Un juego de aventuras

Rodrigo Pizarro Lozano

PID_00236758

Universitat Oberta de Catalunya

Índice

Introducción

En el campo del desarrollo de videojuegos existen distintas herramientas para poder llevar a cabo nuestro cometido. Estas ofrecen la posibilidad de gestionar distintos aspectos del desarrollo al más alto nivel y a través de una interfaz visual, lo que constituye un punto de entrada más accesible al desarrollo de videojuegos y supone agilizar dicho proceso. En la actualidad, existen distintas herramientas de este tipo. Este módulo sirve como punto de partida para aprender a usar una de las más populares: **Unity**.

El propósito de este módulo es ofrecer una visión general sobre el funcionamiento de Unity y sobre cómo empezar a trabajar con esta herramienta. Es importante tener en cuenta que este documento es solo un punto de partida, un breve resumen los aspectos más relevantes de cara a orientar un primer encuentro con Unity. No tiene sentido repetir todo lo que ya se encuentra explicado en la propia documentación oficial. Solo la práctica y la experimentación por parte del estudiante permite el autentico aprendizaje en este sentido, no la lectura de un documento.

Hemos estructurado este módulo en diferentes partes. En la primera, haremos un repaso de algunos conceptos como qué es un motor de juego. En la segunda, nos adentraremos más concretamente en Unity y veremos todo lo necesario para poder orientarnos dentro de la plataforma. Después, profundizaremos en la estructura de un proyecto de Unity y conoceremos todo lo necesario para entender con qué elementos se trabaja en Unity. Así pues, examinaremos algunas de las particularidades de esta plataforma, como son las carpetas especiales de un proyecto. Seguidamente, estableceremos algunas convenciones que recomiendan los responsables de Unity y que sigue la comunidad, en lo que respecta a la generación de código (*scripting*). También veremos algunos ejemplos de programación para Unity, que sirven como una primera referencia de cara a abordar su API.

Para finalizar, y con el propósito de profundizar y poner en práctica todos los conceptos presentados, seguiremos paso a paso el desarrollo de un juego.

Objetivos

En este módulo didáctico presentamos al alumno los conocimientos que necesita para alcanzar los siguientes objetivos:

1. Entender la interfaz del editor Unity.
2. Ser capaz de programar *scripts* básicos para Unity.
3. Poder exportar un proyecto como ejecutable.
4. Ser capaz de desarrollar un pequeño juego usando tanto el editor como el sistema de *scripting*.

1. ¿Qué es un motor de juego?

El primer paso antes de empezar a trabajar con un motor de juego es saber de qué se trata exactamente.

Un ***game engine*** (**motor de juego** o **motor de videojuego**) es una serie de herramientas comunes para la mayor parte de juegos agrupadas para facilitar su desarrollo.

Por ejemplo, la gran mayoría de juegos contienen imágenes, texturas y sonido, por lo que un motor debe incluir funciones para poder leer y mostrar estos archivos. En los inicios del mundo del desarrollo de videojuegos, cada vez que una persona o un equipo empezaban un nuevo proyecto debían programar todas esas funcionalidades o reciclar y modernizar las existentes antes de comenzar a desarrollar el juego propiamente. A medida que la complejidad de los títulos fue aumentando, esta tarea se fue convirtiendo en un obstáculo importante por la cantidad de horas que requería, además de convertirse en una enorme barrera de entrada para los que querían iniciarse. Afortunadamente, hace ya tiempo que diferentes equipos decidieron dar a conocer sus herramientas y liberarlas para que otros pudieran aprender o desarrollar sus ideas sin tener que dedicar tanto esfuerzo.

¿Qué motor usan?

Averigua cual es el motor usado en algunos de tus juegos favoritos en distintas plataformas (PC, móvil, consola).

Aunque hoy en día siguen habiendo empresas que desarrollan sus propios motores o incluso grupos que crean un entorno de desarrollo específico para un proyecto, la tendencia es usar motores comerciales ya existentes. Desarrollar un motor propio puede tener sentido si el proyecto es realmente ambicioso en algún aspecto, como gráficos hiperrealistas, dimensión del mundo, capacidad masiva de multijugador, etc., o porque la mecánica del juego requiere de alguna característica poco común. Pero esta puede ser una estrategia arriesgada, ya que además del coste en tiempo y dinero, si personas clave encargadas de construir el motor cambian de empresa, el impacto en los plazos de desarrollo del juego puede ser significativo, con lo que el proyecto puede verse seriamente afectado. Con un motor conocido, en cambio, atraer talento con experiencia que rinda desde el primer minuto es mucho más sencillo.

Además de facilitar todas las tareas rutinarias, los motores de juegos relativamente modernos también incluyen una interfaz gráfica para que artistas y programadores puedan trabajar sobre la misma plataforma. De esta manera, equipos multidisciplinares pueden colaborar de manera sencilla y ver el progreso de los demás. A la hora de desarrollar un juego, iterar y mejorar continuamente a todos los niveles, es fundamental.

Hay diversos parámetros que debemos considerar a la hora de elegir un motor. Por ejemplo, hay que tener en cuenta las **capacidades gráficas** de cada motor. Estas están marcadas por su trayectoria y sus objetivos. Algunos apostaron claramente por el nicho de los dispositivos móviles y más tarde evolucionaron hacia ser multiplataforma, mientras que otros hicieron el camino contrario. Esto se ve reflejado en la calidad gráfica que puede aportar cada motor.

Otro aspecto importante es evaluar la **dificultad de desarrollo** en cada entorno. Los lenguajes como Java o .NET suelen ser más asequibles para programadores con menos experiencia, mientras que programar en C/C++ es más delicado porque se pueden cometer errores críticos si no se depura bien.

Finalmente, la **comunidad** es otro aspecto fundamental. Cada plataforma tiene sus particularidades y limitaciones, y tener buen soporte es imprescindible. Las empresas dedicadas a crear y mantener motores gráficos se esfuerzan en generar comunidades activas que compartan ejemplos y se resuelvan problemas entre sí porque entienden que eso baja las barreras de entrada a nuevos usuarios y anima a los expertos a aprender nuevas facetas. A su vez, para los usuarios también es claramente positivo porque sus preguntas y dudas se pueden resolver de manera prácticamente instantánea, o puede incluso que lleguen a encontrar trabajo al demostrar sus habilidades a la comunidad.

Los tres motores gráficos más populares actualmente son Unity, CryEngine y Unreal. Es muy habitual ver equipos desarrollando en esas plataformas en *game jams* o *hackathones*. Los tres son entornos excelentes, pero este curso se centrará en Unity porque es un motor que trabaja con *scripting* en .NET y porque es uno de los más accesibles para iniciarse.

2. Conociendo Unity

Lectura complementaria

Podemos ver algunas estadísticas de Unity en: http://unity3d.com/public-relations.

La primera versión de Unity data de junio de 2005 y se presentó en la Worldwide Developers Conference, un evento de Apple. Inicialmente estaba diseñado exclusivamente para OS X, el sistema operativo de los de la manzana, pero rápidamente expandieron su soporte a otras plataformas, con especial atención a los dispositivos móviles. Hoy se puede exportar un proyecto a más de quince plataformas, incluyendo varios sistemas de realidad virtual y realidad aumentada. Su rápida progresión y su accesibilidad lo han catapultado hasta ser una de las herramientas de desarrollo más utilizadas. La compañía (Unity Technologies) nos proporciona estadísticas que publican en su web que nos permiten comprender su enorme repercusión. En el momento de redacción de este documento, Unity cuenta con más de 4,5 millones de desarrolladores registrados y hay 600 millones de personas jugando a algún título creado con esta herramienta. Vemos, pues, que se trata de un motor extremadamente popular.

A continuación empezaremos a conocer este motor internamente, para llegar a ser capaces de desarrollar un juego. Si bien la mayoría de términos usados en Unity pueden ser traducidos, en este documento nos ceñiremos a su nomenclatura original en inglés.

2.1. Estructura de un proyecto Unity

Antes de empezar a crear contenido, vamos a examinar cómo se estructura un proyecto de Unity. A nivel general, un proyecto sigue una estructura jerárquica. Primero de todo, cada proyecto Unity contiene una o más ***scenes*** (‘**escenas**’), que son la manera básica de organizar nuestro juego. Una *scene* normalmente equivale a un nivel en un juego. Por ejemplo, en un juego tipo *Angry Birds* se correspondería a cada pantalla, tanto las de juego como la del el menú principal donde escogemos dónde queremos empezar a jugar.

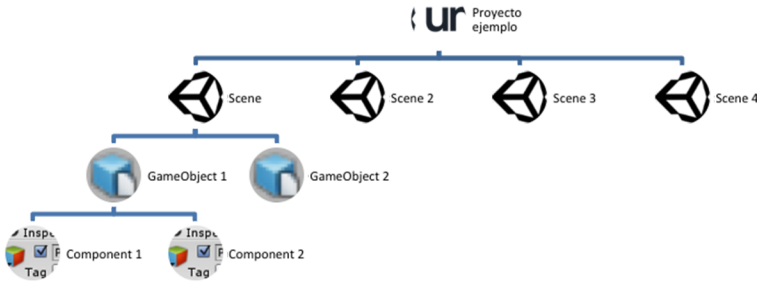
A su vez, cada *scene* es un conjunto de elementos, que Unity denomina individualmente *GameObject*.

Un ***GameObject*** puede ser cualquier cosa de nuestro juego. Por ejemplo, puede ser una luz, un *script*, una textura o un volumen para calcular colisiones.

Los distintos objetos se distinguen entre sí según su comportamiento. Lo que define el comportamiento de un objeto u otro son los ***components*** (‘**componentes**’) que contengan. Cada tipo de *component* tiene un efecto distinto. En

este sentido, nos podemos imaginar un *GameObject* como un contenedor de *components*. Sin embargo, existe un caso especial: el de un *GameObject* vacío; estos objetos no se visualizan y normalmente se usan para organizar la lógica del juego.

Veamos una ilustración de esta estructura:



2.2. Flujo de un proyecto

Lo que un usuario acaba viendo en la pantalla cuando se ejecuta un juego se denominan *frames* ('fotogramas').

Un ***frame*** es una imagen estática generada por la aplicación en un instante de tiempo determinado. La visualización secuencial de los distintos *frames* forma la ilusión del movimiento, al igual que pasa, por ejemplo, en el cine.

***Frames per Second* (FPS)**

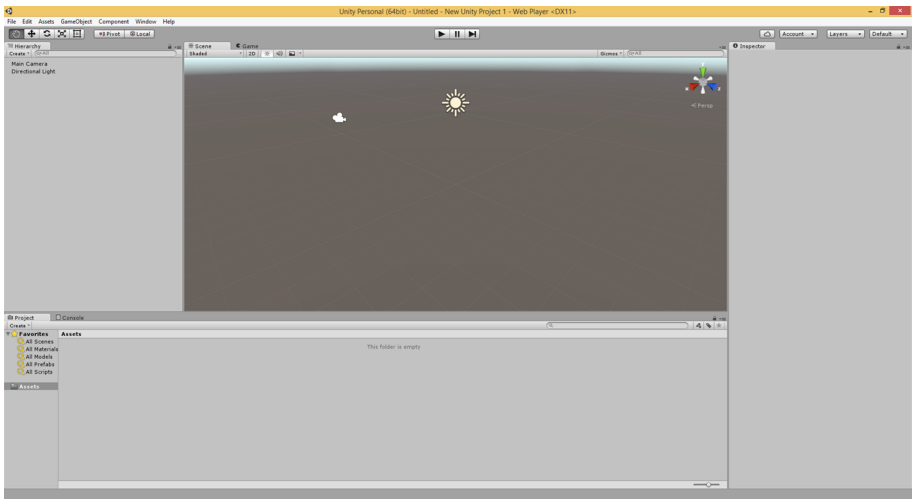
Esta unidad de rendimiento de un videojuego mide el número de imágenes que es capaz de generar per segundo. A mayor valor, mejor rendimiento.

Unity se encarga de todo el trabajo de generar estos *frames* lo más rápido y eficientemente posible, dada una definición de los objetos en una escena y la perspectiva desde donde la queremos ver.

2.3. El editor de Unity

Las herramientas de la mayoría de motores modernos se agrupan en editores gráficos con los que podemos trabajar visualmente. Unity no es una excepción, y su potente editor es posiblemente una de las claves de su éxito.

Por defecto, el editor de Unity se visualiza de la siguiente manera:



El editor está compuesto por ***tabs*** (‘**pestañas**’). Cada pestaña tiene una función específica, y se pueden reordenar –manteniendo el botón del ratón sobre una pestaña y arrastrándola–, ajustar de tamaño o agrupar según nuestras preferencias. Por defecto, las pestañas que aparecen son las de **Hierarchy**, **Scene**, **Game**, **Inspector**, **Project** y **Console**, aunque existen más, que aparecerán a medida que necesitemos opciones más avanzadas.

Nota

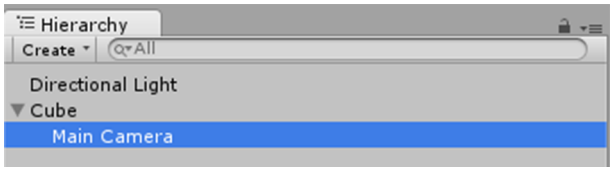
La distribución de pestañas de Unity en un momento dado se conoce como el *layout*.

2.3.1. La pestaña Hierarchy

La pestaña Hierarchy contiene la lista de *GameObjects* de una *scene*. Una *scene* se organiza en forma de jerarquía, de ahí el nombre de la pestaña, de modo que los distintos *GameObjects* se pueden emparentar. Por ejemplo, en la imagen que se muestra a continuación se ha añadido un cubo a la y luego hemos arrastrado con el ratón el objeto llamado «Main Camera» al objeto llamado «Cube»:

Nota

Para añadir el cubo se ha usado la opción «GameObject > 3D Object > Cube».



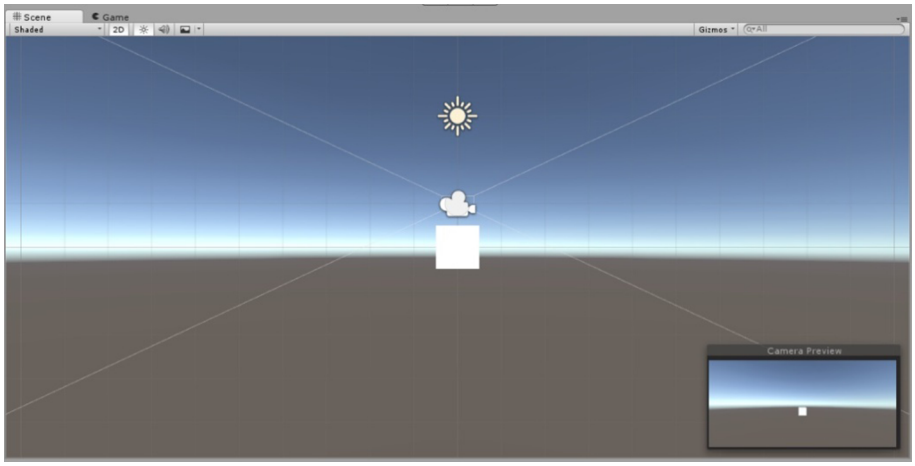
El resultado de emparentar dos objetos resulta en que los objetos hijo se verán afectados siempre de la misma manera que su objeto padre. Así pues, en el ejemplo, «Main Camera» (el *GameObject* hijo) se moverá, rotará, se eliminará o se escalará siempre que «Cube» (el *GameObject* padre) lo haga. Además de ser útil para sincronizar movimientos entre objetos, este tipo de jerarquía también nos servirá para organizar nuestra escena y encontrar nuestros objetos de una manera mucho más rápida y cómoda

Nota

Una jerarquía no está limitada en número de hijos por padre ni en cuantos niveles de jerarquía hay por *scene*.

2.3.2. La pestaña Scene

La pestaña Scene es nuestro editor visual de la escena que estamos desarrollando. Por ejemplo, después de añadir el cubo, el resultado es el siguiente:



Cada uno de los objetos que hayamos añadido en la pestaña Hierarchy nos aparecerá en la pestaña Scene representado por un símbolo. Además, el objeto que tengamos señalado mostrará más detalles.

> El símbolo con el que visualiza un objeto en la pestaña Scene se denomina ***gizmo***. El símbolo con el que se representan varía según los componentes añadidos a su *GameObject*.

En la imagen que acabamos de ver, el objeto seleccionado es el «Main Camera», representado con una cámara porque tiene un componente «Camera». En la escena también hay una luz, que podemos ver por el *gizmo* de un sol. El cubo parece un cuadrado porque nuestra perspectiva es en 2D. Finalmente, en este ejemplo, al tener una cámara seleccionada en la Hierarchy, podemos ver una previsualización (*preview*) de cómo se vería la escena desde el punto de vista de la cámara.

Existen varios modos de interactuar con los objetos y de navegar dentro de esta pestaña. Estos modos se pueden cambiar con los botones que tenemos justo debajo de la barra de menú o con las teclas.

Barra de interacción con los objetos de una escena en el editor.

Nota

Podemos cambiar rápidamente de modo con las teclas QWERT del teclado.

