2018

IES San Vicente

Dionisio Giménez Vera

[HALLOWEEN PARTY]

Halloween Party es un juego de plataforma 2D inspirado en los juegos clásicos de los 80 y 90

1. Introducción:

Este proyecto se basa en el desarrollo de un juego de plataformas 2D para dispositivos Android, para ello se ha utilizado el motor de juegos Unity 3D y la realización de la lógica del juego se ha desarrollado en C#.

Hoy en día hay cantidad de juegos para móviles, pero, a mí juicio, se ha perdido un poco la esencia de los primeros juegos, juegos sencillos, fáciles de jugar, que proporcionaban diversión sin tener que conocer complicados movimientos y memorizar montones de objetos que tienen una funcionalidad específica.

Lo que se pretende realizar es un juego clásico, basado en los juegos de las décadas 80's y 90's pero utilizando tecnologías modernas, a día de hoy, se siguen comercializando imitaciones de máquinas arcade y consolas de aquella época, además de los incontables emuladores que existen.

Para la realización de este proyecto se han definido una serie de objetivos que se detallan continuación:

- Crear la lógica necesaria para controlar al personaje principal, movimientos, daño causado por los enemigos, muerte del personaje, así como las animaciones necesarias y las condiciones para pasar de una animación a otra.
- Crear 8 pantallas de juego basadas en 4 escenarios diferentes, cada 2 pantallas el personaje se enfrentará a un enemigo final.
- Crear un ítem para que el personaje sume puntos, que desaparezca al cogerlo y emita un sonido.
- Crear al menos 6 enemigos básicos con diferentes lógicas de movimiento para cada uno de ellos.
- Crear 4 enemigos finales, para cada uno de ellos se implementará una lógica de movimientos y de ataque diferente, siendo más resistentes que los enemigos básicos será necesario golpearlos varias veces para acabar con ellos.
- Crear una pantalla de inicio donde se podrá elegir entre iniciar un nuevo juego y continuar con un juego guardado, si se selecciona esta opción el juego comenzará en la primera pantalla del bloque donde hayamos muerto.
- Crear una pantalla de muerte, en ella cada vez que el personaje muera se nos mostrará la puntuación obtenida y un botón para regresar a la pantalla de inicio.
- Crear una pantalla para mostrar entre los diferentes bloques del juego, en ella se mostrará un mapa o similar para ver los bloques restantes y un botón para continuar.
- Crear una serie de objetos que serán los que establezcan la condición de victoria en cada bloque, después de eliminar al enemigo final estos aparecerán y al cogerlos nos llevaran a la pantalla del mapa.
- Crear una pantalla para mostrarla cuando finalice el juego, en ella aparecerá la puntuación obtenida y un botón para volver a la pantalla de inicio.

A lo largo de esta memoria veremos antecedentes de juegos reales en los que me he inspirado para realizar el proyecto, se expondrá el análisis del proyecto, se verá en detalle el diseño del mismo explicando las diferentes fases por las que ha pasado el proyecto, se resumirán las conclusiones obtenidas de la realización del mismo y se aportará la bibliografía que se ha utilizado para desarrollarlo.

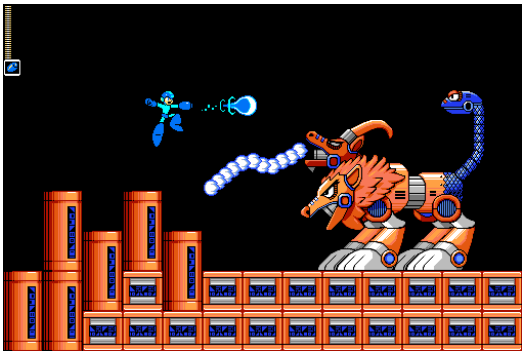
2. Antecedentes:

Super Mario World:

Para mí, el juego por excelencia, en parte la realización de este proyecto se debe a este juego en particular, cumple con todos los requisitos que a mi juicio ha de tener un juego, sencillo, jugable, y en mi caso adictivo.

MegaMan:

Otro clásico que proporcionó incontables horas de diversión, al igual que Super Mario World, era un juego en el que estaba claro desde el principio y el objetivo, en 2D donde solo se puede avanzar en una dirección y al final de cada bloque un enemigo final.



Y así podríamos enumerar incontables de ellos, pero a día de hoy son pocos los juegos que toman como base este principio de que prime la jugabilidad, y no el conseguir más puntos para aparecer arriba en las tablas de rankings de las redes sociales.

3. Análisis:

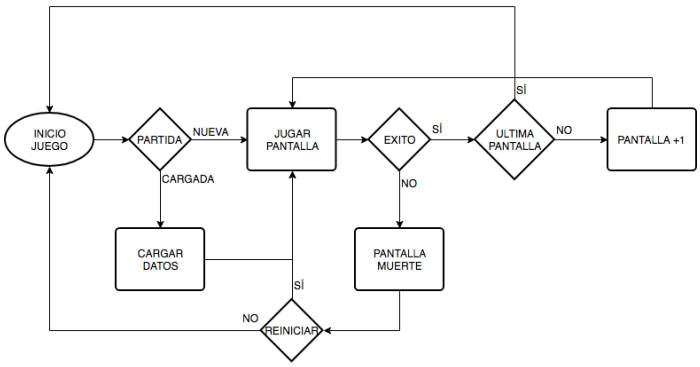
El objetivo es crear un juego de plataformas clásico, en el que prime la jugabilidad, con movimientos sencillos y rápido aprendizaje de las características para proporcionar diversión y entretenimiento al usuario desde los primeros momentos de uso.

