Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un videojuego en Unity con editor de niveles en línea

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Alfonso Hernández Hernández

Tutor: Jorge González Mollá

2020-2021

Resumen

Este trabajo tiene como objetivo desarrollar un videojuego de plataformas en Unity donde los jugadores puedan crear sus propios niveles y también compartirlos a través de Internet con todo el mundo. Se desarrollará el juego en Unity y un repositorio dónde se almacenarán los niveles para que todos los jugadores puedan tanto publicar como descargar niveles. El editor de niveles estará integrado en el propio juego, así como la posibilidad de buscar, descargar y publicar juegos.

Palabras clave: Videojuego; Unity; Plataformas; Editor de niveles; Redes; Desarrollo de Software; Ingeniería del Software

Abstract

This work has the objetive of developing a platformer videogame in Unity where players can create their own levels as well as share them through the Internet with everyone. Alongside the game, we will create a repository where levels will be stored so that players can access them for publication or downloading purposes. The level editor will be integrated in the game as well as the ability to search, download and publish levels.

Keywords: Videogame; Unity; Platformer; Level Editor; Networking; Software Development; Software Engineering

Tabla de contenidos

Contenido

Tabla de figuras

1. Introducción

Los videojuegos están cada día más presentes en nuestras vidas, en algunos casos siendo una parte imprescindible de estas, rompiendo la monotonía y aportando una nota alegre. Desde su origen, han sido una fuente de ideas creativas que no ha parado de entusiasmar a su público, aumentando extremadamente su presencia en el mercado cada año.

El desarrollo de los videojuegos permite crear experiencias únicas que podemos compartir con otros, al crear un videojuego estamos expresando nuestras ideas en este, pero poder crear un videojuego de principio a fin es algo que no está al alcance de todos, se necesitan bastantes conocimientos para poder producir un producto de calidad.

1.1 Motivación

Los videojuegos como norma general proporcionan una experiencia limitada, una vez has hecho todo lo que se te permite no hay más. Pero ¿y si dejásemos al jugador crear sus propias experiencias?, ¿y si estos también pudiesen compartirlas con el resto de los jugadores? Este trabajo pretende explorar esta idea, dejar que los jugadores creen sus propios niveles y los compartan con el mundo, creando así un ecosistema donde los propios jugadores creen el juego que están jugando entre todos.

Otro de los motivos es el desarrollo de un juego en línea, el hecho de que los jugadores puedan compartir información entre ellos de manera sencilla, es un reto como desarrollador que me entusiasma abordar, ya que supone un gran esfuerzo para que la experiencia sea lo más cómoda y sencilla para el jugador.

Por último, desarrollar una aplicación de principio a fin, con los retos que esto supone, diseño, planificación, desarrollo, etc. el hecho de pasar por todas las etapas de desarrollo de un producto, es algo que no había llevado a cabo hasta ahora, y que considero esencial como ingeniero de software. Incluso una vez acabado este trabajo, realizar el mantenimiento de esta aplicación.

1.2 Objetivos

El objetivo de este trabajo es desarrollar un videojuego de plataformas con un editor que permita a los jugadores crear sus propios niveles. Este juego también brindará la posibilidad de compartir estos niveles con otros jugadores, para esto se va a desarrollar un repositorio de niveles, donde los jugadores pueden tanto publicar como buscar y descargar niveles.

Desarrollar el videojuego para que sea compatible con varias plataformas (PC, Web, Mac, Linux, Android e IOS).

Apoyar el proceso de desarrollo de software mediante una herramienta de gestión de proyectos ágiles.

Desarrollar software de calidad para asegurar su mantenibilidad en el futuro, hacer uso de patrones de diseño para que la estructura sea cohesiva y fácil de entender, así como elegir una arquitectura acorde a las necesidades del trabajo.

Desplegar una base de datos accesible por todos los usuarios que sea eficiente y altamente disponible.

1.3 Metodología

Para el desarrollo de este trabajo se ha planteado una metodología ágil organizada por *sprints*. En estos *sprints* abordaremos distintas tareas para desarrollar los módulos funcionales de la aplicación. Para ayudarnos en la planificación, vamos a utilizar una herramienta web para la gestión de tareas, *sprints* y épicas.

Al inicio del proyecto se realizará una especificación de requisitos para establecer los límites del trabajo. No se pretende hacer una especificación completa, pero sí establecer una lista de tareas organizada y lo que se debe hacer en estas. Una vez realizada esta especificación se comenzará el primer *sprint*.

Al inicio de cada *sprint* se elegirá un conjunto de tareas del *backlog* que vamos a realizar, durante el *sprint* iremos abordando estas tareas, es posible que aparezcan nuevas tareas que no habíamos tenido en cuenta durante la especificación de requisitos inicial, debido a que esta no ha sido lo suficientemente detallada, estas

nuevas tareas pasaran a estar en el *backlog* para ser abordadas en un próximo *sprint*. Al acabar cada tarea comprobaremos que se ha implementado correctamente, revisando que se cumplen todas sus pruebas de aceptación.

Si al acabar el *sprint* existen todavía tareas en curso, las devolveremos al backlog y actualizaremos su especificación. Para dar por finalizado el *sprint* deberemos verificar el funcionamiento de todas las tareas realizadas. Antes de iniciar un nuevo *sprint* especificaremos las nuevas tareas que han aparecido durante la realización del *sprint* anterior, si existen.

Repetiremos este proceso hasta acabar el desarrollo propuesto inicialmente, pero sin sobrepasar los límites establecidos en la especificación inicial.

1.4 Estructura

Este documento se estructura en 9 capítulos que van a detallar el trabajo realizado, en cada uno de sus apartados se profundizará en detalles que se consideren de interés para el documento.

Tras esta introducción abordaremos el segundo capítulo, el estado del arte. Aquí daremos un breve repaso a la industria y a los productos que se asemejan a nuestra propuesta. Veremos las características de estos productos, sus fortalezas, debilidades y cuál es nuestra propuesta de valor.

Una vez abordado el estado del arte, en el tercer capítulo, se realizará un análisis del problema en cuestión, arrojando luz sobre el trabajo a realizar. Indagaremos en las necesidades de este trabajo y finalmente redactaremos una especificación de requisitos a seguir durante el proyecto.

Una vez analizado el problema y especificados los requisitos, en el cuarto capítulo, diseñaremos una solución para este. Aquí discutiremos la arquitectura de la aplicación, estudiando el caso de uso de este trabajo y eligiendo la mejor posible, habiendo analizado todas las opciones disponibles.

En el quinto capítulo se va a detallar el desarrollo de las componentes principales de la aplicación, resaltando los detalles que sean de interés para el trabajo. Haremos notar el uso de patrones de diseño en el código, algoritmos de gran interés, etc.

A continuación, en el sexto capítulo abordaremos la implantación de nuestro producto, es decir, el despliegue de los servicios, y las diferentes vías de distribución del juego en las plataformas disponibles.

Posteriormente, en el séptimo capítulo detallaremos el conjunto de pruebas que se han llevado a cabo sobre el producto final y cómo se han realizado.

En el octavo capítulo, analizaremos los resultados del trabajo realizado y recopilaremos las conclusiones a las que hemos llegado, dando fin a este trabajo. También veremos la relación del trabajo realizado con los estudios cursados.

Finalmente, en el noveno capítulo detallaremos propuestas que sería interesante abordar una vez finalizado este trabajo, con el fin de explorar nuevas opciones y características que no hemos podido incluir en este.

1.5 Convenciones

Con el fin de mejorar la legibilidad de los bloques de código incluidos en el documento, se van a introducir en forma de capturas de pantalla de un IDE (Entorno de desarrollo integrado), sin embargo, se puede encontrar el código de estas capturas en forma de texto con la fuente "*consolas*" en el anexo de este documento.

2. Estado del arte

El campo del desarrollo de videojuegos es muy amplio, abarca muchos géneros y estilos, en nuestro caso se pretende desarrollar un videojuego perteneciente al género plataformas. Este género fue de los primeros en aparecer debido a que sus mecánicas eran sencillas de implementar con la tecnología existente.

Originalmente los juegos de plataformas eran muy sencillos, solo consistían en desplazarse verticalmente evitando obstáculos u horizontalmente a lo largo de una serie de plataformas. Un ejemplo de esto es *Donkey Kong* en el que controlamos a *Mario* y tenemos que llegar a la cima de un edificio evitando obstáculos.

Este género ha ido evolucionando hasta el día de hoy, introduciendo nuevas mecánicas cada vez más complejas a medida que la tecnología lo permitía. Hoy en día podemos encontrar títulos como *Super meat boy* o *Crash Bandicoot* que se centran en el movimiento del personaje, el diseño de niveles y en ofrecer una experiencia que rete al jugador.

También podemos encontrar juegos como *Celeste* que haciendo uso del movimiento del personaje a lo largo de una serie de escenarios, es capaz de narrar una historia que transmite emociones al jugador.

Existen también juegos que han explorado el concepto de dejar a los usuarios que creen sus propias experiencias, por ejemplo: *Super Mario Maker*, *Little big planet* o *Geometry dash*, donde el jugador puede crear niveles que aparecerán al resto de jugadores.

Otros juegos que contienen algunos elementos de plataformas, pero no son el foco principal del juego son *Hollow Knight* u *Ori and the blind forest*, estos juegos están centrados en la exploración, el combate y la historia.

Figura 1. Hollow Knight

2.1 Crítica del estado del arte

En el mercado existen varios juegos que se asemejan a este trabajo, por ejemplo: *Super Mario Maker*, *Little big planet* y *Geometry dash*. Los dos primeros tienen una característica en común, son propietarios de una consola, *Nintendo switch* y *Play Station Portable (PSP)* respectivamente. Esto presenta una barrera para la mayoría de los jugadores, ya que no solo se debe comprar el juego, sino que se debe poseer la consola en cuestión.

Al solo estar presente en una plataforma, esta es la única manera de acceder a su información, lo cual es limitante para los usuarios. En el caso de *Super Mario Maker*, al ser un producto de *Nintendo* debemos pagar una subscripción para usar sus servicios en línea.

En el caso de *Geometry dash* encontramos que está disponible en varias plataformas (PC, Mac, iOS, Android), pero la experiencia que presenta es restrictiva, ya que el personaje se mueve constantemente hacia la derecha y lo único que tiene que hacer el jugador es clicar o tocar para saltar.

2.2 Propuesta

La experiencia que pretende ofrecer este trabajo es más amplia y versátil, permitiendo al jugador un rango de movimiento mucho mayor, además se proporciona una herramienta mucho más potente que permite a los jugadores crear niveles con más profundidad y detalle.

Para el desarrollo propuesto utilizaremos un motor que nos dé la posibilidad de construir el juego para múltiples plataformas (PC, Web, Linux, Mac, iOS y Android), presentando una ventaja sobre los juegos mencionados anteriormente ya que los jugadores podrán tener acceso a su información desde múltiples dispositivos.

En cuanto al precio del producto, será algo a estudiar dependiendo de los costes de la solución.

3. Análisis del problema

El análisis de este trabajo consistirá en definir qué debe hacer la aplicación y qué funcionalidades y servicios va a ofrecer al usuario. Para este análisis vamos a hacer uso de técnicas de identificación de requisitos con el fin de conseguir unos requisitos de calidad. También definiremos cual va a ser el plan de trabajo que vamos a seguir, así como los posibles gastos que pueda conllevar, el presupuesto.

3.1 Identificación y análisis de posibles soluciones

Durante la realización de este análisis se ha creado un modelo de dominio, este representa los conceptos que se manejan en la aplicación y las relaciones entre estos. Cabe notar que este diagrama a pesar de estar expresado en notación UML no representa ninguna de las clases de la aplicación, ni el flujo de datos, ni el esquema de base de datos.

