Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Informática

Grado de Ingeniería Informática

▪ Proyecto Fin de Grado ▪

Análisis de técnicas de animación con Unity.

Iñigo Sanz

Noviembre 2017

informatika fakultatea
facultad de informática

Resumen

Este proyecto tiene como finalidad crear una serie de ejemplos que permitan implementar y probar distintas mecánicas de animación usando el motor de juego Unity.

Se definirá una lista de distintas mecánicas de juego, que se programarán en Unity usando C# con ejemplos de juego que muestren las mecánicas en funcionamiento. Cada juego tendrá sus propias mecánicas, y se definirá su alcance y sus componentes. Se describirá su uso y se demostrará su correcto comportamiento con versiones ejecutables y vídeos.

Índice

Lista de figuras

Lista de scripts

Nota: Los scripts de este documento son en su mayoría fragmentos adaptados de los archivos completos, seleccionados para mostrar solo lo relevante a su sección y facilitar su comprensión.

Si se desea descargar los scripts completos, el archivo “Contenido.zip” de cada juego en el enlace goo.gl/eVadM8 contendrá los scripts de cada ejemplo en su carpeta *Assets*.

1

Introducción

En este capítulo se hablará del estado actual de los videojuegos, tanto de las últimas tendencias en la industria como de las numerosas herramientas disponibles para desarrollarlos, y se definirán los objetivos y el alcance del proyecto.

1.1. ¿Qué son los videojuegos?

Los videojuegos, hace no demasiados años afición de unos pocos, son hoy en día uno de los medios de entretenimiento más populares del mundo, compitiendo y a veces superando la industria del cine y de la música [1].

A medida que los jóvenes jugadores de hace unos años se hacen mayores, el espectro de potenciales jugadores crece cada día más. Incitados también por la evolución de los smartphones y la popularización de videojuegos sociales como Candy Crush o Pokémon GO, son cada vez más los adultos y mujeres que juegan a algún videojuego [2].

A causa de esta popularización y al rápido crecimiento de la industria, uno de los grupos más beneficiados en el sector ha sido el de los videojuegos independientes.

1.2. Mercado actual del videojuego

Debido al auge del mercado de los videojuegos y a las últimas facilidades para su distribución gracias a las plataformas digitales, se han creado numerosos estudios independientes intentando hacerse un hueco en la industria.

Plataformas como Steam Direct (sustituyendo al recientemente clausurado Steam Greenlight), Kickstarter, Itch.io, etc. ofrecen medios para la financiación y publicación de videojuegos a grupos de aficionados y pequeños estudios que tendrían dificultades para conseguirlo de otro modo.

La mayoría de estos estudios *indie* se centran en videojuegos "retro", con un estilo *pixel art*, música de 8 bits, juegos *roguelike*… aunque hay estudios que destacan por su alta calidad a pesar de contar con reducidos recursos, tales como Abzu o Firewatch.

Esto es posible gracias a motores gráficos como Unreal Engine 4 o Unity 3D, herramientas muy potentes con las que desarrollar videojuegos con un coste muy asequible, o incluso sin coste alguno (dependiendo de las características del estudio) [3].

Entre estos dos motores, Unreal es el más potente gráficamente y consigue que cualquier escena simple luzca genial con su increíble iluminación, pero Unity ofrece más herramientas para trabajar en 2D y dispone de mucha más documentación, por lo que me decanté por Unity para llevar a cabo mi proyecto.

1.3. Motivación y objetivos

Antes de elegir la carrera de Ingeniería Informática me debatía entre dos opciones principales: la propia Ingeniería Informática, y el Desarrollo de Videojuegos. Tras muchas vueltas me decanté por la ingeniería, ya que abarcaba más campos, y la vi más apropiada

para alguien que todavía no tenía las ideas del todo claras. Además, siempre tendría la opción de especializarme más adelante, y considero importante crear una buena base de conocimiento general, y la ingeniería ofrecía un mejor programa para ello.

Sin embargo, aún me quedó latente el interés por el desarrollo de videojuegos, y vi este proyecto como una oportunidad para adentrarme un poco en este campo y saciar mi curiosidad. Aplicando los conocimientos adquiridos en la carrera en un campo nuevo para mí, he decidido empezar a crear mis primeros videojuegos.

El objetivo de este proyecto es el de crear varios ejemplos diferentes, en los que probar mecánicas variadas y conocer el funcionamiento del motor de juego Unity. Me propongo crear al menos un juego en 2D y otro en 3D para ordenador, en los que probar tanto la animación mediante físicas en 3D como la animación por imágenes o *sprites* en 2D.

No pretendo crear juegos con mucho contenido (respecto a niveles), sino que me centraré en implementar el mayor número de mecánicas distintas posibles, en dos ejemplos que permitan comprobar su funcionamiento. Una vez las mecánicas funcionen, el crear niveles adicionales supone un tipo de trabajo que no considero que me pueda aportar tanto como el desarrollo de las mecánicas, ya que es de un carácter más mecánico o artístico.

1.4. Estructura de los ejemplos

Todo el contenido de los ejemplos tratados en los capítulos 3 y 4 se podrá encontrar en el siguiente enlace de *Google Drive:* goo.gl/eVadM8

Cada uno de ellos se hallará en un directorio distinto, y éstos contendrán las siguientes carpetas:

- **Ejecutable:** esta carpeta contiene la "build" del juego, o versión jugable. Para probarlo no hará falta instalar nada; basta con descargarse el archivo *Juego.zip*, descomprimir su contenido y ejecutar el archivo *.exe* dentro del mismo (no hará falta conceder permisos de administrador). Junto al ejecutable habrá un documento PDF con las instrucciones de uso.
- **Source:** en esta carpeta se guardan todos los componentes del juego. Dentro de *Contenido.*zip, la carpeta Assets contiene: los scripts C# que controlan el comportamiento del juego, los objetos, texturas y materiales para crear los elementos, *sprites* e imágenes, temas musicales, etc. El contenido se puede inspeccionar desde el explorador de Windows/Linux, o bien cargar el proyecto en Unity y hacerlo desde el mismo.

  Junto al archivo .zip hay accesible un vídeo para ver el juego en funcionamiento sin necesidad de descargar el ejecutable, y sus instrucciones de uso.

Nota: *Drive comprime antes los elementos seleccionados para una descarga conjunta. Debido al tamaño de los archivos ".zip", éstos es mejor descargarlos individualmente, o la descarga podría tardar mucho en prepararse. El resto de los archivos se pueden descargar conjuntamente.*

2

Recursos

Para llevar a cabo este proyecto han hecho falta diversos programas y herramientas además de Unity. A continuación se enumerarán estos recursos argumentando por qué se han escogido y para qué se han utilizado.

2.1. Software

2.1.1. Motor de juego:

El motor de videojuegos utilizado en este proyecto ha sido Unity 3D, en su versión 5.5.x. Este motor, originalmente de pago, pasó a ser gratuito en 2015 con el lanzamiento de Unity 5.0. Cualquier individual o empresa con beneficios menores a 100.000$/año puede usar y publicar juegos creados con Unity sin coste alguno. Existe una versión "Pro" de pago que ofrece más herramientas, pero esta no nos interesa.

Unity cuenta con una inmensa documentación y tutoriales online para ayudar a los recién iniciados, muy útiles a la hora de buscar ayuda. Cuenta también con un *plugin* para incorporar su documentación en Visual Studio, pero en este proyecto se ha utilizado Sublime Text para programar y se ha consultado la documentación en el navegador.

2.1.2. Edición de imagen:

Para la edición de imagen se han usado dos herramientas: Microsoft Paint para los recortes e imágenes de la memoria, y FireAlpaca para la edición de *sprites*.

El primero se ha escogido por su facilidad de uso para tareas sencillas, y el segundo por ser una alternativa gratuita a PhotoShop para la edición avanzada de los *sprites*, como recortar secciones de las figuras o mantener las transparencias.

2.1.3. Edición de audio:

Para la creación de los clips de audio utilizados en uno de los ejemplos, primero se ha grabado el sonido con un piano y la grabadora del móvil, y después se ha editado la música y los efectos utilizando el programa de software libre Audacity.

Con Audacity es sencillo "limpiar" el sonido pobre grabado con el móvil para conseguir una calidad mayor, por lo que me ofreció una solución económica a comprar un micrófono de calidad.

2.1.4. Captura de vídeo:

Para la grabación de las demos se ha escogido OBS. OBS es una capturadora de vídeo de código abierto ligera y eficaz, y se ha escogido por su facilidad de uso para grabar vídeos simples que demuestren los ejemplos en acción.

2.2. Lenguaje de programación

Como lenguaje de programación para este proyecto existían dos posibilidades: C# y JavaScript. Dado que el lenguaje más habitual para Unity es C# y en la carrera no he profundizado demasiado en JavaScript, me pareció adecuado aprovechar la oportunidad para aprender C#, ya que es uno de los lenguajes más populares para programar videojuegos. La sintaxis es parecida entre ambos, y no me ha resultado complicado adaptarme al nuevo lenguaje.

2.3. Formación previa

Antes de comenzar el proyecto, se siguió un tutorial oficial para dar los primeros pasos en Unity. En este tutorial, llamado “Roll-a-Ball” [4], se enseñan los pasos a seguir para crear un proyecto, objetos en el juego, etc. y nociones básicas sobre el funcionamiento de Unity.

Con la ayuda de este tutorial se implementaron las siguientes mecánicas:

- **Movimiento sencillo de una pelota:** mediante un script, al pulsar las teclas de dirección se aplicará una fuerza direccional a la pelota para que se mueva.
- **Cámara de seguimiento:** asignando la cámara como “hija” de la pelota, ésta se moverá junto con la pelota, manteniéndose siempre a la misma distancia e inclinación.
- **Coleccionables:** se crearon una serie de coleccionables cúbicos que flotarán sobre el tablero, y desaparecerán al contacto con la pelota.
- **Puntuación:** por cada coleccionable que se recoja, se añadirá un punto al jugador. Cuando se recojan todos los coleccionables, se mostrará un sencillo mensaje de victoria en el centro de la pantalla.

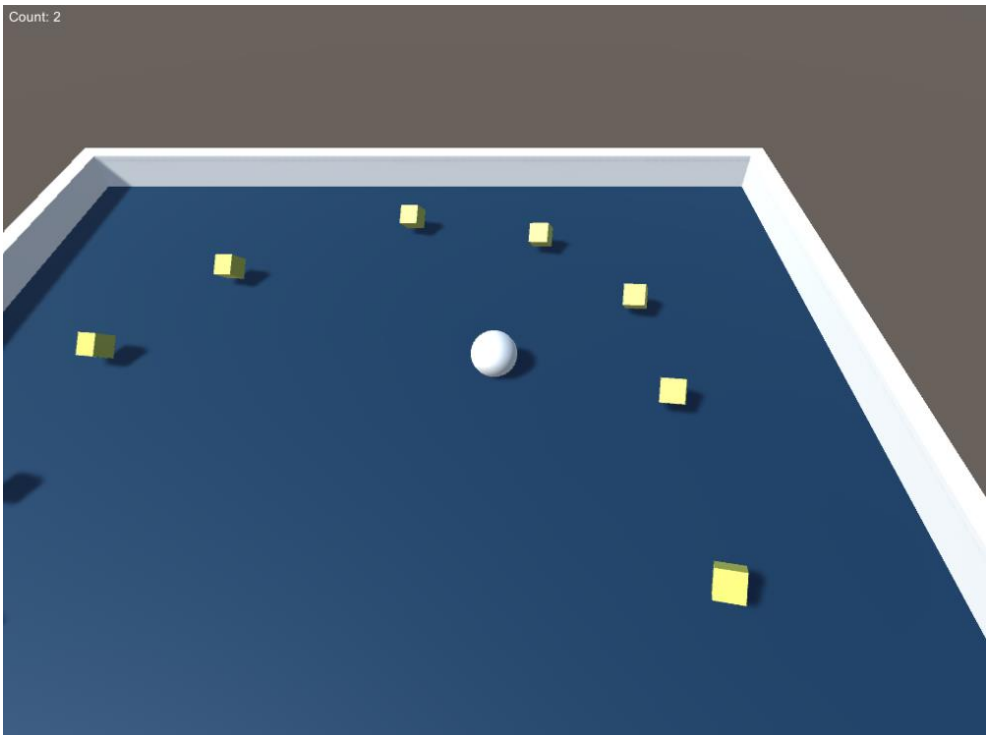


Figura 2.1: Juego Roll-a-Ball creado con el tutorial de Unity.

A medida que avanzaba con el proyecto he ido profundizando sobre los fundamentos del funcionamiento de Unity, y he resumido todo lo imprescindible en el anexo A. Se recomienda leer dicho anexo antes de seguir con los capítulos 3 y 4, ya que facilitará mucho su comprensión.

3

Keep Rolling

El objetivo del ejemplo Keep Rolling es expandir la base del tutorial Roll-a-Ball para dar los primeros pasos sin guía en Unity, y crear un juego en 3D más atractivo que el del tutorial.

En él se añadirán mecánicas interesantes como saltos, enemigos y un cronómetro para poner a prueba la habilidad del jugador. También se intentará hacer el juego más agradable a la vista.

En Keep Rolling el objetivo del jugador será recoger todas las monedas esparcidas por un tablero en el menor tiempo posible. Para ello, controlará una pelota sobre el tablero con obstáculos. Inclinando el tablero se podrá desplazar la pelota, y sortear los obstáculos. (Ver figura 3.1)

Para ver un ejemplo de su funcionamiento, acudir al vídeo del enlace goo.gl/J6J7Bo

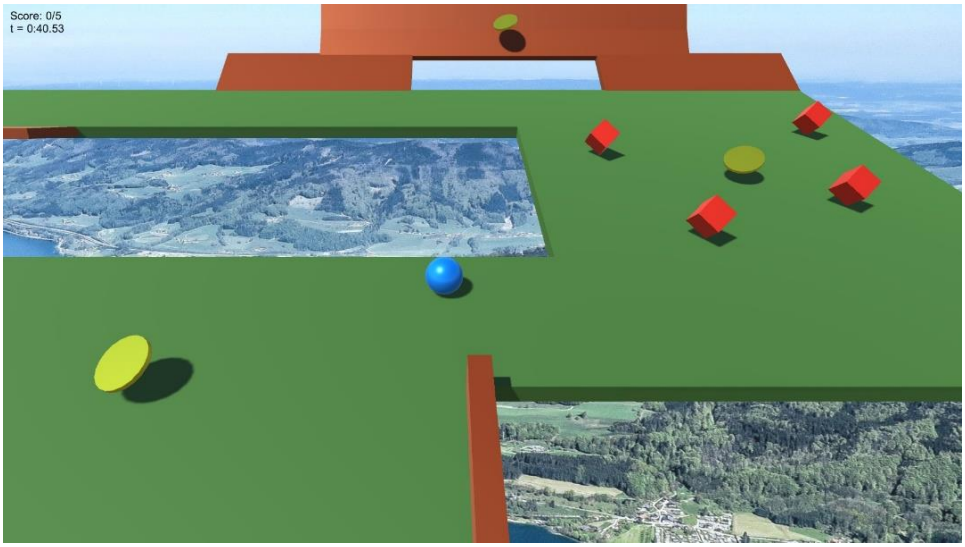


Figura 3.1: Escena del juego Keep Rolling.

3.1. Interfaz de usuario

Al ejecutar la aplicación, el programa mostrará un panel de selección donde podremos comenzar a jugar directamente o configurar varios parámetros. Si pulsamos el botón “Play!”, el juego comenzará con los ajustes que detecte automáticamente. Si no, podremos ajustar parámetros como la resolución de pantalla, la calidad de los gráficos o el modo de ventana.

Una vez comenzado el juego, se cargará la escena del juego y empezará a sonar la música de fondo. Desde este momento, los controles que tendremos a nuestra disposición serán los siguientes:

- *Flechas de dirección* del teclado o teclas *W*, *A*, *S* y *D*: con éstas inclinaremos el tablero, y con ello moveremos la pelota de un lado a otro, con el objetivo de conseguir todas las monedas para completar el nivel.
- *Barra espaciadora*: al pulsar esta tecla la pelota dará un salto Esto será necesario para saltar una brecha y así conseguir una de las monedas. Además, dado que el suelo generará fricción con la pelota, se avanzará más rápidamente por el aire. Esto da la opción a que un jugador experimentado sea capaz de conseguir tiempos más rápidos.

- *R*: si pulsamos la tecla *R* se reiniciará el juego y reseteará el cronómetro. Útil si buscamos mejorar nuestro récord y hemos empezado mal.
- *Esc*: al pulsar la tecla *Escape* se cerrará la aplicación.