Ha de seguir un guion, con unos objetivos claros para conseguir finalizar el juego, una historia en la que el personaje se mueve y consigue pasar de nivel y así estar más cerca del objetivo final del juego.

La historia del juego se basa en un esqueleto que recupera partes de su novia que ha sido raptada por el demonio de Halloween, para ello debe recorrer diferentes escenarios, dentro de estos escenarios encontramos enemigos que intentan evitar que el jugador logre su objetivo, estos enemigos se han diseñado para que estén acorde al escenario en el que se encuentran.

Al tratarse de un juego de plataformas, las partidas pueden prolongarse bastante, para evitar el desistimiento del jugador se ha añadido la posibilidad de recuperar partidas guardadas.

Para no perder la esencia de los juegos antiguos, solo se puede tener una partida guardada, si iniciamos otra se sobrescribiran los datos de la partida anterior.



Para una mejor experiencia de juego, el jugador puede sumar puntos, para ello debe de matar los enemigos que encuentre a su paso y dispone de ítems que puede recoger.

Para conseguir que el juego sea más entretenido, la dificultad aumenta en cada escenario, esto se consigue aumentando el número de enemigos y la dificultad para matarlos, haciéndolos más resistentes y con lógicas de movimiento más elaboradas.

4. Diseño:

Para el desarrollo de este proyecto se ha elegido Unity3D como motor de juegos, aún a pesar de usarse sobre todo para el desarrollo de juegos en 3D ofrece todo lo necesario para juegos en 2D, incluyendo opciones de desarrollo específicas para este tipo de juegos.

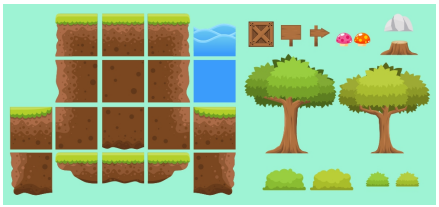
Unity ofrece la posibilidad de desarrollar los scripts en javascript o C#, para el desarrollo de este proyecto se ha optado por esta última.

4.1 Diseño de escenarios.

Para el diseño de las pantallas o "Scenes", Unity ofrece la posibilidad de usar uno o varios "TileMaps" dentro de cada "Scene".

Para poder usar estos componentes se hace necesario disponer de uno o varios TileSet.

Un TileSet es un conjunto de imágenes que casan perfectas entre ellas con una relación de aspecto cuadrada, de forma que si las ponemos una a continuación de otra forman un escenario uniforme.



TileSet

Esto nos permite crear un "TilePallete" y crear nuestra pantalla simplemente dibujando con el cursor el cuadrado seleccionado, a través de esta tecnología se hace más sencillo la creación de pantallas, ya que nos permite plasmar la pantalla realizada en la diseñada con facilidad.

Para el desarrollo de las pantallas se crearán materiales para simular ciertos efectos durante el juego.

Las pantallas de mapa son de transición, solamente para que el usuario conozca su progreso en el juego, por ello, solo se mostrarán y reproducirán una música y quedando a la espera de que el usuario toque la pantalla para continuar.

Elementos dinámicos:

Dentro de las pantallas se incluirán elementos dinámicos que no pueden ser plasmados a través de los TileMaps, para ello se crearán GameObjects con un script para dotarlos de lógica de movimiento.

Cámara:

La configuración de la cámara será la misma para todas las escenas, se ha elegido una cámara Ortográfica que contendrá diferentes elementos asociados a ella, para que se muevan con esta.

Ítems:

En todas las pantallas se añadirán ítems que el jugador puede coger para sumar puntos, recargar vida y pasar de nivel.

Controlador táctil:

Para el manejo del personaje se agregará a la escena un elemento Canvas que contendrá los botones de dirección, salto y ataque.

4.2 Personaje principal.

El personaje principal será un esqueleto, todas las imágenes que se tilizarán para el desarrollo del juego han sido descargadas de una página web (www.craftpix.com), en ella encontramos diferentes caracteres gratuitos y de pago, para el desarrollo del juego decidí crear una cuenta Premium en esta web para tener acceso ilimitado a todos.

Para los movimientos de este se crearán una serie de métodos públicos a los que llamarán los botones que contiene el Canvas, estos métodos asignarán un valor booleano a una serie de variables definidas para el movimiento del personaje.

Estas variables se comprobarán en el método Update() que ejecuta Unity de forma cíclica, realizando los movimientos necesarios.

Componente Animator:

Una de las claves del dinamismo del juego son las animaciones y las transiciones entre ellas, para ello Unity nos ofrece un componente con el que gestionar ambas.

Las animaciones en un juego son una parte fundamental para adquirir una vistosidad y dinamismo altos, en un proyecto real serían competencia de los diseñadores gráficos.

En la mayoría de los casos se pueden crear con unos pocos fotogramas del carácter que se pretende animar, formando un clip, pero esta solución fue descartada porque quedan poco reales según la investigación realizada.

En los caracteres descargables de internet suelen venir estos fotogramas para poder crear la animación, pero una opción más recomendable es realizarlas con la herramienta que ofrece Unity, que normalmente multiplica casi por 10 los fotogramas usados, ya que solo generando 3 o 4 frames clacula las posiciones intermedias del carácter y los completa generando automáticamente el resto.

Otra alternativa es crear la animación con un programa externo, como Spriter, y luego importarlas en el proyecto.

4.3 Enemigos básicos.

Al igual que el personaje principal, todos los enemigos que se usarán para las diferentes pantallas provienen de la web www.craftpix.com, el desarrollo de estos es parecido al del jugador, con la diferencia de han de realizar movimientos por si mismos.

