Facultad de Ciencias

FROG QUEST: DESARROLLO DE UN VIDEOJUEGO 2D CON EDITOR DE NIVELES

(Frog Quest: 2D videogame development with level editor)

Trabajo de Fin de Grado
para acceder al

GRADO EN Ingeniería Informática

Autor: Pedro Monje Onandía

Director: Carlos Blanco Bueno

Febrero – 2023

Resumen

El sector de los videojuegos crece con fuerza desde hace unos años, compitiendo con el cine y la música por el control del mercado del ocio. Esto nos hace pensar que solamente los grandes estudios con millones de euros en presupuesto pueden realizar un buen videojuego, pero no es así: hay equipos pequeños, conocidos como desarrolladores *indie*, que realizan algunos juegos considerados obras de arte por el público.

Debido a mi admiración y respeto hacia los videojuegos desarrollados por equipos pequeños, este proyecto tiene como objetivo la creación de un videojuego *indie*, el cual pertenece al género de plataformas en 2 dimensiones y cuenta con 3 niveles. En estos, el usuario tendrá que batir obstáculos y enemigos para llegar a la meta en el menor tiempo posible, mientras recoge monedas y se apoya en los *powerups* para avanzar con mayor rapidez. Además, el juego cuenta con una tabla de puntuaciones *online* para cada nivel, así como con un editor de niveles en el que los usuarios pueden crear niveles personalizados y compartirlos, permitiendo así que otros jugadores los descarguen y los jueguen.

El desarrollo se ha realizado utilizando el motor Unity junto con el lenguaje de programación C#, ya que funcionan muy bien en conjunto. Para la clasificación *online*, así como para la gestión de los niveles personalizados creados por los usuarios, se ha utilizado la plataforma LootLocker.

Palabras clave: Videojuego**,** Plataformas**,** Editor de niveles**,** Tabla de puntuaciones**,** Unity**,** LootLocker.

Abstract

The video game industry has been growing strongly for a few years now, competing with film and music for the control of the entertainment market. This makes us think that only big studios with budgets of millions of euros can make a good video game, but this is not the case: there are small teams, known as indie developers, who make some games that are considered works of art by the public.

Due to my admiration and respect for video games developed by small teams, this project's goal is to create an indie video game, which belongs to the 2D platform genre and has 3 levels. In these, the user will have to overcome obstacles and enemies to reach the goal in the shortest time possible, while collecting coins and relying on powerups to advance faster. In addition, the game features an online leaderboard for each level, as well as a level editor where the users can create custom levels and share them, allowing other players to download and play them.

The development has been done using the Unity engine in conjunction with C# programming language, as they work very well together. For the online ranking, as well as for the management of the custom levels created by users, different features of the LootLocker platform has been used.

Key words: Videogame, Platformer, Level editor, Leaderboard, Unity, LootLocker.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. Introducción

La industria de los videojuegos ha crecido con fuerza en los últimos años, pudiéndose comparar incluso con el cine o la industria musical, se estima que en el año 2021 el mundo de los videojuegos reunió más de 3 billones de jugadores a lo largo y ancho del globo [1], generando un beneficio de 180.3 billones de dólares (Figura 1).

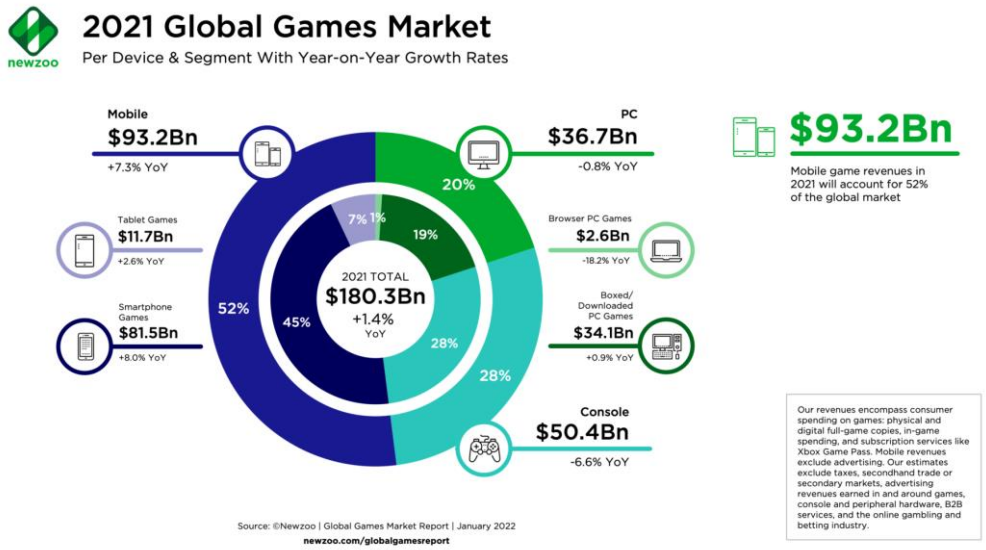


Figura 1. Análisis del mercado global de videojuegos en 2021

El crecimiento de la industria va acompañado de un aumento en la competitividad entre los jugadores, nacida en los salones de máquinas recreativas luchando por el *High Score* y perpetuada hasta nuestros días en las competiciones masivas de *E-Sports,* como la final del campeonato mundial de *League of Legends* de 2021*,* en la que hubo 73 millones de espectadores simultáneos máximos [2].

Dejando el factor competitivo a parte, los videojuegos también tienen un increíble factor comunitario, que provoca que mucha gente comparta y hable de sus experiencias en plataformas como *Youtube* o *Twitch*, o incluso se anime a realizar sus propias creaciones para que cientos de personas que tienen la misma pasión disfruten de ellas.

Gracias a estos “apasionados” creadores, aunque la mayoría de videojuegos que triunfan y generan cantidades de dinero desorbitadas están desarrollados por grandes empresas como *Epic Games,* creadores del archiconocido *Fortnite,* hay una amplia variedad de material manufacturado por desarrolladores *indie*, equipos pequeños que normalmente abogan por la originalidad, y que en muchas ocasiones acaban creando obras muy queridas por el público y la crítica, a pesar de afrontar un auténtico reto al crear *software* con escasos recursos.

1.1. Objetivo

El objetivo principal del proyecto, por tanto, es enfrentarse al reto que supone crear un videojuego *indie* desarrollado por una sola persona, teniendo como metas secundarias ofrecer una faceta competitiva y una posible interacción entre jugadores, dos pilares fundamentales en los juegos de hoy en día.

Para cumplir estos objetivos, este proyecto recogerá el diseño y el desarrollo de un videojuego en 2 dimensiones basado en niveles, en los que habrá que llegar al final sirviéndose de diversas herramientas, intentando a su vez lograr la mejor puntuación que quedará reflejada en un *ranking online,* el cual hace alusión a la faceta competitiva anteriormente mencionada. Además, permitirá a los usuarios crear sus propios niveles y compartirlos con cientos de jugadores para fomentar la interacción entre los mismos.

2. Materiales y herramientas utilizadas

En este apartado se detallan los materiales y herramientas utilizadas para la creación del videojuego.

2.1. Unity

Unity es un motor de videojuego multiplataforma creado por Unity Technologies cuyo uso está muy extendido en el mercado actual (Figura 2), ya que permite crear juegos multiplataforma y experiencias interactivas 2D y 3D.

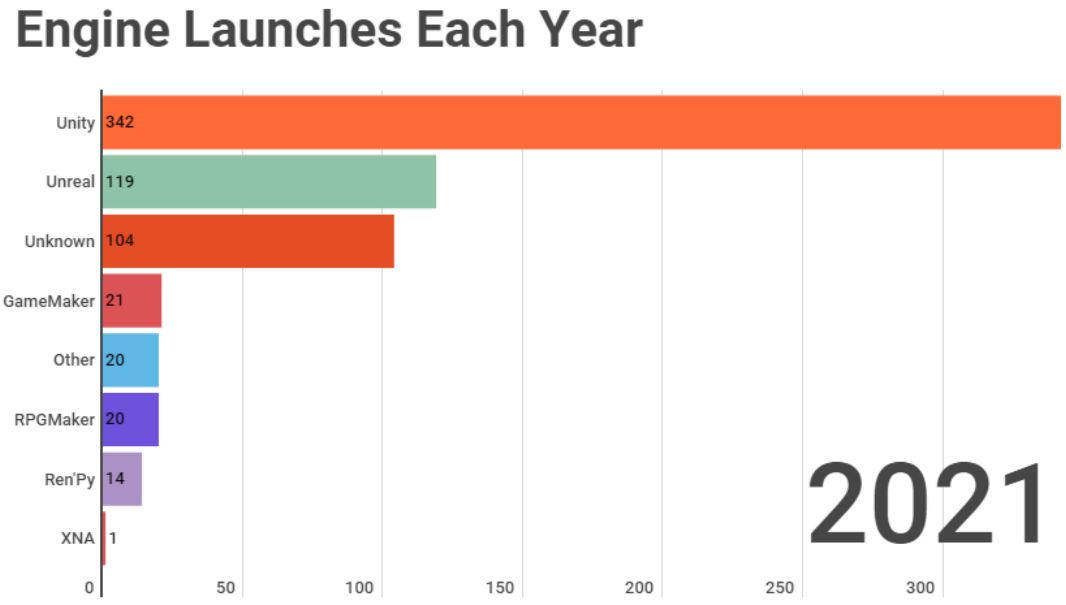


Figura 2. Numero de lanzamientos de juegos en Steam con cada motor, a partir de 4.99 euros de precio de venta

Esta herramienta es una elección perfecta para desarrollos primerizos como este proyecto, pues al estar tan extendida su utilización existen gran variedad de tutoriales, vídeos y ayudas hechas por usuarios, e incluso los propios desarrolladores proporcionan gran cantidad de documentación útil y completa acerca del motor y sus características.

Otro de los motivos para su uso es la variedad de librerías y recursos en la web, que hacen que trabajar con Unity sea mucho más sencillo y fluido que trabajar con otros motores, por ejemplo, Unity tiene una integración nativa con Visual Studio o permite trabajar con la tecnología Unity Cloud Build y almacenar el juego desarrollado en la nube.

Por último, cabe destacar que un motivo importante para la utilización de esta herramienta también es el precio, ya que si el usuario interesado en utilizarla ha obtenido beneficios o financiación inferiores a 100.000 dólares en los últimos 12 meses la herramienta es totalmente gratis. Cuando deja de ser gratuita existe una gran variedad de planes [3] entre los que escoger, siendo escalable por tanto la herramienta en función del crecimiento del proyecto.

2.2. Visual Studio y C#

Visual Studio es un entorno de desarrollo multiplataforma que permite trabajar con multitud de lenguajes de programación como C++, C# o Visual Basic. Se ha escogido este entorno para desarrollar el código por estar equipado con una buena matriz de herramientas y características para mejorar todas las etapas del desarrollo de *software*, así como por su ya mencionada integración nativa con Unity que mejora significativamente el flujo de trabajo.

Se utilizará C# para escribir el código, es un lenguaje de programación moderno, basado en objetos y con seguridad de tipos, desarrollado y estandarizado por la empresa Microsoft. Permite un acceso directo a memoria y recolecta elementos no utilizados para liberar esta. Además, no requiere que métodos y tipos sean declarados en un orden en particular. En resumen, C# es un lenguaje sencillo y flexible, lo que lo convierte en una buena elección.

Figura 3. Ejemplo de uso de C# con Visual Studio

2.3. LootLocker

LootLocker es una plataforma que permite administrar de manera sencilla diversos aspectos de un videojuego, como los datos de los jugadores o la adición de contenido. Es ideal para un desarrollo pequeño pues es gratis para juegos con menos de 10000 usuarios mensuales [4]. LootLocker proporciona un *Software Development Kit* (SDK) que permite una integración sencilla y ágil con Unity, esta compatibilidad con el motor ha sido la que ha provocado que se haya escogido esta plataforma, tanto para gestionar el sistema de creación y descarga de niveles personalizados, como para la gestión de las tablas de puntuaciones.

2.4. Assets, gráficos y animaciones

Todas las imágenes y recursos gráficos utilizados en el proyecto se han descargado de manera gratuita de la web Itch.io. Salvo un par de excepciones, la mayoría de estos pertenecen a los *packs Pixel Adventure 1* y *Pixel Adventure 2*, que constan de variedad de opciones para crear enemigos, *powerups*, personajes principales y elementos del escenario.

3. Metodología

Para el desarrollo del videojuego se ha utilizado la metodología iterativa incremental, que consiste en construir *software* en el que el ciclo de vida global está formado por varias iteraciones y cada una de ellas hace crecer el sistema resultante incrementalmente. Cada iteración se compone de análisis de requisitos, diseño, implementación y pruebas [5]. Este método de trabajo es ideal para algo tan cambiante como un videojuego, ya que permite probar cada característica añadida antes de pasar a la siguiente, evitando así arrastrar errores que puedan lastrar el correcto desarrollo del *software*.

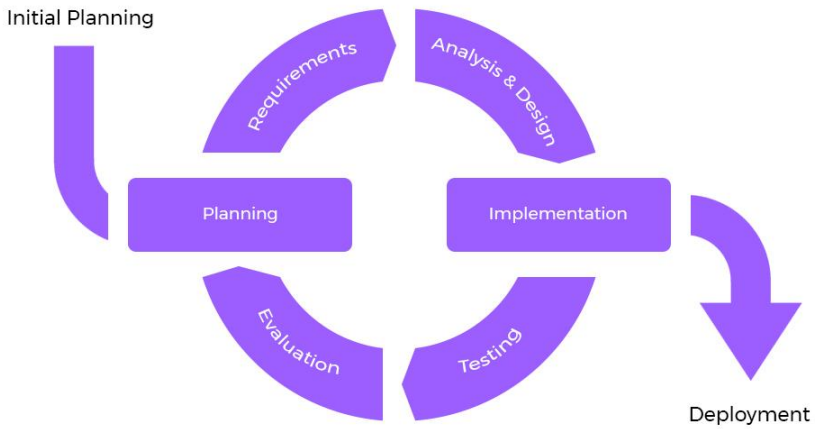


Figura 4. Representación gráfica del modelo iterativo incremental

3.1. Planificación

La creación del videojuego se divide en 3 etapas: preparación, desarrollo y realización de la memoria.

3.1.1. Preparación

Lo primero y más importante a la hora de llevar a cabo un proyecto es tener claro qué se va a hacer y qué herramientas se van a utilizar. En mi caso esto no llevó mucho tiempo, tras una pequeña investigación decidí que quería hacer un juego rápido y variado, para lo cual me parecía idónea la elección del género de plataformas. Además, con el fin de darle más profundidad y vida útil decidí que el juego debía incluir puntuaciones y un sistema de creación y descarga de niveles. En cuanto a las herramientas, escogí la combinación de Unity mezclado con LootLocker por la sencillez de ambas y la integración entre ellas.