Dentro del juego, utilizando estas opciones, el jugador deberá recoger todas las monedas esparcidas por el mapa en el menor tiempo posible. Estas monedas estarán distribuidas de forma que pongan a prueba la habilidad del jugador. El cronómetro se pondrá en marcha en cuanto pulsemos la primera tecla, y dará comienzo al juego.

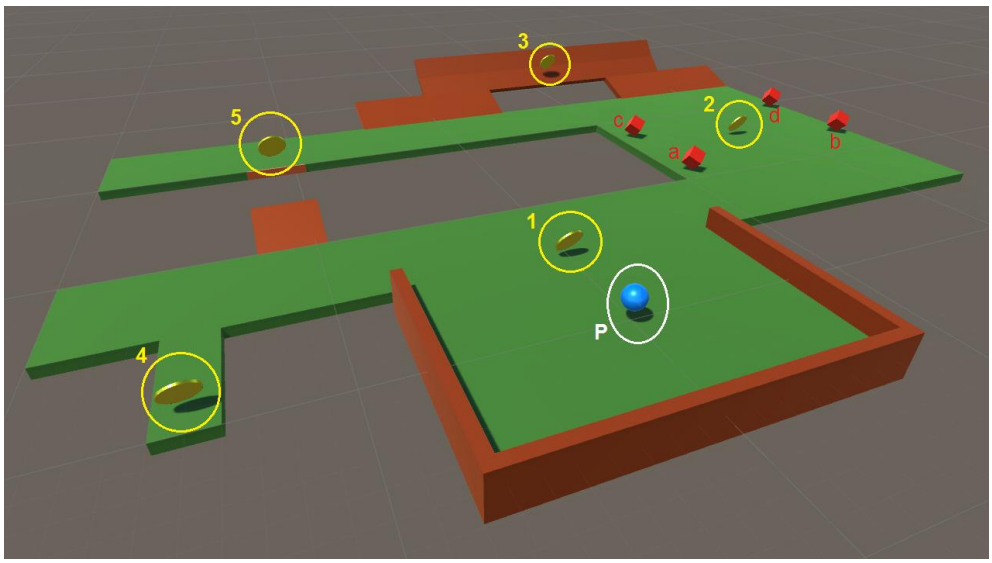


Figura 3.2: Disposición de los elementos del tablero.

La figura 3.2 muestra cinco monedas. La moneda 1 está delante de la posición inicial del jugador (*P*), para facilitar el primer paso del juego: cómo coger un coleccionable. Para ello, solo tendrá que tocarla pasando por encima. Para la moneda número 2, el jugador deberá esquivar los enemigos (cubos rojos *a*-*d*). Si es golpeado por uno de ellos o cae fuera del tablero, el programa colocará la pelota en una posición favorable y podrá seguir intentándolo, simplemente perderá algo de tiempo.

Los enemigos *a* y *b* se moverán más despacio que los *c* y *d*, por lo que el jugador deberá decidir el momento adecuado para pasar o ser ágil para no ser golpeado. Las monedas 3 y 4 están en terreno complicado (en una rampa y en un saliente, respectivamente), por lo que se deberá dominar el movimiento para conseguirlas sin caer. Para la última moneda, se deberá saltar el agujero que hay entre los dos lados con la ayuda de las rampas oscuras situadas en ambos extremos.

Las monedas se pueden recoger en el orden deseado. El jugador deberá decidir la ruta más adecuada para hacerlo en el menor tiempo posible.

3.2. Mecánicas del juego

Aunque este juego se haya desarrollado partiendo del tutorial oficial de Unity "Roll-a-Ball", Keep Rolling es una evolución de este juego, en la que se han explorado distintas opciones y añade las siguientes mecánicas:

- Modificar el tablero: se ha sustituido el plano simple por un tablero más complejo: con forma irregular, rampas, fosos, nuevos materiales, texturas, etc.
- Salto: el jugador podrá hacer saltar la pelota pulsando la barra espaciadora. Solo podrá saltar mientras ésta esté en contacto con alguna superficie.
- Inclinación del tablero: para simular unas físicas más realistas, se ha creado un efecto de cámara que hará parecer que se inclina el tablero para mover la pelota, al estilo *Super Monkey Ball* [5]. Sin embargo, al igual que en este juego o clásicos como *Super Mario* [6], se mantendrá el control de la pelota incluso en el aire, dando así un control más responsivo al jugador.
- Caídas y *respawn*: ahora que es posible saltar y se han eliminado los muros, el jugador podrá salirse del tablero. Si se cae, la pelota reaparecerá (o *respawn*-eará) en una posición segura y el jugador podrá volver a intentarlo. En este juego no se puede perder la partida, y se podrá seguir intentándolo tantas veces como haga falta hasta conseguir todas las monedas.
- Enemigos: se han añadido enemigos móviles que eliminen al jugador si colisionan con la pelota. Al igual que con las caídas, la pelota reaparecerá y se podrá seguir jugando.
- Puntos de control: en vez de hacer reaparecer la pelota siempre en la misma posición, se ha creado un sistema de puntos de control. Al regenerarse si se es eliminado, la pelota reaparecerá en el último punto de control por el que haya pasado.
- Cronómetro: un contador sencillo que mide el tiempo que tarda el jugador en recoger todas las monedas. Se puede ver de modo continuo en la parte superior izquierda de la pantalla, y en cuanto se recogen todas las monedas se detiene y se muestra en el centro.
- Música / Efectos sonoros: se ha creado e introducido un tema musical de fondo, y un indicador sonoro que suena al recoger una moneda.

Estas mecánicas son las más básicas de un juego de plataformas. Disponiendo del tiempo necesario, sería sencillo crear niveles adicionales con dificultad creciente.

3.3. Desarrollo de las mecánicas

3.3.1. Modificar el tablero:

Para modificar el tablero, se han creado varios objetos adicionales en el propio editor, creando una combinación de elementos que forman el nuevo tablero, que se puede ver en la figura 3.2. Se ha sustituido la forma cuadrada por una más irregular e interesante. Sólo se han mantenido los muros en la zona inicial, y se han añadido superficies empinadas. El terreno liso se ha pintado de un verde mate, y las paredes y rampas de marrón, para facilitar diferenciarlas del terreno.

Se ha cambiado la forma de los coleccionables por monedas, y se les ha aplicado un material amarillo metálico para que llamen más la atención del jugador. La pelota del jugador se ha pintado de un azul brillante más vivo.

3.3.2. Salto:

La mecánica de salto es parte de un procedimiento que se ejecuta cada fotograma o *frame*. Se comprueba si el jugador ha pulsado la tecla *espacio*, y si la pelota aún está en el suelo. Si se cumplen ambas condiciones, se le aplicará una intensa fuerza vertical a la pelota durante este *frame*, causando que la pelota salte hacia arriba.

La variable de control de "en el aire" (*onAir*) se hará *true* en cuanto se salte. Cuando la pelota colisione con alguna superficie, a esta variable se le volverá a asignar el valor *false*, y se podrá volver a saltar.

El script 3.1 a continuación contiene los fragmentos relevantes a esta mecánica. Se ha extraído del script "PlayerController.cs".

```
public float speed;        // Velocidad de la pelota

private Rigidbody rb;      // Cuerpo "físico" de la pelota
private bool onAir;        // bool para controlar si está en el aire
...

// Este procedimiento es llamado cada frame
void FixedUpdate() {
  ...
  // Si se pulsa "espacio" y no está en el aire, saltará
  if (Input.GetKey(KeyCode.Space) && !onAir){
    Vector3 jump = new Vector3(0, 30, 0);
    rb.AddForce(jump * speed);
    onAir = true;
  }
  ...
}

// Este procedimiento es llamado al caer y colisionar
void OnCollisionEnter(Collision other) {
  onAir = false;
  ...
}
```

Script 3.1: Fragmento del script PlayerController.cs encargado del control del salto.

3.3.3. Inclinación del tablero:

La pelota en este juego se mueve porque se aplican fuerzas sobre ella, al igual que en el ejemplo del tutorial Roll-a-Ball. Si se pulsa la tecla derecha, se le aplica una fuerza horizontal para que ésta se desplace, y la cámara la seguiría a una distancia constante.

Con el fin de conseguir una sensación más realista, se ha programado un movimiento de cámara que simula una inclinación del tablero. Gracias a esto, parece que es el tablero el que se inclina para "dejar caer" la pelota hacia un lado u otro. Se consigue así un efecto parecido al del juego *Super Monkey Ball*, con una mecánica similar.

Tomando las teclas de movimiento como *input*, la cámara se inclinará ligeramente en el sentido opuesto sobre un eje horizontal. Por ejemplo, si pulsamos la tecla izquierda, la cámara cabeceará hacia la derecha. El tablero seguirá fijo, pero en el juego parecerá que éste se ha inclinado hacia la izquierda (ver figuras 3.3 y 3.4). Para reforzar la sensación, se ha añadido una imagen de fondo que seguirá el mismo movimiento que la cámara, junto con la iluminación global.

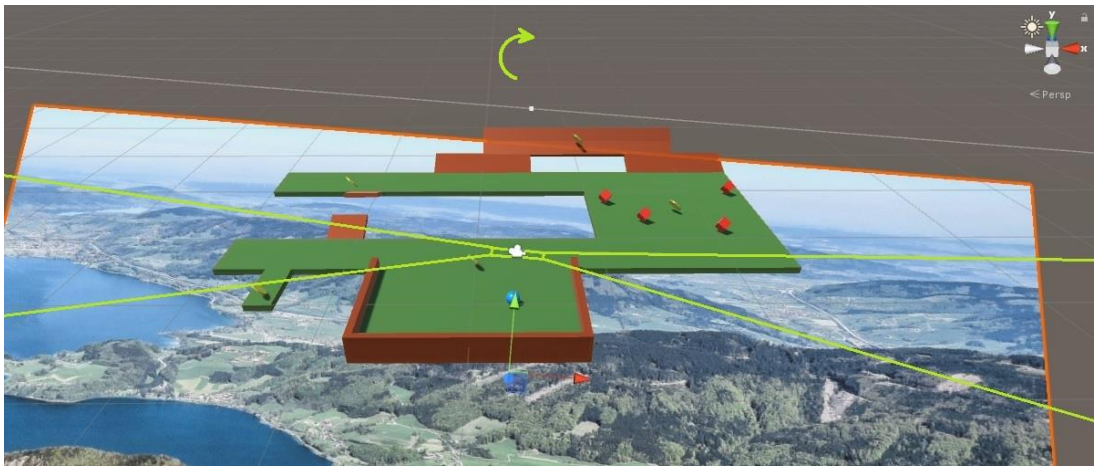


Figura 3.3: Rotación de la cámara (color lima) al pulsar la tecla izquierda.

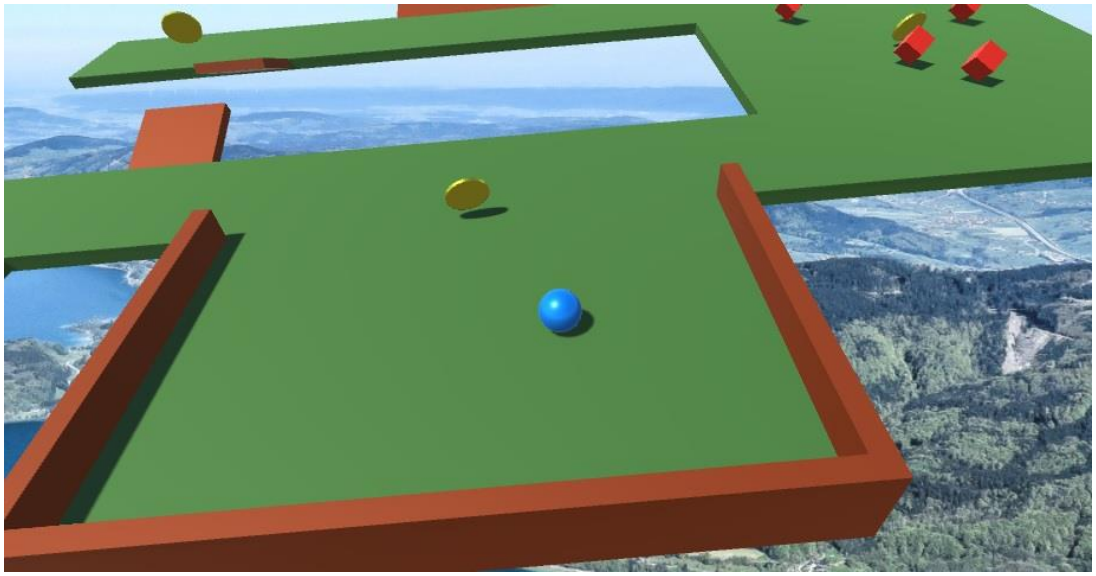


Figura 3.4: Resultado de la inclinación de cámara, con movimiento hacia la izquierda.

Para el movimiento hacia adelante y hacia atrás, además de la inclinación de la cámara también se altera la distancia entre la pelota y la cámara (*offset*), para que la pelota siempre esté situada en el centro de la pantalla.

El script 3.2 a continuación, "CameraController.cs", se encarga de manipular la cámara. La iluminación global y el fondo están emparentados con la cámara, por lo que este script también los controlará indirectamente, reforzando el efecto.

```
public class CameraController : MonoBehaviour {
  public GameObject player;      // Objeto de la pelota
  public Vector3 tiltOffset;     // Diferencia de inclinación
  public int tiltAngle;          // Ángulo de inclinación

  private Vector3 offset; // Distancia entre la pelota y la cámara
  private float tiltX;    // Inclinación vertical de la cámara
  private float tiltZ;    // Inclinación horizontal de la cámara

  // Se calcula el offset entre la cámara y la pelota
  void Start() {
    offset = transform.position - player.transform.position;
  }

  // Este procedimiento es llamado cada frame
  void LateUpdate() {
    tiltX = -tiltAngle * Input.GetAxis("Vertical");
    tiltZ = tiltAngle * Input.GetAxis("Horizontal");
    transform.eulerAngles = new Vector3(45+tiltX, 0, tiltZ);

    transform.position = player.transform.position + offset
                         + tiltX * tiltOffset;
  }
}
```

Script 3.2: Script controlador de la cámara, CameraController.cs

La razón principal por la que se ha decidido crear un movimiento de cámara en lugar de la propia inclinación del tablero como físicas es que uno de los principios fundamentales para que un juego "siente bien" es que siempre reaccione a tus comandos.

En juegos como el clásico *Super Mario Bros* o el mencionado *Super Monkey Ball*, el jugador también puede cambiar su trayectoria mientras está en el aire. Esta posibilidad le resta realismo, pero hace que sintamos que el juego responde mejor a nuestras órdenes, y cuando estamos a punto de caer del tablero apretando la tecla opuesta con todas nuestras fuerzas, salvar una situación que parecía perdida da una sensación de satisfacción.

Esta acción no sería posible si se controlara la pelota con una inclinación real del tablero, ya que no tendría efecto sobre ella en el aire.

3.3.4. Caídas y *respawn*:

El juego original impedía que la pelota saliese del tablero. Ahora el jugador puede caerse de él, y quedaría suspendido en el aire en una caída sin fin. Por eso, la tercera mecánica añadida ha sido detectar la "muerte por caída", y regenerar (*respawn*-ear) al jugador.

Para ello, se ha añadido una amplia superficie invisible unas unidades por debajo del tablero, etiquetada como "Enemigo". En cuanto la pelota colisiona con esta superficie, la detecta como enemigo y vuelve a su posición inicial, tras resetear su velocidad lineal y angular en los tres ejes para deshacerse de la inercia.

```
// Este procedimiento es llamado al colisionar con cualquier objeto
void OnCollisionEnter(Collision other) {
  ...
  // Si el otro objeto es el fondo (o un enemigo), regenerar
  if (other.gameObject.CompareTag("Enemy")) {
    Respawn();
  }
}

// Regenera la pelota en el último punto de control
void Respawn() {
  rb.velocity = Vector3.zero;                 // Eliminar inercia
  rb.angularVelocity = Vector3.zero;
  transform.position = GameVariables.spawn;   // Regenerar
}
```

Script 3.3: funciones encargadas del *respawn* en el script PlayerController.cs

El funcionamiento de los puntos de control almacenados en *GameVariables* se explicará en detalle más adelante, en el apartado 3.3.6, Puntos de control.

3.3.5. Enemigos:

Para crear los enemigos, se han creado unos cubos que se desplazarían en línea recta siguiendo un patrón. Se les ha asignado el color rojo para representar el peligro.

Con su script controlador, se les dará una velocidad positiva o negativa en el eje X dependiendo de su posición. En cuanto lleguen al límite de su recorrido, cambiarán su dirección y volverán a recorrer el camino realizado.