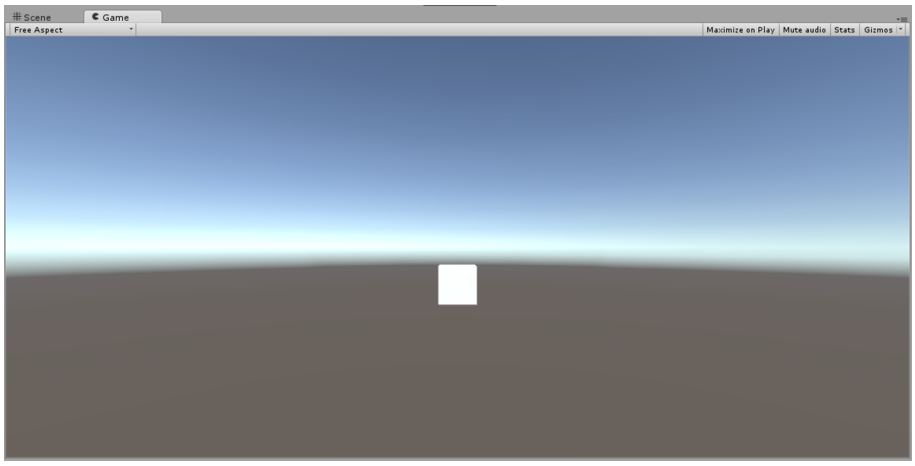
En el primer modo (una mano), podremos navegar por la escena desplazándola a derecha, izquierda, arriba y abajo mediante pulsaciones del ratón y arrastrando hacia alguno de los lados. En el resto de modos, pulsando el botón izquierdo del ratón podemos seleccionar objetos, mientras que con el derecho la navegación es idéntica al modo anterior. Con la rueda del ratón nos acercaremos hacia lo que haya en el centro de la pantalla.

Por lo que respecta a manipular objetos, en el primer modo no tenemos interacción, pero en el resto de modos podemos desplazar, rotar y escalar, respectivamente, los objetos que tengamos seleccionados. El último modo (un cuadro) es realmente una manera distinta de escalar un objeto, a base de mover los puntos que delimitan los extremos del mismo.

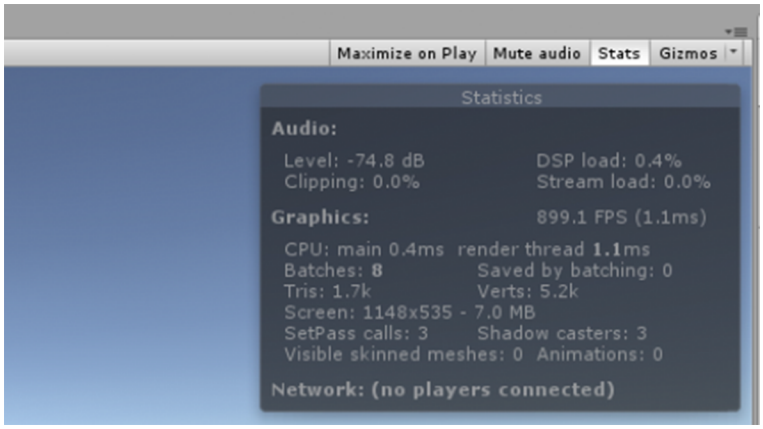
2.3.3. La pestaña Game

La pestaña Game se encuentra por defecto detrás de la de Scene. Para activarla, debemos pulsar sobre la pestaña, de modo que la pantalla del editor queda reemplazada por otra que nos muestra el resultado que obtendríamos si ejecutásemos el proyecto. Es decir, es donde realmente nos tenemos que fijar para comprobar que todo funciona como esperábamos.

Ubicación de la pestaña Game.

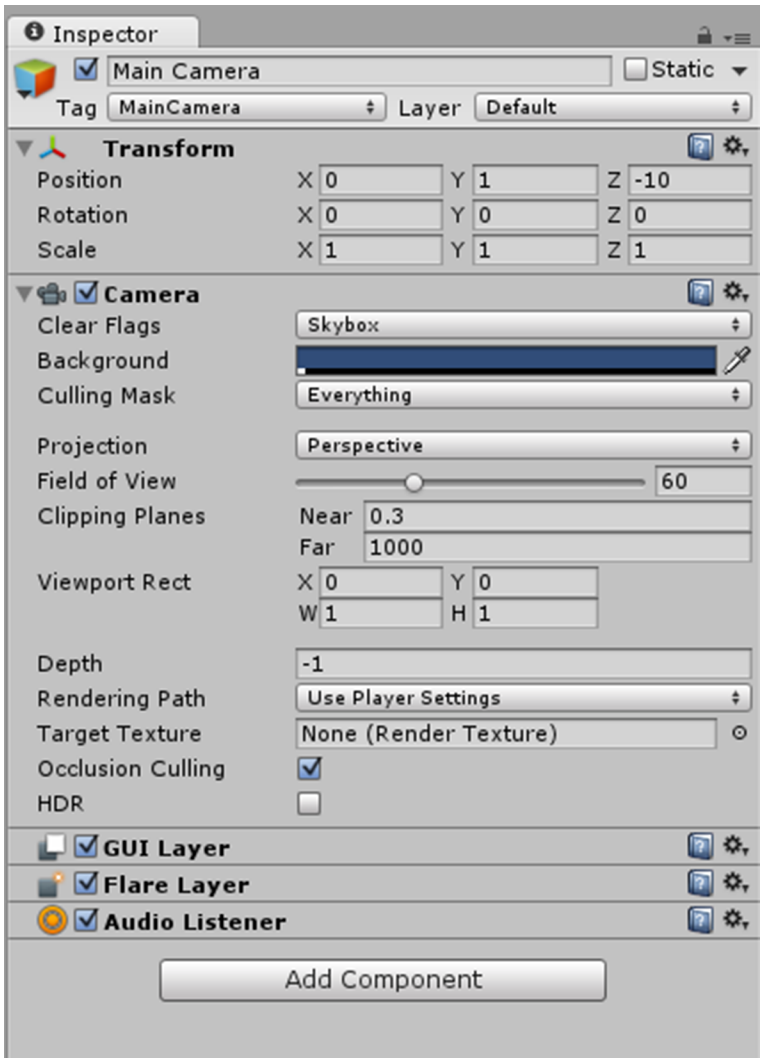


Hay una opción en su barra de menú, llamada «Stats», que es especialmente interesante. Cuando está activada, nos muestra información sobre la *performance* de nuestra aplicación. En particular debemos fijarnos en que nuestro juego tenga un número de *frames per second* (FPS) aceptable. Si no, la experiencia de juego se verá empobrecida. Una aplicación se percibirá como aceptablemente fluida cuando supere los 40 FPS; la experiencia será excelente a partir de los 60 FPS.



2.3.4. La pestaña Inspector

Esta es probablemente la pestaña más importante del editor. Aquí podemos ver qué contienen los *GameObjects* seleccionados en la Hierarchy. Vemos un ejemplo para un objeto llamado «Main Camera».



En este caso, podemos ver que «Main Camera» contiene componentes de tipo «Camera», «GUI Layer», «Flare Layer» y «Audio Listener». Además, tiene un componente de tipo «Transform», que es obligatorio en todos los *GameObjects*.

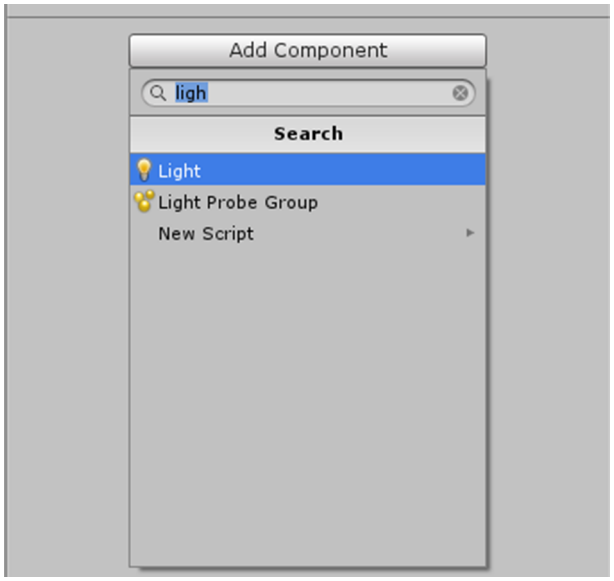
Nota

Dentro del «Transform» de un objeto se indica su posicionamiento y rotación.

Recordemos que, en Unity, un *GameObject* es cualquier elemento del juego, sin definir por sí mismo de qué se trata. Puede ser cualquier cosa: una cámara, un elemento gráfico, un *script*, etc. Para que un objeto se comporte de la manera deseada, debemos añadirle el *component* adecuado.

En Unity, los *components* de un objeto son los que realmente definen cuál es su comportamiento o cómo se va a representar en el juego.

Por ejemplo, si añadimos un componente vacío a nuestra escena, podemos transformarlo en una luz pulsando en el botón «Add Component» que encontramos en la pestaña Inspector y buscando el tipo «Light» entre la lista de componentes.



Nota

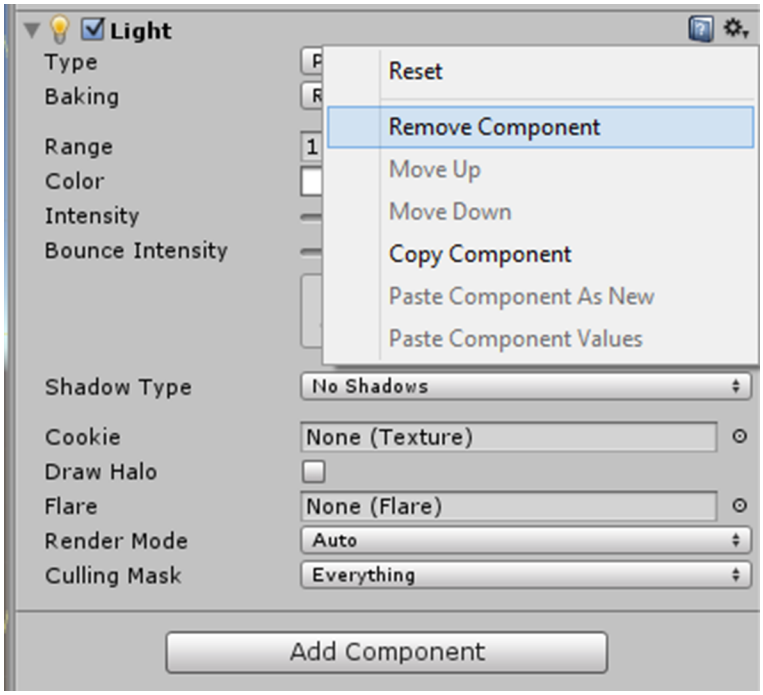
Para crear un objeto vacío se puede usar la opción «Create > Create Empty» en la pestaña Hierarchy.

Nota

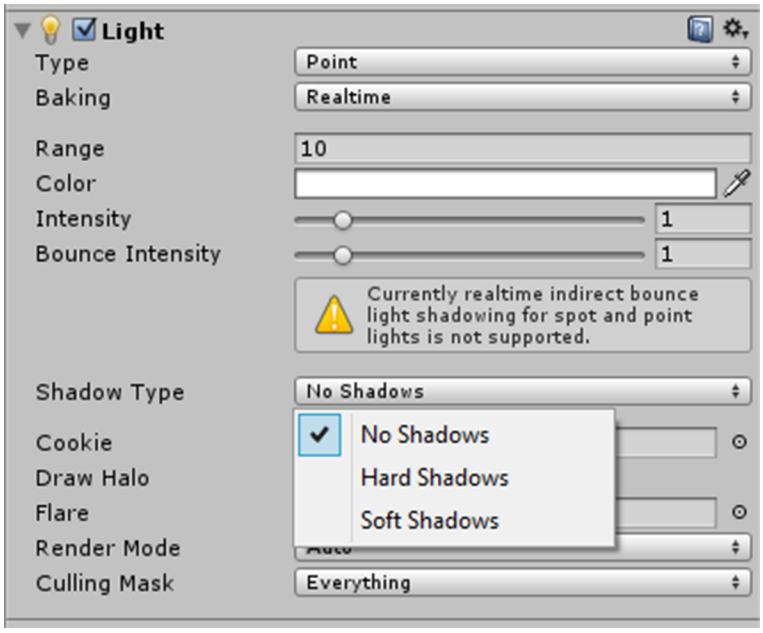
Para encontrar rápidamente un componente en la lista, es recomendable usar el cuadro de búsqueda asociado.

Será solo entonces que el *GameObject* se comportará como una luz. Añadiendo componentes adicionales, se pueden asociar diferentes comportamientos a los *GameObjects*. Por ejemplo, el objeto de nuestro ejemplo puede ser también una cámara si le añadimos el componente adecuado.

Naturalmente, también podemos quitarle componentes a un objeto, pulsando en la parte superior derecha (el icono con el engranaje) para activar el siguiente menú:



Una vez se ha añadido un componente, es posible configurar su comportamiento en la propia pestaña Inspector:



2.3.5. La pestaña Project

Esta pestaña muestra todos los archivos que tenemos dentro de nuestro proyecto, nuestros *assets* ('activos').

Se denomina ***asset*** cualquier archivo importado que sea necesario para la ejecución del proyecto. Por ejemplo: imágenes, *scripts* (fragmentos de código fuente), texturas, videos, mallas 3D, animaciones, etc.

Web recomendada

Podemos encontrar una gran cantidad de contenido libre y gratuito en webs como la siguiente:
http://opengameart.org.

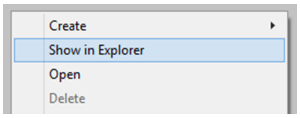
El aspecto inicial de la pestaña es el siguiente. En este caso, aún no tenemos ningún archivo importado.

Supongamos que nos descargamos una imagen de Internet que deseamos incorporar a nuestro proyecto. Para ello, podemos importarla de dos maneras:

- Copiando el archivo en la carpeta Assets del proyecto Unity en el sistema de ficheros de nuestro ordenador.
- Arrastrándola directamente desde el explorador de archivos del ordenador al interior de la propia pestaña Project.

Nota

Una manera rápida de encontrar la carpeta Assets del ordenador es haciendo clic derecho en la pestaña Project y seleccionando «Show in Explorer».



2.3.6. La pestaña Console

Esta pestaña nos mostrará todos los avisos y errores que sucedan en nuestro proyecto. También permite mostrar mensajes de depuración generados directamente desde el código de nuestro juego.

Web recomendada

Podemos encontrar más información sobre el uso de la pestaña Console en:
http://docs.unity3d.com/Manual/Console.html.

2.3.7. Modos de ejecución en Unity

Los dos modos de funcionamiento de Unity son:

- El **modo de edición** (***Editor mode***). Cuando estamos en este modo, el motor de juego está parado y tanto su código como muchas de sus herramientas integradas (como la física, sistemas de partículas, etc.) están detenidas. El modo de edición sirve para que vayamos construyendo la escena sin tener distracciones, como objetos moviéndose o desapareciendo. En esencia, en este modo trabajamos con un editor de niveles.
- El **modo de ejecución** (***Play mode***). En este modo, en cambio, Unity está simulando que un usuario inicia la aplicación que hemos desarrollado. Todas las herramientas del motor se activarán y la experiencia que tenga-

mos viendo la pestaña Game será muy similar a la que tendríamos al iniciar la aplicación real.

En modo ejecución es posible modificar una escena del juego y sus elementos mediante el editor. Sin embargo, todo lo que se modifique en este modo se perderá una vez salgamos de este modo. Por ejemplo, si añadimos un objeto «A» a la escena y borramos un objeto «B» existente, al volver al modo edición, el objeto «A» desaparecerá y el objeto «B» volverá a existir. En caso de querer mantener esas operaciones, hay que llevarlas a cabo siempre en el modo edición.

Para entrar en modo ejecución, simplemente hay que apretar el botón «Play» en el grupo de botones situado en la parte superior central del editor, justo debajo del menú. Para salir, volvemos a apretar «Play», con lo que volveremos a estar en modo edición. El botón de pausa congela la aplicación momentáneamente; lo podemos usar para examinar con calma una escena. El tercer botón sirve para ir *frame* a *frame*, de modo que podemos poder ver los cambios con mucho más detalle.

Botones de control del modo ejecución.

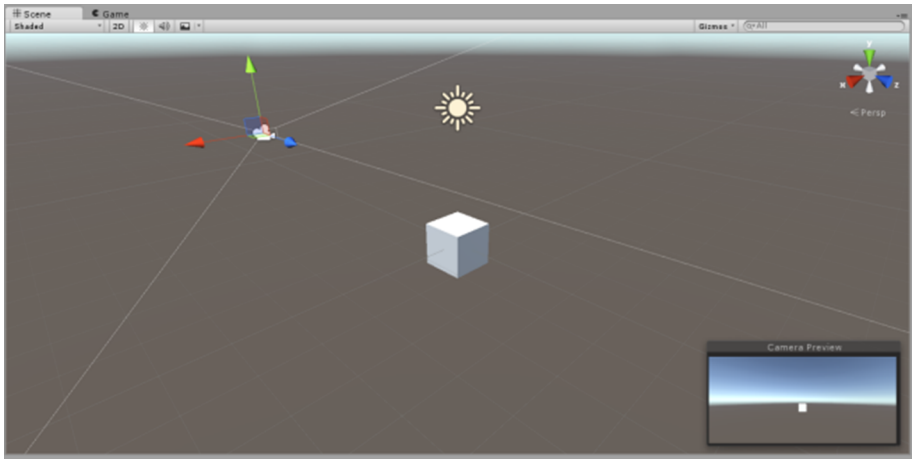
2.4. Unity en 2D *vs* 3D

Unity empezó como un motor 3D, sin soporte para aplicaciones 2D. Sin embargo, diversos desarrolladores añadieron posteriormente funcionalidades 2D a través de *plugins*, lo que aumentó enormemente la popularidad de Unity en el mundo de las aplicaciones para móvil. A partir de la versión 4.3, Unity empezó ya a integrar una gran cantidad de herramientas específicas para 2D. Sin embargo, internamente Unity no distingue entre aplicaciones 2D y 3D. Las diferencias se limitan a cambios de perspectiva y de motor físico. En ese sentido, podemos imaginarnos un proyecto 2D funciona como un teatro de marionetas, donde el eje *Z* se corresponde a la profundidad del escenario. Desde el punto de vista del espectador, vemos un conjunto de capas bidimensionales superpuestas.