Una vez que tuve claro el videojuego a realizar decidí formarme en el uso de Unity, herramienta clave para el proyecto. Para poder entender y controlar el uso de este programa en la creación de videojuegos 2D realicé el curso de Domestika "*Introducción a Unity para videojuegos 2D"* [6], creado por Juan Diego Vázquez Moreno, en el que se trabajan los distintos apartados de la creación de un videojuego como las animaciones o los *scripts*.

Tras realizar este curso, decidí crear un videojuego de prueba (Figura 5) en el que incluí enemigos, un set de movimientos variados y un sistema de monedas, vidas y medidor de tiempo. También incluí en esta pequeña demo una versión simplificada de un editor de niveles (Figura 6) para tener una idea de cómo abordar su creación definitiva en el futuro.

Figura 5. Nivel de prueba

Figura 6. Prototipo del editor de niveles

3.1.2. Desarrollo

La fase de desarrollo se ha dividido en iteraciones, cada una de las cuales consta de análisis de requisitos, diseño, implementación y pruebas:

- **Análisis de requisitos:** Se establecen una serie de objetivos que tiene que satisfacer el *software* desarrollado durante la iteración. Estos objetivos pueden ser nuevos servicios que tiene que prestar el sistema (requisitos funcionales) o restricciones en aspectos como seguridad, rendimiento, disponibilidad… (requisitos no funcionales)

- **Diseño:** Se planea la manera de satisfacer las características recopiladas en el análisis de requisitos. El diseño es algo transversal, que debe centrarse tanto en la programación como en otros aspectos como el visual o el sonoro.

- **Implementación:** Se desarrolla y añade al proyecto todo lo necesario para cumplir con los objetivos marcados en la primera etapa de cada iteración, no solamente se programa, también puede ser necesario por ejemplo configurar herramientas, imágenes o animaciones.

- **Pruebas:** Una vez se ha realizado la creación e incorporación de las nuevas características es necesario verificar que éstas funcionan correctamente, tanto dentro de la propia iteración como en el contexto global del proyecto.

3.1.3. Realización de la memoria

La memoria se ha ido realizando en paralelo con la fase de desarrollo, de tal manera que los avances se fueran documentando al mismo tiempo que se realizaban. Esto permitió tener claro que se estaba haciendo en todo momento y proporcionar una información más enriquecedora de todo el proceso de creación del videojuego.

3.1.4. Diagrama de Gantt

La duración de las fases se muestra en el siguiente diagrama de Gantt (Figura 7).

Nombre de tarea	Duración	Comienzo	Fin
Preparación	**35 días**	**mié 31/08/22**	**mar 04/10/22**
Decidir herramientas	5 días	mié 31/08/22	dom 04/09/22
Curso de Unity	20 días	lun 05/09/22	sáb 24/09/22
Realización de demo	10 días	dom 25/09/22	mar 04/10/22
Desarrollo	**94 días**	**mié 05/10/22**	**vie 06/01/23**
Iteración 1. Nivel 1	10 días	mié 05/10/22	vie 14/10/22
Iteración 2. Nivel 2	15 días	sáb 15/10/22	sáb 29/10/22
Iteración 3. Nivel 3	20 días	dom 30/10/22	vie 18/11/22
Iteración 4. Editor de niveles	30 días	sáb 19/11/22	dom 18/12/22
Iteración 5. Clasificación online	15 días	lun 19/12/22	lun 02/01/23
Iteración 6. Cambios menores	4 días	mar 03/01/23	vie 06/01/23
Realización de la memoria	122 días	mié 05/10/22	vie 03/02/23

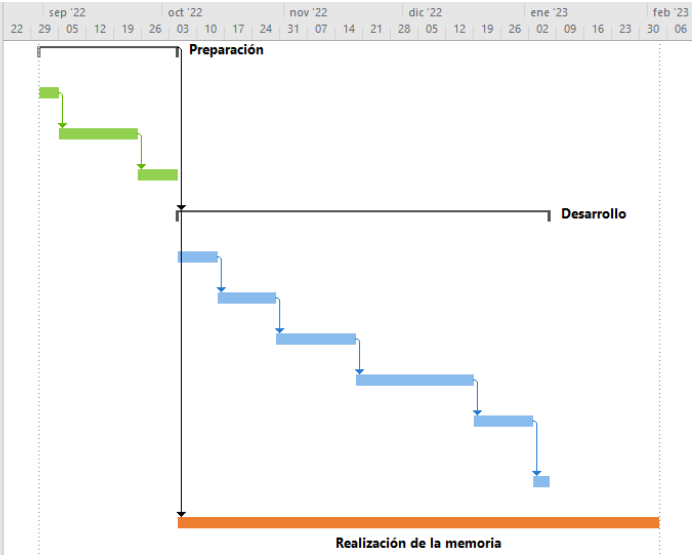


Figura 7. Diagrama de Gantt

4. Análisis de requisitos

En este apartado se proporciona una sencilla explicación conceptual del videojuego, acompañada de algunos *mockups*, además, se exponen los requisitos del proyecto, divididos en funcionales y no funcionales.

4.1. Descripción conceptual del videojuego

Frog Quest pretende ofrecer la posibilidad de jugar varios niveles diferentes con clasificación *online*. Dentro de cada uno de ellos, el objetivo será coger un trofeo intentando obtener la mejor puntuación del *ranking online*, para esto habrá que completar la fase en el menor tiempo posible, eliminando a todos los enemigos y recolectando todas las monedas repartidas por el mapeado. Para completar el nivel, el jugador contará con un *set* de movimientos muy variado, podrá utilizar diversos *powerups*, como por ejemplo corazones para recuperar vida, y podrá servirse de puntos de guardado o *checkpoints* para reanudar la partida si muere. Aparte de los diversos niveles básicos, el juego también dispondrá de un editor que permitirá a los jugadores crear niveles personalizados, que podrán compartirse y descargarse a través de LootLocker.

En la Figura 8 se muestra un *mockup* de la pantalla de juego dentro de un nivel, en esta se le mostrará al jugador información acerca de la partida como la vida que le queda, las monedas recogidas, el tiempo transcurrido y la puntuación actual, utilizando para ello una interfaz o *HUD* ubicada en la esquina superior izquierda. Otro aspecto importante mientras se está jugando es la cámara, la idea es que esta siga al jugador dentro del nivel hasta que toque el trofeo, pues al realizar esta acción se añadirá la puntuación del usuario al *ranking* y se mostrará este en pantalla.

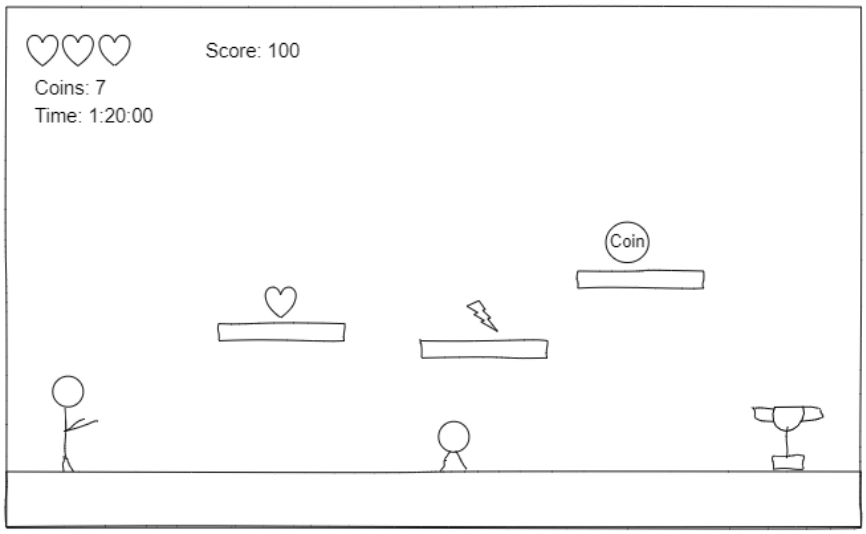


Figura 8. *Mockup* de la pantalla de juego dentro de un nivel

Por otro lado, en la Figura 9 podemos ver un mockup del editor de niveles. Se pretende que este cuente con un *scroll* de objetos que el jugador pueda seleccionar e ir colocando en la pantalla para conformar el nivel, el juego deberá permitir también el borrado de los objetos colocados. Además, a medida que el usuario vaya creando el nivel podrá probarlo pulsando el botón etiquetado como *"Test"* y posteriormente seguir editándolo, o podrá subirlo al servidor si considera que su creación está finalizada, para ello se abrirá un menú al pulsar *"Upload"* que permitirá ponerle un título al nivel y guardar este en las bases de datos de LootLocker. Por último, mediante el botón *"Download"* se mostrará una lista con la información de todos los niveles creados por los usuarios, y desde esta se podrá escoger un nivel, descargarlo y jugar.

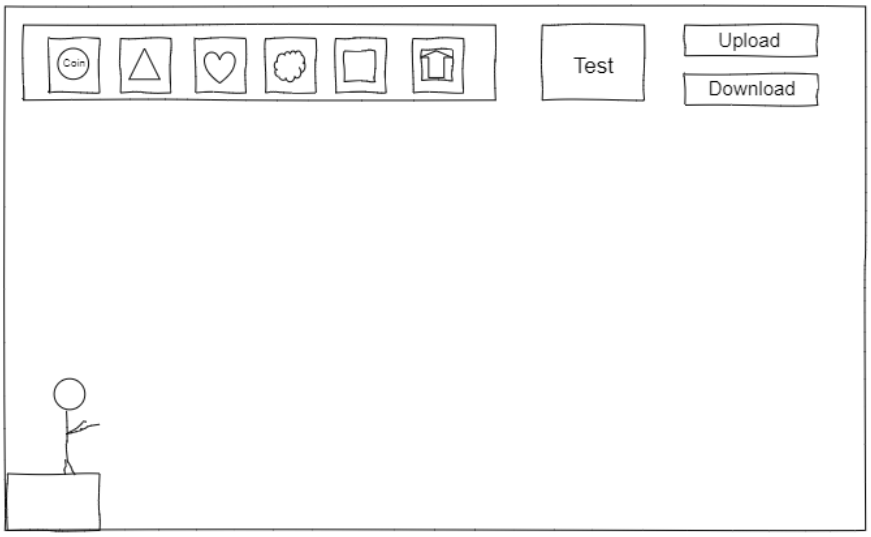


Figura 9. *Mockup* del editor de niveles

4.2. Requisitos funcionales

ID	Descripción
RF1	El jugador podrá moverse hacia a la izquierda y hacia la derecha, en el eje horizontal.
RF2	El jugador podrá saltar.
RF3	El jugador podrá realizar un doble salto.
RF4	El jugador podrá deslizarse por las paredes y saltar desde ellas.
RF5	El menú principal permitirá acceder a los distintos niveles y a una clasificación *online* por cada uno, así como a opciones de cambio de nombre de usuario, opciones de creación/descarga de niveles y un pequeño tutorial.
RF6	El jugador quedará identificado por un nombre de usuario que podrá cambiar cuando desee.
RF7	El juego tendrá 3 niveles básicos, que serán distintos entre sí y de dificultad creciente y contarán con 10 monedas cada uno, enemigos variados y ayudas para el jugador.
RF8	Dentro de los niveles el jugador podrá recoger pociones que le permitirán moverse más rápido de manera temporal.
RF9	Dentro de los niveles el jugador podrá recoger armas arrojadizas y atacar con ellas.
RF10	El jugador podrá recoger monedas en cada nivel.
RF11	El jugador tendrá 3 corazones de salud en cada nivel, que perderá cuando le ataquen los enemigos y podrá recuperar recogiendo curas repartidas por el mapeado.
RF12	El juego dispondrá de un contador de tiempo en cada nivel que se encargue de calcular cuánto está tardando en completarlo el usuario.
RF13	El juego dispondrá de un contador de puntuación en cada nivel, que dependerá del tiempo, las monedas recogidas, los enemigos eliminados y la dificultad del nivel.
RF14	El jugador tendrá una interfaz (*HUD*) que le indique el número de corazones que tiene, las monedas recolectadas, el número de armas arrojadizas que le quedan, y muestre el contador de tiempo y puntuación.
RF15	Los niveles contarán con enemigos variados, tanto en poder como en *set* de movimientos.

RF16	Los niveles contarán con plataformas móviles y temporales.
RF17	Los niveles contarán con terrenos variados, que tendrán efectos distintos en el movimiento del jugador.
RF18	Los niveles contarán con *checkpoints* o puntos de guardado en los que el jugador reaparecerá al morir si están activados.
RF19	Cada nivel contará con un punto de fin o trofeo que si es alcanzado por el jugador significará el fin de la partida.
RF20	La partida también finalizará si el jugador cae a un pozo o los enemigos le quitan todos los corazones y no ha tocado un *checkpoint.*
RF21	Si el jugador muere sin tocar un *checkpoint* podrá volver a intentar el nivel gracias a un menú de fin de juego o G*ame Over.*
RF22	Durante la partida se podrá abrir un menú para volver a la pantalla de inicio o reiniciar el nivel.
RF23	El juego contará con una *leaderboard* o tabla de puntuaciones para cada nivel básico.
RF24	Al finalizar el nivel tocando el trofeo aparecerá un menú, en el que el jugador podrá ver el tiempo que ha tardado y la puntuación conseguida, así como subir esta a la tabla de puntuaciones.
RF25	El jugador podrá crear niveles personalizados y subirlos a un servidor.
RF27	El jugador podrá probar el nivel personalizado creado antes de subirlo.
RF28	El jugador podrá descargar y jugar niveles personalizados creados por sí mismo o por otros usuarios.

Tabla 1. Requisitos funcionales

4.3. Requisitos no funcionales

ID	Tipo	Descripción
RNF1	Portabilidad	El juego se ejecutará en sistemas Windows.
RNF2	Usabilidad	El juego contará con una estética agradable *family friendly* que lo haga apto para todos los públicos.
RNF3	Accesibilidad	Todo el texto estará escrito en inglés para que llegue al mayor público posible.
RNF4	Usabilidad, Accesibilidad	El juego contará con interfaces sencillas e intuitivas y con la letra grande para su mejor comprensión.
RNF5	Rendimiento	El peso total del ejecutable no podrá exceder los 300 MB.
RNF6	Rendimiento	El juego tendrá una tasa de fotogramas por segundo de 60 o superior.
RNF7	Rendimiento	El juego deberá funcionar en distintas máquinas de manera estable, sin fallos en el rendimiento ni consumos altos de RAM o memoria convencional.

Tabla 2. Requisitos no funcionales

5. Diseño e implementación

En este apartado se realiza una explicación general y detallada de la arquitectura e implementación de cada parte del programa.