Los dos cubos más al sur se moverán más despacio que los dos situados más al norte, para evitar que vayan todos al mismo ritmo y aumentar así ligeramente el nivel de desafío. Al igual que la superficie bajo el tablero, llevarán la etiqueta de "Enemigo", y si tocan la pelota ésta reaparecerá en su posición correspondiente (*spawn*).

```
public class EnemyController : MonoBehaviour {
  public float speed;     // Velocidad del enemigo

  private Rigidbody rb;   // Cuerpo "físico" de los enemigos
  private int direction;  // Dirección actual del enemigo

  // Inicializar las variables
  void Start() {
    rb = GetComponent<Rigidbody>();
    direction = 1;
  }

  // Este procedimiento es llamado cada frame
  void Update(){
    if (transform.position[0] < 11) {
      direction = 1;      // Se moverán hacia la derecha
    } else if (transform.position[0] > 24) {
```

```
        direction = -1;      // Se moverán hacia la izquierda
      }

      rb.velocity = new Vector3(direction * speed, 0.0f, 0.0f);
    }
  }
```

Script 3.4: Script controlador de los enemigos, EnemyController.cs

3.3.6. Puntos de control:

Para que el jugador no tenga que reaparecer siempre al inicio del nivel aunque haya sido eliminado en el otro extremo, es esencial disponer de puntos de control distribuidos por el tablero. Estos puntos de control serán cubos invisibles con un *collider* o colisionador del tipo *trigger* (an. A.2), que se activarán cuando la pelota entre en ellos. (Ver figura 3.4)

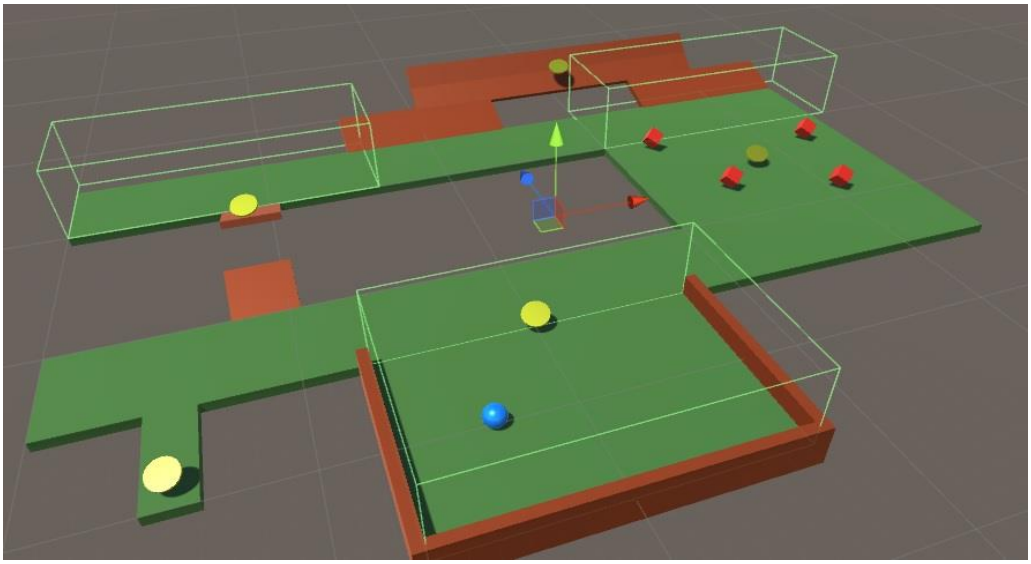


Figura 3.4: disposición de los puntos de control en el tablero.

Para esta mecánica, se han creado dos nuevas clases. La primera, "Checkpoint.cs" (ver script 3.5), es un script que se asignará a cada punto de control, y guardará su posición en una variable global cuando la pelota pase por el punto de control.

```
public class Checkpoint : MonoBehaviour {
  // Este procedimiento es llamado si el jugador entra en el punto
  void OnTriggerEnter(Collider other) {
    // Si la pelota pasa por el punto
    if (other.gameObject.CompareTag("Player")) {
      // Guardar su posición como último punto de control
      GameVariables.spawn = transform.position;
    }
  }
}
```

Script 3.5: Checkpoint.cs, encargado de guardar su posición en la variable global.

La segunda clase creada ha sido GameVariables.cs (ver script 3.6). Cuando el jugador sea eliminado, reaparecerá en la posición que esta clase guarda.

```
// Variables globales del juego
public static class GameVariables {
  public static Vector3 spawn;   // Último punto de control (x,y,z)
}
```

Script 3.6: GameVariables.cs, creada para almacenar variables globales.

La única función de esta clase es almacenar variables globales separadas del PlayerController. De momento solo guarda el actual *spawn* del jugador, pero en caso de añadir la mecánica de muerte y/o habilidades desbloqueables para el jugador, conviene mantener dichas variables de control separadas del jugador, para facilitar su modularidad y evitar la pérdida de información si se bloquea o elimina el objeto del jugador.

3.3.7. Cronómetro:

Para añadir rejugabilidad al juego, se ha implementado un cronómetro. De este modo, el jugador podrá jugar una y otra vez para intentar mejorar su tiempo.

Cuando se pulsa la primera tecla en el juego, el valor del reloj interno se guarda en la variable *startTime*. Cada *frame*, se restará este valor al reloj actual, para calcular el tiempo en juego. Este tiempo se mostrará bajo la puntuación del jugador, en la esquina superior izquierda, con el formato "t = m:s.ms". En cuanto se recoja la última moneda, el cronómetro se detendrá y se mostrará en el centro de la pantalla, junto con un mensaje de victoria. (Ver figura 3.5)

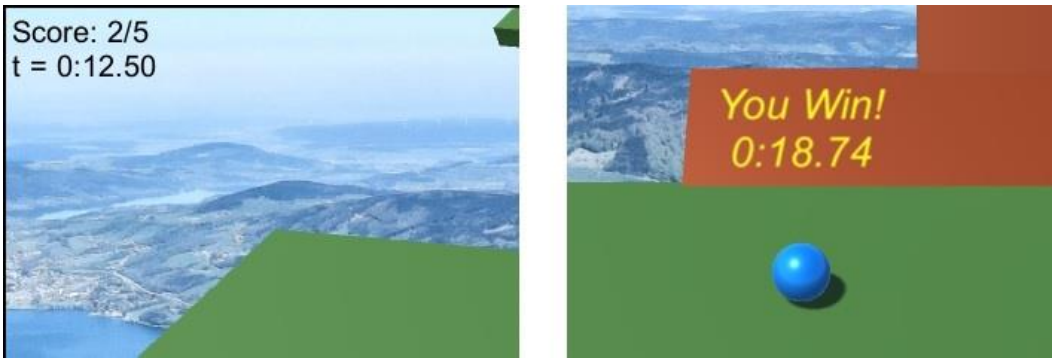


Figura 3.5: Cronómetro durante y al final del juego.

En el siguiente fragmento del script PlayerController.cs se encuentra todo el código relevante al cronómetro del juego. (Ver script 3.7)

```
// Se inicializa el tiempo a 0 cuando se carga el juego
void Start() {
  startTime = 0;
  winText.text = "Collect every coin!";
  ...
}

// Este procedimiento es llamado cada frame
void Update(){
  if (startTime == 0) {          // Comienzo del cronómetro,
    if(Input.anyKeyDown) {       //  con la primera tecla pulsada.
      startTime = Time.time;
      winText.text = "";
```

```
    }
  } else if (score <= 5) {       // Hasta que termine el juego
    float t = Time.time - startTime;
    minutes = ((int) t / 60).ToString();
    seconds = (t % 60).ToString("f2");
    SetScoreText();
}

// Método para actualizar el contador de la interfaz
void SetScoreText() {
  if (score == 5) {              // Ha terminado el juego, centrar
    scoreText.text = "Score: " + score.ToString() + "/5";
    winText.text = "You Win!\n" + minutes + ":" + seconds;
    score++;
  } else if (score < 5) {        // No ha terminado, a la esquina
    scoreText.text = "Score: " + score.ToString() + "/5\n
                     t = " + minutes + ":" + seconds;
  }
}
```

Script 3.7: Control del cronómetro e interfaz del juego.

3.3.8. Música y efectos sonoros:

Se ha creado un tema musical para el juego. Se ha compuesto y grabado a piano, con una estructura que permite enlazar el final con el principio y crear un bucle. Se le ha asignado a la cámara para mantener siempre un volumen constante, y mediante el editor, se ha configurado para que empiece a sonar en cuanto se cargue el juego, y suene en bucle sin ninguna espera.

También se ha grabado con el piano un sonido de moneda clásico, que se ha asignado a la pelota del jugador. En cuanto el jugador recoge una moneda, se reproduce dicho sonido para dar un indicador sonoro mediante el siguiente código, parte del script PlayerController.cs (script 3.8).

```
// Este procedimiento es llamado cuando tocas una moneda
void OnTriggerEnter(Collider other) {
  if (other.gameObject.CompareTag("Pick Up")) {
    this.GetComponent<AudioSource>().Play();  // Reproducir sonido,
    other.gameObject.SetActive(false);        // desactivar moneda
    score++;                                  // y actualizar puntuación.
  }
}
```

Script 3.8: Interacción con las monedas del juego, indicador sonoro y puntuación.

3.4. Análisis con usuarios y ajustes

La primera versión jugable (figura 3.6) tenía una pendiente más amplia y pronunciada para la moneda 3, salientes y rampas más estrechas para las monedas 4 y 5, y los enemigos custodiando la moneda 2 se movían más rápido. Además, las partes inclinadas del tablero tenían el mismo color que el suelo liso.

La intención de este juego era que fuese accesible para cualquier tipo de jugador, por lo que se realizaron pruebas con terceros para comprobar su nivel de dificultad, y si sería necesario hacer cambios en el juego.

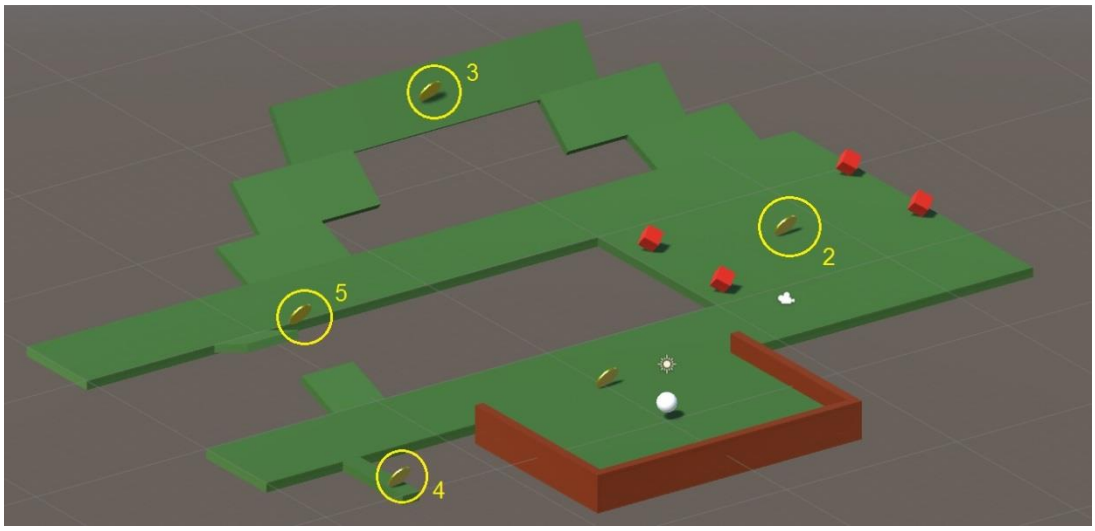


Figura 3.6: Primera versión completa del juego Keep Rolling.

El juego se probó con varias personas de distintas edades y sexos. Los jugadores habituales asimilaron enseguida los controles y no les costó conseguir todas las monedas, con pocas eliminaciones o ninguna. Los más adultos o menos experimentados, sin embargo, tuvieron dificultades para recoger varias monedas. No conseguían hacerse a los controles, y les costaba mantener el control de la pelota. Para facilitar su uso, se han realizado los siguientes cambios en el juego:

- Se ha reducido la pendiente de la moneda 3. Comparando las figuras 3.6 y 3.2 se puede apreciar cómo ha cambiado. Varios jugadores no consiguieron la moneda antes de rendirse, por lo que he decidido hacerla más corta y menos inclinada. Aún existe riesgo de caída, pero es más reducido. Además, se ha cambiado el color de la pendiente al de los muros, para ayudar a resaltar que no es una superficie lisa.
- Se han ensanchado las pasarelas y rampas de las monedas 4 y 5. Al igual que con la moneda 3, los jugadores menos hábiles tuvieron demasiadas dificultades para conseguir estas monedas. También se ha cambiado el color de las rampas para la moneda 5, para resaltar que no son superficies lisas.
- Se ha reducido la velocidad de los enemigos. En la primera versión había que calcular el momento exacto para atravesar el obstáculo. Ahora, con un poco de habilidad se puede maniobrar entre los enemigos sin ser eliminado, abriendo más posibilidades, pero se mantiene el riesgo de ser eliminado.
- Se ha añadido fricción al tablero y la pelota. Con esto se ha conseguido que a los jugadores menos experimentados les cueste menos detener la pelota antes de intentar una maniobra, ya que sin fricción era más complicado pararla mediante movimientos. Además, se avanzará más rápido por el aire. Esto da a los jugadores

más experimentados la posibilidad de calcular sus saltos para avanzar con mayor rapidez por el tablero, y probar nuevos recorridos.

3.5. Resumen de los logros alcanzados

Se ha completado un juego en 3D, y se han conseguido implementar todas las mecánicas propuestas para este ejemplo. Partiendo del resultado del tutorial se ha creado un juego nuevo, y se ha programado todo lo necesario para expandir fácilmente sobre él.

Disponiendo del tiempo necesario, sobre esta base sería sencillo ampliar el juego creando niveles adicionales. Mediante la clase GameVariables.cs (script 3.6) se pueden conservar variables como el tiempo acumulado entre niveles, y cargando una sucesión de niveles en cuanto se recojan todas las monedas se podría crear un juego más completo sin necesidad de añadir mecánicas adicionales.

Con este ejemplo he dado mis primeros pasos en Unity sin la ayuda de un tutorial, y he conseguido desenvolverme y cumplir con los objetivos. He reforzado los fundamentos aprendidos anteriormente, y he obtenido nuevos conocimientos que me serán útiles en mis próximos proyectos.

4

Shooting Robots

El objetivo de Shooting Robots es crear un juego Shoot ‘Em Up de estilo arcade en 2D. Este ejemplo se creará desde cero para tener un control total del diseño, y se implementarán mecánicas como el doble salto, la salud del personaje y la habilidad de disparar.

Se usarán y editarán *sprites* de terceros y se crearán animaciones con ellos para conseguir un estilo visual sofisticado.

El objetivo del jugador en Shooting Robots es conseguir la máxima puntuación posible, destruyendo los robots que irán apareciendo por el escenario mientras esquiva sus disparos, con dificultad creciente. Para ello podrá moverse, saltar y disparar (ver figura 4.1). Si pierde toda la salud, el juego terminará y mostrará la puntuación máxima obtenida.

Se recomienda ver el vídeo del siguiente enlace para ver el juego en acción antes de continuar leyendo: goo.gl/TAcptR

Figura 4.1: Escena del juego Shooting Robots.

4.1. Interfaz de usuario

Al ejecutar la aplicación, el programa mostrará un panel de selección donde podremos comenzar a jugar directamente o configurar varios parámetros. Si pulsamos el botón “Play!”, el juego comenzará con los ajustes que detecte automáticamente. Si no, podremos ajustar parámetros como la resolución de pantalla, la calidad de los gráficos o el modo de ventana.

Una vez comenzado el juego se cargará la escena, y desde este momento, los controles que tendremos a nuestra disposición serán los siguientes:

- *Flechas de dirección* del teclado o teclas *A*, *S* y *D*: con éstas moveremos al personaje de lado a lado (*flechas izquierda*/*derecha* o *A* y *D*), y si mantenemos pulsado *flecha abajo* o *S* mientras el personaje está sobre una plataforma flotante, éste se dejará caer.
- *Barra espaciadora*: al pulsar esta tecla el personaje dará un salto. Si la volvemos a pulsar en el aire, saltará una vez más. Esto se conoce como doble salto, y un doble salto eficaz será útil para llegar a zonas altas más fácilmente.
- *Return* o clic *ratón izdo*: si pulsamos la tecla *Return* o hacemos clic con el *botón izdo.* del ratón, el personaje disparará hacia el lado en el que esté mirando en ese

momento. Si mantenemos la tecla pulsada, el personaje disparará lo más rápido posible, dos disparos/segundo. Se puede disparar en movimiento y desde el aire.

