Un juego de plataformas

Rodrigo Pizarro Lozano

PID_00236759

Universitat Oberta de Catalunya

Índice

Introducción

En este módulo se introduce el uso de elementos gráficos estáticos y física dentro de Unity con el propósito de desarrollar juegos sencillos, pero con un aspecto y un comportamiento típicos de los videojuegos de acción. Unity incluye diferentes formatos de salida para nuestros proyectos; aquí aprovecharemos para ver cómo llevar a cabo el desarrollo en Android, desde el punto de vista tanto de los mecanismos de entrada como de la generación del ejecutable.

Para ello, este módulo se divide en dos partes, una de carácter más teórico y otra más práctica. El propósito de la parte teórica es simplemente dar a conocer el mínimo imprescindible para empezar a trabajar. Sin embargo, se debe tener en cuenta que no se trata de una explicación autocontenida, ya que existen muchos pequeños detalles. La única manera de profundizar es mediante la práctica y el estudio de la documentación oficial. Para guiar este proceso está la segunda parte del módulo, que es la realmente importante.

La parte teórica se centra en cuatro temas. El primero es una introducción a los *sprites*, que son el elemento fundamental para gestionar gráficos, sobre todo en los juegos 2D. También aprenderemos a usar *sprites* múltiples para optimizar recursos. A continuación, hablaremos sobre la física 2D en Unity, que es una herramienta muy útil para poder controlar la interacción entre los elementos de juego de manera sencilla. Se incluyen ejemplos tanto de las herramientas del editor como en formato *script*. Después nos fijaremos en el entorno móvil, para lo que veremos los tipos más comunes de entrada (*input*) que se pueden encontrar en dispositivos móviles; esto incluye tanto gestos táctiles como leer datos de los sensores del giroscopio y del GPS. Y, finalmente, explicaremos todo lo necesario para poder exportar un proyecto a un dispositivo móvil.

En lo que respecta a la parte práctica, y para trabajar más a fondo los aspectos explicados, desarrollaremos un proyecto 2D sencillo pero completo, basado en un popular juego clásico.

Objetivos

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

1. Entender las herramientas básicas para trabajar con *assets* 2D del tipo *sprite*.
2. Poder utilizar la física tanto para simular comportamiento realista como para poder programar comportamiento en base a eventos de física.
3. Ser capaz de utilizar las entradas y los sensores más comunes.
4. Poder exportar un proyecto a una plataforma móvil.
5. Ser capaz de desarrollar un pequeño juego de plataformas 2D usando tanto el editor como el sistema de *scripting*.

1. *Sprites*

Siempre que se desee incluir una imagen en un juego en Unity, es necesario añadir un *sprite* a nuestros *assets*.

Un ***sprite*** es, básicamente, una imagen 2D que podemos usar en un juego.

De manera simple, cada imagen se representaría con un *sprite*. Sin embargo, una optimización muy común es usar un mismo archivo que contenga varios *sprites*. Esto es factible ya es que resulta muy sencillo y eficiente decirle a nuestro hardware que pinte solo una porción concreta u otra de la imagen.

Nota

Un fichero de imagen que combina diversos *sprites* también se conoce como *sprite atlas*.

Si no se nos da bien la creación de gráficos, o simplemente no queremos dedicarle tiempo, tanto en webs especializadas como en la Asset Store de Unity podemos encontrar multitud de recursos para explorar y utilizar en nuestro proyecto. De cara este módulo, vamos a elegir el paquete de *assets 2D Platformer* de la Asset Store de Unity, que es gratuito, como referencia de cara a trabajar con *sprites* correctamente. Para trabajar con este paquete, antes será necesario importarlo a un proyecto.

Web recomendada

Un ejemplo de web con recursos gráficos libres es:
www.opengameart.org.

1.1. El editor de *sprites*

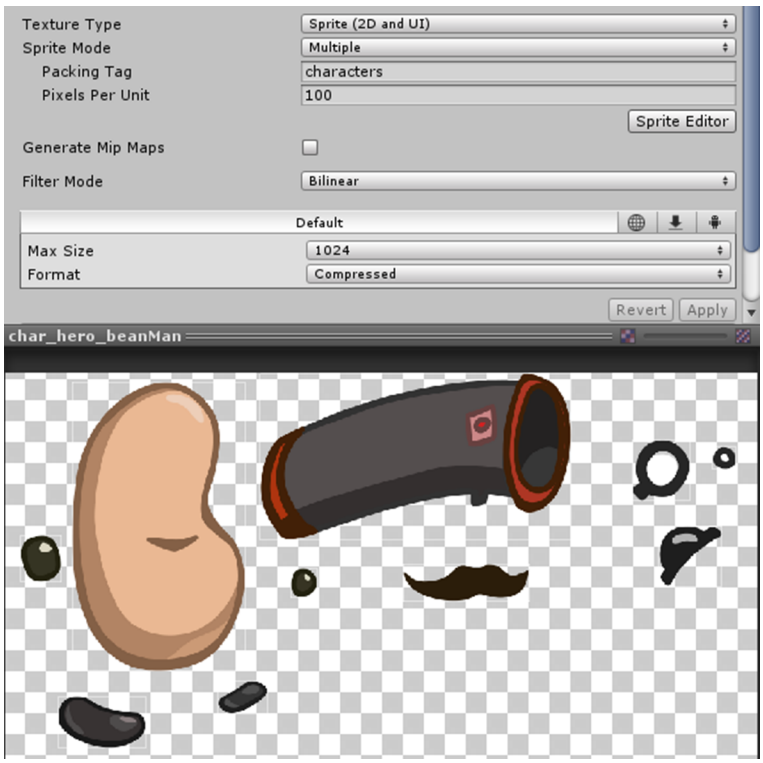
Esta es la herramienta de Unity que permite editar las propiedades de nuestros *sprites*. Para el caso de nuestro proyecto de referencia, el editor permite ver claramente que un *sprite* es realmente un atlas que contiene múltiples imágenes.

En el proyecto de referencia, si seleccionamos «Assets > Sprites > _Character > char_hero_beanMan» podemos visualizar la imagen «beanMan». En este caso, se trata de una imagen que contiene los gráficos que representan todas las partes del cuerpo de un personaje.

Nota

La imagen «beanMan» se corresponde al protagonista de este juego.

En el Inspector podemos ver el modo en que está importado, dentro de la opción «Texture Type». El modo que nos interesa es «Sprite (2D and UI)». Dado que esta imagen es en realidad un atlas, debemos asegurarnos de que el apartado «Sprite Mode» tiene puesto el valor «Multiple».



Nota

El fondo de cuadros blancos y grises indica transparencia.

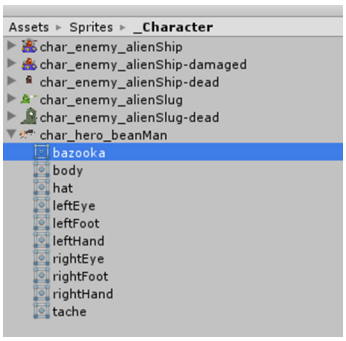
Unity intenta aproximar automáticamente la división del atlas en *sprites* separados, pero no siempre nos dará el resultado que estamos buscando. Para editar manualmente la imagen, pulsaremos el botón «Sprite Editor», de modo que aparecerá una nueva pestaña, el editor de *sprites*.

Dentro de este editor podemos realizar varias acciones. Con el ratón podemos seleccionar las áreas ya creadas automáticamente por Unity haciendo clic en los recuadros que se muestran. Esta acción causa que los bordes del recuadro se muestren de color azul y que lo podamos mover arrastrando el ratón. También podemos redimensionarlo estirando las esquinas, o cambiar el pivote arrastrando el círculo azul. Normalmente, el pivote está centrado en el *sprite*, pero es posible que nos interese cambiarlo. Por ejemplo, el pivote de unos brazos estaría en el hombro, ya que es desde donde en realidad rota.

Nota

El pivote es el punto de apoyo desde el cual un objeto rota.

Una vez editado un *sprite* múltiple podemos ver que en el apartado Project nos aparece un desplegable con todos los *sprites* individuales que hemos puesto en su interior. En nuestro caso «char_hero_beanMan».

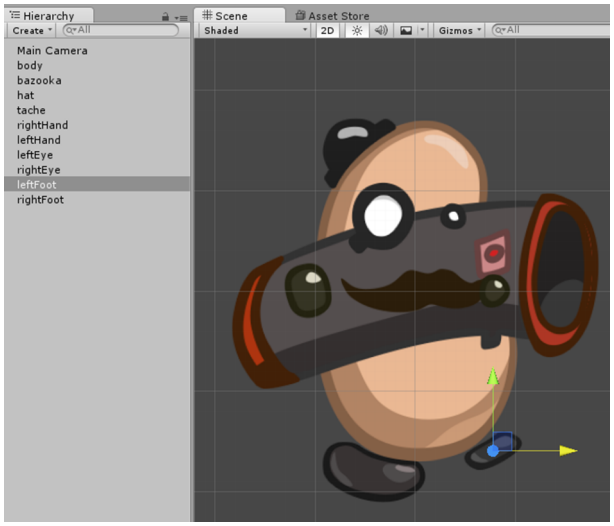


Despliegue de los *subsprites* en el atlas.

Si seleccionamos con el ratón cada *sprite* individual, lo podemos previsualizar en el Inspector.

Con todos los recursos ya preparados, ya podemos empezar a trabajar con ellos. En este caso, para combinarlos y crear un único personaje complejo, el BeanMan. Para ello, vamos arrastrando cada *sprite* individual a la escena, ya

sea sobre el apartado Hierarchy o directamente sobre el apartado Scene del editor. Una vez añadidos todos los elementos y organizados visualmente, el resultado que tenemos puede ser algo parecido a la imagen siguiente:



Nota

Evidentemente, podéis crear un resultado a vuestro gusto.

Durante este proceso, un problema que se hace evidente es que hay elementos que tapan a otros, con un orden de visualización que no es el que estamos buscando.

1.2. *Sorting Layers*

Para solucionar este problema, una opción es usar la distancia a la cámara. A pesar de trabajar en modo 2D, Unity internamente procesa toda la información en 3D. De esta manera, si movemos elementos en el eje *Z*, podremos decidir qué objetos tapan a otros acercándolos o alejándolos de la cámara.

Nota

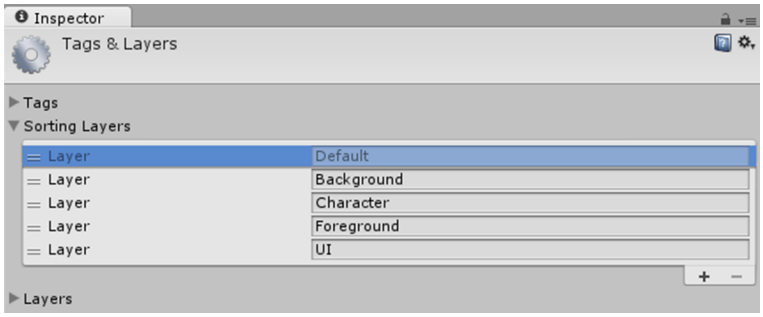
Conceptualmente, las capas en el eje *Z* serían como los decorados en un teatro.

Otra opción más interesante sería definir directamente el orden de visualización de los elementos gráficos.

Los ***Sorting Layers*** ('capas de ordenación') indican a Unity el orden de renderización de los *sprites* de una escena.

La idea principal es que los distintos *sprites* en una escena se organicen en grupos homogéneos de elementos similares. Por ejemplo, muchos juegos tienen un fondo donde hay un paisaje, incluso compuesto por diversos elementos, que lógicamente siempre estará por detrás de los elementos principales, como los personajes o los enemigos de nuestro juego. Conceptualmente, todos estos elementos decorativos conforman un grupo. Para ello, los podemos asignar a un *Sorting Layer* específico, y el resto de elementos de juego a otra. De esta manera, nos aseguramos de que podemos controlar directamente qué elementos tapan a qué otros.

Para definir qué *Sorting Layers* se superponen a otros o crear nuevas capas, podemos ir al menú «Edit > Project Settings > Tags and Layers». Al hacer clic en esa opción, en el Inspector nos aparecen tres desplegables: «Tags», «Sorting Layers» y «Layers».



En el desplegable «Sorting Layers» podemos ver listados todas las capas de nuestro proyecto. En este caso, tenemos las capas por defecto creadas por Unity. El orden es muy importante, ya que debemos saber que los *Sorting Layers* se pintan en orden de arriba abajo. A medida que esto sucede, unos elementos pueden tapar a otros. Por ejemplo, en la configuración por defecto mostrada, los elementos que estén en la capa «Default» podrían quedar tapados por los elementos de «Background», que a su vez podrían ser tapados por elementos de «Character». Y así sucesivamente. Podemos reordenar estas capas simplemente arrastrándolas arriba y abajo con el ratón. Para añadir o eliminar nuestras propias capas podemos usar los botones con los símbolos «+» y «-», abajo a la derecha.

