Práctica 3

Creación de un juego para móvil con animaciones, interfaz y sonidos.

Índice

1. Importar el proyecto y configurar la escena y los modelos 3D

En esta práctica el alumno desarrollará un pequeño videojuego, utilizando modelos animados para dotar de dinamismo la escena. El objetivo del mismo será manejar una araña para eliminar a todos los esqueletos del escenario. Los enemigos no estarán indefensos pues, al acercarse a ellos, atacarán a la araña sin miramiento.

Además de la lógica necesaria para gestionar el juego, también se pide desarrollar una interfaz que posibilite al jugador navegar por el menú principal y poder controlar el juego a través de un *GamePad* virtual, disponible sobre la pantalla.

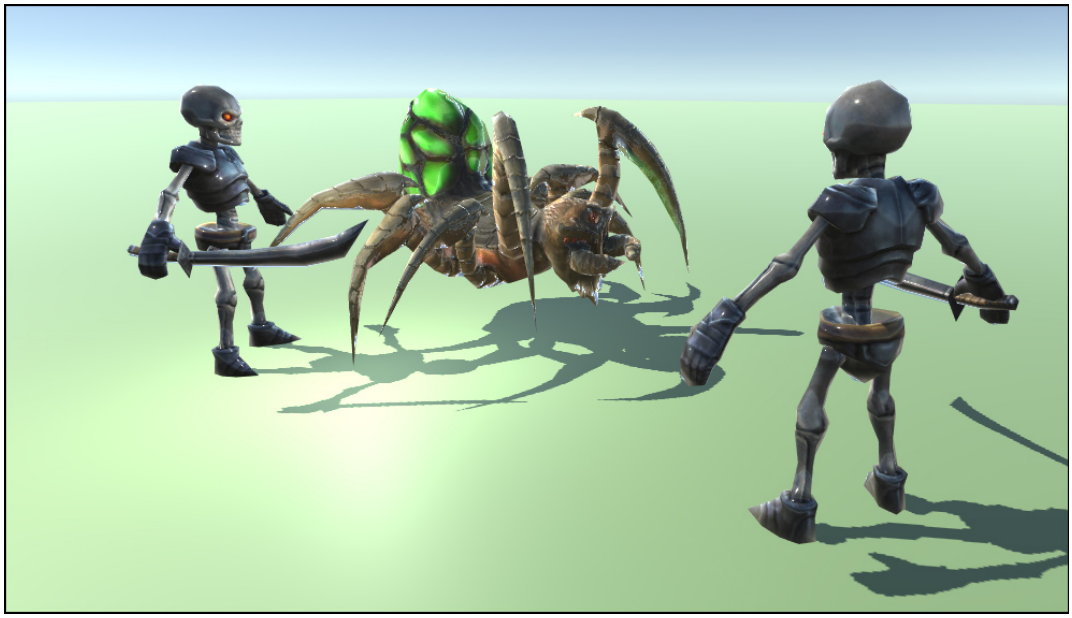

Figura 1. Imagen del juego.

Podemos ver los elementos y el funcionamiento del juego en el siguiente vídeo:
https://goo.gl/Gmaikz

1.1. Importar un nuevo proyecto y configurar la escena

Para empezar debemos descargarnos el proyecto base donde figuran los modelos y sonidos necesarios para realizar la práctica:
https://github.com/oscarviu/Practica-3

A continuación crearemos una nueva escena llamada **`P3.unity`** sobre la cual añadiremos una **primitiva plano** cuyo centro estará ubicado en las **coordenadas (0,0,0)**. Sobre él se colocarán los diferentes elementos que decorarán el escenario, cada uno de ellos con su *collider* correctamente

configurado para evitar que el personaje principal los atraviese. En la figura 2 tenemos un ejemplo de un escenario válido (no es necesario que contenga los mismos elementos).