5.1. Diseño arquitectónico

El videojuego se ha desarrollado mediante un modelo arquitectónico en tres capas, en este tipo de arquitectura la funcionalidad del sistema se organiza en capas separadas (presentación, negocio y datos), y cada una depende exclusivamente de los servicios ofrecidos por la que se encuentra inmediatamente por debajo [7]. Esta arquitectura permite crear *software* muy cambiante y portable, lo que es ideal a la hora de desarrollar un proyecto de este estilo.

Figura 10. Arquitectura en 3 capas aplicada al proyecto

Cada capa cumple una función y proporciona una serie de servicios:

- **Capa de presentación:** Es la encargada de la interacción con el usuario, esta capa muestra información al usuario y también la recopila.

- **Capa de negocio:** Se encarga de comunicarse con la capa de presentación para recibir los datos recopilados y devolver el procesado para que sea mostrado al usuario. Además, también interactúa con la capa de datos, para almacenar o recuperar datos de esta.

- **Capa de datos:** Es la encargada de almacenar los datos del sistema y de los usuarios. Además, también se encarga de devolver datos a la capa de negocio.

5.2. Diseño e implementación del videojuego

En este apartado se comentarán los detalles del diseño y la implementación del videojuego explicados capa a capa, haciendo hincapié en ciertos aspectos considerados novedosos o interesantes.

5.2.1. Diseño e implementación de la capa de presentación

Las interfaces con las que el usuario tiene contacto han sido diseñadas para que sean sencillas e intuitivas: constan de una letra grande y una tipografía fácil de leer, contienen textos en inglés para poder llegar a un público más amplio y algunas contienen imágenes para entender mejor ciertas características. Todas las interfaces y menús están conectados para facilitar la navegación, tal y como se muestra en la Figura 11, de esta manera se consigue que el usuario esté el menor tiempo posible vagando entre interminables menús y listas de opciones, para así poder disfrutar sin impedimentos de la diversión y el desafío que le propone *Frog Quest.* Además, el jugador puede salir del juego desde varios puntos si se cansa de jugar.

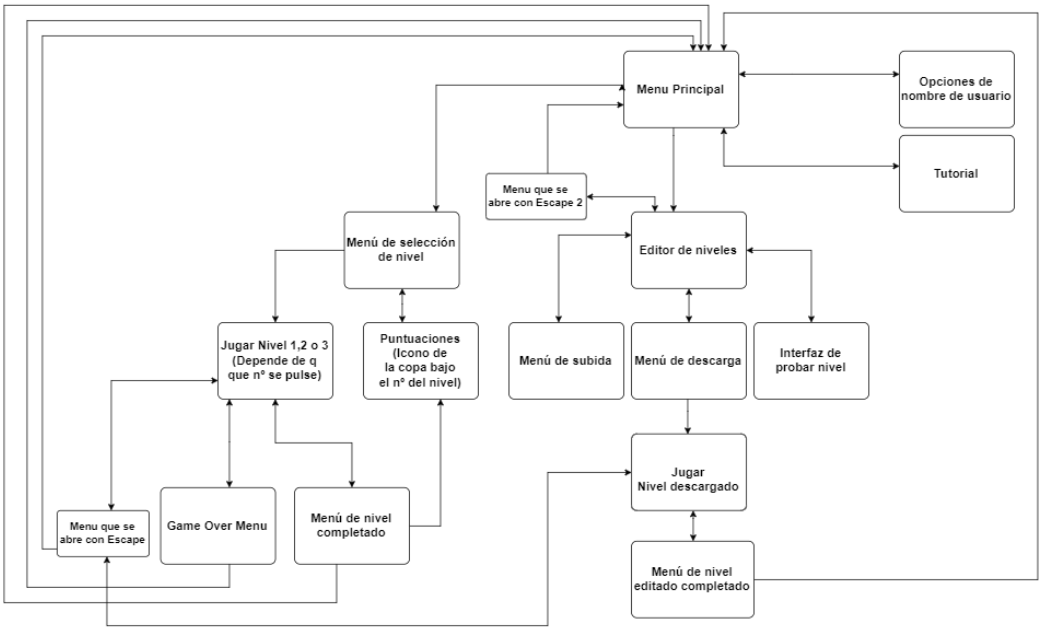


Figura 11. Diagrama de interacción entre menús

Tras mostrar de manera general el diseño de la capa de presentación del videojuego se pasará a explicar con más detalle ciertas vistas, interfaces y menús, añadiendo capturas de pantalla para la correcta comprensión de lo explicado.

5.2.1.1. Menú principal

Al iniciar el juego, el usuario se encontrará con el menú principal y sus diversas opciones. Como se puede observar en la Figura 12, se ha añadido color a los botones para que se note cuándo el puntero está encima de estos. Desde este menú el jugador puede abrir las opciones de nombre de usuario en la esquina superior derecha, entrar en la ventana de selección de niveles con el primer botón, abrir el editor de niveles con el segundo o mostrar un tutorial con el último. Si por el contrario pulsa la X de la esquina superior izquierda el juego se cerrará.

Por otra parte, se ha añadido un mensaje de ayuda que aparece cuándo se está encima de cualquiera de los dos botones superiores. Dicho mensaje indica que es esencial ponerte un nombre de usuario si abres el juego por primera vez (pues este nombre se utiliza para las tablas de puntuaciones y para identificar los niveles creados por un usuario), y que se puede cambiar tu *username* si no te gusta el que tienes.

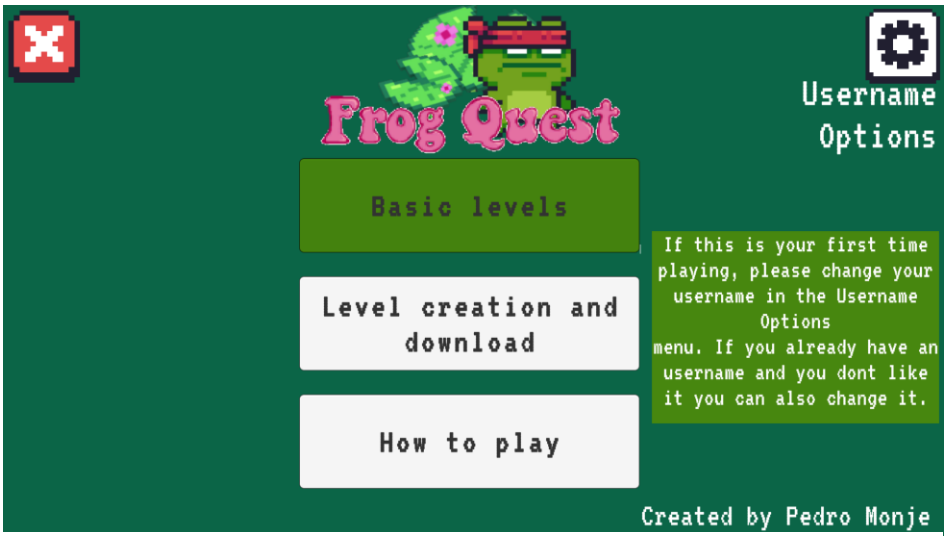


Figura 12. Menú principal

5.2.1.2. Menú de opciones de nombre de usuario

Si en el menú inicial pulsamos el botón que contiene el engranaje, aparecerá la pantalla en la que el usuario puede cambiarse el nombre. La Figura 13 es un ejemplo de cómo sería esta pantalla para un jugador que ya tiene un nombre de usuario asignado. En la parte superior se le muestra el nombre actual y en la inferior una caja de texto para introducir un nuevo nombre y un botón para hacer efectivo el cambio, también se ha añadido un mensaje de aviso que indica que si te cambias el nombre será más difícil encontrar los niveles que has creado, ya que al crear los niveles se concatena el nombre de usuario al nombre del nivel. Si finalmente el usuario decide cambiarse el nombre, este se actualizará en las tablas de puntuaciones de los distintos niveles, así como en la parte superior de esta misma pantalla. Por último, si el usuario pulsa la flecha en la esquina superior izquierda se volverá al menú principal.

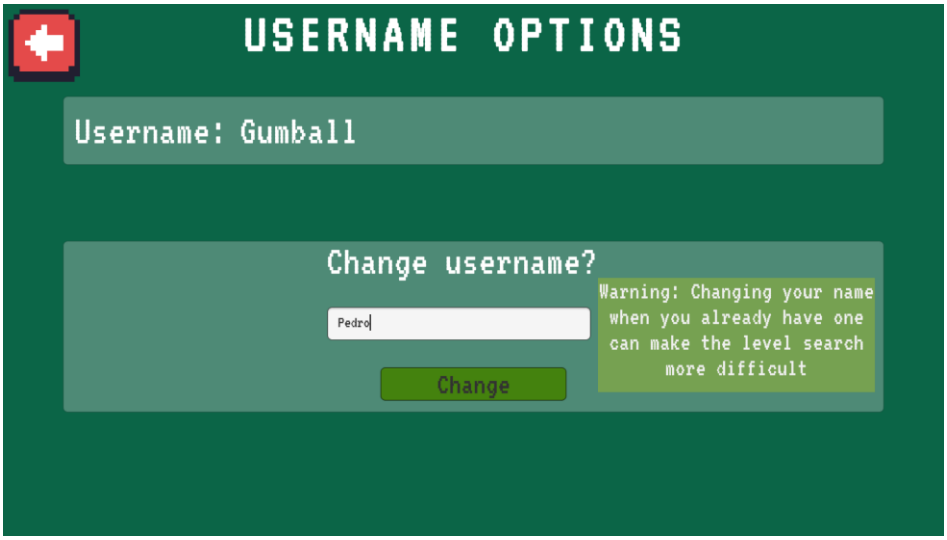


Figura 13. Menú de opciones de nombre de usuario

5.2.1.3. Pantalla de selección de niveles e interfaz de juego

Pasando a las opciones centrales del menú principal, si el usuario escoge *"Basic levels"* se abrirá la interfaz mostrada en la Figura 14, desde la que se puede acceder a los distintos niveles pulsando sus respectivos números o consultar las tablas de puntuaciones (Figura 15) mediante los iconos de copas que se encuentran bajo el nombre de cada nivel. Por otra parte, si el usuario pulsa la flecha en la esquina superior izquierda volverá al menú principal.

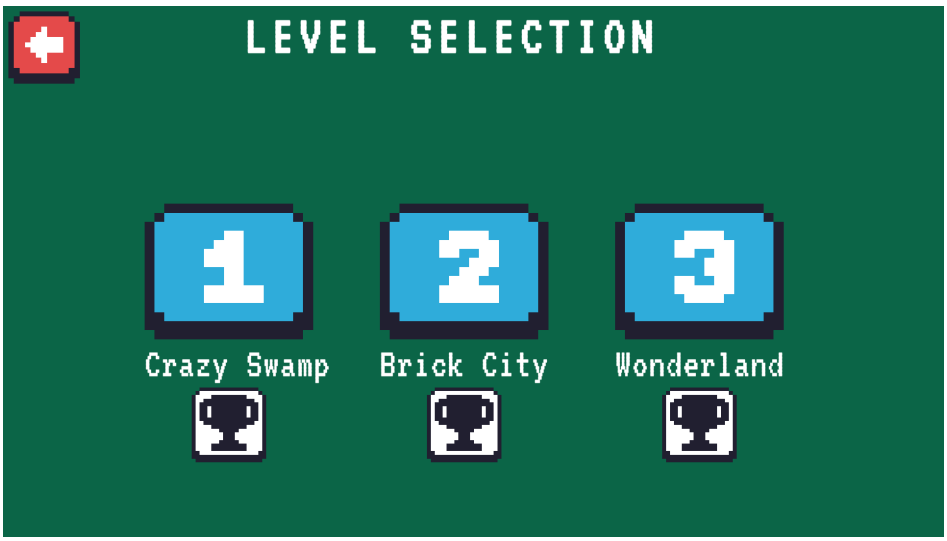


Figura 14. Pantalla de selección de nivel

Figura 15. Tabla de puntuaciones del nivel 2

Si el usuario decide jugar un nivel pulsando uno de los tres números se abrirá la pantalla de juego (Figura 16), con el personaje colocado al principio del nivel. Realizando un barrido visual dentro de esta pantalla tenemos el *HUD* con las vidas, el número de monedas recogidas, las armas arrojadizas restantes y el tiempo en la esquina izquierda, y, en la parte superior central de la pantalla se le indica al jugador los puntos que lleva mediante un contador.

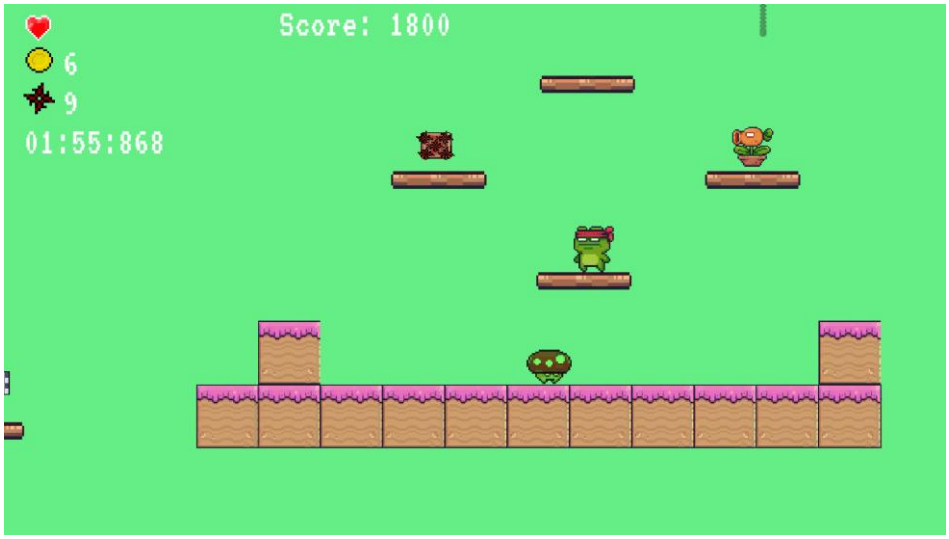


Figura 16. Pantalla de juego en el nivel 3

Cuando el usuario está jugando es muy importante que visualmente entienda lo que está pasando, por ello, uno de los elementos clave dentro de la pantalla de juego es la cámara, ya que influye mucho en la experiencia que tiene el jugador. La cámara está hecha para seguir al personaje de manera fluida y dejarle el suficiente espacio para ver los peligros a su alrededor, su realización ha sido posible gracias al paquete *Cinemachine* de Unity [8], que permite la creación de cámaras fluidas y profesionales de manera sencilla.

Para reforzar la comprensión del nivel mientras se juega es vital proporcionar *feedback* visual, por eso se han tomado varias decisiones importantes relacionadas con este aspecto:

- Todos los enemigos, así como el propio personaje tienen una animación creada que indica cuándo son dañados (Figura 17).

- Ciertas plataformas cambian de color o se mueven para indicar exactamente su funcionamiento (Figura 18).