La asignación de la capa a la que pertenece un elemento gráfico se lleva a cabo a través del Inspector, modificando de la propiedad «Sorting Layer» en su componente «SpriteRenderer».

Nota

El componente «SpriteRenderer» de un objeto gestiona su representación gráfica.

También es posible establecer el orden dentro de una misma capa. Esto nos puede servir, por ejemplo, si tenemos un fondo donde hay un paisaje y unos edificios. En ese caso, muy probablemente querremos que los edificios tapen al paisaje. Sin embargo, seguramente tiene sentido que ambos elementos pertenezcan a la capa «Background», ya que realmente los dos tipos de elementos son parte de la decoración. Para ello, se puede usar la propiedad «Order in Layer» del componente «SpriteRenderer». Si varios elementos dentro de un mismo *Sorting Layer* tienen el mismo valor en «Order in Layer», se decide aleatoriamente qué elementos se superponen a otros.

En este caso, el uso de la propiedad «Order in Layer» es lo que tiene más sentido de cara a ordenar los distintos *sprites* que conforman el personaje BeanMan. Podemos usar el valor -1 con el *sprite* Bazooka, y el valor 1 para los ojos, las manos y el bigote.

El resultado final.

2. Física 2D

Gestionar la física (velocidad de objetos, gravedad, colisiones, etc.) de nuestro juego es una de las herramientas más útiles que nos puede aportar Unity. En sus orígenes, Unity era un motor exclusivamente 3D, y no fue hasta la versión 4.3 que incorporó un motor de física pensado para 2D. En esta sección describiremos los componentes básicos para trabajar con física 2D en Unity, aunque la mayoría de conceptos también son válidos para la física en 3D.

Nota

Unity usa internamente el motor de física llamado Box2D, un conjunto de herramientas de código libre.

Lo primero que veremos es cómo detectar colisiones entre objetos. Para ello, necesitaremos dos componentes distintos: un «Collider 2D» y un «Rigidbody 2D».

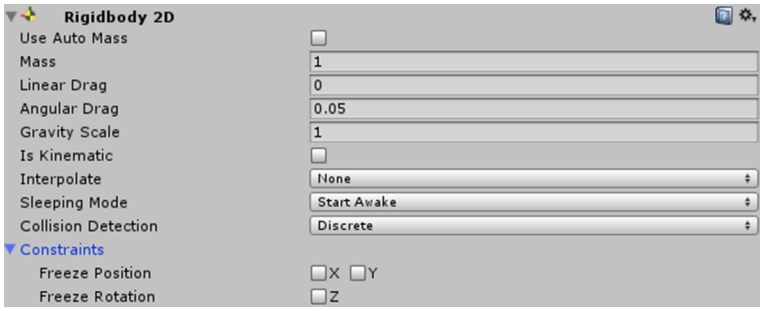
2.1. El componente «Rigidbody 2D»

Si bien Unity dispone del motor de física, este no se aplica por defecto a todos los elementos en una escena.

> Un «Rigidbody 2D» es un componente que le indica a Unity que debe aplicar física sobre el objeto que lo contiene

Por lo tanto, todos los objetos que queramos que tengan un comportamiento realista desde el punto de vista de la física deberán contener este componente. Los que no lo contengan ignorarán todos los aspectos relacionados con la física. Por ejemplo, no se verían afectados por la fuerza de gravedad.

Este componente tiene varios parámetros, pero la mayoría de valores por defecto servirán casi siempre.



Los valores que quizás sí modificaremos en algunas ocasiones son:

- «Gravity Scale»: Permite modificar la aceleración de la gravedad que afecta al *GameObject*. Esto nos puede ser útil para emular objetos que caen con

un paracaídas, por ejemplo, o bien para objetos que queramos que estén suspendidos en el aire (para lo que usaríamos «Gravity Scale = 0»).

- «Constraints»: Sirven para fijar el *GameObject*, ya sea en rotación o en posición. Por ejemplo, podemos simular que un *GameObject* está clavado a una pared y que, por lo tanto, solo puede rotar si lo empujamos (aquí activaríamos «Freeze Position X» y «Freeze Position Y»), o bien podemos simular una plataforma dentro de un tubo que solo se puede desplazar en vertical (aquí activaríamos «Freeze Position X» y «Freeze Rotation Z»).

- «Is Kinematic»: Es necesario activarlo para controlar el movimiento del *GameObject* por *script* o con una animación. Esto es muy típico de los juegos de plataformas, donde muchas plataformas tienen movimientos predefinidos y son el resto de objetos los que calculan su física en base a su movimiento.

2.2. El componente «Collider 2D»

En realidad, un «Collider 2D» es un concepto genérico del que existen varios tipos.

Un «Collider 2D» define un área que el motor de física 2D de Unity controla para detectar colisiones. Tenemos que asegurarnos de que el «Collider» que escogemos es 2D, porque si bien podemos tener un proyecto que use física 2D y 3D a la vez, ambas funcionan con motores independientes y sin interacción entre sí.

«Box Collider 2D»

Es un tipo de «Collider 2D» para formas rectangulares.

La razón de que existan varios tipos distintos es la optimización. Los cálculos físicos pueden ser costosos en recursos, por lo que hay que hacer lo posible para evitar cálculos innecesarios. Existen fórmulas matemáticas para cierto tipo de formas que facilitan enormemente el proceso de cálculo, como por ejemplo los círculos o los rectángulos. De esta manera, siempre que queramos trabajar con objetos que tengan formas parecidas a las de los «Colliders 2D» disponibles, los usaremos.

En caso de tener un objeto muy específico que no se pueda adaptar a los «Colliders 2D» sencillos, existen algunas alternativas. Por ejemplo, podemos usar el «Polygon Collider 2D», que permite editar la forma a nuestro gusto. Al añadirlo a un *GameObject*, en el Inspector podemos ver un botón «Edit Collider» que permite añadir vértices en el área de colisión a través del apartado Scene, tal como muestra la figura:

Un parámetro que tienen todos los «Colliders 2D» es «Is Trigger». Cuando este parámetro está activado, le indica al motor de física que no queremos que se encargue de reaccionar de forma realista a la colisión, sino que simplemente queremos detectar mediante un evento especial que esta ha sucedido, de modo que nosotros la gestionaremos mediante un *script*. Por ejemplo, puede ser muy útil poner un *GameObject* vacío en un sitio específico de nuestra escena con un «Collider 2D» para detectar que nuestro personaje ha llegado allí, como puede ser el final de una pantalla de cara a poder cargar otra pantalla nueva.

Nota

Dos objetos que chocan sin «Is Trigger» activado pueden salir despedidos entre sí, por ejemplo.

Así pues, Unity diferencia entre un componente que gestiona el movimiento de objetos según las leyes de la física («RigidBody 2D») y otro que gestiona las colisiones («Collider 2D»). Sin embargo, hay que aclarar que cuando tenemos dos *GameObjects* y queremos controlar las colisiones entre ellos, es imprescindible que al menos uno tenga un «RigidBody 2D». Si además queremos que reaccionen de manera realista, tenemos que asegurarnos de que ninguno de los dos «Colliders 2D» tenga marcada la opción «Is Trigger» o, sino, solo recibiremos un aviso de que se ha producido la colisión.

También es importante tener claro que a pesar de que podemos mover los *GameObjects* en el eje *Z* (recordemos que, en el fondo, Unity trabaja en 3D internamente), los cálculos de la física 2D ignoran esta dimensión. En este sentido, hay que imaginar que es como si todos los *GameObjects* si estuvieran situados en $Z = 0$.

2.3. *Scripting* con física 2D

El *scripting* de física sigue la misma filosofía que la generación de *frames*, está basado en eventos que recibimos en un objeto MonoBehaviour y procesamos redefiniendo el método adecuado.

En este caso, podemos gestionar dos tipos de eventos en caso de que se produzca una colisión entre objetos:

- `OnCollisionEnter2D(Collision2D coll)`, en caso de que se produzca una colisión entre objetos y ninguno de ellos tenga marcada la propiedad «is Trigger».

- `OnTriggerEnter2D(Collider2D other)`, en caso de que un objeto marcado como «is Trigger» reciba una colisión.

Nota

El parámetro «Collision2D» proporciona información adicional sobre la colisión.

A continuación tenemos un ejemplo de *script* que podemos añadir a un *GameObject* que tenga un «Collider 2D», de modo que lance un mensaje de depuración al recibir una colisión:

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


3.    public class ExampleClass: MonoBehaviour {
4.        void OnCollisionEnter2D(Collision2D coll) {
5.            Debug.Log("Colisión");
6.        }
7.    }
```

Podemos probar de crear dos *GameObjects* en una escena vacía, añadirles un «BoxCollider 2D» a cada uno, y añadirle un «RigidBody 2D» junto este *script* a uno de ellos. Luego apretamos «Play» y con el ratón podemos hacer que colisionen moviendo los objetos en la escena. Comprobaremos que en la pestaña Console aparecerá el mensaje «Colisión».

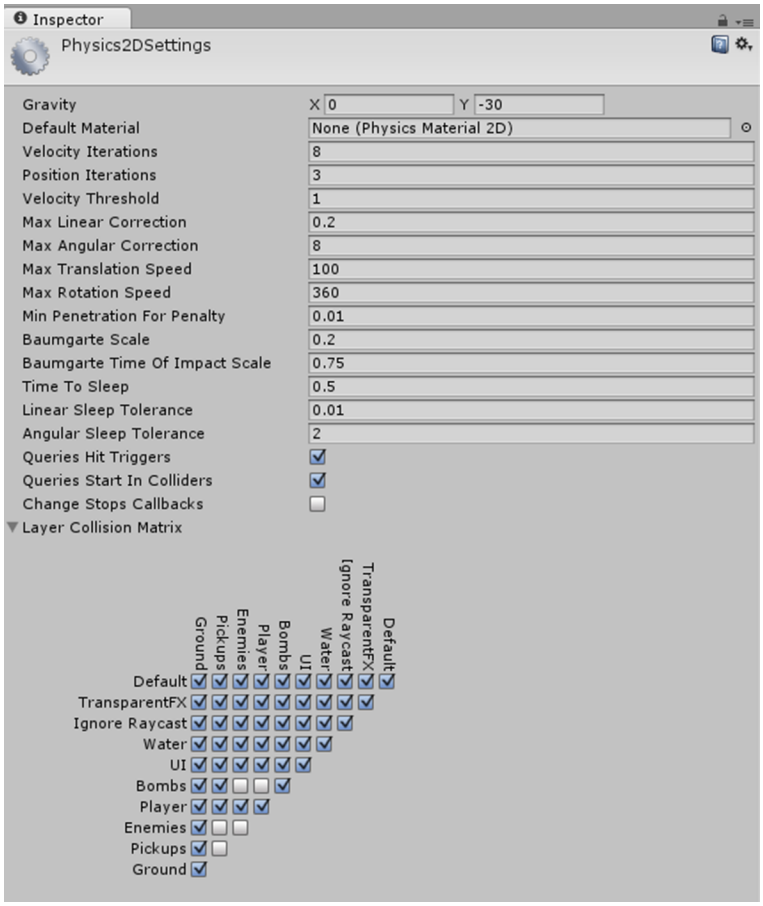
En caso de querer probar el efecto de «isTrigger», podemos activar dicho parámetro en alguno de los «Colliders 2D». En ese caso, podemos modificar el *script* anterior de la siguiente manera:

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


3.    public class ExampleClass: MonoBehaviour {
4.
5.        void OnCollisionEnter2D(Collision2D coll) {
6.            Debug.Log("Colisión con fisica");
7.        }
8.
9.        void OnTriggerEnter2D(Collider2D other) {
10.           Debug.Log("Colisión en trigger");
11.       }
12.   }
```

2.4. Ajustes de la física

Hay proyectos que pueden requerir opciones un poco más específicas, o puede que detectemos que la física está ralentizando nuestro proyecto y queramos intentar optimizarla. En estos casos, podemos ir al menú «Edit > Project Settings > Physics 2D». Al hacer clic nos saldrán las siguientes opciones ajustables:



Como en la mayoría de casos, los valores por defecto nos servirán para casi cualquier proyecto. Sin embargo, hay opciones que probablemente queramos ajustar. Por ejemplo, cambiar el parámetro «Gravity» nos permite simular juegos en el espacio con muy baja gravedad, o simular el balanceo de un barco.

En la parte inferior se encuentra la matriz de capas de colisión, que indica las relaciones entre grupos de objetos que pueden colisionar con otros. Por ejemplo, si tenemos un juego de plataformas, no querremos que el fondo colisione ni con las plataformas ni con los personajes. Con esta matriz podemos seleccionar qué capas colisionan con otras. De esta manera, podemos evitar que el motor de física haga cálculos innecesarios y, por lo tanto, conseguiremos aligerar nuestra aplicación.

