Práctica 2: Rompe Moldes

Nada está perdido si se tiene por fin el
valor de proclamar que todo está perdido
y hay que empezar de nuevo.

Rayuela
Julio Cortázar

2.1. Descripción

Rompe Moldes es un juego en el que se deben romper, utilizando una pelota, todos los ladrillos situados en la parte superior de un tablero. El jugador dispone de una pala con la que va golpeando la pelota, haciendo que ésta rebote por todo el tablero. Cuando la pelota golpea los ladrillos, éstos se rompen. Si el jugador deja caer la pelota sin llegar a golpearla, pierde una vida. El juego termina cuando o bien se han roto todos los ladrillos, o bien se han perdido todas las vidas.

En el tablero existen distintos tipos de ladrillos, representados con colores diferentes, que se comportan de formas distintas. Unos ladrillos se rompen con un golpe de la pelota y otros necesitan más de un golpe. Algunos de los ladrillos, al romperse, liberan objetos que, de ser recogidos por el jugador, le dan habilidades especiales:

- *Power-up* que alarga la pala un 25 %.
- *Power-up* que permite al jugador disparar balas con la pala espaciadora. Las balas equivalen a un golpe con la pelota por lo que se debe disparar las veces necesarias para eliminar ladrillos de dureza extra.

El jugador puede tener un solo *power-up* simultáneamente. Por tanto si, por ejemplo, teniendo el arma recoge el otro, perderá el arma.

2.2. Implementación

Es recomendable realizar la práctica de forma incremental, consiguiendo que funcionen los distintos componentes y *game objects*, antes de incorporarlos en la lógica del juego. Los siguientes apartados están pensados para realizarlos en orden y cada uno de ellos en una escena diferente. Junto con este enunciado, se proporcionan algunos ficheros adicionales para ser utilizados durante el desarrollo.

Es muy importante asegurarse de realizar una configuración cuidadosa tanto de las capas de ordenación *(sorting layers)* como de las capas de colisión *(layers)*.

Escena 2.1: Pixel Art

Nombre de escena: `Assets/Scenes/test/1_8bits`

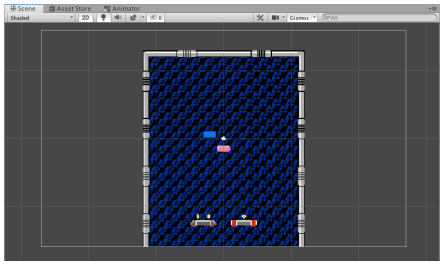
Para darle a nuestro juego el aire retro que se merece, utilizamos los *sprites* disponibles en el Campus Virtual, obtenidos principalmente de nesmaps y de spriters-resource.

Siguiendo las pautas generales explicadas en clase, y las particulares para este tipo de juegos, construiremos una escena que nos permita probar los ajustes adecuados de importación y configuración.

En nuestro **proyecto 2D** creamos una escena de prueba y configuramos la **cámara ortográfica con tamaño 16**.

Practicamos la importación de *sprites* asegurándonos de configurar los ajustes a **8 píxeles por unidad**, sin filtro -*Filter mode: Point (no filter)*- y sin compresión -*Compression: none*- (no usar compresión se debe exclusivamente a que estos *sprites* ocupan muy poco y no queremos perder nada de definición).

De la *sprite sheet* `NES - Arkanoid - Fields`, elegimos alguno de los *sprites* de fondo para el tablero, lo situamos en la posición (0, -1.5, 0) y nos aseguramos de que se dibuje siempre en el fondo. Usamos también algún otro *sprite* para comprobar que las dimensiones obtenidas con las importaciones son correctas, no necesitando cambiar el tamaño (escala) de ninguno de los *sprites* como, por ejemplo, en la siguiente escena de prueba:

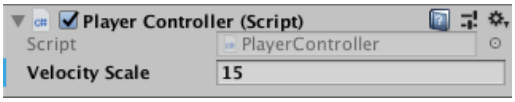


◇

Escena 2.2: Mover al jugador dentro de los límites del tablero

Nombre de escena: `Assets/Scenes/test/2_Player`

Usamos el sprite `VausSpacecraft` para crear el *game object* físico `Player` e implementamos un componente `PlayerController`, responsable de mover al jugador en horizontal. El movimiento aprovecha el **motor de física 2D** de Unity, para mover al objeto cambiando su velocidad en base al eje horizontal de la entrada. El movimiento no se ve afectado por la inercia.