Nota

Según el tipo de proyecto, 2D o 3D, Unity usa el motor físico Box2D o PhysX.

Si queremos trabajar en un proyecto 3D, nuestra escena mostrará el mundo claramente con sus tres coordenadas:

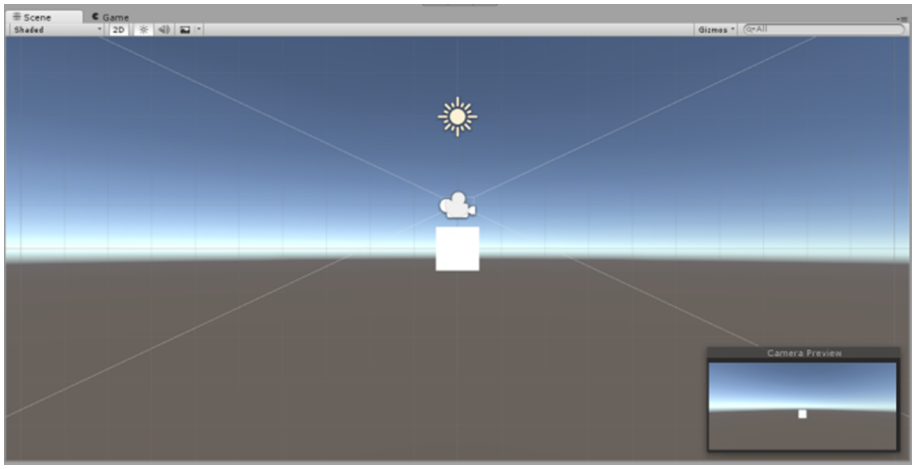


Podemos cambiar la perspectiva entre 2D o 3D pulsando la tecla «2», o bien activando/desactivando el botón «2D» en la parte superior de la pestaña Scene. En el modo 3D, la posición y la orientación del *GameObject* seleccionado se muestran con un eje de coordenadas.

Nota

Eje de coordenadas con las flechas roja, verde y azul, que representan los ejes *X*, *Y* y *Z*, respectivamente, en la esquina superior derecha de la pestaña Scene:

Esta misma escena la podemos observar en modo 2D. Al hacerlo, nos situamos en una perspectiva en la que desaparece el eje *Z* y vemos el mundo siempre centrado en el eje *Z*.



Dado que realmente Unity está trabajando con 3D, podemos incluir objetos situándolos simplemente unos delante de otros en el eje *Z*. En ese sentido, cabe señalar que aunque este módulo se centra en proyectos 2D, casi todo lo que aprendamos será aplicable también a 3D.

2.5. Carpetas especiales

Unity no trata a todos los *assets* por igual, y la distinción la hace según la carpeta en la que están ubicados. La razón es dar más flexibilidad a los desarrolladores. Por ejemplo, podemos crear funcionalidades que sirvan solo como herramientas para acelerar el desarrollo dentro del editor, pero que no tengan sentido dentro del juego. Este es un recurso muy típico al que se suele recurrir cuando los juegos alcanzan un nivel de complejidad alto. Pero no es el único. A continuación veremos algunas de las carpetas especiales de Unity más relevantes y analizaremos su uso.

Web recomendada

Para obtener la información completa de todas las carpetas, podemos acudir a la documentación de Unity:
http://docs.unity3d.com/Manual/SpecialFolders.html.

2.5.1. Carpeta Editor

Podemos generar fragmentos de código en forma de *scripts* que nos ayuden a acelerar nuestro progreso a la hora de desarrollar nuestro juego, pero que no se incluirán en el juego una vez exportado a alguna plataforma concreta. De este modo, optimizamos espacio y evitamos posibles problemas.

2.5.2. Carpeta Plugins

Algunas funcionalidades se crean de manera específica para alguna plataforma. Otras veces, simplemente queremos reaprovechar módulos ya existentes programados en otros entornos y otros lenguajes. En estos casos, normalmente, se generan archivos binarios ejecutables que se puedan llamar directamente desde un *script* de Unity. Para que Unity los reconozca como *plugins*, hay que ponerlos en esta carpeta especial.

Nota

Algunos ejemplos de binarios ejecutables son los archivos .dll en Windows o .so en Android.

2.5.3. Carpeta Resources

Algunos *assets* pueden ser costosos en términos de memoria y puede que solo los necesitemos en algunos momentos en concreto. Para evitar esta carga innecesaria, podemos usar ciertas funciones de código para indicar qué *asset* queremos instanciar. Eso sí, solo tendremos acceso vía código a los *assets* que se encuentren en la carpeta Resources.

2.5.4. Carpeta de recursos *online*

Muchas veces nos encontraremos que necesitamos recursos para nuestro juego. Especialmente si somos un equipo pequeño, es muy complicado tener nociones suficientes de cómo generar animaciones, texturas, música y mallas 3D con suficiente calidad. Sin embargo, en la gran mayoría de ocasiones, lo que necesitamos ya ha sido creado para otro proyecto, y podemos obtenerlo por un precio ajustado o, incluso, gratuitamente.

La **Asset Store** es la tienda oficial de Unity. Contiene todo tipo de recursos, desde texturas y *scripts* hasta animaciones o niveles completos de algún juego.

Nota

Para descargar *assets* desde la tienda es necesario tener una cuenta de desarrollador de Unity.

En esta tienda podemos encontrar recursos gratis y de pago, así como en modalidad de prueba. Podemos acceder a la Asset Store desde la web de Unity o desde el propio editor, pero los *assets* que queramos utilizar solo los podemos descargar desde el editor.

Por otro lado, si creemos que tenemos algo interesante para ofrecer a la comunidad, también es posible poner nuestro contenido a la venta. No son pocos los que viven justamente de vender *assets* en la tienda de Unity, y en el blog de la web de Unity se pueden encontrar testimonios, estrategias y consejos para conseguirlo.

Web recomendada

El blog de la Asset Store se encuentra en:
http://blogs.unity3d.com/category/asset-store.

Más allá de la tienda oficial, hay multitud de recursos *online* que podemos encontrar en diversas webs. Sin embargo, debemos ir siempre con cuidado con los derechos de autor de todo aquello que nos planteemos usar en nuestros proyectos.

2.6. *Prefabs*

Es muy usual durante el desarrollo de un proyecto tener estructuras que se repiten a menudo. Por ejemplo, los enemigos se suelen controlar a través de uno o varios *scripts* y, si tenemos cientos de ellos en una pantalla, hay que añadírselos a todos uno a uno. Huelga decir que añadir todos los *scripts* con todas las configuraciones para cada uno puede ser terriblemente tedioso; además, es muy probable que nos equivoquemos en algún momento. Incluso podría ser peor si en un momento más avanzado del desarrollo decidimos hacer algún cambio y tenemos que repetir el proceso de nuevo: deberíamos asegurarnos de que modificamos absolutamente todos los enemigos. Para evitar este tipo de problemas, Unity tiene un sistema llamado *prefab*.

Podemos imaginarnos un ***prefab*** como una plantilla donde guardamos el estado actual de un objeto y sus componentes. Esta plantilla se puede usar para replicar objetos sin límite en cualquier *scene* del proyecto. Cualquier cambio hecho al *prefab* afecta a todas réplicas.

No estamos obligados a que todos los objetos sean copias exactas. Una vez se ha creado un objeto a partir del *prefab*, podemos personalizar cada instancia individualmente si lo deseamos. Por ejemplo, para el caso anterior, podríamos

Nota

Es recomendable guardar todos los *prefabs* de los *assets* de nuestro proyecto en una carpeta llamada «Prefabs».

hacer que la mayoría de enemigos tengan la misma velocidad, pero que uno en concreto sea más rápido, aún cuando comparta el resto de propiedades exactamente.

Existen diversas maneras de crear un *prefab*, pero una relativamente simple es la siguiente:

- Crear el *GameObject* que deseamos replicar en el editor. Asignar todas las propiedades deseadas hasta tenerlo a punto.

- Arrastrar dicho objeto desde el apartado de Hierarchy al de Project.

- Al hacerlo, veremos que aparece un nuevo *asset* con el nombre remarcado en color azul (en vez de negro, que es el habitual). Esto indica que se trata de un *prefab*.

- Para crear una réplica, basta con arrastrar el *prefab* desde Project hasta Hierarchy, o bien directamente sobre la pestaña Scene.

Para modificar las propiedades de todas las réplicas a la vez, basta con seleccionar el *prefab* y editar en su pestaña Inspector donde corresponda.

3. *Scripting* en Unity

Si bien Unity permite gestionar el comportamiento de los *GameObjects* hasta cierto punto a través del editor, el caso más habitual será programar la lógica del juego a través de *scripts* de código fuente.

Un ***script*** en Unity es un *component* más que podemos definir nosotros, siempre asociado a un *GameObject* concreto.

Podemos elegir entre los lenguajes JavaScript o C#, si bien este módulo se centra en el segundo. Hay diversas razones por la cual se ha elegido este lenguaje, aunque quizás la más importante es que se trata del más utilizado dentro de la comunidad Unity.

Los *scripts* en Unity son clases que heredan de la superclase MonoBehaviour, definida por el propio motor, y que sirve como interfaz para poder implementar varias funciones a las que se llama internamente desde Unity, mediante su redefinición (*override*). Los métodos más importantes y que usaremos más a menudo son los siguientes:

- «Start», que se invoca al iniciarse por primera vez el *script*, una sola vez.
- «OnEnable», invocado cuando se habilita el objeto.
- «OnDisable», es el inverso al anterior, invocado cuando se deshabilita.
- «Update», invocado una vez por *frame*.

Web recomendada

Toda la información relacionada con la clase MonoBehaviour se encuentra en la API de Unity:

http://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

Si buscamos en la documentación de Unity, veremos que la lista es mucho más amplia. Aunque podemos encontrar toda la información en dicha documentación, en esta sección vamos a describir brevemente cuándo se ejecuta cada función y el caso de uso más común, además de las convenciones que vamos a usar a la hora de programar en Unity.

En Unity se suelen usar las mismas convenciones del lenguaje utilizado para programar, en este caso centrándonos en el lenguaje C#. Por tanto, usaremos las convenciones descritas por el creador de dicho lenguaje, Microsoft. Por ejemplo, la notación que emplearemos será CamelCase, donde los nombres de clases y métodos empiezan por mayúscula, mientras que las variables em-

Nota

El lenguaje C# tiene muchas similitudes con el lenguaje Java.

piezan por minúscula. Además, como pasa en algunos lenguajes de programación, Unity no permite que los ficheros de *script* se llamen de manera distinta a la clase que contienen.

Consideraciones sobre *scripting* de Unity

A los programadores más experimentados les suele sorprender el hecho de que no haya que usar *keywords* como «override», o que las llamadas a funciones como «Start» o «Update» puedan tener firmas distintas (pueden ser *public*, *private*, pueden retornar datos, etc.). El motivo de que esto se pueda hacer es que el motor no hace llamadas directas a las funciones, sino que usa *reflection* para cada *script* compilado para encontrar todos los métodos propios del desarrollo de una aplicación Unity. Esto nos ofrece una flexibilidad que muchos motores no son capaces de dar, aunque se paga un pequeño peaje en cuestión de rendimiento. *Reflection* suele ser un proceso mucho más lento que una llamada corriente a una función, y si tenemos del orden de decenas de miles de *scripts*, el sistema puede que se resienta, especialmente con sistemas relativamente antiguos. Hay maneras de solucionar estas limitaciones, como los patrones Manager, pero son temas más avanzados.

3.1. Flujo de ejecución

En la mayoría de motores modernos el flujo de ejecución es muy similar. Esto se debe a que la estructura de ejecución de cualquier juego es prácticamente idéntica a la del resto. Dado que un videojuego es una aplicación altamente interactiva, con muchos elementos que deben actuar de manera independiente en modo concurrente, su flujo de ejecución no puede ser totalmente secuencial. Básicamente, un juego se compone de un tiempo de inicialización (por ejemplo, cuando el juego se está cargando) y, una vez finalizado, se ejecuta un bucle en el que, a cada iteración, se va generando una imagen que describe el estado actual del juego. Durante cada *frame*, cada objeto puede ejecutar código que actualice su estado.

Nota

Cada imagen generada, y por lo tanto cada iteración de bucle, se corresponde a un *frame*.

Debemos tener en cuenta que un usuario tiene que tener una cierta paciencia cuando un juego está arrancando o cargando un nivel. Típicamente, puede esperar varios segundos. Además, si queremos mantener unos FPS aceptables para una experiencia agradable, tan solo tenemos como máximo unos pocos milisegundos por cada *frame*. En este tiempo hay que ejecutar nuestros *scripts*, generar la imagen propiamente dicha y mostrarla por pantalla. Por lo tanto, el uso del tiempo es muy distinto para cada caso.

Nota

Si deseamos 40 FPS, un valor aceptable, hemos de garantizar que todo se ejecuta en no más de 25 milisegundos por *frame*.

Es importante tener en cuenta que los *scripts* en sí mismos, como *assets* dentro de nuestro proyecto, no se ejecutarán a menos que se asignen como componente a algún *GameObject* en la escena del juego. Unity solo ejecuta los métodos de *scripts* asociados a un *GameObject* de la escena en curso. Esto lo debemos hacer desde el editor, seleccionando un *GameObject* en la pestaña Scene. Entonces, desde el Inspector usar el botón «Add Component» y seleccionar un *script* en lenguaje C#.

3.2. Métodos importantes para programar en Unity

Los motores nos proporcionan maneras de programar lo que queremos hacer en el breve intervalo de tiempo de la inicialización y por cada *frame*. En el caso de Unity, el motor se encarga de llamar en los momentos adecuados a los métodos «Start» y «Update», entre otros, de los *scripts* asociados a cada objeto en una escena. De este modo, podemos controlar lo que ocurre en cada instante de nuestro proyecto. A continuación veremos en detalle cómo se usan estas funciones con algunos ejemplos.

3.2.1. «Start»

Este método se ejecuta una sola vez por instancia, cuando Unity detecta que el *script* se ha añadido a un objeto (por ejemplo, al inicio de la aplicación) y sirve para iniciar parámetros o para incluir todas aquellas funcionalidades costosas en tiempo que solo tengan que ejecutarse una sola vez. Todo este tipo de código procuraremos hacerlo en esta función.

Veamos un pequeño ejemplo de *script*:

```
1.    using  UnityEngine;
2.    using  System.Collections;
                                  public class
3.    public  class  ExampleClass:  MonoBehaviour  {
4.        void  Start()  {
5.            Debug.Log("Hello world");
6.        }
7.    }
```

«Debug.Log»

Este método permite escribir texto por la consola durante la ejecución de un *script* de Unity.

Como ya habíamos dicho, nuestro *script* de ejemplo es una clase que hereda de MonoBehaviour, que es una clase definida internamente por Unity. Esta clase la hemos llamado «ExampleClass» y contiene un solo método, la redefinición de «Start». Por lo tanto, el objeto que contenga este *script* escribirá una sola vez por consola, al inicio de la ejecución, el texto `"Hello world"`.

3.2.2. «Update»

Este método es uno de los más importantes a la hora de generar *scripts*. Se ejecuta una vez por *frame*, justo antes de mostrar la imagen resultante por pantalla.

Es muy importante que esta función sea ligera. Si hacemos operaciones complejas (por ejemplo, inteligencia artificial avanzada, cálculos matemáticos extensos, accesos a bases de datos), los FPS pueden bajar en picado y, con ello, la experiencia de juego.

Existen maneras de hacer operaciones que requieran mucho tiempo de manera regular (cada cierto tiempo o cada cierto número de *frames*), pero debemos descartar ejecutar en cada *frame* todo aquello que sea pesado. En el caso de necesitar ejecutar funcionalidades complejas de manera regular, podemos consultar en la documentación de MonoBehaviour más funciones que nos podrán ser útiles.

Veamos un ejemplo sencillo de *script* con la función «Update»:

```
1.    using UnityEngine;
2.    using System.Collections;


3.    public class ExampleClass: MonoBehaviour {
4.        void Update() {
5.            Debug.Log("Esto es un Update");
6.        }
7.    }
```

Este *script* escribe el mensaje "`Esto es un Update`" por cada *frame* y por cada *GameObject* al cual está añadido. Así pues, si nuestro juego se ejecuta a 20 FPS, eso significa que se escribirá 20 veces por segundo.