Nota

La matriz de colisión se basa en los tipos de *Layer*, no de *Sorting Layer*.

3. *Build* a Android

Cuando queramos ejecutar un proyecto en una plataforma determinada, debemos hacer un *build* seleccionando el tipo que corresponda. En esta sección veremos los detalles de cómo exportar a la popular plataforma móvil Android.

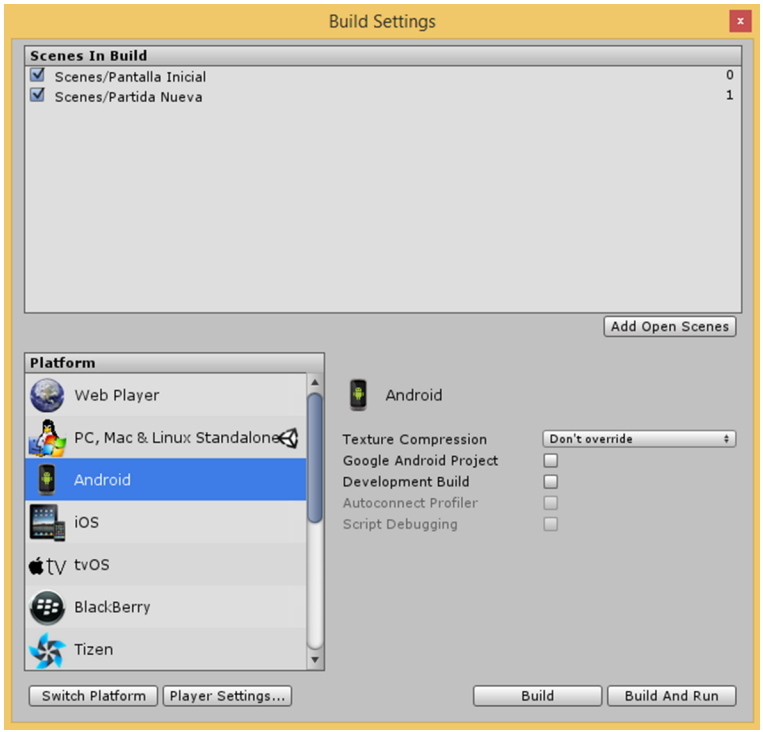
3.1. Exportando a Android

Antes de empezar, tenemos que comprobar que nuestro ordenador tiene el *software development kit* (SDK) de Android y los controladores del móvil instalados, de modo que nuestro sistema lo reconozca al ser conectado por cable USB.

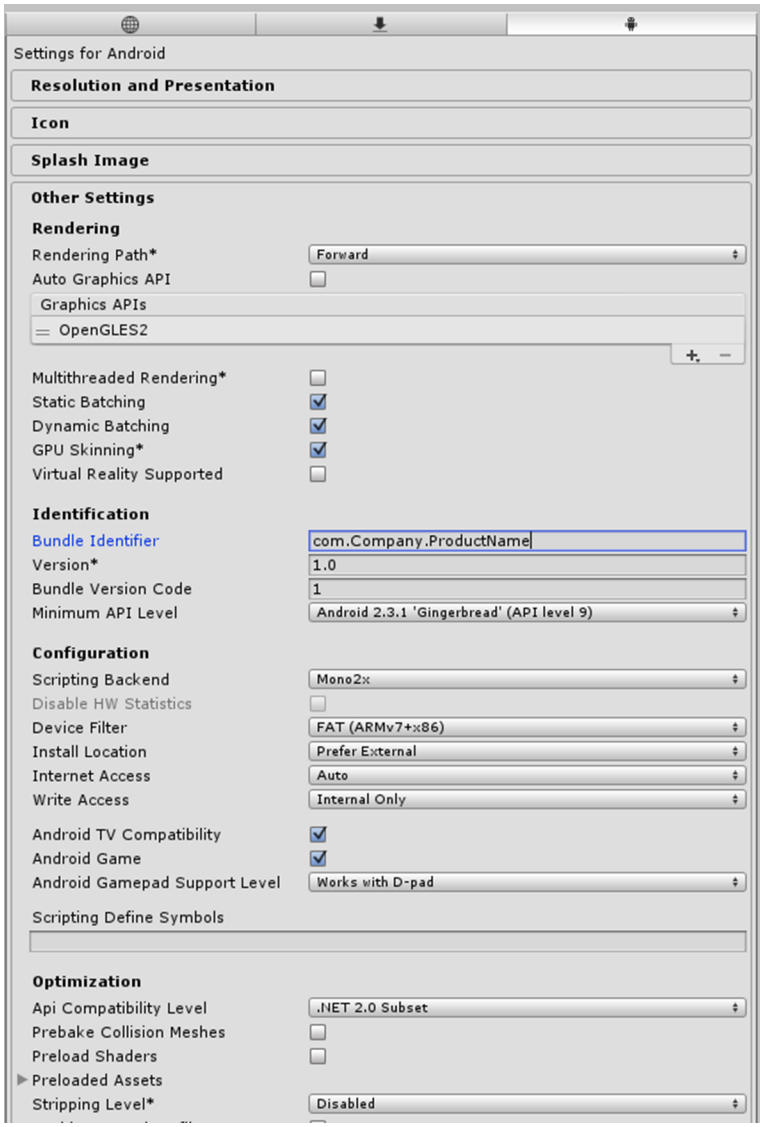
Nota

Si no sabemos qué controladores instalar, podemos hacer una búsqueda en Internet con el nombre del modelo de nuestro móvil.

Una vez que lo hemos comprobado, ya podemos empezar. Tenemos que ir a la pantalla de «Build Settings» («File > Build Settings») y seleccionar «Android»:



Al pulsar el botón de «Switch Platform», Unity empezará a optimizar todos los *assets* para Android, comprimiéndolos específicamente para esa plataforma. Una vez acabe, Unity ya habrá dejado todo casi a punto para exportar el proyecto con el botón «Build And Run» y conectando un móvil Android. Antes, sin embargo, deberemos ajustar algunos parámetros del menú «Player Settings», accesible mediante dicho botón. Este menú tiene el siguiente aspecto:



El parámetro que necesariamente debemos cambiar es el «Bundle Identifier». Este valor es un identificador que le indica a Unity cuál es la aplicación asociada. El identificador tiene que ser único y no puede coincidir con los ya existentes en el terminal. La convención que se suele utilizar es la dirección web de un producto, escrita al revés. Por ejemplo, es común encontrar empresas que lanzan webs con dominios del estilo: «www.nombredejuego.empresa.com». Esta es una práctica común debido a cómo funcionan los subdominios. Si tuviéramos ese dominio, en el «Bundle Identifier» pondríamos «com.empresa.nombredejuego» como valor.

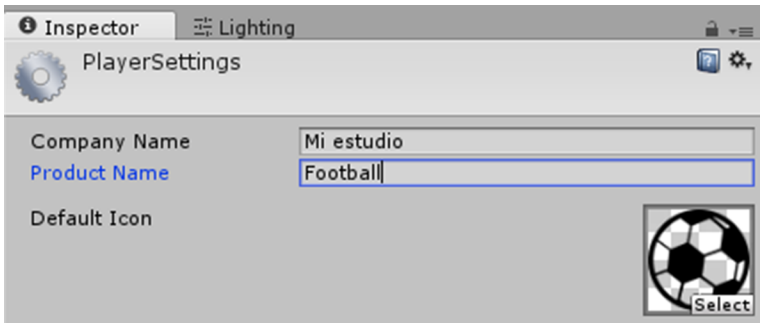
Otro parámetro que probablemente queramos cambiar es el «Minimum API Level». Los terminales con versiones más antiguas de Android suelen ser también teléfonos con muy pocos recursos. Al ser hardwares menos potentes, corremos el riesgo de que nuestro juego no se vea fluido y de que recibamos malas críticas, con lo que puede ser una estrategia eficaz poner un mínimo un poco más elevado. A cambio, claro está, perderemos potenciales usuarios. Unity publica regularmente en su web estadísticas sobre el uso de las versiones del sistema operativo, entre otras valiosas informaciones, con lo que podemos

Web recomendada

Las estadísticas se encuentran en:
http://hwstats.unity3d.com/mobile.

hacernos una idea de cuantos usuarios perderíamos. En el momento de redacción de este documento, por ejemplo, los usuarios de versiones de Android inferiores a la 4.0 suman un 1,1%.

Otro de los detalles relevantes que podemos cambiar en este menú es el icono de nuestro juego. Si no lo hacemos, el icono que se mostrará por defecto es el logo de Unity, lo cual da la sensación de que se trata de un producto en el que no se han cuidado los detalles. Esto lo podemos corregir en la opción «Default Icon» en la parte superior del Inspector mientras los «Player Settings» están abiertos.



En este caso, hemos seleccionado una textura importada a nuestro proyecto anteriormente. También hemos puesto el nombre de nuestro producto en la sección «Product Name», y el nombre de nuestro estudio o nuestra empresa en el campo «Company Name».

Como vemos, hay una gran cantidad de opciones extra para hacer *build*, de un nivel mucho más avanzado.

3.2. Haciendo *debug* en Android

Una de las limitaciones más habituales cuando exportamos a una plataforma es el hecho de que controlar que se ejecuta todo correctamente, ya que solucionar errores es complicado. Afortunadamente, Unity tiene unas cuantas herramientas para ponérnoslo fácil. Solo vamos a ver la más básica, que es poder leer los mensajes.

Para ello, necesitamos abrir una ventana de consola. Una vez la consola abierta, debemos ubicarnos en la ruta donde tenemos el ejecutable «adb» instalado. Este programa viene incluido en el SDK de Android y está dentro de la carpeta «platform-tools». Por ejemplo, una ruta típica en sistemas Windows es la siguiente: «C:\Program Files (x86)\Android\android-sdk\platform-tools».

Nota

En Windows podemos obtener una consola pulsando «Windows + R» y escribiendo «cmd».

Si en la consola escribimos «adb logcat -s Unity», con el dispositivo conectado, podremos ver todos los mensajes que se generan con el método «Debug.Log», así como todos los mensajes que en el editor de Unity se podrían leer en la consola.

4. Entrada para móvil

En esta sección haremos una introducción de los tipos más básicos de interacción que podemos obtener en un dispositivo móvil y veremos cómo distintos tipos de juegos las han utilizado de diversas maneras.

Los móviles y las tabletas tienen habitualmente distintos elementos con los que podemos interactuar para obtener tanto datos del entorno como acciones de un usuario. El elemento más básico es la pantalla táctil, con la que un usuario puede introducir texto o realizar gestos que podemos usar para desencadenar acciones. Además, existen otro tipo de sensores como los giroscopios, que se usan por ejemplo en juegos de carreras para simular un volante, o el GPS, que se usa en juegos de realidad aumentada a nivel de ciudad. Algunos móviles integran más sensores, o también hay fabricantes que han creado hardware con funcionalidades muy diversas, que se acopla a dispositivos móviles, pero Unity no tiene funciones nativas para acceder a ellos. Tendríamos que usar *plugins*.

En esta sección vamos a trabajar con la entrada de usuario en la pantalla táctil y las de los sensores de giroscopio y GPS. Dado que cada elemento de entrada tiene muchos pequeños detalles, no centraremos en proporcionar unos ejemplos elementales con los que conocer las bases y empezar a trabajar.

4.1. Interacción táctil básica

Empecemos por cómo reconocer interacciones con la pantalla táctil. Veamos el siguiente *script*:

```
1. using  UnityEngine;
2. using  System.Collections;
3. public  class  TouchTest:  MonoBehaviour  {
4.       void  Update()  {
5.         Debug.Log("tocado la pantalla con " + Input.touchCount +
" dedos ");
6.   }
7. }
```

«Input»

Con esta clase Unity controla la entrada de usuario.

Este *script* escribe en el *log* de depuración el número de dedos que tocan la pantalla en cada *frame*, si hay alguno. Sin embargo, en el editor no podremos comprobarlo porque las pulsaciones del ratón no se interpretan como *touch*. Al mismo tiempo, si instalamos esta aplicación en un móvil, solo podremos ver si funciona haciendo *debug*, lo cual no es precisamente cómodo.

Una manera de comprobar que este *script* funciona es generando una UI muy sencilla. Si vamos a la jerarquía del proyecto y hacemos «Create > UI > Text», se nos creará un elemento que podremos usar para mostrar la información por pantalla. Podemos ajustar la posición del texto para que se muestre en un lugar concreto de la pantalla mediante los valores de «PosX» y «PosY» de su «RectTransform».

Nota

Los elementos de UI poseen un componente «RecTransform» en vez de «Transform».

