Un juego de artillería

Rodrigo Pizarro Lozano

PID_00236760

UOC
Universitat Oberta de Catalunya

Índice

Introducción

Este módulo está dividido en cinco apartados que introducen distintos conceptos de programación y desarrollo de proyectos en Unity. El objetivo es simplemente presentar cada concepto como punto de partida para empezar a trabajar con él, pero no dar una visión a fondo. Para ver con más detalle cada uno de ellos será necesario acudir a la documentación oficial de Unity.

La primera parte trata fundamentalmente de herramientas de programación destinadas a facilitarnos la ordenación de las secuencias de eventos, tareas repetitivas, y de cómo programar de una manera eficiente todo aquello que es demasiado pesado para ejecutar en un solo *frame* en nuestro juego. Es importante recordar que el código de un *frame* se debe ejecutar lo más rápido posible.

La segunda parte es la introducción a los sistemas de partículas, que son herramientas muy utilizadas en una gran cantidad de juegos para añadir efectos visuales con mucha calidad.

A continuación, introducimos cómo trabajar con los sistemas de animaciones. Unity controla las animaciones que debe ejecutar cada elemento en base a un grafo de estados. Para ello, nos ofrece herramientas muy flexibles para que el desarrollador pueda programar casos complejos, con multitud de animaciones.

Seguidamente veremos dos herramientas muy utilizadas en el desarrollo de juegos, una para poder seleccionar grupos similares de objetos y otra para indicarle a Unity que se comporte de manera distinta con un cierto tipo de objetos.

En el último apartado desarrollaremos un proyecto de juego 2D que usará las herramientas explicadas. Partiremos de un proyecto ya existente y lo modificaremos completamente para recrear un juego de estrategia de artillería, al estilo de *Scorched Earth* o *Worms*.

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. Subprocesos

Unity tiene definidos varios eventos para poder especificar el comportamiento por cada *frame*, pero ¿y si queremos ejecutar una función que dure varios *frames*? También se puede dar el caso de que queramos ejecutar una tarea periódica, pero no a cada *frame* sino a un ritmo mucho menor. Para ello, tenemos las funcionalidades de «Invoke» y *Coroutine*.

1.1. «Invoke»

El método «Invoke» nos permite programar la llamada a otro método en el tiempo que queramos de una manera muy cómoda. La mejor manera de ver su funcionamiento es directamente mediante algunos ejemplos.

Imaginemos que queremos iluminar una escena encendiendo una luz dos segundos después de detectar un clic o un dedo en la pantalla. Este *script* nos haría el trabajo:

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


3.    public  class  ExampleClass:  MonoBehaviour  {
4.        public  Light  luz;
5.
6.        void  TurnOnLight()  {
7.            luz.enabled = true;
8.        }
9.
10.       void  Update()  {
11.           if (!luz.enabled && (Input.touch > 0 || Input.GetMouseButtonDow
n(0)))
12.               Invoke("TurnOnLight ",  2);
13.       }
14.   }
```

«GetMouseButtonDown»

Indica si se ha pulsado un botón del ratón en ese *frame*.

En este *script* vemos que el usuario podrá definir en el editor de Unity qué luz es la que se enciende después de detectar la interacción. Para este caso concreto, podríamos probarlo añadiendo este *script* a cualquier *GameObject* de una escena, y arrastrando cualquier *GameObject* que contenga un componente de tipo «Light» al campo `luz` de este *script*. Para poder notar el efecto, la luz tendría que estar apagada («disabled» en el editor).

Ahora veamos cómo podríamos hacer para simular una luz que no funciona muy bien, típicamente un fluorescente, durante un cierto tiempo. Veremos que podemos encadenar llamadas a «Invoke»:

```
using UnityEngine;
using System.Collections;

public class ExampleClass: MonoBehaviour {
    public float segundosFlicker;
    public Light luz;
    public float ultimoTime;

    void Start() {
        FlickerLight();
    }

    void FlickerLight() {
        luz.enabled = !luz.enabled;
        segundosFlicker -= (Time.time - ultimoTime);
        ultimoTime = Time.time;
        if (segundosFlicker > 0)
          Invoke("FlickerLight",
             Random.Range(0.01f, 0.5f));
    }
}
```

«Time.time»

Mide el tiempo desde que se inició el juego.

En este *script* dejamos que el usuario defina el lapso de tiempo en el que la luz simulará que no funciona muy bien. Esta simulación se basa en encender y apagar intermitentemente la luz a intervalos muy cortos. Esto lo hacemos dentro del método «FlickerLight», llamándose a sí mismo con «Invoke» en un tiempo aleatorio.

Finalmente, para hacer invocaciones periódicas usaremos el método «InvokeRepeating». Imaginemos que tenemos que encender o apagar una luz cada cierto intervalo de tiempo. Lo podemos hacer de la siguiente manera:

```
using UnityEngine;
using System.Collections;

public class ExampleClass: MonoBehaviour {

    public Light luz;
    public float intervalo;

    void Start() {
        InvokeRepeating("ToggleLight ",
                        0, intervalo);
    }

    void ToggleLight() {
        luz.enabled = !luz.enabled;
    }
}
```

Este *script* hará exactamente lo que pedimos, sin condición de final. Si quisiéramos parar este proceso, podemos usar el método «CancelInvoke», que sin parámetros cancela todos los «Invokes» asociados a una, o bien llamar a «CancelInvoke» con el nombre del método que queremos dejar de invocar como parámetro.

Nota

En un *script* podemos hacer tantas llamadas a métodos «Invoke» o «InvokeRepeating» como queramos.

1.2. *Coroutines*

Una *Coroutine* es un método que no tiene por qué ejecutarse entero dentro de un mismo *frame*. Su ejecución se va repartiendo escalonadamente a lo largo de diferentes *frames*, de modo que se ejecutan unas pocas líneas en cada uno. Este mecanismo es obligado, por ejemplo, en fragmentos de código pesado, en los que las tareas que hay que llevar a cabo no se van a poder resolver en el poco tiempo que dura un *frame*. Si lo intentáramos, el juego se ralentizaría.

Nota

Un uso muy típico es para cálculos complejos en segundo plano, o bien para inteligencia artificial (IA).

Una *Coroutine* tiene un `IEnumerator` como parámetro de retorno, y no tiene parámetros. La llamamos con el siguiente método:

- `StartCoroutine("NombreDeLaCoroutine");`

Su código de se ejecutará hasta encontrar alguna de las siguientes instrucciones:

- `yield return null`
- `yield return new WaitForSeconds(numSegundos)`

«yield»

`yield` cede el paso al *frame* actual.

En el primer caso, la ejecución se interrumpe para el *frame* actual, pero volverá a ponerse en marcha cuando empiece el siguiente. En el segundo caso, la ejecución queda en pausa hasta pasados los segundos indicados. En ambos casos, el código sigue ejecutándose justo a partir de donde se hizo `yield`.

Imaginemos que queremos apagar gradualmente una luz, en 100 *frames*. Lo podríamos hacer con el siguiente *script*, en el que «DimLight» está ejecutando una vuelta al bucle por cada *frame*:

```
1.    using     UnityEngine;
2.    using     System.Collections;
3.
4.    public class ExampleClass: MonoBehaviour     {
5.
6.        void     Start()     {
7.            StartCoroutine("DimLight");
8.        }
9.
10.       IEnumerator  DimLight()     {
11.           for (int i = 0; i < 100; i++)
12.           {
13.              luz.intensity  -=  luz.intensity / 100 f;
14.              return yield null;
15.           }
16.       }
17.   }
```

En general, podemos escribir cualquier código dentro de una *Coroutine*, y se ejecutará hasta que llegue una instrucción `return yield`. Vamos a cambiar el ejemplo anterior para que la luz se apague en 100 segundos:

```
using     UnityEngine;
using     System.Collections;

public class ExampleClass: MonoBehaviour     {

    void     Start()     {
        StartCoroutine("DimLight");
    }

    IEnumerator  DimLight()     {
        for (int i = 0; i < 100; i++)
        {
           luz.intensity  -=  luz.intensity / 100 f;
           return yield new WaitForSeconds(1f);
        }
    }
}
```

2. Sistemas de partículas

Un sistema de partículas sirve para acentuar detalles en un juego de una manera muy eficiente. Por ejemplo, la lluvia, el humo o los trozos de algún objeto roto se suelen simular con sistemas de partículas, porque suelen ser sencillos de configurar y normalmente tienen un impacto menor que otras alternativas. Como en muchos otros casos, a pesar de que este apartado esté enfocado a Unity, realmente cualquier motor de juego incluye sistemas similares, aunque los nombres o los parámetros cambien. Por otro lado, Unity llevo a cabo un gran cambio en cuanto a los sistemas de partículas, con la introducción de lo que llamaron Shuriken Particle System. Aún hoy soportan los sistemas antiguos, pero se recomienda siempre trabajar con Shuriken. Este sistema lo podemos encontrar en el menú «Component > Effects > Particle system».

Un **sistema de partículas** es un conjunto de imágenes simples con una geometría muy simple, llamadas partículas, que mostradas en grandes cantidades producen el efecto de volumen. Cada partícula tiene un cierto «Lifetime» ('tiempo de vida'), definido por el usuario dentro del editor de Unity. Durante este tiempo, puede sufrir transformaciones de distintos tipos: cambios de color, cambios de textura, cambios de tamaño, etc. Una vez acabado el «Lifetime», la partícula se destruye y se elimina del sistema.

Otro parámetro de los sistemas de partículas en Unity es el «Emission Rate», con el que indicamos cuántas partículas se generan por segundo. Una partícula se genera dentro de un volumen, que podemos definir y que puede tener forma de esfera, hemisferio, cono, cubo, o bien una forma arbitraria. Con estos dos parámetros, «Lifetime»y «Emission Rate», podemos controlar cuándo se llega a un estado estable, y cuántas partículas hay cuando el sistema es estable.

También podemos definir distintos parámetros que afectan a cómo se transforman las partículas dentro de su «Lifetime». Por ejemplo, les podemos dar una inercia con el parámetro «Velocity Vector», o bien determinar si las partículas van a ser afectadas por la gravedad o las «Wind zones» ('zonas de viento'). El abanico de posibilidades es tan amplio que simplemente nos referiremos a la abundancia de recursos *online* para aprender más y poder probar todo lo necesario. En la Asset Store de Unity podemos encontrar una gran cantidad de paquetes gratuitos de sistemas de partículas en el apartado «Particle Systems».

En lo que respecta al *scripting*, prácticamente todos los parámetros se pueden modificar vía código. Probablemente, el caso de uso más común sea iniciar y parar un sistema de partículas, ya que en muchos casos los usaremos de manera puntual, para simular explosiones, por ejemplo. Veamos el siguiente *script*:

```
    using UnityEngine;
    public class ExampleClass: MonoBehaviour {
        pubic void Explode() {
            ParticleSystem exp = GetComponent<ParticleSystem>();
            exp.Play();
            Destroy(gameObject, exp.duration);
        }
    }
```

Podemos ver en este *script* como el método «Explode» activa un «ParticleSystem» y lo destruye cuando ya ha terminado. De esta manera, si tenemos un sistema de partículas que simula una explosión, le podemos añadir este *script*, y cuando detectemos una colisión, hacer la llamada al método «Explode». Vamos a ver cómo hacerlo:

```
using  UnityEngine;
public  class  CollisionDetectorClass:  MonoBehaviour  {

    public GameObject particleSystem;
    void  OnCollisionEnter2D(Collider2D c)  {
        GameObject particleGameObject = (GameObject) Instantiate(particleSyst
em, transform.position, transform.rotation);
        particleGameObject.GetComponent<ExampleClass>().Explode();
    }
}
```

Este *script* lo podemos añadir a nuestro proyectil. A este le tendremos que añadir también un componente del tipo «Collider2D» y un «Rigidbody2D». De esta manera, en cuento nuestro proyectil colisione con otro objeto, automáticamente tendremos una explosión.

3. Animaciones

Hay varias maneras de trabajar con animaciones dentro de Unity. Esto se debe a que en versiones anteriores a la 4 el motor trabajaba con unas herramientas muy distintas. A partir de la versión 4 integraron un motor de animación totalmente distinto, el llamado Mecanim. El sistema antiguo se basa en curvas que modifican la posición y la orientación de los objetos. El problema es que con este sistema puede ser muy complicado volver a usar las animaciones en objetos distintos. Con este sistema, si tenemos, por ejemplo, varios personajes, tendríamos que tener la misma animación específica para cada personaje.

Mecanim, en cambio, se basa en un sistema interno de *retargeting*, que significa transformar una animación existente para adaptarla automáticamente a los objetos que la quieran usar. El ejemplo más claro es el anterior, si tenemos una animación para un personaje, la podremos volver a usar (hacer *retarget*) para todos los personajes que deseemos. Esto funciona bastante bien incluso aunque haya grandes diferencias; por ejemplo, si tenemos un personaje que es humano, podemos hacer *retarget* de sus animaciones a un personaje fantástico o a un animal.

A continuación veremos cómo trabajar con los dos sistemas.

3.1. Pestaña Animation

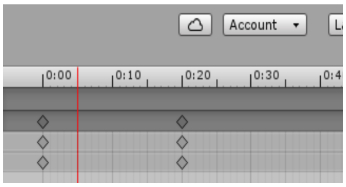
Pestaña Animation

Esta pestaña sirve para editar animaciones existentes o para crear nuevas.

Esta pestaña corresponde al sistema antiguo y permite crear y editar curvas de animación. Se basa en *keyframes* ('*frames* clave'), que son *frames* en los que tenemos información, mientras que en el resto de *frames* los datos se interpolan. Vamos a ver en detalle cómo funciona. Primero vamos al menú «Window > Animation» y vemos que se nos abre una nueva pestaña llamada Animation. Luego seleccionamos un objeto de la escena. En este caso, hemos seleccionado el objeto llamado «Main Camera» de una escena vacía. En este momento, podemos ver un mensaje que dice: «To begin animating Main Camera, create an animator and an Animation Clip».

Botones de la pestaña Animation.

Esto se debe a que las curvas de animación se graban en un archivo de formato propio con extensión .anim. De hecho, al darle al botón «Create», lo primero que tendremos que hacer es grabar un archivo .anim, que es donde se guardará todo lo que editemos. También veremos que la interfaz de Unity cambia ligeramente, en concreto los botones de «Play», «Pause» y de siguiente *frame* se vuelven de color rojo.

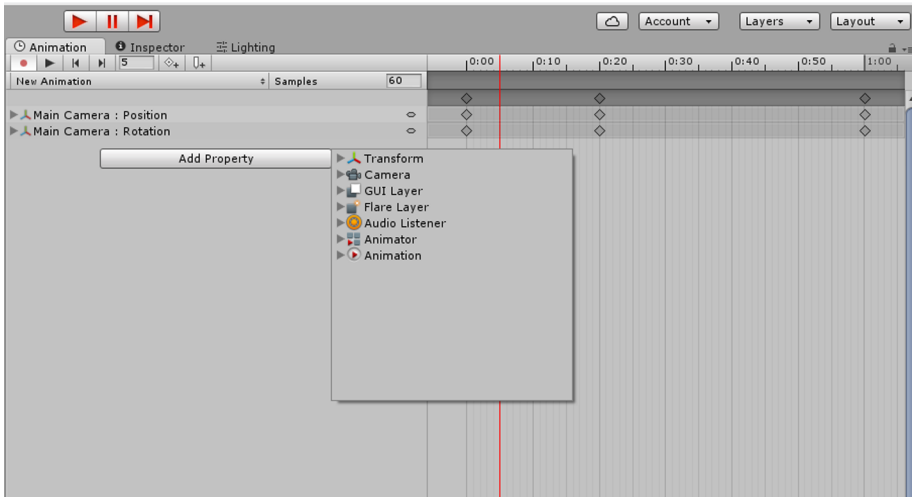


Barra del tiempo de la pestaña Animation en rojo.

En este modo, todo lo que hagamos con los objetos de la escena se puede grabar para luego reproducir. Por ejemplo, vamos a grabar unos movimientos de la cámara. En la parte derecha de la pestaña Animation tenemos una barra de tiempo, donde podemos ver los *keyframes* y el *frame* actual, representado por una línea roja vertical (ver la figura de la derecha). En la parte superior izquierda tenemos varios botones. Los dos primeros (empezando por la izquierda) sirven para grabar y reproducir lo que se ha grabado. Los dos siguientes son para tener un control *frame* a *frame*, mientras que la siguiente opción es un campo donde podemos especificar el *frame* concreto con el que queremos trabajar. El siguiente sirve para crear un *keyframe*, y el último sirve para crear un evento, algo que veremos más adelante para qué sirve.

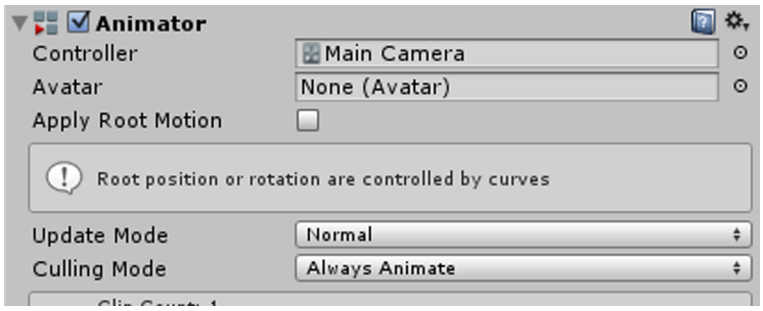
Podemos querer añadir *keyframes* para mejorar el detalle de una animación, o para añadir un movimiento extra entre dos instantes. Para añadir *keyframes* tenemos un botón específico que está a la derecha del todo. Sin embargo, es más práctico e intuitivo simplemente cambiar el *frame* actual (moviendo la barra roja o bien seleccionando el número de *frame*) y realizar algún cambio.

Unity solo guarda en el archivo de animación aquello que le hemos indicado que debe guardar. En este caso, podemos especificar que queremos que grabe posición y rotación, haciendo clic en el botón «Add Property > Transform > Position» y «Add Property > Transform > Rotation».



De todas maneras, Unity es capaz de detectar si estamos haciendo cambios en según qué propiedades. De este modo, si en la escena movemos la cámara de posición y rotación, se creará un *keyframe* en el *frame* que nos interesa, y la animación contendrá estos cambios. Si ahora desactivamos el botón de grabar y apretamos «Play», veremos que este cambio se ha guardado y que se reproduce la animación.

El motivo por el cual se reproduce la animación es porque en el *GameObject* se ha añadido automáticamente un componente «Animator» al hacer clic en el botón «Create», al principio del proceso. Explicaremos cómo funciona este componente en la sección sobre Mecanim. En la siguiente figura vemos el componente que se ha añadido a la «Main Camera».



3.1.1. Eventos

Hay muchas situaciones en las que nos interesa enlazar acciones que ocurren en una animación con cambios de comportamiento. Por ejemplo, si tenemos una animación de una mano cogiendo un objeto, podemos crear un *script* que emparente el objeto a la mano una vez cogido, y así ya tener la posición y orientación correcta del objeto hasta que cambie la situación. Para enlazar la animación con el *script* podemos usar eventos.

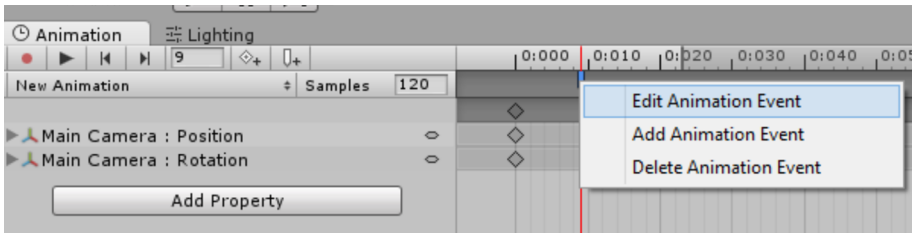
Antes de añadir un evento, tenemos que definir un método que será llamado cuando ocurra el evento. Por ello, vamos a crear un *script* como el siguiente:

```
1.    using UnityEngine;
2.    using System.Collections;
3.    public class TestEvent: MonoBehaviour {
4.        public void MetodoEvento() {
5.            Debug.Log("Evento ha llamado a este metodo");
6.        }
7.    }
```

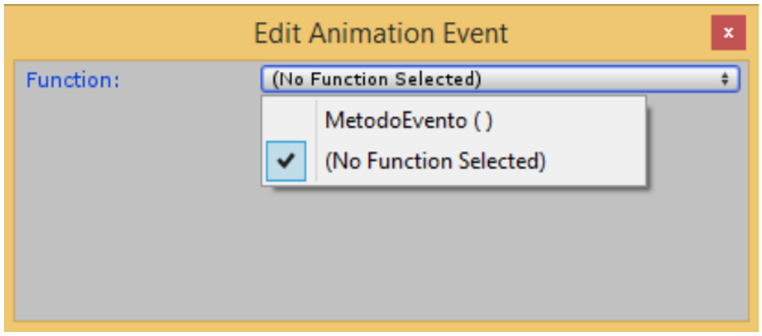
Nota

En una animación se puede unir un evento a una función definida en un *script* siempre que añadamos el *script* al objeto que contiene la animación.

Ahora este *script* lo añadimos al *GameObject* «Main Camera» y configuramos el evento. Un evento se puede crear con el botón «Add Event», en la barra superior de la pestaña Animation. Una vez creado en el *frame* que nos interese, podemos hacer clic derecho en el evento creado y seleccionar «Edit Animation Event».



Esto nos abre una pequeña ventana en la que podemos seleccionar qué método se llama cuando la animación llega al *frame* que hemos escogido, y se dispara el evento.



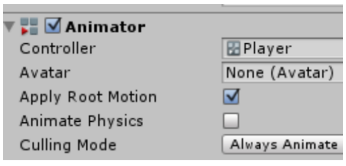
Podemos probarlo con la escena que teníamos anteriormente activando simplemente el «Play». Si todo es correcto, en la pestaña Console veremos que aparecen regularmente mensajes que dicen «Evento ha llamado a este metodo».

3.2. Mecanim

Este es el sistema de control de animaciones por el que ha apostado Unity. Se basa en una máquina de estados (también conocida como grafo de animaciones) en la que hay variables, estados, StateMachineBehaviours y transiciones entre estados. En los estados podemos definir un clip de animación o bien un *blend tree*, que es una manera de mezclar varias animaciones de un mismo grupo según unos parámetros. Las transiciones realmente lo que definen es cómo pasar de un estado a otro, es decir, qué condiciones se tienen que cumplir y cómo debe ser el paso de un estado a otro. Los StateMachineBehaviours son *scripts* que podemos añadir a los estados, para programar comportamiento de manera sencilla según el estado en el que estemos. Finalmente, las variables son un conjunto de valores que se pueden asignar para modificar el estado actual de la máquina de estados. Vamos a ver en más detalle cómo funciona este sistema.

3.2.1. Grafo de animaciones

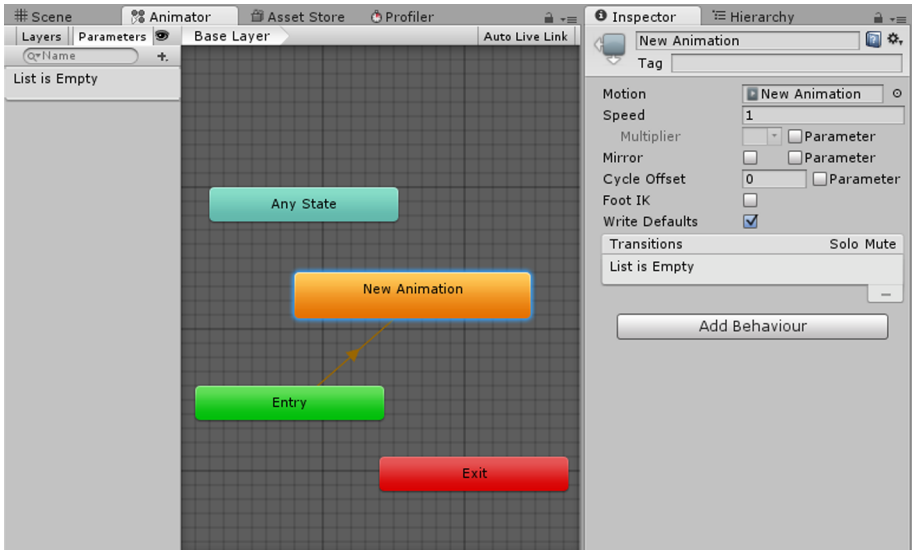
Seguimos con el *GameObject* «Main Camera» del ejemplo de la sección anterior. Podemos observar que se ha añadido automáticamente un componente «Animator». Este componente le indica a Unity que el *GameObject* está animado y que la definición de las animaciones está en el valor de parámetro «Controller», que es un *asset* de tipo «Animator Controller» (en este caso se ha creado automáticamente con el nombre «Main Camera»). El «Animator Controller» es el grafo de animaciones, y podemos crear cuantos queramos mediante «clic derecho > Create > Animator Controller» o bien en el menú «Assets > Create



El componente «Animator» con su «Controller» «Player».

> Animator Controller». Vamos a ver cómo es el grafo de animaciones en este momento. Para ello, podemos ir a la pestaña Project y hacer doble clic en el *asset* «Main Camera», con lo que se nos abrirá la pestaña Animator.

En la siguiente figura vemos que la pestaña Animator tiene dos partes, la izquierda y la derecha. En la parte izquierda vemos que podemos editar los *Layers* y los parámetros. Ya hemos visto qué son los parámetros anteriormente, y los *Layers* sirven para tener máquinas de estado complejas que controlen de manera distinta diversas partes de un objeto. El ejemplo más claro es en humanos, que somos capaces de hacer acciones muy diversas con los brazos y las manos mientras caminamos, corremos y saltamos. Como veremos más adelante, si tuviéramos que diseñar una máquina de estados capaz de controlar la combinación de todos los estados posibles se convertiría enseguida en una tarea muy compleja. Con los *Layers*, sin embargo, podemos diseñar una máquina de estados para los brazos y otra para las piernas en distintos *Layers*.



En la parte derecha, vemos el grafo propiamente dicho. En esta parte podemos añadir estados y transiciones. Para tener una vista mas completa, dado que el número de opciones es bastante grande, recomendamos seguir un tutorial *online* de la documentación de Unity. Seguimos con un ejemplo típico.

Web recomendada

Podemos encontrar un tutorial completo en:
https://unity3d.com/learn/tutorials/topics/animation/animate-anything-mecanim.

Puede que nos sorprenda el hecho de que en esta parte haya varios elementos, aunque solo hayamos creado una animación. Esto se debe a que en cualquier grafo hay tres elementos extra que nos pueden servir en multitud de ocasiones. Dos de ellos son los estados de «Entry» y «Exit», que nos sirven para poder definir cuál es el estado inicial y si queremos que haya un estado final. El estado que esté en color naranja y que tenga una transición desde «Entry» será el estado inicial. Si ningún estado tiene una transición hacia «Exit», siempre habrá algún estado activo en el grafo. Esto es particularmente importante pa-

Nota

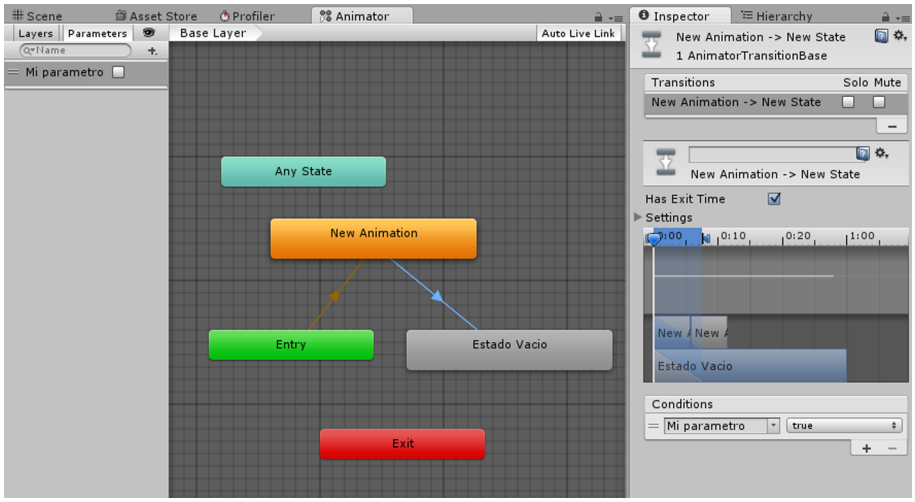
Cada subestado dentro de una máquina de estados se considera una máquina de estado independiente y completa, por lo que mediante el uso de estos nodos «Entry» y «Exit» se puede controlar el flujo de una máquina de estado de nivel superior en sus máquinas subestatales de forma más elegante.

ra las submáquinas de estados. Podemos definir estados que no sean un solo estado, sino una máquina de estados completa, por lo que tendremos que definir recorridos que tengan una entrada y una salida.

El tercer estado extra, llamado «Any State», es realmente un comodín que representa todos los estados del grafo. Por ejemplo, si tenemos una máquina de estados muy compleja que gestiona múltiples animaciones de los brazos pero queremos que con un botón el usuario pueda inmediatamente hacer una acción en concreto, deberíamos crear un estado que gestione esa acción y tener una transición hacia esa acción por cada estado. Sin embargo, podemos hacer lo mismo usando el comodín «Any State» con una sola transición.

También debemos fijarnos en la pestaña Inspector, que nos proporciona información y nos deja editar muchos parámetros de los estados y de las transiciones. Por ejemplo, vamos a ver cómo activar una transición mediante un parámetro en concreto. Para ello, crearemos un estado vacío mediante clic derecho en la parte derecha de la pestaña Animator, seleccionando la opción «Create State > Empty». Con esta acción se nos ha creado un estado que se llama «New State». Podemos cambiarle el nombre en la pestaña Inspector, en el campo superior. Nosotros le podremos el nombre de «Estado Vacio». Ahora crearemos una transición entre el primer estado («New Animation») y el recién creado. Esto lo hacemos con un clic derecho encima del estado «New Animation», seleccionando la opción «Make Transition» y haciendo clic izquierdo en el estado «Estado Vacio». Por otro lado, crearemos una variable en la parte izquierda de la pestaña Animator, en el botón con el símbolo «+». Se nos despliega un pequeño menú con cuatro opciones: «Int», «Float», «Bool» y «Trigger». Esto es el tipo del parámetro, y para este ejemplo podemos usar el tipo «Bool». Lo podemos renombrar haciendo clic en su nombre, y en este caso le hemos llamado «Mi parametro». Una vez tengamos todos estos elementos podemos hacer clic en la transición creada anteriormente, y en la pestaña Inspector podremos ver varias opciones.

En la parte inferior tenemos la sección «Conditions». Configurando esta sección podremos determinar cuándo saltamos de un estado a otro y, por ende, cuál es la animación que se está ejecutando. Si no tenemos ninguna condición, la transición se activará siempre. Si tenemos alguna condición, la transición se activará cuando todas las condiciones sean verdaderas. Si queremos activar una transición cuando sea cierta una condición u otra, podremos crear dos transiciones distintas y asignar las condiciones de cada una. Para asignar una condición la añadimos con el botón que tiene el símbolo «+», y nos aparecerá en la lista de la sección«Conditions».



Para probar este grafo podemos simplemente apretar «Play». Lo que debemos observar es que el objeto «Main Camera» se mueve con la animación que hemos definido antes, y que cuando activamos el parámetro «Mi parametro» (haciendo clic en el recuadro de al lado de su nombre) se activa la transición y el estado actual pasa a ser «Estado Vacio».

3.3. Interacción mediante código

Los parámetros de un grafo de animaciones se pueden cambiar desde el código para alterar el estado actual. También podemos usar StateMachineBehaviours para interactuar con el grafo. Vamos a ver cómo hacerlo.

Nota

«Estado Vacio» es un estado que no tiene una animación asignada. No es obligatorio asignarle una animación a un estado. Esto nos puede ser útil en algunos escenarios. Por ejemplo, si queremos usar un comportamiento definido en un StateMachineBehaviour, o si queremos que el objeto se mantenga quieto. Para asignarle una animación podemos hacerlo en el campo «Motion» de cualquier estado del grafo.

Primero trabajaremos con el componente «Animator» y accederemos a su funcionalidad. Vamos a crear un *script* capaz de ello:

```
1.    using UnityEngine;
2.    using System.Collections;
3.    public class TestInteraction: MonoBehaviour {
4.        private Animator animator;
5.        public void Start() {
6.            animator = GetComponent < Animator > ();
7.            Invoke("ActivaTransicion", 3);
8.        }
9.        private void ActivaTransicion() {
10.           animator.SetBool("Mi parametro", true);
11.       }
12.   }
```

Vemos en este *script* que usamos el método «SetBool» de «Animator», y activamos el parámetro «Mi parámetro» vía *scripting*. Esto tiene el mismo efecto que apretarlo manualmente. Podemos probarlo añadiendo este *script* al *GameObject* «Main Camera» y veremos que tiene el mismo efecto que en la prueba anterior.

«Animator» tiene funciones para asignar valores a todos los tipos de variables que podemos añadir a un grafo. De la misma manera, podemos obtener los valores usando las funciones «Get» de «Animator». Mediante estos controles podemos manejar prácticamente todo lo necesario en una máquina de estados.

También existen funciones para crear, editar y borrar tanto estados como funciones, pero es más complejo y queda fuera del alcance de este curso.

Web recomendada

Podemos ver la API entera de «Animator» en:
https://unity3d.com/ScriptReference/Animator.html.

Finalmente, otra manera de interactuar es mediante StateMachineBehaviours. Estos son *scripts* parecidos a los MonoBehaviours, pero que se añaden a estados de un grafo de animaciones. Aunque los eventos que podemos usar tienen otros nombres y se capturan de otra manera, son bastante equivalentes a los básicos de un MonoBehaviour. Por ejemplo, tenemos la función `OnStateEnter`, que es equivalente a «Start» en un MonoBehaviour, y la función `OnStateUpdate`, que es equivalente a «Update». Vamos a hacer un ejemplo parecido al anterior. Para ello, vamos al grafo de animaciones, seleccionamos el estado «New Animation» y le damos al botón «Add Behaviour». Nosotros le hemos llamado «PruebaSMB».