3.2.3. «OnEnable» y «OnDisable»

Estas funciones se ejecutan cada vez que un *script* pasa a estar habilitado (incluido cuando el *script* se añade a un objeto) o deshabilitado. Puede tener sentido deshabilitar temporalmente un *script* si queremos que momentáneamente deje de ejecutar su código. En caso de querer que el *script* deje definitivamente de ejecutarse, lo más adecuado eliminarlo vía el editor o vía *scripting* con la instrucción `Destroy`.

Nota

Un *script* se habilita o deshabilita a través del *checkbox* de la lista de componentes del Inspector.

Veamos un ejemplo:

```
1.    using UnityEngine;
2.    using System.Collections;


3.    public class ExampleClass: MonoBehaviour {
4.        void OnEnable() {
5.            Debug.Log ("script was enabled");
6.        }
7.        void OnDisable() {
8.            Debug.Log ("script was disabled");
9.        }
10.   }
```

En este caso, al entrar en modo ejecución veremos el mensaje «script was enabled» en la consola, al iniciarse. Si vamos al Inspector y hacemos clic en el *checkbox* que activa/desactiva el *script*, veremos por consola que cada vez que activamos el *script* se nos muestra el mensaje «script was enabled», y que cuando desactivamos el *script* se nos muestra «script was disabled».

Al salir del modo ejecución también veremos el mensaje «script was disabled» ya que, justo antes de salir de ese modo, todos los *scripts* reciben la llamada «OnDisable».

3.3. Edición de campos en el Inspector

Los campos (*fields*) de tipos básicos declarados como públicos de una clase tienen una propiedad muy interesante para los desarrolladores que usan Unity. Podemos ver y configurar su valor a través del editor de Unity, mediante la pestaña Inspector.

Nota

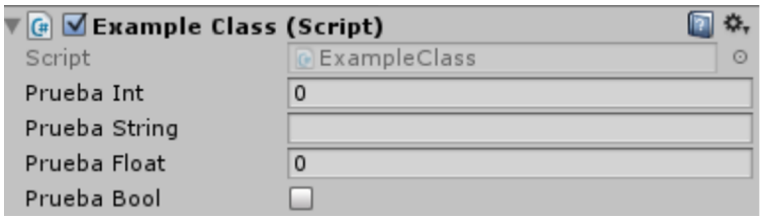
`Int`, `string`, `bool` o `float` se consideran tipos básicos, por ejemplo.

Vamos a ver un ejemplo. Supongamos el siguiente *script*:

```
1.    using     UnityEngine;
2.    using     System.Collections;


3.    public class ExampleClass: MonoBehaviour     {
4.        public   int    pruebaInt;
5.        public   string  pruebaString;
6.        public   float  pruebaFloat;
7.        public   bool  pruebaBool;
8.
9.        void     Start()     {
10.           Debug.Log ("pruebaInt " + pruebaInt);
11.           Debug.Log ("pruebaString " + pruebaString);
12.           Debug.Log ("pruebaFloat " + pruebaFloat);
13.           Debug.Log ("pruebaBool " + pruebaBool);
14.       }
15.   }
```

En este caso, los campos públicos son `pruebaInt`, `pruebaString`, `pruebaFloat` y `pruebaBool`. Si asignamos este *script* a un objeto y damos un vistazo al componente que corresponde al *script* a través la pestaña Inspector, veremos que los cuatro campos aparecen reflejados y podemos introducir datos directamente.



Podemos modificar los valores de estos campos libremente y luego apretar el botón «Play»y para ejecutar el programa. Al hacerlo, se nos mostrarán por consola los últimos valores que tenían antes de pasar a modo ejecución.

No persistencia de estado entre modo edición y ejecución

Recordad que al salir del modo ejecución se pierden todas las modificaciones que se hayan hecho durante ese modo. Por ejemplo, si antes de apretar «Play» `pruebaInt` tenía como valor 3, durante la ejecución lo modificamos para que tenga valor 4 y salimos del modo ejecución pulsando «Play» de nuevo, veremos que `pruebaInt` vuelve a valer 3.

Unity también muestra en el editor aquellos campos que se definan con el atributo `[SerializeField]`. El ejemplo anterior lo podemos modificar de la siguiente manera:

```
using      UnityEngine;
using      System.Collections;


public      class      ExampleClass:      MonoBehaviour
{
    [SerializeField]
    private int    pruebaInt;
    [SerializeField]
    private string    pruebaString;
    [SerializeField]
     private float   pruebaFloat;
    [SerializeField]
    private bool   pruebaBool;

    void      Start()      {

        Debug.Log ("pruebaInt "  +  pruebaInt);
        Debug.Log ("pruebaString "  +  pruebaString);
        Debug.Log ("pruebaFloat "  +  pruebaFloat);
        Debug.Log ("pruebaBool "  +  pruebaBool);
    }
}
```

Esta estrategia es útil si, de acuerdo con el principio de encapsulación y ocultación de la información, deseamos declarar un campo de una clase como privado, pero queremos poder modificar su valor desde el editor del mismo modo que si fuera público. La regla general es que si es un tipo es serializable, se mostrará.

Nota

También es posible mostrar datos vía Inspector a través de funcionalidades más avanzadas como un *script* de editor (*editor script*).

3.4. Clases auxiliares

Unity proporciona una extensa colección de herramientas de *scripting* para facilitarnos la programación de nuestros proyectos. En este apartado enumeramos brevemente las más relevantes, con el simple propósito de dar a conocer su existencia, ya que explicarlas con gran detalle no tiene sentido pues para eso ya existe la documentación oficial. En ese sentido, los ejemplos incluidos solo sirven para dar una visión más clara del propósito de cada clase.

3.4.1. La clase «Input»

Esta clase nos permite incluir interactividad en nuestro proyecto. Todo aquello que el usuario haga con su dispositivo puede recogerse aquí. Veamos algunos ejemplos de cómo usar esta clase:

Web recomendada

Podemos encontrar la API de la clase «Input» en:

http://docs.unity3d.com/Manual/Input.html.

```
using    UnityEngine;
using    System.Collections;


public    class    ExampleClass:    MonoBehaviour    {
    void  Update()  {
        if  (Input.GetKeyUp(KeyCode.Space))  {
            Debug.Log("space pressed");
        }
    }
}
```

«KeyCode»

Esta clase contiene un conjunto de constantes que definen las distintas teclas («Space», «F1», «Asterisk», «A», etc.).

En este *script* vemos que el método que tenemos en la clase es «Update». Dentro de ese método encontramos una llamada a un método de la clase «Input», donde comprobamos si la tecla «Space» ('espacio') se acaba de levantar. Dado este *script*, observamos que, si solo está añadido a un *GameObject*, cada vez que apretamos la tecla de espacio el mensaje «space pressed» aparece en la consola.

Veamos otro ejemplo:

```
1.    using      UnityEngine;
2.    using      System.Collections;


3.    public      class      ExampleClass:      MonoBehaviour
4.    {
5.        void    Update()    {
6.            if(Input.touchCount  >  0  &&  Input.GetTouch(0).phase  ==  Tou
chPhase.Moved) {
7.                Debug.Log ("screen  touched");
8.            }
9.        }
10.   }
```

En este ejemplo estamos llamando a «Input» para obtener el estado de la pantalla táctil. En concreto, veremos el mensaje «screen touched» por la consola cuando movamos el primer dedo que haya tocado la pantalla táctil donde se está ejecutando nuestro proyecto. Obviamente, si el dispositivo que lo está ejecutando no tiene una pantalla táctil, esto no pasará nunca.

«Input» es una clase realmente extensa con multitud de funcionalidades. Siempre que necesitemos obtener algún tipo de dato de interacción del usuario no cubierta en este capítulo (giroscopio, brújula, localización, etc.), podemos acudir a la documentación de Unity.

Consejos sobre «Input»

Dado que Unity es un motor multiplataforma, debemos intentar abstraernos de los dispositivos en concreto. Por ejemplo, un usuario puede interactuar con nuestro juego usando la pantalla táctil cuando juega con un móvil, o el teclado y el ratón cuando juega con un ordenador. Si nos acostumbramos a crear una capa de abstracción propia donde accedamos a «Input» para comprobar tanto movimientos del ratón como de dedos en una pantalla táctil, seremos capaces de exportar nuestro proyecto a múltiples plataformas con muy poco esfuerzo.

Incluso aunque nuestro juego requiera de interacciones muy distintas si estamos jugando con un PC o con un móvil, es buena idea crear una clase auxiliar propia específicamente orientada a consultar «Input», y ejecutar un código propio distinto para cada plataforma. Esto lo podremos hacer, como veremos a continuación, con la clase «Application».

3.4.2. La clase «Application»

Esta clase nos permite acceder a una gran cantidad de información sobre el dispositivo en el que se está ejecutando nuestro proyecto.

Web recomendada

Podemos encontrar la API de la clase «Application» en: http://docs.unity3d.com/ScriptReference/Application.html.

Vamos a ver un ejemplo:

```
using UnityEngine;


public class ExampleClass: MonoBehaviour {
    void Start() {
        Debug.Log("The current platform is: " + Application.platform );

        if(Application.platform == RuntimePlatform.WindowsPlayer)
        {
            Debug.Log ("The current platform is Windows");
        }
    }
}
```

«RuntimePlatform»

Esta clase contiene un conjunto de constantes que definen las distintas plataformas (Android, PS3, Wii U, etc.).

En este ejemplo primero mostraremos por consola la plataforma en la que estamos trabajando. Después, si se da el caso de que estamos en Windows, enseñamos por pantalla un mensaje específico. Hay que tener en cuenta que dentro del editor no estamos en `RuntimePlatform.WindowsPlayer` sino en `RuntimePlatform.WindowsEditor`.

3.4.3. La clase «Time»

Hay bastantes ocasiones en las que queremos hacer operaciones que no dependan del número de *frames* que el dispositivo es capaz de generar por segundo. Por ejemplo, una animación debería empezar y acabar en el mismo momento, o un juego de carreras debería desplazar los coches según la velocidad del vehículo y no según los FPS. Hay que tener en cuenta que es bastante frecuente tener grandes variaciones en los FPS que conseguimos, sobre todo cuando cambiamos de perspectiva en la escena y hay partes con mucho detalle y otras muy sencillas. Para todo este tipo de operaciones tenemos la clase «Time», que nos permite controlar el tiempo discurrido entre distintos eventos, como puede ser entre dos *frames* consecutivos.

Web recomendada

Podemos encontrar la API de la clase «Time» en:
http://docs.unity3d.com/ScriptReference/Time.html.

Vamos a ver un ejemplo de uso:

```
using UnityEngine;
using System.Collections;


public class ExampleClass: MonoBehaviour {
    void Update() {
        transform.Translate(0, 0, Time.deltaTime*1);
    }
}
```

«Transform.Translate»

Este método provoca que el componente «Transform» de un objeto contemple un desplazamiento.

Este *script* causa a su *GameObject* asociado que se traslade una unidad por segundo en el eje *Z*. El motor llama a la función «Update» una vez por *frame*. En cada *frame*, el objeto se desplaza la cantidad de tiempo que ha pasado entre dos *frames*. Si tenemos FPS muy altos, `Time.deltaTime` será una cantidad muy pequeña a cada ejecución, por lo que se moverá muy poco, y viceversa. Independientemente de los FPS, solo se desplazará una unidad cuando haya pasado un segundo completo.

«Time.deltaTime»

Esta función indica el tiempo que ha transcurrido entre la ejecución del último *frame* y el actual.

La función «Time.deltaTime» es de gran utilidad ya que, usada como factor de corrección, permite ejecutar cualquier funcionalidad independientemente de los FPS reales de un juego.

3.5. Ciclo de vida y orden de ejecución de los MonoBehaviour

En los ejemplos que hemos ido viendo hemos añadido el *script* a solo un objeto, pero ¿qué pasaría si lo añadiésemos a más objetos? La respuesta es que cada objeto que tuviese el *script* añadido se comportaría de la misma manera y, por lo tanto, el mismo código se ejecutaría varias veces, una por objeto. Pero, ¿qué instancia se ejecuta primero? ¿Y si tuviéramos varios *scripts* distintos y quisiéramos que un *script* se ejecutase antes que otro? ¿Cómo podemos saber cuándo se va a ejecutar un *script*?

Unity está diseñado con la simplicidad en mente, a costa de perder un poco de control. La filosofía de programación es bastante distinta a la de programar otro tipo de aplicaciones. No tenemos manera de saber el orden en que se ejecutarán nuestros *scripts*, ni podemos establecer ninguna prioridad. Tampoco instanciamos clases con nuestro código propiamente, sino que añadimos *scripts* a *GameObjects* y es el motor el que se encarga de instanciar dichos *scripts*. Si bien esto con la práctica resulta en sencillez en buena parte de desarrollos, puede resultar extraño si tenemos experiencia programando otro tipo de aplicaciones.

A la hora de establecer el orden de ejecución, el motor se encarga internamente de tomar todas las decisiones y no tenemos una manera directa de intervenir. Una manera de acostumbrarse a esta mentalidad es imaginarnos que todos nuestros *scripts* se ejecutan realmente en paralelo, una vez por *frame*. En el diseño de nuestra aplicación debemos plantearnos que la mayoría de *scripts* deben ser lo más simples posibles y deben hacer solo una tarea concreta y completamente autocontenida. Por ejemplo, un *script* puede controlar cómo se mueve un enemigo, y otro controlar el desplazamiento para evaluar si está cansado. Aunque se podría hacer todo en un mismo *script*, por el modo de trabajar de Unity es mejor tener los dos por separado.

Sin embargo, hay casos en los que realmente necesitamos tener el control de qué se ejecuta antes, o enviar información o llamadas de un *script* a otro. En este caso podríamos echar mano de otras funciones:

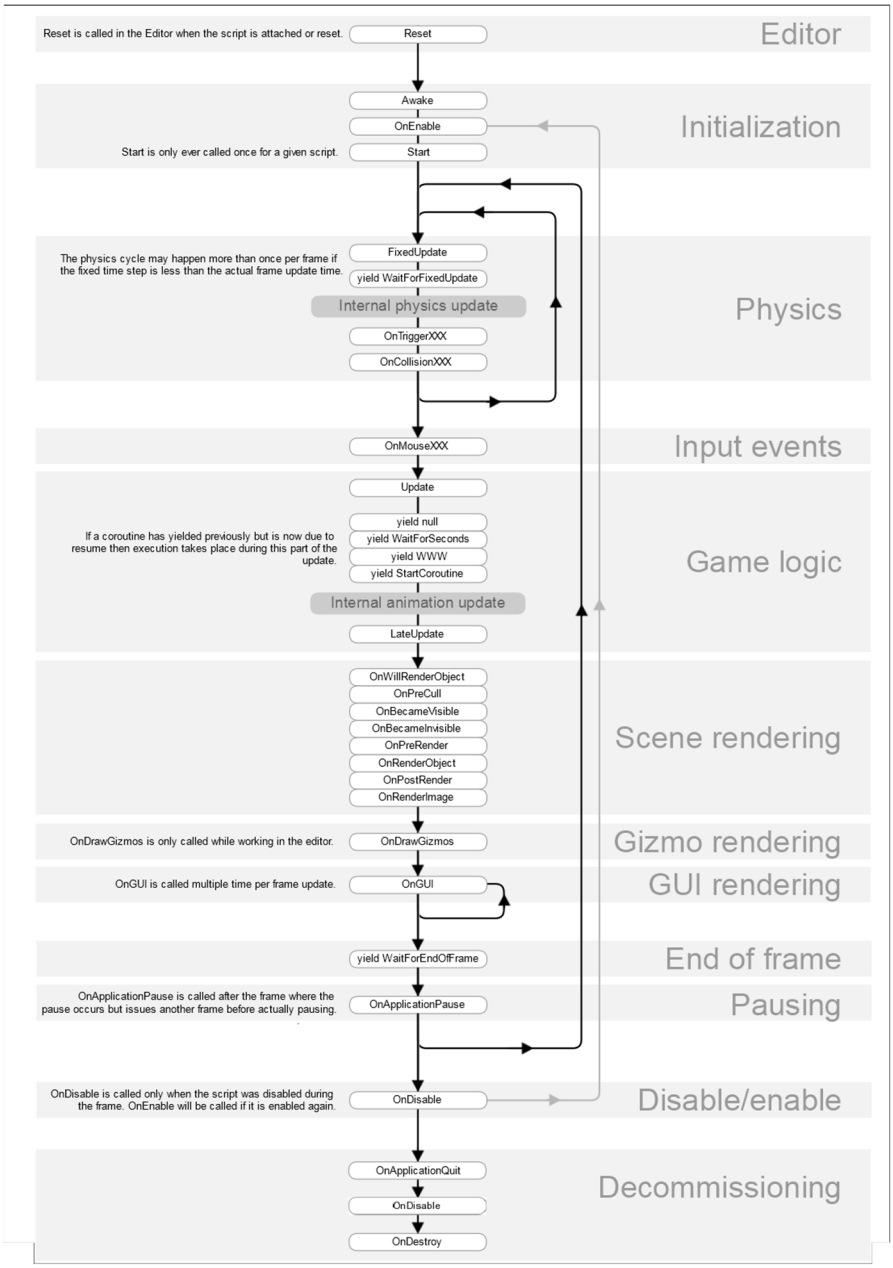
- «Awake», que se ejecuta una sola vez, inmediatamente tras cargarse la instancia del *script*.

Nota

«Awake» se ejecuta incluso antes de que se haya creado completamente el propio *GameObject* asociado.

