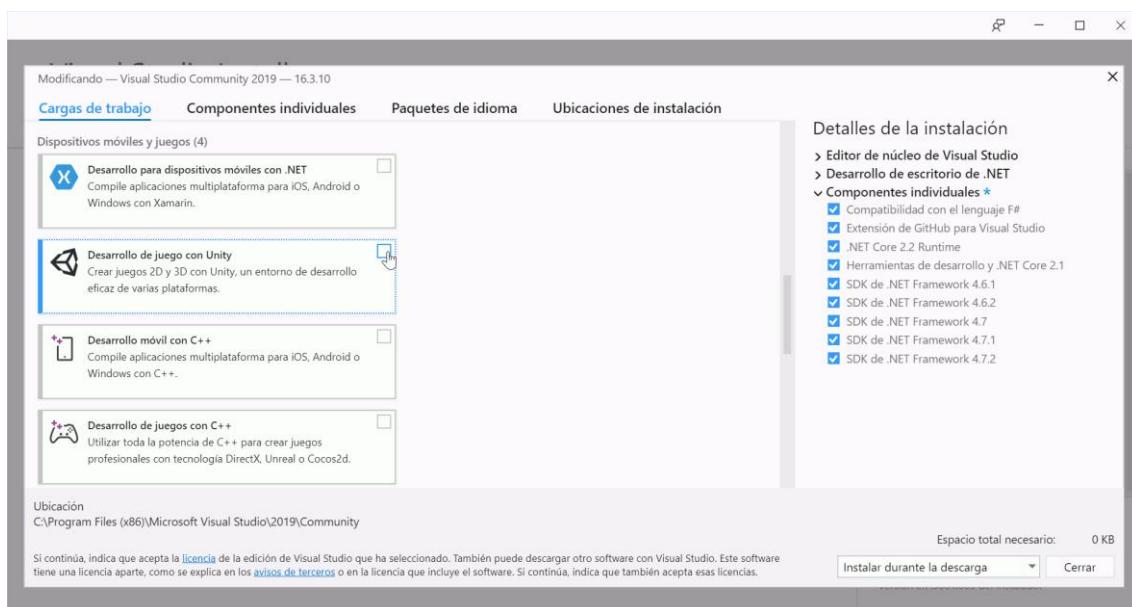
También deberías asegurarte de que tienes instaladas las **herramientas de Unity para Visual Studio**, para que el Intellisense de Visual Studio te ayude a saber los distintos métodos y propiedades que puedes usar en cada momento.

Es lo esperable con la instalación estándar, pero una forma de comprobarlo es: en el método “Update”, escribe “transform.” y pulsa Ctrl+Espacio para ver si te sugiere las propiedades y métodos de un componente “Transform”. Puede no ocurrir si has instalado una nueva versión de Visual Studio después de instalar Unity. En ese caso, podrías optar por usar la versión anterior de Visual Studio (si conservas ambas), por reinstalar Unity desde su web oficial o por añadir soporte

de Unity a tu versión de Visual Studio, con la opción “Obtener herramientas y características” del menú “Herramientas”:



y escogiendo “Desarrollo de juego con Unity”:



Ejercicio propuesto 1.29.1: Limita el movimiento de tu nave, para que no se pueda salir por los laterales de la pantalla. Asegúrate de que Visual Studio te destaca en rojo los posibles errores.

1.30. Mejoras propuestas

Nuestro esqueleto de juego ya es “casi jugable”, pero faltan detalles para que realmente lo sea, y con los conocimientos actuales serías capaz de hacer muchos de ellos. Por ejemplo:

- Que la nave no se pueda salir por los lados de la pantalla.
- Que cada disparo nuestro que impacte en un marciano nos dé 10 puntos.
- Que se muestren los puntos actuales.
- Que haya más enemigos, creados de forma estática o con Instantiate.
- Que cada vez que nos impacte un disparo enemigo o choquemos con un enemigo, perdamos una vida (de 3) y se recoloquen los enemigos restantes.
 - Que se muestren las vidas restantes.
 - Que se vuelva a la pantalla inicial si se pierden todas las vidas.
 - Que haya varios niveles, y al derribar a todos los enemigos de un nivel, se pase al siguiente. Esto tiene la dificultad de cómo conservar datos como los puntos y las vidas entre una escena y la siguiente. Dentro de poco veremos cómo implementar lo que se conoce como un “Singleton”, que nos ayudará en esa tarea, pero hasta entonces puedes probar otras formas de conservar datos, como usar ficheros o atributos “static”.