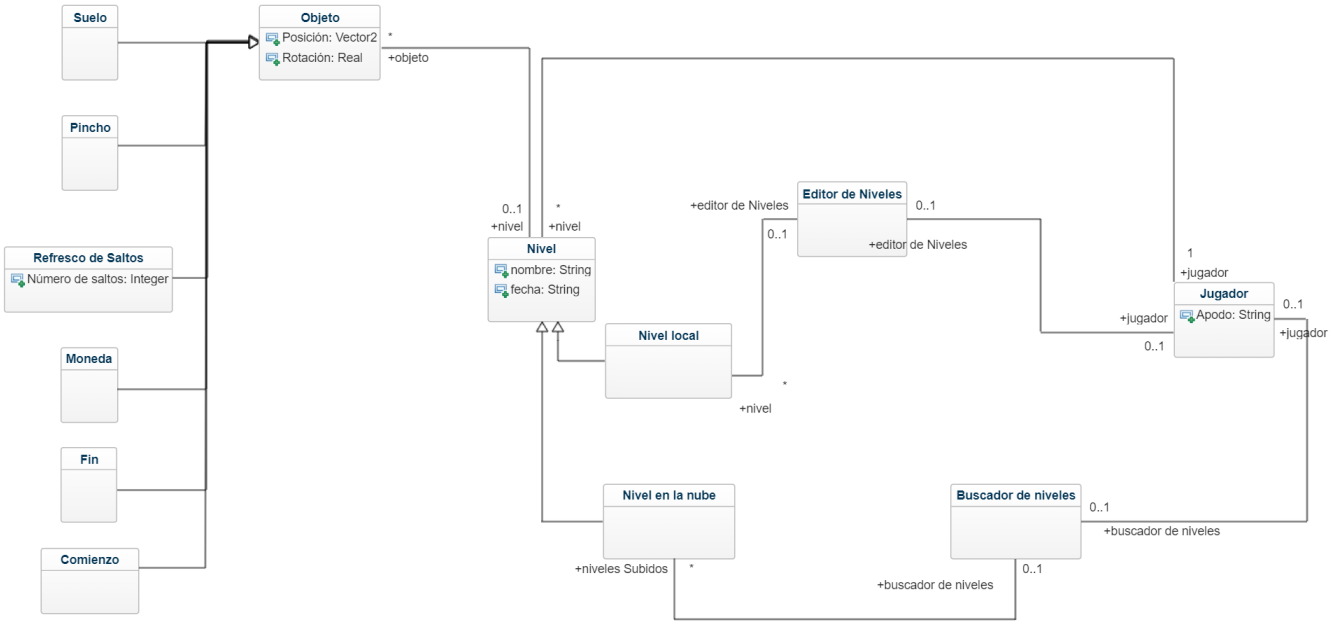

Figura 2. Modelo de dominio

A continuación, podemos comenzar con la especificación de los requisitos de la aplicación, haciendo uso del conocimiento condensado en este diagrama y los objetivos que se han establecido para el trabajo.

3.2 Solución propuesta

En este apartado vamos a exponer los requisitos que se han especificado para la solución propuesta. Empezaremos por dividir los requisitos de este desarrollo en 5 épicas o características: Juego Base, Editor Visual, Serialización, *Back end* e Interfaz.

Ahora repasaremos una a una estas características especificando cada uno de los requisitos necesarios para implementarlas. Cada característica estará compuesta por una lista de casos de uso o unidades de trabajo que describirán las tareas a realizar, así como las condiciones que se deben cumplir para que se consideren completadas.

Juego base: esta característica comprende el desarrollo del videojuego base, las mecánicas de este y el control del jugador.

ID	0
Nombre	Movimiento del personaje
Descripción	Se definirá e implementará cómo interactúa el usuario con el juego, cómo mover al personaje y cómo saltar en el nivel.
Pruebas de aceptación	- Podemos mover al personaje de izquierda a derecha - Podemos saltar más de una vez sin tocar el suelo - Si estamos en el aire la gravedad nos lleva hacia abajo

ID	1
Nombre	Piezas del nivel
Descripción	Se diseñarán e implementarán las piezas básicas que componen un nivel: suelo, pinchos e inicio. Para cada una de estas se implementará su comportamiento.
Pruebas de aceptación	-No podemos atravesar el suelo -Al inicio del nivel aparecemos en el inicio. -Al colisionar con un pincho se lleva al jugador al inicio.

ID	2
Nombre	Mecánica de saltos limitados
Descripción	Implementaremos la mecánica que limita nuestros saltos, así como el elemento que nos proporciona saltos al colisionar con él.
Pruebas de aceptación	-Si nuestro número de saltos es 0, no podremos saltar. -Si nuestro número de saltos es mayor a 0 y saltamos una vez, este número disminuirá en 1 unidad. -Al tocar una plataforma de refresco de X saltos se nos establecerá el número de saltos a X. -Solo se nos establecerán los saltos de una plataforma de refresco si colisionamos con su parte superior. -Al colisionar con un pincho nuestro número de saltos se establece a 0.

ID	3
Nombre	Monedas
Descripción	A lo largo del nivel existirán como máximo 3 monedas que el jugador puede recoger opcionalmente.
Pruebas de aceptación	-Al colisionar con una moneda esta desaparece y se suma a nuestro contador de monedas. -Al colisionar con un pincho vuelven a aparecer todas las monedas. -Al final del nivel se nos muestra cuantas monedas hemos recogido.

ID	4
Nombre	Final de nivel
Descripción	Al final del nivel se debe mostrar un panel con información y opciones.
Pruebas de aceptación	-Al colisionar con el fin se nos muestra una ventana con el número de monedas conseguidas y las opciones de repetir el nivel o ir al menú principal.

ID	5
Nombre	Crear niveles de ejemplo
Descripción	Crearemos una serie de niveles de ejemplo que permitan al jugador explorar y entender las mecánicas del juego.
Pruebas de aceptación	-Existe al menos un nivel de ejemplo que sea sencillo de completar. -Existe al menos un nivel de ejemplo que presente un reto al jugador.

Editor Visual: esta característica se encarga del desarrollo del editor de niveles, las herramientas y facilidades que proporcionará este, y la manera de guardar la información del nivel en tiempo de ejecución.

ID	6
Nombre	Panel Superior
Descripción	Panel que ayudará al jugador a construir y configurar el nivel que está editando.
Pruebas de aceptación	-Aparecerán todos los tipos de objetos para poder seleccionarlos y construir con ellos -Podremos activar la funcionalidad de *Grid*. -Podremos activar la funcionalidad de *Snapping*. -Podremos modificar el nombre del nivel. -Podremos guardar el nivel. -Podremos descartar el nivel.

ID	7
Nombre	Panel Lateral
Descripción	Panel que ayudará al jugador a alterar las propiedades del objeto seleccionado.
Pruebas de aceptación	-Podremos rotar el objeto seleccionado. -Podremos eliminar el objeto seleccionado. -Podremos deseleccionar el objeto seleccionado. -Podremos alterar el número de saltos solo si se trata de una plataforma de refresco de saltos.

ID	8
Nombre	Poner objetos
Descripción	El editor nos permitirá poner objetos si tenemos uno de los elementos seleccionados.
Pruebas de aceptación	-Al clicar sobre un espacio vacío se creará un nuevo objeto del tipo que hemos seleccionado. -Si no tenemos ningún objeto seleccionado, no ocurrirá nada al hacer clic sobre un espacio vacío

ID	9
Nombre	Mover la vista
Descripción	Dentro del editor podremos desplazar la cámara para poder navegar el nivel. También podremos alejar y acercar la vista.
Pruebas de aceptación	-Si no tenemos ningún objeto seleccionado podremos clicar y arrastrar para movernos. -Si clicamos la rueda del ratón podremos desplazarnos independientemente de si tenemos un objeto seleccionado. -Podremos alejar o acercar la vista con la rueda del ratón -Si no disponemos de un ratón también existen 2 botones para alejar y acercar la vista.

ID	10
Nombre	Eliminar objetos
Descripción	Se podrán eliminar objetos del nivel
Pruebas de aceptación	-Al hacer clic derecho sobre un objeto, este se eliminará -Existe la opción de eliminar el objeto que tenemos seleccionado en el panel lateral.

ID	11
Nombre	Mover objetos
Descripción	Podremos mover los objetos arrastrando el puntero
Pruebas de aceptación	-Al clicar y arrastrar sobre un objeto nos permitirá desplazarlo a la posición que deseemos. -Si al arrastrar un objeto soltamos, este no seguirá moviéndose.

ID	12
Nombre	Seleccionar y modificar objetos
Descripción	Al clicar sobre un objeto podremos modificar sus propiedades
Pruebas de aceptación	-Al clicar sobre un objeto podremos seleccionarlo y modificar sus propiedades en el panel lateral.

ID	13
Nombre	Rejilla para alinear objetos (*Grid*)
Descripción	Si la opción de *Grid* o rejilla está activada los objetos solo podrán tomar posiciones en una rejilla imaginaria. Los objetos no podrán ponerse en cualquier posición sino en un conjunto de posiciones uniformemente separadas entre sí.
Pruebas de aceptación	-Si la opción de *Grid* o rejilla está activada, los objetos se colocarán en las posiciones determinadas por esta.

ID	14
Nombre	Acoplamiento de vértices (*Snapping*)
Descripción	Si la opción de *Snapping* está activada los elementos se acoplarán entre ellos por sus vértices.
Pruebas de aceptación	-Si la opción de *Snapping* está activada, al acercar los elementos entre ellos lo suficiente, sus vértices se pegarán haciendo un acoplamiento perfecto.

ID	15
Nombre	Gestión de objetos en memoria
Descripción	Durante la creación del nivel, se debe guardar en todo momento cual es el estado de los elementos de este, para en un futuro poder formatear y exportar esta información.
Pruebas de aceptación	-En cualquier punto de la edición podemos acceder a los elementos que existen en el nivel. -Si borramos un elemento en el nivel, este debe desaparecer de la representación interna. -Si añadimos un elemento al nivel, este debe aparecer en la representación interna.

Serialización: esta característica definirá cómo se almacena la información de un nivel creado para que persista una vez guardado como archivo. También se encargará de cargar la información de un nivel guardado tanto en el editor para realizar modificaciones como para jugar dicho nivel.

ID	16
Nombre	Modelo de un nivel
Descripción	Se diseñará la manera en la que se va a almacenar un nivel, las propiedades que debe contener y la manera de almacenarlas.
Pruebas de aceptación	-El modelo del nivel debe contener el nombre del nivel, el autor del nivel y la fecha de modificación. -El modelo del nivel debe contener la lista de objetos y sus propiedades.

ID	17
Nombre	Serialización JSON
Descripción	Se implementará el sistema que nos permitirá crear archivos JSON que contengan un nivel basándonos en el modelo diseñado.
Pruebas de aceptación	-Al convertir a formato JSON quedan almacenados todas las propiedades presentes en el modelo establecido. -Podemos leer un archivo JSON y extraer la información de un nivel, siempre que se ajuste a el modelo establecido.

ID	18
Nombre	Instanciación de un nivel para jugar
Descripción	A partir de la información de un nivel podremos replicar un nivel que previamente había sido guardado en formato JSON y jugarlo.
Pruebas de aceptación	-Se puede instanciar un nivel a partir de un archivo JSON y se puede jugar. -El nivel replicado debe ser exactamente igual al que una vez se guardó en formato JSON.

ID	19
Nombre	Instanciación de un nivel en el editor
Descripción	A partir de la información de un nivel podremos replicar un nivel que previamente había sido guardado en formato JSON y editarlo.
Pruebas de aceptación	-Se puede instanciar un nivel a partir de un archivo JSON y se pueden editar todos sus elementos. -Se puede guardar correctamente un nivel editado -El nivel cargado en el editor debe ser exactamente igual al que una vez se guardó en formato JSON.

Back End: esta característica se encargará del desarrollo del *back end* configurando los servicios necesarios y programándolos adecuadamente para que cumplan nuestras necesidades.

ID	20
Nombre	Configuración de la API
Descripción	Desarrollaremos las componentes necesarias para poder desplegar una API a la que podamos llamar desde internet y sobre la que implementar nuestros servicios.
Pruebas de aceptación	-Existe una API a la que podemos llamar desde cualquier sitio con internet. -Existe una API con un método de prueba que nos devuelve un valor por defecto.

ID	21
Nombre	*Endpoints* de *login*
Descripción	Desarrollaremos los *endpoints* a los que se tiene que llamar para hacer el *login* y mantener la sesión. Se implementará el código necesario para que se ejecute en el *back end* y se proporcione autenticación a los usuarios.
Pruebas de aceptación	-Existirá un *endpoint* de *login* en la API. -Existirá un *endpoint* para refrescar el token de acceso en la API.

ID	22
Nombre	*Endpoint* de Registro
Descripción	Desarrollaremos los *endpoints* a los que se tiene que llamar para hacer el registro y cambio de contraseña. Se implementará el software necesario para que se ejecute en el *back end* y se registre correctamente al usuario.
Pruebas de aceptación	-Existirá un *endpoint* de registro en la API. -Existirá un *endpoint* de olvidé contraseña en la API. -Existirá un *endpoint* de reestablecer contraseña en la API.

ID	23
Nombre	Almacenamiento de niveles
Descripción	Se diseñará y configurará la base de datos para guardar los niveles generados por los usuarios.
Pruebas de aceptación	-Existe un sistema de bases de datos adecuado donde podemos almacenar y buscar nuestros niveles.