- El juego cuenta con enemigos terrestres móviles, enemigos terrestres inmóviles y enemigos voladores. Para indicar cambios en su comportamiento estos enemigos son distintos visualmente entre sí (Figura 19).

- Hay variedad de *powerups* y son distintos entre sí pues cumplen con funciones diferentes: el corazón te cura, la poción te da velocidad y cambia el color de tu personaje para indicar este cambio, la caja con el dibujo de shurikens te proporciona armas para atacar pues los ataques son limitados y las monedas sirven para aumentar la puntuación. (Figura 20)

- Los objetos que pueden influir en la reaparición y finalización del nivel, es decir los *checkpoints* y el trofeo, están claramente diferenciados del resto. (Figura 21)

Figura 17. Animaciones del personaje principal y de un enemigo básico cuando son dañados

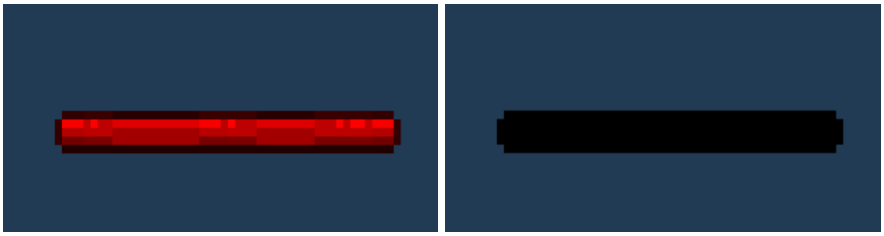


Figura 18. Cambio que sufre una plataforma al interactuar con ella

Figura 19. De izquierda a derecha, enemigo terrestre móvil, inmóvil y enemigo volador

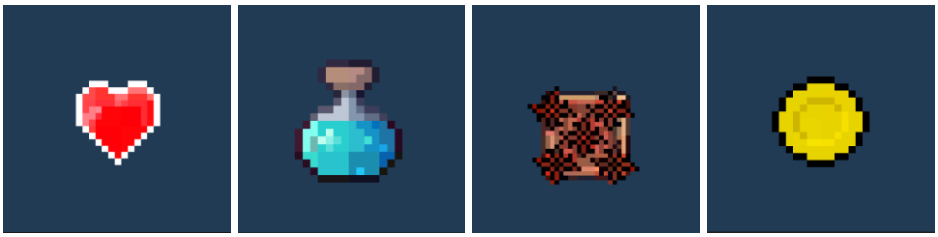


Figura 20. *Powerups*

Figura 21. *Checkpoint* y trofeo

Además del apartado visual es importante cuidar la interacción del usuario con la pantalla de juego, para ello se han incluido diversidad de menús con distintas funciones.

El primero de estos menús se abre con la tecla *Escape* y permite al usuario salir al menú principal o reiniciar el nivel mientras juega, abrir este menú no parará la partida ya que en un juego con niveles cortos como el que se ha desarrollado esto no tiene sentido (Figura 22).

Figura 22. Menú accionado por *Escape* en la pantalla de juego

Los dos últimos menús que pueden aparecer durante un nivel tienen que ver con la manera de finalizar la partida:

- Si el usuario muere sin tocar ningún *checkpoint* este no reaparecerá, sino que se le mostrará en pantalla un menú de *Game Over* (Figura 23) que permite reintentar el nivel o volver al menú principal.

Figura 23. Menú de *Game Over*

- Si por el contrario el nivel termina porque el usuario llega al final y toca el trofeo aparecerá el menú de nivel completado (Figura 24). Desde este se puede salir al menú principal, reintentar el nivel o subir la puntuación a la tabla de puntuaciones pulsando *"Submit"*. Si el usuario realiza esta última acción se le mostrará la tabla de puntuaciones actualizada en pantalla, cabe destacar que la puntuación del jugador en la *leaderboard* solo se actualizará con la obtenida en el nivel actual si esta es mejor que la que ya estaba almacenada.

Figura 24. Menú de finalización de nivel

5.2.1.4. Editor de niveles

Al seleccionar la opción *"Level creation and download"* del menú principal, el usuario entrará en la vista general de edición del editor de niveles (Figura 25), la cual está formada por un *scroll* con 23 objetos que pueden usarse para conformar un nivel, así como por un botón de probar el nivel (botón con la rana), uno para subirlo al servidor ("*Upload"*) y otro para descargar niveles editados por el usuario o por otros jugadores (*"Download"*). Además, este editor cuenta con un pequeño menú que se abre con la tecla *Escape*, el cual nos permite regresar a la pantalla de inicio; el hecho de hacer la salida del editor con un menú en vez de con un botón sirve para no sobrecargar la interfaz y mejorar el flujo de creación.

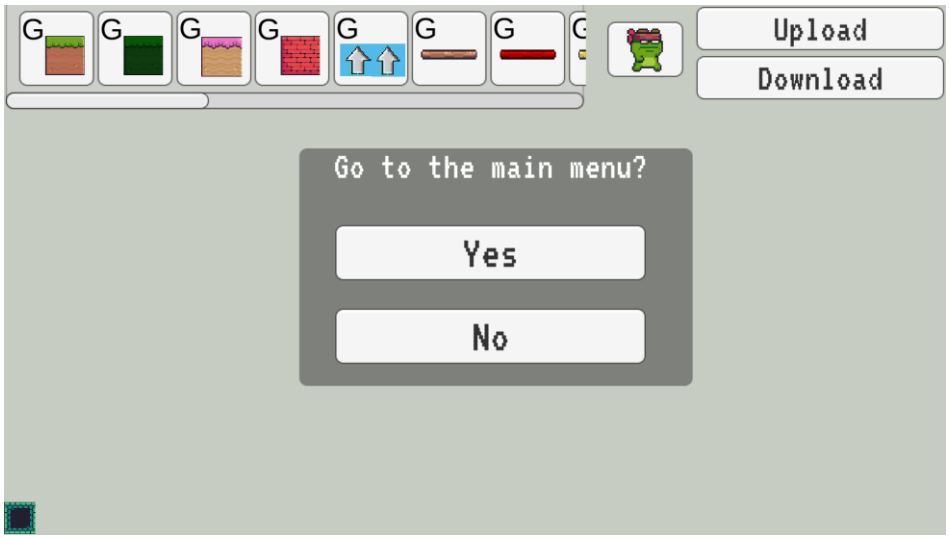


Figura 25. Vista general de edición

La piedra angular sobre la que se sostiene esta vista general de edición es el *scroll* de objetos, que están ordenados en categorías:

- G, categoría *ground*, objetos que actúan como suelo.
- W, categoría *wall*, objetos que actúan como muro.
- E, categoría *enemies*, enemigos variados, así como trampas.
- P, categoría *powerups*, objetos que ayudan al usuario.
- *Check*, categoría formada por los *checkpoints* o puntos de guardado, si se muere tocando un *checkpoint* se reaparece sobre él.
- *End*, categoría formada por el trofeo, si se toca finaliza el nivel.

Para cerrar la explicación de esta barra de objetos, es importante mencionar que algunos tienen restricciones, indicadas con un mensaje de ayuda al colocarnos encima de ellos en la barra superior. Para ser más exactos, el juego no permitirá colocar más de un trofeo, pues solo puede haber un punto de fin por nivel, y ciertos enemigos y plataformas deben colocarse entre bloques de categoría W.

La primera gran funcionalidad que tiene el editor de niveles es que proporciona al usuario herramientas para crear un nivel a su gusto y compartirlo. El primer paso para esto es clicar en uno de los objetos del *scroll*, lo que permitirá al jugador entrar en la vista de colocación de objetos, representada en la Figura 26. Dentro de esta vista, se muestra una guía para que vea dónde va a colocar el objeto y puede poner infinitas copias de este con *click* izquierdo. Para salir del sistema de colocación de objetos solo hace falta pulsar la tecla *C*, lo cual vuelve a mostrar el *scroll* de objetos y los botones, permitiendo clicar en otro *item* para volver a entrar en la vista de colocación y conformar el mapeado total del nivel. Además, el usuario puede eliminar objetos haciendo *click* derecho sobre ellos, tanto dentro de la vista general de edición como en la vista de colocación de objetos.

Figura 26. Vista de colocación de objetos

Como detalle interesante dentro de la creación de un nivel se ha añadido la posibilidad de que el usuario puede probar el nivel antes de compartirlo. Pulsando el botón con la imagen de la rana, se abrirá la vista mostrada en la Figura 27, al entrar en esta vista, el personaje aparece y puede controlarse con normalidad, permitiendo comprobar el correcto funcionamiento y compatibilidad de los objetos colocados. Dentro de esta pantalla hay un *HUD* como el utilizado en los niveles básicos, pero con una ligera modificación, pues nos muestra la vida, las monedas recogidas, los objetos de ataque restantes y el tiempo transcurrido, pero ignora la puntuación, ya que no se utilizará para nada en niveles personalizados. Dentro del modo de prueba si el usuario muere sin tocar *checkpoints* no habrá menú de *Game Over*, sino que reaparecerá en el bloque colocado en la parte inferior izquierda, conocido como bloque inicial, que se encuentra en la misma posición en todos los niveles editados, además, sucederá lo mismo si el jugador toca el trofeo. Por último, para salir del modo prueba y volver a editar el nivel basta con pulsar el botón superior con el texto *"Edit"*, esto ocultará al personaje y el *HUD* y volverá a mostrar el *scroll* de objetos y todos los botones que había anteriormente en pantalla.

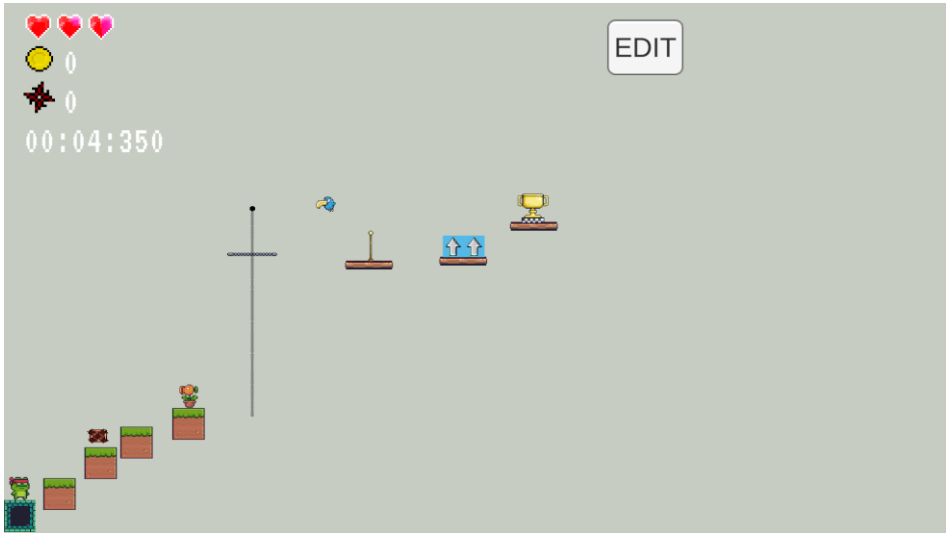


Figura 27. Interfaz de prueba para niveles en edición

Cuando el usuario considere que el nivel está acabado realizará el último paso dentro del proceso de creación de este, que es subirlo al servidor para poder compartirlo con el resto de los jugadores. Para ello debe pulsar el botón *"Upload"* dentro de la vista general de edición, lo que abrirá el menú mostrado en la Figura 28. En este menú tenemos un campo de texto para indicar el nombre del nivel, así como dos botones, uno para confirmar la subida del nivel y otro para cancelarla:

- Si el usuario pulsa el botón *"Upload"*, se subirá el nivel al servidor. Realmente lo que se sube es una captura de pantalla con el mapeado, el nombre del nivel, y un archivo de texto formateado en JSON que incluye la información de este. El nombre del nivel se forma concatenando el campo de texto *"Level Name"* con el nombre de usuario. Si por ejemplo el usuario se llama Manu y el nombre del nivel es nivel12, el nivel constará en el servidor como nivel12-Manu, por este sistema de nombrado es importante no cambiarse mucho el nombre pues sino al usuario le costará encontrar sus propios niveles ya que cuando se cambie el *username* los nombres de los niveles creados por él no se actualizarán.

- Si el usuario pulsa el botón *"Cancel"* se vuelve a la vista general de edición borrando todo el trabajo realizado, tal y como avisa el mensaje añadido. Por esto, es muy importante realizar la subida de nivel cuándo se tiene claro que este ya no necesita más cambios.

Figura 28. Menú de subida de niveles

Tras finalizar la explicación de la creación y subida de niveles hay que abordar la otra gran funcionalidad del editor de niveles: la descarga de estos. Si el usuario pulsa el botón *"Download"* en la vista general de edición, aparecerá ante él una lista de niveles creados por distintos jugadores (Figura 29), dichos niveles pueden descargarse y jugarse en local. Para cada nivel dentro de esta lista se muestra una imagen y el nombre para ayudar al usuario a decidir si quiere jugarlo o no. Cuando el usuario se decida por un nivel puede presionar *"Download"* y esto le abrirá el nivel personalizado en su equipo (Figura 30). Si por el contrario el usuario presiona la X de la parte superior derecha se volverá a la vista general de edición.