Añadimos *colliders* en los bordes del tablero, de manera que podamos comprobar que el movimiento físico funciona correctamente. ◇

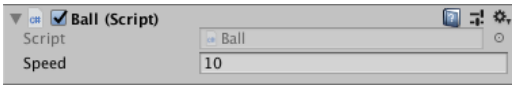
Escena 2.3: Movimiento de la bola

Nombre de escena: `Assets/Scenes/test/3_Ball`

Usamos el sprite `EnergyBall` para construir el *game object* `Ball`, cuya configuración va a estar marcada por las dos "personalidades" diferenciadas de la bola:

- Empieza estando "parada", dejándose llevar por la pala.
- Al pulsar la tecla *Return* (botón *Submit*), la bola se mueve en diagonal hacia arriba, a una determinada velocidad, rebotando contra los muros superior y laterales del tablero.

El cambio en el comportamiento de la bola lo logramos configurando adecuadamente el *game object*, e implementando un componente `Ball` que garantice que la bola es un objeto cinemático, hijo de la pala, mientras que el usuario no la ponga en movimiento. Cuando esto ocurra, la bola pasa a ser un objeto físico (propiedad `isKinematic` a `false`), independiente (sin objeto padre), moviéndose a una determinada velocidad en una cierta dirección, sin perder energía con los choques.



Estamos en condiciones de continuar con el siguiente apartado, dejando para más adelante la mejora del comportamiento de la bola de dos maneras complementarias:

1. Utilizando aleatoriedad para el vector de dirección del movimiento, de forma que no siempre salga en la misma dirección. Debe ir hacia arriba, pero el ángulo puede variar en cada disparo.
2. Teniendo en cuenta en qué punto de la pala choca, para ajustar el cambio del vector de dirección al producirse el rebote, de forma que el usuario pueda aprovechar esta característica al jugar.

Estas mejoras se realizarán para la versión final, una vez terminadas las escenas de prueba.

◇

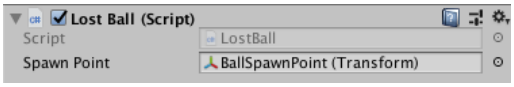
Escena 2.4: Zona de muerte

Nombre de escena: `Assets/Scenes/test/4_DeathZone`

Una *death zone* o zona de muerte es un recurso que se suele utilizar para simular la muerte del jugador cuando se sale de los límites del mundo.

Creamos un *trigger* `DeathZone`, sin componente visual, ocupando la parte inferior de nuestro tablero. Le añadimos un componente `DeathZone`, que interactúa con el objeto con el que colisiona de la siguiente forma: si el otro objeto tiene un componente `LostBall` entonces ejecuta el método `OnLost` de dicho componente; si no lo tiene, destruye al objeto.

Esto implica la necesidad de implementar también un componente `LostBall` para añadirlo a la bola. El método `OnLost` provoca la reaparición de la bola en su "personalidad" cinemática en el punto de *respawn*.

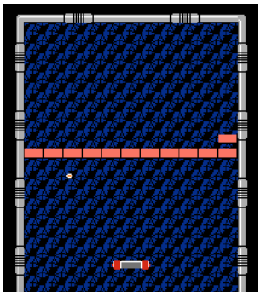


◇

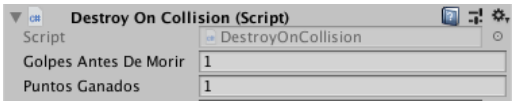
Escena 2.5: Ladrillos

Nombre de escena: `Assets/Scenes/test/5_Bricks`

Usamos algún *sprite* de ladrillo, como `OrangeBrick`, para crear *game objects* de forma que aseguremos golpear algún ladrillo con la bola nada más empezar la ejecución, para así facilitarnos las pruebas.