- «LateUpdate», se ejecuta una vez por *frame*, igual que «Update», pero solo una vez ya se han ejecutado todos los métodos «Update» de todos los objetos.

De todos modos, hay que tener en cuenta algunas consideraciones. «Awake» tiene una firma menos flexible, por lo que se recomienda usar «Start» siempre que sea posible. La secuencia completa de ejecución de los métodos de un *script* desde el motor de Unity se puede encontrar en la documentación oficial, pero a título ilustrativo, para ver su complejidad real, la reproducimos a continuación:



Web recomendada

Este esquema y su explicación asociada se puede encontrar en:

http://docs.unity3d.com/Manual/ExecutionOrder.html.

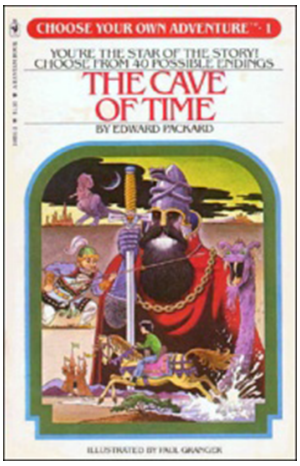
A nivel de este módulo, nos limitaremos a gestionar ciclo de ejecución basado en los métodos «Start» y «Update». Con ellos es más que suficiente para crear nuestros proyectos.

4. Proyecto: Un juego de aventuras

En este apartado crearemos el esqueleto de un juego sencillo paso a paso y de principio a fin. Se trata de una pequeña aventura de texto, basada principalmente en elementos de ***user interface*** (**UI**) o '**interfaz de usuario**'. En este pequeño juego, el usuario tendrá que explorar una habitación e interactuar con varios objetos para resolver un rompecabezas, al estilo de los antiguos libros de «Elige tu propia aventura». El objetivo es crear las bases para personalizarlo y ampliarlo a vuestro gusto.

CYOA

En los libros CYOA (*Choose Your Own Adventure*), el flujo de la línea narrativa depende de las elecciones del lector.



El objetivo no es llevar a cabo un juego complejo, sino simplemente elaborar un proyecto que sirva de hilo argumental para familiarizarse con el editor y empezar a conocer tanto algunos de los tipos de *GameObject* que ya ofrece Unity como sus componentes más sencillos, así como algunas funciones de propósito general muy útiles de cara a generar *scripts*. Para ello, esta sección toma la forma de un ejercicio paso a paso, en el que vamos proponiendo algunos retos que deberéis resolver.

4.1. Flujo del juego

Antes de empezar a desarrollar el proyecto propiamente dicho, vamos a analizar cuál será el flujo del juego para así tener claro qué herramientas necesitaremos.

Nota

Al final de la unidad están las soluciones a todos los retos.

La idea es crear un nivel de nuestro juego, pero diseñando nuestro proyecto de manera que se pueda expandir fácilmente en un futuro. Muchos juegos que tienen distintos niveles suelen tener una pantalla principal donde le preguntan al jugador si quiere ir a un nivel u otro, o si quiere cargar una partida anterior. Como vamos a tener un solo nivel, una pantalla de este tipo no es estrictamente necesaria, pero puede ser muy útil incluirla para luego poder expandir el juego a más niveles.

Nota

Al crear un juego, se recomienda empezar por la generación del recorrido básico entre escenas (pantalla de inicio, de juego, de final de partida, configuración, etc.).

Esta pantalla de inicio de momento solo necesita contener un par de botones, uno para llevarnos a la partida y otro para salir del juego. La pantalla de la partida contendrá básicamente elementos de UI, ya que es una aventura de tipo texto. Cuando el jugador acabe esta pantalla, debemos enseñarle algún mensaje para que quede claro que ha progresado en el juego o, en este caso, que lo ha terminado. Si tuviésemos más niveles, podríamos añadir opciones de guardar la partida o de saltar a otro nivel. En cambio, con solo un nivel de juego, este mensaje puede parecer irrelevante. Sin embargo, por el mismo motivo que anteriormente, es mejor añadir este elemento para poder ampliar el juego más adelante.

De esta manera, vemos que necesitaremos tres partes, la pantalla inicial, la pantalla del juego y la pantalla de final. Vamos a explicar las herramientas fundamentales para desarrollar todas ellas, empezando por la UI.

4.2. Creación de la interfaz de usuario básica

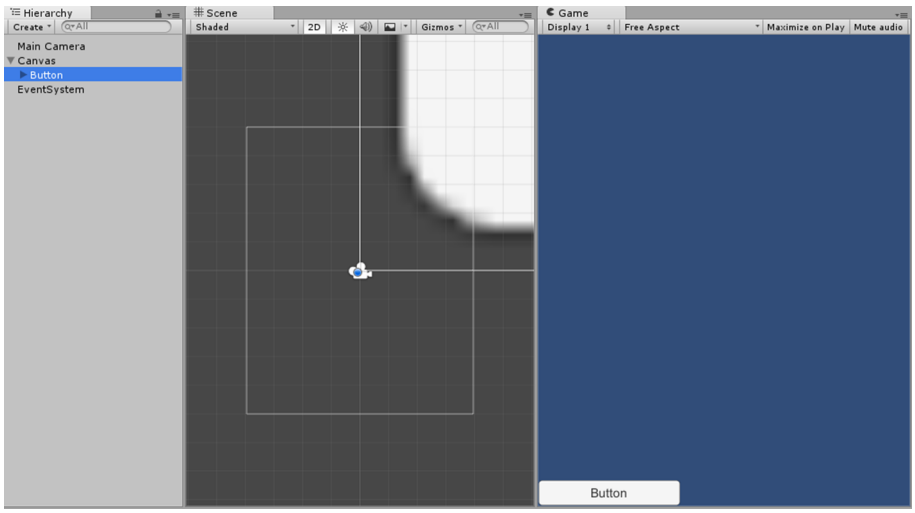
Una interfaz de usuario intuitiva e informativa es esencial para que el usuario sepa cómo manejarse dentro del juego. Una gran mayoría de los juegos empiezan con un menú principal desde donde se pueden cargar partidas o empezar partidas nuevas. Si bien hay juegos que han explorado maneras más originales de cumplir esta función, esta es una fórmula probada que sabemos que no va a fallar. Empezamos por generar esta pantalla.

Primero abriremos un proyecto vacío, recordando seleccionar el modo 2D. Entonces seleccionaremos la pestaña Game y la ubicaremos, arrastrándola, a la derecha de Scene, para trabajar más cómodamente. A continuación, añadiremos un botón mediante la opción de menú «GameObject > UI > Button».

Nota

En el menú «GameObject» se agrupan distintos tipos de objeto. Por ejemplo, «UI» para los de interfaz de usuario.

El resultado debería muy parecido a la siguiente figura, que será el punto de partida:



4.2.1. Elementos UI en Unity

Al añadir el botón, se han creado varios *GameObjects* en el apartado de Hierarchy. Uno de ellos lleva por nombre «EventSystem». Si lo seleccionamos, veremos en el Inspector que contiene dos *components*: un «Event System» y un «Standalone Input Module».

Por una parte, el componente «Event System» permite recibir la interacción del usuario, y simplemente necesitamos que exista uno en la escena para poder llevar a cabo esta función. Por otra parte, un «Standalone Input Module» se encarga de hacer la traducción de las acciones a eventos. Por ejemplo, en este componente podemos seleccionar cuál es el botón de cancelar una acción o

cómo se llaman los ejes verticales y horizontales. En general, si no estamos experimentando con controles específicos para nuestra aplicación, los valores por defecto serán adecuados para ambos casos.

Otro *GameObject* que se ha creado en la Hierarchy es el llamado «Canvas». Si lo seleccionamos y observamos el Inspector, lo primero que nos llamará la atención es que no tiene un componente «Transform», sino que uno llamado «RectTransform». En realidad, se trata de un caso específico de «Transform» que usan los elementos UI dentro de Unity. Internamente funciona igual que un «Transform», pero nos proporciona una manera más cómoda de interacción para el diseño de nuestra UI. Los siguientes tres *components* en este *GameObject* son «Canvas», «Canvas Scaler» y «Graphic Raycaster». El funcionamiento exacto de estos tres *components* es complejo, pero se puede resumir diciendo que nos permiten definir un área donde podemos añadir elementos UI, y que se encargan de calcular a qué elementos afecta la interacción del usuario. Por ahora, esto es suficiente.

Nota

Todos los elementos de la UI han de ser nodos hijo del «Canvas».

Finalmente, está el propio botón, el objeto «Button», que ha aparecido como elemento hijo de «Canvas». A su vez, ha aparecido un objeto «Text» como hijo de «Button». Los nombres son bastante explicativos.

«Text»

Este *GameObject* representa un texto en una UI. Como hijo de un botón, representa el texto que contiene.

Si observamos «Button» en el Inspector, vemos que tiene tres componentes: «Canvas Renderer», «Image» y «Button». «Canvas Renderer» es el componente que le indica a Unity que hay elementos visuales de una UI (por ejemplo, texto, una imagen, etc.) y, por lo tanto, hay que añadirlo para que se vean por pantalla. «Image» permite especificar la imagen que se mostrará en nuestra UI, mientras que «Button» es únicamente una manera de indicarle a Unity que este *GameObject* tiene una interacción de botón, pero no se encarga de mostrar nada por pantalla.

Nota

Cualquier *Gameobject* con un componente «Button» se comporta como un botón, respondiendo a pulsaciones.

Así pues, como vemos, para programar la UI en Unity reutilizamos y combinamos muchos componentes sencillos a través del editor, cada uno con una funcionalidad muy concreta, de modo que se van creando estructuras más complejas, siguiendo la filosofía de la simplicidad.

4.2.2. La pantalla inicial

Es el momento de empezar a trabajar en la pantalla de inicio del juego. Primero duplicamos el botón, y a continuación renombramos ambos botones con los nombres «Salir» y «Nueva partida». También cambiaremos el texto de cada botón por los mismos textos. Para ello, se debe editar el componente «Text» de cada uno.

Nota

Para duplicar un *GameObject*, se puede hacer cortar y pegar como en la mayoría de editores.

Finalmente, situaremos los botones centrados y separados de manera que no se tapen el uno al otro. Podemos arrastrarlos por el editor o asignar unas coordenadas concretas en el componente «Transform» de cada uno de ellos:

Nota

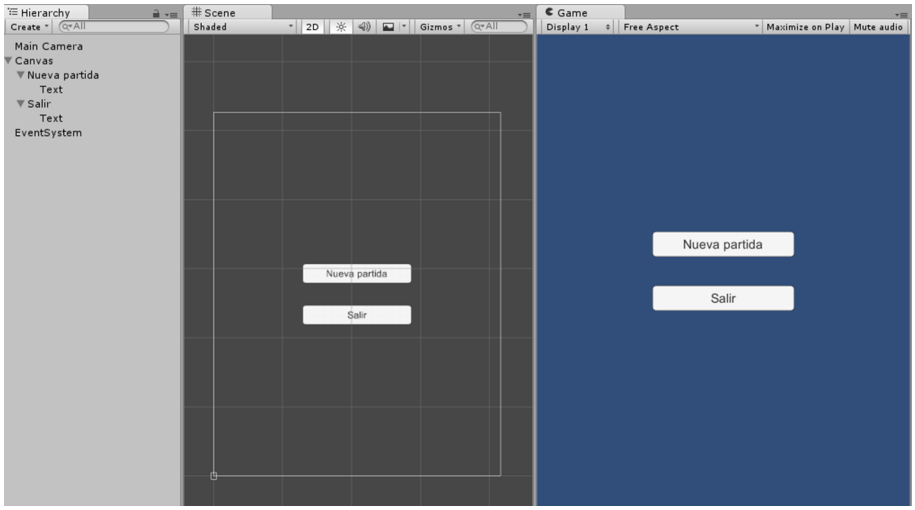
El nombre de un *GameObject* se cambia desde el Inspector.

- «Nueva partida»: «Pos X = 0», «Pos Y = 30» y «Pos Z = 0»
- «Salir»: «Pos X = 0», «Pos Y = -30» y «Pos Z = 0»

Nota

Para desplazar los elementos de la UI directamente en el editor hay que seleccionar el último elemento de la barra de interacción:

El resultado de estas operaciones es el siguiente:



En estos momentos, si entramos en modo ejecución (apretando «Play») veremos que los botones se muestran correctamente y que se pueden pulsar, aunque no tiene ningún efecto.

4.2.3. *Scripting* asociado a la UI

Para que nuestra UI lleve a cabo acciones, es necesario asociar el código correspondiente a cada objeto en forma de *script*. Empezaremos creando el *script* que responderá a la pulsación del botón «Salir», y que simplemente debe cerrar la aplicación. Sin embargo, debido a ciertas particularidades del editor de Unity, empezaremos por otra tarea.

Nota

Para crear un *script*, pulsar el botón derecho del ratón sobre el apartado de Assets (pestaña Project) en el editor y seleccionar «Create > C# Script».

Reto 1

Generad un *script* llamado «GameExit» con un único método llamado «SalirJuego» que cuando sea llamado provoque que se muestre por consola el mensaje «Saliendo del juego...».

Una vez generado el *script*, lo añadiremos como componente al botón «Salir», simplemente arrastrándolo sobre el Inspector. El *script* ahora forma parte de este objeto.

Nota

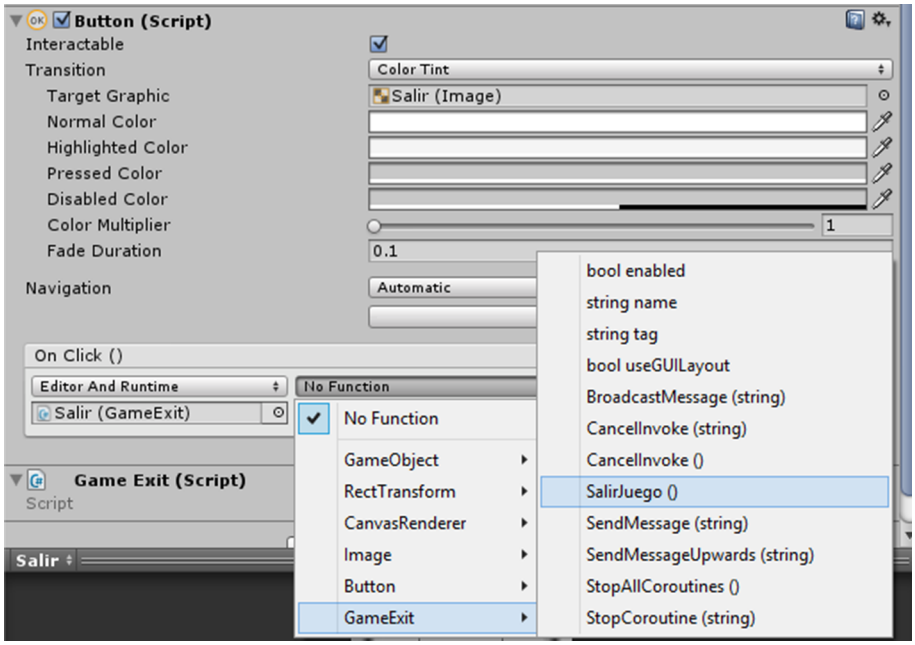
Los *scripts* se asocian como componentes de los objetos del juego.

Ahora es necesario asociarlo a la acción propiamente de pulsar el botón «Salir». Para asociar código a la acción de pulsar un botón se debe asignar vía un *listener*. Esto se lleva a cabo a través del componente «Button» del botón. En la parte inferior del Inspector aparece un recuadro donde se listan los *listeners* del evento «OnClick».

Nota

Un *listener* es el encargado de escuchar y controlar eventos.

Para añadir un *listener* se debe pulsar el símbolo «+»; entonces, aparece un apartado donde se puede indicar el *listener*, inicializado a «None (Object)» (ninguno). Sobre este cuadro se debe arrastrar desde la Hierarchy el objeto que posee el código a ejecutar. En nuestro caso, el objeto que contiene el *script*, que es el propio botón «Salir». Una vez arrastrado, es posible seleccionar en la caja desplegable contigua el método que se ha de ejecutar. Para ello, primero se debe buscar el componente adecuado (en este caso, nuestro *script* «GameExit»), y luego el método, tal como muestra la imagen:



Como podemos comprobar, Unity es consciente de los métodos que contienen los *scripts*, siempre que sean públicos y que tengan una firma compatible con el evento (que tengan los mismos parámetros y que retornen *void*). En concreto, lo que hace Unity es enumerar los componentes del objeto que se seleccione y listar todas las funciones compatibles. Gracias a esta funcionalidad, podemos trabajar muy cómodamente con *scripts* muy sencillos y muy concretos, y separando el *scripting* del editor.

Para probar que el proceso funciona correctamente, podríamos entrar en modo ejecución y pulsar el botón «Salir». Una vez vemos que funciona, podemos generar el código que realmente provocaría la salida del programa.

Reto 2

Modificad el *script* que realmente detendría el juego al pulsar el botón «Salir». Para esta tarea, estudiad la documentación de la clase «Application».

El motivo por el que hemos llevado a cabo esta tarea en dos pasos es que la llamada al método que sale de la aplicación no tiene efecto dentro del editor de Unity. Por lo tanto, no es posible probar realmente su funcionamiento en modo ejecución.

Llega el momento a hacer la funcionalidad del otro botón, el de «Nueva Partida». Antes, sin embargo, será necesario preparar el proyecto para que exista una pantalla de juego (*scene*), que cargaremos al pulsar dicho botón.