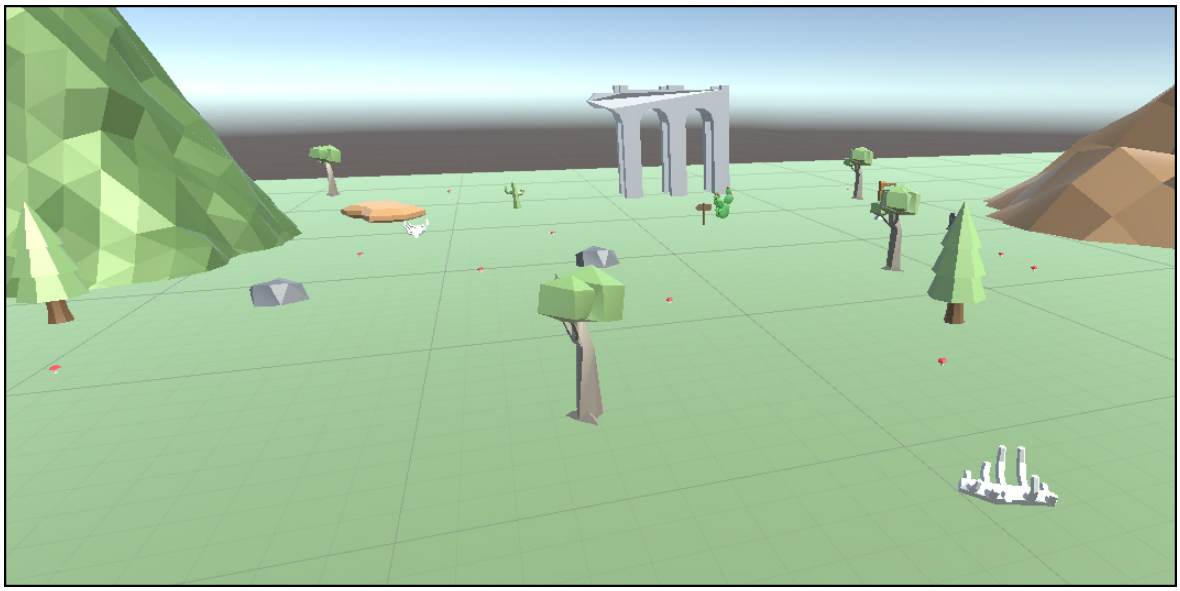

Figura 2. Elementos del escenario.

Todos los elementos deben incluirse dentro de un objeto vacío (ubicado en el centro de coordenadas) llamado *Environment*.

1.2. Preparar las animaciones del jugador principal (araña)

Si navegamos en el interior del proyecto base veremos que tenemos una carpeta con los ficheros que definen el personaje de la araña.

Figura 3. Personaje del jugador principal

Para empezar vamos a crear un **`Animator Controller`** con todas los estados de la araña. Las animaciones se encuentran en el interior del propio modelo (`SPIDER.fbx`). Este controlador tendrá dos capas para independizar las animaciones de movimiento de las de ataque o daño.

En la capa **Base Layer** definiremos los **estados de movimiento**:

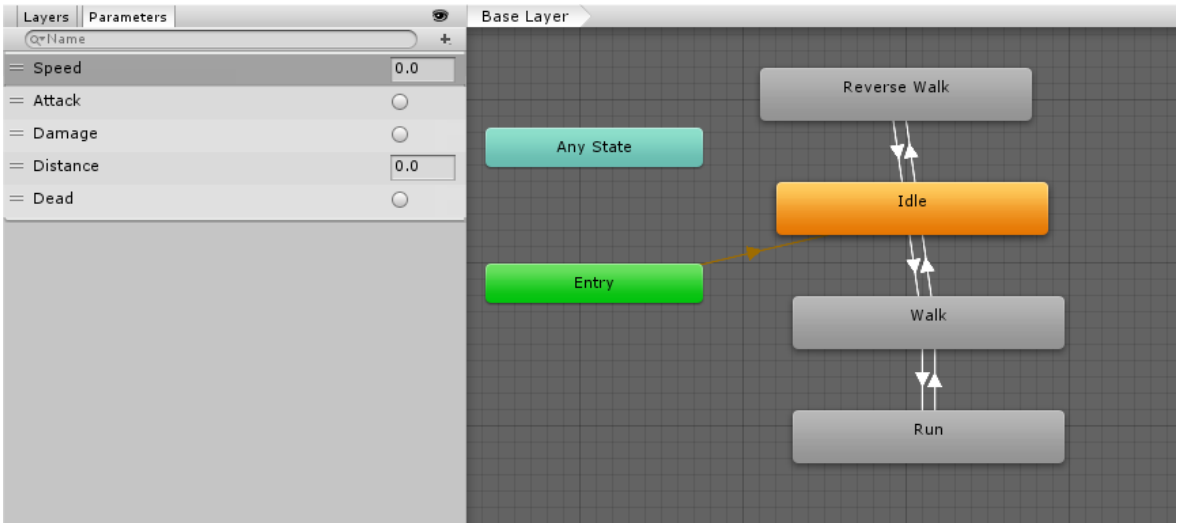

Figura 4. Estados de la capa Base Layer

El parámetro *float* speed indicará si la araña no se mueve (entre -0.1 y 0.1), camina hacia delante (entre 0.1 y 1.1), camina hacia atrás (entre -0.1 y -1), o corre (entre 1.1 y 2).

En la capa **Attack Layer** se definirán los estados para atacar, recibir daño y morir:

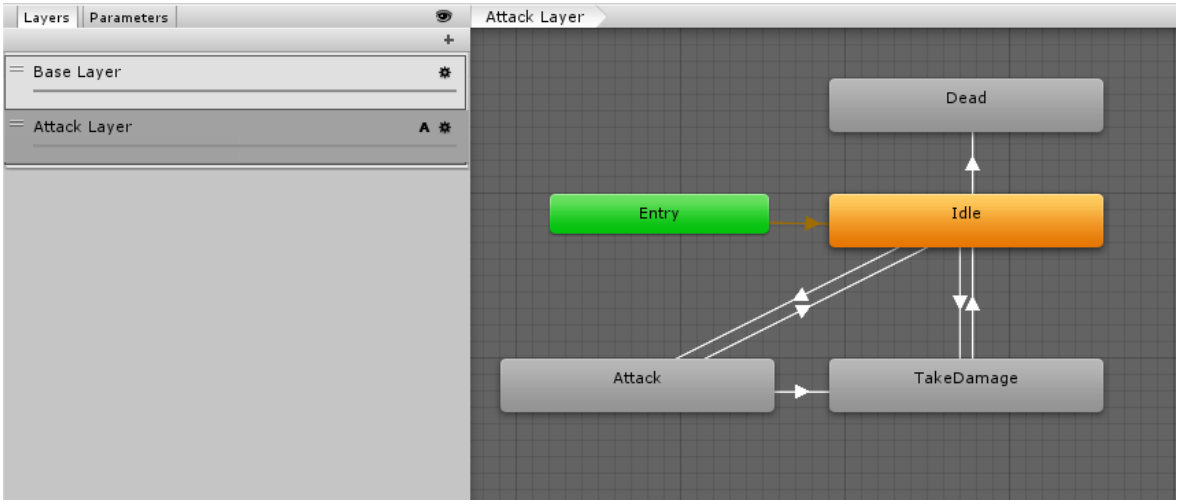

Figura 5. Estados de la capa Attack Layer

El parámetro *trigger* **`attack`** disparará el estado *Attack*, el parámetro **`Damage`** disparará el estado *TakeDamage* y el parámetro **Dead** el estado *Dead*.

Para conocer los detalles de las animaciones ver el siguiente vídeo:
https://goo.gl/uDUJSY

El parámetro **`Distance`** será un parámetro que controle cuando mover el *collider*. Sus valores se definen por una curva sobre la animación de ataque:
https://goo.gl/rzYKgq

1.3. Preparar las animaciones del enemigo (esqueleto)

Este objeto será a la vez el objetivo a abatir y el encargado de evitar que consigamos superar el juego.

Figura 6. Personaje que representa al enemigo.

En principio tenemos un *prefab* con todos los componentes asignados (excepto el *script*), con lo que tan sólo necesitamos programar la máquina de estados. Primero crearemos otro **`Animator Controller`** con los estados de la siguiente figura:

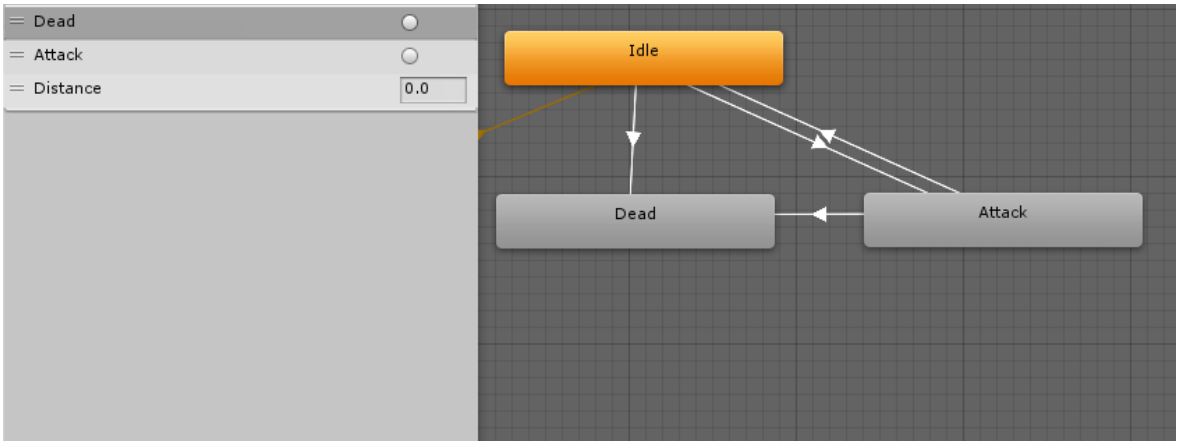

Figura 7. Estados del enemigo.

Tan sólo constará de dos animaciones (atacar y morir) que se dispararán con los parámetros **`Attack`** y **`Dead`**. Para conocer los detalles de la máquina vea el siguiente vídeo:
https://goo.gl/AVqyG6

De igual manera que con la araña, definiremos un parámetro llamado **`Distance`** sobre el cual hemos conectado una curva de animación para mover el *collider* cuando ataque:
https://goo.gl/UUQoAm

2. Preparar la interfaz de juego

Al arrancar el juego nos aparecerá un menú principal con dos opciones: Iniciar una partida o habilitar/deshabilitar el audio.