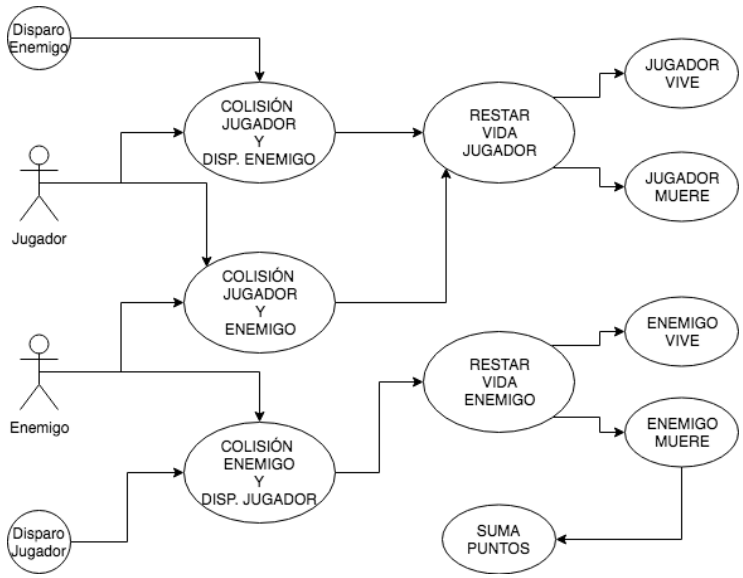
He diseñado una serie de movimientos para los enemigos, para conseguir el efecto de un juego clásico de plataformas, estos siguen un patrón fijo de movimiento.

Se ha diseñado una estructura de variables y métodos que implementan todos los enemigos del juego para manejar su vida, la comunicación entre el enemigo y el personaje principal, y entre el enemigo y el marcador de puntos.

Todos los enemigos tienen en común estos 4 atributos:

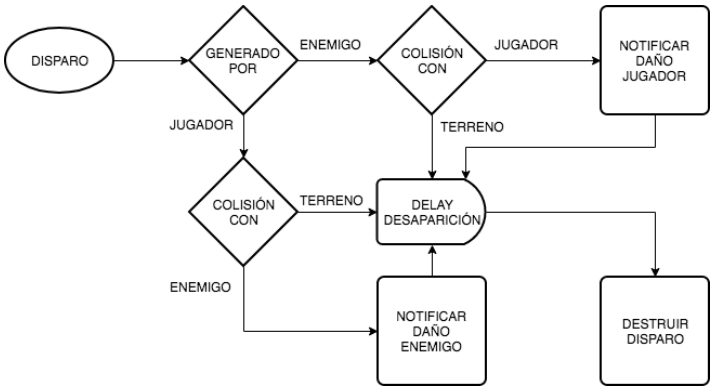
- Vida: Cantidad de colisiones necesarias para acabar con ellos.
- Puntos por muerte: Es la puntuación que obtiene el jugador por acabar con el enemigo en cuestión, gracias a ella podemos premiar al jugador con más puntuación al acabar con un enemigo más difícil de matar.
- Daño Ejercido: Es el daño que el enemigo inflige al jugador al tocarlo.
- Delay Daño: Esta variable es el tiempo en segundos que el enemigo tarda en volver a hacer daño al jugador si nos mantenemos en contacto con él.

Para explicar el proceso de se muestra el diagrama de casos de uso:



El script diseñado para los disparos es común para ambos casos, en su configuración se podrán seleccionar a parte del daño que inflige al GameObject con el que colisiona, el retardo de desaparición y si este proviene de un enemigo o del

jugador, en función de la selección el disparo se comporta de una manera u otra, a continuación se expone el diagrama de flujo de la lógica del disparo.



4.4 Enemigos finales.

Al final de cada escenario el jugador se enfrentará a un enemigo de final de bloque, estos enemigos serán más resistentes que los enemigos básicos y tienen una lógica de movimiento independiente para cada uno de ellos.

Al igual que los enemigos básicos dispondrán de la misma estructura de atributos y métodos para comunicarse con el jugador y el marcador que se ha detallado anteriormente, además todos ellos incluirán un nuevo atributo del tipo GameObject que sirve para indicarle el objeto que han de crear al morir, que dará acceso al siguiente nivel.

La dificultad de estos reside, a parte de la mayor resistencia, en una lógica más compleja, todos ellos generarán objetos en tiempo de ejecución para atacar al jugador.

Todos los enemigos finales permanecen pasivos hasta que el jugador atraviese un GameObject que solo contendrá un Collider y el script encargado de notificar al enemigo que ha de ponerse en marcha, para ello se hace uso de la clase NotificationCenter.cs (ver apéndice A), el enemigo se suscribe a la notificación "ActivarEnemigoFinal", cuando cualquier GameObject ponga una notificación en ella desde el script del enemigo final se llama a un método que ha de tener el mismo nombre que la notificación a la se suscribe.

4.4.1 Desierto:

Para el escenario del desierto se ha elegido como enemigo final una momia, al ser el primer enemigo final del juego se pretende que sea más resistente pero cuya dificultad resulte asequible, por ello se ha diseñado una lógica con un patrón sencillo,

el enemigo salta en la dirección en la que se encuentra el personaje y posteriormente dispara tres objetos que se desplazan a una velocidad constante y siempre en las mismas direcciones (hacia arriba, hacia delante y en diagonal hacia arriba).

4.4.2 Bosque:

Para el segundo escenario se ha elegido un carácter de un árbol, en esta ocasión el enemigo se desplazará de un lado a otro de la pantalla lanzando una bola de hojas.