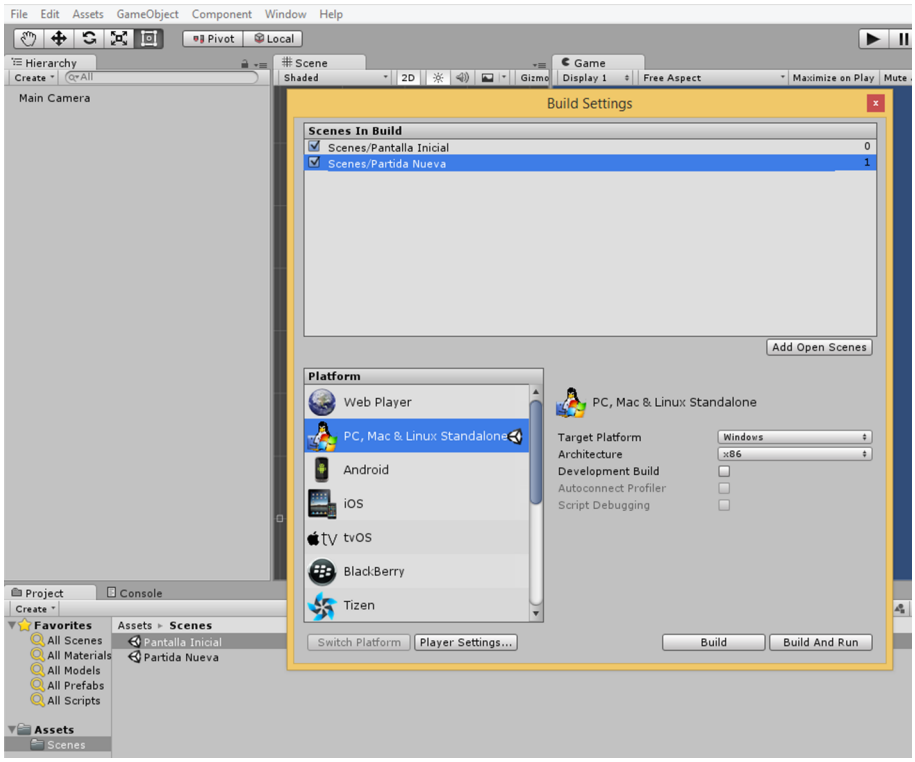
Para definirla, debemos guardar primero la *scene* actual mediante la opción de menú «File > Save Scene», con el nombre «Pantalla Inicial». La recomendación es guardar las *scenes* juntas, en una carpeta Scenes, para mantener un orden dentro del proyecto.

Seguidamente, crearemos una escena nueva con el menú «File > New Scene». Al hacerlo, aparece una nueva pantalla vacía. La guardamos con el nombre «Partida Nueva». De esta manera, Unity ya podría discriminar y cargar distintas pantallas según su nombre. Sin embargo, aún no hemos terminado, antes debemos indicar un orden de las pantallas y, especialmente, indicar cuál es la primera que ha de cargar el juego al iniciarse.

La gestión de *scenes* en un proyecto se lleva a cabo mediante el menú «File > Build Settings». En esta opción podemos indicar distintas opciones sobre la generación del juego, como puede ser la plataforma destino. En la parte superior («Scenes In build») se encuentra la lista de *scenes* que se considera que forman parte del juego. A este campo le podemos arrastrar cualquier *scene* desde el apartado de Assets del editor. La primera de la lista será la primera que se cargará al iniciarse el juego. El resto, se cargan solo cuando lo indiquemos a través de código en un *script*. Pero solo será posible cargar *scenes* que se encuentren en esta lista explícitamente.

Nota

Unity soporta diversas plataformas.



¿Qué sentido tiene esta lista, más allá de la primera *scene*? Es bastante frecuente en proyectos hacer escenas de prueba que no queremos que acaben en la versión final. También hay muchos *assets* en la Asset Store que incluyen *scenes* para demostrar cómo se configuran o cómo pueden mostrarse, que tampoco nos interesa que acaben en nuestro ejecutable final, porque podrían inflar innecesariamente nuestra aplicación. Así pues, de este modo podemos indicar qué acaba realmente en el jugo final.

Una vez completado este pasó, Unity ya podrá cargar nuestras *scenes* correctamente.

Nota

Cada vez que se carga una nueva *scene*, se destruyen todos los objetos de la anterior.

Reto 3

Haced que al pulsar el botón de nueva partida se cargue la *scene* «Partida Nueva». Para ello, generad un *script* llamado «NuevaPartida» con un único método llamado «CargarPartidaNueva» que lleve a cabo esta acción. Para esta tarea, estudiad la documentación de Unity sobre clase «Application».

Recuerda que deberás asociar el método al evento de pulsación del botón mediante el *listener* correspondiente.

Reto 4

Pensad que deberíais hacer si, en vez de usar botones, quisierais usar texto (un *GameObject* tipo «Text», en vez de «Button») y que el usuario pudiera iniciar partida o salir del juego pulsando sobre él.

Ahora que ya sabemos cómo interactuar con una UI simple y cambiar de *scene*, vamos a desarrollar la pantalla de juego de nuestro proyecto.

4.3. La pantalla de juego

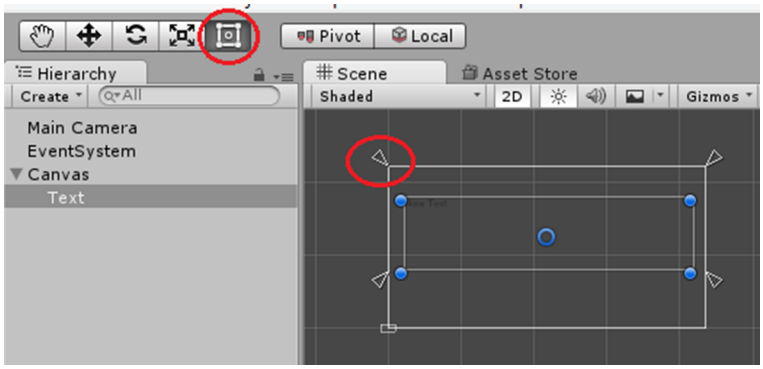
Una pantalla dentro de la partida se compondrá básicamente de dos secciones de elementos UI, una en la que tendremos el texto que describe nuestra situación y otra en la que tendremos todas las opciones posibles de interacción. Vamos a construir estas dos secciones en la escena «Partida Nueva», que corresponderá a la pantalla del juego.

4.3.1. Configuración de la pantalla de juego

Primero vamos a Hierarchy y desplegamos el menú contextual con el botón derecho del ratón sobre el «Canvas». Mediante la opción «UI > Text» podemos crear un elemento UI de texto. Para hacer que ocupe el espacio que nos interesa, podemos usar las herramientas señaladas en la figura. La señalada en la parte superior sirve para poder definir el tamaño de los distintos elementos de UI en el editor de escena, a partir de los puntos azules remarcados. La herramienta señalada con forma de triángulo nos sirve para definir el anclaje del campo de texto. Especialmente en los dispositivos móviles, hay una gran diversidad de tamaños de pantallas y de proporciones; por ello, debemos situar los anclajes correctamente.

Nota

El anclaje (*anchoring*) de un elemento indica su comportamiento cuando se redimensiona la pantalla.



Vamos a añadirle también un fondo. Esta vez añadimos «UI > Image». Ajustamos el tamaño para que sea ligeramente más grande que el texto y ajustamos los anclajes para que sean los mismos que los del texto. Ahora podemos emparentar el texto a la imagen por comodidad. Podemos también añadirle un fondo para que tenga los bordes un poco suavizados. Para hacerlo, se debe asignar a la propiedad «Source Image» de su componente «Image». Podemos pulsar sobre el campo para abrir un selector, o bien arrastrarla directamente desde al apartado de Assets sobre el campo.

Nota

Para poder añadir una imagen al objeto «Image», este antes ha de estar incluido dentro de los *assets* del proyecto.

Reto 5

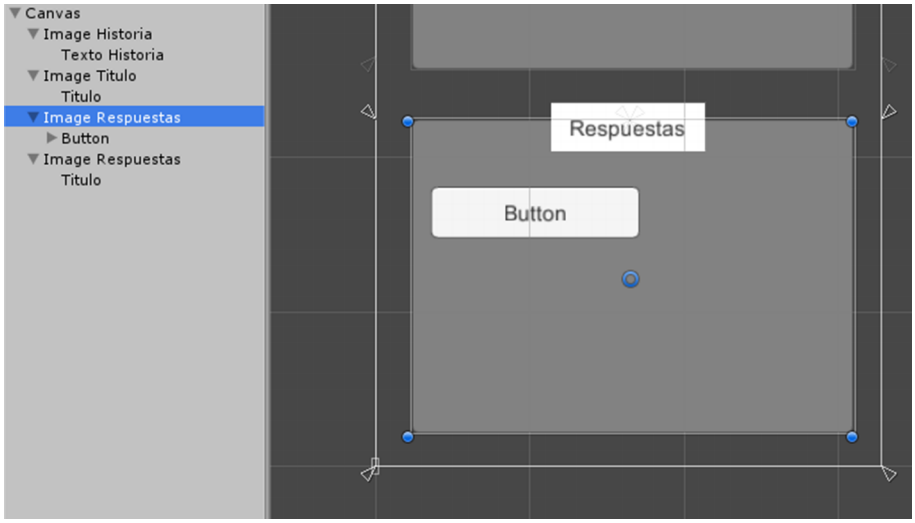
Buscad un fichero con una imagen de fondo que os guste y ponedla como fondo de la pantalla usando el objeto «Image».

Nota

En nuestro ejemplo ponemos simplemente un fondo gris.

Además, vamos a añadir un pequeño título para que el jugador sepa visualmente qué significa esta sección. Podemos duplicar los *GameObjects* que hemos creado y ajustar el tamaño, para que se vean como un simple detalle, y la posición, para que queden centrados y en la parte superior. Si renombramos los *GameObjects* con nombres que reflejen el significado de lo que representan, debería quedarnos una estructura como la de la siguiente figura:

Ahora vamos a crear la sección de respuestas. El proceso será muy similar al anterior, pero situaremos todo en la parte inferior de la pantalla. También crearemos un botón, que hará de modelo de posible respuesta. Existirá un botón por respuesta posible en cada escena. El anclaje de este primer botón lo pondremos en la esquina superior izquierda de la zona de respuestas.



De este modo, tenemos los siguientes objetos de UI en la escena:

- «Texto Historia»: el texto que describe la situación actual al jugador.
- «Títulos»: los títulos de la escena y el apartado de respuestas.

- «Button»: los botones de las respuestas posibles, que servirán para pasar a nuevas pantallas (por ahora, solo uno).

A cada uno de estos objetos se le asocia una imagen de modo que, por su relación en la Hierarchy, si se desplaza la imagen, también se desplaza con ella el elemento de UI correspondiente.

Finalmente, vamos a generar un *prefab* a partir del botón. La idea es que el texto de la historia puede darnos varias opciones de respuesta, y queremos poder instanciar un botón por cada una. Para hacerlo, la opción más cómoda es usando *prefabs*. El tamaño del botón lo hemos dejado en «Width = 150» y «Height = 45», que parece un poco grande, pero así permitimos respuestas un poco largas.

Nota

Para crear un *prefab* fácilmente, basta con arrastrar el objeto de la Hierarchy al apartado de Assets. Su nombre quedará en azul.

Ya tenemos una primera configuración de la escena, pero aún quedan algunos problemas que resolver. El primero es que el texto o los bordes de las imágenes puede que se vean un poco borrosos. Esto se puede solucionar marcando la opción «Pixel Perfect», en el componente de «Canvas», dentro del *GameObject* con el mismo nombre.

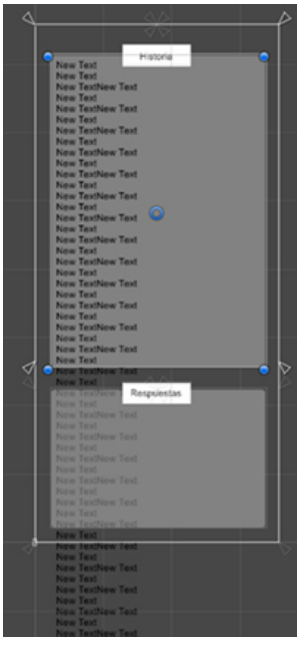
Nota

La propiedad «Pixel Perfect» de un componente «Image»:

Otro problema surge cuando el texto es demasiado largo y no cabe en el espacio que hemos dejado para el texto de la historia. Necesitamos entonces configurar una función de *scroll*. Para ello, primero agrandamos el *GameObject* «Texto Historia» hacia abajo, de manera que sobresalga claramente de la imagen que delimita el texto de la historia (ver figura al lado). Esto, como es de esperar, causa que el texto se vea en zonas donde no debería verse, pero es un paso previo a la solución final.

Nota

Redimensionando el apartado de texto:



Para solucionar el problema, vamos a crear un *GameObject* hijo de «Image Historia» y hermano de «Texto Historia». A este nuevo *GameObject*, que vamos a llamar «Image Mascara», le vamos a añadir una imagen, mediante un *componente* «Image». Lo importante es que se defina el tamaño de manera que delimite dónde se tendría que ver texto únicamente, es decir, debe ser un tamaño ligeramente menor que el del *GameObject* «Image Historia». Al *GameObject* «Image Mascara» le añadimos un componente «Scroll Rect» y otro «Mask». El primero le indica a Unity que todo el contenido que se asigne al campo «Content» se puede desplazar en modo *scroll*. El segundo limita la visualización de un objeto hijo en la Hierarchy, de modo que nunca ocupe más tamaño que el objeto padre. Por lo tanto, tendremos que emparentar el contenido que queramos delimitar con «Image Mascara».

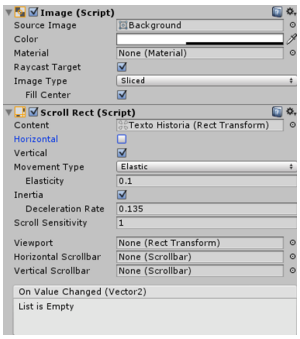
En estos dos componentes, «Scroll Rect» y «Mask», hay que configurar algunas propiedades. De entrada, el componente «Image» está pintando una imagen que realmente no nos interesa mostrar y que está tapando el contenido que hay debajo. Para corregir esto, en el componente «Mask» desmarcamos la op-

ción «Show Mask Graphic». Ahora que ya le hemos indicado a Unity cómo funciona el *scroll* y qué zonas se deben ver, debemos emparentar correctamente el contenido. Como hemos dicho, los elementos hijos de «Image Mascara» son los que se verán correctamente. Por lo tanto, debemos emparentar el *GameObject* «Texto Historia» al «Image Mascara», y entonces el texto ya sí que solo se debería ver en las zonas que nos interesan.

Por otra parte, la propiedad «Content» del «Scroll Rect» se debe asignar al *GameObject* «Texto Historia» (por ejemplo, arrastrando desde la Hierarchy) para indicarle a Unity que este es el elemento que funciona en modo *scroll*. En este mismo componente también deseleccionamos el campo «Horizontal», para que el *scroll* solo sea vertical. Con todo esto, si al ejecutar el juego ponemos un texto muy largo, podremos desplazarlo arrastrando el contenido con el ratón. Por otro lado, es habitual definir barras de *scroll* para poder operar de una manera más intuitiva.

Nota

Configuración del componente «Scroll Rect»:

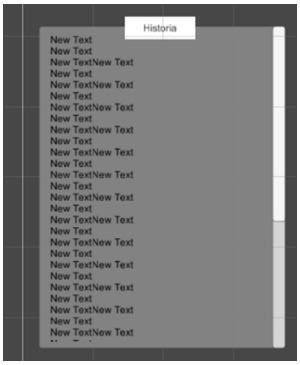


Por comodidad, se suelen poner las barras a los lados, para poder hacer desplazamiento fácilmente. Esto se puede hacer añadiendo un objeto «UI > Scrollbar» en el «Canvas», que representa una barra de desplazamiento. Inicialmente, se nos mostrará una barra horizontal, que no es adecuada para nuestro diseño. En el campo «Direction», en su componente «Scrollbar», podemos seleccionar el sentido de la barra. En nuestro caso nos interesa «Bottom To Top».

Ahora debemos asignar este *GameObject* al campo «Vertical Scrollbar» del componente «Scroll Rect» que hemos creado. Finalmente, ajustamos el tamaño y la posición de la barra de *scroll* para que quede situada cómodamente en la parte derecha. Con todas estas operaciones deberíamos tener un texto con desplazamiento, y probar el correcto funcionamiento apretando «Play».

Nota

Texto con barra de desplazamiento vertical:



En general es una buena idea tener los *GameObjects* bien organizados y por categorías. En ese sentido, a veces conviene tener *GameObjects* vacíos (*empty*), que no tienen ninguna función en el juego pero que hacen posible agrupar otros objetos de manera que formen una estructura que permita tener ordenados los elementos de juego por grupos. Esto mejora la legibilidad del proyecto.

Reto 6

Reorganizad la jerarquía usando objetos vacíos, de modo que al final quede todo ordenado según la figura de debajo. Dividiremos los elementos de la pantalla de juego según dos apartados: «Seccion Historia» y «Seccion Respuestas».

Organización de la jerarquía de objetos en el proyecto.

