**CFGM DESARROLLO DE APLICACIONES MULTIPLATAFORMA**

**Proyecto Final de Ciclo**

****

**Súper juego de Diego**

****

**Autor: Diego Ripoll Ramirez**

**Tutor: Alvaro Ortega Marmol**

**Fecha de entrega: 30/05/2020**

**Convocatoria:** 2S 2019-2020

Índice de contenidos

[1. INTRODUCCIÓN. 1](#_Toc40858953)

[1.1.- MOTIVACIÓN. 1](#_Toc40858954)

[1.2.- ABSTRACT. 2](#_Toc40858955)

[1.3.- OBJETIVOS PROPUESTOS (GENERALES Y ESPECÍFICOS). 3](#_Toc40858956)

[2. METODOLOGÍA UTILIZADA 3](#_Toc40858957)

[3. TECNOLOGÍAS Y HERRAMIENTAS UTILIZADAS EN EL PROYECTO. 6](#_Toc40858958)

[4. ESTIMACIÓN DE RECURSOS Y PLANIFICACIÓN 8](#_Toc40858959)

[5. DESARROLLO DEL PROYECTO. 10](#_Toc40858960)

[5.1.- ANÁLISIS 10](#_Toc40858961)

[5.2.- DISEÑO 10](#_Toc40858962)

[5.3.- IMPLEMENTACIÓN 11](#_Toc40858963)

[6. DESPLIEGUE Y PRUEBAS 29](#_Toc40858964)

[7. CONCLUSIONES 30](#_Toc40858965)

[7.1.- OBJETIVOS ALCANZADOS 30](#_Toc40858966)

[7.2.- CONCLUSIONES DEL TRABAJO 30](#_Toc40858967)

[7.3.- VÍAS FUTURAS 31](#_Toc40858968)

[8. GLOSARIO 32](#_Toc40858969)

[9. BIBLIOGRAFÍA 34](#_Toc40858970)

[10. ANEXOS 36](#_Toc40858971)

[10.1.- MANUAL DE INSTALACIÓN. 36](#_Toc40858972)

[10.2.- MANUAL DE USUARIO 36](#_Toc40858973)

# 1. INTRODUCCIÓN.

La motivación de este proyecto nace de la intención de desarrollar un juego en 2 dimensiones similar al juego “Mario Bross” pero con 3 ventajas principales:

* Poder ir guardando el progreso; resulta frustrante para el usuario tener que comenzar cada vez por el principio y ver perdido el progreso acumulado cada vez que reinicia el juego.
* Poder mejorar las capacidades del personaje; una forma de hacer el juego más inmersivo es permitir al usuario ir adaptando al personaje a su gusto. De este modo, según va avanzando su personaje se irá adaptando más a su modo de juego haciendo que una misma partida sea diferente según quien la juegue.
* Duración; puesto que el jugador podrá ir guardando tanto el progreso de su personaje, como su progreso dentro del juego esperaba poder crear más niveles de modo que su duración fuera aceptable y no terminase en unas pocas horas.

Los objetivos alcanzados en este sentido han sido crear un juego en el que es muy sencillo ampliar los escenarios con nuevas áreas, enemigos y retos. En cuanto a las capacidades del personaje he incluido algunas habilidades que serán necesarias para ir avanzando ya que como elemento diferenciador he incluido que las monedas ralenticen al personaje y que cada nivel sea más largo que el anterior. Esto va a plantear al jugador la decisión de si continuar recogiendo monedas para mejorar sus habilidades al finalizar el nivel, o si por el contrario, ir menos cargado para asegurarse estar en condiciones de llegar a la meta. En cuanto a la duración, aunque actualmente solo tengo 3 fragmentos de nivel que se van combinando entre ellos, se pueden duplicar los mismos y crear nuevos y completamente diferentes niveles (respetando únicamente los puntos de entrada y salida de nivel) de modo que se crea un juego infinito. Debido a la mejora que va recibiendo el personaje a medida que progresa, he ampliado la longitud de los niveles, así como el coste de las habilidades, para forzar al usuario a recoger más monedas en cada nivel si quiere continuar progresando con su personaje. Esto aumenta la dificultad de los niveles al aplicarse el peso que reduce tanto su velocidad de movimiento como su capacidad de salto.

## 1.1.- MOTIVACIÓN.

La motivación para realizar este proyecto ha sido un curso que adquirí en Udemy hace tiempo y que tenía olvidado, aunque con ganas de continuar. EL curso en cuestión se llama “Nuevo Curso de Desarrollo de Videojuegos con Unity 2018 y C#” de Juan Gabriel Gomila Salas y en él, como indica el título, va mostrando como utilizar el motor gráfico Unity y el lenguaje de programación C# para realizar distintos tipos de juegos. Hace tiempo había terminado la primera parte, en que hacía uso de Assets descargados de la Asset Store con sus scripts en C# ya creados, y tenía pendiente comenzar la creación de un juego 2D desde cero, creando nosotros mismos todo. En este sentido he de reconocer que ha superado mis expectativas ya que al proponer el proyecto no esperaba tener la posibilidad de encadenar fragmentos de nivel para hacer un juego interminable, lo cual me evita tener que ir creando nuevos niveles constantemente para que el juego no se acabe.

El principal motivo de utilizar este motor gráfico es, además de su simplicidad y que es gratuito, su facilidad para exportar el juego a cualquier plataforma. Con esta facilidad, el juego creado para Android es adaptable en unos segundos para otras plataformas, como pueden ser IOS o PC.

Teniendo en cuenta que la industria de los videojuegos es una de las industrias más potentes del mundo y que este proyecto me permitía profundizar más en mis conocimientos de programación, bases de datos y motores gráficos, considero que ha sido un proyecto interesante en que aplicar los principales conocimientos adquiridos durante el curso. Si bien es cierto que la utilización de bases de datos no la habría incluido de no ser un requisito del proyecto, ya que todos los datos que vamos a guardar en sus tablas se podrían guardar en las preferencias de usuario del propio dispositivo al ser un juego de un único jugador.

## 1.2.- ABSTRACT.

The main objective of this project was to combine the knowledge acquired during the course by making an interesting application, and at the same time, to make a funny game with which to entertain anyone. In this sense, the game I have prepared has many good qualities.

It is an endless game, which means that, although I am able to include new levels (in a very easy way) to make it less repetitive, the user will never find himself without anything to do.

It is a game suitable for all ages, the friendly graphics, easy controls and absence of violence or other adult content make that even children might have a good time playing this game.

It is an easy game where your character must move along an endless world collecting coins to buy great improves to him, while avoiding get hit by robots. The mechanics of the game make that carrying too many coins or being injured will affect your moving or jumping speed. I have always considered that being overloaded with gold or nearly dead must anyway affect the capabilities of the character.

As a conclusion I want to say that the development of this game has exceeded my expectations, I have learned how to use one of the 2 main graphics engine as well as many new libraries of the C# programming, combining it with a Java server, and by the same time I have made an interesting game with many options in which I can continue including even more (and of course adapting and debugging it).

## 1.3.- OBJETIVOS PROPUESTOS (GENERALES Y ESPECÍFICOS).

El objetivo general del proyecto era la creación de un juego sencillo habilitado para todas las edades y que permitiese al usuario jugar tanto tiempo como desease sin sentir la frustración de no conseguir avanzar.

En este sentido, he conseguido una interfaz sencilla con pocos botones que permite una curva de aprendizaje alta en cuanto a manejabilidad del juego se refiere. No obstante, considero que es necesario un manual de usuario para comprender las mecánicas del juego, ya que algunos factores como el peso de las monedas o el efecto de la salud en el movimiento del personaje pueden ser difíciles de intuir a simple vista.

En cuanto a la imagen, son escenarios amigables y muy sencillos (a falta de incluir más variedad para que no sea tan repetitivo) en los que además he incluido la opción de reiniciar el nivel, dando la opción de gastar previamente todas las monedas recogidas hasta el momento para quien no sea capaz de terminarlo con las condiciones arrastradas del nivel anterior. Esto, si bien hace el juego extremadamente fácil para quien tiene paciencia, permite su uso para personas con una coordinación más limitada o niños.

# 2. METODOLOGÍA UTILIZADA

La creación de este proyecto está dividida en varias secciones interrelacionadas pero independientes.

He comenzado por la creación de un nivel inicial organizando los distintos elementos en profundidad y asignándoles sus respectivas capas y etiquetas que usarán nuestros scripts para reconocer a cada GameObject incluido en la escena. A pesar de ser un juego en 2D la profundidad es importante para decidir que objetos de ven delante o detrás. Las capas serán utilizadas para dividir los elementos en categorías a la hora de colisionar con ellas. En esta sección simplemente he colocado el suelo, plataformas y decoración donde comenzará el personaje al principio de cada nivel.

A continuación, he creado el propio personaje, para lo cual le he incluido un Sprite renderer, un rigidbody y un Collider. Más adelante le incluiremos una serie de animaciones, un audio source y un script con sus características y controles.

Tras eso creo los controles del juego, esto es, los métodos básicos del script del personaje. En este caso simplemente era andar (hacia adelante y hacia atrás) y saltar por lo que simplemente había que determinar la fuerza de salto y de carrera, así como rotar el Sprite (mas adelante la animación) cuando camine o salte hacia la izquierda. Más adelante afectaremos a estas fuerzas a través de los coleccionables y habilidades.