- *Esc*: al pulsar la tecla *Escape* se pausará el juego y se mostrará un menú con distintas opciones: *Continue* para reanudar el juego, *Restart* para reiniciarlo y *Exit* para cerrar la aplicación. Haciendo clic con el ratón podremos escoger la opción que queramos, o cerrar el menú y continuar jugando volviendo a presionar la tecla *Escape*.

  Si perdemos toda la salud, una variación de este menú aparecerá automáticamente, sustituyendo el botón *Continue* por nuestra puntuación.

Una vez comenzado el juego, utilizando estas opciones, el jugador deberá destruir el mayor número de robots que le sea posible mientras esquiva sus disparos, ya sea saltando o cubriéndose sobre las plataformas flotantes (ver figura 4.2). Cuando el jugador sea alcanzado por disparos o choque contra el fuego suficientes veces, perderá toda su salud y el juego terminará.

En la figura 4.2 a continuación se puede ver el escenario en el que se desarrolla el juego, y muestra los distintos elementos del mismo.

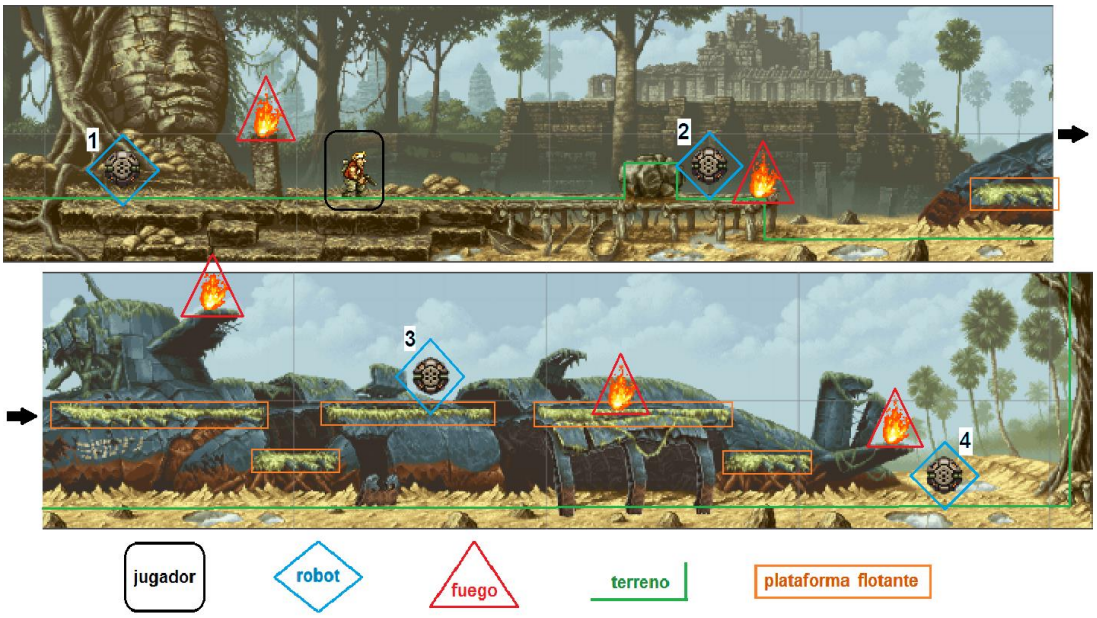


Figura 4.2: Elementos del juego y su disposición inicial.

El jugador controlará el personaje humano marcado en el primer cuadrado. Puede posarse sobre el terreno y las plataformas flotantes, y dejarse caer de las últimas manteniendo pulsada la tecla *abajo*. Solo puede disparar horizontalmente, ya sea en el suelo o en el aire, y puede destruir los robots disparándoles tres veces. Dispone de nueve puntos de salud representados con corazones en el margen superior izquierdo de la pantalla (figura 4.1).

Los robots, marcados con rombos, se moverán de lado a lado buscando al jugador para dispararle. Si éste entra en su radio de alcance, le dispararán tres veces antes de seguir avanzando, a no ser que se aleje, en cuyo caso se empezarán a mover en cuanto salga de su

radio de alcance. Pueden disparar en cualquier dirección, y restarán dos puntos de salud al jugador si lo alcanzan con un disparo.

El robot número 1 de la figura 4.2 se quedará sobre la superficie elevada, mientras que los robots 2, 3 y 4 patrullarán la zona arenosa. En cuanto los cuatro hayan sido destruidos, la velocidad del juego aumentará ligeramente y aparecerán otros cuatro robots. Este proceso se repetirá hasta que logren acabar con el jugador.

Los fuegos, en los triángulos, están colocados para obligar al jugador a maniobrar en zonas de otra forma seguras. Si el jugador choca con ellos le restarán un punto de salud, y le provocarán un ligero retroceso en la dirección opuesta. No se pueden apagar.

El terreno marcado está dividido en dos zonas, la primera hasta la roca y la zona arenosa. Ningún elemento del juego puede atravesarlo. El jugador deberá recorrerlo para encontrar y destruir todos los robots desplegados por él.

Las plataformas flotantes sobre la zona arenosa, destacados con rectángulos, ofrecen al jugador una cobertura fácil. El jugador podrá atravesarlas desde abajo, pero se posará sobre ellas una vez esté encima. Los proyectiles de los robots no las atravesarán, por lo que podrá saltar y cubrirse sobre ellas si se ve rodeado. Para dejarse caer, solo hay que mantener pulsada la tecla *abajo* y el personaje las atravesará.

4.2. Mecánicas del juego

Para crear este juego ha hecho falta conseguir una serie de elementos externos. A la hora de crear un juego en 3D, se pueden importar objetos 3D o bien crear y editar las figuras en el propio motor Unity, ya sean cubos, esferas, planos... Para un juego en 2D, en cambio, es imprescindible disponer de imágenes o *sprites* que importar para dar una imagen física a los distintos elementos del juego.

Para el escenario podría bastar con una sola imagen estática, pero para conseguir sensación de movimiento en los elementos móviles hacen falta muchas imágenes con las que crear secuencias para simular el movimiento. Dado que el objetivo de este proyecto es probar distintas mecánicas y aprender sobre el funcionamiento de Unity, se ha optado por utilizar *sprites* de terceros [7]. Las hojas de *sprites* completas se pueden ver en el anexo B.

Excluyendo los *sprites* externos que se han utilizado, este juego se ha creado desde cero. Inspirado por los *Shoot 'Em Up* de plataformas de los arcades, se ha intentado emular el estilo de juego para crear una experiencia similar. Estas son las mecánicas que se han implementado en Shooting Robots:

- Físicas / Movimiento en 2D: se ha utilizado un nuevo método para mover los objetos, consiguiendo un control mucho más responsivo y preciso.
- Saltos: además del salto tradicional, se ha implementado la funcionalidad de doble salto. Para ello, se han añadido diversos componentes con los que controlar la situación del personaje y determinar si aún puede saltar o no.

- *Sprites* y animaciones: una vez preparada la estructura del juego, se han añadido *sprites* (imágenes) para el fondo y se ha animado el personaje principal. En esta fase se han creado animaciones y transiciones para la postura estática, en movimiento y en salto del protagonista.
- Cámara dinámica: se ha programado una cámara que, en lugar de tener al personaje centrado en todo momento, se adelantará a él dependiendo de su dirección para mostrar más al frente del personaje que a su espalda. Se han suavizado estas transiciones para que el movimiento de cámara resulte fluido.
- Terreno y plataformas flotantes: se ha adecuado el terreno de juego a la imagen de fondo, y adaptado para habilitar el movimiento adecuado donde corresponda. Se han creado plataformas flotantes a las que poder saltar desde abajo, y en las que posarse una vez atravesadas. Manteniendo pulsada la tecla *flecha abajo* o *S*, el jugador podrá dejarse caer por ellas.
- Daños al jugador: se han añadido puntos de salud al personaje del jugador, que perderá bajo ciertas circunstancias. Junto con la función de perder salud se ha programado un retroceso y una breve fase de invulnerabilidad, para así evitar poder perder toda la salud de golpe. Se ha añadido una pequeña animación para representar la pérdida de salud.
- HUD: se ha creado un HUD (*Head-Up Display*) o interfaz simple con el que mostrar de forma clara y visual la salud restante del jugador.
- Obstáculos: para crear pequeños peligros en zonas sin enemigos, se ha creado un fuego que colocar en distintas posiciones que dañará al jugador si entra en contacto con él. Este fuego no se ve afectado por la física para poder colocarlo a distintas alturas sobre el fondo.
- Menú de pausa: se ha implementado un menú que pausará el juego y mostrará un panel con distintas opciones para el jugador.
- Game Over: cuando el jugador pierda toda la salud, se reproducirá una animación de fin de juego y se desplegará un menú similar al de pausa, pero mostrando la puntuación conseguida y ofreciendo unas opciones limitadas.
- Disparos del jugador: se ha implementado la acción de disparar del jugador. Se han creado las animaciones necesarias para ello, tanto para el personaje como para los proyectiles.
- Enemigos: se han diseñado unos enemigos más complejos que los de el ejemplo Keep Rolling para este juego. Estos enemigos son robots flotantes que buscarán y dispararán al jugador. Están formados por varios componentes que controlan distintos aspectos de su comportamiento, y se irán generando a medida que avance el juego y el jugador destruya otros robots.
- Objetivo del juego: se ha programado el control del juego de forma que genere nuevos robots a medida que el jugador destruya los antiguos. Cada vez que destruya

cuatro, se mostrará un mensaje y aumentará la velocidad del juego antes de generar otros cuatro.

Como ya se han probado el sonido y los efectos sonoros en el ejemplo Keep Rolling, se ha decidido omitir este apartado y algunas funcionalidades en este juego, con el fin de poder dedicar más tiempo a las nuevas mecánicas.

4.3. Desarrollo de las mecánicas

4.3.1. Físicas / Movimiento en 2D:

Puesto que los juegos arcade tienen en su mayoría controles precisos y responsivos, se decidió cambiar la forma en la que se controlaría al personaje de Shooting Robots. Si se controlara aplicando fuerzas sobre él como la pelota de Keep Rolling, el personaje tardaría en acelerar hasta alcanzar su velocidad máxima. Asimismo, en cuanto el jugador soltara la tecla de dirección, el personaje avanzaría un poco más hasta detenerse debido a la fricción. El objetivo es que el movimiento en este ejemplo sea instantáneo, y comience y termine exactamente cuando el jugador lo ordene.

Para esto, tras una breve investigación, se han encontrado las dos alternativas de control más populares: el control mediante *raycast* y el control mediante velocidades. Con el *raycast*, cada fotograma se crean (*cast*) una o varias líneas (*ray*s) desde el cuerpo en movimiento en la dirección a la que se desea mover, y con éstas se determina si el movimiento resultaría en el objeto en cuestión colisionando o atravesando algún otro objeto. Si sería así, el movimiento no se realiza. Si no, se alteran los parámetros de posición del objeto para moverlo. Este método es sobre todo adecuado para juegos *top-down* o con "vista desde arriba", en los que el movimiento puede ser en cualquier dirección y no se tiene en cuenta la gravedad.

La otra alternativa es el movimiento mediante velocidades. Cada fotograma se comprueba si el jugador ha pulsado una tecla de dirección, y si lo ha hecho, se alteran los parámetros de velocidad del objeto; si no está pulsando ninguna tecla, la velocidad se hace 0 y el objeto se detendrá en seco. Si fuera a colisionar con otro objeto, el propio motor de físicas de Unity se encarga de impedir el movimiento. Además, se sigue pudiendo aprovechar la gravedad para controlar el movimiento vertical. Puesto que nuestro personaje podrá saltar y caer, este es el movimiento que se ha decidido implementar.

El siguiente fragmento del script PlayerController.cs controla el movimiento horizontal del personaje:

```
public float moveSpeed = 5f;     // Velocidad por defecto.
private float moveHorizontal;    // Para leer el input.

...

// Este procedimiento es llamado cada frame
void FixedUpdate() {
```

```
    if (knockbackLeft < 0f) {
      moveHorizontal = Input.GetAxisRaw("Horizontal");

      if (grounded) {
        rb.velocity = new Vector2(moveHorizontal * moveSpeed,
                                  rb.velocity.y);
      } else {
        rb.velocity = new Vector2(moveHorizontal * moveSpeed * 1.25f,
                                  rb.velocity.y);
      }
    }
  }
```

Script 4.1: Fragmento de PlayerController.cs encargado del movimiento horizontal.

El procedimiento *FixedUpdate()* (an. A.3) es llamado automáticamente cada frame, y en éste primero se comprueba si aún le queda tiempo de retroceso al jugador (explicado en la sección 4.4.6), dado que en este periodo el personaje no será controlable. Si está libre, se leerá el *input* del jugador sin tratar (*raw*), es decir, será '-1', '0' o '1'. De este modo se puede multiplicar este valor por la constante de velocidad escogida para el personaje directamente, y la velocidad resultante será fija en todo momento. Se mantendrá la velocidad vertical sin modificar, para dejar así que la gravedad afecte al personaje.

Si la variable de control *grounded* es *false*, el personaje se moverá más rápido ya que significa que está en el aire (sección 4.3.2 a continuación).

Además, mediante los ajustes del objeto en el editor de Unity se ha bloqueado la rotación en el eje Z del personaje, para que la física no pueda tumbar al personaje a causa del movimiento.

4.3.2. Saltos:

Para este juego, además del salto desde el suelo se ha implementado la opción de doble salto. Para controlar si el personaje está en contacto con el terreno o no, se ha añadido un nuevo componente y script al personaje. Se ha añadido un *trigger* bajo el *collider* (an. A.2) o colisionador principal del personaje, con un script que comprobará cada *frame* si éste está en contacto con el suelo o no, como se puede ver en la figura 4.3:

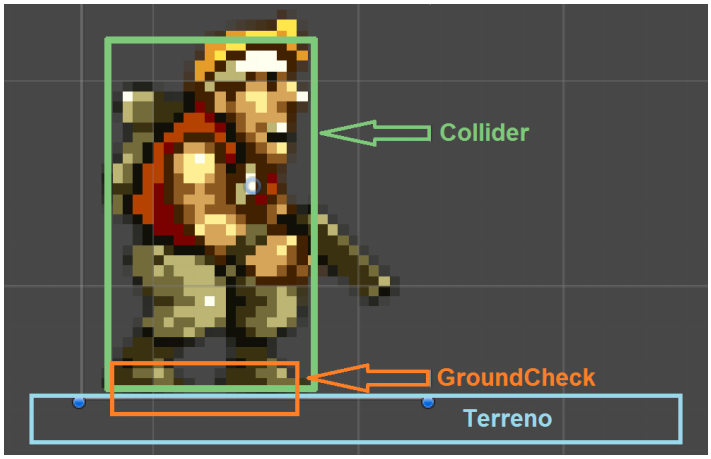


Figura 4.3: Collider y GroundCheck del personaje principal.

Este *trigger* "GroundCheck" se moverá pegado al personaje y no ocupa espacio físico, por lo que puede atravesar objetos. Mientras esté intersecando con el terreno, se encargará de que el valor *grounded* del jugador sea *true*. Si no, este valor se hará *false* para hacer saber al controlador del personaje que está en el aire. El script GroundCheck.cs a continuación se encarga de lo descrito.

```
public class GroundCheck : MonoBehaviour {
  private PlayerController player;  // Script del personaje

  ...

  // Si GroundCheck interseca con el terreno
  void OnTriggerEnter2D(Collider2D other) {
    if (other.gameObject.CompareTag("Terrain")) {
      player.grounded = true;    // Marcar que está en el suelo
    }
  }

  // Mientras GroundCheck inteseque con el terreno
  void OnTriggerStay2D(Collider2D other) {
    if (other.gameObject.CompareTag("Terrain")) {
      player.grounded = true;    // Marcar que está en el suelo
    }
  }

  // Si GroundCheck sale del terreno
  void OnTriggerExit2D(Collider2D other) {
    if (other.isTrigger == false && !other.CompareTag("Enemy")) {
      player.grounded = false;   // Está en el aire
    }
  }
}
```

Script 4.2: Script GroundCheck.cs para determinar si el personaje está en el aire.