Creamos un componente `DestroyOnCollision` que permita eliminar un ladrillo tras un número configurable de golpes. También podemos configurar desde el editor el número de puntos que consigue el jugador al eliminarlo.



◇

Escena 2.6: Gestor del juego

Nombre de escena: `Assets/Scenes/test/6_GameManager`

Añadimos un *GameManager*, tal y como se ha explicado en clase, y nos ocupamos de que los distintos componentes le hagan los avisos pertinentes para mantener actualizada la información del juego.

En particular, hemos de llevar la cuenta de la puntuación y de las vidas del jugador. Al empezar el juego, el jugador contará con 3 vidas y 0 puntos. Perderá una vida cada vez que se le escape una bola.

El componente `GameManager` tiene un método booleano público `PlayerLoseLife ()` que reduce el número de vidas, presenta por consola la cantidad de vidas restantes y devuelve `true` solo si el jugador sigue vivo. Este método deberá ser usado por el componente `LostBall` para determinar si *respawnea* la bola o no. En el caso de que ya no queden vidas, `LostBall` destruirá definitivamente la bola.

El componente `GameManager` también cuenta con un método público `void AddPoints (int amount)` para incrementar la puntuación, el cual será llamado por cualquier componente que deba informar de que el jugador ha ganado algún punto.

Necesitamos también llevar el control del número de ladrillos que hay en cada momento. Para ello, haremos lo siguiente:

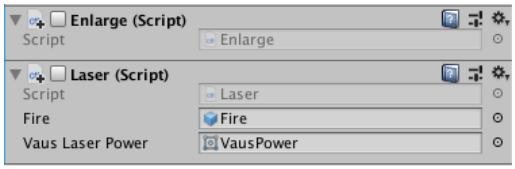
- Implementar un componente `Brick` para los ladrillos, que informará al `GameManager` del nacimiento (al inicio) o destrucción de un ladrillo (cuando ésta se produzca).
- Implementar dos métodos públicos en el `GameManager` que permitan procesar adecuadamente la información anterior: `void AddBrick ()` y `void BrickDestroyed ()`.

Cada vez que se cambia la información del juego, mostramos la información actualizada del juego por consola. ◇

Escena 2.7: Power-ups

Nombre de escena: `Assets/Scenes/test/7_PowerUps`

Debemos implementar los componentes `Enlarge` y `Laser` para los dos tipos de *power-ups*, y asignárselos al jugador. Los componentes estarán desactivados al principio de la partida. Para probarlos, los activaremos/desactivaremos desde el editor, recordando que puede haber un solo *power-up* activo de forma simultánea.