ID	24
Nombre	Sistema de búsqueda de niveles
Descripción	Se diseñará el sistema que nos permite realizar búsquedas sobre nuestra base de datos.
Pruebas de aceptación	-Existe un *endpoint* que nos permite realizar una búsqueda dado el nombre del nivel o el nombre del autor.

ID	25
Nombre	Conexión a la API desde el juego
Descripción	Se implementará el módulo que actuará de interfaz con el *back end* y se encargará de realizar las llamadas REST para nuestras peticiones.
Pruebas de aceptación	-En la aplicación existe un módulo que a través de llamadas REST se puede comunicar con la API desplegada.

ID	26
Nombre	*Endpoint* para subir nivel
Descripción	Desarrollaremos el *endpoint* que nos permitirá subir un nivel a la base de datos para que esté disponible al resto de jugadores
Pruebas de aceptación	-Existe un *endpoint* que al proporcionarle la autenticación necesaria y un nivel nos permite subirlo a la base de datos. -Si intentamos subir un nivel cuando ya hemos subido 10, debe mostrarse un mensaje de error.

ID	27
Nombre	*Endpoint* para borrar nivel
Descripción	Desarrollaremos un *endpoint* que nos permita proporcionar la ID de un nivel y eliminarlo de la base de datos.
Pruebas de aceptación	-Existe un *endpoint* que al proporcionarle la autenticación necesaria y el ID de un nivel nos permite eliminarlo de la base de datos. -Si el ID de nivel no corresponde a ningún nivel o corresponde a un nivel que no es nuestro debe mostrar un mensaje de error

ID	28
Nombre	*Endpoint* para descargar nivel
Descripción	Desarrollaremos un *endpoint* que nos permita descargar niveles dado su ID de nivel.
Pruebas de aceptación	-Existe un *endpoint* que al proporcionarle la autenticación necesaria y el ID de un nivel nos permite descargarlo para poder jugarlo. -Si el ID del nivel no corresponde a ningún nivel nos debe devolver un mensaje de error.

Interfaz de la aplicación: esta característica describe cómo será la interfaz que unirá todas las funcionalidades de la aplicación, desarrollando menús y maneras de representar la información intuitivas.

ID	29
Nombre	Menú principal
Descripción	Este menú contará con las funciones que le permitirán al usuario hacer log in y registrarse, así como acceder a otros menús.
Pruebas de aceptación	-Existe un formulario para hacer log in. -Existe un formulario para registrarse. -Existe un formulario para recuperar la contraseña. -Existe un botón para acceder al menú de niveles. -Existe un botón para acceder al menú de búsqueda. -Existe un botón para acceder al editor de niveles.

ID	30
Nombre	Menú de Niveles
Descripción	En este menú podremos visualizar todos los niveles que tengamos a nuestra disposición, para jugarlos, editarlos y gestionarlos como deseemos.
Pruebas de aceptación	-Existe una pestaña para los niveles incluidos con el juego. -Existe una pestaña para nuestros niveles guardados localmente. -Existe una pestaña para los niveles que hemos descargado. -Existe una pestaña para los niveles que tenemos subidos en la nube. -Para cada nivel podremos hacer las siguientes acciones: Jugar, lanzar en el editor, borrar localmente, subir a la nube (si aplica) y borrar de la nube (si aplica).

ID	31
Nombre	Menú de Búsqueda
Descripción	En este menú podremos encontrar un cuadro de búsqueda para encontrar los niveles que deseemos y descargarlos.
Pruebas de aceptación	-Existe una entrada de texto para buscar niveles por nombre de nivel o autor. -Existe una lista donde se muestran los resultados de nuestra búsqueda -Existe un formulario para poder descargar un nivel directamente dado su ID.

Requisitos no funcionales:

-El sistema no debe mostrar una imagen congelada por más de 5 segundos, de debe informar al usuario de que la aplicación está procesando.

-El sistema debe tener una disponibilidad de al menos 98%

-El sistema se desarrollará para las plataformas: PC, mac, Linux, iOS, Android y web.

-La aplicación no debe consumir más de 750MB de ram.

-El proceso de desarrollo se gestionará por medio de una determinada herramienta web para gestionar el proceso de desarrollo de software.

-Los colores de la aplicación serán oscuros con un color de acento azul.

3.3 Plan de trabajo

Para planificar el trabajo vamos a dividir el tiempo disponible en 5 sprints. Cada uno de estos sprints estará dedicado en su mayoría a una característica o épica de la aplicación, esto se debe a que las características tienen una naturaleza secuencial o contienen dependencias entre ellas.

Esta organización no implica que en un sprint solo pueda haber Unidades de trabajo que pertenezcan a una sola épica, esto dependerá de la planificación que se haga, de si han quedado UTs pendientes de un sprint anterior o de si alguna UT se han desglosado en más. Siempre encontraremos puntos de nuestra aplicación donde trabajaremos en varios módulos a la vez o en la comunicación entre estos.

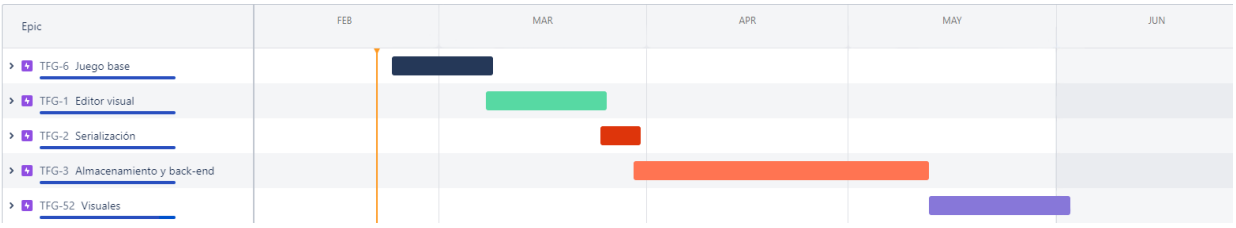

Figura 3. Plan de trabajo

3.4 Presupuesto

El presupuesto del proyecto durante el desarrollo será el salario de 1 desarrollador, que será el alumno. A posteriori tendremos en cuenta el coste de mantener la solución, es decir, el coste de los servicios del juego en sí mismo y de su distribución.

4. Diseño de la solución

En este capítulo se va a discutir sobre el diseño de la arquitectura de la aplicación y cómo estarán estructuradas sus componentes. Expondremos varias maneras de abordar la solución y elegiremos la más adecuada para este trabajo.

Una vez acordado cómo va a ser la arquitectura del sistema discutiremos sobre las tecnologías que tenemos a nuestra disposición. De nuevo decidiremos cuál es la más adecuada para el trabajo.

Tras escoger las herramientas que vamos a utilizar para el trabajo, detallaremos el diseño a realizar para la solución.

4.1 Arquitectura del sistema

Para la arquitectura del sistema vamos a plantear varias opciones y analizaremos sus pros y contras una a una. Las arquitecturas planteadas son las siguientes:

-Cliente Servidor

Esta arquitectura consta de 2 tipos de componentes, un conjunto de clientes que realizan peticiones y un servidor centralizado que responde a estas.

Esta arquitectura es sencilla de abordar, ya que solo son solamente 2 componentes, tendríamos que implementar el servidor para que satisfaga las peticiones de los usuarios y el cliente para que los usuarios se comuniquen con el servidor.

Sin embargo, esta arquitectura presenta varios inconvenientes, al ser un único servidor, este podría ser fácilmente desbordado si el número de usuarios es muy grande, y cada vez que quisiéramos escalar su capacidad, se requeriría mantenimiento que dejaría sin servicio a todos los clientes, si queremos evitar esto, tendríamos que configurar un servidor de *backup*. Además, el hecho de que sea un único servidor presenta un problema ante ataques malintencionados, fallos en el hardware o desastres naturales (incendios, terremotos, etc.)

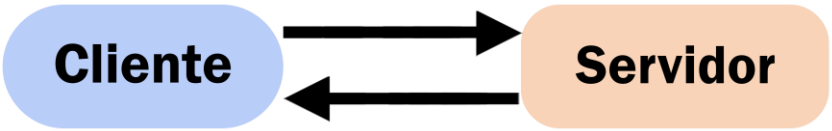

Figura 4. Arquitectura cliente servidor

-Microservicios

La arquitectura de microservicios consiste en separar las funcionalidades en múltiples componentes independientes que se comunican entre sí mediante un protocolo de comunicación sencillo como por ejemplo HTTP o TCP. Estas componentes se caracterizan por servir una única funcionalidad que pueden satisfacer sin necesidad de comunicarse con otras componentes, por ejemplo, un sistema de autenticación, que toma las credenciales de los usuarios y los identifica para poder utilizar otros servicios.

Esta arquitectura presenta varios beneficios, al ser componentes independientes se pueden desplegar en sitios geográficos distintos, evitando desastres naturales afecten al sistema por completo, también debido a su independencia, podemos escalar componentes por separado dependiendo de su carga, por ejemplo, si observamos que solamente el sistema de autenticación se ve sometido a mucha carga podemos escalar este, solucionando el problema sin tener que alterar el resto de las componentes.

Los microservicios también presentan inconvenientes, la comunicación entre estos servicios es un posible punto de fallo que los hace menos fiables, además presentan los inconvenientes de un sistema distribuido como el problema de la consistencia de los datos. Es difícil manejar un sistema de microservicios cuando tenemos un gran número de estos, es el desarrollador el que tiene que gestionar el balanceo de carga y la latencia de la comunicación. La fase de pruebas de los servicios se puede hacer complicada, al tener que depurar muchas instancias simultáneamente. Muchas de las ventajas de los microservicios solo pueden ser aplicadas en grandes aplicaciones o empresas, puede ser excesivamente complicado parar proyectos o empresas pequeñas con poca capacidad.

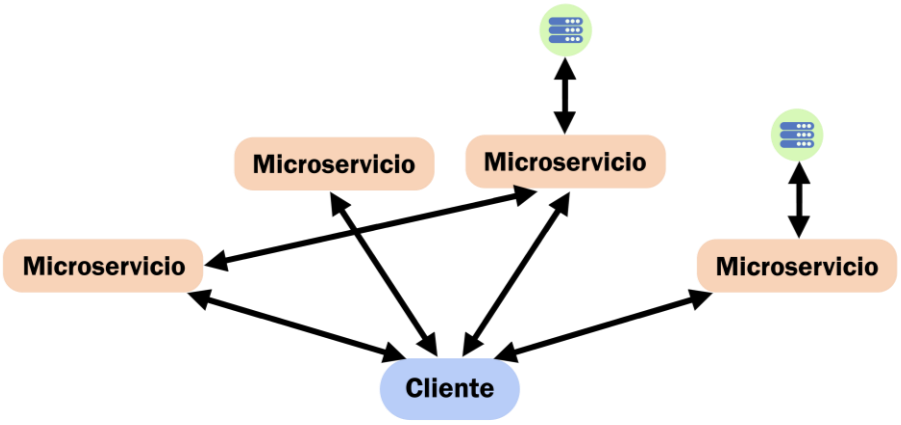

Figura 5. Arquitectura microservicios

-Serverless

La arquitectura *Serverless* es una arquitectura alojada en un servicio gestionado por un proveedor cloud ajeno al proyecto. Esta arquitectura también se conoce cono FaaS (*Function as a Service*), ya que consiste en estructurar la aplicación en funciones individuales que escalan automáticamente.

En esta arquitectura es el proveedor cloud quien gestiona la infraestructura del *back end*, se encarga de escalar los servicios en función de la carga recibida, incluso desactivándolos si no hay actividad para ahorrar costes. Un proveedor cloud nos puede asegurar cierta disponibilidad, y tolerancia a desastres naturales gracias a su gran infraestructura que, de tener que costear nosotros, nos supondría una inversión astronómica.

La arquitectura favorece tanto a proyectos pequeños, como a proyectos de gran calibre debido a su naturaleza de pago por uso, permite a proyectos pequeños crecer de unos pocos usuarios a millones realizando cambios mínimos en su configuración. Al estar todo el *back end* gestionado por el proveedor cloud, la comunicación entre las componentes es fiable y muy sencilla de implementar, lo que facilita mucho el trabajo a los desarrolladores. Otra característica que presentan la mayoría de los proveedores cloud es la capacidad de realizar actualizaciones o mantenimiento de nuestra aplicación sin someter a los usuarios a una pérdida de servicio. A pesar de también ser componentes separadas, como el caso de los microservicios, los proveedores cloud nos proveen de numerosas herramientas que facilitan las tareas de *testing* y *debugging* enormemente.