```
1.   using UnityEngine;
2.   using System.Collections;
3.   public class PruebaSMB: StateMachineBehaviour {
4.
5.       override public void OnStateEnter(Animator animator, AnimatorState
Info stateInfo, int layerIndex) {
6.               animator.SetBool("Mi parametro", true);
7.           }
8.
9.       override public void OnStateExit(Animator animator, AnimatorStateI
nfo stateInfo, int layerIndex) {
10.              Debug.Log("Transicion activada, cambiamos de estado");
11.          }
12.  }
```

Este *script* activa la transición nada más entrar y escribe «Transicion activada, cambiamos de estado» al salir del estado. El *script* anterior también cambia el valor del parámetro «Mi parámetro», pero lo hace tres segundos más tarde, debido al uso de «Invoke». De esta manera, cuando el método `ActivaTransicion` de la clase `TestInteraction` se llama, ya estamos en el estado «Estado Vacio», que es un estado donde no hay transiciones que usen este parámetro como condición. Por otro lado, el modificar un valor en un estado que no lo use no tiene consecuencias ni causa ningún error. Podemos ver la ejecución y cómo se van activando las transiciones seleccionando en la pestaña Hierarchy el *GameObject* que queremos. De esta manera, en la pestaña Animator veremos el estado actual del grafo para el objeto seleccionado, y veremos también cómo evoluciona.

4. *Layers* y *Tags*

Es bastante común en los juegos tener un cierto número de elementos de un tipo determinado. Por ejemplo, es típico tener varios enemigos y aliados, que pueden ser de tipos distintos. Muy frecuentemente querremos encontrar todos los elementos de un cierto tipo de manera práctica y eficiente, o bien aplicarles alguna operación o lo contrario: discriminarlos de alguna manera para evitar aplicar alguna operación a un cierto tipo de elementos. Para todo ello tenemos los *Tags* y los *Layers*.

4.1. *Tags*

Un *Tag* es una etiqueta asociada a un *GameObject*. Aunque Unity por defecto nos crea algunos de los *Tags* más usuales, podemos definir nuestros propios *Tags* en el menú «Edit > Project Settings > Tags and Layers». En la pestaña Inspector nos aparecen tres desplegables: «Tags», «Sorting Layers» y «Layers».

En el desplegable «Tags» podemos crear, editar y eliminar todos los *Tags* de un proyecto. Vamos a añadir un *Tag* nuevo apretando el botón de abajo a la derecha con el símbolo «+», y le vamos a llamar «my tag». Ahora vamos a añadir el siguiente *script* a cualquier *GameObject* de una escena cualquiera:

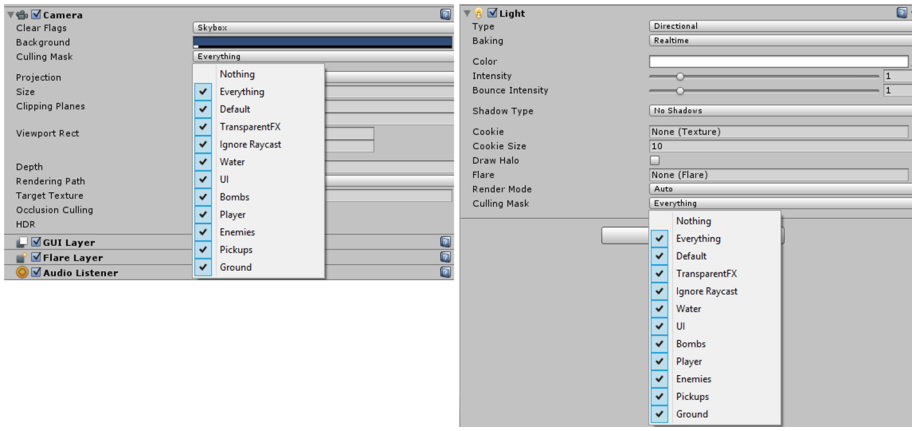
```
1. using UnityEngine;
2. using System.Collections;
3. public class FindTag: MonoBehaviour {
4.
5.     void Start() {
6.         foreach(GameObject go in GameObject.FindGameObjectsWithTag("my tag
"))
7.             Debug.Log(go.name);
8.     }
9. }
```

Este *script* recorre todos los *GameObjects* que tengan el *Tag* «my tag» asignado, y muestra en la pestaña Console los nombres de esos *GameObjects*.

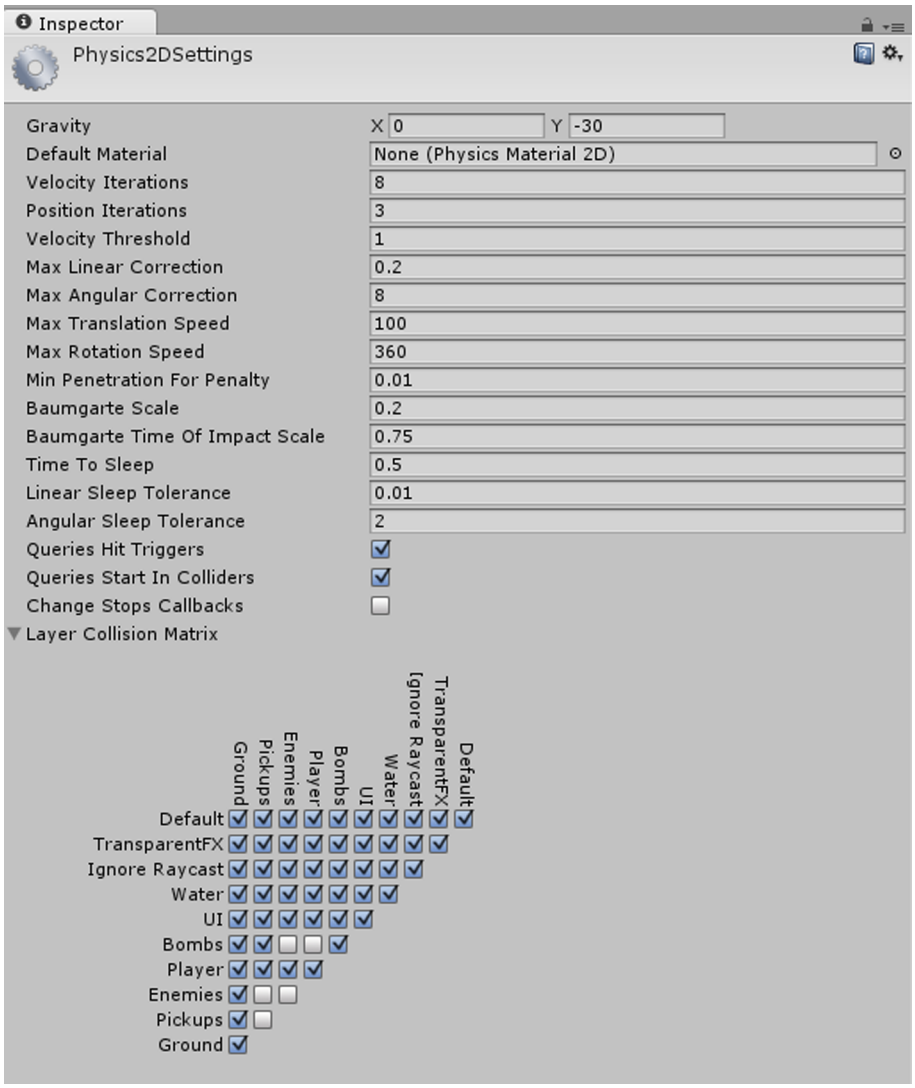
4.2. *Layers*

Los *Layers* funcionan de una manera similar a los *Tags*, aunque su utilidad es distinta. En general, están orientados a la optimización. Así como los *Tags* solo nos aportan utilidad a nivel de nuestro propio código, los *Layers* sirven para decirle a Unity que haga ciertas cosas o no en función del *Layer* de un *GameObject*. Por ejemplo, podemos configurar un componente «Camera» para que renderice solo los *Layers* que nos interesen (por ejemplo, para optimizar en dispositivos más lentos, podemos ahorrarnos mostrar detalles decorativos),

o podemos indicar a un componente «Light» que ilumine solo ciertos *Layers* (siguiendo el mismo ejemplo anterior, nos podemos ahorrar la iluminación de muchos objetos). En la figura podemos ver cómo configurar estos *components*.



Otro de sus usos más comunes es ahorrarse cálculos de física, definiendo las colisiones posibles en la matriz de colisiones de física (editable en el menú «Edit > Project Settings > Physics 2D», veáse la figura).



5. Proyecto: Un juego de artillería

En esta sección veremos cómo desarrollar un juego 2D basado en un clásico del género, *Artillery Strategy*, que tuvo un gran éxito hace unos cuantos años. De hecho, tuvo tanto éxito que se han hecho varias versiones, algunas en 3D. Es un juego por turnos donde el jugador tiene que atacar a los soldados del oponente usando uno de sus soldados hasta acabar con todos ellos. Buscaremos simplicidad en este proyecto, así que obviaremos alguno de los aspectos del juego original.

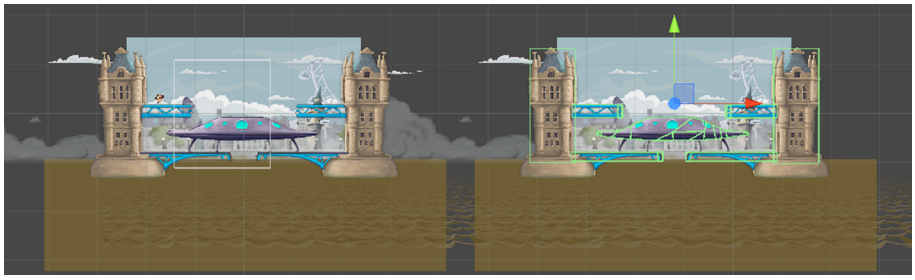
Antes de empezar, bajaremos un paquete de *assets* que nos servirán para empezar nuestro proyecto. Se llama *2D Platformer* y lo podemos encontrar en la Asset Store de Unity. De hecho, se trata de un *pack* creado por la propia Unity para que los desarrolladores puedan usarlo y probar lo que quieran.

En este paquete hay una escena que tiene muchos de los elementos que buscamos. Tiene un escenario creado con «Colliders» y un héroe principal con los *scripts* configurados; se encuentra en «Assets > Scenes > Level». Vamos a coger esta escena como base y a modificarla para hacer nuestro juego.

5.1. Modificando «Level»

Primero vamos a hacer una copia de la escena, para no perder información si cometemos algún error. Aunque no es estrictamente necesario porque siempre podemos volver a importar el *pack*, puede sernos útil y es en general una buena práctica. Nosotros hemos llamado a la copia «Nivel».

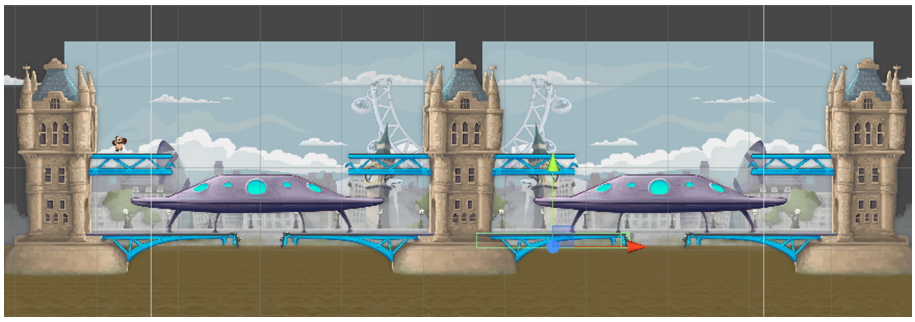
El objetivo de modificar «Nivel» es que la escena tenga dos partes simétricas, a derecha e izquierda. Vamos a empezar por duplicar el fondo y los elementos que forman el terreno. Esto lo podemos hacer copiando y pegando (o usando el atajo «Ctrl + d») los *GameObjects foregrounds* y *backgrounds*, y moviendo las copias hacia un lado.



Hay varios problemas: primero, que si los solapamos hay una torre que tapará a la otra. El segundo es que hay elementos decorativos dinámicos que se mueven de un lado a otro, que se ven mal si se ven parcialmente o si se solapan. El

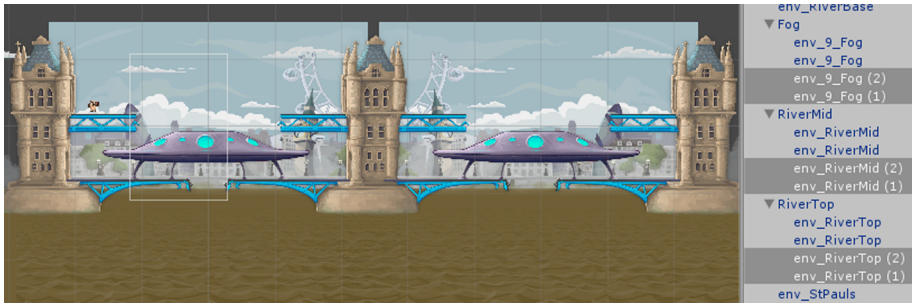
tercer problema es que hay elementos decorativos que solo aparecen en una parte, como los coches y autobuses. Por último, la cámara no capta la nueva parte. Solucionemos estos problemas.

Primero vamos a eliminar una de las torres y a encajar el fondo. También podemos borrar uno de los *sprites* marrones que hacen de agua, ya que podemos escalar el otro para cubrir todo el espacio. Por otro lado, la imagen del fondo se puede invertir escalando la X a -1 de los *GameObjects* que tienen *sprites* para el fondo, o bien en los «SpriteRenderer» seleccionar «Flip X». Si hacemos todas estas operaciones, el resultado debería ser parecido al de la figura siguiente. Hay un espacio entre las dos partes, pero en la imagen final no se aprecia porque el color de fondo coincide con el color del parámetro «Background» de la cámara.



Hay un detalle que nos da problemas, y es que la base de las torres no es simétrica, por lo que el puente izquierdo de la parte derecha parece que quede colgando. Lo podríamos solucionar con un editor de imágenes, aunque por simplicidad lo que hemos hecho es desplazar el puente ligeramente hacia la izquierda.

Ahora vamos a arreglar las animaciones. La manera más sencilla es duplicando los hijos de los *GameObjects* que controlan el movimiento. En concreto, «Fog», «RiverMid» y «RiverTop» tienen dos *sprites* cada uno, y si duplicamos los hijos (como vemos en la figura) y los encajamos justo a la derecha, tendremos una animación válida para toda la escena. Esto funciona porque la animación realmente lo que hace es mover el padre. De esta manera, los *sprites* hijos se mueven en consonancia. En la copia de los *GameObjects backgrounds* que hemos usado para la escena derecha podemos borrar estos tres objetos.



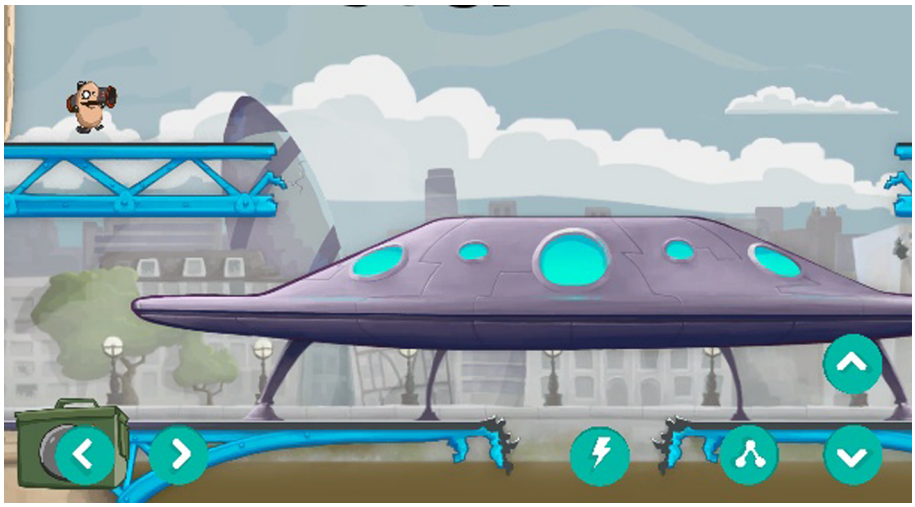
Reto 1

Arreglad los elementos decorativos que solo aparecen en la parte izquierda (el autobús, el coche y la cigüeña). Estos se generan aleatoriamente con el *script* que está en el *GameObject* «backgroundAnimation». Modificad los parámetros para que funcionen también en la parte derecha.

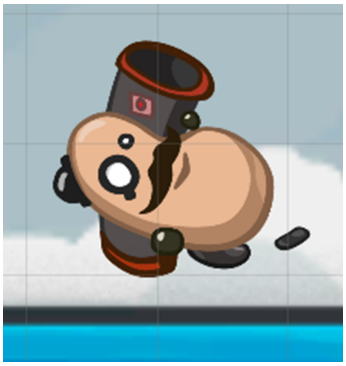
El *GameObject* «spawners» no nos interesa para el juego que queremos, por lo que lo podemos borrar. Ahora solo nos queda arreglar la cámara, pero esto forma parte de cómo funciona el juego. Empecemos a añadir interacción para poder crear luego «Gameplay».

5.2. Interacción

Tal y como está diseñada la escena, no tenemos controles para dispositivos móviles, así que vamos a añadirle algunos botones. Podemos bajar el paquete de botones llamado *ColorfulButtons* de la Asset Store. Con estos *assets* podemos situar varios botones que nos servirán para controlar a nuestro personaje, como vemos en la figura de abajo. Con los botones de la izquierda podremos mover el personaje de izquierda a derecha, con el que tiene la imagen de un relámpago podemos disparar, el siguiente botón nos servirá para saltar, y los dos de la derecha, para inclinarnos.

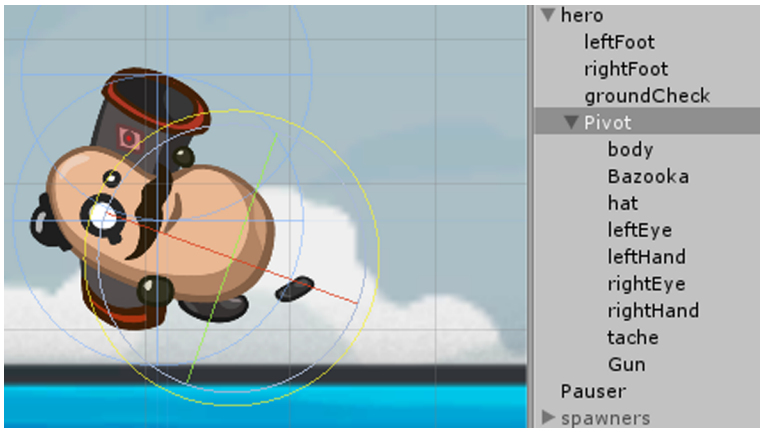


También vamos a crear una animación nueva, para poder inclinar nuestro personaje (*GameObject* llamado «hero») hacia arriba, como vemos en la figura de la derecha. Esto lo podemos hacer de varias maneras. Una forma sería con la pestaña Animation, pero sería complicado darle al usuario un buen control de la inclinación. Otra manera es animar al personaje proceduralmente, por *script*. Nuestro objetivo es que todos los elementos del personaje (ojos, bigote, etc.) se muevan en consonancia con el cuerpo, pero que los pies se queden apoyados en el suelo. Si probamos a rotar el personaje, no lo conseguiremos. Si lo cogemos desde el padre, los pies también rotarán, y además el pivote de rotación está por el centro del cuerpo. Vamos a solucionar este aspecto.



El personaje principal, inclinado hacia arriba.

Para ello, primero creamos un *GameObject* vacío y lo situamos donde nos interesa que esté el pivote de rotación (ver los ejes en la figura siguiente). Después debemos emparentarlo al personaje, y emparentar los elementos de nuestro personaje que queremos que roten a este nuevo *GameObject*, que hemos renombrado a «Pivot».



Ahora debemos asignar los botones a controles del personaje. Para ello, debemos modificar el *script* «PlayerControl». Empezamos por añadir algunas variables más al *script*, cambiando la inicialización:

```
private float tilt;
private List < KeyCode > acciones = new List < KeyCode > ();
private Transform pivot;

void Awake() { // Setting up references.
    groundCheck = transform.Find("groundCheck");
    anim = GetComponent < Animator > ();
    pivot = transform.FindChild("Pivot");
}
```

Reto 2

Añadid los métodos necesarios para poder crear eventos desde el editor de Unity. Debemos crear métodos para rotar a izquierda y derecha, y para mover el personaje a la izquierda y la derecha. Además, ya que usaremos el componente «Event Trigger», debemos crear dos versiones de cada método, el «Up» y el «Down». Es una forma habitual de gestionar acciones de botones, que se ejecutan mientras se aprietan. Hemos visto en el módulo «Un juego de plataformas» un ejemplo parecido.

Nota

Debemos usar los «KeyCode» y la lista acciones, añadiendo a la lista el «KeyCode» cuando sean eventos «Down», y quitándolos de la lista en eventos «Up».

Finalmente, hay que modificar ligeramente el método «Fixed Update»:

```
void  FixedUpdate()  {  // Cache the horizontal input.

    float  h  =  Input.GetAxis("Horizontal");
    if  (h  ==  0)  {
       if  (acciones.Contains(KeyCode.LeftArrow))  {
          h = anim.GetFloat("Speed")  -  0.1  f;
       }
       if  (acciones.Contains(KeyCode.RightArrow))  {
          h = anim.GetFloat("Speed")  +  0.1  f;
       }
    }
    if  (acciones.Contains(KeyCode.UpArrow))  {
       tilt  +=  1.0  f;
    }
    if  (acciones.Contains(KeyCode.DownArrow))  {
       tilt  -=  1.0  f;
    }
    tilt  =  Mathf.Clamp(tilt,  0,  75);
    pivot.rotation  =  Quaternion.Euler(0,  0,  tilt);
```

El método «FixedUpdate»

Todos los cálculos de la física y las actualizaciones se producen inmediatamente después de «FixedUpdate».

Ahora ya podemos añadir componentes «Event Trigger», con los eventos «Pointer Down» y «Pointer Up» de los botones izquierda, derecha, arriba y abajo, y asociarlos a los métodos adecuados de «PlayerControl».

Reto 3

Añadid un método en «PlayerControl» que haga saltar al personaje.

Finalmente, solo nos queda controlar el disparo. Podemos modificar el *script* llamado «Gun» de la siguiente manera:

```
1.    using UnityEngine;
2.    public class Gun: MonoBehaviour {
3.       public Rigidbody2D rocket; // Prefab of the rocket.
4.       public float speed = 20 f; // The speed the rocket will fire at.
5.       private Animator anim; // Reference to the Animator component.
6.       void Awake() { // Setting up the references.
7.         anim = transform.root.gameObject.GetComponent<Animator>();
8.       }
9.
10.      public void Fire() {
11.        anim.SetTrigger("Shoot");
12.        GetComponent < AudioSource > ().Play();
13.        Rigidbody2D bulletInstance = Instantiate(rocket, transform.position
, transform.rotation) as Rigidbody2D;
14.        bulletInstance.velocity = transform.right.normalized*speed;
15.      }
16.   }
```

Ahora, si signamos el evento «OnClick» del botón de ataque (el que parece un relámpago), vemos que el personaje dispara hacia donde está apuntando el cañón. Sin embargo, los proyectiles no se ven afectados por la gravedad, con lo que es imposible llegar a nuestro enemigo. Hay que aumentar el «Gravity Scale» del «Rigidbody2D» de los proyectiles. Esto lo hacemos cambiando el valor del *prefab*, que está en «Assets > Prefabs > rocket»; nosotros lo hemos dejado con un valor de 0.8.

Ahora ya podríamos atacar a nuestro enemigo, pero está demasiado lejos. Debemos añadir alguna funcionalidad para que el usuario pueda controlar la velocidad con la que salen los proyectiles. Para ello debemos modificar el *script* anterior:

```
1.    using UnityEngine;
2.    public class Gun: MonoBehaviour {
3.        public Rigidbody2D rocket; // Prefab of the rocket.
4.        private Animator anim; // Reference to the Animator component.
5.        private enum States {
6.            Down, Fire, Up
7.        }
8.        private States state = States.Up;
9.        public float speed = 0 f;
10.       void Awake() { // Setting up the references.
11.         anim = transform.root.gameObject.GetComponent<Animator>();
12.       }
13.       void Update() {
14.           switch (state) {
15.               case States.Down:
16.                   speed += Time.deltaTime * 30;
17.                   break;
18.               case States.Fire:
19.                   Fire();
20.                   state = States.Up;
21.                   break;
22.           }
23.       }
24.       public void FireUp() {
25.           state = States.Fire;
26.       }
27.       public void FireDown() {
28.           state = States.Down;
29.       }
30.       public void Fire() {
31.           anim.SetTrigger("Shoot");
32.           GetComponent < AudioSource > ().Play();
33.           Rigidbody2D bulletInstance = Instantiate(rocket, transform.posit
ion, transform.rotation) as Rigidbody2D;
34.           bulletInstance.velocity = transform.right.normalized * speed;
35.           speed = 0;
36.       }
37.   }
```

Y ahora solo queda añadir al botón de ataque un «Event Trigger», con los eventos «Pointer Down» y «Pointer Up» asignados a «FireDown» y «FireUp», respectivamente. Ya podemos atacar a nuestro enemigo, aunque con dificultad, ya que no tenemos ninguna respuesta visual que nos indique cómo de fuerte estamos disparando. Vamos a añadirla, cambiando el *script* anterior:

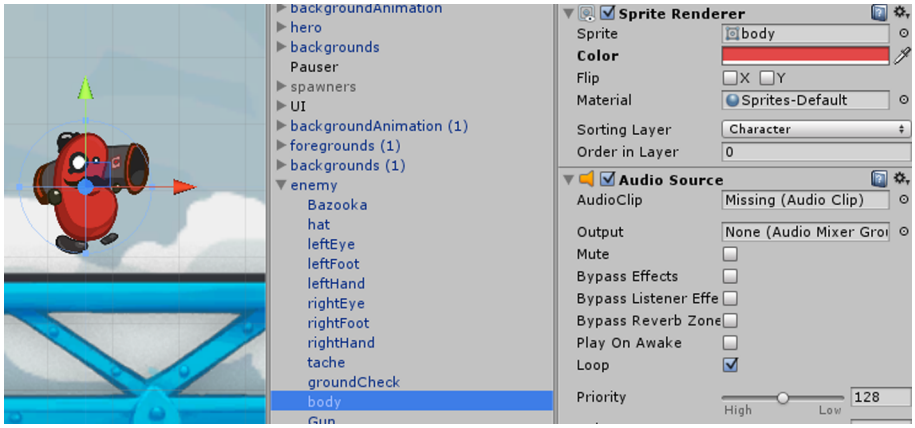
```
1.    using UnityEngine;
2.    public class Gun: MonoBehaviour {
3.        public Rigidbody2D rocket; // Prefab of the rocket.
4.        public Sprite attackBarSprite;
5.        private Animator anim; // Reference to the Animator component.
6.        private enum States {
7.            Down, Fire, Up
8.        }
9.        private States state = States.Up;
10.       private float speed = 0 f;
11.       private Transform attackBar;
12.       void Awake() {
13.
14.           anim = transform.root.gameObject.GetComponent<Animator>();
15.           GameObject attackBarObject = new GameObject("Power");
16.           attackBar = attackBarObject.transform;
17.           attackBar.SetParent(transform);
18.           attackBar.localPosition = Vector3.zero;
19.           attackBar.localRotation = Quaternion.identity;
20.           attackBar.localScale = Vector3.up * 2 + Vector3.forward;
21.           SpriteRenderer rend = attackBarObject.AddComponent < SpriteRende
rer > ();
22.           rend.sprite = attackBarSprite;
23.           rend.sortingLayerID = transform.root.GetComponentInChildren < Sp
riteRenderer > ().sortingLayerID;
24.       }
25.       void Update() {
26.           switch (state) {
27.               case States.Down:
28.                   speed += Time.deltaTime * 30;
29.                   attackBar.localScale += Vector3.right * 0.01 f;
30.                   attackBar.GetComponent < SpriteRenderer > ().color = Col
or.Lerp(Color.green, Color.red, attackBar.localScale.x);
31.                   break;
32.               case States.Fire:
33.                   Fire();
34.                   state = States.Up;
35.                   break;
36.           }
37.       }
38.       public void FireUp() {
39.           state = States.Fire;
40.       }
41.       public void FireDown() {
42.           state = States.Down;
43.       }
44.       public void Fire() {
45.           anim.SetTrigger("Shoot");
46.           GetComponent < AudioSource > ().Play();
47.           Rigidbody2D bulletInstance = Instantiate(rocket, transform.posit
ion, transform.rotation) as Rigidbody2D;
48.           bulletInstance.velocity = transform.right.normalized * speed;
49.           speed = 0;
50.           attackBar.localScale = Vector3.up * 2 + Vector3.forward;
51.       }
52.   }
```

Ahora solo tenemos que añadir un *sprite* al campo «AttackBarSprite», usando el editor. Nosotros hemos usado el *sprite* «Health», que podemos encontrar en «Assets > Sprites > _UI > Health». Si lo hemos hecho correctamente, al disparar se verá una barra similar a la de la figura de la derecha.

El personaje principal, inclinado hacia arriba.

5.3. «Gameplay»

Nos queda crear nuestro enemigo y establecer una mecánica de juego. Podemos duplicar nuestro personaje y llevarlo al otro lado, en el puente derecho de la parte derecha de nuestra escena. También lo podemos renombrar y llamarlo «enemy», y darle un tinte de un color para que el jugador pueda diferenciarlo e identificar que es el enemigo.



Ahora vamos a añadir inteligencia artificial a nuestro enemigo:

```
using UnityEngine;
public class EnemyAI : PlayerControl{
    private Transform hero;
    private Gun gun;
    private float shootAngle = 30;

    void Start()
    {
        hero = GameObject.FindGameObjectWithTag("Player").transform;
        gun = GetComponentInChildren < Gun > ();
    }

     float CalculaVelocidad(Transform target, float angle)
    {
    }

    void LateUpdate()
    {
        if (hasTurn)
        {
            if (facingRight)
               Flip();

            if (tilt != shootAngle)
            {
                if (tilt < shootAngle)
                    RotaIzquierdaDown();
                else
                    RotaDerechaDown();
            }
            else {
                gun.Fire(CalculaVelocidad (hero, shootAngle));
                RotaDerechaUp();
                RotaIzquierdaUp();
            }
        }
        else {
            shootAngle = Random.Range(25, 45);
        }
    }
}
```

Como vemos, este *script* hereda de «PlayerControl» y contiene dos métodos relevantes: «CalculaVelocidad» y «LateUpdate». «LateUpdate» usa los métodos de «PlayerControl» para inclinar a nuestro enemigo de manera que el proyectil pueda superar el obstáculo central. La razón por la que usamos «LateUpdate» es simplemente porque no queremos ocultar «Update», sino que queremos que se ejecuten los dos. Esta es una manera cómoda de hacerlo.

Reto 4

Programad el método «CalculaVelocidad» de modo que calcule la velocidad necesaria para llegar a un punto dadas las coordenadas y un ángulo. Le aplicamos también un factor de aleatoriedad para que el enemigo no sea demasiado perfecto y cometa fallos.

Como vemos, hay un parámetro nuevo, que es «hasTurn», que no está declarado en «EnemyAI». Esto es porque lo hemos declarado en «PlayerControl», ya que con esto le indicaremos a los *scripts* si un personaje se puede mover o no. Hay que recordar que este es un juego por turnos, por lo que solo un personaje se puede mover en cada turno. Hemos añadido esta variable y hemos modificado otra de «PlayerControl» de la siguiente manera:

```
1.     [HideInInspector]
2.      public  bool  hasTurn  =  false;
3.      protected float tilt;
```

De esta manera, podemos acceder al valor de *tilt* en «EnemyAI». Además, hemos añadido una comprobación en «FixedUpdate», con la que si «hasTurn» es falso, terminamos el método. Por último, hemos añadido otra función al *script* «Gun», de modo que podemos indicar la velocidad a la que queremos que se lance el proyectil. Este método se llama «Fire» y tiene como parámetro la velocidad que queremos. Esto nos es útil porque la inteligencia artificial del enemigo no usa los botones como el usuario.

```
1.  using UnityEngine;
2.  public class Gun: MonoBehaviour {
3.      public Rigidbody2D rocket; // Prefab of the rocket.
4.      public Sprite attackBarSprite;
5.      public delegate void GunFired();
6.      public event GunFired gunFired;
7.      private Animator anim; // Reference to the Animator component.
8.      private enum States {
9.          Down, Fire, Up
10.     }
11.     private States state = States.Up;
12.     private float speed = 0 f;
13.     private float targetSpeed = 0 f;
14.     private Transform attackBar;
15.     void Awake() { // Setting up the references.
16.       anim = transform.root.gameObject.GetComponent<Animator>();
17.       GameObject attackBarObject = new GameObject("Power");
18.       attackBar = attackBarObject.transform;
19.       attackBar.SetParent(transform);
20.       attackBar.localPosition = Vector3.zero;
21.       attackBar.localRotation = Quaternion.identity;
22.       attackBar.localScale = Vector3.up * 2 + Vector3.forward;
23.       SpriteRenderer rend = attackBarObject.AddComponent < SpriteRenderer
> ();
24.       rend.sprite = attackBarSprite;
25.       rend.sortingLayerID = transform.root.GetComponentInChildren < Sprite
Renderer > ().sortingLayerID;
26.     }
27.     void Update() {
28.         if (targetSpeed > 0) {
29.             state = States.Down;
30.             if (speed >= targetSpeed) state = States.Fire;
31.         }
32.         switch (state) {
33.             case States.Down:
34.                 speed += Time.deltaTime * 30;
35.                 attackBar.localScale += Vector3.right * 0.01 f;
36.                 attackBar.GetComponent < SpriteRenderer > ().color = Color
.Lerp(Color.green, Color.red, attackBar.localScale.x);
37.                 break;
38.             case States.Fire:
39.                 Fire();
40.                 state = States.Up;
41.                 break;
42.         }
43.     }
44.     public void FireUp() {
45.         state = States.Fire;
46.     }
47.     public void FireDown() {
48.         state = States.Down;
49.     }
50.     public void Fire() {
51.         anim.SetTrigger("Shoot");
52.         GetComponent < AudioSource > ().Play();
53.         Rigidbody2D bulletInstance = Instantiate(rocket, transform.positio
n, transform.rotation) as Rigidbody2D;
54.         if (transform.root.GetComponent < PlayerControl > ().facingRight)
bulletInstance.velocity = transform.right.normalized * speed;
55.         else bulletInstance.velocity = new Vector2(-
transform.right.x, transform.right.y).normalized * speed;
56.         bulletInstance.GetComponentInChildren < Rocket > ().ignoreTag = tr
ansform.root.tag;
57.         targetSpeed = speed = 0;
58.         attackBar.localScale = Vector3.up * 2 + Vector3.forward;
59.         if (gunFired != null) {
60.             gunFired();
61.         }
62.     }
63.     public void Fire(float targetSpeed) {
64.         this.targetSpeed = targetSpeed;
65.     }
```

De esta manera, podemos indicarle la fuerza con que un proyectil debe ser disparado con una sola función. En este *script* hacemos una llamada a otro *script* que también hemos modificado, que es «Rocket». En concreto, le hemos añadido una variable pública llamada «ignoreTag», que usaremos para distinguir si hay que ignorar la colisión o no.

Reto 5

Modificad el método `OnTriggerEnter2D` para que solo ejecute el código cuando no ha colisionado con un objeto que hay que ignorar. Usad el *Tag* de la colisión y comparadlo con «ignoreTag».

Nota

El uso de los *Tags* es habitual y muy eficiente, como ya hemos podido comprobar anteriormente.

Esta comprobación la hacemos para asegurarnos de que el proyectil no se detona nada más crearse, ya que tanto nuestro personaje como el enemigo tienen un «Collider». Si no la hiciésemos, nada más crearse el proyectil provocaría una explosión.

Reto 6

Modificad el mismo método del *script* «Rocket» de manera que se cree un *GameObject* que contenga un «CircleCollider2D» de radio 2,5 sin activar la opción «isTrigger», y que se autodestruya después de 0,5 segundos.

Este reto tiene dos objetivos: el primero es que cuando haya impacto el personaje se desplace, y el segundo es que podamos detectar el impacto en un *script*, con lo que podremos controlar el nivel de vida.

El *script* «PlayerHealth» se encarga de gestionar el nivel de vida, pero tenemos que cambiar una línea de código. Además, debemos asignarle el *Tag* «ExplosionFX», ya que es un *Tag* que ya existe y está relacionado con lo que queremos. En cambio, en la clase «PlayerHealth» se detecta el *Tag* «Enemy» en el método `OnCollisionEnter2D`. Debemos cambiarlo de la siguiente manera si queremos detectar las explosiones:

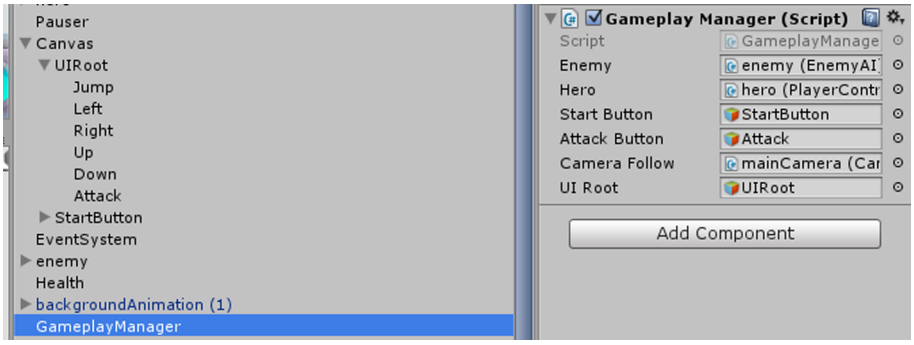
```
if (col.gameObject.tag  ==  "ExplosionFX")
```

Ahora simplemente debemos quitar en nuestro enemigo el *script* «PlayerControl» y añadir «EnemyAI», y con eso ya tendremos todos los personajes controlados.

Nos quedan algunas tareas, sin embargo. Necesitamos establecer los turnos y arreglar los movimientos de la cámara, ya que ahora mismo solo sigue a nuestro protagonista. Para ello hemos creado un *GameObject* vacío en el que hemos añadido un *script* «GameplayManager» para gestionar la dinámica del juego:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System.Collections.Generic;
public class GameplayManager: MonoBehaviour {
    public EnemyAI enemy;
    public PlayerControl hero;
    public GameObject startButton;
    public GameObject attackButton;
    public CameraFollow cameraFollow;
    public GameObject UIRoot;
    private Text[] mainText;
    private int timeRemaining = 10;
    private GameObject[] UI;

    void Start() {
       enemy.GetComponentInChildren<Gun>().gunFired += SwapTurn;
       hero.GetComponentInChildren<Gun>().gunFired += SwapTurn;
       mainText = startButton.GetComponentsInChildren<Text>();
       UIRoot.SetActive(false);
    }

    public void StartGame() {
        startButton.GetComponent < Button > ().enabled = false;
        enemy.hasTurn = true;
        InvokeRepeating("DecreaseTime", 0, 1);
        SwapTurn();
    }

    void SwapTurn() {
        StartCoroutine(SwapTurnCoroutine());
    }

    IEnumerator SwapTurnCoroutine() {
        timeRemaining = 10;
        hero.hasTurn = !hero.hasTurn;
        UIRoot.SetActive(hero.hasTurn);
        cameraFollow.SetPlayerToFollow(hero.hasTurn ? hero.transform : ene
my.transform);
        if (!enemy.hasTurn) {
            yield
            return new WaitForSeconds(2f);
        }
        enemy.hasTurn = !enemy.hasTurn;
    }

    void DecreaseTime() {
        timeRemaining--;
        if (timeRemaining < 0) SwapTurn();
        foreach(Text t in mainText) {
            t.text = "" + timeRemaining;
        }
    }
}
```