2.1. Menú principal

Lo primero que debemos crear es el menú principal. Crearemos un panel que ocupará toda la pantalla, dentro del cual ubicaremos el resto de elementos:

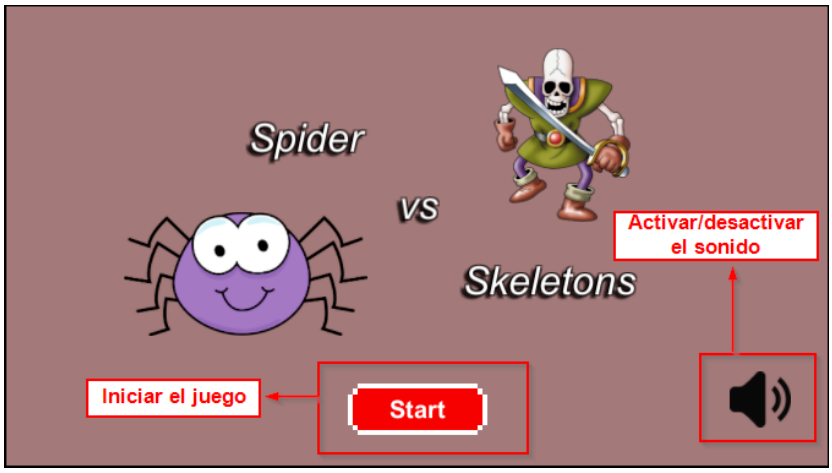

Figura 8. Menú principal del juego.

Justo al crear el panel principal deberemos modificar el componente *Canvas Scaler* del objeto *Canvas* con los valores de la siguiente figura:

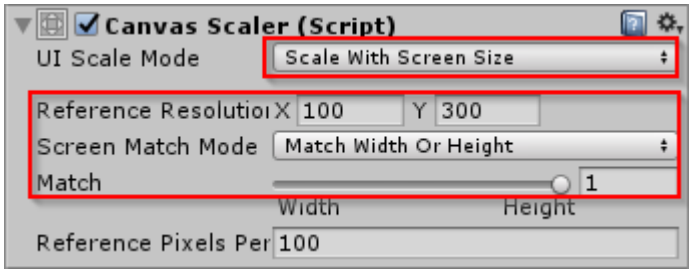

Figura 9. Valores de Canvas Scaler.

Los elementos de la interfaz son todos ellos decorativos excepto dos botones que aparecen marcados en rojo (para iniciar el juego y activar/desactivar el sonido).

Las dos imágenes (araña y esqueleto) y los textos centrales, están referenciados al centro de la pantalla.

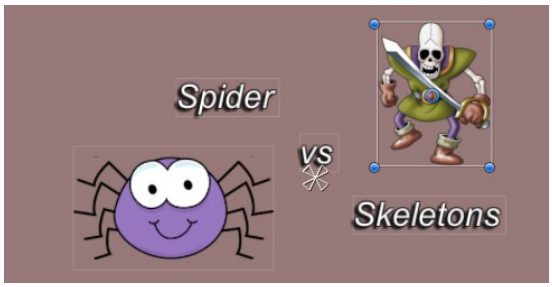

Figura 10. Anclas de los elementos decorativos.

Los botones *Start* y *Sound*, en cambio, están referenciados a la parte inferior e inferior-derecha respectivamente.

Figura 11. Anclas de los botones del menú principal.

El botón ***Start*** utiliza como fuente de imagen (*source image*) el *sprite* `button.png` de tipo *sliced* y cuyos bordes se encuentran en el interior del dibujo.

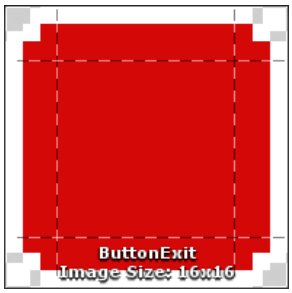

Figura 12. Sprite del botón Start.

2.2. Interfaz durante el juego (HUD)

Este panel (que será un objeto vacío llamado ***GamePanel***, tendrá definidos en su interior un botón de salida (*Exit*), un panel con la puntuación (*Score*), un panel que aparecerá al finalizar el juego (*Final Panel*) y un *Gamepad* virtual del que hablaremos más adelante.

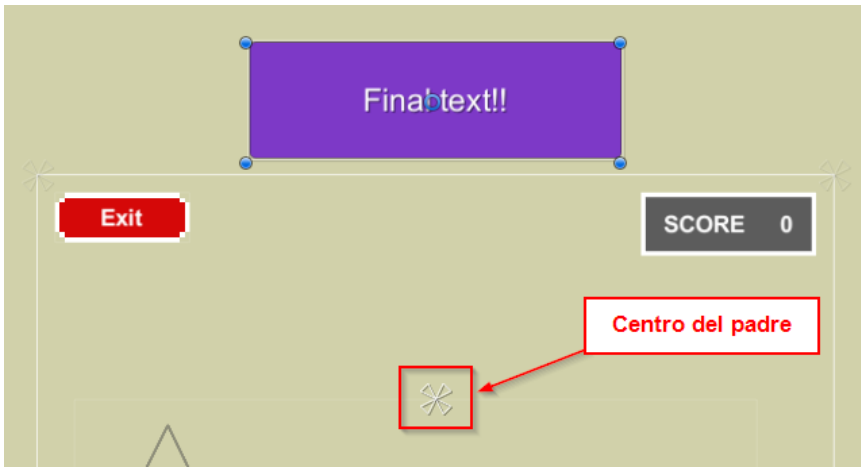

Figura 13. *Sprite* del botón Start.