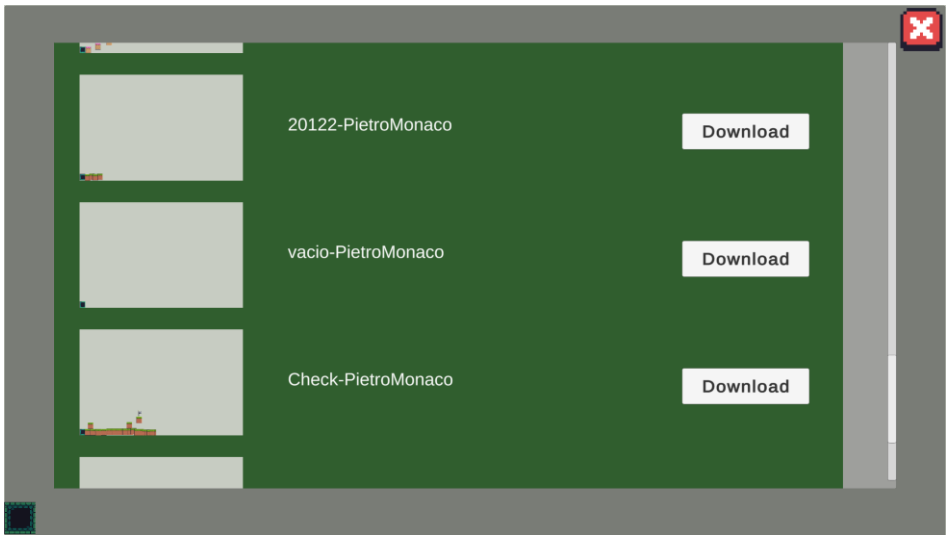


Figura 29. Menú de descarga de niveles

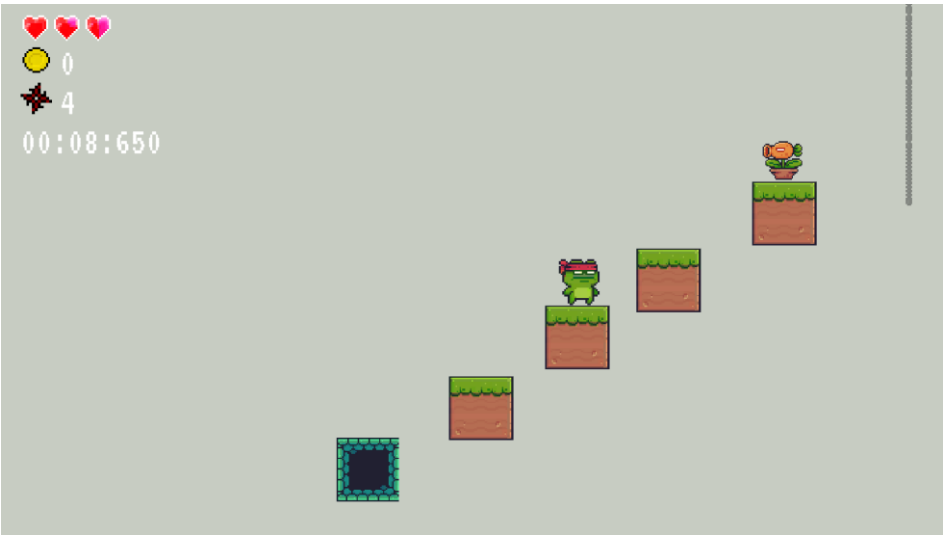


Figura 30. Pantalla de juego en un nivel descargado

El funcionamiento del nivel personalizado descargado es muy sencillo, ya que es muy parecido al de uno de los tres niveles básicos: la cámara se acerca y sigue al personaje, hay un *HUD* en el que se ignora la puntuación como se comentaba anteriormente, podemos mover el personaje libremente, si morimos sin tocar el *checkpoint* reapareceremos en el bloque inicial de abajo a la izquierda y además, si tocamos el trofeo se nos mostrará un menú que nos permitirá reintentar el nivel personalizado o volver al menú principal. Por último, cabe destacar que pulsando la tecla *Escape* se mostrará el mismo menú que en los niveles básicos, que permite reiniciar el nivel sin haberlo acabado o volver al menú principal.

5.2.1.5. Tutorial

Al seleccionar la última opción del menú principal (*"How to play"*), se le mostrará al usuario un pequeño tutorial (Figura 31) dividido en tres bloques, en el bloque superior se explican los controles y el movimiento del personaje, en la parte inferior izquierda se explica cuál es el objetivo del juego y, por último, en la parte inferior derecha se muestran los *powerups* que pueden servirle al usuario para cumplir el objetivo marcado por el juego de una manera más fácil.

Figura 31. Tutorial

5.2.2. Diseño e implementación de la capa de negocio

Los distintos *scripts* que forman el videojuego recogen toda la lógica del sistema. Estos están agrupados en 6 categorías según la función que realizan, tal y como se muestra en la Figura 32.

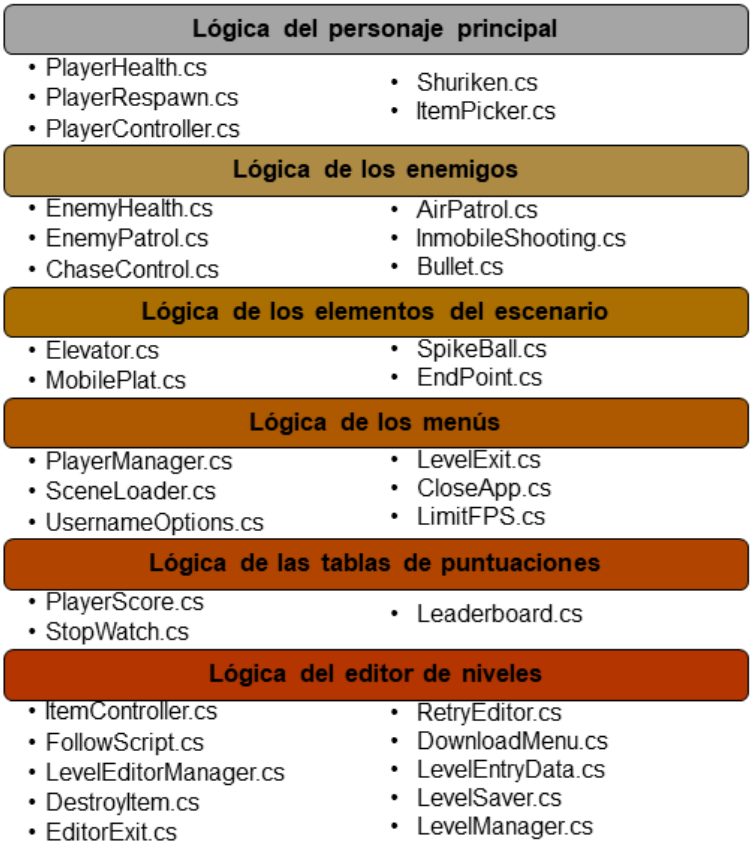


Figura 32. Distribución por categorías de los scripts del videojuego

A continuación, se realizará una explicación más detallada de cada uno de los tipos de *scripts* mencionados, así como se mostrará brevemente la función de cada archivo.

5.2.2.1. Lógica del personaje principal

Los archivos dentro de esta categoría son los encargados de todo lo relativo al personaje, desde el movimiento hasta la gestión de la salud pasando por las reapariciones y la interacción con el escenario.

***PlayerHealth.cs*:** Gestiona la salud del personaje, mediante dos funciones, una para restarle vida y otra para añadírsela. Además, estas actualizan la interfaz que muestra la salud.

```
//Funcion para restar salud al personaje
0 referencias
public void AddDamage (int amount)
{
    //Se le resta la cantidad indicada de salud al jugador,
    //no todos los enemigos hacen el mismo daño
    health = health - amount;

    //Importante llamar a la animacion de hit, con un trigger
    _animator.SetTrigger("Hit");

    if (health <=0)
    {
        //Se reaparece en el checkpoint o se muestra el menú de Game Over
        _respawn.Respawn();
        //Al reaparecer se reinicia la vida
        health = totalHealth;
    }

    heartUI.sizeDelta = new Vector2(heartSize * health, heartSize);//Se cambia el tamaño del hud
}

//Funcion para sumar salud al personaje
0 referencias
public void AddHealth(int amount)
{

    //Se le suma la cantidad indicada de salud al jugador
    health = health + amount;

    if (health > totalHealth) //Si ya se tiene la vida maxima, no se suma más
    {
        health = totalHealth;
    }

    heartUI.sizeDelta = new Vector2(heartSize * health, heartSize);//Se cambia el tamaño del hud
}
```

Figura 33. Métodos para restar y sumar salud al personaje dentro de *PlayerHealth.cs*

***PlayerRespawn.cs*:** Gestiona la reaparición del usuario, permitiendo que, según el desarrollo de la partida, reaparezca en los puntos de control o se le muestre el menú de *Game Over*.

***PlayerController.cs*:** Es el alma del movimiento del personaje y permite que el usuario controle a este mediante las diferentes teclas establecidas. El grueso de código se encuentra en la función *Update* [9], que se llama en cada fotograma y es la que se utiliza para detectar las entradas de teclado y gestionar los saltos y los ataques. Por otra parte, la función *FixedUpdate* [10] de Unity se utiliza para actualizar ciertas físicas del personaje como el movimiento horizontal o aéreo mientras que en *LateUpdate* [11], ejecutada tras las dos mencionadas anteriormente, se cambian las animaciones.

```
// A Update se le llama una vez por fotograma, se recogen las entradas de teclado en este método
@ Mensaje de Unity | 0 referencias
void Update()
{
    if (!_isAttacking) //Si se está atacando el personaje permanece quietp
    {
        //Se toman los valores del eje x, es decir se mira a ver si el usuario está tocando teclas
        float horizontalInput = Input.GetAxisRaw("Horizontal");
        _movement = new Vector2(horizontalInput, 0f);

        //Si esta tocando la tecla izquierda y el personaje mira a la derecha se gira el sprite para que no ande de espaldas
        if (horizontalInput < 0f && _facingRight == true)
        {
            Flip();
        }
        else if (horizontalInput > 0f && _facingRight == false)
        {
            Flip();
        }
    }

    //Se comprueba si el personaje toca el suelo
    _isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, groundLayer);
    //Se comprueba si el personaje toca la pared
    _touchingFront = Physics2D.OverlapCircle(frontCheck.position, groundCheckRadius, wallLayer);


    //Este bloque de código sirve para realizar el salto y el doble salto
    if (Input.GetButtonDown("Jump"))
    {
        //Si el usuario está tocando el suelo y toca la barra espaciadora, se le aplica la fuerza de salto y se activa el doble salto
        if (_isGrounded)
        {
            _rigidbody.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
            _canDoubleJump = true;
        }
        else if (!_isGrounded) //Cuando el usuario está en el aire se comprueba si puede realizar un doble salto
        {
            if (_canDoubleJump == true)
            {
                _animator.SetTrigger("DoubleJump"); //Lanzamos un trigger para la animacion de doble salto
                _canDoubleJump = false; //Se cambia el booleano, pues solamente se puede hacer 1 único salto estando ya en el aire
                _rigidbody.velocity = new Vector2(_rigidbody.velocity.x, 0); //Se hace para evitar que las fuerzas se sumen y el personaje acabe saltando demasiado alto
                _rigidbody.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
            }
        }
    }
```

Figura 34. Método *Update* de *PlayerController.cs*

```
//Se utiliza para el control de las físicas del personaje
Mensaje de Unity | 0 referencias
void FixedUpdate()
{
    //Si el personaje está en el suelo se le mueve horizontalmente en función de
    //las entradas de teclado recogidas en el Update
    if (_isGrounded)
    {
        float horizontalVelocity = _movement.normalized.x * Mathf.Abs(speed);
        _rigidbody.velocity = new Vector2(horizontalVelocity, _rigidbody.velocity.y);

    }

    //Si está en el aire se le añade cierta resistencia al movimiento para que sea más realista
    else if (!_isGrounded && !_isWallSliding && _movement.normalized.x != 0)
    {
        _rigidbody.AddForce(new Vector2(airMoveSpeed * _movement.normalized.x, 0));
        if (Mathf.Abs(_rigidbody.velocity.x) > speed)
        {
            float horizontalVelocity = _movement.normalized.x * Mathf.Abs(speed);
            _rigidbody.velocity = new Vector2(horizontalVelocity, _rigidbody.velocity.y);
        }
    }

}
```

Figura 35. Método *FixedUpdate* de *PlayerController.cs*

```
//Tras el Update y el FixedUpdate se ejecuta el LateUpdate, en el que se configuran las animaciones
Mensaje de Unity | 0 referencias
private void LateUpdate()
{
    //Se cambian los parametros del Animator para realizar correctamente el cambio entre animaciones
    _animator.SetBool("Idle", _movement == Vector2.zero);
    _animator.SetBool("IsGrounded", _isGrounded);
    _animator.SetBool("TouchingFront", _touchingFront);
    _animator.SetFloat("VerticalVelocity", _rigidbody.velocity.y);

    //Si se está haciendo la animacion de atacar significa que se está atacando,
    //para eso le ponemos un tag o etiqueta a la animacion
    if (_animator.GetCurrentAnimatorStateInfo(0).IsTag("Attack"))
    {
        _isAttacking = true;
    }
    else
    {
        _isAttacking = false;
    }
}
```

Figura 36. *LateUpdate* de *PlayerController.cs*

Shuriken.cs: Gestiona el comportamiento de los proyectiles lanzados por el personaje, trasladándolos por el aire y fijando tanto la cantidad de daño que harán a los enemigos como el tiempo que durarán activos.

ItemPicker.cs: Gestiona la interacción del personaje con los *powerups*, el punto de fin, los *checkpoints* y un tipo de plataformas que desaparecen al tocarlas. Este *script* se basa en las colisiones del usuario con el resto de los elementos dentro de cada nivel, por ejemplo, si el jugador toca una moneda esta desaparece, se suma a un contador y se actualiza el *HUD*, o, si toca una caja de armas esta desaparece otorgándole 4 ataques más y actualizándose el *HUD*.

5.2.2.2. Lógica de los enemigos

Los *scripts* dentro de esta categoría se encargan de fijar el comportamiento general de los enemigos, proporcionando patrones de movimiento y ataque diferentes según se trate de enemigos terrestres móviles, enemigos terrestres inmóviles o enemigos voladores.

EnemyHealth.cs: Gestiona la salud de los enemigos, puede personalizarse para que estos tengan más vida y así tarden más golpes en morir y desaparecer.

EnemyPatrol.cs: Permite a los enemigos terrestres móviles patrullar de izquierda a derecha entre dos muros y atacar al usuario cuerpo a cuerpo.

ChaseControl.cs: Fija una zona de detección del usuario que servirá para determinar el comportamiento de los enemigos voladores.

AirPatrol.cs: Gracias a este *script* los enemigos voladores perseguirán y atacarán cuerpo a cuerpo al usuario si entra en la zona de detección determinada en *ChaseControl.cs*. Sin embargo, cuando salga de dicha zona de detección los enemigos volverán a su punto de partida.

InmobileShooting.cs: Permite a los enemigos terrestres inmóviles disparar al usuario cada cierto tiempo.

Bullet.cs: Fija el comportamiento de las balas disparadas por los enemigos terrestres inmóviles.

5.2.2.3. Lógica de los elementos del escenario

Los *scripts* que se encuentran aquí agrupados son necesarios para gestionar el comportamiento de ciertos elementos del escenario.

Elevator.cs: Proporciona una rutina de movimiento para los ascensores, que se mueven arriba y abajo entre dos límites.

MobilePlat.cs: Permite a ciertas plataformas moverse de izquierda a derecha entre muros.

SpikeBall.cs: Determina el comportamiento de las bolas de pinchos, que se mueven trazando círculos alrededor de un punto fijo, causando daño al jugador si colisionan con él.

Endpoint.cs: Dictamina el comportamiento del trofeo de fin de nivel, al llegar a él aparece el menú de nivel completado.

5.2.2.4. Lógica de los menús

Para gestionar el inicio de sesión de los usuarios, así como el comportamiento de los menús no relacionados con el editor de niveles, se utilizan los *scripts* de esta categoría. Además, dentro de la misma también se encuentra código necesario para funcionalidades auxiliares del videojuego.