Ahora podemos modificar el *script* de la siguiente manera para usar la UI que acabamos de crear:

```
1.   using UnityEngine;
2.   using System.Collections;
3.   using UnityEngine.UI;
4.
5.   public class TouchTest: MonoBehaviour {
6.      public Text text;
7.      void Update() {
8.         text.text = "tocado la pantalla con " + Input.touchCount
+ " dedos ";
9.
10.     }
11.  }
```

De este modo, asignando el *GameObject* que contiene el componente «Text» desde el editor, ya podríamos probar este *script*.

4.2. Gestos táctiles

Ahora vamos a ver cómo reconocer gestos del usuario. En concreto, *swipe* y *pinch*, que son, respectivamente, el gesto de mover un dedo hacia un lado (vertical u horizontalmente) y el de tocar la pantalla con dos dedos y acercarlos (un «pellizco»).

Swipe

Este es un gesto con el que una gran cantidad de usuarios se sienten cómodos y que utilizan una gran cantidad de aplicaciones como, por ejemplo, para pasar las fotos en un álbum.

Antes de ver el *script*, vamos a introducir el concepto de «event Action» en C#. Esta sintaxis es realmente una manera más corta de declarar un *delegate* que tiene un parámetro de un cierto tipo y no retorna nada. De esta manera, si declaramos un «Action» de un tipo y alguien se subscribe, actuará como un «event». Con este *script* podemos detectar un *swipe* en cualquier dirección:

```
1. using System.Collections;
2. using System;
3.
4. public class MiniGestureRecognizer: MonoBehaviour {
5.     public enum SwipeDirection {
6.       Up, Down, Right, Left
7.     }
8.    public static event Action <SwipeDirection> SwipeAction;
9.    private bool swiping = false;
10.    private bool eventSent = false;
11.    private Vector2 lastPosition;
12.
13.    void Update() {
14.      if (Input.touchCount == 0) return;
15.      if (Input.GetTouch(0).deltaPosition.sqrMagnitude != 0){
16.        if (!swiping) {
17.          swiping = true;
18.          lastPosition = Input.GetTouch(0).position;
19.          return;
20.        } else {
21.          if (!eventSent) {
22.            if (SwipeAction != null) {
23.              Vector2 direction = Input.GetTouch(0).position -
 lastPosition;
24.              if(Mathf.Abs(direction.x)>Mathf.Abs(direction.y)){
25.                if (direction.x > 0)
26.                  SwipeAction(SwipeDirection.Right);
27.                else
28.                  SwipeAction(SwipeDirection.Left);
29.              } else {
30.                if (direction.y > 0)
31.                  SwipeAction(SwipeDirection.Up);
32.                else
33.                  SwipeAction(SwipeDirection.Down);
34.              }
35.              eventSent = true;
36.            }
37.          }
38.        }
39.      } else {
40.        swiping = false;
41.        eventSent = false;
42.      }
43.    }
44.    }
```

«deltaPosition»

Este campo indica los cambios de posición en la entrada desde el último *frame*.

El código primero mira `Input.touchCount` para comprobar que hay dedos en la pantalla, y si no los hay vuelve al estado inicial. En cambio, si detecta dedos en la pantalla y hemos movido el dedo (comprobado con `deltaPosition.sqrMagnitude`), hace el grueso de la tarea del método. Esta tarea no es más que una sencilla máquina de estados, donde comprobamos que no hayamos enviado ya el evento, y si no lo hemos enviado hacemos el cálculo. Este cálculo es simplemente detectar en qué dirección se está moviendo el dedo, calculando el vector y el mayor de los componentes. Un detalle es que funcionará aunque tengamos más de un dedo moviéndose por la pantalla, pero solo funcionará con el primero que hayamos apretado, por las llamadas a `Input.GetTouch(0)`.

Por otro lado, puede que este código sea demasiado sensible y queramos más recorrido antes de lanzar el evento de «SwipeAction». En ese caso, podemos aumentar el valor de la condición `if (Input.GetTouch(0).deltaPosition.sqrMagnitude != 0)`.

Ahora ya tenemos un *script* que detecta los gestos de *swipe*. Podemos probarlo y usarlo en otro *script*:

```
using UnityEngine;
using System.Collections;
public class SwipeTest: MonoBehaviour {
    void Start() {
        MiniGestureRecognizer.Swipe += Swipe;
    }
    void Swipe(MiniGestureRecognizer.SwipeDirection direction) {
        switch (direction) {
            case MiniGestureRecognizer.SwipeDirection.Left:
                Debug.Log("Swipe izquierda");
                break;
            case MiniGestureRecognizer.SwipeDirection.Right:
                Debug.Log("Swipe derecha");
                break;
            case MiniGestureRecognizer.SwipeDirection.Up:
                Debug.Log("Swipe arriba");
                break;
            case MiniGestureRecognizer.SwipeDirection.Down:
                Debug.Log("Swipe abajo");
                break;
        }
    }
}
```

Añadiendo estos dos *scripts* a cualquier *GameObject* en una escena podremos probar a hacer el gesto de *swipe*. Si notásemos que es demasiado sensible, podemos modificar esta línea:

```
if (Input.GetTouch(0).deltaPosition.sqrMagnitude != 0){
```

y aumentar el mínimo. Si no queremos usar el *debug*, podemos hacer un sistema de UI similar al que hemos hecho anteriormente.

Pinch

Podemos hacer un sistema similar al anterior, donde tenemos un *script* que reconoce el gesto y otro que ejecuta una función con él. En este caso, sería el siguiente:

«getTouch»

Proporciona las interacciones ordenadas por índice.

```
using System;
using UnityEngine;
public class PinchRecognizer: MonoBehaviour {
  public static event Action < float > PinchAction;

  void Update() {
    if (Input.touchCount == 2) { // Necesitamos dos toques.
       Touch touchZero = Input.GetTouch(0);
       Touch touchOne = Input.GetTouch(1);

       Vector2 touchZeroPrevPos = touchZero.position -
 touchZero.deltaPosition;
       Vector2 touchOnePrevPos = touchOne.position -
 touchOne.deltaPosition;

       //Calculamos el cambio entre frames.
       float prevTouchDeltaMag = (touchZeroPrevPos -
 touchOnePrevPos).magnitude;
       float touchDeltaMag = (touchZero.position -
 touchOne.position).magnitude;
       float deltaMagnitudeDiff = prevTouchDeltaMag - touchDeltaMag;
       if(PinchAction!=null)
          PinchAction (deltaMagnitudeDiff);
     }
    }
   }
```

Este *script*, como vemos, usa la misma estrategia que el de *swipe*, calculando los vectores de dirección de cada dedo, pero en este caso los vectores los construimos entre la posición de cada dedo, porque lo que buscamos es saber la distancia entre dedos, si aumenta o disminuye. El *script* que podemos usar para comprobar el gesto sería el siguiente:

```
1.    using UnityEngine;
2.    public class ZoomTest: MonoBehaviour {
3.
4.        public float perspectiveZoomSpeed = 0.5 f;
5.        public float orthoZoomSpeed = 0.5 f;
6.        private Camera myCamera;
7.
8.        void Start() {
9.            myCamera = GetComponent < Camera > ();
10.           PinchRecognizer.PinchAction += Pinched;
11.       }
12.
13.       void Pinched(float deltaMagnitudeDiff) { //Hacemos cosas distintas
 si la cámara es ortho o perspectiva
14.           if (myCamera.orthographic) { //Cambiamos el tamaño de la cámara

15.               myCamera.orthographicSize += deltaMagnitudeDiff * orthoZoomS
peed; // Asegurarse que estamos en un rango adecuado
16.               myCamera.orthographicSize = Mathf.Max(myCamera.orthographicS
ize, 0.1 f);
17.           } else { //Cambiamos el field of view
18.               myCamera.fieldOfView += deltaMagnitudeDiff * perspectiveZoom
Speed; //Asegurarse que estamos en un rango adecuado
19.               myCamera.fieldOfView = Mathf.Clamp(myCamera.fieldOfView, 0.1
 f, 179.9 f);
20.           }
21.       }
22.   }
```

En este caso, hemos preparado este *script* para añadirlo a un *GameObject* que tenga un *component* «Camera», ya que el gesto *pinch* se suele usar para hacer zum.

4.3. Giroscopio y brújula

El giroscopio de un dispositivo móvil permite determinar cuál es su dirección. Normalmente, lo que nos interesa es el vector que nos indica dónde está el suelo o hacia dónde está mirando el dispositivo. Para algunas aplicaciones, como las relacionadas con mapas, también pueden interesar los datos de la brújula. Vamos a ver el primer caso:

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


3.    public class ExampleClass: MonoBehaviour {
4.        void Update() {
5.            transform.rotation = Input.gyro.attitude;
6.        }
7.    }
```

Este *script* hace girar el objeto que lo tenga añadido según el giroscopio. Si lo dejamos así, estará orientado hacia el suelo, lo que para algunas aplicaciones (por ejemplo, para simular un volante) puede no interesarnos. En ese caso, simplemente deberíamos añadir una rotación extra de 90º en el eje *X*.

Para obtener los datos de la brújula podemos usar el *script* siguiente:

```
using UnityEngine;
using System.Collections;


public class ExampleClass: MonoBehaviour {
    void Start() {
        Input.location.Start();
    }

    void Update() {
        transform.rotation = Quaternion.Euler(0, -
Input.compass.trueHeading, 0);
    }
}
```

«Input.location»

Es la propiedad que permite el acceso a los servicios de localización del dispositivo, como la brújula o el GPS.

El objeto que incluya este *script* girará hacia el norte geográfico. No debemos olvidarnos de activar el servicio de localización [`Input.location.Start()`], ya que sirve para calcular la desviación entre el norte magnético y el geográfico. Si nos interesase el norte magnético, deberíamos usar el método `Input.compass.magneticHeading`.

4.4. GPS

El GPS es un dispositivo que suele gastar bastante batería, por lo que normalmente los sistemas operativos y aplicaciones intentan minimizar su uso. Además, suele tardar cierto tiempo en encenderse y recibir señal; es por ello por lo que debemos evitar que bloquee la aplicación cuando esté realizando alguna de estas operaciones. Para ello, Unity nos ofrece una manera de definir los métodos llamada *Coroutine* ('corrutina'). Básicamente, una ***Coroutine*** es una manera específica de definir un método que no queremos esperar que se acabe de ejecutar. Se define con un `IEnumerator` como valor de retorno, y en vez de hacer una llamada a `return`, la tenemos que hacer a `yield`. La próxima vez que se ejecute el método, partirá desde la línea que hay a continuación de `yield`.

Nota

`yield` es especialmente útil en métodos que deben pausar su ejecución periódicamente, como los «Update».

El siguiente *script* sigue muchas de las recomendaciones sobre el uso del GPS gracias a las *Coroutines*:

```
1. using UnityEngine;
2. using System.Collections;
3. public class TestLocationService: MonoBehaviour {
4.     private bool isInitialized = false;
5.
6.     IEnumerator Start() {
7.    // Comprobamos que tenemos permiso para
8.    // acceder a datos de localización, y sino no continuamos
9.         if (!Input.location.isEnabledByUser) yield break;
10.            Input.location.Start();
11.            int maxWait = 20;
12.            // Esperamos a que se inicialice
13.            while (Input.location.status == LocationServiceStatus.Initializ
ing && maxWait > 0) {
14.                yield return new WaitForSeconds(1);
15.                maxWait--;
16.            }
17.            // Si no se ha podido inicializar en
18.            // 20 segundos algo ha ido mal
19.            if (maxWait < 1) {
20.                print("Error, no se ha podido iniciar");
21.                yield break;
22.            }
23.
24.            // Si algo ha salido mal acabamos el script
25.            if (Input.location.status == LocationServiceStatus.Failed) {
26.                print("Error, no se puede localizar dispositivo");
27.                yield break;
28.            } else {
29.                isInitialized = true;
30.            }
31.        }
32.
33.        void Update() {
34.            if (isInitialized) {
35.                Debug.Log("Latitud " + Input.location.lastData.latitude);
36.                Debug.Log("Longitud " + Input.location.lastData.longitude);