Los dos elementos en los extremos izquierda y derecha se encargarán de posibilitar la salida al menú principal (botón *Exit*) y de proporcionar la puntuación actual (panel *Score*).

Figura 14. Anclas de los elementos superiores

El botón *Exit* utiliza el mismo formato que el botón *Start* del menú principal. En cambio, *Score* es una combinación de diferentes elementos (imagen de fondo gris, 4 barras blancas decorando los extremos y dos textos en blanco). Durante el juego tan sólo se modificará por *script* el texto con la puntuación (el número).

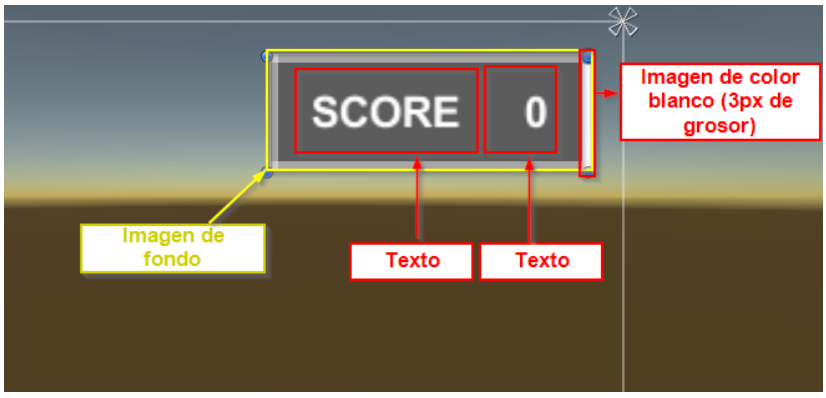

Figura 15. Elementos del panel *Score*.

El panel superior, en cambio, es un panel animado que se mostrará en el caso de que ganemos el juego o lo perdamos (con diferente texto informativo). Habrá que primero crear la animación (desplazar desde fuera de la pantalla hasta el centro de la misma verticalmente) y luego configurar su **`Animator Controller`**. El valor del texto se actualizará por un *script* dependiendo de si ganamos o perdemos.

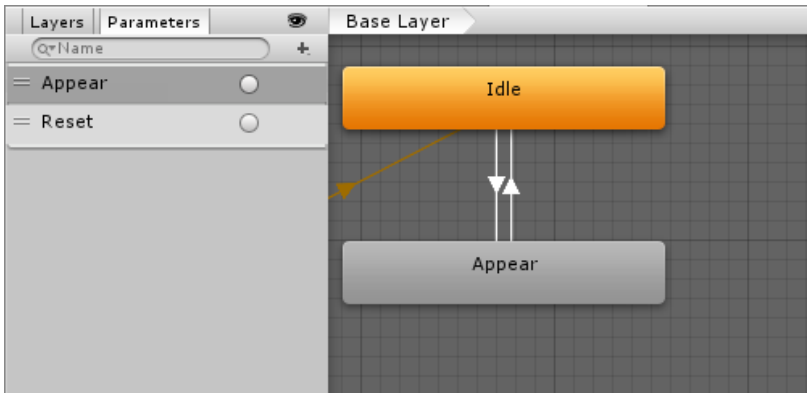

Figura 16. *Animator Controller* de Final Panel.

Para conocer los detalles de esta máquina ver el siguiente vídeo:
https://goo.gl/gj4JCY

2.3. Interfaz durante el juego (GamePad)

Esta interfaz representa un **GamePad virtual** que sustituirá a teclado o mando físico para poder jugar al juego sobre una **plataforma táctil** (un móvil o una *tablet*).

Todos los elementos son de tipo **Imagen** (no son botones) y se encuentran bajo un objeto vacío (llamado *GamePad*). Las anclas de este objeto están ubicadas sobre las esquinas inferiores del padre (Panel).

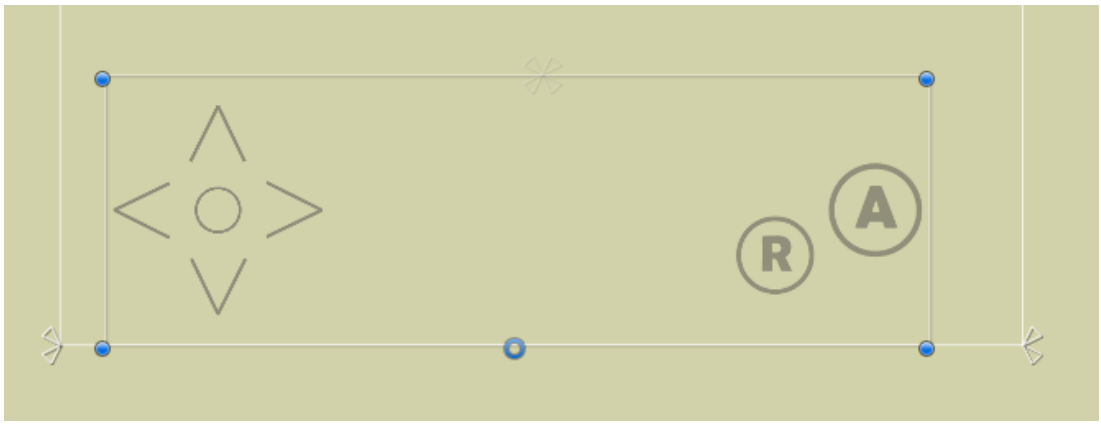

Figura 17. *Animator Controller* de Final Panel.