PlayerManager.cs: Este *script* permite que al abrir el juego se realice un inicio de sesión como invitado en LootLocker mediante el perfil del usuario, lo que es necesario para la gestión de las tablas de puntuaciones y para el editor de niveles. Si es la primera vez que se abre el juego, la llamada a *StartGuestSession* crea un perfil de usuario y posteriormente inicia sesión, las siguientes veces solamente se inicia sesión utilizando el perfil de usuario ya existente.

```
public class PlayerManager : MonoBehaviour
{

    // Nada más iniciar este Script se llama a la rutina de inicio de sesión
    Mensaje de Unity | 0 referencias
    void Awake()
    {
        StartCoroutine(LoginRoutine());
    }

    //Se realiza un inicio de sesión como invitado en Lootlocker
    1 referencia
    IEnumerator LoginRoutine()
    {
        bool done = false;

        //Se utiliza un método de la Sdk para intentar el inicio de sesión
        LootLockerSDKManager.StartGuestSession((response) =>
        {
            if (response.success) //Si el inicio de sesión es correcto se guarda en local el PlayerId
            {
                Debug.Log("Log in succesful");
                PlayerPrefs.SetString("PlayerID", response.player_id.ToString());
                done = true;
            }
            else //Si hay un fallo se muestra un mensaje en consola
            {
                Debug.Log("Log in unsuccesful");
                done = true;
            }
        });
        yield return new WaitWhile(() => done == false);
    }

}
```

Figura 37. Código para el inicio de sesión dentro de *PlayerManager.cs*

SceneLoader.cs: Se utiliza para cambiar entre las diferentes pantallas del videojuego.

UsernameOptions.cs: Se utiliza para gestionar el menú de opciones de nombre de usuario, cuenta con dos funciones, una que permite cambiar el nombre de usuario y otra que muestra el nombre actual.

LevelExit.cs*:* Permite mostrar el menú que se abre con *Escape* dentro de los niveles básicos.

CloseApp.cs: Este script permite que pulsando los botones con el texto *"Quit"* o con una X se cierre el videojuego y se detenga su ejecución.

LimitFps.cs: Limita los fotogramas por segundo del videojuego a 60.

5.2.2.5. Lógica de las tablas de puntuaciones

Se utilizan estos *scripts* para calcular y mostrar la puntuación del usuario durante el transcurso y finalización de un nivel, así como para gestionar la manipulación de las tablas de puntuaciones que residen en LootLocker.

PlayerScore.cs: Calcula la puntuación del usuario utilizando la fórmula que aparece en la Figura 38. Además, el script se encarga de mostrar tanto la puntuación durante la partida como la puntuación final que se consigue al completar el nivel.

$$\text{Puntuación} = \frac{(NúmeroDeMonedasRecogidas * 200) + (NúmeroDeEnemigosEliminados * 200) + BonusPorFinalizaciónDeNivel}{\frac{TiempoDeFinalización}{20}}$$

1. El bonus de finalización varía según el nivel. Nivel 1: 2000, Nivel 2: 4000, Nivel 3: 6000
2. El tiempo de finalización entre 20 se redondea al entero más cercano.

Figura 38. Fórmula para el cálculo de la puntuación

StopWatch.cs: Permite calcular el tiempo que el usuario tarda en pasarse un nivel, tanto básico como personalizado. Además, también se utiliza para mostrar en pantalla dicho dato.

LeaderBoard.cs: Este script se utiliza para subir las puntuaciones de los usuarios a las tablas de puntuaciones, y también para descargar los datos de estas y mostrarlos.

```
1 referencia
public IEnumerator SubmitScoreRoutine(int score)
{
    bool done = false;
    string playerID = PlayerPrefs.GetString("PlayerID");
    //Con el PlayerID guardado en local se sube la puntuación del usuario
    //a la tabla de puntuaciones identificada por leaderboardID
    LootLockerSDKManager.SubmitScore(playerID, score, leaderboardID, (response) =>
        {
            if (response.success) //Si la salida es positiva la puntuación queda guardada
            {
                Debug.Log("Successfully uploaded score");
                done = true;
            } else //Si hay un fallo se muestra mediante la consola
            {
                Debug.Log("Failed" + response.Error);
                done = true;
            }
        });
    yield return new WaitWhile(() => done == false);
}

0 referencias
public void UploadScore()
{
    string[] tArray = scoreText.text.Split(':');
    //Se llama a la rutina para subir la puntuación al servidor
    StartCoroutine(SubmitScoreRoutine(int.Parse(tArray[1])));
}
```

Figura 39. Rutina para subir la puntuación de un usuario, dentro de *LeaderBoard.cs*

```
public IEnumerator FetchTopHighScoresRoutine()
{
    yield return new WaitForSeconds(2.0f);
    bool done = false;
    //Se utiliza un método proporcionado por Lootlocker para tomar las puntuaciones
    //de una de las tablas mediante un identificador llamado LeaderboardID
    LootLockerSDKManager.GetScoreList(leaderboardID, 10, 0, (response) =>
    {
        if (response.success) //Si la salida del método es positiva se muestra la tabla de puntuaciones
        {

            string tempPlayerNames = "Names\n";
            string tempPlayerScores = "Scores\n";

            //Se encapsulan en un array los usuarios que aparecen en la Leaderboard
            LootLockerLeaderboardMember[] members = response.items;

            //Se recorren dichos usuarios, construyendo línea a línea la tabla de puntuaciones
            for (int i = 0; i < members.Length;i++)
            {
                tempPlayerNames += members[i].rank + ". ";
                if (members[i].player.name != "")
                {
                    tempPlayerNames += members[i].player.name;
                } else
                {
                    tempPlayerNames += members[i].player.id;
                }
                tempPlayerScores += members[i].score + "\n";
                tempPlayerNames += "\n";
            }
            done = true;
            playerNames.text = tempPlayerNames;
            playerScores.text = tempPlayerScores;
            //Gracias al bloque de código anterior se muestra la LeaderBoard de la siguiente manera
            // Names                        Scores
            // 1.PlayerID o NombreUsuario    100
            // 2.PlayerID o NombreUsuario    50
        }
        else //Si la salida es negativa se muestra el error por consola
        {
            Debug.Log("Failed" + response.Error);
            done = true;
        }
    });
    yield return new WaitWhile(() => done == false);
}

0 referencias
public void LoadScore()
{
    StartCoroutine(FetchTopHighScoresRoutine()); //Se llama a la rutina para cargar la tabla de puntuaciones
}
```

Figura 40. Rutina para descargar y mostrar las puntuaciones, dentro de *LeaderBoard.cs*

5.2.2.6. Lógica del editor de niveles

Para que el editor de niveles funcione tal y como se ha mostrado es necesario tener una serie de *scripts* que fijen su comportamiento.

ItemController.cs: Cuando se escoge un objeto del *scroll* de la vista general de edición, este *script* instancia una guía visual para que el usuario vea donde va a colocar el objeto.

FollowScript.cs: La guía visual que instancia *ItemController.cs* se mueve siguiendo el cursor del ratón gracias a este *script*.

LevelEditorManager.cs: Permite colocar objetos para conformar el nivel, los objetos se colocan con *click* izquierdo y se deja de colocar un tipo de objeto pulsando la tecla *C*. Al colocar el objeto se destruye la guía visual.

DestroyItem.cs: Permite borrar los objetos colocados en la vista general de edición pulsando *click* derecho.

EditorExit.cs: Permite mostrar el menú que se abre con la tecla *Escape* en el editor y en los niveles descargados.

RetryEditor.cs: Contiene una función auxiliar necesaria para poder reintentar y recargar el nivel personalizado descargado.

LevelSaver.cs, LevelManager.cs, DownloadMenu.cs y LevelEntryData: Estos *scripts* trabajan de manera conjunta para gestionar la subida y descarga de niveles personalizados.

- ***Subida de niveles:*** Cuando el usuario entra al menú de subida de niveles, se realiza una captura de pantalla del mapeado del nivel y se almacena en local mediante el método *TakeScreenshot()* de *LevelManager.cs*, dentro de este método hay una llamada a *FindSavableAssets()* de *LevelSaver.cs*, que recoge la información de los objetos dispuestos en el mapeado personalizado, la cual posteriormente se formatea a JSON y se almacena en un fichero gracias al método *SaveToFile(),* dicho fichero se encuentra en la misma ubicación que la captura anteriormente realizada. Tras esto, cuando el usuario le ha puesto un nombre al nivel y ha pulsado el botón para subirlo al servidor, se llama al método *CreateLevel()* de *LevelManager.cs*, que crea la estructura para almacenar la información del nivel en LootLocker, y luego llama a *UploadLevelData(int LevelID),* función que, mediante el *LevelId* proporcionado por el servidor al crear la estructura mencionada, encuentra esta y almacena dentro el nombre del nivel, la captura de pantalla del mapeado y el fichero JSON que lo representa.

```
//Cuando subas un nivel al servidor hay que hacerle una captura para la miniatura
1 referencia
public void TakeScreenshot()
{
    //Se crea el path en el que almacenar la captura
    string folder = Path.Combine(Application.persistentDataPath, "/LevelData/");
    //Si la carpeta todavía no existe se creará
    if (!Directory.Exists(folder))
        Directory.CreateDirectory(folder);
    //Se toma la captura de pantalla
    ScreenCapture.CaptureScreenshot(Path.Combine(folder, "Level-Screenshot.png"));
    //Se llama a LevelSaver para que encuentre los objetos a guardar y los escriba en un fichero
    GetComponent<LevelSaver>().FindSavableAssets();
}
```

Figura 41. Método *TakeScreenshot()* de *LevelManager.cs*

```
//Encuentra los objetos con el tag "savable", es decir, aquellos que conforman
//el mapeado del nivel personalizado
1 referencia
public void FindSavableAssets()
{
    //Se limpia el array utilizado para recoger la información de los objetos
    if (assetsToSave != null) { for (int i = 0; i < assetsToSave.Length; i++)
        { Destroy(assetsToSave[i]); } }
    //Se encierran en un array los objetos  que forman el mapeado del nivel creado
    assetsToSave = GameObject.FindGameObjectsWithTag(savableAssetTag);

    //Se recorre el array de objetos a guardar,
    //separando sus nombres y posiciones en dos estructuras diferentes
    assetNames = new string[assetsToSave.Length];
    assetPositions = new Vector3[assetsToSave.Length];
    for (int j = 0; j < assetsToSave.Length; j++)
    {
        assetNames[j] = assetsToSave[j].name;
        assetPositions[j] = assetsToSave[j].transform.position;
    }
    //Se escriben los datos del mapeado del nivel en un fichero
    SaveToFile();
}
```

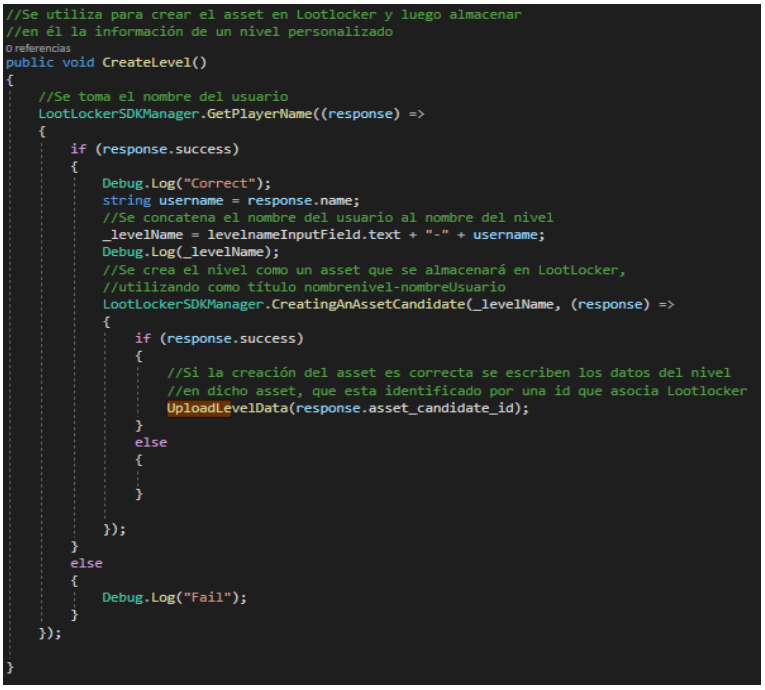
Figura 42. Método *FindSavableAssets()* de *LevelSaver.cs*

```
//Función que se utiliza para escribir en un fichero
//la información de los objetos que conforman el nivel
1 referencia
public void SaveToFile()
{
    //Se crea el path en el que se encuentra la carpeta
    string folder = Path.Combine(Application.persistentDataPath,"/LevelData/");
    //Si dicha carpeta no existe se creará
    if (!Directory.Exists(folder))
        Directory.CreateDirectory(folder);

    //Se indica el path en el que hay que escribir
    string filepath = Path.Combine(folder, "Level-Data.json");
    //Se crea la estructura necesaria para la escritura
    StreamWriter writer = new StreamWriter(filepath, false);

    //Se recorren los objetos guardados
    //anteriormente en FindSavableAssets
    for (int i = 0; i < assetsToSave.Length; i++)
    {
        //La información de cada objeto se formatea a Json gracias a la clase auxiliar
        //EditedObject y posteriormente se escribe en el fichero
        EditedObject obj = new EditedObject(assetNames[i], assetPositions[i].x.ToString(),
            assetPositions[i].y.ToString());
        writer.WriteLine(JsonUtility.ToJson(obj));
    }
    writer.Close();
}
```

Figura 43. Método *SaveToFile()* de *LevelSaver.cs*



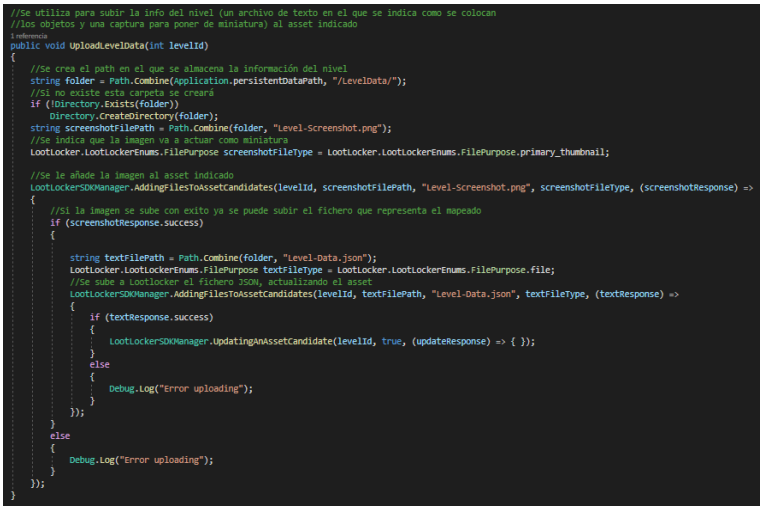
```
//Se utiliza para crear el asset en Lootlocker y luego almacenar
//en él la información de un nivel personalizado
0 referencias
public void CreateLevel()
{
    //Se toma el nombre del usuario
    LootLockerSDKManager.GetPlayerName((response) =>
    {
        if (response.success)
        {
            Debug.Log("Correct");
            string username = response.name;
            //Se concatena el nombre del usuario al nombre del nivel
            _levelName = levelnameInputField.text + "-" + username;
            Debug.Log(_levelName);
            //Se crea el nivel como un asset que se almacenará en LootLocker,
            //utilizando como título nombrenivel-nombreUsuario
            LootLockerSDKManager.CreatingAnAssetCandidate(_levelName, (response) =>
            {
                if (response.success)
                {
                    //Si la creación del asset es correcta se escriben los datos del nivel
                    //en dicho asset, que esta identificado por una id que asocia Lootlocker
                    UploadLevelData(response.asset_candidate_id);
                }
                else
                {

                }

            });
        }
        else
        {
            Debug.Log("Fail");
        }
    });

}
```