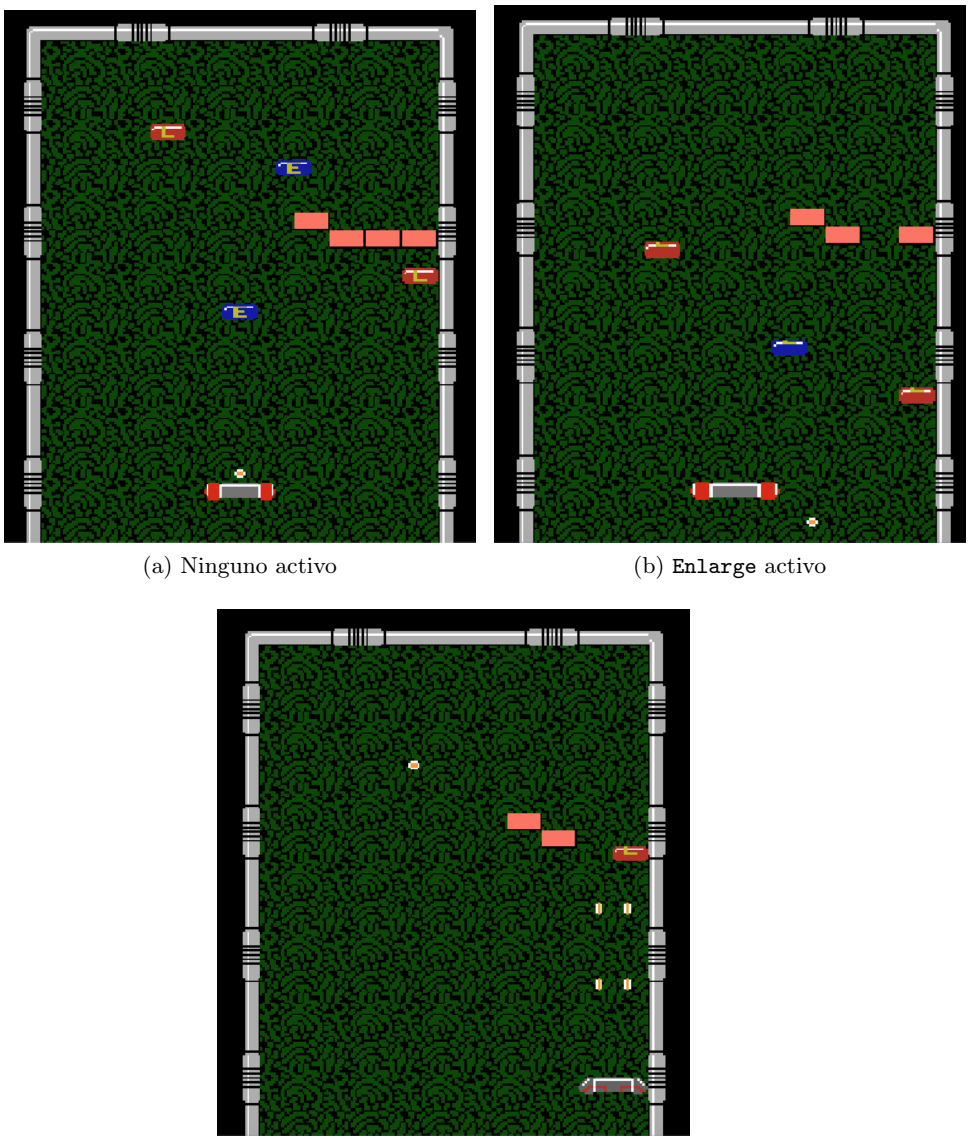


Cuando `Enlarge` está activo, el jugador ve incrementada su anchura en un 25 %, como puede apreciarse en la figura 2.1b. Al desactivarse, recupera su tamaño normal. Como es natural, cuando sea más ancho, no debe atravesar los bordes ni la bola, ni tampoco incrementar el tamaño de la bola.

En cuanto a `Laser`, al activarse cambia el *sprite* del jugador y cada vez que el usuario pulsa el botón de salto (barra espaciadora), se instancia un objeto predefinido `Fire` saliendo de la pala, a modo de balas, como puede observarse en la figura 2.1c. Al desactivarse, recupera su *sprite* habitual. Los nuevos *sprites* se pueden obtener de la *sprite sheet* `NES - Arkanoid - Vaus.png`.

Los *game objects* para los *power-ups* han de estar animados, a partir de la *sprite sheet* `NES - Arkanoid - Powerups.png`. Usamos, para ello, los *sprites* de la cuarta y quinta fila (E y L).

Para el movimiento descendente de los *power-ups*, así como para el movimiento ascendente de las balas, implementamos un componente `SetInitialSpeed` que permite configurar la velocidad inicial del objeto físico al que pertenece.

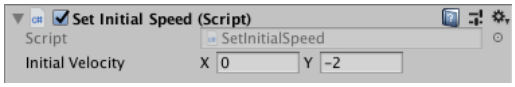


(a) Ninguno activo

(b) `Enlarge` activo

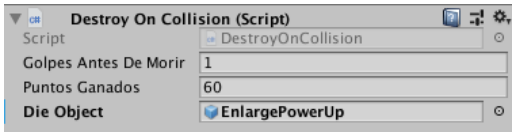
(c) `Laser` activo

Figura 2.1: Escena `7_PowerUps`



La activación y desactivación de los *power-ups* se gestiona en el jugador. Lo hacemos con la ayuda de los componentes `ActivatePowerUpOnCollision` y `PowerUpManager` proporcionados. El primero se debe asignar a los *power-ups*, configurándolo con el nombre del componente asociado, y el segundo al jugador, el cual debe pertenecer a la capa física `Player`, única capa con la que deben colisionar los objetos de la capa física de los *power-ups*.