Como vemos, «GameplayManager» no solo controla los turnos, sino que hace algunas tareas más. Una de ellas es asegurarse de que los controles solo se muestran cuando el jugador tiene el turno. Para ello hemos creado un *GameObject* vacío en el que hemos emparentado todos los controles de usuario excepto el botón «Start», y se lo hemos asignado a la variable «UIRoot».



En cada cambio de turno, comprobamos que sea el turno del jugador y activamos la interfaz si es adecuado.

En «GameplayManager» también nos encargamos de mover la cámara. Esto lo hacemos ajustando el «Transform» que sigue el *script* «CameraFollow». Para que funcione, solo tenemos que añadir el método `SetPlayerToFollow` en el «CameraFollow».

Reto 7

Añadid el método `SetPlayerToFollow` en el *script* «CameraFollow», de manera que la cámara siga al «Transform» que le indicamos por parámetro.

Finalmente, en «GameplayManager» nos encargamos de mostrarle al jugador una cuenta atrás, para forzarle a que haga algún movimiento.

Ya solo nos queda arreglar la condición de acabar el juego. Nuestro juego se acaba cuando uno de los dos personajes (el héroe o el enemigo) se queda sin vida y muere.

En ese caso, se activa la animación «Die», que hace que el personaje caiga hacia abajo. La caída provoca que se active un «Trigger» en el *GameObject* «killTrigger». El problema es que este *GameObject* solo cubre la parte izquierda, con lo que debemos cambiarle el tamaño del «BoxCollider2D». Nosotros le hemos puesto un tamaño de X = 60.

Reto 8

Modificad el *script* «Remover» para que detecte la muerte tanto de nuestro personaje como del enemigo.

Con todos estos cambios ya tenemos un proyecto sencillo pero jugable.

5.4. Soluciones a los retos propuestos

Reto 1

La solución más sencilla es duplicar y cambiar los parámetros de los *scripts*. En concreto, dejaremos los parámetros en «Left Spawn Pos X = 24» y «Right Spawn Pos X = 72». El *script* se encarga de aleatorizar tanto el sentido como la velocidad o el ritmo de creación, pero para que no se note demasiado la simetría deberíamos cambiar algunos valores, en especial «Min Time Between Spawns» y «Max Time Between Spawns».

Reto 2

```
private void ActualizarAccionDown(KeyCode code) {
    if (!acciones.Contains(code)) acciones.Add(code);
}
private void ActualizarAccionUp(KeyCode code) {
    if (acciones.Contains(code)) acciones.Remove(code);
}
public void MueveDerechaDown() {
    ActualizarAccionDown(KeyCode.RightArrow);
}
public void MueveIzquierdaDown() {
    ActualizarAccionDown(KeyCode.LeftArrow);
}
public void RotaIzquierdaDown() {
    ActualizarAccionDown(KeyCode.UpArrow);
}
public void RotaDerechaDown() {
    ActualizarAccionDown(KeyCode.DownArrow);
}
public void MueveDerechaUp() {
    ActualizarAccionUp(KeyCode.RightArrow);
}
public void MueveIzquierdaUp() {
    ActualizarAccionUp(KeyCode.LeftArrow);
}
public void RotaIzquierdaUp() {
    ActualizarAccionUp(KeyCode.UpArrow);
}
public void RotaDerechaUp() {
    ActualizarAccionUp(KeyCode.DownArrow);
}
```

Reto 3

```
public void Jump() {
    jump = true;
}
```

Con este simple método podemos asociar el evento «OnClick» del botón de saltar (el que está a la derecha del botón con forma de triángulo) al método «Jump».

Reto 4

```
42.
43.         float  CalculaVelocidad(Transform  target,  float  angle)
44.         {
45.             var  dir  =  target.position  -  transform.position;
46.             var  h  =  dir.y;
47.             dir.y  =  0;
48.             var  dist  =  dir.magnitude;
49.             var  a  =  angle  *  Mathf.Deg2Rad;
50.
51.             dist  +=  h  /  Mathf.Tan(a);
52.
53.             return  Mathf.Sqrt(dist  *  Physics.gravity.magnitude  /  Math
f.Sin(2  *  a))  *  Random.Range(1.2 f, 1.8 f);
54.         }
```

Reto 5

```
 void  OnTriggerEnter2D (Collider2D  col)       {

       if (col.tag  !=  ignoreTag)
       {

            // Call the explosion instantiation.
           OnExplode();

           // Destroy the rocket.
           Destroy (gameObject);
       }
   }
```

Reto 6

```
 void  OnTriggerEnter2D (Collider2D  col)       {

       if (col.tag  !=  ignoreTag)
       {
           GameObject  explosion  =  new  GameObject("Explosion");
           explosion.transform.position  =  transform.position;
           explosion.tag  =  "ExplosionFX";
           Destroy(explosion,  0.5 f);
           CircleCollider2D explosionRadius = explosion.AddComponent<CircleCollider2
D>();

           explosionRadius.radius  =  2.5 f;

            // Call the explosion instantiation.
           OnExplode();

           // Destroy the rocket.
           Destroy (gameObject);
       }
   }
```

Reto 7

```
public void SetPlayerToFollow(Transform target)
{
    player = target;
}
```

Reto 8

En el *script* «Remover» debemos cambiar la primera línea de código del método `OnTriggerEnter2D` por la siguiente:

```
if (col.gameObject.tag == "Player" || col.gameObject.tag == "Enemy")
```

De esta manera, detectará la muerte tanto de nuestro personaje como de nuestro enemigo.

Resumen

En este módulo didáctico hemos visto algunos conceptos un poco más avanzados de Unity. Los primeros son los métodos de «Invoke» y las *Coroutines*. «Invoke» nos sirve para programar llamadas a métodos en un futuro, o para definir llamadas periódicas de una manera muy sencilla. Las *Coroutines*, en cambio, nos permiten ejecutar funciones complejas sin afectar al rendimiento del juego. Hay que recordar que Unity ejecuta todos nuestros *scripts* en cada *frame*, y si tenemos operaciones muy complejas, como cálculos de inteligencia artificial, podríamos no ser capaces de mantener un buen rendimiento. Con las *Coroutines*, sin embargo, podemos definir estos métodos de manera que se ejecuten sin afectar tanto al rendimiento.

Otro de los elementos que hemos visto en esta unidad son los sistemas de partículas. Con ellos podremos decorar con relativamente poco esfuerzo nuestro juego, y le darán un aspecto visual mucho más cuidado y trabajado. Además, estos sistemas son muy eficientes y, si no abusamos de ellos, tendrán un impacto muy bajo en el rendimiento de nuestro videojuego.

También hemos visto una de las joyas de la corona de Unity: Mecanim. Los responsables del motor decidieron para la versión 4 de Unity que necesitaban un sistema de animación moderno y eficaz, y compraron e integraron Mecanim. Este sistema se basa en una máquina de estados que podemos definir y configurar de manera visual. Hemos visto cómo hacerlo y cómo interactuar con ella vía *scripting*.

Seguidamente, hemos visto los conceptos de *Tags* y *Layers*. Tanto los unos como las otras son utilizados de manera bastante habitual en una gran cantidad de proyectos, especialmente cuando son complejos. Gracias a los *Tags* podemos encontrar todos los objetos de una escena a los que hayamos asignado un *Tag* en concreto. Esto es realmente útil para, por ejemplo, ejecutar una acción para todos los enemigos a la vez. Los *Layers*, como hemos visto, se usan de manera parecida, pero su función es muy distinta. Sirven para indicarle a Unity que trate un objeto de manera distinta, y podemos configurar ciertos *components* para que actúen de manera diferente para algunos *Layers*.

Finalmente, hemos cogido un ejemplo existente de un juego sencillo y lo hemos adaptado para construir un proyecto totalmente distinto. Tanto la mecánica como el escenario han sido modificadas sensiblemente, con el resultado de un juego inspirado en un clásico.