Seguidamente establezco la lógica del juego en cuanto a sus los posibles estados en que se puede encontrar el juego. Esto lo hago mediante un script que llamo GameManager, que se encargará de gestionar el estado de la partida. Puesto que es un componente muy importante se detallará en profundidad en la sección de implementación dentro del desarrollo del proyecto.

Tras ello he creado un sistema de generación automática del nivel que he llamado LevelGenerator. Este sistema se encarga de ir colocando bloques de nivel a medida que el jugador avanza hacia adelante y eliminando los que va dejando atrás para evitar sobrecargar el sistema. Aunque es posible mover el personaje hacia atrás y por lo tanto ver que parte del nivel ha desaparecido, he considerado que era más importante mantener un nivel de carga bajo en el procesador para garantizar que funcione en cualquier dispositivo que evitar que el jugador pueda ver que no es capaz de volver atrás. Esto además, es posible solucionarlo incluyendo en los niveles cambios de altura insalvables para el personaje o simplemente comentando la parte del código encargada de borrar bloques antiguos.

Posteriormente he creado los coleccionables, aunque el código está diseñado para incluir monedas, bastoncillos de caramelo, pociones de vida y pociones de mana, en los niveles creados hasta ahora he incluido únicamente monedas, con las que comprar habilidades y bastoncillos con los que recuperar la salud perdida en caso de chocar con uno de los robots que pasean por la escena.

He continuado con los obstáculos, que en los niveles actuales son principalmente 2, un enemigo que va recorriendo el mapa en un circuito y empujara al personaje haciéndole daño según avanza y las posibles caídas al vacío desde las plataformas. Los enemigos harán un daño moderado (30 puntos de vida) mientras que la caída de la plataforma supondrá una muerte inevitable.

A continuación, he preparado la GUI, esto es la interfaz con la interaccionará el usuario para controlar el juego. Está compuesto básicamente de 5 interfaces que irán cambiando según el gameManager determine en qué estado del juego estamos. Son el menú principal, la pantalla de juego, la interfaz del nivel completado, el game over y la pantalla de adquisición de habilidades. Hablaremos más de cada una de ellas más adelante.

Tras eso he implementado las animaciones tanto en el personaje como en las plataformas que nos servirán durante los niveles para alcanzar las posiciones elevadas y el robot enemigo que se pasea por las distintas zonas del juego. La del jugador y el enemigo las he creado a través de un Tileset dividiéndolo en distintos Sprites y colocándolos en orden en la ventana de animation, y las plataformas las he creado directamente modificando su posición en esa misma ventana de Unity.

Seguidamente he incluido un registro de los datos del jugador y los he incluido de forma visual en la GUI. Estos datos son el proceso del nivel en función de la distancia recorrida, el número de monedas y el número máximo de monedas recogidas (recordar que la dificultad aumenta cuantas más se tenga). la salud y mana del personaje (barras roja y azul de la esquina superior derecha de la pantalla). La salud deberá estar por encima de cero para poder continuar con el nivel y el maná se utilizará para ejecutar un espectacular salto que permitirá subir a posiciones inaccesibles de otro modo, especialmente cuando se va sobrecargado.

Posteriormente he implementado las habilidades tanto en su ventana del GUI como su efecto en las distintas habilidades del personaje. Estos efectos proporcionan importantes ventajas, aunque dependerá del usuario en cuáles de ellos invertir sus monedas. Estas pueden aumentar la salud, mana, velocidad, capacidad de salto, capacidad de carga, etc.

Por ultimo he creado la base de datos con sus tablas y he conectado el juego con ella para que cargue los datos cuando pase por el menú principal y los guarde cuando finalicemos un nivel. Esta conexión la realizara mediante un servidor para evitar incluir en mi aplicativo las claves de la base de datos, ya que supondría un fallo importante de seguridad. Dicha base de datos la he creado en una Raspberry PI 4.

# 3. TECNOLOGÍAS Y HERRAMIENTAS UTILIZADAS EN EL PROYECTO.

En este proyecto he utilizado principalmente las siguientes tecnologías:

* **Unity** es un motor de videojuego multiplataforma creado por Unity Technologies. El ser multiplataforma le permite crear un juego y adaptarlo a cualquier plataforma de manera muy sencilla. Está disponible para Microsoft Windows, Mac OS y Linux. Ya no soporta el desarrollo de contenido para navegador a través de su plugin web ya que ahora utiliza WebGL. Tiene una versión gratuita, Unity Personal que se puede usar siempre que se obtengan ingresos inferiores a 100.000 USD en los últimos 12 meses.

En el desarrollo de este proyecto he utilizado este motor gráfico ya que integra los cálculos para aplicar físicas y colisiones que de otra forma requeriría mucho tiempo y conocimientos avanzados de matemáticas y física utilizar.

* **Visual Studio** es un entorno de desarrollo integrado para Windows, Linux y macOS. Este IDE es compatible con múltiples lenguajes de programación, tales como C++, C#, F#, Java, Python, Visual Basic .NET, Ruby y PHP, ASP.NET MVC, Django, etc.

Permite crear sitios y aplicaciones web, así como servicios web en entornos compatibles con la plataforma .NET. De este modo, permite crear aplicaciones que se comuniquen entre estaciones de trabajo, páginas web, dispositivos móviles, dispositivos embebidos y videoconsolas, etc.

En el proyecto he utilizado este IDE para crear en el lenguaje de programación C# los scripts que contendrán la lógica del juego y que serán asignados a los diferentes GameObjects.

* **MySQL** es un sistema de gestión de bases de datos relacional, desarrollado en su mayor parte en ANSI C y C++.4, por Oracle Corporation bajo licencia dual (Licencia pública general/Licencia comercial) considerada como la base de datos de código abierto más popular del mundo, y una de las más populares en general junto a Oracle y Microsoft SQL Server.

Este esquema de doble licenciamiento se debe a que una empresa privada posee el copyright de la mayor parte del código. La base de datos se distribuye en varias versiones, una Community, distribuida bajo la Licencia pública general de GNU, y varias versiones Enterprise, para aquellas empresas que quieran incorporarlo en productos privativos. Las versiones Enterprise incluyen servicios adicionales tales como asistencia técnica o herramientas de monitorización.

Este es el sistema gestor de base de datos que he utilizado para guardar los datos del usuario según progresa de nivel.

* **Audacity** es un software multiplataforma de edición de audio. Es un programa gratuito de código abierto y multiplataforma que permite recortar los audios de forma sencilla. Este ha sido importante para manipular la duración de los audios descargados para el juego, ya que normalmente los sonidos vienen rodeados por un tiempo de silencio en el clip.

# 4. ESTIMACIÓN DE RECURSOS Y PLANIFICACIÓN

Para el desarrollo de este proyecto he intentado agrupar las tareas en bloques más o menos independientes. Asignando una semana para cada tarea de modo que antes del final del plazo pudiera terminar todo el proyecto. El plan de tareas repartidas por semanas ha sido el siguiente:

| Tareas | Planificación y diseño | Desarrollo | Pruebas | Documentación |
| --- | --- | --- | --- | --- |
| Semana 1 | Planteamiento y definición del proyecto |  |  |  |
| Semana 2 | Definición de los objetivos específicos y características. | Descarga de Sprites y tilemaps | Prueba de fondo transparente |  |
| Semana 3 |  | Creación del nivel inicial |  |  |
| Semana 4 |  | Diseño del personaje y sus controles | Prueba de audios y manejabilidad con teclas (doble salto). |  |
| Semana 5 |  | Diseño de la lógica del juego: game manager | Prueba del cambio de estado y bloqueos de funciones fuera del estado de juego |  |
| Semana 6 |  | Diseño del level generator | Prueba de posición de los nuevos bloques y accesibilidad a nuevos bloques en juego |  |
| Semana 7 |  | Diseño de los coleccionables y enemigos | Prueba de desaparición tras recolección, cálculo del daño |  |
| Semana 8 |  | Diseño de animaciones de plataformas, enemigos y jugador | Prueba de posibilidades del personaje |  |
| Semana 9 |  | Actualización de los datos del jugador en la interfaz grafica | Prueba progreso hacia atrás. |  |
| Semana 10 |  | Diseño de habilidades e integración con las capacidades del personaje | Prueba de capacidades y efecto de los incrementos |  |
| Semana 11 |  | Creación del modelo cliente-servidor-base de datos |  |  |
| Semana 12 |  |  |  | Creación de la memoria |
| Semana 13 |  |  |  | Preparación de la presentación |

Aunque este plan se ha conseguido seguir para el desarrollo, debo mencionar que la tarea de documentación estaba diseñada originalmente repartida en todas las semanas, de modo que cada semana documentara la parte relativa al trabajo realizado, sin embargo, finalmente se ha hecho en las últimas 2 semanas. Las pruebas sí se han ido realizando a medida que se hacía la parte de desarrollo correspondiente para comprobar su correcto funcionamiento.

# 5. DESARROLLO DEL PROYECTO.

## 5.1.- ANÁLISIS

El principal objetivo de este proyecto es asentar los conocimientos adquiridos durante el curso, pero también es importante cumplir todos los requisitos establecidos. Por ello los requisitos funcionales son entre otros los establecidos por el centro para el mismo. Esto es:

* Una base de datos con al menos 3 tablas.
* Una estructura cliente servidor para conectar con la base de datos

El resto de requisitos establecidos por el centro son tan básicos que era imposible desarrollar el proyecto sin ellos.