La creación de un "Phisics Material" especifico con un mayor rebote junto con el RigidBody, harán que el disparo no tenga un patrón de movimiento fijo, sino que rebote por la escena de manera aleatoria.

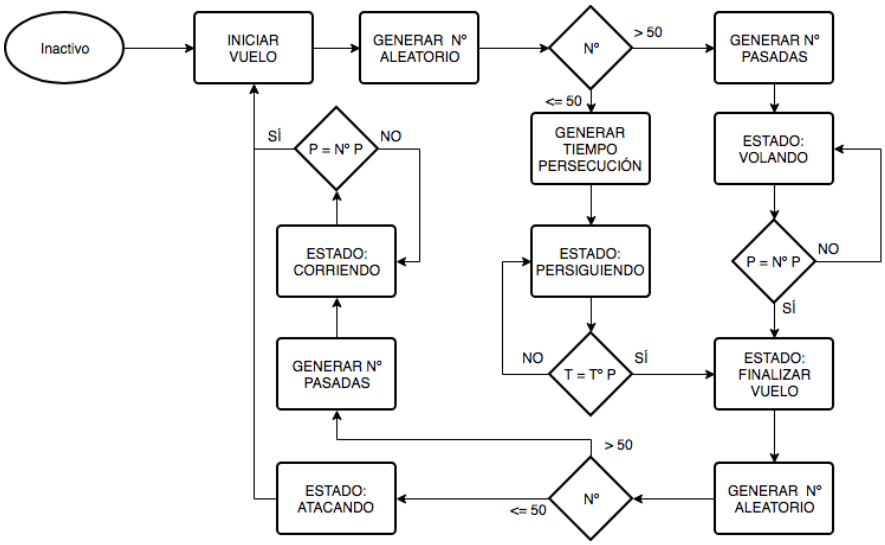
4.4.3 Nieve:

Para el enemigo perteneciente al tercer escenario se ha optado por un carácter que emula un Golem de hielo, para aumentar la dificultad con respecto al enemigo anterior, aparte del movimiento de lado a lado de la pantalla se ha agregado al patrón de movimiento un método que genera tres bloques de hielo que caen desde la parte superior de la pantalla.

4.4.4 Cementerio

Para el último enemigo final del juego se ha elegido un demonio, al tratarse del enemigo final este ha de ser el más difícil de todos, para conseguirlo se ha diseñado una lógica de movimientos más compleja, a través de su script asociado, el enemigo realizará tres movimientos y dos ataques diferentes.

Para mejor comprensión se adjunta un diagrama de flujo con los posibles cambios de estado del enemigo.

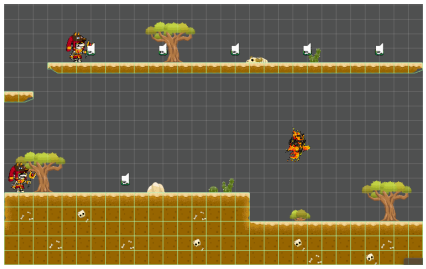


5. Resultados:

5.1 Escenarios.

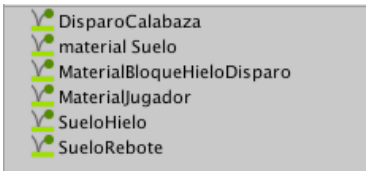
En este proyecto se han realizado varios “TileMaps” por pantalla para así poder configurarlos de forma diferente en función de su cometido.

TilePallete de uno de los escenarios



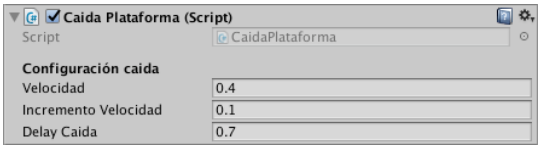
Ejemplo diseño de una pantalla

Unity permite la creación de materiales y el ajuste de su fricción y su rebote, consiguiendo efectos como una cuerda elástica o un bloque de hielo, se han creado varios para mejorar la experiencia de juego.



<u>Elementos dinámicos:</u>

Se han creado diferentes elementos para según qué escenario y se han desarrollado los scripts configurables para adaptarlos a las necesidades de uso.

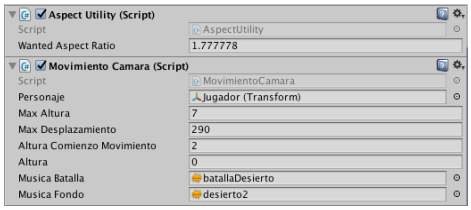


Las pantallas de mapa del juego no se han creado de esta manera, para la creación de estas se han dispuesto en la pantalla las imágenes necesarias y el carácter del jugador en la posición correspondiente.

Cámara:

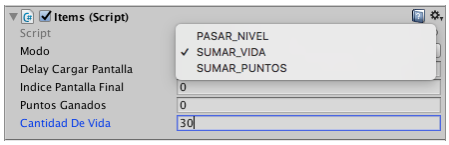
Los elementos que se han añadido a la cámara son:

- Dos destructores de objetos, uno a la izquierda de esta y otro por debajo, su función es la de ir destruyendo de la escena los enemigos que ya han quedado atrás o han caído por algún hueco. También se encargan de matar al personaje si este cae, lo hacen gracias a un script asociado a ellos.
- Fondo de pantalla, se mueve con la cámara y tiene un script que hace que se mueva simulando el avance del personaje.
- Un script que mantiene la relación de aspecto de la pantalla de juego independientemente del tamaño de pantalla que se use (Ver apéndice A, AspectUtility).
- Un script para el movimiento, en él se pueden configurar los límites de movimiento de esta, a que gameObject ha de seguir y la música que sonará en la pantalla.