Empecemos por las dos imágenes de la derecha: Estos dos elementos representan los botones de **ataque** y **correr** (‘A’ y ‘R’ respectivamente); sus anclas están asociadas con el **extremo derecho** de ***GamePad***.

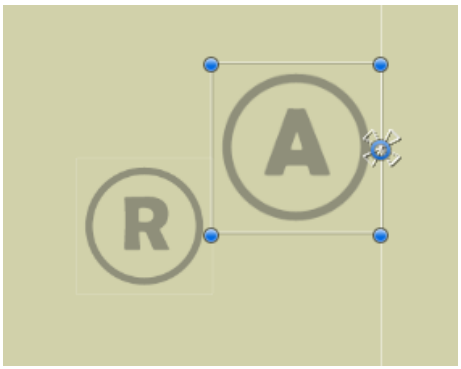

Figura 18. “botones” disponibles para móvil.

La cruceta, en cambio, son varios elementos agrupados bajo un objeto vacío llamado **`Cross`** que se encuentra asociado al extremo izquierdo de ***GamePad***. Las anclas de esta agrupación se encuentran ubicadas en la parte izquierda de *GamePad*, mientras que cada una de las partes tiene sus anclas ubicadas en la intersección de su punta y los límites de *Cross*.

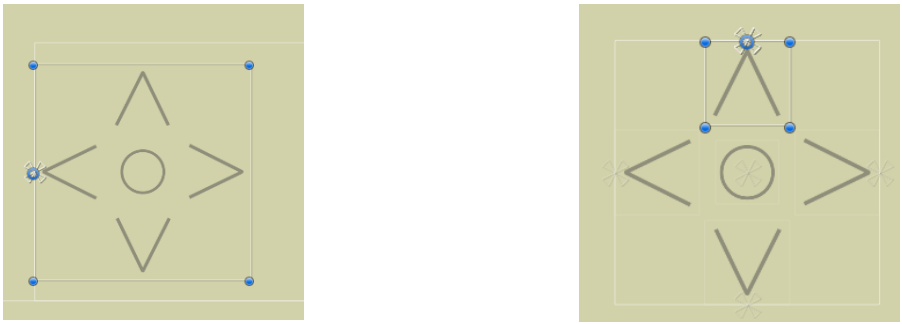

Figura 19. Cruceta que agrupa las diferentes partes (izquierda) y una de las partes seleccionada.

Sobre **cada una de las partes** (flechas) se deberá asignar el *script* `UICrossButton` configurado con su dirección (UP, LEFT, RIGHT, DOWN), además del componente *Event Trigger* para detectar dos eventos:

- ***Pointer Enter*** ==> Donde llamaremos al método `update(true)` de `UICrossButton`.
- ***Pointer Exit*** ==> Donde llamaremos al método `update(false)` de `UICrossButton`.

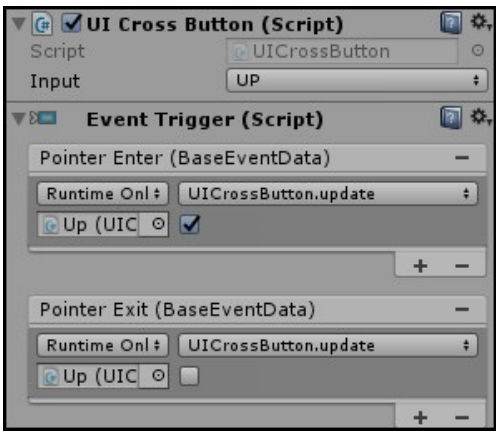

Figura 20. Componentes asignados a cada una de las flechas dela cruceta. En este ejemplo se trata de la fecha “arriba”.

A modo de resumen, la jerarquía de objetos dentro del *canvas* es la siguiente:

Figura 21. Jerarquía del canvas.

Como reto opcional podríamos mostrar una interfaz dentro del *GamePanel* con las vidas restantes. Podemos enseñar esta información de forma numérica o como un contador de corazones (similar al utilizado en la famosa saga Zelda).

3. Programar los comportamientos

En esta parte del desarrollo vamos a ocuparnos de rellenar y asignar los *scripts* encargados de toda la lógica del juego.

Todas las clases necesarias para diseñar la lógica existen dentro de la carpeta *Scripts*. Los esqueletos de las clases vienen dados y tan sólo se han de rellenar todos los espacios marcados con **`//TODO`**.