El *trigger* GroundCheck es necesario para nuestro juego, ya que en caso de determinar la situación del personaje con su propio collider podríamos encontrarnos con la siguiente situación de la figura 4.4:

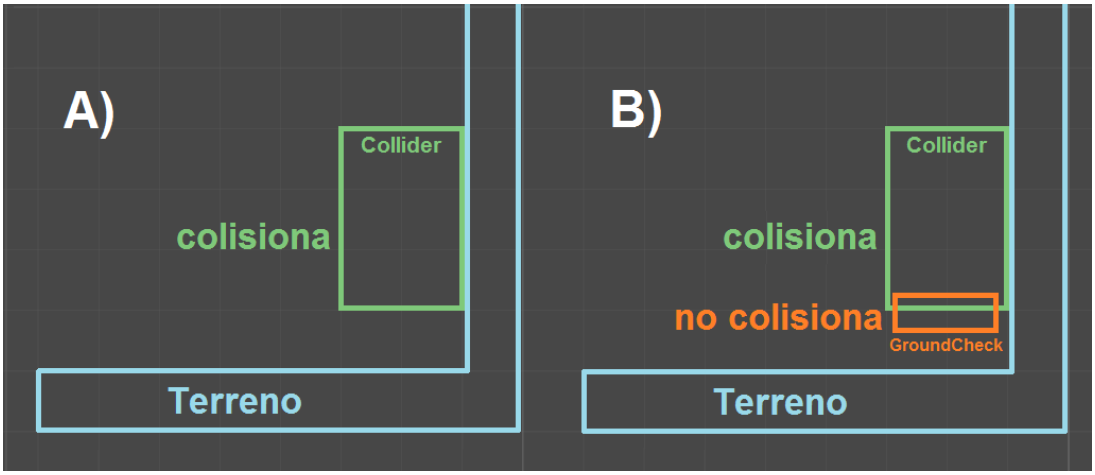


Figura 4.4: Ejemplo de situación con y sin un "ground check".

Como se puede ver en la figura 4.4, un *ground check* ayuda a determinar cuándo un cuerpo realmente está en el suelo y a evitar falsos positivos como en el ejemplo A.

Con la ayuda de este controlador, es sencillo implementar un doble salto. El siguiente fragmento del script PlayerController.cs tiene todo lo necesario para habilitar el salto y doble salto:

```
public float jumpForce = 10f;           // Impulso base del salto.
public bool grounded = true;            // Booleano de control.
public bool canDoubleJump = false;      // Booleano de control.

...

// Este procedimiento es llamado cada frame
void Update() {

  ...

  // Si se pulsa la tecla espacio y no está en retroceso
  if (knockbackLeft <= 0f && Input.GetButtonDown("Jump")) {
    if (grounded) {       // El personaje está en el suelo
      rb.velocity = new Vector2(rb.velocity.x, jumpForce);
    } else if (canDoubleJump) {  // Aún puede hacer un salto doble
      rb.velocity = new Vector2(rb.velocity.x, jumpForce * 0.8f);
      canDoubleJump = false;
    }
  }

  ...

  if (grounded) {
    canDoubleJump = true;
  }

  ...

}
```

Script 4.3: Fragmentos de PlayerController.cs relevantes al salto.

Mientras el personaje esté en el suelo la variable de control *canDoubleJump* será *true*, y ésta se hará *false* en cuanto se haga un doble salto. En Shooting Robots se sustituye la velocidad vertical al saltar en lugar de sumarla, para que siempre ascienda la misma distancia. El doble salto, sin embargo, tendrá un 20% menos de fuerza para dificultar un poco el acceso rápido a las plataformas flotantes (apartado 4.3.5).

4.3.3. *Sprites* y animaciones:

Una vez implementado el movimiento básico, se ha empezado a crear las animaciones del personaje y las imágenes del fondo. Aunque esta tarea requiera de poca programación, hay cuestiones que hay que tratar mediante scripts.

Primero, desde el propio editor se ha clasificado cada *sprite* individual de la hoja del personaje (an. B.2), y se ha creado una animación para correr con una secuencia de ellos. Por ejemplo:

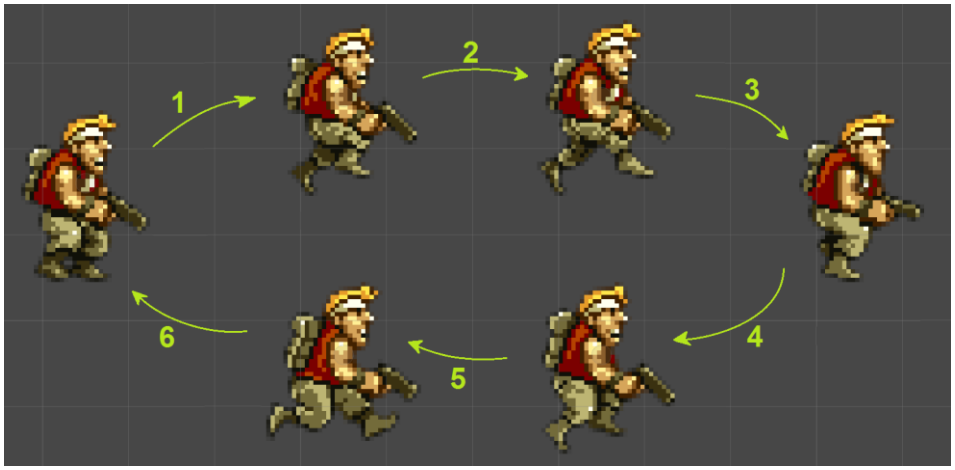


Figura 4.5: Secuencia de movimiento del personaje principal.

Una vez creada esta animación, se ha añadido al script PlayerController.cs las variables de control que indicarán al Animador del personaje (an. A.2) su situación actual. El siguiente fragmento de PlayerController.cs actualiza las variables que el Animador comprueba para reproducir la animación que le corresponda:

```
// Este procedimiento es llamado cada frame
void Update() {
  // Dar al animador los valores del personaje -> ("Variable", valor)
  animator.SetBool("Grounded", grounded);
  animator.SetFloat("Speed", Mathf.Abs(rb.velocity.x)); }

  ...
}
```

Script 4.4: Fragmento del script PlayerController.cs para la animación de correr.

El animador sabrá ahora si el personaje está en el suelo y si su velocidad absoluta es mayor que 0, por lo que podrá determinar si debe reproducir la animación de estar quieto ("Idle") o la de correr ("Walk"). Para ello, es necesario configuar el animador desde el editor:

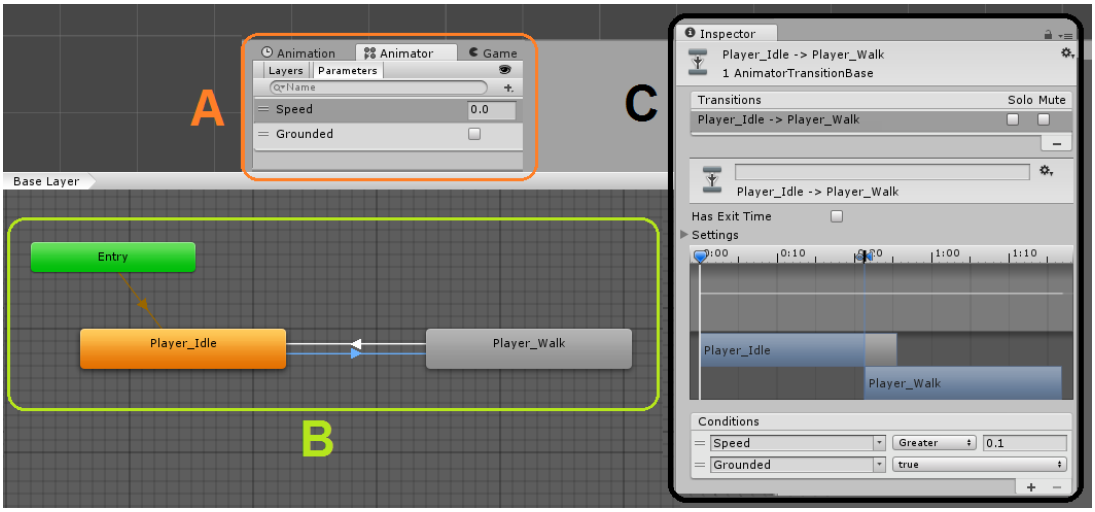


Figura 4.6: Ajustes del Animador del personaje principal.

Tras definir los estados de "Idle" y "Walk" (B), se pueden crear y configurar las transiciones entre ellos. En el panel C se puede ver la configuración de la transición "Idle -> Walk". Para la mayoría de las transiciones queremos que puedan ser interrumpidas para poder cambiar instantáneamente, por lo que la opción "Has Exit Time" quedará sin marcar. Debajo, se puede ver que las condiciones necesarias para que se reproduzca la animación de correr es que la variable *Speed* debe ser mayor que 0,1, y *Grounded* debe ser true. Estas son las variables del panel A, las mismas que hemos tratado anteriormente con el script 4.4.

Siguiendo el mismo procedimiento se han creado las animaciones de estar de pie, saltar, disparar y morir (explicadas más adelante), y el grafo de estados resultante es el siguiente:

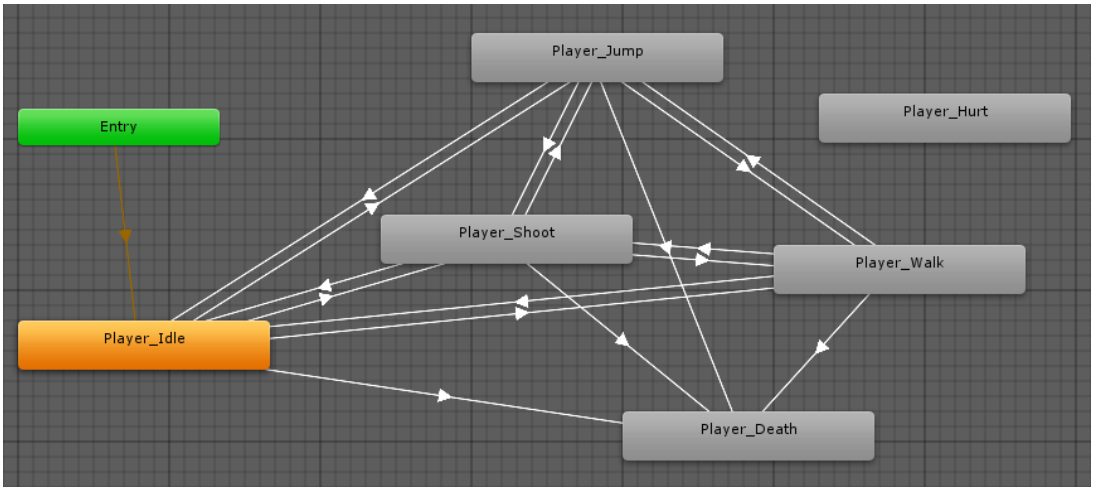


Figura 4.7: Grafo de estados completo del personaje.

Para evitar necesitar *sprites* con el personaje mirando en ambas direcciones, se volteará el personaje alterando su escala. Con el siguiente código se puede voltear la imagen horizontalmente según el *input* que se reciba del jugador.

```
if (moveHorizontal < 0f) {              // Input es Izda.
  transform.localScale = new Vector3(-1, 1, 1);
} else if (moveHorizontal > 0f) {       // Input es Dcha.
  transform.localScale = new Vector3(1, 1, 1);
}
```

Script 4.5: Fragmento de PlayerController.cs para voltear el *sprite* horizontalmente.

Además, comprobando el valor de `transform.localScale.x` se podrá saber si el personaje está mirando hacia la derecha (1) o hacia la izquierda (-1) en cualquier momento.

Para crear el escenario, se han añadido dos capas de *sprites*. La primera va en el fondo, y es una imagen fija y completa del terreno de juego. La segunda capa son recortes del *sprite* anterior que irán entre el personaje y la cámara, para dar profundidad al fondo. Los *sprites* completos se pueden encontrar en el anexo B, pero en la siguiente figura se puede observar el efecto creado por las capas.

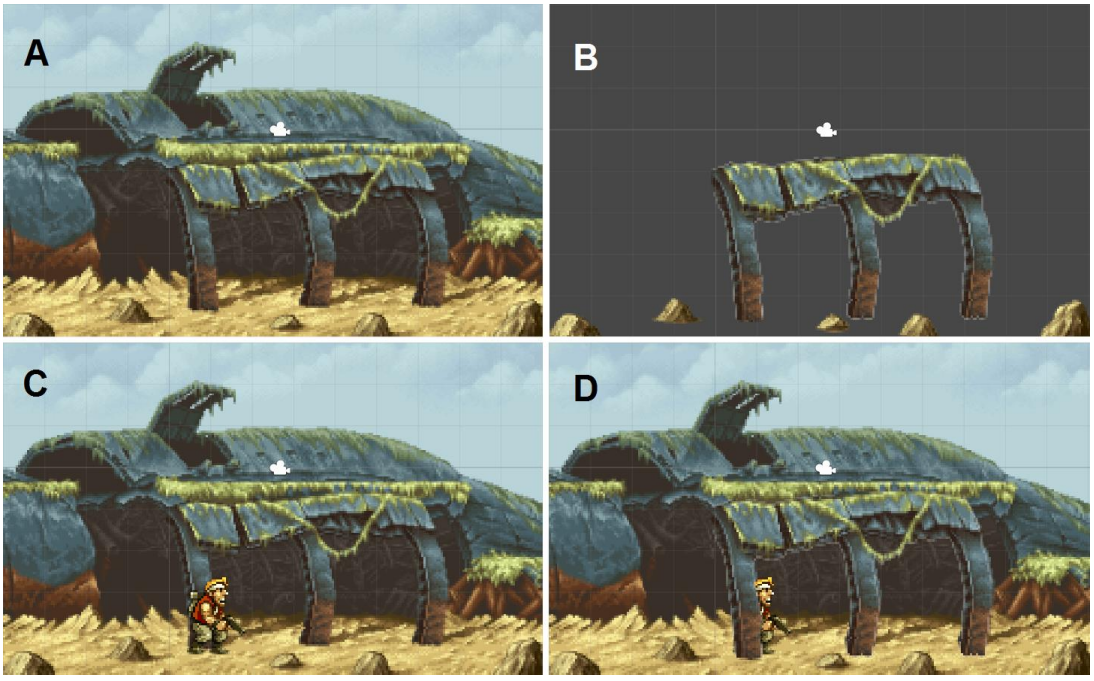


Figura 4.8: resultado de las capas del fondo.

En las secciones A y B de la figura 4.8 se pueden ver fragmentos de los *sprites* que forman ambas capas del fondo, y en las secciones C y D la sensación de profundidad conseguida colocando la capa B entre la cámara y el personaje.

4.3.4. Cámara dinámica:

Para la cámara del juego, se ha decidido implementar una cámara dinámica que se adapte a la situación, en lugar de desplazarse constantemente con el personaje totalmente centrado.

Para ello se ha utilizado la función SmoothDamp de la librería Mathf de Unity. Ésta se encarga de calcular los puntos intermedios por los que pasará la cámara para hacer un movimiento suave a lo largo de ciertos *frames*. Tras recibir la posición que corresponda ésta se corregirá con la función Clamp de la misma librería, para acotarla dentro de los límites del

mapa definidos previamente y que no muestre el juego fuera de los límites. El siguiente es el script CameraController.cs, que controla el movimiento recién mencionado.

```
public class CameraController : MonoBehaviour {
  public float smoothTimeX;      // Tiempo en el que se ajustará
  public float offset;           // Distancia que se adelantará

  public Vector2 minCameraPos;   // Límite x izquierdo
  public Vector2 maxCameraPos;   // Límite x derecho
  public bool bounds;            // Límites activos?

  private float direction = 1f;  // Dirección a la que "adelantarse"
  private Vector2 velocity;      // Necesario para SmoothDamp

  private GameObject player;     // Para la posición del jugador

  // Conseguir la instancia del jugador
  void Start() {
    player = GameObject.FindGameObjectWithTag("Player");
  }

  // Comprobar la dirección del jugador cada frame
  void Update() {
    direction = player.transform.localScale.x;
  }

  // Cada frame, calcular y corregir la posición de la cámara
  void FixedUpdate() {
    float posX = Mathf.SmoothDamp(transform.position.x,
                  player.transform.position.x + direction * offset,
                  ref velocity.x, smoothTimeX);
    transform.position = new Vector3(posX, transform.position.y,
                                     transform.position.z);

    if (bounds) {
      transform.position = new Vector3(Mathf.Clamp(
        transform.position.x, minCameraPos.x, maxCameraPos.x),
        transform.position.y, transform.position.z);
    }
  }
}
```

Script 4.6: Script CameraController.cs encargado del movimiento de la cámara.

Es posible editar los valores "smoothTimeX" y "offset" desde el propio editor, lo que facilita mucho las pruebas hasta encontrar el punto justo para que la cámara resulte fluida y no demasiado brusca al mismo tiempo.