Además, hay otros requisitos generales del proyecto, están los específicos de mi juego.

En primer lugar, quería que fuera un juego 2D diseñado con el motor de videojuegos Unity, por ser unos de los más importantes hoy en día.

Quería que fuera fácilmente ampliable para evitar que tuviera un final cercano, o según lo he hecho finalmente, que no tuviera fin. Al tener la posibilidad de guardar el juego quería que la duración del juego no fuera un problema.

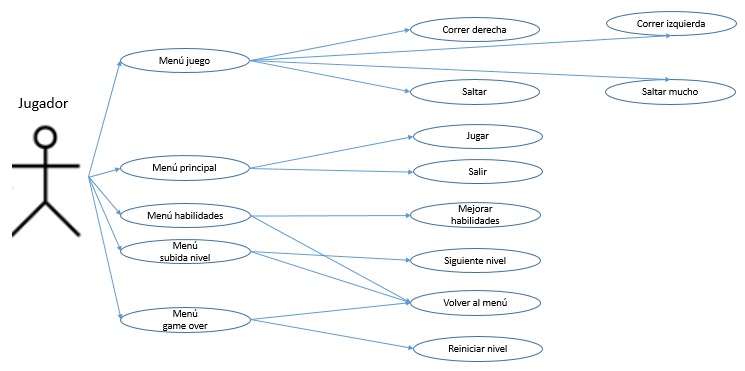
Que estuviera preparado para incorporarle más elementos con distintas funcionalidades.

Que fuera posible personalizar al personaje, de modo que cada jugador tendrá uno diferente.

## 5.2.- DISEÑO

En cuanto al diseño, como en cualquier juego, las posibilidades del personaje dependerán del menú en que se encuentre. Las operaciones de guardado y cargado se realizan automáticamente después de comprar mejoras para el personaje (guardado) y antes del comienzo de cada nivel (cargado) por lo que no requieren interacción con el usuario. Así mismo, estas tareas utilizan el identificador del teléfono, por lo que no será necesario introducir credenciales para jugar.

A grandes rasgos sus posibilidades serán las siguientes:



Como se puede observar en el diagrama de casos de uso, el jugador tendrá entre 2 y 4 opciones teniendo todos los controles mediante botones táctiles en pantalla. En todos los menús, salvo el de juego estos botones estarán en el centro de la pantalla y coloreados de; verde el botón de juego, azul el de compra de habilidades y rojos los que llevan a la salida. La selección de estos colores es para incentivar al jugador a continuar jugando antes de salir. La opción de salir del menú principal no es necesaria en el caso de un juego para dispositivos móviles, o incluso prohibida en el caso de IOS, ya que estos tienen un botón expreso para salir de cualquier aplicación por lo que usuario hará uso principalmente de ella, pero personalmente prefiero incluir esa opción en la interfaz gráfica por comodidad para el jugador.

Además de los botones incluidos en el diagrama, con posterioridad he necesitado incluir un botón de rendición en el menú de juego para el caso de que el jugador haya recogido demasiadas monedas antes de completar el nivel, ya que es sencillo llegar a un punto en que el personaje se encuentre bloqueado. Este botón, también rojo, permitirá gastar las monedas acumuladas antes de reiniciar el nivel.

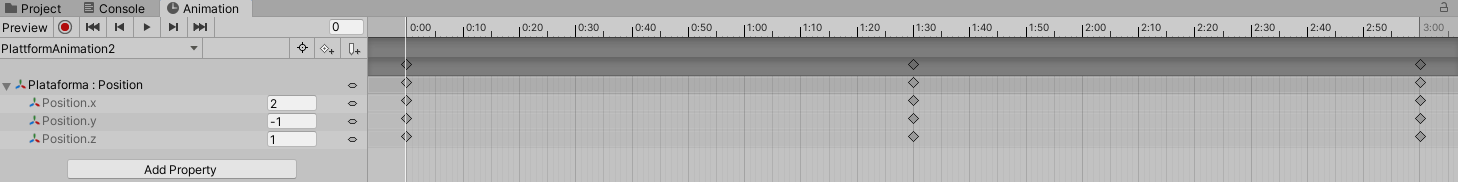
## 5.3.- IMPLEMENTACIÓN

El proyecto se ha realizado con el motor de videojuegos Unity. En su realización he creado un nuevo proyecto llamado Ilerna\_DiegoRipollRamirez\_DAM donde he incluido una única escena. Al ir generándose y destruyéndose el escenario según el personaje se mueve no necesitábamos incorporar más, ya que en ningún momento llegara a ser pesado para el procesador.

Por la parte de Unity he dividido el proyecto en diferentes carpetas para tener todos los Assets fácilmente localizables para incluirlos en los distintos GameObjects que he necesitado.

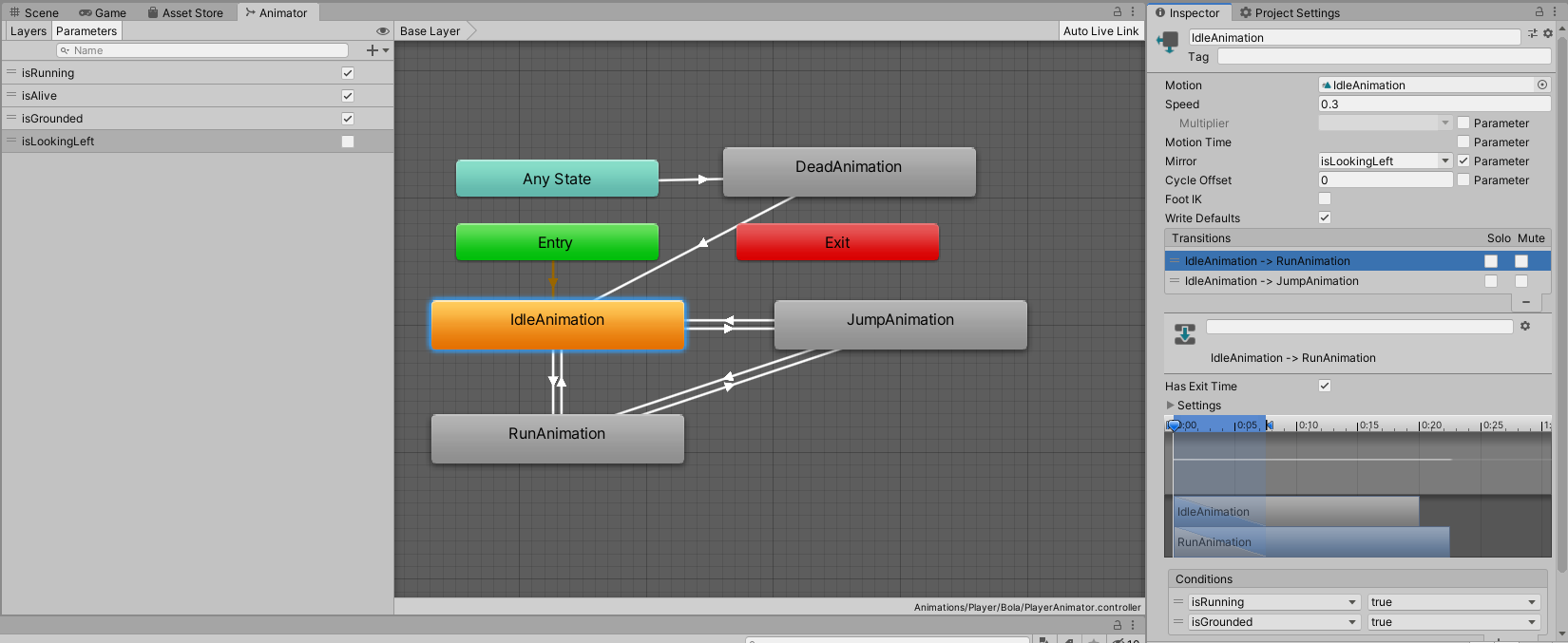
Estas carpetas son:

**Animations**; en esta carpeta he incluido las animaciones tanto del personaje como de los enemigos y las diferentes plataformas móviles que forman los LevelBlocks. Estas animaciones o bien se importan directamente a modo de conjunto de Sprites que se arrastran directamente al GameObject, donde luego en la pestaña de Animation se ajustan los tiempos y se modifica lo necesario, o bien, directamente en esa pestaña se incluyen las características a modificar en cada intervalo de tiempo.



La imagen anterior representa la animación creada para mover una de las plataformas. En esta animación únicamente modificamos la posición de la plataforma, de modo que, en el inicio, se encuentre en la altura -1, tras un segundo y medio haya subido hasta la altura 3 y tras otro segundo y medio haya vuelto a bajar a la posición -1. Como se puede ver en la imagen, esta ventana nos deja modificar muchas propiedades del GameObject y determinar que valores tendrán cada 0,5 segundos. El propio motor gráfico se encargará de alterar las propiedades de forma lineal entre ambos intervalos de tiempo, pero este nivel de detalle nos permite, modificar la velocidad de esos cambios entre distintos intervalos, así podremos, por ejemplo, dar sensación de aceleración o freno en los movimientos.