Items:

Se ha creado un único script configurable para todos ellos, en él se puede elegir que acción debe realizar el ítem al que se lo asignamos a través de un desplegable y asignar los valores correspondientes.



Controlador táctil:

En el Canvas que actúa como controlador también se encuentran:

- La barra de vida del personaje. Esta barra de vida se ha descargado del AssetStore de Unity de forma gratuita, simplificando la visualización de la vida del personaje, esta expone un método estático para actualizar el valor de la barra y pintarla correctamente.
- El contador de tiempo, si el tiempo se acaba el personaje muere y vamos a la pantalla de GameOver.
- El marcador de puntuación.

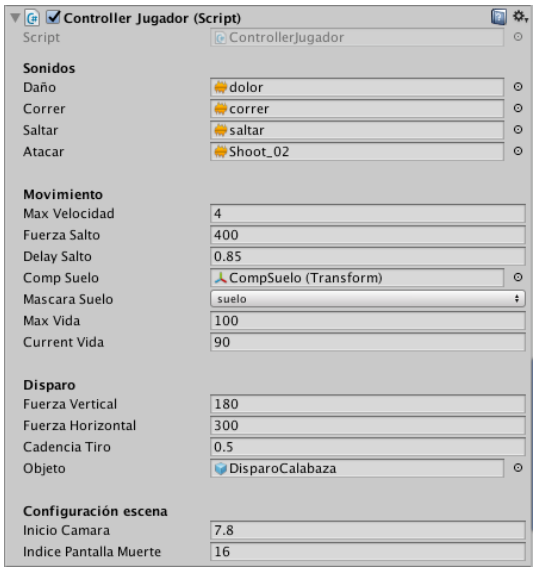
Los botones son imágenes libres de derechos descargadas de www.opengameart.org

5.2. Personaje principal.

Se ha añadido al personaje principal un componente RigidBody, para el movimiento en el eje X me decantado por la propiedad velocity del RigidBody que a mi juicio proporciona un movimiento más natural teniendo cierta inercia al dejar de pulsar los botones de movimiento. Para realizar el salto se utiliza la función AddForce del mismo componente, de esta manera la subida del personaje viene definida por la fuerza aplicada y la caída se realiza por gravedad gracias a la librería Physics2D que obtiene unos muy buenos resultados en el comportamiento de los objetos.

Para el ataque en un principio se realizó un ataque de corto alcance, ya que el sprite usado para el personaje disponía de una espada y un escudo, pero, para mejorar la jugabilidad, se eliminaron estos componentes de las animaciones y decidí un ataque arrojando objetos, en este caso, siguiendo la tónica del juego, el personaje lanza calabazas de Halloween.

A continuación se detalla las opciones de configuración del script asociado al GameObject del personaje principal.



Sonidos: Sonidos que se reproducen durante el juego.

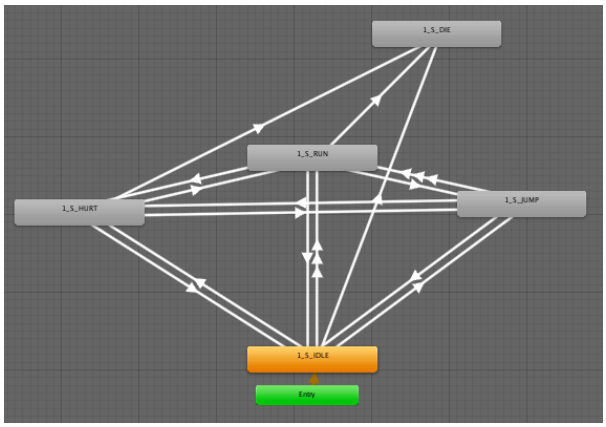
Movimiento: Aquí se definen las variables de movimiento horizontal y vertical así como la vida del personaje y los componentes que hacen posible saber si el personaje está tocando el suelo.

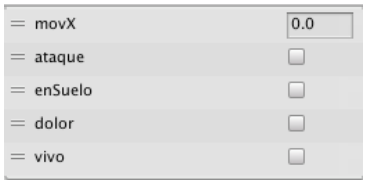
Disparo: Para definir que objeto se va a usar como disparo y la fuerza con la que va a ser lanzado.

<u>Componente Animator:</u>

Para esta parte del proyecto se ha añadido un proyecto externo, Spriter2UnityDX (ver apéndice A), que tiene la capacidad de reconocer los archivos .scml creados por los programas de creación de animaciones externos y generar un Prefab con todas las animaciones y el controller necesario para gestionarlas.

Para manejar las transiciones de las animaciones se han creado las variables necesarias en el script y se han asignado al componente animator del personaje, para que funciones hay que crear las relaciones entre estas en el controller.





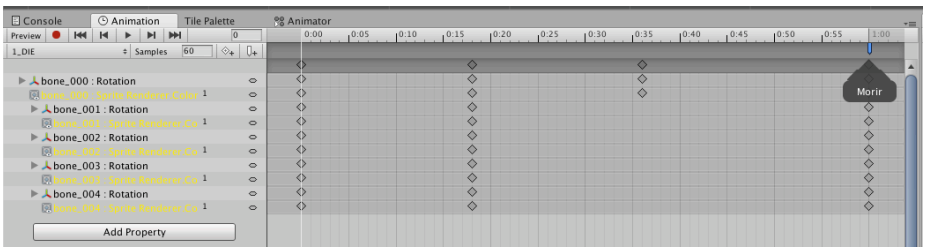
Variables transiciones

Controller personaje Principal

5.3. Enemigos básicos.

Se han realizado 8 lógicas de movimiento diferentes que se utilizan en un total de 16 enemigos que podemos encontrarnos a lo largo de los 4 escenarios del juego, para la configuración de estos, se ha seguido el mismo método de configuración que para el personaje principal, creando variables públicas que aparecen en el inspector de Unity y a las que se les puede asignar un valor para cambiar el comportamiento de estos.