Sin embargo, esta arquitectura no es perfecta y presenta varios inconvenientes, el principal siendo la dependencia a un determinado proveedor cloud. Una vez hemos desplegado nuestra aplicación con un proveedor, es muy costoso migrar dicha solución a otro, debido a que en muchas ocasiones los proveedores utilizan tecnologías o infraestructuras propietarias, es decir, que no están presentes en otra plataforma. Para poder migrar a otro proveedor cloud, tendríamos que plantearlo prácticamente como un proyecto desde cero. Existe también el problema conocido como *cold start* o arranque en frio, debido a que es un servicio de pago por uso, no tendremos a todas horas una instancia esperando recibir peticiones, al recibir una, se comenzará a inicializar dicha instancia para servir peticiones, por ello, la primera petición que se recibe es mucho más lenta que el resto. En muchos casos, pueden existir problemas de seguridad, ya que múltiples aplicaciones pueden estar corriendo en la misma máquina física, esto puede dar pie a un robo de información por parte de un cliente del proveedor cloud.

-Peer to Peer

Una arquitectura P2P (*Peer to Peer*), consiste en un conjunto de usuarios que se comunican entre sí, sin necesidad de un servidor centralizado. Generalmente todos los usuarios son iguales, aunque puede diferir dependiendo del caso de uso. En algunas ocasiones 2 clientes no pueden comunicarse entre sí, así que es posible que exista un servidor de *relay* o repetidor, que se dedica a redirigir los mensajes entre los clientes.

La ventaja más evidente de esta arquitectura es la ausencia de un servidor, lo que abarata los costes en gran medida. Ya no tenemos la necesidad de escalar la solución para dar respuesta a un mayor número de usuarios, con la misma solución con la que partimos se pueden satisfacer usuarios casi ilimitados (seguimos limitados por la infraestructura de internet). De nuevo, gracias a la ausencia de un servidor es más complicado que suframos algún tipo de ataque que afecte a todos los usuarios.

Esta arquitectura presenta también varias desventajas, por ejemplo, si uno de los usuarios de la red es infectado, puede transmitir este virus a el resto de los usuarios rápidamente. También es difícil compartir información globalmente, por ejemplo, si un usuario quiere compartir un archivo con el resto de los usuarios tendrá que enviárselo a todos, o enviarlo a unos pocos y hacer que estos a su vez lo envíen, además, si algún usuario no está conectado a la red en ese momento no recibirá el archivo.

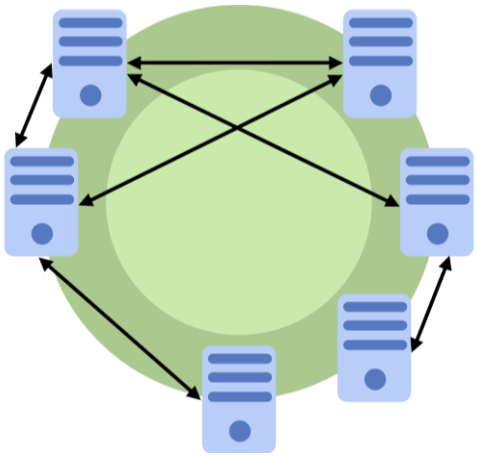

Figura 6. Arquitectura Peer to Peer

-Conclusiones

Una vez expuestas cada una de las propuestas, vamos a ver cómo se adaptarían a nuestro caso de uso y cuál es la que más beneficia a este trabajo.

La arquitectura cliente-servidor es una opción a tener en cuenta, es adaptable a nuestro caso de uso ya que vamos a tener un conjunto de usuarios a los que dar servicio, sería coherente implementar un servidor que satisfaga sus peticiones. Este planteamiento sería sencillo de abordar para este trabajo, pero nos vemos obligados a descartar esta opción ya que presenta muchos inconvenientes a la hora de mantener el sistema una vez acabado el desarrollo.

La arquitectura de microservicios presenta varios beneficios, por ejemplo, la independencia de las componentes ayuda a paralelizar el desarrollo, pero también nos deja la responsabilidad de manejar el balanceo de carga y la latencia de las comunicaciones. Visto que para nuestro caso de uso no existirían muchas componentes, no nos aprovecharíamos de todos los beneficios que aporta esta arquitectura, sobre todo siendo un equipo de desarrollo pequeño. Por estas razones también hemos decidido descartarla.

La arquitectura *serverless* resuelve muchos de los problemas que presenta la arquitectura de microservicios, además aporta otros beneficios, presentándose como una buena alternativa dado nuestro caso de uso y equipo de trabajo. Es cierto que existe el problema de la migración de proveedor, pero en este estado del proyecto no contemplamos que esto se vaya a realizar en un futuro cercano. El problema que presenta el *cold start* no repercute un gran impacto en el rendimiento de la aplicación, y los problemas de seguridad que ocurren al compartir hardware son muy poco

probables y los ataques son difíciles de ejecutar. Por estas razones hemos elegido esta arquitectura sobre el resto. Para mitigar el problema de la migración de proveedor, realizaremos un estudio sobre qué proveedor cloud nos es más conveniente para nuestra aplicación.

Por último, *peer to peer* es una arquitectura que nos ofrece una gran escalabilidad con unos costes mínimos, el problema más importante que se nos plantea para nuestro caso de uso es el de la transmisión de información a todos los usuarios, cuando un jugador quiera compartir un nivel para que esté disponible para el resto de los usuarios, va a ser muy costoso. Al ser una acción que se va a realizar recurrentemente no nos podemos permitir que sea tan compleja. Otro inconveniente es que la lógica para comunicar todos los clientes entre sí sería muy compleja y habría que tener en cuenta los posibles fallos de seguridad. En conclusión, para nuestro caso de uso sería mucho más conveniente una arquitectura centralizada, por ello vamos a descartar esta opción.

Tras haber elegido *Serverless* como la arquitectura de nuestro sistema, debemos detallar también la arquitectura que tendrá el juego en sí mismo y la arquitectura dentro de nuestro proveedor cloud, ya que *Serverless* solamente define la estructura general de nuestro sistema, ahora tenemos que detallar la arquitectura de las componentes principales, el *front end* o videojuego y el *back end serverless*.

4.1.1 Arquitectura del *front end*

El *front end* de esta solución va a consistir en el juego en sí mismo y la comunicación con el *back end* en nuestro proveedor cloud. Para la arquitectura de nuestro juego vamos a plantear diferentes arquitecturas:

-Basada en eventos

La arquitectura basada en eventos consiste en la interacción con el usuario, es decir, en reaccionar a entradas o eventos. La mayoría del tiempo se pasa esperando a que ocurra algo y entonces realizar las acciones especificadas.

Esta arquitectura es muy adaptable a entornos complejos y poco predecibles, escala fácilmente y es muy extensible. Sin embargo, las pruebas pueden ser difíciles en esta arquitectura, sobre todo si los módulos tienen muchas dependencias entre ellos.

Por último, el manejo de errores es difícil de estructurar sobre todo cuando varios módulos deben manejar el mismo evento.

-Modelo Vista Controlador

La arquitectura MVC (Modelo Vista Controlador) divide la aplicación en 3 tipos de componentes, el modelo, que maneja la lógica principal y los datos de la aplicación, la vista, que se encarga de mostrar información al usuario y el controlador, que maneja la entrada del usuario.

Esta arquitectura, al separar la aplicación en 3 tipos de componentes que trabajan juntas, se pueden dar tiempos de desarrollos más rápidos y permite que los cambios solo afecten a una parte de la aplicación, facilitando el mantenimiento. Estos factores la ponen como una buena opción para desarrollar aplicaciones tanto de pequeña como gran escala.

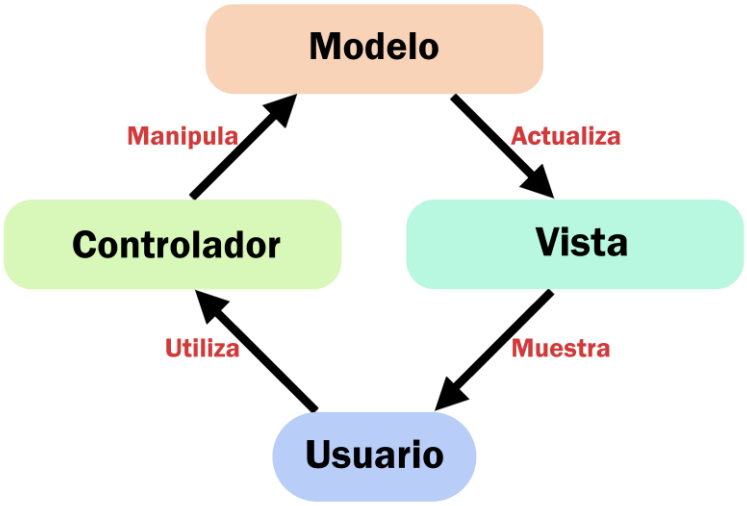

Figura 7. Arquitectura modelo vista controlador

-Por capas

La arquitectura por capas estructura la aplicación en n capas, según las necesidades de la aplicación, la comunicación sólo es posible entre las capas adyacentes.

Debido a que la comunicación está restringida a las capas adyacentes, evitamos complejidad innecesaria y como ocurría en la arquitectura MVC, la separación de componentes ayuda al mantenimiento de la aplicación y a localizar errores más fácilmente. Es una arquitectura extensible, no es complicado añadir nuevas funcionalidades una vez está implementada la aplicación.

Otra característica que tiene a su favor es su sencillez ya que no requiere invertir demasiado esfuerzo en la organización de esta.

Esta arquitectura también presenta varios inconvenientes, su estructura es monolítica, no promueve la independencia de sus módulos, y debe ser desplegada como un conjunto.

-Conclusión

La arquitectura basada en eventos es muy útil para aplicaciones que en su mayoría sean interfaz de usuario, ya que facilita el desarrollo y es intuitivo, por otro lado, nuestra aplicación no consta solo de interfaz, a pesar de ser un juego que maneja eventos y entradas del usuario, existen varios módulos que de seguir esta arquitectura se convertirían en demasiado complejos de implementar. Por estas razones vamos a descartar esta opción para nuestro *front end*.

Los inconvenientes que presenta la arquitectura por capas no nos afectan demasiado, al ser una aplicación que se va a distribuir como un solo paquete y no necesita ser escalable, cada usuario tendrá su propia instancia. Esta arquitectura presenta algunos de las ventajas que ya nos daba MVC, pero siendo mucho más permisiva con su estructura y aportando otros beneficios. Por estas razones hemos escogido la arquitectura por capas para desarrollar nuestro videojuego.

La arquitectura del juego consistirá en 4 capas que gestionarán diferentes aspectos de este. Por una parte, tenemos la capa de presentación, la cual se encarga de la navegación entre ventanas, mostrar al usuario la información correspondiente y trasladar las peticiones del usuario a la capa de lógica.

La capa de lógica se va a encargar de procesar las peticiones que haga el usuario y en el caso de que sea necesario se comunicará con la capa de datos. Dentro de la capa de lógica también tendremos una parte que se encargará de gestionar la lógica del juego, es decir, gestionar las entradas del jugador y transformarlas en acciones dentro del juego.

La capa de datos, esta se encargará de comunicarse con el *back end* en nuestro proveedor cloud y de la persistencia de la información.

Por último, tenemos la última capa, que será la base de datos y la lógica alojada en nuestro proveedor cloud, es decir, el *back end.*

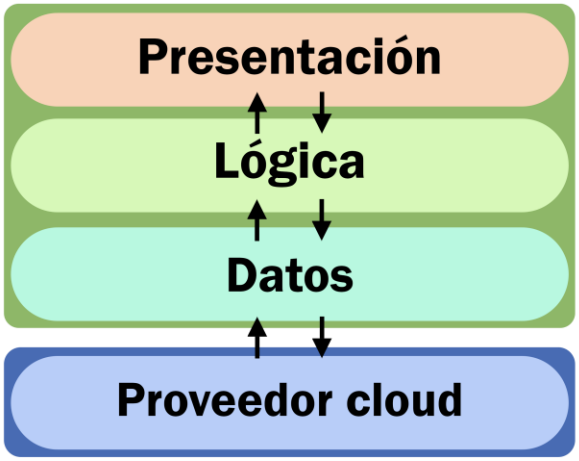

Figura 8. Arquitectura del videojuego

4.2 Tecnología utilizada

En esta sección empezaremos por elegir el proveedor cloud para nuestra solución, vamos a analizar los principales competidores de este mercado y elegir el que mejor se ajuste a nuestra solución. Los proveedores cloud líderes en el mercado son: Google *cloud*, *Amazon web services* y Microsoft Azure.