4.3.2. Control del juego

Ahora que ya tenemos todos los elementos de la UI, ya podemos empezar a crear la lógica del juego. Cuando tratamos lógica que no está claramente asociada a un objeto concreto en la interfaz (por ejemplo, los botones de la pantalla de inicio), lo normal es crear un *GameObject* vacío que sirve como controlador o gestor del juego; este *GameObject* vacío no tiene visualización en pantalla, solo sirve para agrupar ciertos *scripts* generales. En este caso, llamaremos a dicho objeto «GameplayManager» y contendrá el *script* con el código.

La estrategia a seguir para controlar el juego va a ser la siguiente. Si bien podríamos generar tantas pantallas como localizaciones tiene nuestro juego (si nuestro juego tiene diez localizaciones, crear diez *scenes* en el proyecto), dado que todas las pantallas van a tener un aspecto idéntico salvo en el texto explicativo y las opciones disponibles, lo que haremos es trabajar sobre una sola pantalla de juego («Pantalla inicial») y cambiar el contenido de los cuadros de texto según corresponda. No tiene sentido ir replicando exactamente la misma *scene*, ya que sería un trabajo sumamente tedioso e ineficiente.

Nota

Las localizaciones del juego se corresponderían, de hecho, a las páginas de un libro de «Elige tu propia aventura».

De cara generar el código que controle las distintas localizaciones del juego, el punto de partida va ser una estructura de datos en forma de grafo dirigido que representa el mapa de localizaciones. Cada vértice es una localización y sus aristas salientes son las distintas opciones, cada una apuntando a su vértice destino.

El código del *script* que define esta estructura de datos va a ser el siguiente:

```
1.  using UnityEngine;
2.  using UnityEngine.UI;
3.  public class GameplayManager: MonoBehaviour {
4.
5.      public class StoryNode {
6.          public string history;
7.          public string[] answers;
8.          public StoryNode[] nextNode;
9.          public bool isFinal = false;
10.          public delegate void NodeVisited();
11.          public NodeVisited nodeVisited;
12.
13.     }
14.
15.      private StoryNode current;
16. }
```

Este *script* define una estructura de datos en forma de la clase StoryNode. A partir de esta, generaremos el grafo y todo el contenido de las localizaciones. Sus atributos son:

- `history`: El texto que describe la localización.
- `answers[]`: La lista de textos con las respuestas posibles.
- `nextNode[]`: La lista de nodos destino, en el mismo orden que en la lista de respuestas. O sea, si se elige la respuesta en el índice 1 de «answers», el jugador se desplazará al nodo indicado por `nextNode[1]`.
- `isFinal`: Indica que este nodo corresponde a un final, donde acaba la aventura.
- `NodeVisited()` y `nodeVisited`: Permite asignar una referencia a un método delegado que se ejecutará siempre que se visite esa localización. Esta estrategia permite ejecutar código que manipule el mapa al llegar a ciertas localizaciones.

Nota

Para asignar un método delegado a una referencia de este tipo, la sintaxis es:

```
nodeVisited = () => {
  //código
};
```

Modificando el mapa en tiempo de ejecución

En algunos momentos del juego va a ser necesario modificar el mapa en tiempo de ejecución. Por ejemplo, una ubicación puede tener una puerta cerrada, que solo se va a poder abrir si se ha cogido la llave correspondiente.

En un libro de «Elige tu propia aventura» esto se resuelve normalmente con un texto al estilo «Si tienes la llave, ves a la página..., en caso contrario, ves a...». En nuestro juego la estrategia es la siguiente. Se van a crear dos nodos, uno que indica que la puerta no se puede abrir (N1) y otro que representa que se abre con éxito (N2). Inicialmente, el nodo asignado en «nextNode» a la opción de abrir la puerta es N1. Será en la localización que representa que se coge la llave donde asignaremos el método delgado que reemplace N2 por N2, de modo que, a partir de entonces, al llegar a la puerta, la acción de abrir tendrá éxito automáticamente.

Además, en la clase GameplayManager existe el atributo `current`, que indicará en todo momento en qué nodo se encuentra actualmente el jugador. El valor que se le asigne al iniciar el juego será la primera localización.

Nuestra primera tarea será inicializar el mapa de localizaciones. En este ejemplo, para simplificar, lo haremos de manera estática directamente mediante código. Evidentemente, esta solución no es escalable ni la mejor en un pro-

yecto real. Sería mejor cargarlo desde un fichero de datos estructurado, por ejemplo. Sin embargo, nuestra opción a veces es una buena manera de generar prototipos rápidamente para comprobar que todo funciona; más tarde se puede reformular si se considera necesario.

La inicialización la llevará a cabo otro *script* que llamaremos «StoryFiller» y que definirá una clase auxiliar que no se asignará a ningún objeto. De este modo se separa la gestión del juego de la carga de datos, manteniendo así el principio de encapsulación de código.

```
1. using System;
2. public class StoryFiller {
3.
4.     public static GameplayManager.StoryNode FillStory() {
5.         //Código de inicialización
6.
7.     }
8. }
```

Nota

Al no tratarse de un *script* asociado como componente a un objeto, «StoryFiller» no hereda de MonoBehaviour.

Su valor de retorno es el nodo inicial, la que será la primera localización del juego.

Reto 7

Rellenad el código del método «FillStory» de modo que generéis un mapa de localizaciones para el juego. Será necesario ir creando los nodos dinámicamente e inicializar sus atributos de algún modo. Si el método delegado da problemas, podéis ignorar esta posibilidad. Simplemente, el mapa será estático a lo largo de todo el juego.

Una vez tenemos el código que carga el mapa del juego, podemos llevar a cabo el contenido del *script* «GameplayManager», que es el que realmente va gestionar la pantalla de juego. Su función principal va a ser actualizar la interfaz de usuario con los datos asociados a la ubicación actual del jugador. Principalmente, actualizar los textos en pantalla, eliminar los botones actuales y crear los nuevos que correspondan, según el número de acciones disponibles en dicha localización.

Para destruir y crear objetos dinámicamente, Unity ofrece dos métodos. Por una parte, «Destroy» elimina el objeto pasado como parámetro. Por otra parte «Instantiate» crea objetos a partir de uno ya existente, que es replicado. En este caso, sus parámetros son una referencia al objeto a copiar, su posición y su rotación. Vale la pena destacar que dicho método genera copias de objetos, no los crea realmente desde cero. Normalmente lo que se hace no es copiar realmente un objeto ya preexistente en la Hierarchy (lo que podría ser un problema si no hay ningún otro objeto del mismo tipo aún creado), sino usar un *prefab*.

Nota

Podéis ver todos los detalles de ambos métodos en la documentación oficial.

Accediendo a *prefabs* desde el código de un *script*

Los *prefabs* se crean desde la pantalla del editor y se guardan en forma de *asset* en el apartado Project. Una manera muy sencilla para que vuestro código pueda acceder a dicho *prefab* es la siguiente. Primero, crear un atributo público de tipo *GameObject* en

vuestro *script*. Esto hace que en el Inspector, dentro del componente «Script», aparezca un campo asignable. Entonces, arrastrar el *prefab* de Project a dicho apartado. A partir de ese momento, siempre que se ejecute el *script*, en dicho atributo estará disponible el *prefab*.

En general, recordad que todo atributo declarado público en el código de un *script* aparece como un apartado asignable en su componente en el editor, que puede ser inicializado arrastrando el elemento correspondiente. Por ejemplo, si queremos tener un modo de acceso fácil a un elemento de la UI[1] o a alguno de sus componentes, podemos hacer exactamente lo mismo. Solo habrá que definir el atributo del tipo que corresponda (por ejemplo, «Text» para un campo de texto, «Transform» para una componente «Transform», etc.). Unity dispone de tipos de datos para prácticamente todos los elementos asociados a objetos en el editor.

(1)Las clases que representan los elementos de una UI en Unity están en la biblioteca «UnityEngine.UI».

Una vez creado un objeto a partir de un *prefab*, seguramente querremos personalizar algunas de sus propiedades. Para ello, deberemos modificar sus componentes, también dinámicamente. Para acceder a una componente de un *GameObject* es necesario invocar su método «GetComponent<TipoComponente>()». Por ejemplo, para acceder al componente «RectTransform» de un elemento de UI, se ejecutaría «GetComponent< RectTransform>()». Una vez se dispone del componente, es posible acceder y manipular todas sus propiedades a través de los métodos y atributos públicos que ofrece.

Nota

En versiones de Unity anteriores a la 5 la sintaxis varía para algunos componentes concretos.

Otro aspecto que deberemos gestionar dinámicamente es añadir *listeners* a elementos de UI (en este caso, los nuevos botones), de modo que cada nuevo botón nos desplace a la ubicación que corresponda. En este sentido, los componentes de UI que esperan *listeners* disponen de atributos que permiten asignar referencias a métodos. Por ejemplo, el componente «Button» dispone de la variable «onClick», que a su vez dispone del método «AddListener».

Nota

Recordad que el *listener* se añade a un componente «Button» de un *GameObject*, no al objeto en sí.

Un ejemplo de código asociado a esta operación sería:

```
1.    Button buttonComp = myButton.getComponent<Button>();
2.
3.    buttonComp.onClick.AddListener( () => {
4.       //código del listener
5.    });
```

Nota

Básicamente, es la misma sintaxis que en los métodos delegados.

Finalmente, otro mecanismo muy útil a la hora de gestionar objetos a través de un *script* es poder recorrer los elementos de su jerarquía de acuerdo con la estructura mostrada en la Hierarchy del editor. Esta tarea se puede llevar a cabo mediante el componente «Transform» de un objeto:

```
6.    Transform parent = myObject.getComponent<Transform>();
7.
8.    foreach(Transform child in parent) {
9.        //código que manipula sus hijos vía la variable “child”
10.   }
```

Nota

Recordad que toda manipulación del componente «Transform» de un objeto se hereda a todos los hijos en la jerarquía.

Relacionado con este mecanismo de recorrido, el componente «Transform» posee también el atributo `parent`, que contiene su padre en la jerarquía. Asignando el valor correspondiente es posible crear relaciones padre-hijo.

Nota

El atributo `gameObject` contiene el objeto asociado al «Transform» en cuestión.

Reto 8

Completad la clase «GameplayManager». Primero, debe encargarse de generar el mapa al inicio de la partida. Pensad qué método debe incorporar dicho código. Inmediatamente después, debe inicializar la UI con los datos de la primera ubicación. A partir de aquí, debe encargarse de actualizar la ubicación del personaje y mostrar los nuevos datos en la UI cada vez que se pulsa alguno de los botones que enumeran las acciones disponibles.

4.4. La pantalla de fin de juego

Hay varias opciones para decirle a un jugador que el juego se ha acabado. Diversos juegos nos han demostrado que puede haber una gran variedad en este aspecto, desde dejarnos rebobinar en el tiempo hasta pantallas específicas relacionadas con la muerte. Sin embargo, la mayoría de juegos optan por una solución más sencilla y menos creativa, aunque conocida por cualquier jugador con un mínimo de experiencia. Esta solución es simplemente un mensaje que le indica al jugador que la partida se ha acabado y le muestra varias opciones, que pueden ser repetir la partida o ir hacia otro menú.

En nuestro caso, ya que tenemos una pantalla inicial, podemos mostrar un simple mensaje y dar la opción de reiniciar la partida o de volver a la pantalla principal. Vamos a crear una UI sencilla, con dos botones, según la figura. El botón de la izquierda queremos que nos lleve a la pantalla inicial, mientras que el botón de la derecha reinicia directamente la partida.

Reto 9

Cread esta pantalla, de manera similar a cuando elaborasteis la pantalla de inicio, y haced que «GameplayManager» la muestre al acabar el juego.

Aprovechando que tenemos una pantalla que permite volver al menú de inicio o reiniciar una partida, podemos incluir una funcionalidad de pausa en el juego, de manera que cuando se pulsa la tecla «Esc», aparezca esta pantalla. Así, en cualquier momento podemos salir de la partida de manera limpia.

Reto 10

Añadid la opción de cancelar la partida con la tecla «Esc».

Ahora, tan solo tenemos que asociar los eventos «OnClick», el del botón «Menú Inicial» al método «LoadStartScene», y el del botón «Reiniciar partida» a «RestartScene». También tenemos que añadir una variable pública de tipo *GameObject* en el *script* «GameplayManager», que asociaremos en el editor a la UI que acabamos de crear. Finalmente, esta UI la debemos posicionar en el centro de la pantalla y desactivarla, para que el jugador la vea cuando se activa, pero que esté invisible por defecto.

Como detalle, podemos observar que esta UI es perfectamente válida para cuando pausamos el juego. De esta manera, hemos escogido la tecla «Escape» para enseñarla, como se puede ver en el método «Update».

4.5. Incorporación de efectos sonoros

El sonido es una parte fundamental en la gran mayoría de juegos. Muchos jugadores recuerdan perfectamente tanto la música de fondo como los efectos sonoros de sus juegos favoritos. Por lo tanto, es un aspecto que no debemos olvidar. Unity nos ofrece una gran cantidad de herramientas para poder dotar a nuestro proyecto de un sonido de calidad. Vamos a dar un pequeño repaso de cuáles son estas herramientas antes de añadir el sonido a nuestro proyecto.

Todos los *assets* de sonido que importemos a nuestro proyecto (ficheros .mp3, .ogg, entre muchos otros) se convierten en *Audio Clips* ('clips de audio'). De esta manera, abstraeremos el tipo de fichero y de compresión, y trabajaremos con el sonido de una única manera.

Nota

Añadiremos un «AudioSource» como componente de los objetos que generan sonidos.

El elemento básico para generar sonidos es el *component* «AudioSource», con el que podemos hacer sonar un *Audio Clip* asignado a la propiedad correspondiente. El origen del sonido estará allí donde esté situado el *GameObject* con dicho componente. En este tipo de componente, por defecto, la opción «Play On Awake» de los «AudioSource» está activada. Esto provoca que el sonido se ponga en marcha en el momento en que se crea el objeto, lo que puede ser adecuado para música de fondo, pero no para efectos de sonido en momento puntuales (conviene desactivarla). La opción más importante es el *Audio Clip* que le asignamos para que se escuche, y también si queremos que se repita automáticamente, lo que se consigue mediante la opción «Loop».

Otro aspecto a tener en cuenta es el *component* que indica a Unity desde dónde queremos escuchar los sonidos: «AudioListener». En este sentido, hay que tener en cuenta que Unity hace una simulación de la propagación del sonido a través del espacio, de manera que lo que esté más lejos sonará más flojo. De todos modos, este *component* se añade normalmente en el mismo *GameObject* en el que está la cámara, de la misma manera que nuestros ojos y nuestras orejas están cerca los unos de los otros. Una *scene* suele tener como máximo uno activado a la vez.

Nota

El resto de propiedades de «AudioSource» y «AudioListener» se puede consultar en la documentación de Unity.

De cara a aplicar estos componentes en nuestro proyecto, primero será necesario explorar nuestros recursos para obtener sonidos. Un buen punto de partida es la Asset Store, buscando sonidos de ambiente gratuitos.

Reto 11

Añadid una música de fondo para cada una de las pantallas del juego. De cara a generar los sonidos, podéis ubicar su origen («AudioSource») en la propia cámara, junto con el «AudioListener».

Nota

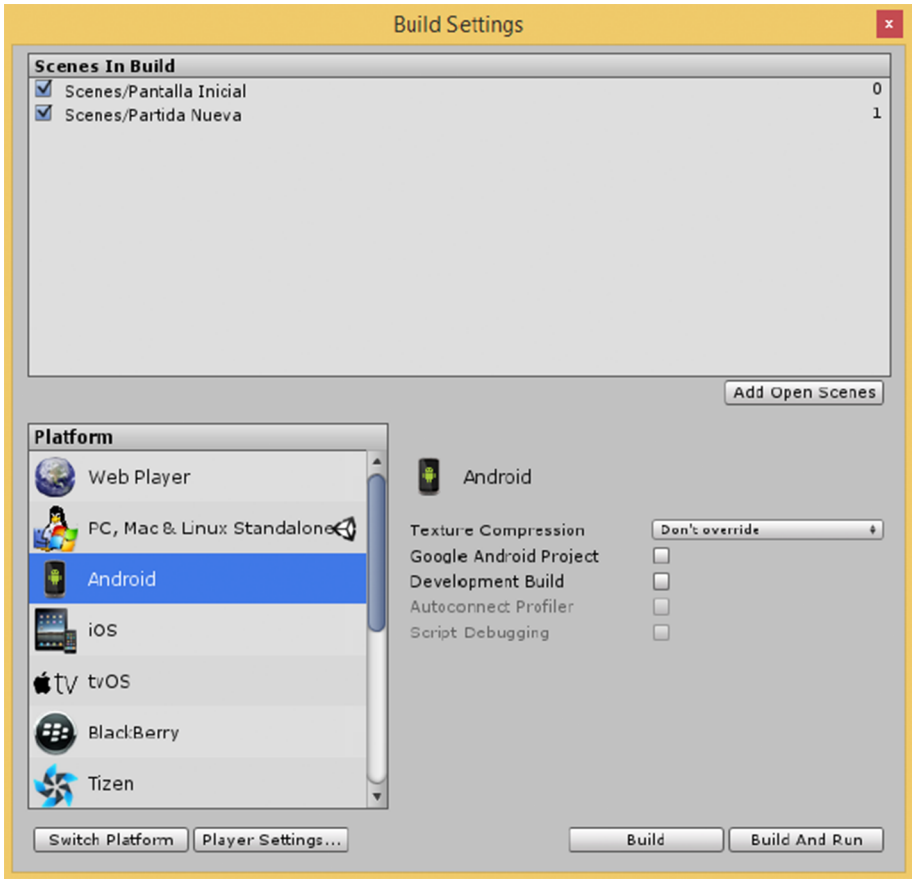
Poner el «AudioSource» en el mismo objeto que el «AudioListener» es una práctica muy común.