4.3.5. Terreno y plataformas flotantes:

Tras adecuar la geometría del terreno para que se correspondiera a la imagen del escenario, se ha decidido implementar "plataformas flotantes" para los escombros del submarino (figura 4.2). Estas plataformas no simplemente flotan, sino que el jugador podrá atravesarlas de abajo a arriba con un salto, y luego posarse sobre ellas.

Para ello Unity cuenta con un ajuste en el editor para configurar plataformas como tales, pero en caso de activarla se pierde la opción de implementar más funcionalidades ya que entran en conflicto con los ajustes de su propia configuración. Puesto que se quiere que las

plataformas también puedan atravesarse de arriba abajo a voluntad del jugador (característica que las de Unity carecen), se ha implementado una versión propia de las plataformas.

Para ello se ha añadido un *trigger* pegado bajo las plataformas para detectar cuándo el jugador va a entrar en contacto ellas, y las plataformas son "desactivadas" hasta que el personaje las atraviese. En cuanto esté encima y deje de intersecar con el *trigger*, las plataformas vuelven a activarse. La figura 4.9 muestra la estructura y funcionamiento de las plataformas.

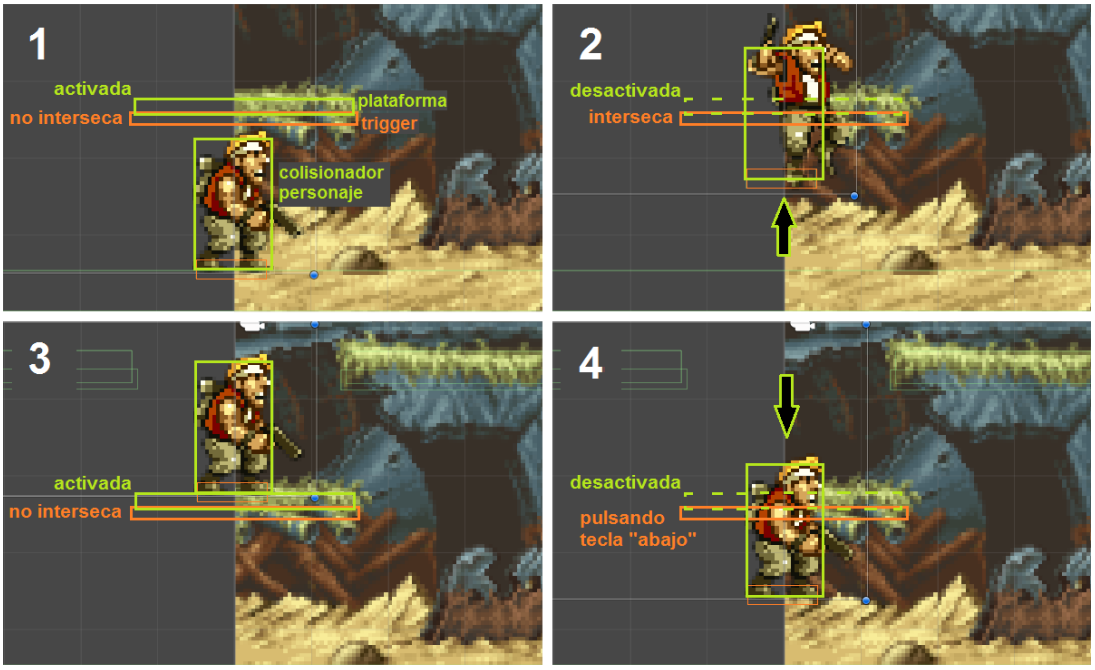


Figura 4.9: plataformas flotantes y funcionamiento.

Para añadir a las plataformas la opción de dejarse caer por ellas a voluntad (parte 4 de la figura 4.9), se han programado para desactivarse mientras el jugador mantenga pulsada la tecla *flecha abajo* o *S,* además de al saltarlas desde abajo. Así, podrán atravesarse en ambas direcciones. El siguiente es el script de los *triggers*, que controlan la plataforma a la que estén adjuntos.

```
public class PlatformController : MonoBehaviour {
  // Flag para saber si el personaje interseca con el trigger
  private bool colliding;

  void Start() {
    colliding = false;
  }

  // Este procedimiento es llamado cada frame
  void Update() {
    // Si el personaje interseca o "abajo" está pulsada, desactivar
    if (Input.GetAxisRaw("Vertical") < 0 || colliding == true) {
      transform.parent.gameObject.
        GetComponent<BoxCollider2D>().enabled = false;
    // Si no, ativar la plataforma
```

```
      } else {
        transform.parent.gameObject.
          GetComponent<BoxCollider2D>().enabled = true;
      }
    }

    // Mientras el personaje esté en el trigger, activar el flag
    void OnTriggerStay2D(Collider2D other) {
      if (other.CompareTag("Player")) {
        colliding = true;
      }
    }

    // En cuanto el personaje salga, desactivar el flag
    void OnTriggerExit2D(Collider2D other) {
      if (other.tag == "Player") {
        colliding = false;
      }
    }
}
```

Script 4.7: Script PlatformController.cs para controlar las plataformas flotantes.

4.3.6. Daños al jugador:

El jugador podrá recibir daño gradualmente en Shooting Robots, en lugar de ser eliminado inmediatamente. Se han asignado 9 puntos de salud al personaje, que irá perdiendo a medida que sea alcanzado por disparos o se queme (secciones 4.3.12 y 4.3.8). Para gestionar este daño recibido se han creado dos métodos públicos dentro del script PlayerController.cs del personaje, que podrán ser llamados desde cualquier objeto que pueda dañar al jugador.

El primer método creará un retroceso en el personaje, con los parámetros de duración, fuerza y dirección. Mientras dure el retroceso, el personaje no podrá controlarse (como se mencionó en los scripts 4.1 y 4.3) y será invulnerable al daño. Esta mecánica sirve para evitar que el jugador pueda quedar atrapado entre dos peligros y perder toda la salud sin poder hacer nada.

El segundo restará salud al personaje si éste no está en retroceso/invulnerable, acorde al valor que reciba el método como parámetro. Si el jugador pierde toda la salud, terminará el juego (4.3.10).

El siguiente script tiene todo lo relevante a la salud y el retroceso de PlayerController.cs:

```
// Variables de control de salud
private float knockbackLeft;
public int playerHealth;
public int maxHealth = 9;
public bool dead = false;

void Start() {
  rb = gameObject.GetComponent<Rigidbody2D>();
  ...

  playerHealth = maxHealth;      // Inicializar los parámetros
  knockbackLeft = 0f;
}

// Método para aplicar el retroceso
```

```
  public void Knockback(float knockDuration, float knockPower,
                        Vector3 knockDirection) {
    knockbackLeft = knockDuration; // Guardar la duración
    knockDirection.Normalize();    // Normalizar

    rb.velocity = Vector3.zero;    //  Resetear  velocidad  y  aplicar
fuerza
    rb.AddForce(new Vector3((knockDirection.x) * 30,
                (knockDirection.y + 1) * 25, 0) * knockPower);
  }

  // Método para dañar al personaje
  public void Damage(int dmg) {
    if (knockbackLeft <= 0f) {     // El personaje es vulnerable
      if (playerHealth - dmg <= 0) {      // Si el personaje muere
        playerHealth = 0;
        dead = true;
      } else {                            // Si aún vive
        playerHealth -= dmg;
      }
      gameObject.GetComponent<Animation>().Play("Hurt");
    }
  }
```

Script 4.8: Fragmento del script PlayerController.cs para el daño y retroceso.

Con estos dos métodos será posible restar salud al personaje desde cualquier otro script. Además, se ha añadido una animación "parcial" que se reproducirá cuando se dañe al jugador. Esta animación parcial cambiará el tono de la animación actual para que ésta parpadee en rojo y dar una pista visual de que se ha recibido daño (figura 4.10).

Figura 4.10: parpadeo y retroceso a causa del daño recibido.

4.3.7. HUD:

Para poder visualizar la salud del personaje de forma clara, se ha creado una interfaz (HUD) simple que mostrará en todo momento la salud restante en forma de corazones. En la esquina superior izquierda de la pantalla habrá siempre tres corazones, que podrán estar llenos (3 puntos), tocados (2 puntos), casi vacíos (1 punto) o huecos (0 puntos). De esta forma será posible conocer la salud restante con un vistazo rápido.

El siguiente script de la cámara se encargará de mostrar siempre la imagen que corresponda en la esquina:

```
public class HUD : MonoBehaviour {
  public Sprite[] HeartSprites;  // Array de sprites que se usará
  public Image HealthBar;        // Objeto en el que irán los sprites
```

```
    // Script del jugador para conseguir su salud actual
    private PlayerController player;

    ...

    // Cada frame, mostrar el sprite que corresponda -> Array[salud]
    void Update() {
        HealthBar.sprite = HeartSprites[player.playerHealth];
    }
}
```

Script 4.9: Script HUD.cs de la cámara para actualizar el indicador de salud.

Cada combinación de corazones se ha guardado en la posición del *array* que le corresponde a su salud desde el editor, por lo que bastará con utilizar la salud restante como índice para conseguir la imagen a mostrar.

En la siguiente figura se pueden ver un par de ejemplos del indicador de salud, con 7 y 5 puntos restantes respectivamente:

Figura 4.11: HUD con 7 y 5 puntos de salud.

Todos los *sprites* utilizados para crear el HUD se pueden ver en el anexo B.6.

4.3.8. Obstáculos:

Se ha creado fuego que dañará al jugador y lo hará retroceder si éste lo toca. El fuego es estático y no se moverá ni extenderá, pero el jugador no lo puede apagar. La principal función del fuego es ser un obstáculo para que el jugador tenga que maniobrar en zonas de otra forma seguras y tener más cuidado. Añade algo de dificultad al juego. En la figura 4.10 o el vídeo del juego se puede ver el fuego en funcionamiento.

El fuego está formado por un *trigger* y un script simple. En cuanto el jugador active el *trigger*, el script llamará a los métodos del jugador para dañarlo y provocarle el retroceso. El siguiente es el script que lo controla:

```
public class Hazard : MonoBehaviour {
  public int damage;              // Daño del obstáculo
  public float knockDuration;     // Duración del retroceso
  public float knockPower;        // Fuerza del retroceso

  private PlayerController player;

  ...

  // Si es el jugador quien entra en el collider
  void OnTriggerEnter2D(Collider2D other) {
    if(other.CompareTag("Player")) {
```

```
            player.Damage(damage);              // Aplicar daño y retroceso
            Vector3 knockDirection = other.transform.position
                                   - transform.position;
            player.Knockback(knockDuration, knockPower, knockDirection);
        }
    }
}
```

Script 4.10: Script para los obstáculos del juego.

Los valores del daño, duración y fuerza del retroceso se fijan desde el editor una vez asignado el script, por lo que sería posible utilizar el mismo script para distintos tipos de obstáculos: fuegos, láseres, pinchos... En el caso del fuego se restará un punto de salud al jugador, y le provocará un retroceso moderado.

Para la apariencia física del fuego, se ha creado una animación de llamas (siguiendo el mismo procedimiento que en el punto 4.3.3) y no se le ha asignado un cuerpo físico, por lo que la gravedad no tendrá efecto sobre él. De este modo es posible colocarlo en cualquier parte del juego, incluso flotando.

4.3.9. Menú de Pausa:

Se ha creado un menú de pausa que detendrá el juego y ofrecerá distintas opciones. Esto facilitará la selección de reiniciar el juego, cerrarlo, etc. en comparación con el juego Keep Rolling, donde cada función tenía una tecla asignada en el teclado.

Figura 4.12: Menú de pausa de Shooting Robots.

Para ello se ha creado un menú con distintos botones en el editor de Unity, que estará desactivado por defecto. Con un script asignado a la cámara, GameControl.cs, se comprobará cada frame si este menú debería estar activo (visible) o no. Para ello mirará la variable “paused”, que alternará los valores *true*/*false* al pulsar la tecla *Escape*. Mientras esta

variable sea *true*, además de mostrar el menú se reducirá la velocidad del juego a 0, para "congelar" o pausar así el juego.

Los siguientes fragmentos de script son todo lo relevante al menú de pausa en GameControl.cs:

```
  public class GameControl : MonoBehaviour {
    public GameObject PauseUI;            // Objeto de la interfaz

    private PlayerController player;      // Script del jugador

    private bool paused = false;          // Pausa activa?
    public float gameSpeed = 1;           // Velocidad del juego

    ...

    void Start() {
      PauseUI.SetActive(false);           // Desactivar pausa al inicio

      ...
    }

    // Este procedimiento es llamado cada frame
    void Update() {
      if (player.dead == false) {         // Si el juego no ha
terminado
        if (Input.GetButtonDown("Pause")) {
          paused = !paused;               // (Des)activar pausa
        }

        if (paused) {
          PauseUI.SetActive(true);        // Mostrar menú pausa
          Time.timeScale = 0;             // Pausar juego
        }
        if (!paused) {
          PauseUI.SetActive(false);       // Ocultar menú pausa
          Time.timeScale = gameSpeed;     // Reanudar juego
        }
      ...
    }

    // Botón "Continue" -> Desactivar pausa
    public void Continue() {
      paused = false;
    }

    // Botón "Restart" -> Volver a cargar la escena
    public void Restart() {
      SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    // Botón "Exit" -> Cerrar el juego
    public void Exit() {
      Application.Quit();
    }

    ...
  }
```

Script 4.11: Fragmento del script GameControl.cs para el menú de pausa.

Cada uno de los tres últimos métodos del script 4.11 está asignado a su correspondiente botón, y se llamarán cuando el jugador haga clic sobre dichos botones con el ratón.

4.3.10. Game Over:

Se ha creado una variación del menú de pausa para el fin del juego, que se mostrará solo cuando el jugador pierda toda la salud. Este menú sustituye el botón de reanudar el juego por un panel con la puntuación conseguida, y no se puede desactivar/reactivar pulsando la tecla *Escape*.

Figura 4.13: Menú de fin del juego.

Para acompañar el menú, se ha añadido una nueva animación parcial al personaje principal, y se ha implementado un procedimiento para ralentizar el juego al 30% durante un determinado tiempo antes de parar el juego y mostrar el menú de Game Over.

El control de este menú se hace en GameControl.cs, asignado a la cámara, junto con el menú de pausa. Los siguientes fragmentos son los necesarios para mostrar el fin del juego:

```
public class GameControl : MonoBehaviour {
  public GameObject DeathUI;            // Objeto de la interfaz

  private PlayerController player;      // Script del jugador

  public float gameSpeed = 1;           // Velocidad del juego

  ...

  void Start() {
    ...

    DeathUI.SetActive(false);           // Desactivar al inicio
  }
```

```
    // Este procedimiento es llamado cada frame
    void Update() {
      if (player.dead == false) {

        ...

      } else {                              // Si el jugador ha muerto
        Time.timeScale = 0.3f;              // Ralentizarlo al 30%
        StartCoroutine(ShowDeathMenu(1.5f));    // Iniciar Game Over
      }
    }

    // Botón "Try again!" -> Volver a cargar la escena
    public void Restart() {
      SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }

    // Botón "Exit" -> Cerrar el juego
    public void Exit() {
      Application.Quit();
    }

    // Función para iniciar el Game Over
    private IEnumerator ShowDeathMenu(float delay) {
      yield return new WaitForSecondsRealtime(delay);  // Esperar delay
      Time.timeScale = 0;                   // Detener el juego
      DeathUI.SetActive(true);              // Mostrar menú y puntuación
      player.scoreText.text = "You got:  " + player.score + " points!";
    }
  }
```

Script 4.12: Fragmento del script GameControl.cs para el fin del juego.

Para este menú se ha hecho uso de las corrutinas de Unity (an. A.3), con las que es posible ejecutar una función a lo largo de varios *frames*. De este modo, es posible iniciar la cámara lenta y con una sola llamada mostrar el menú a los varios segundos reales (1,5 en este caso), ya que este contador es independiente al reloj del juego, que se acelera o decelera cuando cambiamos la velocidad del juego.

4.3.11. Disparos del jugador:

Se ha dado al jugador la habilidad de disparar. Al pulsar la tecla *Return* o al hacer clic con el *botón izdo.* del ratón, el personaje disparará en la dirección en la que esté mirando.