-Google Cloud

Google cloud es uno de los proveedores más populares, en parte debido al respaldo que le ofrece el gigante que es Google en la industria. A pesar de esto, no se ha adoptado por muchas empresas debido a que no ofrece la misma cantidad de herramientas que otros proveedores. También se debe a que no fue el primero en llegar el mundo del cloud y tampoco pudo recuperar la dominación que *Amazon web services* había conseguido. De todos modos, es una opción a contemplar y se espera que evolucione mucho en los próximos años.

-Amazon web services

Amazon web services es el proveedor número 1 hoy en día, se presenta muy por delante de sus competidores ofreciendo un servicio de calidad y en constante evolución. Como consecuencia de la pandemia del covid-19, el confinamiento de la población y el consecuente incremento en el uso de los servicios cloud, este proveedor que ya lideraba el mercado ha crecido enormemente, innovando y aprovechando esta oportunidad de oro. AWS es la opción por defecto para muchas empresas que quieren lanzarse al mundo cloud y la tendencia es que siga siendo así al menos durante algunos años.

-Microsoft Azure

Azure lanzó su servicio cloud un poco más tarde que sus competidores, a pesar de esto, ha conseguido mantenerse entre los mejores, pero siempre a la sombra de su principal competidor, Amazon. Las soluciones que propone Azure son de calidad, pero su foco está en la integración con sus otros servicios, lo que es muy conveniente para muchos negocios que ya forman parte del ecosistema Microsoft. Azure nos ofrece soluciones de gran calidad, aunque no lleguen a superar a *Amazon web services*, pero como hemos visto tiene otras cualidades que pueden hacer de Azure la opción por defecto para algunas empresas.

Figura 9. Comparativa de los servicios cloud principales del mercado

Una vez hemos analizado a los principales proveedores, podemos ver que sin ninguna duda *Amazon web services* es la opción a elegir. La oferta de Google es aún un poco carente, y las oportunidades de integración de Azure no nos benefician de ninguna manera. Por ello, vamos a utilizar AWS para el desarrollo de nuestro *back end*.

A la hora de desarrollar un juego una de las primeras cuestiones que debemos plantearnos es qué motor utilizar. En nuestro caso necesitamos elegir un motor que sea multiplataforma, como por ejemplo Unity, Godot o Unreal Engine.

-Unity

Unity es uno de los motores de videojuegos más populares entre los desarrolladores independientes ya que ofrece multitud de herramientas, guías y documentación gratuita. Unity será gratuito mientras que no obtengamos más de 100.000$ en ingresos con nuestro juego en el último año.

Una de las principales ventajas de Unity es que se puede exportar a casi cualquier plataforma, las plataformas disponibles son: Windows, macOS, Linux, Android, iOS, WebGL, Xbox One, PS4, Universal Windows platform, tvOS, Nintendo switch y realidad virtual.

El motor de Unity ofrece una interfaz en C# que hace la experiencia de desarrollo mucho más amigable para los usuarios inexpertos. La documentación presente en la web es muy útil y suele presentar ejemplos para mejorar su comprensión. Unity tiene la *Asset store*, una tienda donde los usuarios pueden publicar recursos que pueden ser utilizados por el resto de los usuarios, ayudando enormemente al desarrollo, sobre todo cuando se está aprendiendo.

Aunque el proyecto Unity no sea *open source*, el equipo de Unity está constantemente trabajando en nuevas actualizaciones que mejoran la experiencia de los desarrolladores y mejoran el motor. Unity lanzó su primera versión al público en 2005, es un motor que lleva bastante tiempo en la industria y que ha probado ser una opción fiable que ofrece resultados de calidad.

Este motor también presenta algunos inconvenientes, por ejemplo, no soporta de manera nativa un lenguaje de scripting visual, sigue utilizando el compilador “Mono”, que a día de hoy empieza a estar un poco anticuado, por último el editor no está bien optimizado para construir proyectos de gran envergadura.

-Godot

Godot es un proyecto *Open Source*, es completamente gratuito y está en constante desarrollo gracias a los aportes de la comunidad. En origen era un motor dedicado a juegos 2D, pero gracias al apoyo de la comunidad desde hace poco también soporta el desarrollo de juegos en 3D.

Al igual que Unity, podemos exportar nuestro juego a varias plataformas, en este caso: Windows, macOS, Linux y BSD (*Berkeley Software Distribution*).

Para desarrollar juegos en Godot, debemos aprender a utilizar su lenguaje propietario *GDScript*, aunque recientemente se ha dado la opción de desarrollar en C#, C++ (nativo) o scripting visual.

El motor es bastante nuevo, lanzó su primera versión al público en 2014, a pesar de esto, es un motor en constante desarrollo que poco a poco se está convirtiendo en una opción muy viable para algunos proyectos, aunque le falte mucho para poder ofrecer las mismas calidades que otros motores más asentados en la industria como Unity o Unreal engine.

-Unreal Engine

Unreal Engine es un motor muy conocido en la industria, su popularidad se debe a que ha sido utilizado en proyectos de muy gran calibre, como videojuegos fotorrealistas o películas. Ha demostrado tener la capacidad de proveer gráficos asombrosos sin impactar demasiado en el desempeño de la aplicación.

Unreal Engine también exporta a una cantidad de plataformas muy amplia y similar a las que soporta Unity: Windows, macOS, Linux, Android, iOS, WebGL, Xbox One, PS4, Universal Windows platform, tvOS, Nintendo switch y realidad virtual.

Sin embargo, este motor no es perfecto, está muy enfocado a desarrollar juegos 3D de alta fidelidad, encontrando muy pocas herramientas para juegos en 2D. Además, utilizar Unreal Engine no es una tarea sencilla, a pesar de que contiene un editor basado en scripting visual, no es muy intuitivo y requiere de formación específica. La alternativa a este método de visual scripting es su API de C++, un lenguaje de programación complejo que requiere de bastante experiencia para manejar con facilidad. A la hora de aprender a utilizar este motor, aunque dispongamos de documentación, no llegaremos muy lejos solo con esta, y hay una escasez de recursos en línea para aprender las bases del motor, comparado con otros motores como Unity o Godot que tienen comunidades de desarrolladores creando recursos para el resto. Unreal Engine es un motor más enfocado a la industria profesional y requiere de mucho tiempo para manejar con soltura.

-Conclusión

Teniendo en cuenta las dimensiones del proyecto y el tiempo que disponemos para realizarlo, vamos a elegir Unity sobre Unreal Engine y Godot. Godot es un motor muy joven que aún tiene muchas cosas por mejorar, el desarrollo sería mucho menos problemático en Unity o Unreal Engine. Por último, hemos elegido Unity, ya que aprender Unreal nos llevaría mucho más tiempo que Unity, además, no vamos a poder explotar todo el potencial que ofrece Unreal Engine en cuanto a potencia gráfica.

Para este desarrollo vamos a elegir la versión 2019.4.18f1 ya es la versión más estable y la designada por Unity como LTS o *long term support*. Además, elegiremos como IDE (Entorno de desarrollo integrado) Visual studio ya que se integra perfectamente con Unity y nos ofrece muchas opciones para depurar nuestra aplicación.

La conexión entre el repositorio y los clientes debe establecerse siguiendo algún protocolo de conexión. En este caso vamos a descartar UDP ya que suponen una potencial pérdida de información. Vamos a elegir un protocolo de conexión basado en TCP, en concreto HTTP.

La comunicación será Rest (*Representational State Transfer*), un protocolo de transferencia de estado que generalmente se usa sobre HTTP y se basa en utilizar JSON para el formato de sus mensajes. Rest no solo es una manera de enviar mensajes, sino que es un protocolo, la comunicación no tiene estado, cada una de las llamadas es independiente de la anterior, el servidor no tiene que recordar el estado de cada uno de los clientes.

Al tratarse de un proyecto de desarrollo de software es recomendable utilizar un software para llevar a cabo una planificación del desarrollo y poder organizar mejor el trabajo. Podemos encontrar múltiples opciones como Worki, Jira, Trello o Agilean. En este caso, la mayoría de las opciones ofrecen una experiencia base similar, para el alcance de nuestro proyecto y al ser un equipo de una persona, no vamos a poder aprovechar todo el potencial de estas herramientas. Por estas razones, vamos a elegir una de las herramientas más usadas en la industria, Jira.

4.2.1 Paradigma Serverless

Antes de continuar con el diseño detallado de la solución, vamos a profundizar un poco más en el paradigma *serverless* y su presencia en Amazon web services.

El paradigma *serverless* consiste en no aprovisionar ningún servicio para nuestro *back end*, es decir, no vamos a mantener ninguna instancia encendida a no ser que se vaya a usar, y el número de instancias que sirven peticiones escalará proporcionalmente con la carga que se presente en ese momento al sistema. Este paradigma nos permite ahorrar mucho en costos ya que estos dependerán siempre del uso que se le dé a nuestro sistema y, en el caso de que nadie use el sistema este no repercutirá ningún gasto. Este paradigma nos permite despreocuparnos del escalado ya que los servicios escalarán automáticamente manteniendo un desempeño correcto en todo momento, siempre que no lo limitemos por algún presupuesto.

Estos no son todos los beneficios, también al desplegar nuestro servicio sobre un proveedor cloud evitamos el trabajo de administrar nuestros propios sistemas hardware y minimizamos la cantidad de trabajo que requiere escalar nuestros sistemas para dar servicio a un número de usuarios, en la mayoría de los casos este trabajo se limita a configuraciones desde la consola web o incluso ninguno.

-Componentes *Serverless* AWS

Para el proyecto vamos a utilizar un subconjunto de las componentes consideradas *Serverless* dentro del ecosistema AWS, estas componentes son las siguientes:

-AWS Lambda es un servicio que permite ejecutar funciones sin aprovisionar ni administrar servidores. Lo único que tenemos que hacer es suministrar un bloque de código y configurar el recurso desde la consola de AWS, este código solo es ejecutado cuando es necesario y escala de manera automática. El código se ejecuta en una infraestructura eficiente y de alta disponibilidad.

-AWS Api Gateway es un servicio de AWS que nos permite crear de manera sencilla múltiples *endpoints* que reciben llamadas HTTP (GET, POST, PUT…) y comunica estas peticiones con otros servicios de AWS como por ejemplo AWS Lambda. Api Gateway Ofrece funcionalidades extra como la posibilidad de crear claves para el api o cachear las peticiones de los usuarios para ser más eficiente.

-AWS Cognito es un servicio que gestiona *pools* (Piscinas/Grupos) de usuarios. Cognito nos ofrece una manera sencilla de registrar, autenticar y autorizar usuarios para utilizar los servicios de AWS como en nuestro caso algunas de las llamadas a nuestro API Gateway.

-AuroraDB es una base de datos relacional dentro del ecosistema de AWS, pero ofreciendo los beneficios del paradigma *Serverless*, es decir, esta base de datos no tendrá siempre una instancia encendida y escalará en función de la carga, ofreciendo los beneficios de sql y ahorrando muchos costes.

-Simple Storage Service (S3) es un servicio de AWS que nos ofrece el almacenamiento de cualquier tipo de archivo en la nube a un precio muy bajo. La manera de acceder a esta información es mediante su clave, ya que están almacenamos como pares clave-valor.

Desde la consola nos puede dar la impresión de que existen carpetas dentro de un S3, pero esto no es cierto del todo, ya que simplemente son archivos que comparten parte de su clave como si fuese una ruta, “Imágenes / archivo1.jpg” y “Imágenes / archivo2.jpg”.

4.3 Diseño detallado

En este apartado detallaremos el diseño de la solución, pero en este caso teniendo en cuenta los detalles de las tecnologías que hemos elegido, aportando mayor detalle y profundidad al diseño.