37.                Debug.Log("Altitud " + Input.location.lastData.altitude);
38.            }
39.        }
40.
41.        void OnDestroy() {
42.            if (isInitialized)
43.                Input.location.Stop();
44.        }
45.    }
```

Como podemos ver, hemos definido el método «Start» como una *Coroutine*, y Unity se encarga de llamarla correctamente (ver documentación *online* sobre cómo llamar a *Coroutines*). Cada vez que queremos que el método se espere, hacemos la llamada `yield return new WaitForSeconds(1);`, la cual pausará este método, pero no la aplicación, durante un segundo. En caso de error, en vez de hacer una llamada a `return`, lo que hacemos es usar `yield break;`.

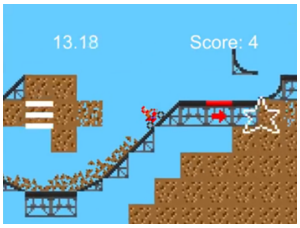
El *script* podría ser más complejo si tratase los casos en que el *script* se pausa, o si hubiese otros *scripts* en paralelo que usasen los servicios de localización, pero tal y como está nos sirve para la gran mayoría de casos.

5. Proyecto: Un juego de plataformas

En esta sección vamos a describir cómo desarrollar un juego 2D sencillo pero completo. Nuestro proyecto estará basado en un juego de plataformas clásico, en el que el usuario controla una moto por un terreno difícil y debe evitar caer boca abajo. Este proyecto va a trabajar fundamentalmente con la física y con la entrada del usuario.

Nota

En un juego de plataformas, el motor de física de Unity nos puede ser de gran utilidad.



5.1. El motorista

Vamos a bajar algunos recursos visuales para nuestro personaje principal. Como hemos dicho, este es un juego 2D basado en un juego clásico de controlar un motorista, por lo que podemos hacer una búsqueda en Open Game Art de la imagen de una moto. Partiremos de una escena vacía.

Reto 1

Añadid la imagen como *sprite* a la escena. A continuación, añadid un componente «RigidBody 2D» y tres «Colliders», uno para cada rueda y otro para el centro de la moto. Ajustadlos correctamente.

Nota

La moto se basará en el *sprite* «spr_chopper_0»:

http://opengameart.org/content/2d-bike-sprite-1.

Si ahora apretamos «Play», veremos que nuestro motorista cae porque nuestra escena aún no tiene un suelo. Vamos a empezar a construir la pista por la que jugaremos.

5.2. La escena

Hay multitud de recursos 2D *online* para crear un terreno. Lo que nos interesa para este proyecto es que podamos crear fácilmente desniveles que le añadan dificultad al terreno, y que los bloques se puedan añadir contiguamente sin que se vean discontinuidades (lo que se conoce como *tiles*). Elegiremos un fichero que contenga una cantidad de *sprites* considerable, con la que podamos hacer terrenos bastante variados. Aunque podríamos usar los *sprites* uno a uno, por separado, trabajar con un *sprite* compuesto es más eficiente y más cómodo si tenemos que cambiar algún parámetro a la hora de importar.

Nota

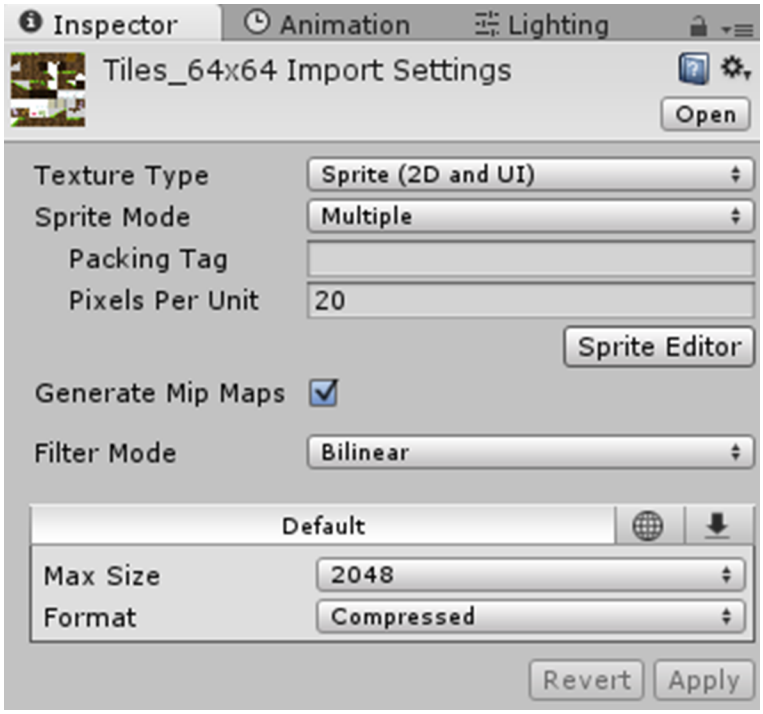
El terreno se basará en el atlas/*tileset Tiles_64x64*:

http://opengameart.org/content/platform-tileset-nature.

Reto 2

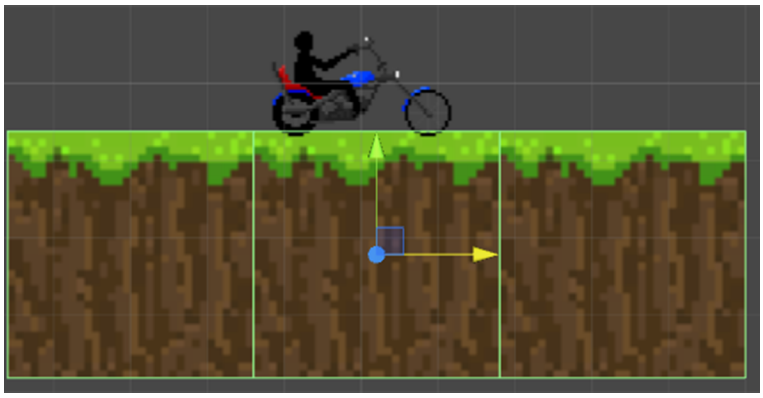
Importad correctamente el atlas de *sprites*, de modo que se pueda acceder de manera individual a cada una de las imágenes que lo componen. Para saber el tamaño de cada imagen individual, podéis consultar los datos proporcionados por el autor en Open Game Art.

Otro parámetro que debemos tener en cuenta es el tamaño que ocupa en pantalla cada *tile*, o porción del *sprite* múltiple. Para ello, podemos añadir un *tile* a la escena, cerca del motorista, y jugar cambiando el valor de «Pixels Per Unit» dentro de las opciones de importación del *sprite* múltiple:



Para el caso de este proyecto, un valor adecuado puede ser 20, ya que con este valor cada *tile* es de aproximadamente el mismo tamaño que nuestro motorista, que hemos importado con un valor de «Pixels Per Unit» igual a 100.

Al añadir una *tile*, será necesario incorporar «BoxCollider2D». Al tratarse de imágenes ya de por sí, Unity es inteligente y ajusta el tamaño del «BoxCollider2D». Podemos situar algunos *tiles* debajo del motorista, apretar «Play» y ver cómo se estabiliza:



Antes de hacer un diseño completo del nivel, vamos a darle comportamiento al motorista, a comprobar que su velocidad es adecuada para los obstáculos que le queremos poner, entre otros aspectos.

5.3. Entrada

Vamos a añadirle comportamiento al motorista para que responda a nuestra interacción. Lo que queremos es tener cuatro acciones distintas: acelerar, frenar y girar en los dos sentidos (elevando la rueda delantera o la trasera). Vamos a programar estas funciones:

```
1.  using UnityEngine;
2.  using System.Collections;
3.  public class MotoControl: MonoBehaviour {
4.      public float velocidadRotacion = 50;
5.      public float velocidadLineal = 1;
6.      Rigidbody2D rigidbody;
7.      void Start() {
8.          rigidbody = GetComponent < Rigidbody2D > ();
9.      }
10.     public void MueveDerecha() {
11.         rigidbody.velocity
+= new Vector2(transform.right.x * velocidadLineal, transform.right.y * veloci
dadLineal) * Time.deltaTime;
12.     }
13.     public void MueveIzquierda() {
14.         rigidbody.velocity -
= new Vector2(transform.right.x * velocidadLineal, transform.right.y * velocid
adLineal) * Time.deltaTime;
15.     }
16.     public void RotaDerecha() {
17.         rigidbody.MoveRotation(rigidbody.rotation -
 velocidadRotacion * Time.deltaTime);
18.     }
19.     public void RotaIzquierda() {
20.         rigidbody.MoveRotation(rigidbody.rotation + velocidadRotacion * Ti
me.deltaTime);
21.     }
22.     void Update() {
23.         if (Input.GetKey(KeyCode.LeftArrow)) {
24.             MueveIzquierda();
25.         }
26.         if (Input.GetKey(KeyCode.RightArrow)) {
27.             MueveDerecha();
28.         }
29.         if (Input.GetKey(KeyCode.UpArrow)) {
30.             RotaIzquierda();
31.         }
32.         if (Input.GetKey(KeyCode.DownArrow)) {
33.             RotaDerecha();
34.         }
35.     }
36. }
```

Con este *script* ya podemos controlar el motorista si lo añadimos al *GameObject* que contiene el *sprite* que lo contiene. Aún nos quedan algunas cuestiones por arreglar. Por ejemplo, aún no detectamos si el motorista se ha dado la vuelta o no, y no controlamos si estamos tocando el suelo para acelerar (para este juego nos puede interesar hacer esta comprobación ya que le añade dificultad). Le podemos añadir esta funcionalidad en este mismo *script*:

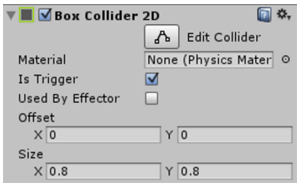
```
1.  using UnityEngine;
2.  using System.Collections;
3.
4.  public class MotoControl: MonoBehaviour {
5.      public float velocidadRotacion = 50;
6.      public float velocidadLineal = 1;
7.      public Transform ruedaTrasera;
8.      public event eliminadoDelegate eliminado;
9.      public delegate void eliminadoDelegate();
10.
11.     private Rigidbody2D rigidbody;
12.     private float radioRueda;
13.
14.     void Start() {
15.         rigidbody = GetComponent < Rigidbody2D > ();
16.         radioRueda = GetComponent < CircleCollider2D > ().radius + 0.1 f;

17.     }
18.
19.     public void MueveDerecha() {
20.         if (TocaElSuelo()) rigidbody.velocity += new Vector2(transform.rig
ht.x * velocidadLineal, transform.right.y * velocidadLineal) * Time.deltaTime;