Figura 44. Método *CreateLevel()* de *LevelManager.cs*



```
//Se utiliza para subir la info del nivel (un archivo de texto en el que se indica como se colocan
//los objetos y una captura para poner de miniatura) al asset indicado
1 referencia
public void UploadLevelData(int levelId)
{
    //Se crea el path en el que se almacena la información del nivel
    string folder = Path.Combine(Application.persistentDataPath, "/LevelData/");
    //Si no existe esta carpeta se creará
    if (!Directory.Exists(folder))
        Directory.CreateDirectory(folder);
    string screenshotFilePath = Path.Combine(folder, "Level-Screenshot.png");
    //Se indica que la imagen va a actuar como miniatura
    LootLocker.LootLockerEnums.FilePurpose screenshotFileType = LootLocker.LootLockerEnums.FilePurpose.primary_thumbnail;

    //Se le añade la imagen al asset indicado
    LootLockerSDKManager.AddingFilesToAssetCandidates(levelId, screenshotFilePath, "Level-Screenshot.png", screenshotFileType, (screenshotResponse) =>
    {
        //Si la imagen se sube con exito ya se puede subir el fichero que representa el mapeado
        if (screenshotResponse.success)
        {

            string textFilePath = Path.Combine(folder, "Level-Data.json");
            LootLocker.LootLockerEnums.FilePurpose textFileType = LootLocker.LootLockerEnums.FilePurpose.file;
            //Se sube a Lootlocker el fichero JSON, actualizando el asset
            LootLockerSDKManager.AddingFilesToAssetCandidates(levelId, textFilePath, "Level-Data.json", textFileType, (textResponse) =>
            {
                if (textResponse.success)
                {
                    LootLockerSDKManager.UpdatingAnAssetCandidate(levelId, true, (updateResponse) => { });
                }
                else
                {
                    Debug.Log("Error uploading");
                }
            });
        }
        else
        {
            Debug.Log("Error uploading");
        }
    });
}
```

Figura 45. Metodo *UploadLevelData (int levelId)* de *LevelManager.cs*

- ***Descarga de niveles:*** Cuando el usuario abre el menú de descarga de niveles, se activa el método *OnEnable()* [12] que se encuentra dentro de *DownloadMenu.cs*, este método realiza una llamada a *DownloadLevelData()* de *LevelManager.cs*, que sirve para descargar la información de todos los niveles personalizados almacenados en LootLocker y crear una entrada en el menú para cada uno de ellos. Tras tener el menú actualizado, si el usuario presiona el botón para descargar un nivel personalizado se hace una llamada a la función *LoadLevel()*, dentro de *LevelEntryData.cs*, que es el *script* que está asociado a las distintas entradas del menú. La función mencionada se encarga de descargar el fichero JSON que representa el mapeado del nivel seleccionado, almacenarlo en local y luego llamar a la función *LoadLevel()* de *LevelSaver.cs*, que recorre el archivo JSON recogiendo la información de los objetos, para finalmente llamar a *CreateAssets()*, que instancia estos para crear el nivel personalizado, además, esta función también activa el personaje, el *HUD* y cambia la cámara para permitir jugar al nivel recién descargado.

```
//Se recorren los datos del nivel descargados recogiendo los nombres y posiciones de los objetos
//a crear para reproducir el nivel
2 referencias
public void LoadLevel()
{
    //Se limpia la escena antes de cargar el nivel
    foreach(GameObject savableObject in GameObject.FindGameObjectsWithTag("Savable"))
    {
        Destroy(savableObject);
    }

    //Se crea el path en el que hay que buscar el fichero del nivel
    string folder = Path.Combine(Application.persistentDataPath,"Level-Data.json");

    //Se crea una estructura para contar las lineas del fichero
    StreamReader reader = new StreamReader(folder);
    int numOfLines = 0;
    //Se inicializan los arrays de lectura, cada linea del fichero
    //representa un objeto diferente
    while (reader.ReadLine() != null) { numOfLines++; }
    Debug.Log("NumofLines " + numOfLines);
    assetNames = new string[0];
    assetPositions = new Vector3[0];

    assetNames = new string[numOfLines];
    assetPositions = new Vector3[numOfLines];

    reader.Close();

    //Con este segundo StreamReader se lee el fichero y se
    //guarda el nombre y la posición de los objetos en varios arrays
    StreamReader reader2 = new StreamReader(folder);
    while (!reader2.EndOfStream)
    {
        for (int i = 0; i < numOfLines; i++)
        {
            string line = reader2.ReadLine();
            Debug.Log(line);
            EditedObject obj = JsonUtility.FromJson<EditedObject>(line);
            assetNames[i] = obj.name;
            assetPositions[i].x = float.Parse(obj.x);
            assetPositions[i].y = float.Parse(obj.y);
            assetPositions[i].z = 0.0f;
        }
    }
    reader2.Close();
    //Con la información de los objetos ya recogida se instancian estos
    CreateAssets();
}
```

Figura 46. Método *LoadLevel()* de *LevelSaver.cs*

```
//Se recorren los objetos "Savable" recolectados en LoadLevel y se instancian.
1 referencia
public void CreateAssets()
{
    //Se recorren los nombres de los objetos recolectados en LoadLevel,
    //se comparan para ver que objeto representan en el juego y luego se instancian
    //según las posiciones que marcaba el fichero
    for (int i = 0; i<assetNames.Length;i++)
    {
        for (int j= 0; j < possibleObjects.Length;j++)
        {
            if(possibleObjects[j].name == assetNames[i])
            {
                Debug.Log("Nombre del asset: " + possibleObjects[j].name + " Nombre dw: " + assetNames[i]);
                GameObject obj = Instantiate(possibleObjects[j], assetPositions[i], Quaternion.identity);
            }
        }
    }
    //Se activa el player tras construir el nivel
    player.SetActive(true);
    //Se cambia la cámara
    editingCamera.SetActive(false);
    playingCamera.SetActive(true);
    //Se activa el HUD
    hud.SetActive(true);
    //Se desactiva la interfaz del editor
    menuEditor.SetActive(false);
    tooltip1.SetActive(false);
    tooltip2.SetActive(false);
}
```

Figura 47. Método *CreateAssets()* de *LevelSaver.cs*

5.2.3. Diseño e implementación de la capa de datos

Esta capa hace uso del propio equipo del usuario y de las bases de datos gestionadas por LootLocker para almacenar los perfiles de usuario, el estado de la partida mientras se está jugando, las tablas de puntuaciones, y la información relativa a los niveles personalizados. A continuación, se explica con más detalle la gestión de cada tipo de dato.

5.2.3.1. Perfiles de usuario

Cada usuario cuenta con un perfil que se almacena en LootLocker, este identifica al jugador de manera univoca mediante un entero asignado por la plataforma llamado *PlayerID*, además contiene también el nombre que el usuario ha seleccionado. Cabe destacar que el *PlayerID* se utiliza para la gestión de las tablas de puntuaciones, por lo que se almacena en local utilizando la clase *PlayerPrefs* de Unity [13], que permite guardar enteros, números decimales o texto que se mantienen entre sesiones.

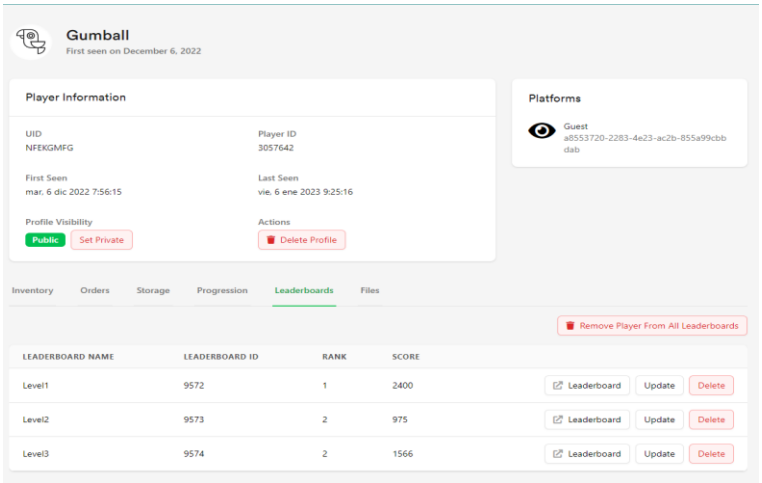


Figura 48. Perfil del usuario en LootLocker

5.2.3.2. Estado de la partida

Cuando el usuario está en un nivel el propio juego almacena el número de monedas que ha recolectado, los corazones que tiene, las armas que le quedan y el tiempo transcurrido en el nivel, así como la puntuación conseguida. Estos datos se utilizan para poder proporcionar al jugador información en tiempo real.

5.2.3.3. Tablas de puntuaciones

El juego cuenta con 3 tablas de puntuaciones que se han creado gracias a la plataforma LootLocker y se almacenan en las bases de datos de esta. Para acceder a ellas mediante código se utilizan métodos del *Software Development Kit* de LootLocker, que a su vez utilizan el *Player ID* ya mencionado y un identificador único asignado a cada clasificación llamado *LeaderboardID.*

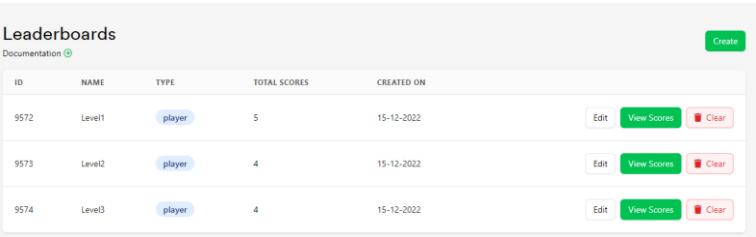


Figura 49. Tablas de puntuaciones en LootLocker

5.2.3.4. Niveles

Los niveles personalizados se almacenan en LootLocker, utilizando estructuras que contienen el nombre del nivel, una captura de pantalla del mapeado y un fichero en formato JSON llamado *Level-Data* (Figura 50), que contiene la información de los objetos colocados (nombre y posición en eje X e Y). Sin embargo, se utiliza un directorio de datos persistentes al que se accede mediante la directiva de Unity *Application.persistentDataPath* [14] para almacenar el nivel de manera temporal durante el proceso de compartirlo o de descargarlo para jugar. Para manipular los datos de los niveles y acceder a ellos se utilizan métodos del *Software Development Kit* de LootLocker, al igual que para la gestión las tablas de puntuaciones.


```
{"name":"InitialBlock","x":"-0,6220026","y":"-1,236113"}
{"name":"Dirt","x":"0,2342353","y":"-0,7733419"}
{"name":"Dirt","x":"0,9601612","y":"-0,2400086"}
{"name":"Dirt","x":"1,64905","y":"0,189621"}
{"name":"Dirt","x":"2,523124","y":"0,6933248"}
{"name":"AttackBuff","x":"0,9453464","y":"0,1525841"}
{"name":"InmobilePlant","x":"2,50831","y":"1,093325"}
{"name":"Elevator","x":"3,456457","y":"1,893325"}
{"name":"BlueBirdGroup","x":"6,360161","y":"2,84888"}
{"name":"WoodenPlat","x":"4,774976","y":"2,804436"}
{"name":"WoodenPlat","x":"5,915717","y":"2,819251"}
{"name":"WoodenPlat","x":"6,960161","y":"3,122954"}
{"name":"CheckPoint","x":"4,789791","y":"3,189621"}
{"name":"Endpoint","x":"6,974976","y":"3,50814"}
{"name":"BouncyBlock","x":"5,915717","y":"3,048881"}
```


Figura 50. Fichero JSON que representa el mapeado de un nivel

6. Pruebas

En este apartado se detallan las pruebas realizadas para comprobar el correcto funcionamiento del videojuego durante su desarrollo, así como aquellas comprobaciones realizadas para verificar que el producto final es tal y como se esperaba.

6.1. Pruebas Unitarias

Las pruebas unitarias son aquellas que sirven para comprobar que un fragmento aislado del código funciona correctamente. Este tipo de pruebas son complicadas de realizar durante el desarrollo de un videojuego, ya que hay muchos métodos que dependen de otros, por lo que el correcto funcionamiento de la mayoría de las funciones se ha verificado mediante mensajes en consola. Sin embargo, gracias a la herramienta *Test Runner* de Unity [15] se han podido realizar algunas pruebas unitarias de vital importancia, por ejemplo, se ha verificado que las funciones para añadir y restar vida al personaje funcionaban correctamente.

```
[Test]
//Comprueba que al sumar la salud la salida es correcta
0 referencias
public void plusHealth()
{
    //Preparación
    GameObject go = new GameObject();
    go.AddComponent<PlayerHealth>();
    //Inicializamos a 2 la salud
    go.GetComponent<PlayerHealth>().health = 2;
    //Test, se simula que se ha ganado 1 de vida
    go.GetComponent<PlayerHealth>().AddHealth(1);
    //Comprobación, se verifica que la salud se ha sumado
    Assert.AreEqual(3, go.GetComponent<PlayerHealth>().health);
}
```

```
[Test]
//Comprueba que al restar la salud la salida es correcta
0 referencias
public void minusHealth()
{
    //Preparación
    GameObject go = new GameObject();
    go.AddComponent<PlayerHealth>();
    //Inicializamos a 3 la salud
    go.GetComponent<PlayerHealth>().health = 3;
    //Test, se simula un golpe, restando 1 a la vida del personaje
    go.GetComponent<PlayerHealth>().AddDamage(1);
    //Comprobación, se verifica que la salud ha bajado  1 punto
    Assert.AreEqual(2,go.GetComponent<PlayerHealth>().health);
}
```

Figura 51. *Tests* unitarios de añadir y restar salud

6.2. Pruebas de integración

Tras realizar las pruebas unitarias, que verifican que los distintos componentes funcionan correctamente de manera aislada, es necesario comprobar el funcionamiento conjunto de estos, así como la correcta interacción entre ellos, para lo que se realizan las pruebas de integración.