4.3.1 Arquitectura detallada

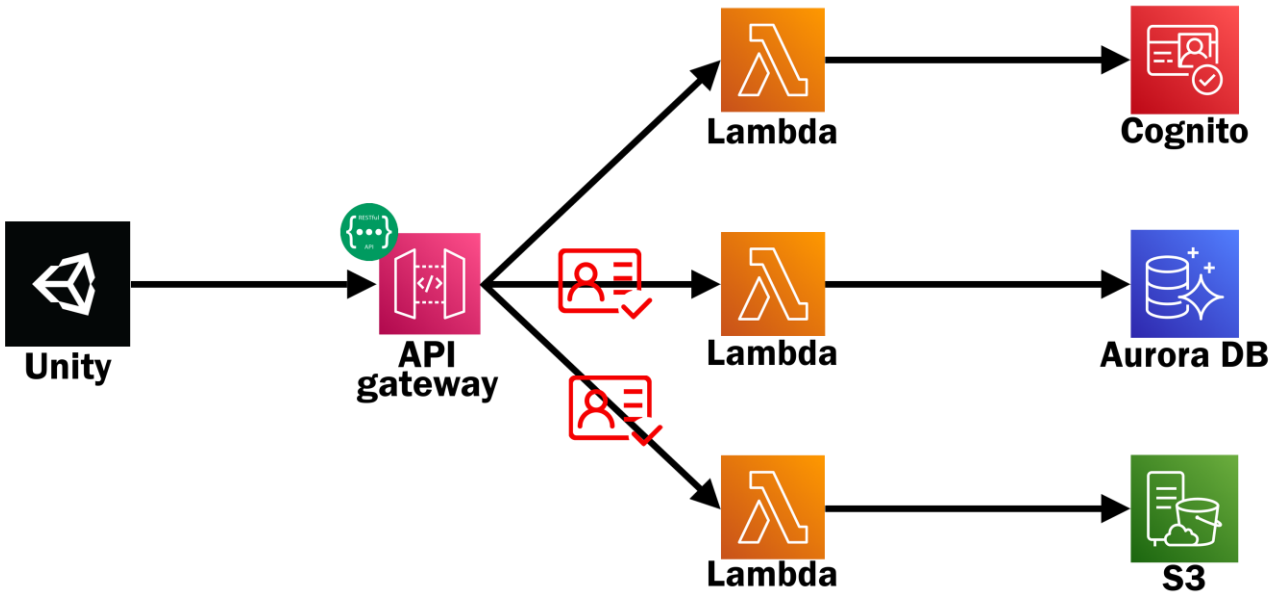

Figura 10. Arquitectura AWS

Los 3 servicios principales que hemos escogido para este proyecto son Cognito, AuroraDB y S3, esto es debido a su naturaleza *Serverless*. El resto de las componentes como Lambda y API Gateway también son *Serverless* y dentro de nuestro *back end* cumplen una función de comunicación entre el cliente y las componentes principales.

Para el desarrollo de nuestras Lambdas afortunadamente podemos hacer uso del SDK de AWS, lo que simplifica el código en gran medida y hace la comunicación entre componentes AWS muy sencilla. El SDK de AWS está disponible para cualquier lenguaje soportado por AWS Lambda.

Cognito nos facilita el registro e identificación de usuarios, a un coste muy bajo, nos comunicaremos con este servicio mediante API Gateway y Lambdas. Los usuarios deberán registrarse y autenticarse para realizar ciertas operaciones, para esto, una vez registrados deberán introducir sus credenciales y Cognito les dará un *ID Token* que deben utilizar en las llamadas autenticadas a la API y un *Refresh Token*. Este *ID token* tiene una cierta caducidad, una vez se caduca debemos renovarlo haciendo una nueva llamada al api, con el uso del *Refresh Token*.

AuroraDB es una base de datos SQL relacional *Serverless*, en esta almacenaremos la información de los niveles, pero no los niveles en sí mismos. Esta decisión se ha tomado ya que el coste del almacenamiento en AuroraDB es muy costoso y podemos beneficiarnos del barato almacenamiento de S3 para almacenar los niveles en sí.

De todos modos, sería muy poco eficiente que para cada búsqueda que se realiza, devolviésemos todos los niveles que hemos encontrado, esto ralentizaría las peticiones. Lo óptimo sería que la petición de búsqueda solo devolviese información sobre los niveles y luego el usuario decida qué nivel descargar.

Al tratarse de 2 peticiones independientes podemos hacer esta separación y dejar los niveles, que es la parte que más espacio consume, en un servicio diferente con el propósito reducir el coste de la solución.

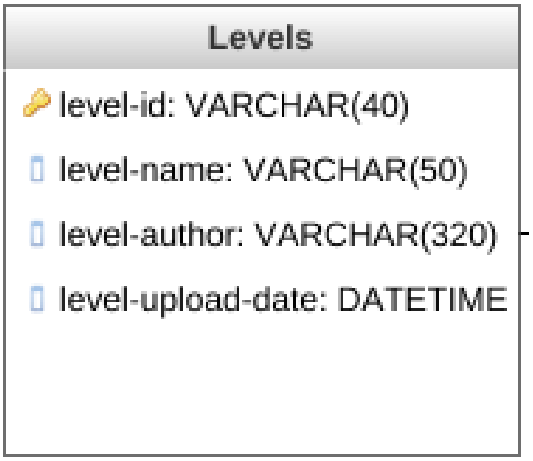

Figura 11. Esquema relacional para la base de datos SQL

Un inconveniente de S3 y el hecho que hace que no nos podamos deshacer de AuroraDB es que no podemos realizar búsquedas por parámetros, solo podemos acceder a los objetos mediante su clave.

Una vez tengamos la ID de un nivel podemos acceder al directamente sin tener que realizar ninguna búsqueda. Esto también permite que si el usuario conoce la ID de un nivel, puede descargarlo directamente sin tener que hacer uso del servicio de búsqueda que hemos implantado en AuroraDB.

El juego se comunicará con el *back end* mediante una API Gateway que recibe las llamadas en formato Rest por parte de nuestros clientes y las redirige a la función lambda correspondiente, esta se encargará de ejecutar código que gestione la petición del cliente, comunicándose con el servicio correspondiente haciendo uso del SDK de AWS. API Gateway se encargará de comprobar que los usuarios estén autenticados a la hora de hace ciertas llamadas. Esto lo hace comprobando que las llamadas contienen una cabecera *Authentication* y que esta contiene un IDToken válido.

Por una parte, existirán *endpoints* en nuestro API Gateway que no necesiten de autenticación para ser invocados, este será el caso de las llamadas responsables del registro o el inicio de sesión, por definición no pueden requerir de autenticación.

Por otra parte, el resto de *endpoints* requerirán que el usuario esté registrado en el sistema y se autentique, este es el caso de los *endpoints* que acceder a la información de cada usuario, les permitirán ver sus niveles, subir nuevos, etc.

5. Desarrollo de la solución propuesta

En este capítulo vamos a profundizar en los detalles de la implementación que se consideran de relevancia para el trabajo, así como problemas que nos hemos encontrado durante el desarrollo y las decisiones que hemos tomado para resolverlos:

-Transmitir información entre escenas (*Singleton*)

Durante el desarrollo nos hemos encontrado con el problema de transmitir información de una escena de Unity a otra, pero antes de indagar en el problema voy a hacer una introducción a lo que es una escena en Unity.

En Unity una escena es una colección de objetos, estos objetos tienen componentes, que le dan propiedades al objeto, por ejemplo, una componente "*Image*" nos permite que se pueda ver una imagen en pantalla, o una componente "*Collider*" le da propiedades para colisionar con otros objetos. En Unity los scripts se deben asignar a un objeto como una componente si queremos que se ejecuten en algún momento, debido a esto, cuando cambiamos de escena el objeto deja de ser accesible y por ende también el script. La opción de utilizar solamente una escena complicaría mucho el diseño de la interfaz, haría que fuese un infierno trabajar en esta e incrementaría en mucho el uso de memoria por parte del juego.

Para esto podemos hacer uso del patrón *Singleton*, gracias a este podemos guardar una referencia al script y podemos acceder desde otras escenas. A pesar de que el objeto ya no sea accesible, si guardamos una referencia a su script podemos acceder e interactuar con el a pesar de que el objeto que la contiene no esté activo.

Un ejemplo de esto es cuando cargamos un nivel desde la escena del menú principal, la escena del menú desaparece y cargamos la escena del juego, pero necesitamos de alguna manera obtener el nivel para poder instanciarlo y jugarlo. Gracias a esta solución podemos acceder de una manera tan sencilla como esta:

```
UIManager.GetInstance().LevelToLoad;
```

Al implementar este patrón *singleton* nos encontramos con un problema, no podemos aplicar la definición por defecto del *singelton*, no podemos hacer uso del constructor de la clase *MonoBehaviour* (Clase de la que heredan las componentes en Unity).

Afortunadamente, podemos hacer uso de los métodos que ofrece esta clase, en concreto el método "Awake()". Este método se ejecutará cuando el objeto se active por

primera vez en la escena, dentro de este método nos guardaremos en una variable estática la referencia a la componente para que se pueda acceder posteriormente por quien la necesite. Si dejamos la componente en un objeto activo en la primera escena del juego, este *singleton* estará presente en todo momento, ya que nada más iniciar el juego, se guardará la referencia.

-Ejecución de llamadas autenticadas

Como hemos visto antes, el usuario obtiene un *IDToken* al iniciar sesión, pero este token tiene una caducidad y debe ser regenerado. Existe la posibilidad de que fallen algunas llamadas al API, para solucionar esto de la manera más sencilla posible hemos creado un método que dada una llamada al *back end* que necesite autenticación, intentará ejecutarla, y si falla porque el *IDToken* ha expirado, regenerará el token y volverá a intentar la llamada.

```
public static T ExecuteAuthenticatedMethod<T>(Func<T> exec)
{
    try
    {
        return exec();
    }
    catch (BackEndException e)
    {
        if (e.GetErrorCode() != 415) throw e;

        if (e.Message == "Access Token has expired")
        {
            UIManager.Instance.LogOutUser();
            throw e;
        }
        Debug.LogError("Refreshing token");
        LoginService.GetInstance().RefreshToken();
        return exec();

    }
}
```

Figura 12. Método envoltorio para llamadas autenticadas

Como vemos, solo si el *back end* nos proporciona el código de error correspondiente refrescaremos el *IDToken* del usuario para reintentar la petición. También se puede dar el caso de que la regeneración del *IDToken* falle, ya que el *RefreshToken* utilizado para realizar el refresco también tiene una caducidad (60 días), en este caso cerraremos la sesión del usuario y le obligaremos a logarse de nuevo y obtener nuevos tokens.

-Pantalla de carga

Cuando se implementó la comunicación con el *back end* a través de llamadas rest se encontró que el juego se congelaba, esto ocurría porque el hilo principal de Unity esperaba a la respuesta de la petición y dejaba de dibujar fotogramas.

La manera de solucionar esto es llevar el procesamiento a un nuevo hilo para que no bloquee al hilo principal, esta solución no es tan sencilla como parece ya que al hacer esto nos encontramos con otro problema, solo el hilo principal de Unity puede editar componentes de Unity, por ejemplo, si queremos que al acabar la petición se muestre un mensaje de notificación con la información recibida no podemos hacerlo.

Para hacer solucionar esto hemos creado el siguiente método:

```
14 referencias
public void PopLoadingScreen<T,Q,O>(Func<T,Q> exec1, Func<Q,O> exec2, T input)
{

    StartCoroutine(LoadingScreen(exec1, exec2,input));
}

1 referencia
public IEnumerator LoadingScreen<T,Q,O>(Func<T,Q> exec1, Func<Q,O> exec2, T input)
{
    GameObject go = CreateLoading();
    yield return new WaitForEndOfFrame();

    Q result = default(Q);
    bool Exception = false;

    Thread thread = new Thread(() =>
    {
        try
        {
            result = exec1(input);
        }
        catch (Exception e)
        {
            Debug.LogError(e);
            UIManager.Instance.ExceptionHandler(e);
            Exception = true;
        }
    });
    thread.Start();
    while (thread.IsAlive)
    yield return new WaitForEndOfFrame();

    if(!Exception) exec2(result);
    Destroy(go);
}
```

Figura 13. Método de pantalla de carga

Este método se trata de una Corrutina, estos métodos tiene una propiedad especial, y es que pueden devolverle el control a Unity en cualquier momento invocando:

```
yield return new WaitForEndOfFrame();
```

Esta corrutina recibe 2 funciones, y una entrada, la primera función es un bloque de código que se debe ejecutar fuera del hilo principal de Unity para no bloquearlo, y la segunda función es una función que recibirá el resultado de la primera y se ejecutará en el hilo principal de Unity, teniendo acceso a todos los elementos de la interfaz. También se proporciona una entrada que se le pasará a la primera función para su ejecución.

Si ocurre una excepción durante la ejecución del hilo, se llamará al *ExceptionHandler* de la UI, que dependiendo del error le mostrará un mensaje adecuado al jugador.

Como hemos abstraído la función lo máximo posible, podemos aplicarla a multitud de situaciones diferentes dentro de todo el proyecto.

-Snapping o acoplamiento en el editor

En el editor de niveles existe una funcionalidad que nos permite juntar piezas de manera que se acoplen perfectamente, es decir al acercar lo suficiente sus vértices, estos pasarán a tener la misma posición.