Habrá diferentes clases para gestionar la interfaz y la lógica durante el juego. Empecemos, pues, con las clases que definen la lógica del juego: *PlayerBehaviour*, *SkeletonBehaviour* y *GameManager*.

Antes de empezar a programar deberemos **etiquetar** al objeto araña con el *tag* **Player** y a los esqueletos con el *tag* **Enemy**.

3.1. Clases que definen la lógica del juego

- **`PlayerBehaviour`**: Esta clase definirá cómo se comportará el personaje con el que jugaremos el nivel (la araña).
    - Empezaremos con **3 vidas** que iremos perdiendo a medida que los enemigos nos alcancen.
    - **Moveremos** la araña con las **flechas del teclado** (si mantenemos la t**ecla *shift*** a la vez con la flecha 'arriba' **correrá** en lugar de caminar). Al pulsar la tecla 'espacio' la araña **atacará**.
    - Para saber cuando hemos alcanzado al objetivo utilizaremos el **evento de colisión** entrante y los ***tags***. Cuando 'choco' con el enemigo (tag == Enemy) comprobaré si estoy en el estado *Attack* además de si el parámetro *Distance* > 0. En ese caso el ataque será válido y destruiré al enemigo, llamando al método *kill()* de *SkeletonBehaviour*.
    - Tendremos un método que al ejecutarse **recibiremos daño** (perderemos una vida y dispararemos la animación *damage*), e informaremos al *GameManager* si nos quedamos sin vidas. Este método lo ejecutarán los enemigos cuando nos ataquen.
- **`SkeletonBehaviour`**: Esta clase representa al enemigo y su estructura es muy sencilla:
    - Al contrario que la araña, el enemigo permanecerá quieto hasta que el jugador (araña) se le acerque a menos de 1m. Entonces el enemigo se orientará hacia *player* (no se moverá) y activará el ataque cada segundo que la araña esté a su alcance (frecuencia de 1s).
    - Para conocer si el esqueleto ha atacado (impactado) con la araña utilizaremos la misma lógica que en *PlayerBehaviour* (comprobar estado y *Distance* en el evento de colisión entrante). Si el ataque es válido llamaré al método *recieveDamage()* de *PlayerBehaviour*.

- Tendremos un método para que al ejecutarse nos destruyan. En este método dispararemos la animación de muerte e informaremos al *GameManager* de que nos han matado.

- **`GameManager`**: Esta clase representa la lógica del juego en sí, controlando los enemigos restantes y la puntuación actual del jugador. Para su acceso haremos uso del patrón *Singleton* que nos permitirá acceder al mismo utilizando el atributo de clase **`instance`** (**`GameManager.instance`**). este comportamiento se encontrará asociado a un objeto vacío de la escena y actuará de árbitro cada vez que se produzcan los siguientes eventos:

  - **`notifyEnemyKilled()`**: Este método los ejecutarán los esqueletos cuando mueran y lo que haremos será aumentar la puntuación actual, informando justo después al UIManager (manager de interfaz), y si no quedan enemigos en el nivel indicar que el juego ha finalizado ("nivel superado").

  - **`notifyPlayerDead()`**: De manera similar al anterior, este método lo ejecutará *player* cuando reciba un ataque y se quede sin vidas. Lo que haremos será indicar que el juego ha finalizado ("has perdido").

  - **`onStartGameButton()`**: Este método lo invocará el usuario al pulsar sobre el botón *Start* del menú principal (conectado con el evento del *onClick* del botón) y lo que hará será arrancar el juego (ocultar el menú principal y despausar a player).

  - **`onExitGameButton()`**: Este método se ejecutará al hacer "click" sobre el botón *Exit*, mostrando de nuevo el menú principal y resetearemos todos los elementos de la escena.

3.2 Clases para la gestión de la UI

Las clases que gestionan el comportamiento de las interfaces son las siguientes

- **`UISoundButtonBehaviour.cs`**: esta clase estará asociada al icono de sonido en el menú principal y lo que hará será activar o desactivar el audio en el juego.

- **`UIManager.cs`**: este *script* representa al **manager de interfaz**. De nuevo se trata de un singleton con lo que podemos acceder a él desde cualquiera de nuestros *scripts*. Su función es la de **comunicar al *GameManager* con la interfaz**. En principio estará asociado al *canvas* de la interfaz.

- **`FinalPanelBehaviour.cs`**: esta clase se encuentra anclada al panel *FinalPanel* y contendrá **métodos para dispara la animación** del panel cuando el *GameManager* informe de que el juego ha finalizado. *GameManager* utilizará a *UIManager* para indicar este evento y disparar la animación.

En el siguiente vídeo se detalla cada una de las clases que necesitamos para el juego:
https://goo.gl/GcGz6Z

3.3 Música y sonidos

Una vez ya tenemos programadas las clases *GameManager*, *PlayerBehaviour* y *SkeletonBehaviour* vamos a incluir el sonido en nuestro proyecto.