En cuanto al personaje, dado que deberá modificar su animación en función de muchos factores decididos en cada momento por el jugador hay que fijar una serie de variables que determinen cual mostrar en cada momento.



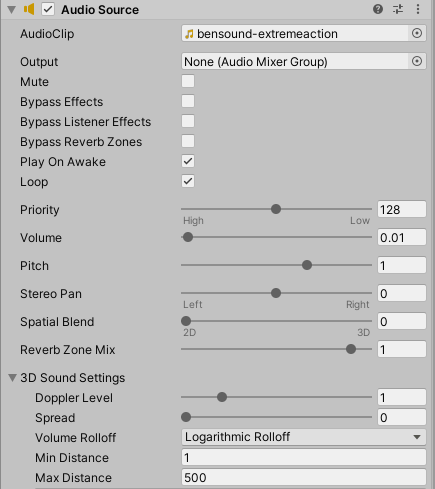
En la imagen anterior se ve la pestaña Animator del personaje. En la parte central de la imagen, se pueden ver las relaciones entre las distintas animaciones del personaje. En el lado izquierdo de la imagen se puede observar las variables booleanas que serán las que mediante sus variaciones (alteraciones que se hacen en el script del personaje) determinarán el salto de una animación a otra. Así, en el lado derecho de la imagen hay una sección llamada “Transitions”, que controla las variables necesarias para hacer la transición de una animación a otra. En este caso, se pueden ver las 2 transiciones posibles desde la animación IdleAnimation (animación para cuando el personaje está quieto), que son hacia la “RunAnimation” y hacia la “JumpAnimation”. Además, en la parte inferior derecha, se puede ver los cambios que deben producirse para pasar de la IdleAnimation a la RunAnimation. En este ejemplo será que las variables isRunning e IsGrounded sean verdaderas. Esto significa que si el personaje no está corriendo (isRunning) o si está saltando (IsGrounded) no entrará en esta animación. Del mismo modo las animaciones de destino tienen transiciones para salir de ellas cuando estas variables dejen de cumplirse.

El resumen de los estados posibles de la animación, así como sus condiciones de entrada y salida son:

* + IdleAnimation: animación para el personaje quieto. Se aplica cuando las variables isAlive y isGrounded son true y la variable isRunning es falsa.
  + RunAnimation: se utiliza cuando el personaje está corriendo por el suelo. Se aplica cuando las tres variables son verdaderas, está vivo, está corriendo y está en el suelo.
  + JumpAnimation: variable para cuando el personaje está saltando. Se aplica cuando la variable isAlive es verdadera y la variable isGrounded es falsa. En este caso nos es indiferente si la variable isRunning es verdadera o falsa ya que este quieto o no, si está saltando debe usar esta animación.
  + DeadAnimation: esta animación es para cuando el personaje muera, por lo tanto, se aplicará únicamente cuando la variable isAlive sea falsa.

**Audios**; en esta carpeta he guardado todos los audios que se utilizan durante el juego, que son básicamente la música de fondo que suena desde que se inicia la aplicación hasta que se cierra, el sonido de cuando el personaje salta y el sonido que hace al recoger una moneda. Estos audios los podemos tomar de diferentes páginas web como <https://incompetech.com/>, <https://freesound.org/> o <https://www.tunefruit.com/>. Todas ellas tienen músicas sin derechos de autor que utilizar en nuestro proyecto, sin embargo, es muy probable que antes de utilizarlas haya que cortarlas para adaptar su duración a nuestras necesidades. Esto lo he hecho con la aplicación Audacity. Esta es una aplicación gratuita y muy fácil de usar que se puede descargar en el siguiente enlace <https://www.audacityteam.org/>. Para editar sonidos con ella, simplemente hay que arrastrar el archivo encima de la aplicación abierta y cortar la parte del audio donde se ve que no hay sonido. En este tipo de audios es normal que el primer medio segundo aproximadamente sea silencio, pero esto generaría un desfase en nuestro juego entre el momento en que se produce el evento y en que suene su audio.

Para que nuestros GameObjects puedan usar estos sonidos deberán tener un componente AudioSource.

Este ejemplo corresponde al audioSource del GameObject BackgroundMusic. En él, podemos ver que el clip de audio que le he incluido es bensound-extremeaction, que se ejecutará tan pronto como se despierte el GameObject y que se reproducirá en bucle.

En caso de querer incluir más sonidos a un GameObject tendremos que dejar el campo del AudioClip vacío e ir indicándole a través del script del propio objeto que música debe sonar en cada momento.

Esto lo haremos creando un objeto AudioClip público al que en el editor de Unity le asignaremos una pista de audio, también crearemos un objero Audiosource en el script y le indicaremos que su clip es el clip creado anteriormente, por ultimo le daremos play.

El componente AudioSource, ofrece muchas otras opciones para modificar el sonido, aunque en este proyecto principalmente he adaptado el volumen para que sea el 0.01 de su máximo, y que sea el propio jugador quien a través de su dispositivo lo suba en caso de desearlo.

**Fonts**; en esta carpeta incluimos los distintos tipos de fuente que queremos usar en nuestra interfaz. Para el proyecto he descargado de la página web <https://www.dafont.com/es/> la letra MangoDrink por adaptarse a la idea que tenía para los textos de la interfaz. Esta letra se incluirá en los componentes Text de la interfaz donde pondremos el texto a mostrar por pantalla. Esta letra además puede adaptarse añadiéndole otros componentes como el Outline que le incluirá un reborde del color y grosor que queramos, o el Shadow que ha lo propio con la sombra que genere el texto.

**Prefabs**: como hemos visto hasta ahora, Unity permite añadir un montón de componentes a los GameObjects para hacer que se adapten a las necesidades que tengamos de ellos, sin embargo, una vez que está listo es posible que queramos utilizarlos en más de una ocasión sin tener que repetir todo el trabajo para cada copia. En esta carpeta he incluido los Prefabs. Estos son como modelos de los que podremos hacer tantas copias como queramos para posteriormente incluirlas en nuestro juego. De este modo cada vez que queramos crear un objeto habitual, como monedas o enemigos no tendremos que generarlo de nuevo. Otra gran ventaja de este sistema que nos ofrece Unity es la posibilidad de modificar el Prefab padre y aplicarle los cambios automáticamente a todos sus hijos. Así, si por ejemplo le quiero cambiar al audio a todos ellos, no tendría más que cambiárselo al padre y aplicar cambios en los hijos.

En nuestro proyecto, los prefabs más importantes son:

* LevelBlocks; estos prefabs tienen ya montados todos los componentes del bloque de forma que se puedan ensamblar automáticamente cuando sean llamados por el LevelGenerator. Además del suelo y demás componentes tienen los siguientes elementos.
  + Collectables; haciendo uso del prefab del colectable he incluido unas cuantas monedas y bastoncillos en todos ellos.
  + Enemigos; igual que con los colectables.
  + StartPoint, ExitPoint, ExitZone, KillZone y Freno caída son otros elementos importantes que, dada su relación con el script de generación de bloques, serán descritos en esa sección.
* Collectable; estos prefabs tienen ya incluidos su imagen, animación, Collider, AudioSource y script. Estos elementos servirán para darle su imagen y permitir que el personaje los coleccione emitiendo además un sonido cuando esto ocurra.
* Enemy; en este caso hemos incluido en un GameObject hijo del principal su Sprite, que contendrá un trigger Collider y su script que hará que dé la vuelta cuando choque contra un objetivo. Quedando por tanto en el principal su rigidbody, el Collider que permitirá al personaje eliminarlo y la otra parte del script que sirve para dar la vuelta. Esta separación es importante porque queremos que el Collider de cada GameObject haga cosas diferentes en función de la etiqueta que tenga cada uno.

**Sprites**: en esta carpeta he ido guardando todos los Sprites que he ido utilizando en la creación del juego. Un Sprite es una imagen en 2D. Esta puede representar un objeto o una parte de él. Por ejemplo, el suelo puede estar formado por un conjunto de Sprites colocados consecutivamente, o el personaje estar formado por otro grupo de ellos, de modo que cada brazo, pierna, ojo y torso sean distintos Sprites que colocados juntos y sobre los que aplicando una animación que los mueva a todos de forma sincronizada creen nuestro personaje.

Es habitual que al importar los Sprites vengan varios en una única imagen agrupados por temáticas, este conjunto de imágenes se llama Tilemap.

En esta imagen podemos ver un Tilemap de los distintos elementos que he ido utilizando para crear los LevelBlocks. Una vez importada la imagen, debemos recortarla haciendo uso del Sprite editor que nos aparecerá en el inspector de elementos cuando seleccionemos un Sprite cualquiera. Es habitual que estos Tilemaps tengan sus elementos separados por una distancia regular, de modo que en el editor simplemente haya que seleccionar la distancia de corte de los elementos, tanto vertical como horizontalmente, y Unity se encargue de generar tantos Sprites como trozos hayamos creado. Con ellos podremos ir componiendo nuestro nivel como si de un puzle se tratase. Por lo general estos Sprites se pueden descargar de la Asset Store, lo que en muchas ocasiones incluirá también las animaciones en caso de tratarse de objetos móviles (personajes, enemigos, etc.). En caso de descargarlos desde internet deberemos asegurarnos de que no tengan fondo y, si vamos a publicarlo, derechos de autor.