Para ello, se ha creado un *prefab* (an. A.2) del proyectil. Éste está formado por un *trigger* para detectar las colisiones, un *RigidBody* sin gravedad para poder moverlo, un animador para las animaciones y un script para controlarlo todo. Este script volteará el *Sprite* en la dirección en la que se mueva, destruirá el proyectil si viaja una distancia determinada o reproducirá la animación de colisión si éste impacta con algo. El siguiente es el script de los proyectiles del jugador:

```
public class PlayerBulletController : MonoBehaviour {
  public float maxDistance;      // Distancia de vuelo máxima
  private float pos;             // Posición X inicial
  private bool hit = false;      // Ha impactado?

  private Rigidbody2D rb;        // Cuerpo físico
  private Animator animator;     // Animador
```

```
  void Start() {
    rb = gameObject.GetComponent<Rigidbody2D>();
    animator = gameObject.GetComponent<Animator>();
    pos = transform.position.x;  // Posición X inicial
  }

  // Este procedimiento es llamado cada frame
  void Update() {
    animator.SetBool("Hit", hit);       // Actualizar var. animador

    if (rb.velocity.x > 0) {            // Va hacia la derecha
      transform.localScale = new Vector3(1, 1, 1);   // No voltear
    } else if (rb.velocity.x < 0){      // Va hacia la izquierda
      transform.localScale = new Vector3(-1, 1, 1);  // Voltear
    }

    // Si el proyectil avanza el alcance máximo
    if (transform.position.x > pos + maxDistance
        || transform.position.x < pos - maxDistance) {
      Destroy(gameObject);              // Destruir el proyectil
    }
  }

  void OnTriggerEnter2D(Collider2D other) {
    // Si impacta con el terreno o un robot
    if (other.isTrigger == false || other.CompareTag("Enemy")) {
      hit = true;                       // Para el animador
      rb.velocity = Vector2.zero;       // Detener proyectil
      Destroy(gameObject, 0.25f);       // Destruir tras la animación
    }
  }
}
```

Script 4.13: Script PlayerBulletController.cs para los proyectiles del jugador.

Para que el jugador pueda disparar, se ha añadido el código del script 4.14 al PlayerController.cs del personaje, y se ha creado una animación de disparo. A diferencia del resto de animaciones ésta no se podrá interrumpir hasta que termine, por lo que bastará con tenerla activa un solo *frame* para que se reproduzca al completo.

Para evitar que se pueda disparar más rápido que la frecuencia de disparo fijada, cuando el jugador pulse *Return* se comprobará que el tiempo desde el último disparo sea mayor que la cadencia de disparo. Si es así, se activará la variable para el animador del personaje, se instanciará un proyectil en la dirección a la que se esté mirando y se reseteará el tiempo desde último disparo. El script 4.14 a continuación contiene todo lo relevante al disparo del jugador:

```
public float rateOfFire;          // Cadencia de fuego
public float bulletSpeed;         // Velocidad del proyectil
private float lastBulletShot;     // Tiempo desde último disparo
private bool shooting = false;    // Variable para el animador

...

// Este procedimiento es llamado cada frame
void update() {
  ...
  animator.SetBool("Shooting", shooting);     // Actualizar animador
  shooting = false;                           // Resetear variable
```

```
    ...

    // Actualizar tiempo desde el último disparo
    lastBulletShot += Time.deltaTime;

                                    // Si ya puede disparar
    if (Input.GetButton("Fire1") && lastBulletShot >= rateOfFire) {
      shooting = true;              // Var. para el animador

      // Conseguir dirección del disparo (hacia dónde mira)
      Vector2 direction = new Vector2(transform.localScale.x, 0);
      // Instanciar un proyectil
      GameObject bullet;
      bullet = Instantiate(playerBullet, barrel.transform.position,
                           barrel.transform.rotation) as GameObject;
      // Fijar su velocidad
      bullet.GetComponent<Rigidbody2D>().velocity =
                                          direction * bulletSpeed;
      // Resetear tiempo desde el último disparo
      lastBulletShot = 0;
    }

    ...
}
```

Script 4.14: Fragmento de PlayerController.cs para los disparos del jugador.

Desde el editor se le asignará el *prefab* del proyectil al personaje, y éste disparará como se ha descrito anteriormente y se puede ver en la figura 4.14 a continuación:

Figura 4.14: Animaciones de disparo, proyectil e impacto.

4.3.12. Enemigos:

Se ha diseñado un enemigo moderadamente complejo para Shooting Robots (fig. 4.15). Tiene una IA simple con la que patrullará el terreno, y se detendrá para disparar al jugador cuando lo encuentre. Está compuesto por varios objetos, y tiene sus propias animaciones. También se ha creado un nuevo tipo de proyectil para el robot.

Los robots se destruirán cuando reciban tres disparos del jugador, y se generarán nuevos robots cuando todos los presentes hayan sido destruidos.

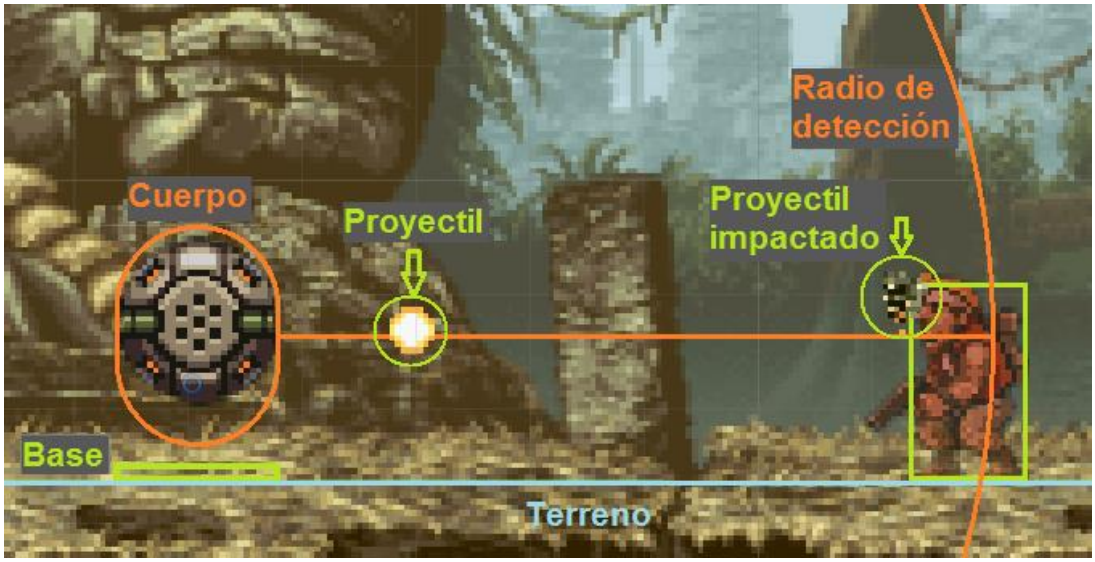


Figura 4.15: Enemigo de Shooting Robots y sus proyectiles.

El cuerpo del robot es un *trigger* con forma de cápsula para representar el área en el que levita. Si este *trigger* es alcanzado por un proyectil del jugador, el controlador del robot se restará un punto de salud (de tres). Para poder levitar sobre el terreno sin que el cuerpo toque el suelo, se le ha añadido un pequeño *collider* en forma de base unas unidades por debajo. De este modo las físicas le afectarán de un modo normal, pero el "cuerpo" del robot estará flotando.

El robot tiene también un gran *trigger* circular como radio de detección. En cuanto el jugador entre en él, el robot se detendrá para dispararle.

Por último, el robot dispone de dos animaciones: una en la que levita y se reproducirá continuamente, y otra en la que cae y estalla que se reproducirá en cuanto sea destruido. Al igual que con el personaje principal, se ha bloqueado la rotación sobre el eje Z para que no se pueda caer.

La mayor parte del control del robot (movimiento, pausas, salud y animaciones) se hace con el script 4.15 asignado al cuerpo del mismo. La acción por defecto del robot será moverse en una dirección. Si el robot lleva parado ¾ de segundo y aún está buscando al jugador (porque intenta avanzar contra un muro/terreno), cambiará de dirección y se seguirá moviendo. Si se topa con el jugador, el controlador del radio de detección lo alertará de ello y se detendrá (durante un máximo de tres segundos o hasta que el jugador se aleje) para dispararle.

Para evitar que se amontonen los robots o el jugador se atasque entre uno o varios de ellos y/o algún obstáculo, se ha programado de modo que el *collider* de la base ignore toda colisión con otro robot o el jugador, por lo que los robots se podrán atravesar.

Si el cuerpo detecta que ha sido alcanzado por un proyectil del jugador, se restará un punto de salud a sí mismo. Si es el último punto de salud restante, se actualizará la puntuación del juego (sección 4.3.13), se reproducirá la animación de explosión y se destruirá la instancia cuando ésta termine.

El siguiente script 4.15 es el controlador de los enemigos, DroneController.cs:

```
public class DroneController : MonoBehaviour {
  public float moveSpeed = 3f;          // Velocidad de movimiento
  public bool lookingForPlayer = true;  // Buscando al jugador?
  public int moveDirection = -1;        // -1 = IZDA, 1 = DCHA.

  private float lastPosition;           // Última posición
  private float newPosition;            // Posición actual
  private float timeHalted;             // Tiempo detenido

  public int droneHealth;               // Salud actual
  public int maxHealth = 3;             // Salud máxima
  public bool dead = false;             // Para el animador

  private GameControl gameControl;      // Script control del juego
  private PlayerController player;      // Script del jugador
  private Rigidbody2D rb;               // Cuerpo para físicas
  private Animator animator;            // Animador

  // Conseguir instancias, inicializar variables
  void Start() {
    ...

    droneHealth = maxHealth;
  }

  // Este procedimiento es llamado cada frame
  void Update() {
    animator.SetBool("Dead", dead);     // Actualizar animador

    newPosition = transform.position.x;
    if (newPosition == lastPosition) {
      timeHalted += Time.deltaTime;     // Está parado
    } else {
      timeHalted = 0f;                  // Está en movimiento
    }
    lastPosition = newPosition;

    // Si lleva 3/4 segundos parado y está buscando al jugador
    if (timeHalted > 0.75f && lookingForPlayer == true) {
      moveDirection = moveDirection * (-1);   // Media vuelta
    } else if (timeHalted > 3f) {       // 3s disparando...
      lookingForPlayer = true;          //  seguir avanzando
    }
  }

  // Este procedimiento es llamado cada frame
  void FixedUpdate() {
    // Si busca al jugador, avanzar
    if (lookingForPlayer == true) {
      rb.velocity = new Vector2(moveDirection moveSpeed,
                                rb.velocity.y);
    // Si lo ha encontrado, detenerse
    } else {
      rb.velocity = new Vector2(0f, rb.velocity.y);
    }
  }

  // Actualizar valor para el animador y puntuación del juego
  private void Die() {
    dead = true;                 // Activar flag del animador
    player.score++;              // Actualizar puntuación
    gameControl.activeDrones--;
    Destroy(gameObject, 1.4f);   // Destruir tras la animación
  }
  // El cuerpo es alcanzado por un disparo del jugador
  void OnTriggerEnter2D(Collider2D other) {
```

```
      if (other.CompareTag("PlayerBullet")) {
        if (droneHealth > 1) {      // Aún no es destruido
          droneHealth--;
          gameObject.GetComponent<Animation>().Play("Hurt");
        } else if (dead == false) {
          Die();                    // Es destruido
        }
      }
    }

    // Si colisiona con otro enemigo o el jugador
    void OnCollisionEnter2D(Collision2D col) {
      if (col.gameObject.tag == "Enemy" ||
          col.gameObject.tag == "Player") {
        // Ignorar la colisión
        Physics2D.IgnoreCollision(col.collider,
          gameObject.GetComponentInChildren<BoxCollider2D>(), true);
      }
    }
  }
```

Script 4.15: Script DroneController.cs para el control de los robots.

Para que disparen, se ha creado el script DroneCannonController.cs (4.16) asignado al rango de detección de los robots. Es parecido al fragmento para disparar del jugador (4.14), pero tiene varias diferencias. En lugar de disparar siempre horizontalmente, los robots calcularán la dirección en la que se encuentra el jugador y dispararán apuntando (figura 4.16).

Figura 4.16: Ejemplo de trayectoria posible de los proyectiles enemigos.

Utilizarán otro proyectil distinto diseñado para los enemigos, y modificarán una variable del script principal del robot para advertir de que el jugador está al alcance y ha de

detenerse. El método utilizado para limitar la cadencia de fuego e instanciar los proyectiles es el mismo que el del jugador, pero cambian algunos valores.

```
public class DroneCannonController : MonoBehaviour {
  public GameObject droneBullet;

  public float bulletSpeed;      // Velocidad del proyectil
  public float rateOfFire;       // Cadencia de fuego
  private float lastBulletShot;  // Tiempo desde el último disparo

  private GameObject player;           // Personaje del jugador
  private DroneController drone;       // Script del robot

  ...

  // Este procedimiento es llamado cada frame
  void Update() {
    lastBulletShot += Time.deltaTime;

    if (lastBulletShot >= rateOfFire) {
      // Calcular vector hacia el jugador
      Vector2 direction = player.transform.position -
                          this.transform.position;
      direction.Normalize();

      // Si el robot debe disparar
      if (drone.lookingForPlayer == false && drone.dead == false) {
        GameObject bullet;
        bullet = Instantiate(droneBullet, transform.position,
                             transform.rotation) as GameObject;
        bullet.GetComponent<Rigidbody2D>().velocity = direction *
                                                      bulletSpeed;

        lastBulletShot = 0;
      }
    }
  }

  // Si el jugador entra al radio de detección
  void OnTriggerEnter2D(Collider2D other) {
    if (other.CompareTag("Player")) {
      drone.lookingForPlayer = false;  // Para indicar disparar
    }
  }

  // Si el jugador sale del radio de detección
  void OnTriggerExit2D(Collider2D other) {
    if (other.CompareTag("Player")) {
      drone.lookingForPlayer = true;   // Para indicar avanzar
    }
  }
}
```

Script 4.16: Script DroneCannonController.cs para controlar los disparos enemigos.

El script para los proyectiles enemigos es una combinación de los de los proyectiles del jugador (script 4.13) y los obstáculos (script 4.10). Cambiarán de animación y se destruirán las instancias cuando impacten contra el jugador o el terreno al igual que los proyectiles del jugador, pero dañarán al jugador con el mismo formato de daño/retroceso que los obstáculos. La diferencia con el fuego es que los proyectiles restarán dos puntos de salud, y provocarán un retroceso menor.

El siguiente es el script de los proyectiles de los robots:

```
public class DroneBulletController : MonoBehaviour {
  public int damage;             // Daño del proyectil
  public float knockDuration;    // Duración del retroceso
  public float knockPower;       // Fuerza del retroceso

  private Rigidbody2D rb;              // Cuerpo del proyectil
  private Animator animator;           // Animador del proyectil
  private PlayerController player;     // Script del jugador

  private bool hit = false;      // Variable para el animador

  ...

  // Este procedimiento es llamado cada frame
  void Update() {
    animator.SetBool("Hit", hit);       // Actualizar animador
  }

  // Impacta contra algo
  void OnTriggerEnter2D(Collider2D other) {
    if (other.isTrigger == false) {
      hit = true;                       // Animación de impacto
      rb.velocity = Vector2.zero;       // Detener el proyectil

      // Si es el jugador, dañarlo
      if (other.CompareTag("Player") && damage > 0) {
        player.Damage(damage);
        Vector3 knockDirection = other.gameObject.transform.position
                                 - transform.position;
        player.Knockback(knockDuration, knockPower, knockDirection);
        damage = 0;                 // Hacer el proyectil inofensivo
      }
      Destroy(gameObject, 0.25f);  // Y destruirlo tras la animación
    }
  }
}
```

Script 4.17: Script DroneBulletController.cs para los proyectiles enemigos.

Al igual que con el fuego, los valores de las variables públicas se asignan desde el propio editor de Unity para su fácil manipulación.

4.3.13. Objetivo del juego:

Se ha añadido un contador de enemigos a GameControl.cs, que se utilizará para generar más robots cuando todos hayan sido destruidos. A la vez que genere los nuevos robots, aumentará la velocidad del juego en un 20%. Este proceso se repetirá hasta que el jugador pierda toda la salud y termine el juego.

Para ello se han añadido las siguientes líneas al script GameControl.cs:

```
...
public float gameSpeed = 1;             // Velocidad del juego

public GameObject dronePrefab;          // Prefab de enemigo
public int activeDrones = 4;            // Número de enemigos
private float timeUntilRespawn;         // Tiempo para crear enemigos
private Vector3[] droneSpawns;}         // "Spawns" de enemigos
```

```
  void Start() {
    ...

    timeUntilRespawn = 3f;  // Resetear el tiempo para crear más
robots

    droneSpawns = new [] {  // Definir "spawns" de los enemigos
      new Vector3(-6f, 5.5f, 0f),
      new Vector3(20f, 5.5f, 0f),
      new Vector3(48f, 5.5f, 0f),
      new Vector3(70f, 5.5f, 0f),
    };
    SpawnDrones();
  }

  // Este procedimiento es llamado cada frame
  void Update() {
    if (activeDrones == 0 && !paused) {         // Si no quedan robots
      timeUntilRespawn -= Time.deltaTime;       // Esperar al spawn...
      player.scoreText.text = "Prepare for another wave...";
    }

    // Y generar los nuevos robots / aumentar la velocidad
    if (timeUntilRespawn <= 0) {
      activeDrones = 4;
      timeUntilRespawn = 3f;
      gameSpeed += 0.2f;

      SpawnDrones();
    }
  }

  // Generar un robot en cada uno de los 4 spawns
  private void SpawnDrones() {
    GameObject drone;
    for (int i=0; i<4; i++) {
      drone = Instantiate(dronePrefab, droneSpawns[i],
                          Quaternion.identity) as GameObject;
    }
  }
  ...
```

Script 4.18: Fragmento de GameControl.cs para el objetivo del juego.