En la carpeta ***Sounds*** encontrarás los sonidos disponibles en el juego. Entre ellos se encuentra una melodía que se escuchará desde el principio y en bucle, y sonidos que se iniciarán en el momento que indica su nombre (cuando ataca/muere/recibe daño la araño o el esqueleto).

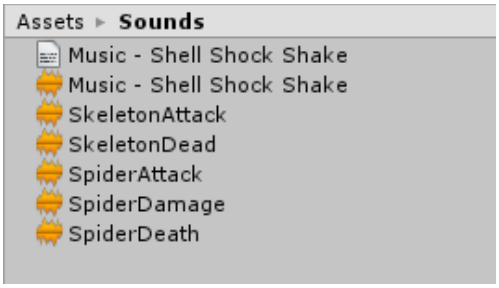

Figura 22. Ficheros de audio que se utilizarán en el juego.

Deberemos crear un **Audio Source** sobre la **araña** y los **esqueletos**(utiliza un prefab para evitarte trabajo) y modificar las funciones ***PlayerBehaviour*** y ***SkeletonBehaviour*** para incluir en sus parámetros los ficheros de audio a reproducir y lanzar los sonidos en los momentos indicados (método **`Play()`** de *AudioSource*).

Probablemente el sonido no case con la animación y haya que retrasar su reproducción. Utilizaremos el método **`PlayDelayed(float seconds)`** para lanzar sonidos con retraso.

Somos libres de añadir nuevos sonidos para, por ejemplo, dotar de audio a los botones cuando hacemos click sobre ellos, o que al caminar la araña suene un sonido de patas golpeando el suelo. No obstante, no es un requisito de esta práctica.

3.3 Adaptación del juego a móvil

Para que podamos utilizar el móvil a la hora de jugar a nuestro juego necesitaremos que el *GamePad* virtual responda a los **eventos táctiles**. Para ello, siguiendo los pasos de la página 11, incluiremos el script **`UICrossButton.cs`** sobre cada una de las flechas, configurando el parámetro ***Input*** según la dirección de la misma. En cuanto a los botones "Correr" y "Atacar" podemos crear un *script* más simple basándonos en *UICrossButton* y asociárselo ,o utilizar el componente *Button* para procesar el momento de cuando pulsamos el mismo.

El siguiente de la paso será modificar la clase ***PlayerBehaviour*** para utilizar los eventos de ***UICrossButtons*** en la función *FixedUpdate()*. Para ello utilizaremos el método de clase ***UICrossButton.GetInput(InputType input)*** para saber si un botón en particular está siendo pulsado.

```
// Si giro izquierda: _angularSpeed = -rotateSpeed;
if (Input.GetKey(KeyCode.LeftArrow) || UICrossButton.GetInput(InputType.LEFT))
{
    _angularSpeed = -rotateSpeed;
}
```

Figura 23. Utilización de la clase UICrossButton.

Para terminar deberemos cambiar la plataforma a Android, instalar lo necesario para compilar el proyecto (java, Android SDK) y configurar los parámetros de **Player Settings** con los siguientes valores:

- **Nombre de la compañía**: GameEngines I
- **Nombre del juego**: Practica 3
- **Orientación**: Auto-rotación solamente en *Landscape*.
- **Nombre del paquete**: com.GameEngines1.Practica3
- **Versión**: 1.0

A modo de resumen tan sólo recordar la estructura de objetos requerida en la práctica. Podemos observar la misma en la siguiente figura:

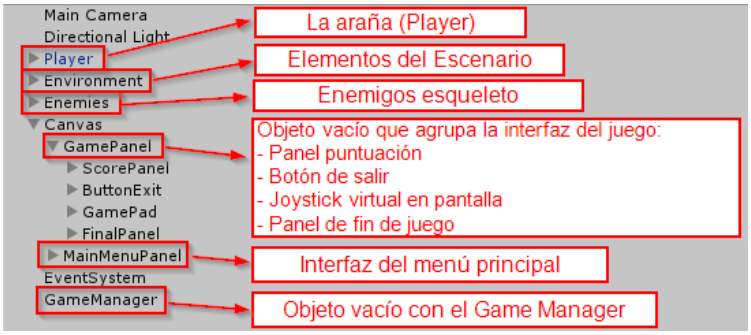

Figura 24. Resumen de los objetos que contendrán los elementos de la práctica.

Calificación de la práctica	
1.1. Importar un nuevo proyecto y configurar la escena	10%
1.2. Preparar las animaciones del jugador principal (araña)	10%
1.3. Preparar las animaciones del enemigo (esqueleto)	10%
2.1. Menú principal	10%
2.2. Interfaz durante el juego (HUD)	10%
2.3. Interfaz durante el juego (GamePad)	10%
3.1. Clases que definen la lógica del juego	10%
3.2 Clases para la gestión de la UI	10%
3.3 Música y sonidos	10%
3.3 Adaptación del juego a móvil	10%

Para realizar la entrega se deberá exportar el proyecto completo indicando, como nombre del archivo, vuestro nombre completo con la siguiente estructura:
`P3_Apellido1_Apellido2_Nombre.unitypackage`