En el caso concreto de este proyecto, este tipo de pruebas se han realizado de manera incremental, comprobando que las funcionalidades recién añadidas realizaban correctamente su trabajo, sin lastrar el resto de procesos que ya se habían verificado anteriormente. Para esto, al final de cada iteración se ha utilizado el *Play Mode* de Unity [16], corriendo el juego dentro del propio programa para comprobar las características añadidas y verificar que lo ya desarrollado seguía funcionando sin problemas.

Figura 52. Esquema de las pruebas realizadas

6.3. Pruebas de sistema

Este tipo de pruebas se utilizan para comprobar el correcto funcionamiento general del sistema, siendo especialmente útiles a la hora de verificar el cumplimiento de los diversos requisitos no funcionales establecidos inicialmente.

6.3.1. Pruebas de portabilidad

Solamente se ha recogido un requisito de este tipo, que dictaba que el juego tenía que poder ejecutarse en sistemas Windows.

Se ha probado el *software* mediante un ejecutable en dos dispositivos diferentes, un ordenador portátil modelo Asus X553M y un equipo de sobremesa, con Windows 8.1 y 10 respectivamente. El videojuego no presentó fallos en ninguna de las dos máquinas por lo que se consideró que el requisito mencionado se cumplía.

6.3.2. Pruebas de usabilidad y accesibilidad

El principal requisito de usabilidad que debía de cumplir el *software* desarrollado era adherirse a una estética familiar y agradable apta para todos los públicos. Para cumplir con este requisito, todo el apartado visual del juego se ha escogido para que fuese parecido a dibujos animados, predominando los tonos pastel y evitando la violencia explícita. Como comprobación, he permitido que mi prima, de 7 años, probase el juego bajo la supervisión de su madre, la más joven de las dos ha quedado satisfecha con la cantidad de colores y la variedad de diseños mientras que la mayor ha considerado que el juego era apto para una niña, con lo que tras esta prueba se consideró que el juego es definitivamente apto para todos los públicos.

Para hacer el juego accesible a un amplio sector de la población se fijó que este tenía que estar completamente en inglés. Para comprobar que el videojuego estaba en perfecto inglés sin faltas de ortografía ni errores gramaticales lo he revisado cuidadosamente apoyándome en recursos *online*, ya que cuento con una certificación de nivel intermedio-alto en esta lengua, tras arreglar varios errores menores el requisito quedó cumplido.

Por último, el juego tenía que ofrecer un conjunto de interfaces sencillas, intuitivas y con letra grande para mejorar su compresión. Cumplir este requisito podía hacer que el juego fuese más sencillo de usar, así como permitir que gente con discapacidad visual pudiera disfrutar de él, por lo que se consideró un requisito tanto de usabilidad como de accesibilidad. Para que este requerimiento quedase cubierto se han desarrollado unos menús con pocas opciones, representadas con una tipografía fácil de leer y letra de un tamaño considerable. Además, las transiciones entre los menús son rápidas y los tiempos de carga muy cortos.

6.3.3. Pruebas de rendimiento

Los requisitos de rendimiento son muy importantes en los videojuegos, en este caso particular, se fijaron 3 requisitos de este tipo: que la tasa de fotogramas fuese igual o superior a 60 fps, que el peso total del ejecutable no excediese los 300 MB y que el juego funcionase de manera estable en distintos dispositivos.

Para comprobar que la tasa de fotogramas era de 60 fps o superior, se ha utilizado la herramienta *Profiler* de Unity [17], que permite obtener información sobre el rendimiento de la aplicación tanto desde un ejecutable como desde el propio editor. Al utilizar el *Profiler* por primera vez se observó que el juego superaba con creces los 60 fps, pero esto hacía que el rendimiento general fuese errático en ocasiones, por lo que se decidió limitar la tasa de fotogramas a 60 fps mediante un *script*, llegando a obtener las métricas de la Figura 53.

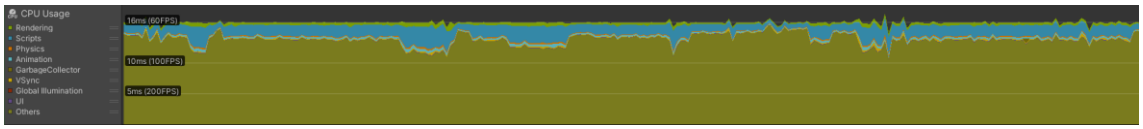


Figura 53. Tasa de fotogramas del videojuego

Para verificar que el juego era ligero y el ejecutable no excedía los 300 MB bastó con entrar en las propiedades de la carpeta en la que se almacena este, como se puede observar en la Figura 54 el ejecutable apenas llegaba a los 100 MB, lo que demuestra que el requisito quedó cumplido.

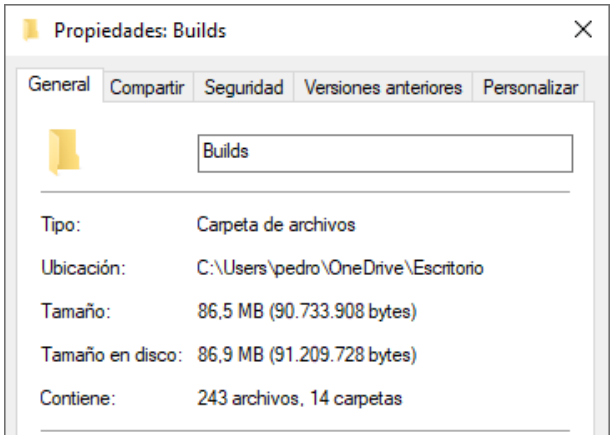


Figura 54. Peso del ejecutable en MB

Por último, para confirmar el correcto funcionamiento del *software* en máquinas distintas, se ha ejecutado el juego en dos dispositivos de características diferentes (los mismos utilizados para las pruebas de portabilidad), un portátil Asus X553M con 4GB de RAM, procesador Intel Celeron N2840 de 2 núcleos y 500 GB de disco duro HDD y un ordenador de sobremesa con 16 GB de RAM, procesador Intel I5 de 4 núcleos y 500GB de disco duro SSD. Mientras el juego estaba en ejecución se ha podido comprobar con el administrador de tareas de Windows que funcionaba de manera estable en ambos equipos, sin consumos altos de RAM ni fallos graves, por lo que el requisito quedó cumplido.

6.4. Pruebas de aceptación

Las pruebas de aceptación son las últimas en ser realizadas, en estas el usuario utiliza el *software* y verifica que cumple con sus expectativas. En mi caso particular, para realizar estas pruebas he compartido el juego con amigos y familiares, para que lo probasen y así conseguir opiniones de gente en distintas franjas de edad, desde niños hasta adultos de mediana edad.

En general, tras varios días de juego, la mayoría de los usuarios consideraron que el juego era divertido, intuitivo y correcto visualmente, aunque mencionaron que era ligeramente complicado aprender a usar el editor y algunos sugirieron una traducción al castellano. En conclusión, valorando la experiencia general de los jugadores, se consideró que el producto estaba aceptado y el desarrollo finalizado a pesar de estas objeciones menores.

7. Conclusiones y trabajo futuro

En este último apartado se incluyen las conclusiones obtenidas tras la finalización del desarrollo, así como un análisis en el que se detalla si se han cumplido los objetivos marcados inicialmente. Por otra parte, también se proporciona una breve explicación de cómo será el videojuego en un hipotético futuro.

7.1. Conclusiones

Antes de analizar el grado de cumplimiento de los objetivos marcados, me gustaría mencionar que estoy muy contento con el resultado obtenido, ya que he creado un videojuego completamente funcional, fácil de entender y divertido. Además, la realización de este me ha hecho comprender la importancia del diseño tanto arquitectónico como visual, ya que al principio consideraba estas partes relativamente sencillas y realmente ha sido bastante duro lidiar con ellas. Por último, gracias a este proyecto he podido obtener bastantes conocimientos acerca del manejo de Unity y de la programación en C# utilizando Visual Studio, lo que considero que puede ser útil para mi carrera profesional en un futuro.

El objetivo del proyecto era enfrentarse a un desarrollo *indie* consiguiendo crear un videojuego 2D basado en niveles, teniendo como metas secundarias proporcionar una faceta competitiva mediante *rankings online* y permitir la interacción entre jugadores mediante un editor de niveles.

El objetivo general de desarrollar un videojuego 2D basado en niveles considero que se ha cumplido de manera satisfactoria, ya que se ha logrado obtener un *software* bastante correcto e incluso disfrutable que permite al usuario jugar 3 niveles diferentes de dificultad ascendente, duración distinta y con mapeados variados.

El propósito de proporcionar una faceta competitiva mediante las tablas de puntuaciones ha sido un éxito y tiene un funcionamiento incluso mejor del esperado, pues se ha conseguido una gestión sencilla tanto para el usuario como para el desarrollador gracias a la integración y buen funcionamiento conjunto de la plataforma LootLocker con el motor Unity.

Por último, proporcionar una interacción entre jugadores mediante el editor de niveles es la meta que se ha completado en menor medida, ya que, aunque el editor cumple con su función, podría ser considerablemente más profesional con unas ligeras mejoras. Esta parte ha sido la más complicada durante el desarrollo del proyecto, ya que tuve que realizar una extensa investigación inicial y después fue bastante complicado conseguir formatear los niveles como archivos JSON para compartirlos y descargarlos mediante LootLocker.

7.2. Trabajo futuro

Gracias a este proyecto he descubierto que elaborar videojuegos es algo que quiero seguir haciendo en el futuro, abordando el desarrollo de juegos de otros géneros y plataformas, pero antes de empezar la creación de un nuevo *software* me gustaría darle algunos retoques a *Frog Quest*.

En primer lugar, me interesaría formarme en diseño con píxeles para poder proporcionarle al videojuego un apartado visual más pulido, añadiendo fondos a los niveles y a los menús y mejorando ciertas animaciones que fueron creadas de manera rudimentaria.

Para hacer el juego más disfrutable e incluso mejorar la jugabilidad me gustaría añadir al videojuego un sistema de sonidos personalizados, así como una pequeña banda sonora, que podría crear de manera sencilla mediante el uso de la inteligencia artificial Mubert [18].

Como ya mencioné en el apartado anterior, el editor de niveles es correcto, pero podría mejorarse. En un futuro, sería interesante modificar el menú de descarga de niveles, añadiendo un buscador que permitiera filtrar por nombre de nivel o nombre de creador. Además, mediante el paquete *Cinemachine* podría hacerse una cámara que siguiese al cursor permitiendo así la creación de niveles más grandes y espectaculares, sin la limitación de espacio que presenta hoy en día la vista general de edición.

Tras realizar todas las mejoras mencionadas anteriormente, me gustaría publicar el proyecto en la plataforma de venta de videojuegos virtuales Steam. Este movimiento iría acompañado con un cambio en el *login*, los usuarios ya no iniciarían sesión como invitado, sino que lo harían utilizando su cuenta en la propia plataforma lo que permitiría proporcionarles estadísticas detalladas acerca del tiempo que han jugado o incluso crear un sistema de logros integrado con Steam. Cabe destacar que este cambio en el inicio de sesión sería posible de realizar de manera relativamente sencilla gracias a la SDK y características ofrecidas por LootLocker.

Para finalizar, si al publicarlo en Steam el juego triunfa en varios países, sería interesante permitir cambiar el idioma del juego desde el menú principal.

Bibliografía

[1] Wijman, T. (22 de diciembre de 2021). *The Games Market and Beyond in 2021: The Year in Numbers*. Newzoo. https://newzoo.com/insights/articles/the-games-market-in-2021-the-year-in-numbers-esports-cloud-gaming

[2] Kelly, M. (21 de noviembre de 2021). *Worlds 2021 peak viewership numbers increase by 60 percent from 2020, break all-time record*. Dot Esports. https://dotesports.com/league-of-legends/news/worlds-2021-viewership-increases-60-percent-from-2020-breaks-all-time-record

[3] Unity Technologies. (2022). *Elige el plan adecuado para ti!*. Unity Store. Recuperado el 6 de octubre de 2022. https://store.unity.com/es/compare-plans

[4] LootLocker Inc. (2022). *Pricing*. Recuperado el 6 de octubre de 2022. https://lootlocker.com/pricing

[5] Mitchell, S. M., & Seaman, C. B. (Octubre de 2009). A comparison of software cost, duration, and quality for waterfall vs. iterative and incremental development: A systematic review. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 511-515.

[6] Vázquez Moreno, J. D. (2019). *Introducción a Unity para videojuegos 2D*. Domestika. https://www.domestika.org/es/courses/716-introduccion-a-unity-para-videojuegos-2d

[7] Sommerville, I. (2015). *Software Engineering* (10.ª ed.). Pearson Educación.

[8] Unity Technologies. (11 de noviembre de 2019). *Cinemachine.* Unity Learn. Recuperado el 17 de enero de 2023. https://learn.unity.com/tutorial/cinemachine#

[9] Unity Technologies. (20 de enero de 2023). *MonoBehaviour.Update().* Unity Documentation. Recuperado el 21 de enero de 2023. https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

[10] Unity Technologies. (20 de enero de 2023). *MonoBehaviour.FixedUpdate().* Unity Documentation. Recuperado el 21 de enero de 2023. https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html

[11] Unity Technologies. (20 de enero de 2023). *MonoBehaviour.LateUpdate().* Unity Documentation. Recuperado el 21 de enero de 2023. https://docs.unity3d.com/ScriptReference/MonoBehaviour.LateUpdate.html

[12] Unity Technologies. (20 de enero de 2023). *MonoBehaviour.OnEnable().* Unity Documentation. Recuperado el 21 de enero de 2023. https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnEnable.html

[13] Unity Technologies. (20 de enero de 2023). *PlayerPrefs.* Unity Documentation. Recuperado el 22 de enero de 2023. https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

[14] Unity Technologies. (20 de enero de 2023). *Application.persistentDataPath.* Unity Documentation. Recuperado el 22 de enero de 2023. https://docs.unity3d.com/ScriptReference/Application-persistentDataPath.html

[15] Unity Technologies. (20 de enero de 2023). *Unity Test Runner.* Unity Documentation. Recuperado el 23 de enero de 2023. https://docs.unity3d.com/es/2018.4/Manual/testing-editortestsrunner.html

[16] Unity Technologies. (20 de enero de 2023). *Configurable Enter Play Mode.* Unity Documentation. Recuperado el 23 de enero de 2023. https://docs.unity3d.com/Manual/ConfigurableEnterPlayMode.html

[17] Unity Technologies. (20 de enero de 2023). *Profiler Overview.* Unity Documentation. Recuperado el 23 de enero de 2023. https://docs.unity3d.com/Manual/Profiler.html

[18] Mubert Inc. (2021). *Human X AI Generative Music.* https://mubert.com/