21.     }
22.
23.     public void MueveIzquierda() {
24.         if (TocaElSuelo()) rigidbody.velocity -
= new Vector2(transform.right.x * velocidadLineal, transform.right.y * velocid
adLineal) * Time.deltaTime;
25.     }
26.
27.     public void RotaDerecha() {
28.         rigidbody.MoveRotation(rigidbody.rotation -
 velocidadRotacion * Time.deltaTime);
29.     }
30.
31.     public void RotaIzquierda() {
32.         rigidbody.MoveRotation(rigidbody.rotation + velocidadRotacion * Ti
me.deltaTime);
33.     }
34.
35.     void Update() {
36.         if (Input.GetKey(KeyCode.LeftArrow)) {
37.             MueveIzquierda();
38.         }
39.         if (Input.GetKey(KeyCode.RightArrow)) {
40.             MueveDerecha();
41.         }
42.         if (Input.GetKey(KeyCode.UpArrow)) {
43.             RotaIzquierda();
44.         }
45.         if (Input.GetKey(KeyCode.DownArrow)) {
46.             RotaDerecha();
47.         }
48.     }
49.
50.     void OnTriggerEnter2D(Collider2D other) {
51.         if (other.gameObject != gameObject) {
52.             if (eliminado != null)
53.                eliminado();
54.         }
55.     }
56.
57.     bool TocaElSuelo() {
58.         if (Physics2D.OverlapCircleAll(ruedaTrasera.position, radioRueda).
Length > 0) {
59.             return true;
60.         } else {
61.             return false;
62.         }
63.     }
64. }
```

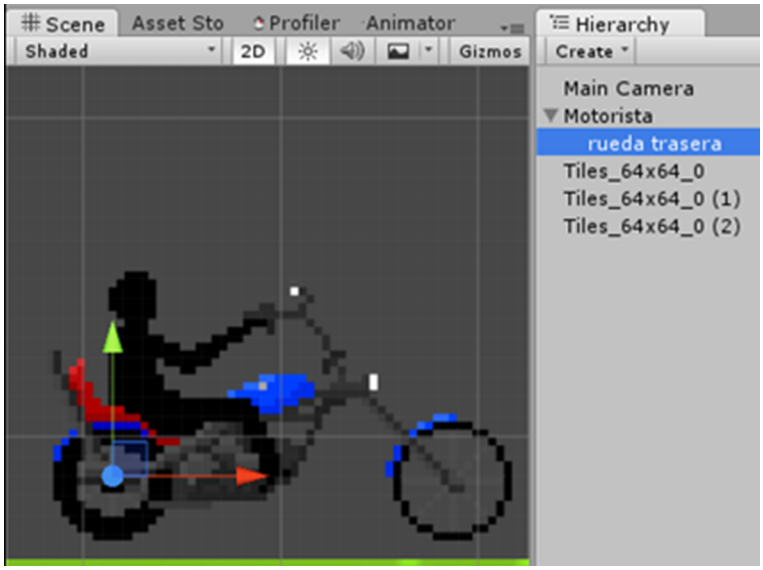
Nota

Marcamos el «BoxCollider2D» como «Is Trigger» para entrar en el método `OnTriggerEnter2D` a cada colisión con la caja.



Como vemos, hemos incluido un evento con la forma `void eliminado()`, que se activará cuando detectemos que el motorista se ha dado la vuelta. Esto lo logramos marcando el «BoxCollider2D» como «Is Trigger», que nos hará una llamada a `OnTriggerEnter2D` cuando detecte una colisión. También hemos añadido la función «TocaElSuelo», que comprueba si la rueda trasera

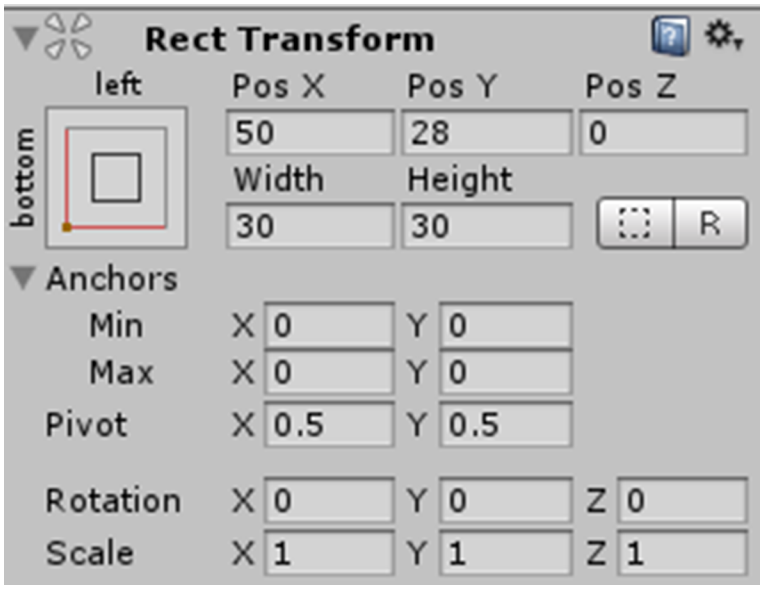
está tocando el suelo o no. Para que funcione, en el editor debemos crear un *GameObject* hijo del motorista, moverlo al centro de la rueda y asignarlo al parámetro «ruedatrasera»:



Creamos el *GameObject* «rueda trasera».

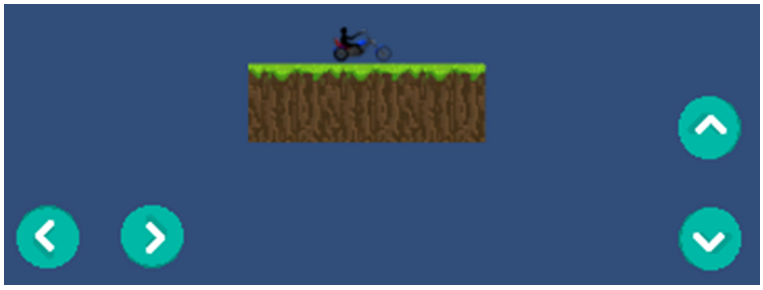
Ahora que tenemos un control que funciona con el teclado, vamos a adaptarlo a móvil y vamos a crear una pequeña interfaz. El paquete de *assets* que hemos usado en esta ocasión se llama *Colorful Buttons* y lo podemos encontrar en la Asset Store de Unity. De este paquete usaremos el icono que se llama «Left» (está en la ruta «Colorful Buttons > Left», dentro de nuestros *assets*); se lo pondremos a todos los botones añadiendo una rotación. De esta manera, solo cargaremos un icono en memoria. Es una optimización pequeña, pero todo cuenta.

Lo que haremos será ir a «Menú > GameObject > UI > Button», lo que nos añade el botón por defecto a la escena. Lo primero que hay que hacer es cambiar el *sprite* del *component* «Image» en el *GameObject* «Button» (que acabamos de generar) y asignarle el *sprite* «Left» desde el paquete que hemos descargado de la Asset Store. Como los tamaños son muy distintos, tenemos que ajustar este parámetro. En el «RectTransform» del *GameObject* «Button» hay que configurar varios parámetros. En la siguiente figura vemos un ejemplo de configuración válido, aunque la posición y el tamaño se pueden ajustar según el gusto de cada uno.



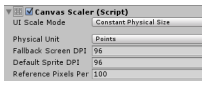
«RectTransform» del *GameObject* «Button» configurado.

Los otros tres botones los podemos colocar de la manera que se muestra en la siguiente figura. Nótese que para los botones que están en la parte derecha, tendremos que cambiar los anchos «Min X» y «Max X» a 1 para que se queden fijos en la parte derecha de la pantalla sin importar el tamaño ni la resolución de la pantalla.



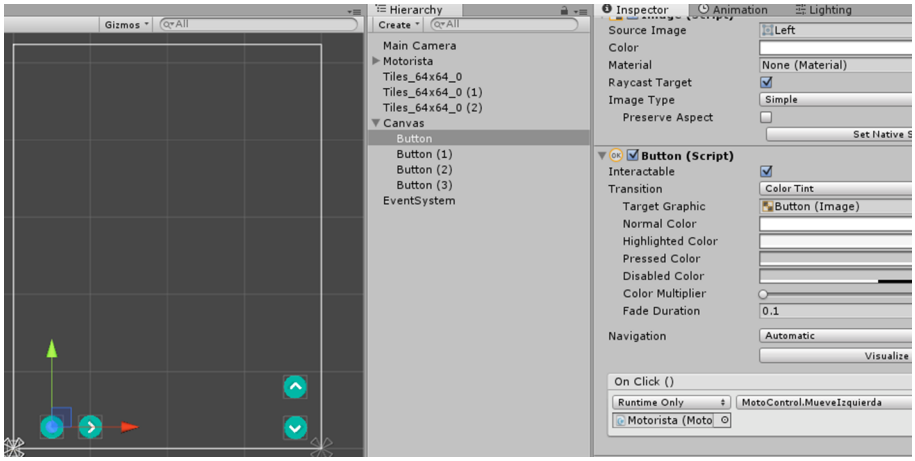
Diseño de los botones para el usuario.

Es posible que al probar el estado actual en un móvil, los botones aparezcan minúsculos. Para arreglarlo, debemos cambiar el parámetro de UI «Scale Mode», en el *component* «Canvas Scaler» del *GameObject* «Canvas», en la opción «Constant Physical Size»:



El «Canvas Scaler» configurado para que los botones se vean bien.

Para darles funcionalidad a los botones, tenemos que añadir a «OnClick» las funciones pertinentes. Cada botón tendrá asignado el método correspondiente del *script* «MotoControl» (el botón «Left» se asigna a `MueveIzquierda`, el botón «Right» a `MueveDerecha`, etc.).



El botón con el evento «OnClick» asignado correctamente.

Sin embargo, si lo configuramos así tenemos un problema. «OnClick» se activa una sola vez por cada vez que tocamos el botón, es decir que, por mucho tiempo que aguantemos con el dedo encima del botón, solo recibiremos un evento. Nuestro *script*, en cambio, está pensado para acelerar/frenar recibiendo eventos continuamente.

Es por eso que Unity nos ofrece algunas herramientas más, como el componente «EventTrigger». Sin embargo, para nuestro caso los eventos que nos ofrece «EventTrigger» no nos son suficientes, ya que solo tenemos «OnPointerDown» y «OnPointerUp», que detectan cuando empezamos a apretar un botón y cuando dejamos de apretarlo. Necesitamos alguna manera de ejecutar un código mientras se aprieta el botón.

Nota

«EventTrigger» se utiliza para especificar las funciones que deseamos llamar para cada evento «EventSystem». Añadir este componente a un *GameObject* hará que el objeto intercepte todos los eventos, y ningún evento se propagará a los objetos padre.

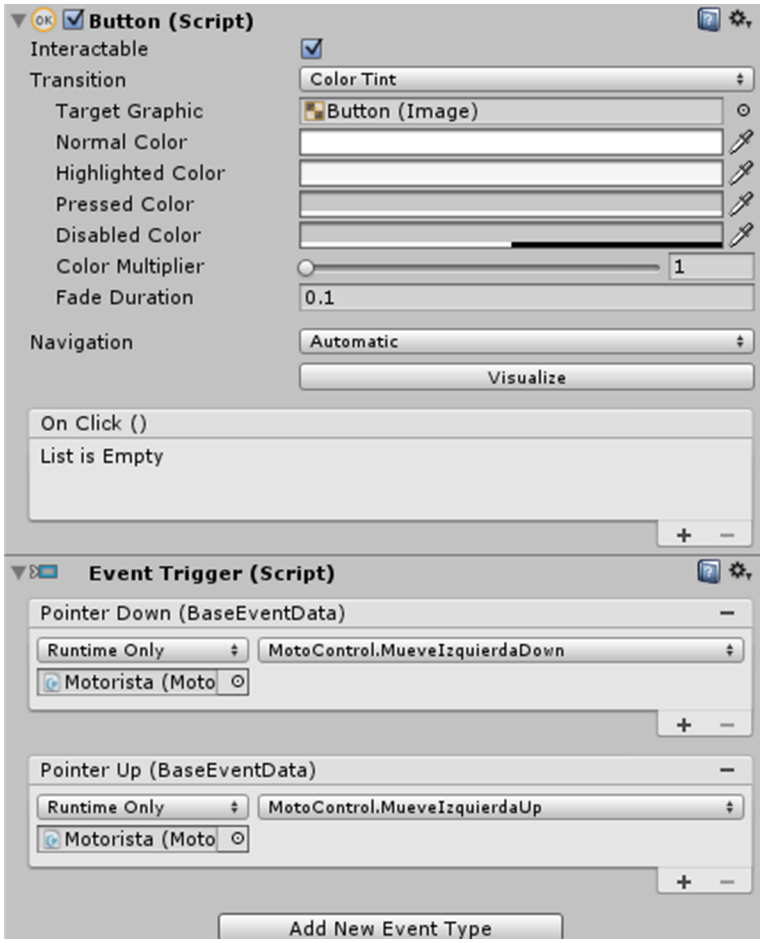
Reto 3

Adaptad el código de «MotoControl» para aprovechar los eventos de «OnPointerDown» y «OnPointerUp» de manera que se ejecuten los métodos `MueveIzquierda`, `MueveDerecha`, `RotaIzquierda` y `RotaDerecha` una vez por *frame* mientras los respectivos botones están apretados. Cread métodos llamados `MueveIzquierdaUp`, `MueveIzquierdaDown`, `MueveDerechaUp`, `MueveDerechaDown`, `RotaIzquierdaUp`, `RotaIzquierdaDown`, `RotaDerechaUp` y `RotaDerechaDown`.

Nota

Una forma de hacerlo es utilizando una «List» de «KeyCode». En reacción a «OnPointerDown» podemos añadir un «KeyCode» y en el «OnPointerUp» removerlo, etc. Así, la «List» tiene las informaciones necesarias para los movimientos.

Respecto a cómo se deberían configurar los botones, primero deberíamos borrar todos los eventos «OnClick» que habíamos configurado antes. Después deberíamos añadir un componente «EventTrigger» por cada botón y añadir dos tipos de eventos por cada «EventTrigger» usando el botón «Add New Event Type». Los tipos de eventos son «Pointer Down» y «Pointer Up», y los debemos asociar a los métodos de «MotoControl» adecuados; veamos un ejemplo:

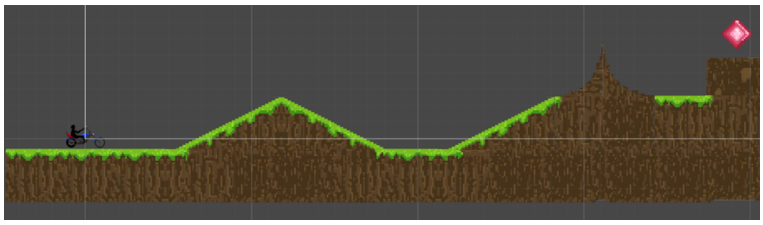


Configuración del botón izquierdo para hacer las llamadas correctas al control del motorista.

Con la entrada funcionando, podemos empezar a diseñar un nivel y añadirle elementos de juego.

5.4. Diseño de un nivel

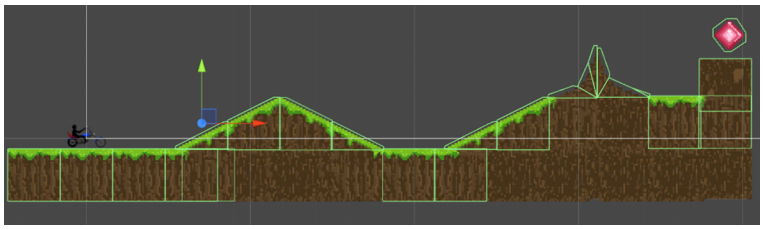
La clave a la hora de diseñar un nivel es el equilibrio en la dificultad. Un nivel demasiado difícil puede ser frustrante, pero si es demasiado fácil puede que no resulte un desafío interesante. En este caso nosotros hemos diseñado el nivel como en la siguiente figura:



Nivel completo diseñado para este juego.

El objetivo del jugador en este nivel es llegar hasta la parte derecha, donde está el diamante rosa. Hay varias cuestiones que tenemos que tener en cuenta para que un nivel esté completo y sea correcto. La primera, es que debe ser posible acabarlo y que debe tener comportamientos coherentes. Por ejemplo, debemos asegurarnos de que cada *tile* en donde se puede apoyar el motorista tiene

un «Collider» añadido. Si no fuera así, el motorista caería al vacío y el jugador lo percibiría como incorrecto. Podemos hacer una comprobación rápida seleccionando todo el terreno para poder ver en el editor todos los «Colliders» en verde. Para los *tiles* que no son cuadrados hemos usado «PolygonCollider2D», y Unity ha ajustado automáticamente la forma.



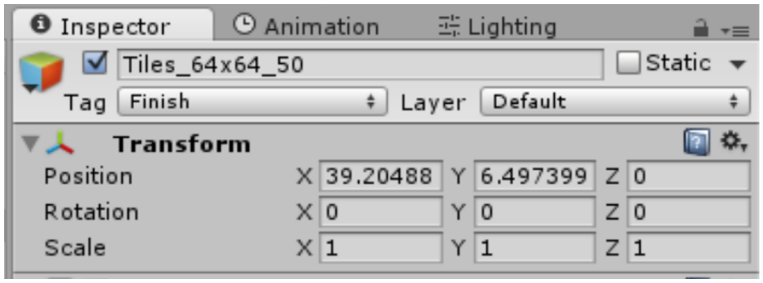
Terreno seleccionado, los «Colliders» se marcan en verde.

En segundo lugar, el motorista debe poder ser controlable y también capaz de llegar hasta el final. Si dejamos la configuración tal como estaba hasta ahora, no será capaz de subir cuestas. Haciendo algunas pruebas podemos ver que una velocidad adecuada (parámetro «Velocidad Lineal» del motorista) es 10.

Un tercer aspecto es que hay que ofrecer un aliciente de gamificación, es decir, un sistema de recompensas y de desafíos para el jugador, además de un principio y un final. Vamos a ver cómo hacerlo de manera simple.

El jugador tiene que ser consciente de cuándo empieza y cuándo acaba el juego, y también de cuándo lo acaba con éxito o no. Hemos dicho que el objetivo del jugador es llegar al diamante rosa, y un desafío simple que podemos ofrecer es el tiempo que tarda en lograrlo. También tenemos que hacer algo cuando el motorista se da la vuelta, además de controlar el movimiento de la cámara. Vamos a empezar por detectar cuando llegamos al diamante, cuando se da la vuelta y cuando se cae al vacío.

Primero tenemos que poder distinguir entre si lo que está tocando el motorista es el diamante o cualquier otra cosa. Lo podemos hacer mediante una herramienta muy útil de Unity, que son los ***Tags***. Por ejemplo, podemos asignar el *Tag* «Finish» al diamante y luego detectar la colisión con el diamante leyendo el *Tag* del objeto colisionado.



Asignando el *Tag* «Finish» al diamante.

Tags

Un *Tag* es simplemente una etiqueta, que podemos leer dentro de un *script* para realizar una acción u otra.

Reto 4

Adaptad el código de «MotoControl» para saber cuándo hemos tocado el diamante rosa. Cread un nuevo evento llamado `eliminado` que se llame solo cuando la moto toque el diamante rosa, leyendo el *Tag* del objeto colisionado.

Para conseguirlo, hay que utilizar el *Tag* para identificar qué objeto entra en colisión con el motorista en el método `OnTriggerEnter2D`.

Reto 5

Cread un *script* que reinicie el nivel cuando la moto haya colisionado con un objeto que no sea el diamante, como, por ejemplo, cuando la moto se da la vuelta. Usad el evento implementado en el reto anterior.

En este caso, tenemos que aprovechar el evento `eliminado`. Sabemos que para suscribir el evento tenemos que añadir una función a `eliminado` y que esta función será llamada automáticamente cuando ocurra el evento eliminado, es el principio del *delegate*, etc. Por lo que solo hace falta hacer una nueva clase con un método para reiniciar la escena. Lo habitual es hacer un *script* «GamePlay», en el que ponemos todos estos métodos relacionados al funcionamiento del juego.

Reto 6

Completad la escena de manera que, sin modificar *scripts*, también detecte que la moto se ha salido de la pantalla. Añadid objetos vacíos, solo con «Colliders», para utilizar el sistema de detección de colisiones de Unity.

Objetos vacíos

En Unity se usan mucho los objetos vacíos. Por ejemplo, para ejecutar un *script*, detectar algo, activar un evento, etc.

Cuando detectamos que el motorista colisiona con uno de estos objetos vacíos, ya ha fallado y reiniciamos el level.

También tenemos que tener en cuenta que si el motorista se mueve, pero la cámara no, lo perderemos de vista. Con un simple *script* podemos arreglar este problema.

Reto 7

Creas un *script* que mueva la cámara siguiendo a la moto, siempre desde la misma distancia.

«Transform»

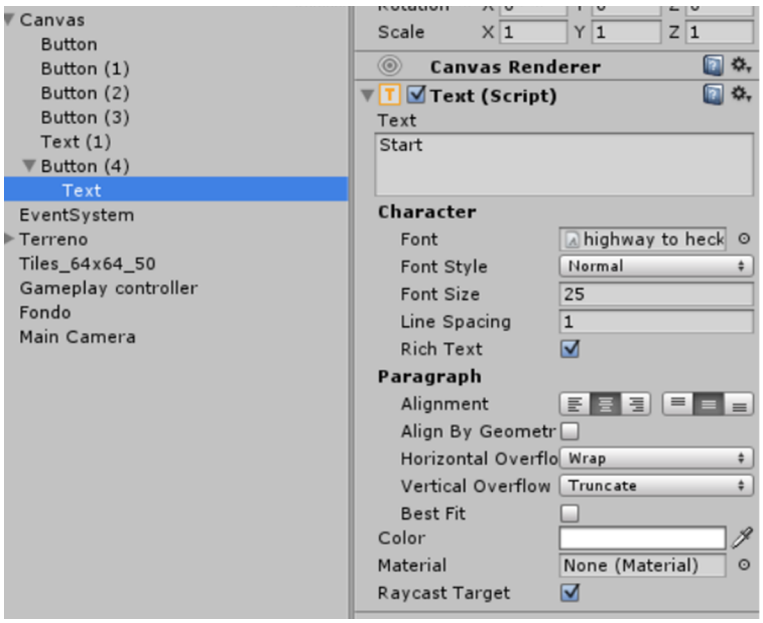
En Unity, cada objeto en una escena tiene un «Transform». Se utiliza para almacenar y manipular la posición, rotación y escala del objeto. Cada «Transform» puede tener un padre, lo que le permite aplicar la posición, la rotación y la escala jerárquicamente.

Para lograrlo, hay que añadir una distancia fija a la posición actual del motorista para obtener la posición de la cámara. Esto también se debería añadir al *script* «GamePlay».

5.5. La UI

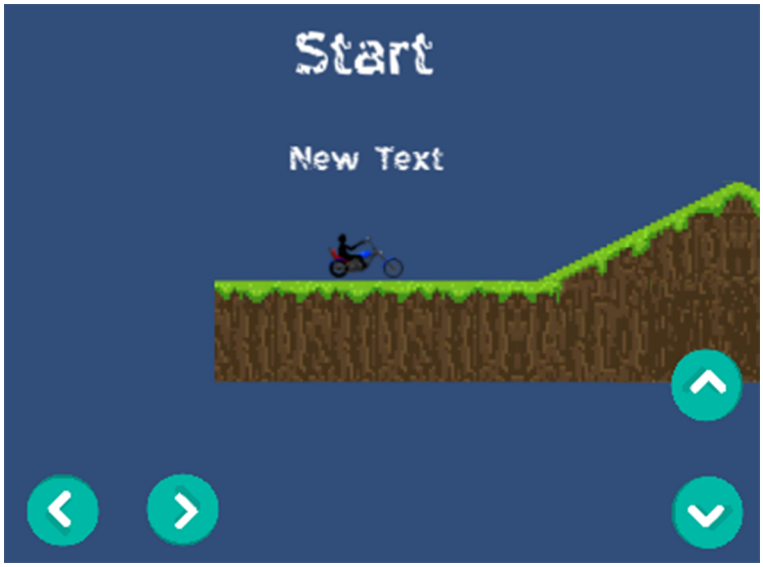
Ahora vamos a gestionar el caso de éxito y mostrar el tiempo transcurrido en una interfaz, guardando el récord. Para hacerlo de manera un poco más vistosa, primero nos bajaremos un *pack* de fuentes. En este caso, hemos utilizado el paquete llamado *Grunge Font Pack* de la Asset Store.

Primero hemos añadido un botón hijo del *GameObject* «Canvas» mediante «Menú > GameObject > UI > Button». El botón por defecto nos muestra un *sprite* (en el campo «Source Image») que no nos interesa en este caso. Lo podemos borrar y cambiar el alfa del campo «Color» a 0, de manera que nos quedará transparente. El texto del botón, que está como hijo del botón, tiene una fuente por defecto que vamos a cambiar. Le pondremos en este caso la «highway to heck», y le cambiaremos el tamaño de la fuente y el color a blanco (como vemos en la imagen). En el campo «Text» pondremos «Start», para indicar que hay que apretarlo para empezar.



Texto del botón.

También añadiremos un campo de texto mediante «Menú > GameObject > UI > Text», en el que realizaremos las mismas operaciones, pero le pondremos un tamaño de fuente un poco más pequeño, por ejemplo 14. La idea es que nos queden centradas y en la parte superior, como en siguiente figura:



La UI de nuestro juego.

Ahora hay que controlar estos elementos, para que muestren lo que nos interesa.

Para ello podemos modificar el *script* «GamePlay» definido en los retos anteriores de la siguiente manera:

```
1.  using UnityEngine;
2.  using UnityEngine.UI;
3.  using System.Collections;
4.
5.  public class GamePlay: MonoBehaviour {
6.      public MotoControl motorista;
7.      public Text recordText;
8.      public Button startButton;
9.      private int segundosParaEmpezar = 3;
10.     private Text mainText;
11.     private float tiempoInicial;
12.
13.     void Start() {
14.         motorista.eliminado += Reiniciar;
15.         motorista.finalNivel += Final;
16.         motorista.enabled = false;
17.         mainText = startButton.GetComponentInChildren <Text> ();
18.     }
19.
20.     void Reiniciar() {
21.         UnityEngine.SceneManagement.SceneManager.LoadScene(UnityEngine.Sce
neManagement.SceneManager.GetActiveScene().buildIndex);
22.     }
23.
24.     public void StartGame() {
25.         startButton.enabled = false;
26.         mainText.text = "" + segundosParaEmpezar;
27.         InvokeRepeating("CuentaAtras", 1, 1);
28.     }
29.
30.     void CuentaAtras() {
31.         segundosParaEmpezar--;
32.         if (segundosParaEmpezar <= 0) {
33.             CancelInvoke();
34.             JuegoEmpezado();
35.         } else mainText.text = "" + segundosParaEmpezar;
36.     }
37.
38.     void JuegoEmpezado() {
39.         motorista.enabled = true;
40.         tiempoInicial = Time.time;
41.     }
42.
43.     void Update() {
44.         if (motorista.enabled) {
45.             mainText.text = "Tiempo: " + (Time.time -
 tiempoInicial).ToString("##.##");
46.         }
47.     }
48.
49.     void Final() {
50.         motorista.enabled = false;
51.         mainText.text = "Final! " + (Time.time - tiempoInicial);
52.     }
53. }
```

Nota

Si no tenéis un *script* «GamePlay», se puede hacer en cualquier *script*, siempre que tenga los elementos necesarios, como el motorista.

En esta versión del *script* hay algunos conceptos que no hemos visto. Por ejemplo, `InvokeRepeating` es un método propio de Unity que se encarga de llamar al método que le indiquemos cada cierto tiempo. Se podría implementar de múltiples maneras, pero esta es muy cómoda en este caso. Llamamos a `CuentaAtras` hasta que nuestro contador llega a 0, y cancelamos el `InvokeRepeating`.

También proponemos usar *PlayerPrefs*, que es un equivalente a una pequeña base de datos que gestiona Unity. Podemos guardar números enteros, números con decimales o texto. En este caso, es muy adecuado para guardar el récord de tiempo del usuario.

PlayerPref

En Unity, *PlayerPref* almacena y accede a las preferencias de los jugadores entre sesiones de juego. El lugar concreto y la forma de guardar los datos dependen de la plataforma, pero es transparente para el usuario.

Reto 8

Modificad el *script* para guardar el récord de tiempo del usuario usando *PlayerPrefs*.

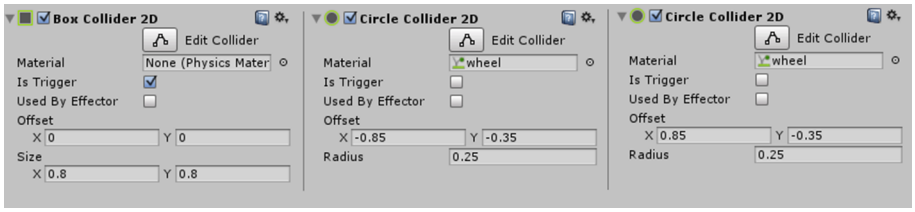
Ahora asignamos al evento «OnClick» del botón «Start» que acabamos de crear el método «StartGame», de esta manera el jugador podrá intentar llegar al diamante rosa, y si lo consigue se guardará el record. Sino, la partida se reiniciará.

Tenemos un nivel del juego completo, podemos ahora intentar modificar y mejorar el juego añadiendo otros niveles.

5.6. Soluciones a los retos propuestos

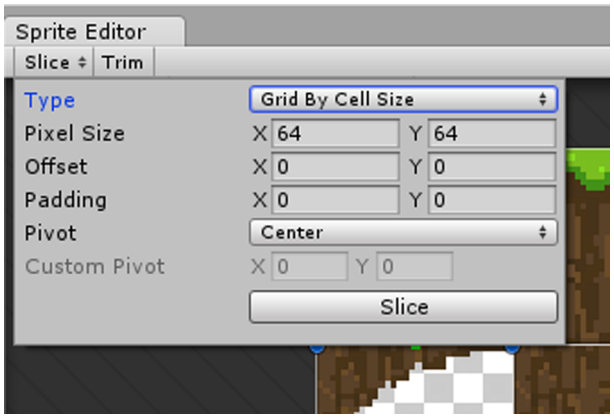
Reto 1

Una posible configuración de los componentes, dado el *sprite* «2d-bike-sprite-1», puede ser la siguiente.



Reto 2

Para importar correctamente un *sprite* con múltiples imágenes es necesario seleccionar «Sprite Mode Multiple». Entonces, en el editor de *sprites*, seleccionar «Slice» y la opción «Type» como «Grid By Cell Size», con «Pixel Size» igual a 64 x 64.


Sprite Editor
Slice
Trim
Type
Grid By Cell Size
Pixel Size
X 64
Y 64
Offset
X 0
Y 0
Padding
X 0
Y 0
Pivot
Center
Custom Pivot
X 0
Y 0
Slice

Reto 3