Por último, extendemos el componente `DestroyOnCollision` para que se pueda configurar de manera que instancie un objeto en el momento de la colisión. Utilizamos esta nueva característica para construir los ladrillos que liberan *power-ups* al destruirse.



Nótese que este componente puede utilizarse igualmente para la destrucción de *power-ups* y balas, en caso de colisión con la pala o con los bordes del tablero, respectivamente.

◇

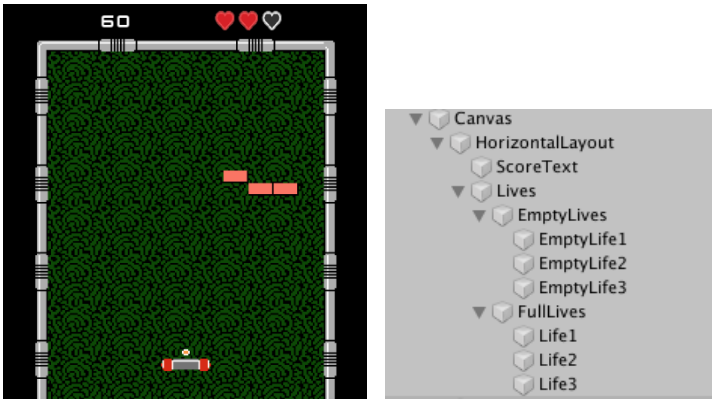
Escena 2.8: UI

Nombre de escena: `Assets/Scenes/test/8_UI`

Crearemos una interfaz de usuario (*UI o HUD*) para nuestro juego, compuesta por un texto para reflejar la puntuación y un conjunto de imágenes para representar las vidas del jugador.

Para diseñar la UI es conveniente fijar un tamaño del mundo, de manera que el tamaño del texto y el de las imágenes que aparezcan pueda diseñarse de acuerdo al tamaño elegido. Por lo tanto, empezaremos creando un *aspect ratio* nuevo `RompeMoldes` de 480×512.

Aunque la organización final de objetos del canvas se muestra en la figura 2.2b, crearemos la interfaz paso a paso, siguiendo las indicaciones que se ofrecen a continuación.



(a) Aspecto del HUD (b) Organización del canvas

Figura 2.2: HUD del juego

Creamos un canvas y un objeto vacío como hijo suyo. A este objeto vacío le añadimos un componente de *Horizontal Layout* que configuramos de forma centrada en la parte superior del canvas, independientemente de las dimensiones de la pantalla. Le asignamos altura 50, ancho 300 y elegimos alineación centrada en el medio para sus futuros hijos.

Creamos un objecto de UI de tipo `Text`, al que llamamos `ScoreText`, y lo hacemos hijo del objeto `HorizontalLayout`. Usamos la fuente proporcionada `kenvector_future`, en color blanco, y configuramos tamaño y alineación para que quede como en el ejemplo.

Para las vidas vamos a crear dos filas de imágenes superpuestas. La fila de imágenes con corazones vacíos se dibuja debajo y la fila de imágenes con corazones llenos se dibuja encima. De este modo, al ocultar cada uno de los corazones llenos se verán los corazones vacíos. Para ello:

1. Creamos un objeto vacío `Lives`, hijo de `HorizontalLayout`, para el conjunto de vidas, que servirá para colocar las vidas en el lugar adecuado y para que mantengan su posición al modificar el tamaño de la pantalla.

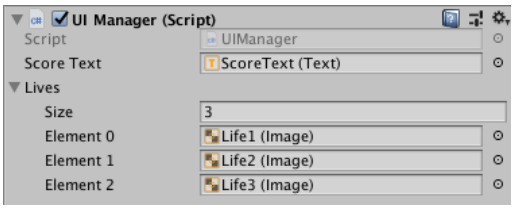
2. Creamos otro objeto vacío `EmptyLives`, hijo de `Lives`, con un componente de *layout* para organizar los objetos horizontalmente, que ajustaremos para la visualización satisfactoria de los elementos que agrupará. Una de los ajustes necesarios consiste en desmarcar las opciones por defecto de expansión automática forzosa de los hijos.