**Scripts**: esta es la carpeta más importante ya que contendrá toda la lógica del juego. En ella guardaremos los scripts que se asignaran a cada GameObject indicándole lo que debe hacer. Las clases en Unity heredan de la clase MonoBehaviour cuyos métodos principales son:

* Awake(); este método es el que se ejecuta nada mas arrancar la aplicación, antes incluso del Start(). He utilizado este método para asignar los componentes como los RigidBodies o los singleton.
* Start(); este método se ejecuta antes del primer update. En él asignamos los valores iniciales a las variables, pues se utiliza para establecer la situación inicial de la partida.
* Update(); este método se ejecuta una vez por frame, va comprobando constantemente la activación de eventos y las variaciones en los GameObjects. Es el método encargado de hacer que todo vaya modificándose cuando se produzcan cambios, bien por acciones del jugador (saltar, correr, morir) o por la propia física del juego (caerse, chocar)
* FixedUpdate(); este método es igual que el método Update() con la sencilla diferencia que las actualizaciones se producen a tu tiempo constante, en vez de verse afectados por ralentizaciones en cuanto a los frames. Esto es importante por ejemplo cuando necesitamos que se produzca un movimiento contante sin ver saltos que podrían frustrar mucho al jugador al no entender porque al hacer lo mismo ocurren cosas diferentes por motivos que no puede controlar.

Dada su importancia voy a detallar cada script, así como sus variables y métodos.

* GameManager

Este script será el encargado de gestionar la partida. El juego tendrá 4 estados posibles que están agrupados dentro de un enumerator. Estos estados son:

1. Menu; el menú principal del juego al que se accede nada más arrancar la aplicación o cada vez que se termina un nivel, con éxito o no.
2. InGame; será el estado de juego en que el jugador podrá controlar a su personaje y jugar.
3. GameOver; es el estado de juego al que se accede cuando el personaje muere.
4. LevelEnd; es el estado de juego al que se accede cuando el personaje completa un nivel con éxito
5. SkillsMenu; es el estado que abre la pantalla de adquisición de habilidades para mejorar al personaje.

Además, es importante que el juego esté gestionado por un único GameManager, puesto que de haber varios podría encontrarse con instrucciones contradictorias y no sabría a cuál hacer caso. Por ello creamos un singleton, esto es, una instancia estática del juego que será la que se use para dar instrucciones a la partida. Al ser estática nos aseguramos que solo haya una y no nos encontremos con contradicciones.

Mediante el método SetGameState fijamos el estado de juego en el corresponda de los 4 listados anteriormente y en función del que nos encontremos activaremos el Canvas correspondiente.

Además de este método, están los siguientes:

* Awake(); en este método fijamos los elementos que queremos que se carguen primero. En este caso, asignar el singleton a la clase, el identificador del dispositivo para cargar los datos cuando se juegue y la orientación de la pantalla.
* StartGame(); es el método que iniciara la partida. Entre sus funciones están eliminar los bloques creados durante un juego anterior y recolocar el bloque original volviendo el juego a la posición inicial, indicarle al script del personaje que debe comenzar sus acciones iniciales (colocar sus variables en los valores por defecto, su posición, etc.), y devolver la cámara a la situación inicial. Como ésta sigue al jugador de forma fluida, si no la devolvemos automáticamente, regresaría avanzando por todo el mapa hacia atrás en vez de reaparecer en la posición correcta.
* GameOver(); este método será llamado por el script del jugador cuando el personaje haya modificado su estado de isAlive a False bien por haber perdido toda su vida o bien por haberse caído de las plataformas. En este método cambiamos el estado de juego a GameOver, lo que mediante la función SetGameState mencionada con anterioridad activará el Canvas correspondiente.
* BackToMenu(); este método será llamado por el jugador al pulsar los botones Menu en los diferentes Canvas. Cambiará el estado de juego a menú, colocará la vida del personaje en uno (por si hubiera muerto como consecuencia de daño sufrido) y activará el Canvas de menú. Ademas, al estar este método vinculado a un botón del UI, he incluido aquí la grabación de los datos tras terminar la compra de habilidades. De este modo el jugador guardará automáticamente sus compras al terminar un nivel y estarán disponibles la siguiente vez que juegue.
* SkillsMenu(); este método será llamado por el jugador al pulsar los botones Skills en los diferentes Canvas. Cambiará el estado de juego a skillsMenu y permitirá acceder a la ventana de compra de habilidades.
* ExitGame(); este método permitirá salir de la aplicación al pulsar el botón exit en el menú principal. Este botón en Android está permitido, aunque tenga su botón central para salir de sus aplicaciones, pero Apple no permite su inclusión para sus dispositivos, por lo que habría que eliminarlo si se quiere subir a la Apple Store.
* EndLevel(); este método lo activara el Player\_script cuando el personaje haya completado con éxito el 100% del nivel. Colocará el estado de juego en levelEnd y permitirá acceder el menú de subida de nivel.

En este script, ya que cambiara el estado de juego, tendremos que asignarle los diferentes Canvas que hemos creado en el juego para que vaya activando y desactivando según el estado en que nos encontremos.

* CoinScript

A pesar de su nombre este script gestiona las acciones a realizar por todos los coleccionables. El script está preparado para colocar 4 tipos de coleccionables, aunque por el momento solo he incluido 2 (el juego está preparado para facilitar la incorporación futura de mejoras). Estos son pociones de vida con la apariencia de bastoncillos de caramelo y monedas. Los primeros servirán para restituir la vida al personaje en caso de que la haya perdido en sus encuentros con enemigos y las monedas servirán para comprar habilidades que mejoren las capacidades del personaje.

Para poder asignar el valor de cada coleccionable desde el propio motor gráfico, he incluido una variable publica value, que se podrá modificar en el script asociado a cada GameObject. También he creado una variable type para poder diferenciar los coleccionables entre los 4 tipos que he preparado en el enumerado. Esta variable se puede asignar, igual que la anterior, desde el propio GameObject. Además, cada coleccionable tendrá su propia pista de audio que sonará cuando sea recogido por el personaje.

Para que un personaje pueda recoger coleccionables en Unity se utiliza el método OnTriggerEnter, o en este caso al ser un juego 2D, OnTriggerEnter2D. Este método hace una acción en caso de que el Collider asociado al objeto que contiene este metodo colisione con otro Collider. Como no quería que los enemigos las recogiesen, establecí que solo se activase en caso de que el otro objeto con el que colisionase tuviese la etiqueta “Player” que obviamente coloqué únicamente al personaje del jugador. En caso de que el Collider del jugador colisionase con un coleccionable que tuviese este método y con la variable type en moneda, las monedas recogidas en el nivel sumarán 10, y en caso de que el coleccionable sea del tipo pocionVida llamará al método de ganar vida del Player\_Script.

Por ultimo este método destruirá el game Object en cuanto termine el clip de audio.

* Enemy y TriggerMovement

Estos 2 script que tendrán los enemigos gestionan el movimiento de los enemigos, y hacen que vayan avanzando en unan dirección hasta que choquen con algo (que no sea un coleccionable o el jugador), en ese momento invertirán la marcha. Comenzando por el script de TriggerMovement está compuesto por un único método OnTriggerEnter2D que mediante la variable booleana movingForward alterará la variable turnAround. Esto es, desde el Game Object fijaremos el valors inicial de movingForward en true o false según hacia donde este mirando nuestro Sprite y en base a eso haremos que por defecto se mueva hacia adelante o hacia atrás. Asumamos que se mueve hacia adelante y por lo tanto movingForward es true, el enemigo seguiría moviéndose en esa dirección hasta que colicionase con algo, en cuyo caso cambiaría el turnAround a true (será utilizado en el script Enemy) y volverá la variable movingForward a false. Por la parte del script Enemy, gestionaremos el movimiento del enemigo fijando la velocidad en positiva o negativa en función del valor de turnAround fijada en el otro script. Por ello este script tendrá un rigidbody que permitirá que los enemigos se vean afectados por fuerzas. Mientras turnAround sea falso el enemigo continuara avanzando hacia adelante. En el momento en que el Collider del enemigo colisionase, se activaría el método OnTriggerEnter2D del script TriggerMovement y colocaría la variable turnAround en true, con esto la velocidad de movimiento pasaría a ser negativo y la rotación del eje y del enemigo pasaría a ser 180, girando el Sprite y su animación en la dirección opuesta. Al volver a chocar, cambiaria de nuevo el valor de turnAround volviendo a la dirección inicial, y repitiendo este bucle hasta que el objeto sea destruido por el personaje.