Al igual que ocurre con el jugador principal los enemigos disponen de un componente Animator que se encarga de las transiciones entre animaciones, además en el caso de estos se han añadido eventos a las animaciones que actúan como triggers haciendo más sencilla la sincronización con los disparos que realizan o la muerte de estos.



Ejemplo de una animación donde se observa que antes de finalizar dispara el evento morir

Estos eventos llaman a métodos que se encuentran en el script que tiene asociado el enemigo al que pertenece la animación.

Todos los enemigos detectan la posición del jugador en la escena y se activan cuando están dentro del rango que configuramos, de esta manera permanecen parados en la posición original hasta el momento adecuado.

Para simplificar el proceso de añadir los enemigos a las escenas se crean Prefabs, que arrastramos a la posición deseada en las escena.

Con la intención de reducir la duplicación de código, se ha creado una clase (UtilidadesMovimiento.cs) con una serie de métodos estáticos que exponen una serie de funcionalidades de movimiento que pueden usar todos los enemigos.

Estos métodos abarcan los movimientos más generales que de otra forma habría que repetir en cada script, así como el método "Voltear" que es válido para cualquier GameObject, y tiene como función girar el Sprite 180º en su eje X.

5.4 Enemigos finales.

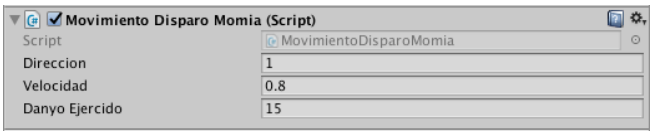
A continuación se detalla el resultado de los enemigos y sus movimientos así como la configuración que se puede realizar sobre ellos y los ataques que realizan.

5.4.1 Desierto:

Este enemigo tiene un componente cuya función es detectar cuando está en el suelo, se hace necesario para que no intente realizar un salto cuando todavía no está tocando el suelo.

Para ello hay que indicarle también la capa con la que ha de colisionar para que el valor sea true.

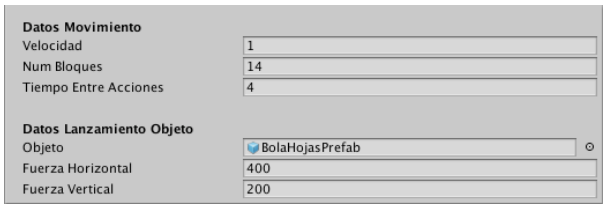
Para los disparos se ha creado un script específico, donde a través de un entero se le puede indicar en qué dirección debe desplazarse al crearlo.



A continuación se muestran las posibilidades de configuración que aparecen en el inspector para ajustar los valores que definen el movimiento.

5.4.2 Bosque:

Para la bola de hojas que hace las veces de disparo del enemigo no se ha creado un script específico, se ha utilizado el Disparo.cs genérico poniendo a true la opción de "Disparo Enemigo". En la siguiente captura se muestra la posible configuración del carácter, se han omitido las variables comunes a todos los enemigos.



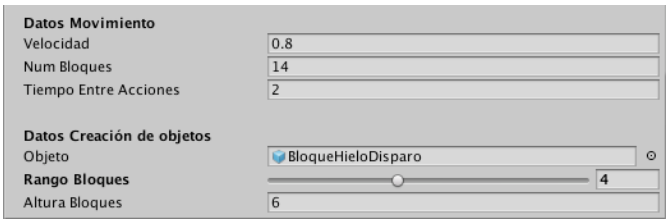
El objeto que lanza está creado por medio de un editor de fotografías externo a partir de las hojas que componen al propio carácter, de esta manera se consigue que al producirse la animación de ataque, parezca que saca la bola de su propio cuerpo.

Ajustando el "Order in layer" del componente "Sprite renderer" a una posición menor que el carácter del enemigo conseguimos que el objeto se cree por detrás de este y no aparezca en la pantalla de repente, la bola de hojas generada tiene un componente RigidBody 2D, el cual hace que le afecten las leyes del motor de física del juego, así podemos añadir una fuerza vertical y horizontal para el movimiento.

5.4.3 Nieve:

Con el fin de hacer menos previsible el lugar donde van a aparecer los bloques generados desde el techo , se detecta la posición donde se encuentra el personaje y genera los bloques de forma aleatoria dentro de un rango que establecemos a través de variables públicas expuestas en el inspector, de esta manera dado el presumible movimiento del jugador por la pantalla y la creación en posiciones aleatorias de los bloques, se aumentan las posibilidades de que estos impacten con el jugador.

Al igual que la bola de hojas del enemigo anterior estos bloques incluyen el script Disparo.cs y un RygidBody con un material específico que aparte de tener más rebote reducen la fricción simulando bloques de hielo reales.



5.4.4 Cementerio:

Para implementar los estados del enemigo se ha creado una enumeración, de esta manera podemos comprobar en cual se encuentra y realizar el movimiento o ataque correspondiente.