```
1.  using UnityEngine;
2.  using System.Collections.Generic;
3.  public class MotoControl: MonoBehaviour {
4.      public float velocidadRotacion = 50;
5.      public float velocidadLineal = 1;
6.      public Transform ruedaTrasera;
7.      public event eliminadoDelegate eliminado;
8.      public delegate void eliminadoDelegate();
9.      private Rigidbody2D rigidbody;
10.     private float radioRueda;
11.     private List <KeyCode> acciones = new List <KeyCode> ();
12.     void Start() {
13.         rigidbody = GetComponent < Rigidbody2D > ();
14.         radioRueda = GetComponent <CircleCollider2D>().radius + 0.1 f;
15.     }
16.     public void MueveDerecha() {
17.         if (TocaElSuelo()) rigidbody.velocity += new Vector2(transform.rig
ht.x * velocidadLineal, transform.right.y * velocidadLineal) * Time.deltaTime;

18.     }
19.     public void MueveIzquierda() {
20.         if (TocaElSuelo()) rigidbody.velocity -
= new Vector2(transform.right.x * velocidadLineal, transform.right.y * velocid
adLineal) * Time.deltaTime;
21.     }
22.     public void RotaDerecha() {
23.         rigidbody.MoveRotation(rigidbody.rotation -
 velocidadRotacion * Time.deltaTime);
24.     }
25.     public void RotaIzquierda() {
26.         rigidbody.MoveRotation(rigidbody.rotation + velocidadRotacion * Ti
me.deltaTime);
27.     }
28.     void Update() {
29.         ActualizarAccionTeclado(KeyCode.LeftArrow);
30.         ActualizarAccionTeclado(KeyCode.RightArrow);
31.         ActualizarAccionTeclado(KeyCode.UpArrow);
32.         ActualizarAccionTeclado(KeyCode.DownArrow);
33.
34.         if (acciones.Contains(KeyCode.LeftArrow)) {
35.             MueveIzquierda();
36.         }
37.         if (acciones.Contains(KeyCode.RightArrow)) {
38.             MueveDerecha();
39.         }
40.         if (acciones.Contains(KeyCode.UpArrow)) {
41.             RotaIzquierda();
42.         }
43.         if (acciones.Contains(KeyCode.DownArrow)) {
44.             RotaDerecha();
45.         }
46.     }
47.     private void ActualizarAccionTeclado(KeyCode code) {
48.         if (Input.GetKeyDown(code)) {
49.             ActualizarAccionDown(code);
50.         }
51.         if (Input.GetKeyUp(code)) {
52.             ActualizarAccionUp(code);
53.         }
54.     }
55.     private void ActualizarAccionDown(KeyCode code) {
56.         if (!acciones.Contains(code)) acciones.Add(code);
57.     }
58.     private void ActualizarAccionUp(KeyCode code) {
59.         if (acciones.Contains(code)) acciones.Remove(code);
60.     }
61.     public void MueveDerechaDown() {
62.         ActualizarAccionDown(KeyCode.RightArrow);
63.     }
64.     public void MueveIzquierdaDown() {
65.         ActualizarAccionDown(KeyCode.LeftArrow);
66.     }
67.     public void RotaIzquierdaDown() {
68.         ActualizarAccionDown(KeyCode.UpArrow);
69.     }
70.     public void RotaDerechaDown() {
71.         ActualizarAccionDown(KeyCode.DownArrow);
72.     }
73.     public void MueveDerechaUp() {
74.         ActualizarAccionUp(KeyCode.RightArrow);
75.     }
76.     public void MueveIzquierdaUp() {
77.         ActualizarAccionUp(KeyCode.LeftArrow);
78.     }
79.     public void RotaIzquierdaUp() {
80.         ActualizarAccionUp(KeyCode.UpArrow);
81.     }
82.     public void RotaDerechaUp() {
83.         ActualizarAccionUp(KeyCode.DownArrow);
84.     }
85.     void OnTriggerEnter2D(Collider2D other) {
86.         if (other.gameObject != gameObject) {
87.             if (eliminado != null) eliminado();
88.         }
89.     }
90.     bool TocaElSuelo() {
91.         if (Physics2D.OverlapCircleAll(ruedaTrasera.position, radioRueda).
Length > 0) {
92.             return true;
93.         } else {
94.             return false;
95.         }
96.     }
97. }
```

Reto 4

```
using UnityEngine;
using System.Collections.Generic;
public class MotoControl: MonoBehaviour {

    public event triggerDelegate eliminado;
    public event triggerDelegate finalNivel;
    [...]// Falta el código que no ha cambiado.


    void OnTriggerEnter2D(Collider2D other) {
        if (other.gameObject.tag.CompareTo("Finish") != 0) {
            if (eliminado != null) eliminado();
        }
        else
            if (finalNivel != null) finalNivel();
    }

    bool TocaElSuelo() {
        if (Physics2D.OverlapCircleAll(ruedaTrasera.position, radioRued
a).Length > 0) {
            return true;
        } else {
            return false;
        }
    }
}
```

Hemos añadido un *event* más, para diferenciar cuál es el objeto con el que hemos colisionado, si es el diamante o no. También hemos modificado el método `OnTriggerEnter2D`, con el mismo objetivo.

Reto 5

```
using UnityEngine;
using System.Collections;
public class GamePlay: MonoBehaviour {
    public MotoControl motorista;
    void Start() {
        motorista.eliminado += Reiniciar;
    }

    void Reiniciar() {
        UnityEngine.SceneManagement.SceneManager.LoadScene(UnityEngine.S
ceneManagement.SceneManager.GetActiveScene().buildIndex);
    }
}
```

Si en el editor asignamos al motorista en el campo correspondiente del *script* «GamePlay», veremos que cada vez que se da la vuelta el nivel se reinicia.

Reto 6

Podemos crear un objeto vacío y situarlo debajo del nivel. Si le añadimos un «BoxCollider2D» con tamaño X 100 y tamaño Y 1, cada vez que nuestro motorista se caiga al vacío se reiniciará el nivel. También podemos replicar este objeto por los lados para rodear la pantalla y asegurarnos de que no tendremos errores.

Reto 7

```
using UnityEngine;
using System.Collections;
public class MotoristaFollower: MonoBehaviour {

  public Transform motorista;
  void Update() {
    transform.position = motorista.position - Vector3.forward*10;
    }
}
```

Simplemente debemos añadir este *script* a la cámara y asignar el motorista al campo correspondiente en el editor.

Reto 8

El *script* «GamePlay» completo:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class GamePlay: MonoBehaviour {
    public MotoControl motorista;
    public Text recordText;
    public Button startButton;
    private int segundosParaEmpezar = 3;
    private Text mainText;
    private float tiempoInicial;
    private float tiempoRecord;

    void Start() {
        motorista.eliminado += Reiniciar;
        motorista.finalNivel += Final;
        motorista.enabled = false;
        mainText = startButton.GetComponentInChildren <Text> ();
        tiempoRecord = PlayerPrefs.GetFloat("tiempo record", 0);
        if (tiempoRecord > 0) recordText.text = "Record: " + tiempoRecor
d.ToString("##.##");
        recordText.enabled = false;
    }

    void Reiniciar() {
        UnityEngine.SceneManagement.SceneManager.LoadScene(UnityEngine.Sce
neManagement.SceneManager.GetActiveScene().buildIndex);
    }

    public void StartGame() {
        startButton.enabled = false;
        mainText.text = "" + segundosParaEmpezar;
        InvokeRepeating("CuentaAtras", 1, 1);
    }

    void CuentaAtras() {
        segundosParaEmpezar--;
        if (segundosParaEmpezar <= 0) {
            CancelInvoke();
            JuegoEmpezado();
        } else mainText.text = "" + segundosParaEmpezar;
    }

    void JuegoEmpezado() {
        motorista.enabled = true;
        tiempoInicial = Time.time;
        if (tiempoRecord > 0) recordText.enabled = true;
    }

      void Update() {
          if (motorista.enabled) {
              mainText.text = "Tiempo: " + (Time.time -
 tiempoInicial).ToString("##.##");
          }
      }

      void Final() {
          motorista.enabled = false;
          mainText.text = "Final! " + (Time.time - tiempoInicial);
          PlayerPrefs.SetFloat("tiempo record", (Time.time -
 tiempoInicial));
      }
  }
```

Resumen

En este módulo didáctico hemos trabajado con algunos de los conceptos más importantes para desarrollar juegos 2D. El primero es el de los *sprites*, que son básicamente imágenes o texturas. Los elementos visuales en juegos 2D son en esencia *sprites*, y Unity tiene unas cuantas herramientas para optimizar el uso de las texturas. Por ejemplo, podemos tener varios elementos dentro de una misma textura para ahorrar memoria. Unity nos ofrece maneras de poder trabajar en esta situación de manera prácticamente transparente.

La física es otro de los componentes más necesarios a la hora de desarrollar un videojuego. Ya sea para simular un comportamiento físico realista, ya para detectar colisiones y poder actuar nosotros de alguna manera, la física nos puede ayudar enormemente. Hemos visto en el segundo apartado los elementos en Unity que nos permiten definir qué elementos queremos simular físicamente, así como los que nos permiten detectar colisiones. También hemos visto cómo podemos capturar los eventos de física vía *scripting*.

Además, hemos introducido lo básico respecto a la interacción con el usuario y con los sensores de un dispositivo móvil. Típicamente, interactuamos mediante la pantalla táctil de nuestro móvil, tanto con toques como con gestos con uno o varios dedos. Más allá de la simple pulsación, algunos de los gestos más comunes son *swipe* y *pinch*. También hemos trabajado con los sensores de giroscopio y de GPS, que pueden sernos muy útiles para una gran cantidad de posibles aplicaciones.

Seguidamente hemos visto cómo se exporta un proyecto a una plataforma móvil. Unity es un motor capaz de generar aplicaciones de un mismo proyecto para una gran cantidad de plataformas, y en algunas de ellas debemos tener en cuenta algunos parámetros. Hemos aprendido aquí todo lo necesario para configurar Unity de manera que podamos instalar y probar nuestras aplicaciones en nuestro dispositivo móvil.

Finalmente, hemos seguido todos los pasos necesarios para diseñar un juego 2D desde cero. Este proyecto está basado en un juego clásico que tuvo bastante éxito hace unos cuantos años, y en este módulo hemos visto cómo crear un nivel entero.