Esto se ha conseguido creando un objeto por cada vértice y colocándolo justo encima de este. Estos objetos *snap* tendrán una componente *CircleCollider* que les permitirá saber cuándo han entrado en contacto con otro objeto como él, cuando ocurra esto, el objeto que estamos colocando se desplazará para que estos objetos *snap* tengan la misma posición.

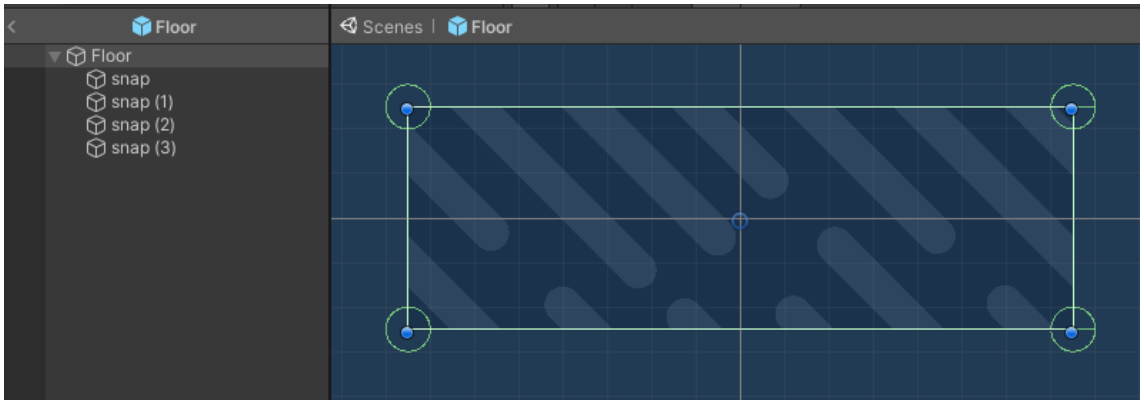

Figura 14. Detalle del objeto "Floor"

-Serialización eficiente en espacio.

A la hora de realizar la serialización de los niveles en formato JSON debemos hacerla lo más eficiente posible, ya que, si tenemos un gran volumen de usuarios creando niveles y almacenándolos en nuestra base de datos, pequeñas optimizaciones en la serialización pueden marcar una gran diferencia.

Por ejemplo, la primera optimización ha sido eliminar la indentación, ya que esta supone gran parte del espacio utilizado y no contiene información alguna, de esta manera hemos obtenido una reducción en espacio del 44.87%.

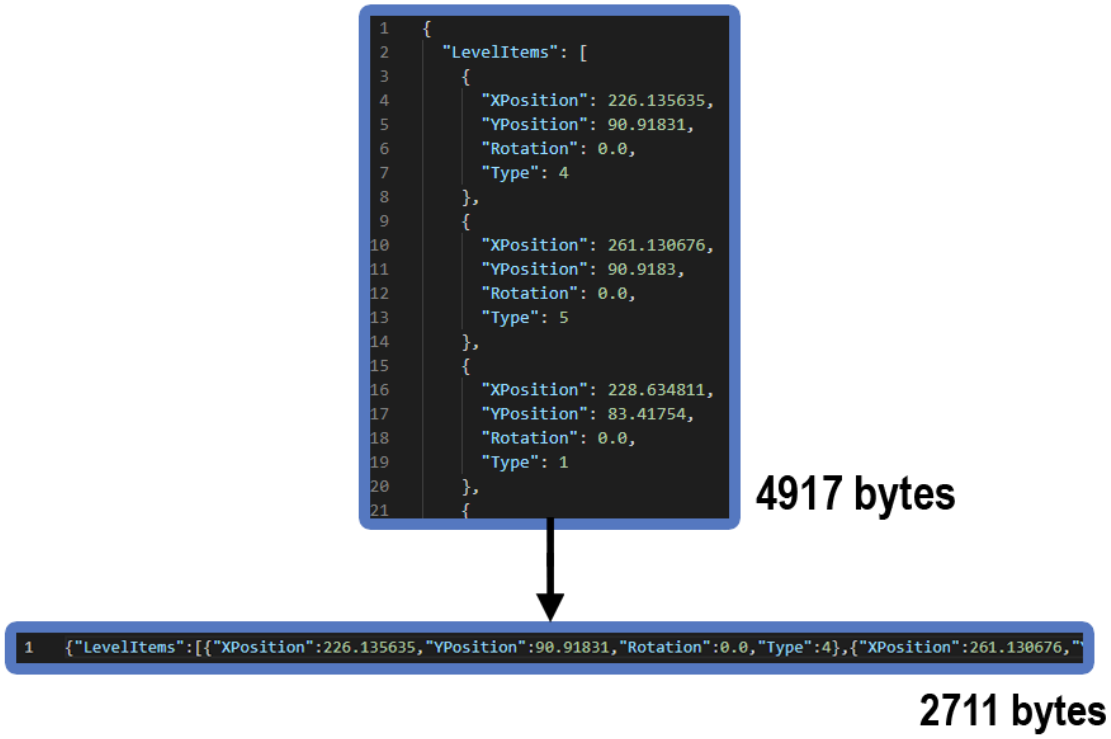

Figura 15. Comparativa de optimización JSON 1

La siguiente optimización ha sido reducir la longitud de cada uno de los atributos, por ejemplo, en vez de guardar "*Xposition*" por cada elemento, guardaremos "x". De esta manera, reduciremos el espacio utilizado por cada elemento almacenado en el nivel. Haciendo esto hemos conseguido una reducción en espacio del 35.85%.

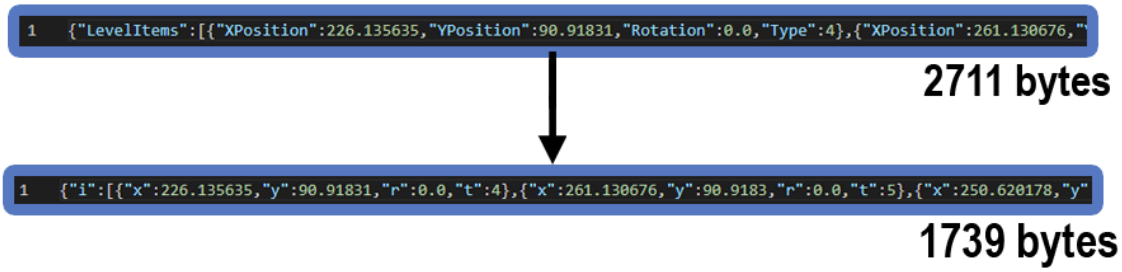

Figura 16. Comparativa de optimización JSON 2

En total gracias a estas optimizaciones hemos reducido el espacio que ocupa un nivel en un 64.63%. Los datos proporcionados se han medido sobre un nivel de ejemplo.

-Instanciación de niveles

A la hora de instanciar un nivel, ya sea para editarlo o para jugarlo debemos analizar los contenidos de un archivo JSON y generar todos los objetos necesarios. Para hacer esto hemos hecho uso del patrón *builder* o constructor, que implementa toda la lógica para traer el nivel a la vida. A este solo tenemos que pasarle el nivel a instanciar en formato JSON y se encargará de deserializar el objeto nivel, de crear cada uno de los objetos en la escena, así como iniciar el controlador del juego o el editor.

Además, se ha visto que cuando se instancian niveles de gran tamaño, el juego puede quedarse congelado mientras se procesan todos los elementos, para solucionar esto hemos hecho del método encargado de instanciar los objetos una corrutina. De esta manera el método puede devolver el control al hilo principal para que pueda dibujar fotogramas entre cada objeto y ver cómo se va generando el nivel poco a poco. Sin embargo, esto hace que la operación sea un poco más lenta, así que se ha decidido que se devolverá el control a Unity una vez cada 40 elementos, de esta manera los niveles pequeños cargarán muy rápido y en los niveles muy grandes que requieren mucho tiempo daremos *feedback* a los usuarios, evitando que piensen que la aplicación se ha colgado.

Durante todo este proceso también se mostrará la pantalla de carga, indicando a los usuarios que una operación se está realizando y evitando que puedan interactuar con la interfaz prematuramente.

-Compresión de texturas para dispositivos móviles

La primera vez que se construyó la aplicación para Android se encontró que no podía iniciarse, la aplicación se cerraba. Se descubrió que esto ocurría debido a que no había suficiente memoria en el dispositivo para alojar todos los recursos del juego.

Como podemos ver en esta instantánea de memoria, son las texturas las que ocupan la mayoría del espacio de la aplicación:

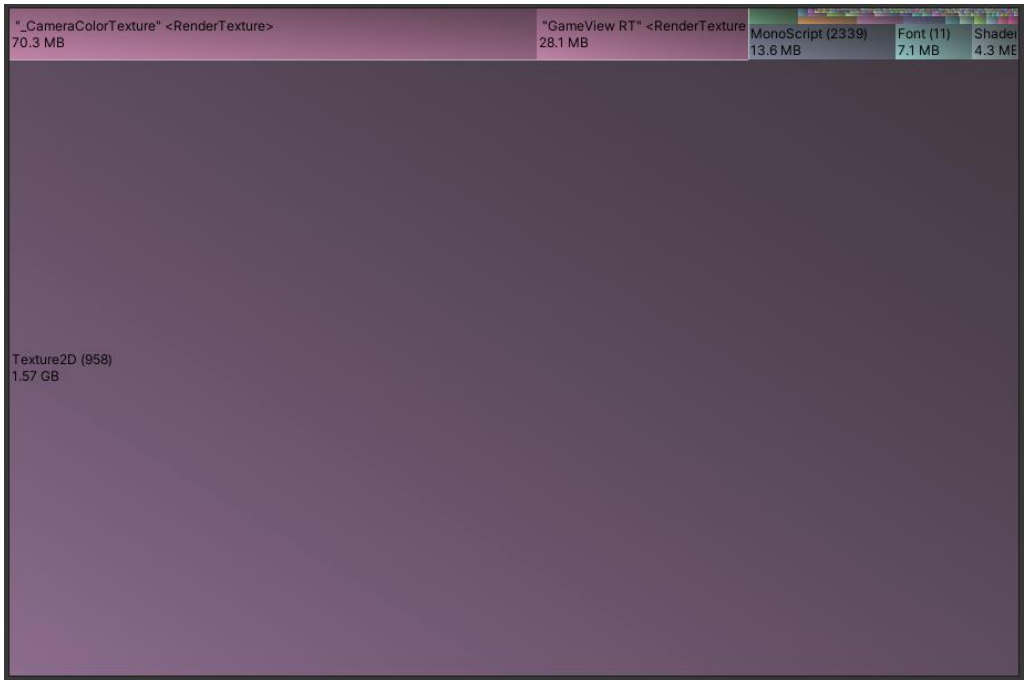

Figura 17. Instantánea de memoria antes de optimizar

Para solucionar esto, debemos configurar la compresión y el tamaño máximo de cada textura en el juego, debemos ajustar el tamaño máximo para cada imagen de tal manera que la textura no empiece a perder calidad. Si bajamos demasiado el tamaño máximo para una textura empezaremos a ver los píxeles que componen a esta, queremos evitar esto, por ello tendremos que analizar cada caso individualmente.

Una vez hecho esto el juego se ejecuta fluidamente en un dispositivo móvil. Aquí se muestra una instantánea de memoria después de realizar este proceso:

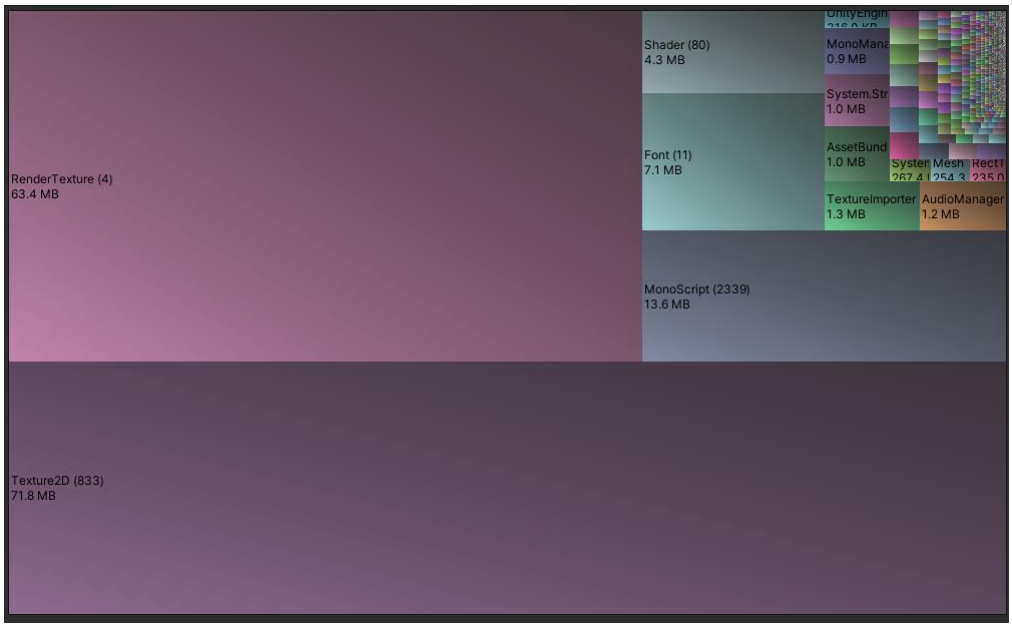

Figura 18. Instantánea de memoria después de optimizar

Por ejemplo, esta es la configuración para la textura del jugador:

Figura 19. Detalle de la configuración de una de las imágenes

Como vemos en la imagen hemos podido reducir el tamaño máximo a 128 pixeles, esto quiere decir que no podrá ser más de 128 píxeles de alto o ancho, para llegar a este número desde Unity podemos ir bajando este número y ver en tiempo real como se verá nuestra imagen en el juego con ese valor. También hemos escogido un formato de RGBA comprimido para ahorra aún más espacio.