3. Creamos tres imágenes, hijas de `EmptyLives`, usando como imagen el *sprite* del corazón vacío.

4. Hacemos lo mismo con los corazones llenos `FullLives`, aprovechando el trabajo anterior.

La comunicación para actualizar la UI se hará siempre a través del *GameManager*. Los objetos de la escena seguirán avisando al `GameManager` de los cambios relevantes para que pueda mantener actualizada la información del juego, y éste se comunicará con el componente responsable de actualizar la interfaz, del que tendrá una referencia, conseguida tal y como hemos explicado en clase.

Creamos el componente `UIManager` y lo añadimos al *game object* `Canvas`. Debe presentarse al `GameManager` como *UIManager* de la escena, al cargarse ésta. A este componente le añadiremos las referencias necesarias para la gestión de la UI de cada escena, y éstas serán configurables desde el editor. En particular, necesitamos una referencia al texto de la puntuación, así como un array con las imágenes de cada uno de los corazones llenos.



Creamos un método `UpdateScore (int points)` para actualizar el texto que muestra la puntuación, el cual será llamado por el `GameManager` en el momento oportuno.

Por último, añadimos un método `LifeLost` para desactivar la imagen de una de las vidas, de cara a reflejar así la pérdida de una vida.

◇

Escena 2.9: Escena menú

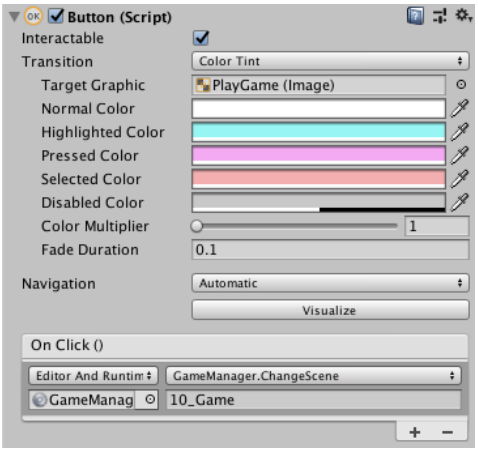
Nombre de escena: `Assets/Scenes/test/9_Menu`

Creamos una nueva escena que hará las veces de menú. Debe contener un canvas con, al menos, un texto `Title` y un botón `PlayGame` con color cambiante, dependiendo de si está en estado normal, destacado, pulsado o seleccionado.

La escena incluirá también una imagen de fondo y una animación creada a partir de *sprites*, además de un *GameManager*. Dado que esta escena no es una evolución de las anteriores, es conveniente que hagamos *prefab* al *game object GameManager* de la anterior escena.

Añadimos al `GameManager` un nuevo método `ChangeScene(string sceneName)`, que hace uso del `SceneManager` para cambiar de escena. Será necesario añadir las escenas a los *Build Settings* para poder hacer el cambio de escenas.

Configuramos el botón para que, al pulsarlo, se llame a este método, con la escena destino explicitada, de manera que esta acción produzca el cambio de escena. Para lograr la configuración correcta, conviene que seleccionemos al *prefab GameManager* de nuestra carpeta de recursos.



En la escena de juego habrá que modificar los métodos convenientes para que se vuelva al menú cuando el jugador gane o pierda, es decir, cuando se hayan destruido todos los ladrillos o se hayan perdido todas las vidas. También habrá que asegurarse de que se inicialicen las variables que reflejan el estado del juego.