Estados:

- Iniciar vuelo.
- Finalizar vuelo.
- Volando
- Atacando
- Corriendo
- Persiguiendo

Para hacer la batalla más dinámica se han incorporado números aleatorios, que se generan dentro de un rango definido a través de los atributos públicos, para determinar el siguiente estado del enemigo. De esta manera no sigue un patrón fijo de movimientos, sino que se genera un patrón único en tiempo de ejecución.

A continuación, se detallan los posibles estados:

Iniciar Vuelo:

Es el estado al que pasa el enemigo al activarse, en este estado el enemigo emprende el vuelo y se eleva hasta una posición en el eje y que se calcula en base a la altura inicial del enemigo más la cantidad de desplazamiento vertical que le indiquemos.

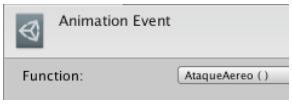
Finalizar Vuelo:

Es el movimiento contrario al anterior, devuelve al enemigo a su altura inicial.

Volando:

Mientras que el enemigo se encuentre en este estado, se moverá de un lado a otro de la pantalla, la cantidad de desplazamiento que debe realizar viene dado por el valor de un atributo público presente en el script (numeroBloques).

Durante este estado se generan GameObjects que contienen un rigidBody 2D y el script Disparo.cs mediante un método al que se llama desde un evento creado en la animación de volar.



Antes de entrar en este estado se asigna un valor aleatorio a la variable "numPasadas", y el enemigo se mantendrá en él mientras que el número de pasadas realizadas sea menor que el valor generado.

Corriendo:

En este estado se realiza el mismo movimiento que en Volando, pero en este caso, como se realiza a ras de suelo no se crean disparos, además, para dificultar que el jugador dispare al enemigo se incrementa la velocidad a la que se realizan las pasadas, para ello se declara una variable pública en el script y se aumenta la velocidad en el porcentaje indicado en ella.

Persiguiendo:

Antes de entrar en este estado se crea un valor aleatorio dentro del rango definido en las variables públicas que determina cuanto tiempo estará en este modo, para que el enemigo persiga al personaje, se utiliza un método estático definido en UtilidadesMovimiento.cs al que tenemos que pasarle el transform al que queremos perseguir, el transform que queremos mover y la velocidad.

Para conseguir un movimiento uniforme del enemigo, este método, comprueba que distancia es mayor entre el enemigo y el jugador, la del eje X o el Y, y al movimiento correspondiente a ese eje le asigna la velocidad constante definida en la variable "velocidad" del scrpit, y para movimiento del otro eje se calcula la velocidad adecuada para que ambos desplazamientos terminen a la vez.

Una vez que termina el tiempo de persecución el enemigo vuelve a la posición inicial para pasar al estado Finalizar Vuelo.

Atacando:

En este estado el enemigo ejecuta un ataque en el que después de mover su bastón se crean explosiones desde el suelo

Cada explosión es un GameObject diferente con un Collider en modo trigger y el script de Disparo.cs ajustado a "Disparo enemigo", la separación entre explosiones y el delay que existe entre ellas a la hora de crearse es configurable desde dos variables públicas definidas en el script, al final de la animación se crea un evento que llama al método que destruye el GameObject.

Precisamente debido al delay entre la creación de objetos, se generaba un problema al ejecutar este método en el hilo principal, ya que hubiera supuesto la paralización del juego mientras dure el ataque, se contempló la posibilidad de crearlos teniendo en cuenta el tiempo, pero se hacía complejo manejar esto y la cantidad creada al mismo tiempo.

Para solucionar el problema en un principio se optó por la ejecución de este método en un hilo diferente, pero Unity ofrece la posibilidad de simular una ejecución multihilo sin necesidad de crearlos realmente a través de las "Corutines".

El uso de una Corutine es muy sencillo, solo debemos hacer que el método que queramos convertir en esta devuelva un IEnumerator, y dentro de este añadir:

```
yield return new WaitForSeconds(t);
```

donde "t" es el tiempo en segundos que queremos que pase entre una ejecución y otra.

Combinando todos los estados y añadiendo el componente de aletoriedad al pasar de uno a otro, conseguimos un enemigo más dinámico y menos previsible, y se consigue incrementar su dificultad.

6. Conclusiones:

6.1. Descripción del desarrollo del proyecto.

Para este proyecto se ha superado el tiempo previsto de desarrollo en alrededor de 25 horas, una vez finalizado y estudiando el proceso del mismo, se ha determinado que el exceso de tiempo invertido se debe en gran medida al diseño gráfico del juego, incluyendo la obtención de recursos gráficos gratuitos o a un precio contenido.

Otro de los mayores problemas es crear el controller del personaje principal, ya que aún a pesar de todos los ejemplos e información disponibles en la red, conseguir un movimiento fluido y una buena interacción con el entorno y los enemigos se hace primordial para una buena sensación de juego.

6.2. Conclusiones trabajo realizado.

El juego resultante se hace entretenido pero de corta duración, para prolongarlo se hacen necesarias más pantallas y escenarios, invirtiendo en recursos para poder crear más escenarios diferentes se podría lograr un juego de 8-10 horas de duración.

En el proyecto se han alcanzado todos los objetivos propuestos, incluso superando alguno como, por ejemplo, el desarrollo de diferentes lógicas de movimiento para los enemigos.

La creación de caracteres propios para el desarrollo del juego se hace inviable en un proyecto de tan pocas horas de duración, e incluso la creación de las animaciones de los caracteres es un proceso que requiere invertir mucho tiempo para un resultado óptimo, por ello conseguir Sprites con animaciones ya creadas reduce significativamente el tiempo de desarrollo.