* LevelGenerator y LevelBlock

Estos serán los métodos encargados de ir generando el escenario y de ir destruyéndolo según avance el personaje. Para comprenderlos hay que entender primero como están fabricados los LevelBlocks. Estos están compuestos por 5 elementos principales:

* Zona de juego: es el área donde el el personaje se moverá a los largo de este LevelBlock. Dentro de este área se incluirán todos los elementos del nivel, monedas, enemigos, suelo, plataformas, etc. Estara delimitado en ambos lados por el StartPoint y el EndPoint.
* StartPoint y EndPoint: son los puntos que marcaran el inicio y final de nivel y deben estar a la misma altura. En ellos es importante que el LevelBlock no termine por debajo del endPoint , o en caso de hacerlo hay que darle una opción para elevar la posición del personaje (como ocurre en mi LevelBlock 3 que termina con un ascensor). Estos puntos son fundamentales porque serán los puntos de union de unos LevelBlocks con otros. Si un nivel terminase por debajo del anterior y otro empezase muy alto se podría dar el caso de que al unirse ambos levelBlocks hubiera un camino imposible se continuar por la diferencia de altura.
* KillZone: es una zona inferior al LevelBlock a partir de la cual no debe continuar la partida. Si el personaje llega hasta ella, significa que ha salido de la zona de juego y debe reiniciar el nivel. Esta zona utiliza el script KillTrigger que explicaremos en el siguiente punto.
* FrenoCaida: Unity no tiene registrada una velocidad de caída terminal, esto implica que si un personaje se cae de una plataforma podría continuar cayendo y acelerando indefinidamente. Esto implica que si le dejas caer durante demasiado tiempo, el personaje ira tan rápido que cuando quieras reiniciar la partida este atraviese el suelo porque la posición del mismo pasaría de estar encima del suelo en un frame y debajo del mismo en el siguiente, sin llegar a chocar con el en ningún momento. Para evitar esto, he colocado el Freno Caida, un GameObject con un Collider físico (no simplemente trigger, como los que se utilizan para activar los métodos OnTriggerEnter2D) por dejajo del KillZone este Collider frenará la caída del personaje antes de que gane suficiente velocidad para atravesar nuestros Colliders de suelo.
* ExitZone: este será el punto que señalará al script LevelGenerator que debe borrar el modulo anterior y colocar el siguiente.

Una vez vistos los componentes del LevelBlock vamos a ver cómo funcionan ambos scripts:

El script LevelBlock simplemente asigna a dos variables de tipo transform el StartPoint y ExitPoint. Al asignarle este script al LevelBlock, podemos arrastrar en el inspector de Unity ambos puntos hasta estas variables para que nuestro script las relacione.

En cuanto al segundo, LevelGenerator, vamos a utilizar también un singleton de sí mismo, de modo que nunca haya dos generadores de bloques funcionando simultáneamente. Así, cuando desde el GameManager se llame a este singleton no se podrán duplicar. Para este script vamos a necesitar, además de su singleton, una lista con todos los bloques elegibles, una lista con todos los bloques en escena y un punto de origen por donde comenzar a colocar bloques.

Como de costumbre vincularemos el singleton sharedInstance a la propia clase en el método Awake().

Despues comenzaremos creando los 2 primeros bloques en el Start. Esto lo haremos llamando al método addInitialBlocks() que hará 2 llamadas al método addLevelBlock. Este último es el bloque que se encargara de ir colocando los bloques cuando el jugador pase por las leaveZone, y el método removeOldestLevelBlock ira eliminando los bloques que vayan quedando atrás. Por ultimo tenemos un último método que elimina todos los LevelBlocks cuando terminamos un nivel.

Comencemos analizando el método addLevelBlock. Cada vez que se llama a este método comienza generando un numero entero aleatorio para elegir al azar un LevelBlock de nuestra lista de allLevelBlocks. Después cogerá el LevelBlock que se encuentre en esa posición (salvo que sea el primero que va a colocar, en cuyo caso cogerá uno que he determinado yo para evitar que pueda ser muy caótico) y lo colocará en la posición spawnPosition. Esta spawnPosition será, en el caso del primer bloque, la que hayamos definido mediante la vinculación de un GameObject vacío creado en Unity con una variable publica que hemos llamado en este script levelStartPoint, y en el caso de que sea un bloque posterior, la que coincida con el exitPoint del bloque anterior. Todos estos levelBlocks los crearemos como hijos del LevelGenerator, por lo que la posición de los sucesivos LevelBlocks necesitarán una corrección en su posición que será simplemente restarle a los ejes x e y de spawnPosition los valores de esos mismos ejes pero del starPoint del propio LevelBlock.

Por ultimo añade el LevelBlock incorporado a la escena a la lista de currentLevelBlocks para que pueda ser eliminado por los métodos removeOldestlevelBlock() y removeAllBlocks.

El método removeOldetsLevelBlock selecciona el elemento 0 de la lista de currentLevelBlocks y destruye el GameObject, y el método removeAllBlocks simplemente va ejecutando el método anterior mientras queden elementos en la lista.

Para incorporar nuevos levelBlocks simplemente habría que copiar uno anterior para tener ya todos los elementos imprescindibles, modificarlo hasta que nos guste e incluirlo en la lista de allLevelBlocks desde el inspector de Unity.

* LeaveZone

Este es el último método relacionado con la generación de LevelBlocks. Su función es activar los métodos addLevelBlock() y removeOldestLevelBlock() explicados anteriormente cuando el collider del jugador choque con el Collider del GameObject exitZone de cada LevelBlock, esto lo hace con el método OnTriggerEnter2D. Por último, he incluido un temporizador para que si el jugador atravieso el mismo Collider varias veces seguidas esta renovación de bloques más que cada 3 segundos.

* KillTrigger

Este script se le asigna al GameObject killZone de cada LevelBlock. Este script tiene un único método onTriggerEnter2D que hace que al chocar el Collider del jugador con el Collider de este objeto, el jugador ejecute el método kill() del Player\_script.

* Player\_script; este es el script que controlara lo que le ocurra al personaje, asi como lo que este puede hacer. Dada su complejidad explicarè los métodos de uno en uno indicando las relaciones con los otros scripts que los desencadenan.
  + Awake; como de costumbre en este método asignamos e inicializamos las variables mas importantes.
  + StartGame; aquí iniciamos, o reiniciamos, el personaje a sus valores iniciales para el comienzo de una nueva partida.
  + Update; en este método incluimos todo lo que el sistema debe estar constantemente revisando. Aquí se ejecutarán las funciones que revisen el estado del personaje de cara a usar una animación u otra, comprobara si el jugador está pulsando los controles para que el personaje reaccione a ellos si está en el menú de juego y comprobara la posición para determinar si ha avanzado lo suficiente para subir de nivel.
  + SlowCoins; este método fijará la ralentización que sufrirá el personaje en caso de ir sobregargado. Como hay una habilidad que reduce el valor de esta penalización, solo se aplicará si el efecto es negativo para el personaje.
  + IsTouchingGround; este método lanza una línea desde el personaje hacia abajo determinando que si el rayo choca contra algún Collider de un GameObject en la capa “ground” en menos de 0.5f, el personaje está en el suelo, y que sino, está en el aire. Esto será importante para decidir si el personaje puede saltar o no.
  + IsRunning; comprueba la velocidad del personaje, si no es cero es que el personaje se está moviendo. Esto afectará a la animación a mostrar en cada momento.
  + MoveLeft y MoveRight; estos métodos serán los que harán que el personaje se mueva cuando el jugador pulsa las flechas de control en el menú de juego. Además girarán el Sprite del personaje en función de hacia donde se esté moviendo.
  + DontMove; este método se activa cuando el personaje suelte el botón de moverse y detendrá el movimiento. Esto era necesario porque el componente onClick hacia que se moviese solo un poco, por lo que para moverlo tuve que utilizar un componente que ejecutara la función al pulsarlo y este se ejecuta al soltarlo. Esto no es necesario para movimientos con teclas de ordenador, pero si para utilizar los botones táctiles de un móvil.
  + Kill; este es el método encargado de matar al personaje. Esto ocurre, bien cuando su salud lleva a cero o cuando choca con el gameOjbect KillZone de los Level Blocks. Además controlará el número de monedas en el inventario en el momento de la muerte y las guardara en caso de superar el máximo anterior.
  + GetDistance; restara la posición x actual del personaje a la inicial para determinar cuanta distancia se ha movido. Con esto fijamos la subida de nivel.
  + CollectHealth; este es el método que usaran los bastoncillos para devolver salud al personaje.
  + OnTriggerEnter2D; este método nos permite interactuar con los enemigos, si les damos en el Collider de la cabeza se destruirán, pero si nos dan con el cuerpo nos harán daño.
* PalyerBar; con este simplemente controlamos el nivel de las barras de vida y mana, asignándoles los valores actuales del personaje.
* CameraFollow; este script permite que la cámara siga al personaje de una forma fluida y evitando saltos bruscos. Establecerá la posición del jugador respecto del mapa y de la escena para colocarlo en el centro de esta última. También, en su método ResetCameraPosition devolverá la cámara a la posición inicial cuando el nivel comience, evitando el regreso de forma fluida por todo el mapa anterior.
* Skills; este script controla las mejoras que puede comprar el personaje con sus monedas en el menú skills. Estas habilidades son:
  + HealthBonus; un incremento a la salud máxima de 10 puntos.
  + ManaMobuns; un incremento al mana máximo de 5 puntos.
  + SuperJump; un descuento en el consumo de mana para realizar esa habilidad de 5 puntos.
  + RunningSpeedBonus; incrementa la velocidad de movimiento en 0.1f.
  + JumpForceBonus; aumenta la fuerza de salto en 0.1f permitiéndole saltar más alto.
  + CoinResistance; reduce el efecto negativo de cargar con monedas permitiendo cargas más antes de verse penalizado por ellas.
  + SkillsDiscount; reduce el coste todas las mejoras en una moneda.
* ViewCanvasGame; este script controla los valores que se muestran en las interfaces de usuario respecto a monedas, nivel, y coste de habilidades en el menú de compra. Todos los cambios se incluyen en el update porque deben mostrarse en tiempo real. Funciona creando unas variables tipo texto públicas que asignaremos en el editor de Unity a los texto que quedemos que afecten y asignándoles un valor en el script (normalmente un texto y su valor).
* Client; este es el script que permitirá al game manager guardar y cargar los datos desde la base de datos. El game manager hará uso de métodos SaveGame y LoadGame que tomarán por parámetro un String.
  + LoadGame; este método carga los datos de la base de datos. Ejecuta 3 metodos a su vez:
    - Connect; conecta nuestra aplicación al servidor con DNS “ripoll.ddns.net” y al puerto 12345.
    - Send; envía la información contenida en el String convertida a bytes mediante un socket. La información de este String varia en function de si lo usamos en la carga o el guardado.