◇

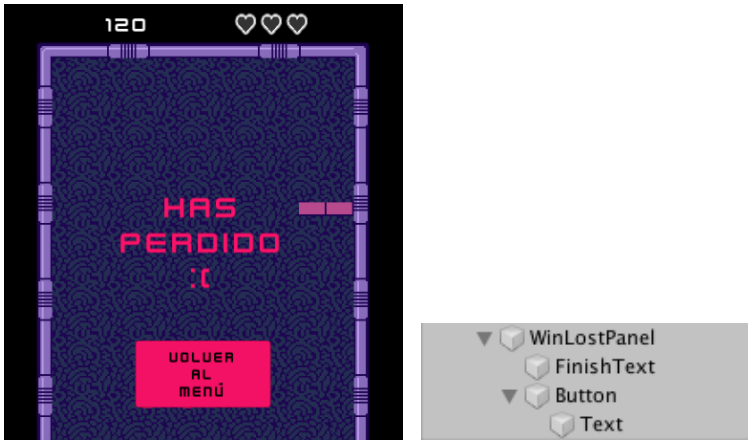
Escena 2.10: Ganar o perder

Nombre de escena: `Assets/Scenes/test/10_Game`

En el estado actual de la implementación, el juego vuelve de manera abrupta al menú, sin presentar el típico mensaje de final de partida. Para remediarlo, crearemos un panel como el de la figura 2.3a que mostrará el resultado final del juego ("Has perdido" o "¡Has ganado!") y un botón para volver al menú.

Tanto la visibilidad del panel, como el texto concreto que aparecerá, serán gestionados por el `UIManager`, que dispondrá de un método `FinishGame(bool playerWins)` que servirá para que el `GameManager` le avise de que se ha acabado el juego. En la figura 2.4 se muestran todas las referencias que necesita el `UIManager`.

◇



(a) Posible final (b) Panel en el canvas

Figura 2.3: Panel con posible final de partida

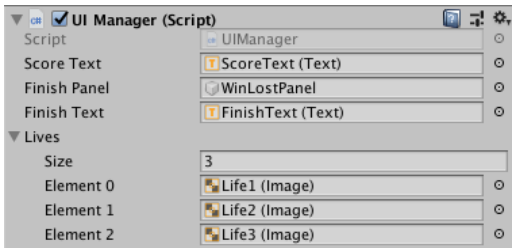


Figura 2.4: `UIManager` completo

2.3. Nuestro juego final

Nombres de escenas: `Assets/Scenes/Menu`, `Assets/Scenes/Level1` y `Assets/Scenes/Level2`

A estas alturas, ya estamos en condiciones de unir todas las piezas y diseñar nuestro juego, el cual estará compuesto por un menú y dos niveles. Necesitamos, por lo tanto, diseñar dos niveles del juego, ajustar para cada ladrillo la puntuación, los golpes necesarios para destruirlo, y si libera o no algún *power-up*, etcétera.

Para lograr que todo cuadre, nos falta tener en cuenta algunos detalles importantes, que surgen del hecho de no haber considerado hasta ahora el paso automático al segundo nivel cuando se gana el primero. Esto solamente afecta a la implementación del `GameManager` y del `UIManager`.

El que todos los ladrillos se acaben mientras estamos jugando ya no asegura ganar la partida, sino haber terminado un nivel con éxito. Necesitamos distinguir en el `GameManager` qué acción debe realizarse tras la

consecución del nivel. Implementamos para ello un método `LevelFinished (bool playerWins)` en el `GameManager`, el cual será llamado desde el propio `GameManager` cuando se detecte que no quedan vidas o que no quedan ladrillos en un nivel.

- Si se ha acabado la partida, se comunicará con el `UIManager`, para informarle de si hemos ganado o no y que éste actualice la UI convenientemente.
- Si se ha finalizado el primer nivel, se realizará un cambio de escena al siguiente nivel.
- También es el responsable de realizar las inicializaciones de variables necesarias por si se vuelve a jugar, una vez que el juego ha terminado.

Al cargarse la escena del nuevo nivel, se crea un nuevo *UIManager*, el cual ha de mostrar la información correcta. Para ello, el `GameManager` debe pasarle la información al nuevo `UIManager` en el momento en el que éste se presenta, es decir, en el método `SetUIManager` aprovechará el saludo para comunicarle la puntuación y las vidas restantes. Para ello utilizará:

- El método ya implementado `UpdateScore`
- Y un nuevo método `RemainingLives (int remainingLives)` del `UIManager` que se ocupará de que las vidas indicadas se muestren correctamente

Esto nos permite entregar un juego completo y funcional, con un diseño personalizado de las escenas pedidas, como el ejemplo mostrado en la figura 2.5, que reproduce la distribución de ladrillos de los dos primeros niveles del Arkanoid original.