4.6. Exportar a una plataforma

Cuando desarrollamos un proyecto dentro del editor no tenemos manera de ejecutarlo sin iniciar el editor. Para poderlo hacer, tenemos que generar un ejecutable para la plataforma que nos interese. Lo que tenemos que hacer en este caso es ir al menú «File > Build Settings» y en la pantalla que nos aparece seleccionar la plataforma con la que queremos trabajar. Esto lo hacemos seleccionando la opción que nos interese en la lista de plataformas que hay en la parte inferior izquierda.

Nota

El proceso de creación del ejecutable en Unity se conoce como *build*.



Una vez seleccionada, hacemos clic en «Switch Platform», si no era ya la opción seleccionada. Hay que tener en cuenta que Unity hace una gran cantidad de optimizaciones internas por cada *asset* según la plataforma. Por este motivo, cada vez que cambiemos de plataforma, Unity lanzará un proceso que puede ser bastante lento. De hecho, cuanto más complejo sea el proyecto, más tardará, por lo que si tenemos claro que solo queremos exportar a una plataforma deberíamos seleccionar esa plataforma cuanto antes. En caso de querer exportar a múltiples plataformas, se aconseja dejar el proyecto lo más pulido y testeado posible para una plataforma y luego realizar este proceso, para minimizar cambios de plataformas, que ya hemos dicho que son procesos lentos.

Reto 12

Una vez comprobado que el proyecto funciona correctamente, cread el ejecutable de manera que se pueda probar desde un navegador.

4.7. Soluciones a los retos propuestos

Reto 1

```
1.    using UnityEngine;
2.    public class GameExit: MonoBehaviour {
3.        public void SalirJuego() {
4.            Debug.Log("Saliendo del juego…");
5.        }
6.    }
```

Reto 2

```
7.    using UnityEngine;
8.    public class GameExit: MonoBehaviour {
9.        public void SalirJuego() {
10.           Application.Quit();
11.       }
12.   }
```

Reto 3

```
11.   using UnityEngine;
12.   public class NuevaPartida: MonoBehaviour {
13.       public void CargarPartidaNueva() {
14.           SceneManager.LoadScene("Partida Nueva");
15.       }
16.   }
```

Nota

Antes de la versión 5 de Unity, esto se podía hacer de la siguiente mañera:

```
Application.LoadLevel("Partida Nueva");
```

Pero desde la versión 5, la API se cambió por:

```
SceneManager.LoadScene ( "Partida Nueva");
```

Reto 4

El aspecto que hace que un objeto se comporte como un botón (y, por ejemplo, pueda ser pulsado) es el hecho de que contenga un *component* de tipo «Button». Por lo tanto, si queremos que un bloque de texto, un objeto de tipo «Text», se comporte de este modo, basta con añadirle dicho componente mediante el botón «Add Component» del Inspector.

Reto 5

Primero, es necesario añadir el fichero de imagen de proyecto como *asset*. En caso contrario, Unity no podría encontrarlo. No busca en el sistema de ficheros, solo entre los activos del proyecto. Para hacerlo, basta con arrastrar el fichero desde el explorador del sistema al apartado Asset de Unity, que lo importará automáticamente. Entonces, hay que arrastrar el nuevo *asset* sobre la propiedad «Source image» del componente «Image».

Reto 6

Para reorganizar los objetos basta con crear dos nuevos *GameObjects* vacíos dentro del «Canvas» (opción de menú «Create Empty») con los nombres indicados. Entonces, hay que arrastrar los elementos ya existentes, manteniendo la relación jerárquica entre ellos, de modo que los dos nuevos objetos sean los únicos nodos que cuelgan directamente del «Canvas».

Reto 7

Un ejemplo de mapa de localizaciones podría ser el siguiente. En este caso, usamos un método estático («CreateNode») para construir los nodos a partir de dos parámetros: el texto de la localización y la lista de opciones, como *array* de *string*. El resto de atributos se asignan directamente, aprovechando que han sido declarados como públicos.

```
using System;
public class StoryFiller {
    public static GameplayManager.StoryNode FillStory() {
        GameplayManager.StoryNode root = CreateNode("Te encuentras en una habitación y no recuerdas nada. Quieres salir.", new string[] {
            "Explorar objetos", "Explorar la habitación"
        });
        GameplayManager.StoryNode nodo1 = CreateNode("Hay una silla y una mesa con una planta a la izquierda. A la derecha hay una estantería con libros. Detrás parece que hay unas cajas", new string[] {
            "Ir a la derecha", "Ir a la izquierda", "Ir hacia atrás", "Explorar la habitación"
        });
        root.nextNode[0] = nodo1;
        GameplayManager.StoryNode nodo2 = CreateNode("Nada interesante.... aunque hay un libro que llama la atención... botánica para astronautas?", new string[] {
            "Explorar el resto de objetos de la habitación", "Averiguar más del libro raro"
        });
        nodo1.nextNode[0] = nodo2;
        nodo2.nextNode[0] = nodo1;
        GameplayManager.StoryNode nodo3 = CreateNode("Parece que habla de plantas, qué sorpresa. Hay una señalada, se llama plantus corrientis", new string[] {
            "Dejar el libro en su sitio y explorar el resto de objetos de la habitación"
        });
        nodo3.nextNode[0] = nodo1;
        nodo2.nextNode[1] = nodo3;
        GameplayManager.StoryNode nodo4 = CreateNode("Nada interesante en las cajas, están llenas de libros... deberían estar en la estantería", new string[] {
            "Volver y explorar el resto de objetos de la habitación"
        });
        nodo1.nextNode[2] = nodo4;
        nodo4.nextNode[0] = nodo1;
        GameplayManager.StoryNode nodo5 = CreateNode("Humm, una silla. Te duele la cabeza, así que te sientas", new string[] {
            "Eso está muy bien, pero yo quiero ver lo de la mesa también"
        });
        nodo1.nextNode[1] = nodo5;
        GameplayManager.StoryNode nodo6 = CreateNode("La mesa en sí no tiene nada de especial, tiene un poco de tierra de la planta. Los cajones de la mesa parecen entreabiertos", new string[] {
            "Explorar los cajones", "Volver a explorar los otros objetos"
        });
        GameplayManager.StoryNode nodo6bis = CreateNode("La mesa en sí no tiene nada de especial, tiene un poco de tierra de la planta. La etiqueta de la planta pone plantus corrientis. Los cajones de la mesa parecen entreabiertos", new string[] {
            "Explorar los cajones", "Mirar la planta de cerca", "Volver a explorar los otros objetos"
        });
        nodo5.nextNode[0] = nodo6;
        nodo3.nodeVisited = () => {
            nodo5.nextNode[0] = nodo6bis;
        };
        nodo6bis.nextNode[2] = nodo6.nextNode[1] = nodo1;
        GameplayManager.StoryNode nodo7 = CreateNode("Los cajones están vacios, mejor explorar otra cosa", new string[] {
            "Volver a explorar los otros objetos"
        });
        nodo6.nextNode[0] = nodo7;
        nodo7.nextNode[0] = nodo1;
        GameplayManager.StoryNode nodo8 = CreateNode("¡¡Al levantar la planta te encuentras una llave!! ¿Qué abrirá?", new string[] {
            "Explorar la habitación"
        });
        nodo6bis.nextNode[1] = nodo8;
        GameplayManager.StoryNode nodo9 = CreateNode("La habitación tiene un par de ventanas y una puerta", new string[] {
            "Ir a la ventana #1", "Ir a la ventana #2", "Ir a la puerta"
        });
        root.nextNode[1] = nodo1.nextNode[0] = nodo8.nextNode[0] = nodo9;
        GameplayManager.StoryNode nodo10 = CreateNode("La ventana está tapiada, no se puede abrir", new string[] {
            "Ir a la otra ventana", "Ir a la puerta"
        });
        nodo9.nextNode[0] = nodo9.nextNode[1] = nodo10;
        nodo10.nextNode[0] = nodo10;
        GameplayManager.StoryNode nodo11 = CreateNode("La puerta está cerrada con un candado", new string[] {
            "Explorar los objetos de la habitación"
        });
        GameplayManager.StoryNode nodo11bis = CreateNode("La puerta está cerrada con un candado", new string[] {
            "Explorar los objetos de la habitación", "Usar la llave"
        });
        nodo11bis.nextNode[0] = nodo11.nextNode[0] = nodo1;
        nodo9.nextNode[2] = nodo10.nextNode[1] = nodo11;
        nodo8.nodeVisited = () => {
            nodo9.nextNode[2] = nodo10.nextNode[1] = nodo11bis;
        };
        GameplayManager.StoryNode nodo12 = CreateNode("¡¡Has salido de la habitación!!", new string[] {
            "Salir del juego"
        });
        nodo11bis.nextNode[1] = nodo12;
        nodo12.isFinal = true;
        return root;
    }
    private static GameplayManager.StoryNode CreateNode(string history, string[] options) {
        GameplayManager.StoryNode node = new GameplayManager.StoryNode();

        node.history = history;
        node.answers = options;
        node.nextNode = new GameplayManager.StoryNode[options.Length];
        return node;
    }
}
```

Reto 8

Un posible código para gestionar el juego es el siguiente. Nótese como se añaden algunos atributos (`histroyText`, `answersParent`, `buttonAnswer`) que, inicializados correctamente desde el editor, permiten acceder a ciertos elementos del proyecto (*prefabs*, elementos de la UI) desde el *script*.

De cara a destruir los botones existentes, la estrategia es la siguiente. Se guarda en un atributo (`buttonAnswer`) con el *GameObject* padre en la jerarquía de botones. Entonces, a través de su componente «Transform», es posible recorrer los componentes «Transform» de todos los botones existentes y manipularlos.

Al crear botones nuevos, estos se añaden a la jerarquía de nuevo asignando `buttonAnswer` como padre a cada uno de sus componentes «Transform».

Nota

Se ha de tener presente que todas las operaciones sobre la Hierarchy siempre son a nivel de componentes «Transform».

```
1. using UnityEngine;
2. using UnityEngine.UI;
3. public class GameplayManager: MonoBehaviour {
4.     public class StoryNode {
5.         public string history;
6.         public string[] answers;
7.         public StoryNode[] nextNode;
8.         public bool isFinal = false;
9.         public delegate void NodeVisited();
10.          public NodeVisited nodeVisited;
11.     }
12.
13.     public Text historyText;
14.     public Transform answersParent;
15.     public GameObject buttonAnswer;
16.     private StoryNode current;
17.     void Start() {
18.         current = StoryFiller.FillStory();
19.         historyText.text = "";
20.         FillUI();
21.     }
22.
23.     void AnswerSelected(int index) {
24.         print(index);
25.         historyText.text += "\n" + current.answers[index];
26.         if (!current.isFinal) {
27.             current = current.nextNode[index];
28.             FillUI();
29.         }
30.         else
31.         {
32.             //Final de partida
33.         }
34.     }
35.
36.     void FillUI() {
37.         historyText.text += "\n" + "\n" + current.history;
38.         foreach(Transform child in answersParent.transform) {
39.             Destroy(child.gameObject);
40.         }
41.         bool isLeft = true;
42.         float height = 50;
43.         int index = 0;
44.         foreach(string answer in current.answers) {
45.
46.             GameObject buttonAnswerCopy = Instantiate(buttonAnswer);
47.             buttonAnswerCopy.transform.parent = answersParent;
48.             float x = buttonAnswerCopy.GetComponent < RectTransform > ().r
ect.x * 1.1 f;
49.
50.             buttonAnswerCopy.GetComponent < RectTransform > ().localPositio
n = new Vector3(isLeft ? x : -x, height, 0);
51.
52.             if (!isLeft) height += buttonAnswerCopy.GetComponent < RectTran
sform > ().rect.y * 2.5 f;
53.
54.             isLeft = !isLeft;
55.             FillListener(buttonAnswerCopy.GetComponent<Button>(), index);
56.             buttonAnswerCopy.GetComponentInChildren<Text>().text = answer;

57.             index++;
58.         }
59.     }
60.
61.     void FillListener(Button button, int index) {
62.         button.onClick.AddListener(() => {
63.             AnswerSelected(index);
64.         });
65.     }
66. }
```

Reto 9

Ver las soluciones a los retos 2 y 3. De hecho, podemos volver a usar el *script* del caso de iniciar partida, no es necesario crear un nuevo. Sí que será necesario para volver a la pantalla de inicio.

Reto 10

Para llevar a cabo esta tarea, hay que controlar constantemente la entrada de usuario para ver si se pulsa la tecla «Esc». Por lo tanto, se debe usar el método «Update» para, en cada *frame* del juego, ver si dicha tecla ha sido pulsada. Esta funcionalidad tiene sentido que la lleve a cabo el «GameplayManager», ya que es el gestor del estado del juego.

```
1.  void Update() {
2.      if (Input.GetKeyUp(KeyCode.Escape))
3.          FinishGameMenu();
4.  }
5.
6.  void FinishGameMenu() {
7.      pauseMenu.SetActive(!pauseMenu.activeSelf);
8.  }
9.
10. public void RestartScene() {
11.     UnityEngine.SceneManagement.SceneManager.LoadScene(UnityEngine.SceneMa
nagement.SceneManager.GetActiveScene().buildIndex);
12. }
13.
14. public void LoadStartScene() {
15.     UnityEngine.SceneManagement.SceneManager.LoadScene("Pantalla Inicial")
;
16. }
```

Reto 11

Podéis elegir la música que más os guste. En este documento se parte del conjunto de sonidos llamado *BitMusic Pack*, de ASSETCREW, que se puede encontrar en la Asset Store. Este paquete es un conjunto de pistas preparadas para integrarlas en nuestro juego. Están diseñadas para que se puedan repetir indefinidamente, lo cual es ideal para música de fondo.

Elegimos una pista de audio por cada *scene*, en este caso «Village_ogg» y «Cemetery_ogg», como pistas de fondo para nuestras pantallas. A continuación, cargamos la *scene* «Pantalla Inicial» y seleccionamos el *GameObject* «Main Camera» y le añadimos un *component* «AudioSource». Una vez añadido, arrastramos el *asset* «Village_ogg» desde Project al campo «AudioClip» del componente. Entonces, seleccionamos las opciones «Loop» y «Play On Awake», como en la figura:

De esta manera, nada más empezar el juego ya estará sonando la música de fondo, y se repetirá automáticamente cuando se acabe.

Si hacemos la misma operación para cada *scene* cambiando solo la pista de audio que asignamos a cada «AudioSource», tendremos ya toda la música de fondo en nuestro proyecto. Dado que cada vez que se carga una escena se destruyen los objetos de la anterior y se crean los de la nueva, esto garantiza que se reproduce una única música en cada pantalla.

Reto 12

La opción para llevar a cabo esta tarea es mediante la plataforma WebGL.

Resumen

En este módulo didáctico hemos introducido varios conceptos importantes. El primero es el de motor de juego, que es básicamente una recopilación de herramientas existentes relacionadas con todo lo necesario para desarrollar un videojuego. Aunque hay proyectos que se desarrollan desde cero sin estos motores, la tendencia de la industria es usarlos para evitar duplicidades y para poder encontrar talento entrenado fácilmente.

Hay una gran variedad de motores existentes que pueden cubrir prácticamente cualquier tipo de necesidad. Nosotros hemos escogido uno de los más populares actualmente, Unity, para mostrar cómo desarrollar un juego sencillo, pero prácticamente todo lo que hemos explicado tiene equivalencias directas en otros motores. Las razones principales por las que hemos escogido este motor son que tiene una comunidad muy activa detrás, admite programación con lenguajes más sencillos, se puede exportar a una gran variedad de plataformas y tiene una versión gratuita con muy pocas limitaciones.

A la hora de conocer lo básico de Unity hemos visto cómo se estructura el editor y un proyecto, y qué función cumple cada parte del editor. También hemos visto cómo encontrar recursos *online* para poderlos usar en nuestros juegos y así suplir la falta de tiempo o habilidades para poder crear todo lo que necesita un juego moderno. Seguidamente hemos entrado en cómo configurar nuestros proyectos. Hemos conocido los conceptos de *component* y *prefab*, que son los elementos básicos en Unity. Con *components* podemos hacer que cualquier objeto se comporte de la manera que nos interese. Los *prefabs* son básicamente objetos ya preconfigurados que podemos replicar en una escena, muy prácticos para acelerar el desarrollo.

También hemos visto cómo programar en Unity. Este motor se basa en *scripts*, y admite varios lenguajes. Nosotros hemos escogido el lenguaje C#, y hemos dado ejemplos sobre cómo capturar los eventos y cómo interactuar con algunas de las funcionalidades de Unity para poder programar comportamientos básicos.

Finalmente, hemos visto todos los pasos necesarios para diseñar un juego desde cero. Si bien es un juego muy sencillo, cubre todo lo necesario para empezar a montar proyectos más complejos, especialmente en la parte de la interfaz de usuario (UI).