Para guarder datos este String contiene la información del deviceUniqueIdentifier, nivel, HealthBonus, ManaBonus, CoinResistance, RunningSpeedBonus, JumpForceBonus, SuperJumpCost,CandiStickCure, SkillsDiscount; HealthBonusCost, ManaBonusCost, CoinResistanceCost, RunningSpeedBonusCost, JumpForceBonusCost, SuperJumpCostCost, CandiStickCureCost, SkillsDiscountCost separados por “;”.

En el caso de la carga, solo se envia el deviceUniqueIdentifier.

El servidor es quien controla el tamaño del String enviado para decidir si queremos guardar o cargar.

Receive; recibe la respuesta del servidor y almacena los trozos del String en las variables del script de skills con sus setters. Las distintas partes del String vienen separadas por un espacio en blanco.

* + - Disconnect; este método cierra la conexión con el servidor, lo que le vuele a poner en espera de una nueva conexión.
  + SaveGame
    - Connect;
    - Send;
    - Disconnect
* Servidor; el servidor está formado por 2 clases, la primera se encarga de conectar con la aplicación y la segunda de conectar con la base de datos.

Por la parte de la conexión con la aplicación tenemos 2 métodos, el primero que abre la opción a conectarse al cliente y otra que se mantiene siempre esperando un cliente y que se reinicia en bucle cuando este se va. Una vez establecida la conexión va recibiendo bits y almacenándolos en una variable hasta que recibe el símbolo “\*”. Una vez recibido lo parte en trozos separados por “;” y en función de los trozos que tenga hace una cosa u otra. Si el mensaje solo tiene un trozo, significará que el cliente está solicitando información, pero si tiene más, será para guardarlos. Para cargar los datos recurrirá al método load de la otra clase que le devolverá un String con todos los datos y se lo enviará, y para guardarlos se los enviará como parámetros del método save.

Por la parte de la conexión con la base de datos, nos encontraremos 4 métodos:

* + ConnectBBDD; este método conecta nuestro servidor con la base de datos y crea el statement donde se almacenaran las consultas.
  + Disconnect; cierra la conexión.
  + Load; hace una consulta a la tabla de nivel, a la tabla de personaje y a la de habilidades y las coloca en un String.
  + Save; coloca en variables los datos recibidos del cliente, borra los datos de la tabla relativos al identificador del teléfono y guarda los nuevos, repite el proceso para las otras 2 tablas.
* DDBB

Para guardar nuestros datos de una partida a otra, utilizaremos una base de datos en MySQL almacenada en un servidor. Para el proyecto, he utilizado una raspeberry Pi 4. En la misma he instalado Raspbian como sistema operativo por ser el recomendado por el fabricante. Una vez instalado el sistema operativo, he instalado una base de datos MySQL y la JVM. Utilizaremos este último para ejecutar nuestro programa intermediario entre el juego y la base de datos. La base de datos se llama Unity y esta formada por 3 tablas:

* + Level; en esta tabla se almacena el identificador del usuario y el nivel en que se encuentra. A futuro espero ampliar su funcionalidad incluyendo elementos que varíen en función del nivel y que deberán almacenarse aquí.
  + Personaje; aquí guardaremos las habilidades compradas en el Canvas skills, de modo que en cada inicio de partida el personaje recupere las habilidades que había adquirido hasta el momento.
  + Skills; en esta última tabla guardamos el coste de cada habilidad. Por el momento el coste es lineal, pero quizás haya que alterar el crecimiento de estos costes en función del progreso del juego para favorecer unas habilidades frente a otras según la forma de juego del jugador.

# 6. DESPLIEGUE Y PRUEBAS

A medida que iba implementando partes del juego he tenido que ir probando no solo su correcto funcionamiento, sino también su efecto en el resto de componentes. Por ello las pruebas realizadas han sido las siguientes

|  |  |
| --- | --- |
| 6.1.- PLAN DE PRUEBA **Nº** | **ESPECIFICACIÓN DE PRUEBAS** |
| **1** | Objetivo probado: caída atraviesa colliders  Requisitos probados:  Pruebas Que Realizar: |
| **2** | Objetivo probado: Aparicion y borrado de bloques.  Requisitos probados:  Pruebas Que Realizar: |
| **3** | Objetivo probado: Conexión cliente-servidor-BBDD  Requisitos probados:  Pruebas Que Realizar: |

Velocidad y capacidad de salto básicas del pj

Efecto mejoras en características, cantidad de salto, cuanto incrementar cada variable

Efecto de monedas en el personaje, intensidad

Efecto de los Canvas en las funciones de los GameObjects.

Distribución de Colliders en los enemigos, es posible darles en la cabeza sin sufrir daño

Posición de los elementos móviles respecto a las capacidades de los personajes

|  |
| --- |
| **Prueba de fondo transparente** |
|  |
| **Prueba de audios y manejabilidad con teclas (doble salto).** |
| **Prueba del cambio de estado y bloqueos de funciones fuera del estado de juego** |
| **Prueba de posición de los nuevos bloques y accesibilidad a nuevos bloques en juego** |
| **Prueba de desaparicion tras recoleccion, calculo del daño** |
| **Prueba de posibilidades del personaje** |
|  |
| **Prueba de capacidades y efecto de los incrementos** |

# 7. CONCLUSIONES

## 7.1.- OBJETIVOS ALCANZADOS

Se ha alcanzado el objetivo principal, que era diseñar un juego en Unity que cumpla con todos los requisitos funcionales establecidos inicialmente en el proyecto. En este sentido el juego cuenta con muchas opciones de mejora del personaje, dando al jugador completa libertad en cuanto a cómo proceder al tener 8 habilidades que ir mejorando.

En cuanto a la duración del juego, no solo se ha cumplido el objetivo de tener un par de niveles, sino que se han sobrepasado al diseñar un juego interminable, donde la adición de niveles sirve para darle más variedad a los escenarios.

Aunque se ha incluido una base de datos de cara a cumplir los requisitos impuestos por el centro para el desarrollo del proyecto, considero que su uso es un poco forzado, ya que nada en este juego necesita realmente acceder a una base de datos externa, y se podría almacenar toda esa información en las propias preferencias del sistema.

Si bien se le podría haber dado más calidad gráfica importando mejores Sprites, que generalmente son de pago, y animaciones, considero que no era relevante de cara a poner en prueba los conocimientos adquiridos durante el curso.

## 7.2.- CONCLUSIONES DEL TRABAJO

En mi opinión el trabajo ha sido interesante ya que me ha hecho trabajar con muchas librerías que desconocía, utilizar un motor gráfico muy potente y programar en 2 lenguajes de programación (C# para el cliente y Java para el servidor), además de poner en marcha una base de datos y acceder a ella desde un aplicativo usando un servidor de intermediario. Con esto último he aprendido la necesidad de establecer este sistema de cara a no proporcionar a los clientes las claves de nuestra base de datos, lo que les permitiría hacer un uso inapropiado de ella.

Además, me ha permitido comprobar lo importante que es ceñirse a lo establecido en la fase de diseño, ya que según avanza el proyecto van surgiendo ideas que si bien permiten mejorar el resultado en gran medida suponen un riesgo enorme de no terminar en plazo.

Por otro lado, ha habido momentos terriblemente desesperantes, por ejemplo, si durante la creación de los prefabs estos no se encuentran en su posición 0,0,0 pueden dar muchos problemas al incorporarlos en la escena ya que su posición variará. Esto me ha ocurrido en el LevelGenerator, que hasta que descubrí a que se debía que los nuevos bloques aparecieran en posiciones extrañas, estuve unas cuantas horas ajustando las posiciones para cada bloque en su script.

## 7.3.- VÍAS FUTURAS

De cara a continuar con su mejora lo prioritario a mi entender es añadir más levelblocks, obviando la búsqueda de un nombre más comercial. Actualmente cuenta con 3 ya que para probar su funcionamiento a nivel educativo era indiferente incluir 3 o 300 ya que iba a funcionar de la misma manera, pero si queremos que sea menos repetitivo y más divertido creo que sería interesante añadir más niveles, quizás con elementos que solo sean accesibles cuando el personaje ya tenga cierto nivel de desarrollo, como plataformas o monedas más altas o más lejanas.

Creo que también sería divertido incluir más variedad de enemigos. Si bien seria simplemente cambiarles el Sprite y la animación visualmente puede significar una mejora importante.