Con respecto a los enemigos, según la experiencia obtenida, se obtiene mejor resultado creando scripts que admitan configuración en los movimientos y ataques, de esta manera, aunque el desarrollo del script sea más complejo, nos da la posibilidad de reutilizarlos.

6.3. Trabajos futuros.

Como trabajos futuros, se podrían añadir más pantallas a los escenarios que ya hay e incluso añadir más escenarios, otra de las cosas que me hubiera gustado, es crear alguna pantalla oculta y de bonus.

7. Bibliografía:

Para el desarrollo este juego se han utilizado los siguientes recursos como material de apoyo:

- Guía de ayuda oficial de Unity, disponible online en https://docs.unity3d.com/ScriptReference/

- Material de apoyo proporcionado en la asignatura de Programación multimedia y dispositivos móviles de 2º Curso de DAM.

8. Apéndices:

Apendice A: Librerias.

Para el desarrollo de este proyecto se ha utilizado tres librerías ajenas, una para la creación de las animaciones de los caracteres y otra para mantener la relación de aspecto de la pantalla.

- Spriter2UnityDx:

Se trata de un proyecto que se puede descargar de forma gratuita desde GitHub (https://github.com/Dharengo/Spriter2UnityDX), el cometido de esta es generar un prefab a partir de una serie de Sprites y un archivo .scml, este archivo se crea a partir de un software de creación de animaciones externo. Muchos de los caracteres disponibles en la red a menudo traen las partes del personaje y este archivo que contiene todas las animaciones.

La forma de usarla es muy sencilla, tenemos que importar el paquete a la raíz de nuestro proyecto Unity y posteriormente añadir al proyecto, en primer lugar las imágenes que forman el carácter, y después el archivo .scml, aparece una ventana para elegir el tamaño de asset a importar y al darle ok genera el prefab correspondiente al personaje.

Toda la lógica de transición ha de ser implementada, pero el prefab creado ya contiene un componente Animator con todas las animaciones.

- NotificationCenter:

Esta librería en realidad es un fichero que contiene dos clases (Notification y NotificationCenter), a través de ella podemos enviar mensajes entre GameObjects sin necesidad de instanciarlos, el funcionamiento es muy sencillo y aporta reducción y limpieza de código.

Al utilizarse a través de un método estático, solamente tenemos que llamar a este con el nombre de la clase y decir si queremos publicar una notificación, o suscribirnos a una, la única desventaja es que funciona a través de Strings y hay que tener cuidado con los nombres de las notificaciones, si nos suscribimos a una notificación hay que añadir un método al script con el nombre de esta, y se ejecutará cuando se publique una notificación de ese tipo.

En este caso se ha utilizado para abstraer el tiempo de la pantalla del personaje, así la gestión del tiempo queda en manos de la escena y el personaje recibirá una notificación cuando este se acabe.

- AspectUtility:

Al igual que la anterior es un solo fichero, su cometido es el de mantener la relación de aspecto de las cámaras independientemente de la relación de pantalla del dispositivo en el que ejecutemos el juego, el script rellena con dos bandas azules el espacio sobrante de pantalla.

Para utilizarlo solo es necesario añadirlo a las cámaras que usemos en nuestro juego e indicarle que relación de aspecto queremos mantener.

- SumPause:

Esta librería ha sido descargada del AssetStore de Unity, tiene como objetivo gestionar la pausa del juego, a grandes rasgos lo que hace es comprobar el estado del Time.timeScale y variar su valor entre 0 y 1.

Hay que tener en cuenta que con este método solamente se reduce la velocidad del juego a 0, por lo tanto si realizamos movimientos modificando la propiedad position del transform de un GameObject, estos no se detendrán.

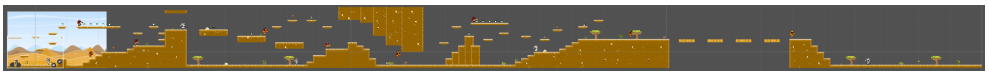
Para solucionar este problema, en los métodos estáticos del script UtilidadesMovimiento, se comprueba antes de variar su posición si el valor del timeScale está por encima de 0.

Apendice B: Pantallas.

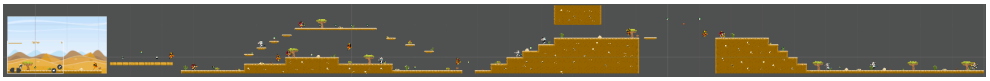
A continuación se muestran capturas de los niveles completos.

Desierto:

1.1



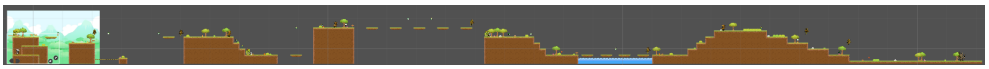
1.2



Bosque:

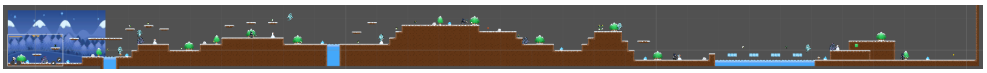
2.1

2.2

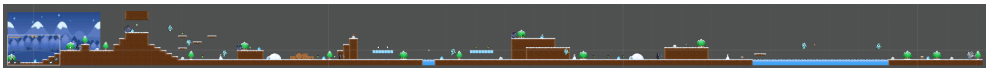


Nieve:

3.1

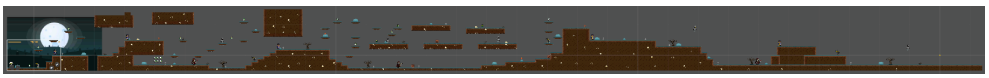


3.2



Cementerio:

4.1



4.2