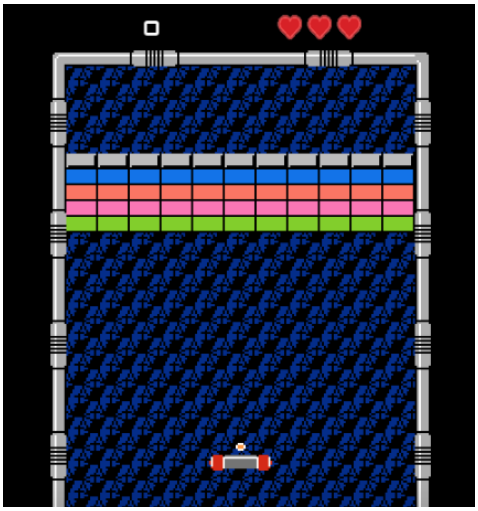
2.4. Entrega

La entrega de la práctica completa (10 escenas de prueba y 3 escenas de juego) se realizará respetando las indicaciones de entrega, que añadimos al presente documento.

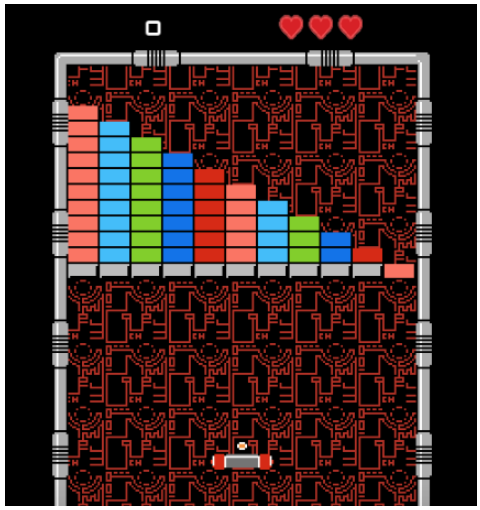
Fecha límite: Viernes 20 de diciembre a las 20:02 horas

Durante las clases restantes aclararemos las posibles dudas que puedan surgir sobre la interpretación del enunciado.

(a) Posible menú



(b) Posible nivel 1



(c) Posible nivel 2

Figura 2.5: Escenas de un posible juego completo

Normas de entrega

Entrega de prácticas

- Cada práctica tendrá una fecha límite de entrega. No se admitirán prácticas entregadas fuera del plazo establecido.
- Las prácticas se entregarán exclusivamente a través del Campus Virtual, utilizando el mecanismo de entrega de prácticas.
- Únicamente un miembro del grupo de prácticas deberá encargarse de realizar la entrega. En el caso de que ambos componentes efectuasen el envío de la práctica, corregiremos aleatoriamente uno de ellos.
- Para realizar la entrega, será necesario crear un directorio (carpeta), llamado `GrupoNN`, siendo `NN` el número del grupo al que pertenece (con dos dígitos: 01, 02, ..., 40).
- En el directorio `GrupoNN` se incluirá un fichero `alumnos.txt`, en el que se indicará el nombre de los componentes del grupo que hayan intervenido en el desarrollo de la práctica.
- Además, se incluirán en la carpeta `GrupoNN` **únicamente** las carpetas `Assets`, `Packages` y `ProjectSettings` del proyecto de Unity.
- Se enviará un único archivo comprimido con formato .zip[1], llamado `GrupoNN.zip`, resultado de la compresión del directorio (carpeta) `GrupoNN`.
- No se tolerarán plagios[2] ni se permitirá hacer las prácticas entre varios grupos.

[1] Para comprimir el archivo puedes utilizar cualquiera de los compresores disponibles en el laboratorio con la condición de que generen ficheros zip. Ten en cuenta que el programa 7-zip puede generar ficheros .zip, pero para eso hay que indicar explícitamente que se desea ese formato.

[2] Los alumnos participantes en un caso de copia, de forma activa o pasiva, serán remitidos a la Comisión de Copias de la Facultad, la cual decidirá las sanciones a imponer.