Con este script el juego ya puede continuar hasta que el jugador sea eliminado, y el juego arcade queda terminado.

4.4. Resumen de los logros alcanzados

Se ha conseguido crear un juego tipo arcade en 2D partiendo desde cero, implementando las numerosas nuevas mecánicas que me había propuesto. Se ha comprobado que la experiencia obtenida en los ejemplos anteriores es aplicable incluso a nuevos juegos de distinto carácter, y se ha podido crear un juego con mayor libertad que en Keep Rolling.

Al igual que con Keep Rolling, disponiendo del tiempo necesario no sería difícil expandir el juego sobre la base que se ha construido. Sería posible construir nuevos niveles a los que avanzar una vez destruidos todos los enemigos (en lugar de intentar sobrevivir en un mismo nivel), crear nuevos enemigos con distintos patrones… Sin embargo, para ello haría falta dibujar o conseguir *sprites* adicionales para el nuevo contenido, y es una tarea que no he considerado relevante para el proyecto actual.

No soy el propietario de los *sprites* utilizados en este ejemplo, por lo que la posibilidad de mercado de Shooting Robots está descartada a menos que se reemplacen las imágenes. Sin embargo, dado que el objetivo del proyecto era el aprendizaje de Unity y he conseguido implementar abundantes mecánicas, considero el ejemplo un éxito, y siento que he interiorizado los fundamentos de Unity.

5

Conclusiones

5.1. Resultados obtenidos

Se ha creado un juego en 3D completo partiendo de la base de un tutorial, y un juego en 2D de tipo arcade más avanzado. Se ha aprendido sobre el funcionamiento de Unity, y obtenido valiosa experiencia con un nuevo lenguaje de programación (C#).

En Unity, se ha conseguido animar tanto mediante físicas como con un control más personalizado. Se ha aprendido a crear animaciones y efectos visuales en 2D, y a añadir música y efectos sonoros. Se ha desarrollado la capacidad de diseñar sistemas y enemigos moderadamente complejos, y se han conocido las posibilidades y limitaciones del motor de juego Unity.

5.2. Posibilidades de mercado

No interesa lanzar al mercado los juegos creados en este proyecto por varias razones: Keep Rolling, al estar basado en el primer tutorial de Unity, cuenta ya con infinidad de variaciones gratuitas publicadas por amateurs dando sus primeros pasos, al igual que yo. Por ello cabe la posibilidad de publicar el juego en repositorio público para su libre uso, pero las posibilidades de comercialización son nulas.

Shooting Robots en cambio utiliza *sprites* de terceros, por lo que, aunque pueda usarlos para fines académicos, no tengo los derechos para sacar beneficio económico de ellos. Por eso, Shooting Robots tampoco podría comercializarse a menos que cambiase sus *sprites* por propios.

5.3. Futuro y mejoras

Debido a las limitaciones recién mencionadas estos dos juegos ofrecen pocas posibilidades de evolución, pero con el conocimiento adquirido desarrollando estos dos ejemplos, será posible diseñar y llevar a cabo proyectos más complejos a un más largo plazo, y profundizar así aún más en el aprendizaje y desarrollo con Unity.

Bibliografía

[1] Video Games Bigger than the Movies?: *www.gamesoundcon.com/single-post/2015/06/14/Video-Games-Bigger-than-the-Movies-Dont-be-so-certain*

[2] 2017 Video Game Trends and Statistics: *www.bigfishgames.com/blog/2017-video-game-trends-and-statistics-whos-playing-what-and-why*

[3] Unity Store: *store.unity.com/es*

[4] Roll-a-Ball tutorial: *unity3d.com/es/learn/tutorials/s/roll-ball-tutorial*

[5] Super Monkey Ball: *en.wikipedia.org/wiki/Super_Monkey_Ball_(video_game)*

[6] Super Mario 64: *en.wikipedia.org/wiki/Super_Mario_64*

[7] Páginas de recursos de *sprites*: *www.spriters-resource.com, spritedatabase.net*

Anexo A: Fundamentos de Unity

En este anexo se explicarán los fundamentos de Unity para ayudar a comprender las partes técnicas del proyecto. Se abarcan tanto la metodología de programación en Unity como los distintos componentes de los objetos creados, los métodos más habituales y las distintas opciones del editor.

A.1. Metodología de programación

El planteamiento de los scripts y la programación en Unity en general es algo distinto a las aplicaciones tradicionales (generalmente). En lugar de ejecutar una secuencia de comandos al abrir la aplicación o reaccionar a las acciones del usuario, cuando se ejecuta un juego se calcula y renderiza todo lo que hay en él varias veces por segundo, una vez por fotograma a mostrar.

Cada fotograma o, de ahora en adelante, *frame*, se llamará a todos los scripts activos en el juego, y se ejecutarán determinados procedimientos una vez por *frame*. Además, el propio Unity realizará sus cálculos automáticamente tales como las físicas, la iluminación…

Debido a esto, hay un pequeño periodo de adaptación al programar para Unity ya que se trata básicamente de hacerlo siempre dentro de un bucle, incluso cuando no tendría por qué ser necesario hacerlo.

A.2. Componentes de Unity

Para crear una escena o nivel del juego, es necesario crear "objetos" en el editor. Estos objetos son contenedores a los que añadiremos los componentes necesarios para crear el elemento que queramos o necesitemos. Todos los elementos como un personaje, terreno, enemigos, cámara, etc. son objetos en el editor.

Al crear un objeto tendremos opciones como crear un Objeto 3D (cubo, esfera, plano…), un Objeto 2D (*sprite*), una Luz, una Cámara o un objeto vacío, entre otros. Sin embargo, las propiedades de un objeto las definen sus componentes. Es posible cambiar todos los componentes de un objeto 3D y convertirlo en una cámara, por ejemplo.

Es posible crear jerarquías de objetos, y no hay dependencias forzadas. Es decir, será posible asignar la cámara como hija de un personaje o un personaje como hijo de la cámara. Aunque es común usar contenedores vacíos como "carpetas" en las que agrupar distintos objetos, como por ejemplo todos los elementos del terreno, esto no es imperativo y en algunos casos es recomendable mantener una relación *parent-child* entre dos objetos en lugar de agruparlos bajo un mismo contenedor.

Si vamos a necesitar múltiples copias de un mismo objeto (monedas, enemigos, proyectiles…) podemos designar un objeto como *prefab* o "plantilla", y bastará con arrastrarlo desde el editor para crear varias copias.

Los siguientes son los componentes más relevantes en los objetos de este proyecto:

- *RigidBody* 2D/3D: este componente añade propiedades físicas como la masa, fricción, si es afectado por la gravedad, etc. a un objeto. Si un objeto tiene este componente se verá afectado por las fuerzas físicas, y podrá caer por la gravedad, ser desplazado por colisiones, etcétera. Se puede desactivar el efecto de la gravedad en él con un ajuste en el editor.

  Es posible mover un objeto sin *RigidBody* en juego mediante un script, pero para ello habría que alterar su posición directamente, ya que no será posible aplicar fuerzas sobre el objeto.

- *Collider* 2D/3D: los *colliders* o colisionadores definen el área física que un objeto ocupará en el juego. Pueden tener forma de cubo, esfera, cápsula… tanto en 2D como en 3D. Es posible crear una forma personalizada que se adapte exactamente a la apariencia del objeto, pero tiene un coste computacional mucho mayor.

  Dos objetos con *colliders* corrientes (ver siguiente punto) colisionarán, y no podrán atravesarse.

- *Trigger* 2D/3D: un *trigger* o disparador es un tipo de *collider* especial que no provocará colisiones. En lugar de ocupar un espacio físico, éste podrá ser atravesado y contará con métodos distintos en los script*s* para controlar dichas interacciones (ver an. A.3) .

  Un objeto solo podrá tener un *collider* o *trigger* propio, pero cada uno de sus hijos podrá tener otro más.

- *Mesh* (3D) / *Sprite* (2D): una *mesh* o malla dictará la apriencia de un objeto 3D. Puede ser una figura geométrica básica como un cubo o un cilindro, o un modelo complejo importado al editor.

  En el caso del 2D, los *sprites* o imágenes cumplirán la función de las *mesh*. Estas imágenes han de ser importadas, y se organizarán por *layers* o capas para determinar qué se mostrará por encima de qué.

- *Animator* (2D): el animador nos permitirá controlar las animaciones de un objeto del juego. Es una herramienta en la que crearemos las secuencias de *sprites* que formarán cada animación, y un grafo con transiciones para controlar qué se reproducirá en cada situación. Para determinar la animación adecuada comprobará varias variables que definiremos y que controlaremos mediante *scipts*.

- Script: un script será un archivo de código que añadiremos al objeto para controlar su comportamiento en ejecución. Unity admite los lenguajes C# y JavaScript, pero este proyecto se ha programado en su totalidad en C#.

  Un mismo script podrá utilizarse en varios objetos a la vez, y un objeto podrá tener varios script*s.*

A.3. Métodos y procedimientos habituales

Para escribir los script*s*, Unity ofrece varios métodos y procedimientos predefinidos. Es necesario entenderlos y hacer uso de ellos para poder hacer script*s* válidos. A continuación, se explicarán las peculiaridades y los requisitos para la programación en Unity.

- Variables públicas/privadas: su funcionamiento es similar al de las variables púbicas y privadas en la programación tradicional: las públicas se pueden acceder desde otras clases (o script*s*, en este caso), y las privadas no, a menos que implementes métodos públicos para ello.

  En Unity, sin embargo, tienen una diferencia más. Las variables públicas se mostrarán en el editor si seleccionamos su script, y se podrán cambiar mientras se ejecuta el juego. Gracias a esto será posible probar distintos valores sin tener que detener el juego, editar el script, guardar los cambios y volver a ejecutarlo.

  Además, si queremos definir un elemento externo (ya sea cualquier otro componente u objeto) en el script, si esta variable es pública bastará con arrastrar dicho elemento sobre la variable en el editor para fijarlo, y no será necesario buscarlo mediante el script cuando se ejecute el juego, acción que puede resultar costosa en algunos casos.

- *Start()*: el procedimiento *Start()* será lo primero que se ejecute siempre que se cargue un script, ya sea cuando se ejecute el juego si el objeto comienza activo (el personaje) o cuando se instancie su objeto (un proyectil).

  Este procedimiento se usa para inicializar variables, buscar objetos en el juego o iniciar contadores, entre otros.

- *Update()* y *FixedUpdate()*: como se explicó al principio de este anexo (en A.1), los script*s* son llamados varias veces por segundo en Unity. Cada vez que son llamados, se ejecutarán los procedimientos *Update()* y *FixedUpdate()*.

  La diferencia entre estos dos es que *FixedUpdate()* siempre se ejecutará a intervalos regulares, independientemente del *framerate* o fotogramas por segundo del juego, mientras que *Update()* se ejecutará siempre una vez por *frame*.

  Por esta razón, es altamente recomendable reservar *FixedUpdate()* para casos concretos como aplicar una fuerza continua a un objeto que requieran esa precisión, y realizar el resto de las acciones como una fuerza instantánea para un salto o que no requieran de la física en *Update()*, para no causar comportamientos no deseados.

- Colisiones: si el objeto que controla el script o uno de sus hijos tiene un *collider* como componente, Unity ofrece dos métodos para gestionar las colisiones: *OnCollisionEnter(Collision)*, y *OnCollisionExit(Collision)*. El primero se llamará cada vez que el *collider* colisiones con otro *collider*, pero no si el otro es del tipo *trigger*. El parámetro que recibirá el método será del tipo "*Collision*", que contendrá tanto el objeto con el que ha colisionado como la velocidad y puntos de contacto.

OnCollisionExit(Collision) se llamará cuando otro objeto que esté en contacto con el del script deje de estarlo. Al igual que el primer método, el parámetro que recibirá será del tipo *Collision*.

Un script solo podrá controlar un *collider* o *trigger*, por lo que no es recomendable asignar más de uno de éstos a cada objeto. Emparentando varios objetos y scripts, sin embargo, es posible crear estructuras más complejas.

- *Triggers*: si el *collider* del objeto es del tipo *trigger* no podrán usarse los métodos del punto anterior, pero a cambio tendremos disponibles *OnTriggerEnter(Collider)*, *OnTriggerExit(Collider)* y *OnTriggerStay(Collider)*. Los dos primeros funcionan casi como sus variantes para las colisiones, con la diferencia de que recibirán como parámetro el *collider* (ya sea otro *trigger* o un *collider* normal) del otro objeto, ya que al tratarse de un *trigger* no habrá una colisión como tal.

  El último método, *OnTriggerStay(Collider)*, se llamará cada *frame* mientras el otro objeto interseque con el *trigger*.

- Corrutinas: las corrutinas son funciones especiales que devuelven la clase de Unity *IEnumerator*. Sirven para implementar funciones que se ejecuten a lo largo de varios *frames*, o tras esperar cierto tiempo.

- Métodos corrientes: además de las funciones recién mencionadas, es posible definir métodos propios en cada script (ya sean públicos o privados) que se podrán ejecutar en cualquiera de los dos *updates*.

Anexo B: *Hojas de sprites*

En este anexo se pueden ver todas las imágenes y hojas de *sprites* utilizadas en este proyecto.

B.1: Fondo utilizado en Keep Rolling.

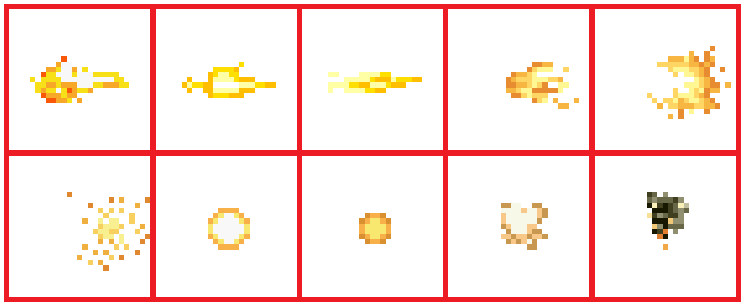
B.2: Hoja de *sprites* del personaje de Shooting Robots.

B.3: Fondo de Shooting Robots.

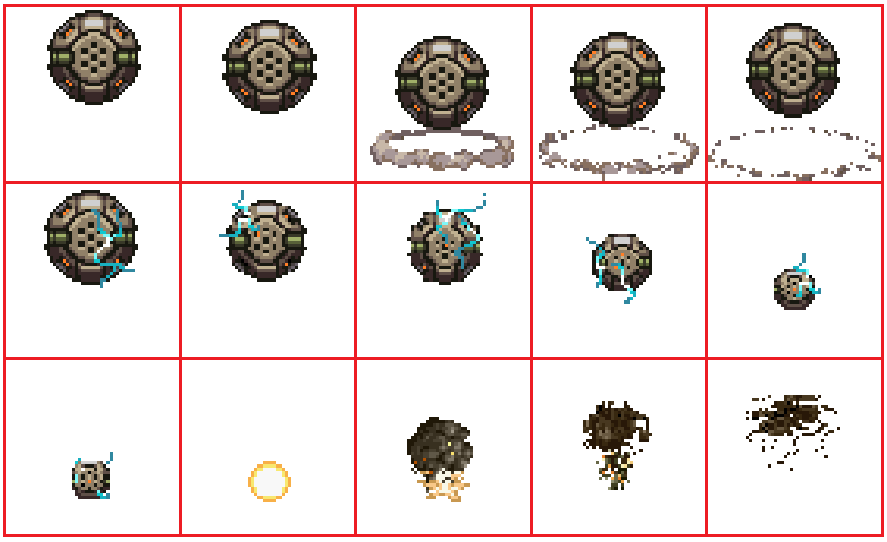
B.4 y B.5: Detalles del fondo de Shooting Robots.

B.6: HUD de Shooting Robots.

B.7: Fuego de Shooting Robots.



B.8: Proyectiles de Shooting Robots.



B.9: Enemigo para Shooting Robots.