Otro aspecto importante sería la mejora de los Sprites existentes, por de baja calidad gráfica. Aunque no afectan a la jugabilidad ni a su diseño, son importantes de cara al interés que puede generar en os jugadores.

También podría incluir animaciones distintas, como plataformas que se muevan horizontal o diagonalmente, o incluso que dependan unas de otras para saltar de un punto a otro.

También había pensado en la posibilidad de incluir otras habilidades para el personaje que se desbloqueasen cuando tuviese alguna de las actuales en un nivel lo suficientemente alto. Esto puede generar la sensación de premio tan buscada por el jugador y las ganas de continuar avanzando para desbloquearlas.

Aumentar la dificultad a medida que se suben niveles, además de la dificultad actual de que cuanto más se mejora una habilidad más cara es y acumular esas monedas durante la partida implica una dificultad añadida por su peso.

Respecto a su conexión a la base de datos si se espera que haya más jugadores simultáneos solicitando información, habría que adaptar el servidor para que pase las comunicaciones a hilos. También crear más tablas o ampliar las existentes si añado más características almacenables

# 8. GLOSARIO

* **Sprite**: Los Sprites son mapas de bits que representan objetos gráficos 2D. Son esencialmente texturas estándar, pero hay técnicas especiales para combinar y manejar texturas Sprites por eficiencia y conveniencia durante el desarrollo.
* **Sprite renderer**: es el componente de Unity que controla la apariencia del Sprite. Con el podemos modificar muchas características del Sprite, como el material o el color.
* **Tilemap**: son mapas de Sprites agrupados en una única imagen que se dividirán en Sprites individuales con el Sprite editor. Se utilizan para generar escenarios con pequeñas piezas como si fuese un puzle.
* **Animación**: es la capacidad de los objetos de cambiar. Esto se consigue de varias formas, como puede ser utilizando una serie de Sprites que se van cambiando para dar la sensación de movimiento (como los antiguos dibujos animados) o alterando alguna de las características del objeto, como la posición de las plataformas del juego.
* **Audio source:** es el componente que se añade a los GameObject para que reproduzcan sonidos. En él se puede incluir una pista de audio o asignársela por código. Con él se puede controlar además cuando debe comenzar, el volumen, si debe reproducirse en bucle, etc.
* **GameObject**: Cada objeto del juego es un GameObject, desde personajes y objetos coleccionables hasta luces, cámaras, etc. Sin embargo, un GameObject no puede hacer nada por sí mismo; necesitamos asignarle componentes para haga lo que necesitamos. El único componente básico que incluye todo GameObject es el Transform, que le asigna una posición dentro del juego. Unity tiene muchos tipos de componentes integrados diferentes, y también podemos crear nuestros propios scripts utilizando la Unity Scripting API.
* **Prefab:** el sistema de prefabs de Unity permite configurar y guardar GameObjects completos, con todos sus componentes. El prefab hara de plantilla que permitirá crear nuevas instancias del objeto original. Estas instancias se pueden modificar con posterioridad y tienen la comodidad de que cualquier cambio en el prefab padre pueda aplicarse automáticamente a los hijos. Esto es especialmente útil cuando hay muchas copias de un objeto, como por ejemplo las monedas.
* **RigidBody**: es el componente principal que permite el comportamiento físico de cualquier GameObject. Si asignamos un Rigidbody a un GameObject, este responderá a la gravedad. Además, le asignamos un collider el objeto se verá afectado por colisiones entrantes. El componente Rigidbody tiene una propiedad llamada Is Kinematic que elimina al GameObject del control del motor de física y permite que se mueva de forma cinemática desde un script. Aunque se puede cambiar el valor de Is Kinematic desde un script para que la física se active y desactive, esto supone una sobrecarga de rendimiento.
* **Collider**: Los componentes Collider definen la forma de un objeto para los propósitos de colisiones físicas. Un Collider no necesita tener exactamente la misma forma que el mesh del objeto y normalmente una aproximación es más eficiente e indistinguible a efectos practicos.

Los colliders más sencillos de procesar son Box Collider, Sphere Collider y Capsule Collider. Sus correspondientes en 2D, serán Box Collider 2D y Circle Collider 2D.

Un único objeto puede tener muchos de estos al mismo tiempo para aproximar la forma de un manteniendo una sobrecarga baja del procesador. En los casos en que la agrupación de estos Colliders primitivos no son lo suficiente precisos se puede utilizar Mesh Colliders (3D o el Polygon Collider 2D para encajar la figura del mesh del objeto exactamente, aunque estos colliders son mucho más intensivos al procesador que los anteriores.

* **UI**: La interfaz de usuario (user interface, UI) permite la interacción entre la aplicación y el usuario. Hace referencia a todos los elementos que con los que debe interactuar el personaje, como la gestión de vida y mana, controles de movimiento y salto, monedas, puntuación, etc. En el proyecto he colocado todos los elementos relativos a la UI dentro de varios Canvas que van mostrándose y ocultándose en función del estado del juego en que nos encontremos en cada momento.
* **Canvas**: El Canvas es el área donde se agrupan todos los elementos UI. El Canvas es un Game Object con un componente Canvas en él, del cual cuelgan todos los elementos UI. El Canvas se muestra como un rectángulo en la Vista de Escena, de esta forma es fácil posicionar los elementos UI en él. Los elementos UI se dibujan en el Canvas en el mismo orden que aparecen en la Jerarquía de modo que el primer hijo es dibujado primero, después el segundo, etc.
* **IDE:** es una aplicación informática que facilita al programador el desarrollo de software. Normalmente, tiene 3 partes: un editor de código fuente, herramientas de construcción automáticas y un depurador. La mayoría de los IDE tienen auto-completado inteligente de código (IntelliSense). Algunos IDE, como NetBeans o Eclipse, contienen un compilador, un intérprete, o ambos. Muchos IDE modernos también cuentan con un navegador de clases, un buscador de objetos y un diagrama de jerarquía de clases, para su uso con el desarrollo de software orientado a objetos.

# 9. BIBLIOGRAFÍA

* Curso Udemy “Nuevo Curso de Desarrollo de Videojuegos con Unity 2018 y C#” <https://www.udemy.com/>
* Api de Unity <https://docs.unity3d.com/es/530/ScriptReference/index.html>
* Api de C# <https://docs.microsoft.com/es-es/dotnet/csharp/>
* Página de consulta <https://stackoverflow.com/>
* Wikipedia <https://es.wikipedia.org/>

# 10. ANEXOS

## 10.1.- MANUAL DE INSTALACIÓN.

Para la instalación del aplicativo solo es necesario descargar el archivo.APK y abrirlo desde el teléfono. Al ser una aplicación para dispositivos Android, en cuanto se pulse en su icono se abrirá el proceso de instalación, solo hay que darle a aceptar y esperar a que su instalación finalice.

Requisitos:

* 29mb libres de espacio en disco,
* Versión de Android 4.4 Kitkat o superior.

Por la parte que yo he instalado en mi servidor, he utilizado una Raspberry Pi 4 con las siguientes características:

* Procesador: Quad Core Cortex A-72 1,5 GHz,
* Memoria RAM: 4 GB LPDDR4,
* Tarjeta de memoria microSDXC de 64GB.

Además, será necesario tener instalado en el servidor:

* La JVM,
* La Base de datos MySQL, cuya construcción incluyo en este proyecto, creada a través del MySQL Workbench.

## 10.2.- MANUAL DE USUARIO

Tras la instalación indicada en el punto anterior y la apertura del juego, este se encontrará en el menú principal. Este ofrecerá 3 botones:

* Play: carga los datos de una sesión anterior e inicia una nueva partida continuando con el nivel y estado del personaje en que terminara en la sesión anterior.
* Buy Skills: este botón te lleva a la zona de compra de habilidades. Estas se comprarán con las monedas que se recogen en el modo de juego. Al comenzar cada nivel se pierden aquellas que no se hayan utilizado.
* Exit: cierra la aplicación.

Dentro del menú Skills aparecerán las distintas habilidades que se pueden adquirir (detalladas en el apartado 5.3 de este manual) así como su coste. Todas ellas tienen un efecto en el personaje y/o en las propias habilidades.

Los menús de Game Over y Level End tienen cada uno las opciones de volver al juego, reiniciando la partida actual en el caso del primero o comenzar el siguiente nivel en el caso del segundo, y la posibilidad de regresar al menú principal.

Por último y más importante está la pantalla de juego. En ella se habilitarán 5 botones que permitirán movernos a derecha e izquierda, saltar, usar una habilidad especial para dar un gran salto consumiendo nuestro mana y por ultimo rendirnos para reiniciar el nivel en caso de ver que no somos capaces de completarlo.

En el modo de juego, nuestra misión será ir avanzando por el nivel sin chocar contra los enemigos que patrullan sus zonas ni caernos de las plataformas. Durante nuestro avance podremos ir recogiendo monedas y bastones de caramelo. Los bastones de caramelo repondrán nuestra salud tras un encuentro con algún enemigo y las monedas las usaremos para mejorar nuestras características en el menú de compra de habilidades. Sin embargo, hay que ser cuidadoso en su recogida ya que una sobrecarga de ellas podría ralentizar tanto nuestro personaje haciéndole incapaz de superar los obtaculos que encontraremos en nuestro camino hacia el fin del nivel.