-Ejemplo de programación de una lambda

La programación de las lambdas ha sido relativamente sencilla, ha consistido en recibir parámetros desde la API e interactuar con los servicios de AWS en la mayoría de las ocasiones. Aquí se muestra un ejemplo de una lambda y su estructura:


```
const AWS = require('aws-sdk');

exports.handler = async function handler (event, context, callback)
{
    var levelID = event.levelid;
    try{
        var s3 = new AWS.S3({region : 'eu-west-3'});

        var params = {
            Bucket: "4lfonsohh-untitledgame-database-levels",
            Key: levelID+".json"
        };
        var response = await makeS3DownloadRequest(s3,params);
        return returnResponse(200, 'Level Downloaded', response.message.Body.toString('utf-8'));

    }catch(e)
    {
        return returnResponse(400, e.message.message)
    }
}

async function makeS3DownloadRequest(service,params)
{
    return new Promise((resolve, reject) => {
        service.getObject(params,(err,data)=>
        {
            if(err)
            {
                reject
                ({
                    error: true,
                    message: err,
                })
                console.log(err);
            }else
            {
                resolve({
                    message: data,
                    })
                console.log(data);
            }
        });
    });
}

function returnResponse(code, message, data ='None'){

    return {
            'statusCode': code,
            'body': {
                'message': message,
                'data': data
            }
        }
}
```

Figura 20. Estructura del código de una Lambda

Si ocurre alguna excepción durante la ejecución de la lambda, se envía un código de excepción al cliente para que este muestre el mensaje de error correspondiente al usuario y este sepa cuál ha sido el error. Un ejemplo de esto es cuando un usuario introduce unas credenciales erróneas, la lambda redirige este mensaje de error al

cliente y se le muestra una notificación al usuario. Evidentemente, si se trata de un error técnico no se lo vamos a mostrar al usuario, solo le proporcionaremos la información necesaria.

-Arquitectura final

Para acabar con el capítulo de desarrollo e implementación vamos a repasar cómo ha quedado la arquitectura de nuestra aplicación mostrando la separación en capas de las distintas clases. Como se decidió en el diseño de la solución, hemos dividido la arquitectura en tres capas (y con el *back end en aws* cuatro), en el diagrama podemos ver estos tres grupos generales, así como subgrupos divididos según funcionalidad.

En la capa de presentación tenemos 2 bloques, la navegación entre las ventanas de la aplicación compuesta por las componentes necesarias para realizar esto y la interfaz del editor de niveles, que dirige la interacción con el usuario.

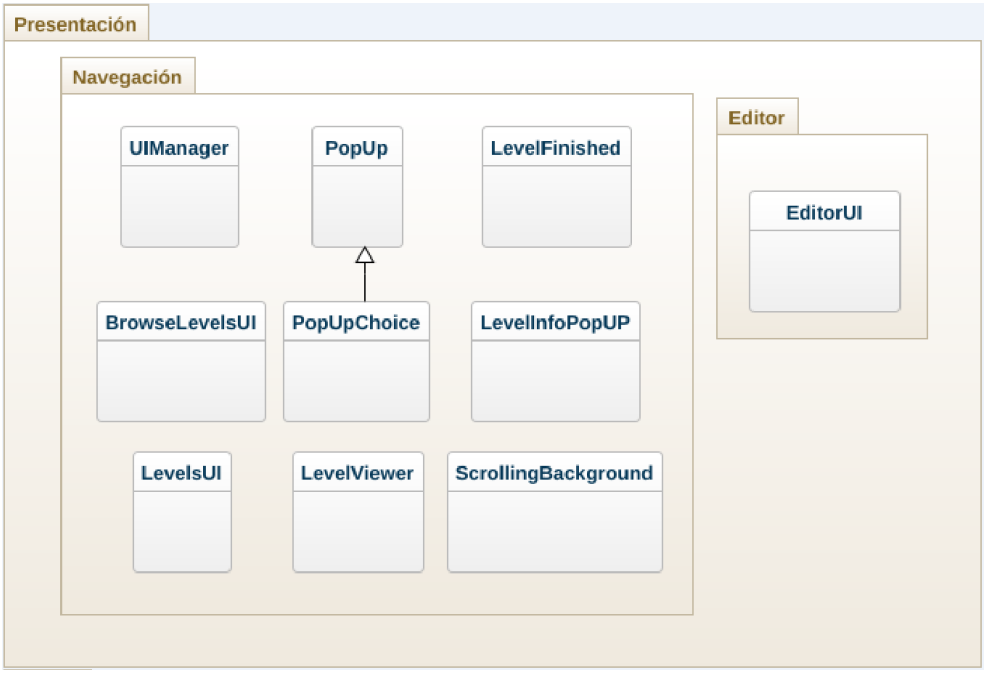

Figura 21. Capa de presentación

En la capa de lógica tenemos el bloque correspondiente al editor de niveles, las componentes responsables del acoplamiento, arrastrar objetos, almacenar información, etc. También existe el bloque que corresponde a la lógica del juego, el movimiento del jugador, el movimiento de la cámara y cada una de las componentes que corresponden a las componentes del juego. Y por último en esta capa tenemos los

modelos en base a los que se serializan y deserializan los niveles y la lógica encargada de instanciar los niveles.

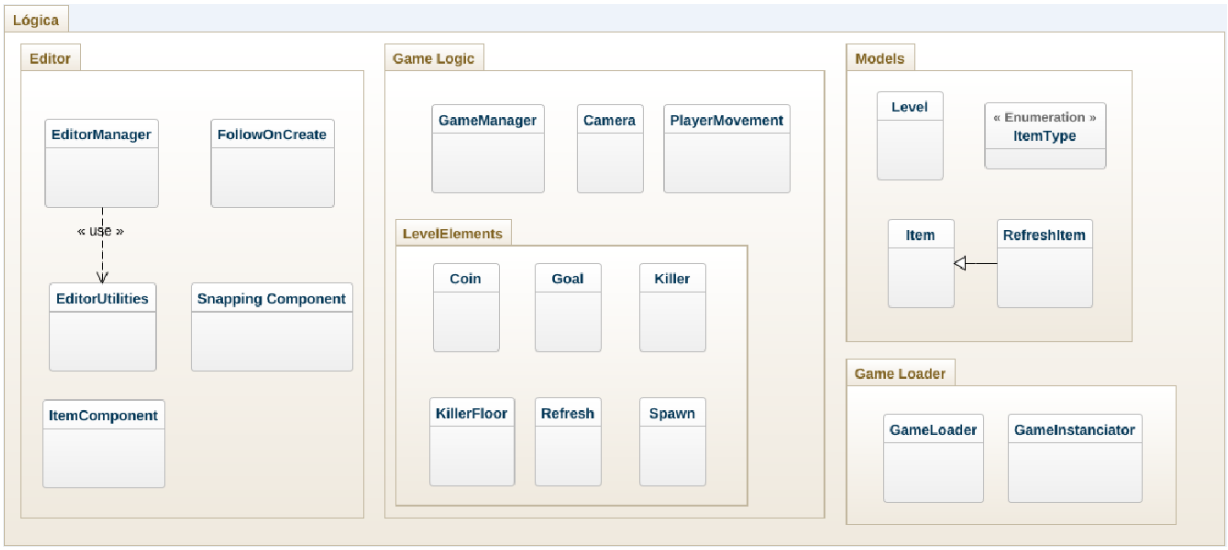

Figura 22. Capa de lógica

En la última capa, la capa de datos tenemos el código responsable de comunicarse con el *back end* de la aplicación, aquí encontramos clases que nos ayudan a envolver y simplificar la interacción con los servicios como es el caso del "*LoginService*". También encontramos las clases que se encargan de almacenar los niveles localmente, "*LevelStorage*"

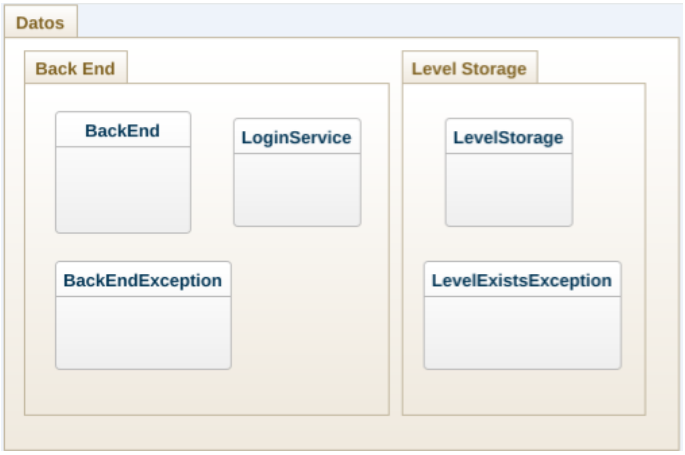

Figura 23. Capa de Datos

6. Implantación

En este capítulo vamos a analizar las opciones disponibles para realizar el despliegue de nuestra aplicación, pero no se va a llevar a cabo ya que de momento no se tiene intención de hacer el juego públicamente disponible, además de que esto podría conllevar costes innecesarios.

Hacer público el juego conllevaría cierto grado de riesgo ya que, aunque pusiéramos un precio a este, el coste del uso de los servicios podría superar los ingresos en las ventas. Habría que estudiar un modelo de monetización que permita tener un flujo constante de ingresos que nos permita mantener activos los servicios del juego. Se ha decidido dejar este tema fuera del trabajo ya que no se trata de un TFG de emprendimiento.

Plataformas para las que se ha construido el juego:

-Windows:

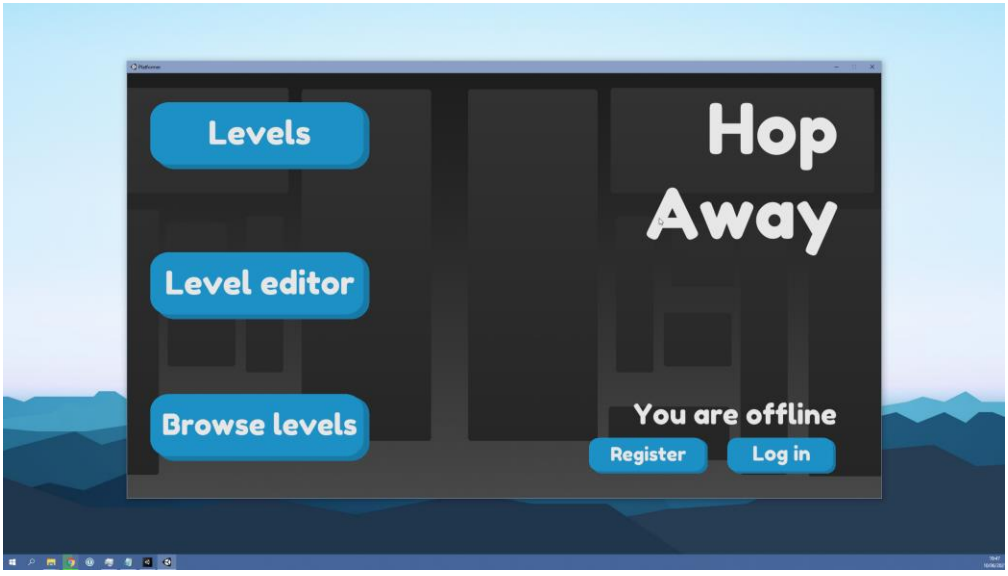

Figura 24. Videojuego corriendo en windows

-macOS:

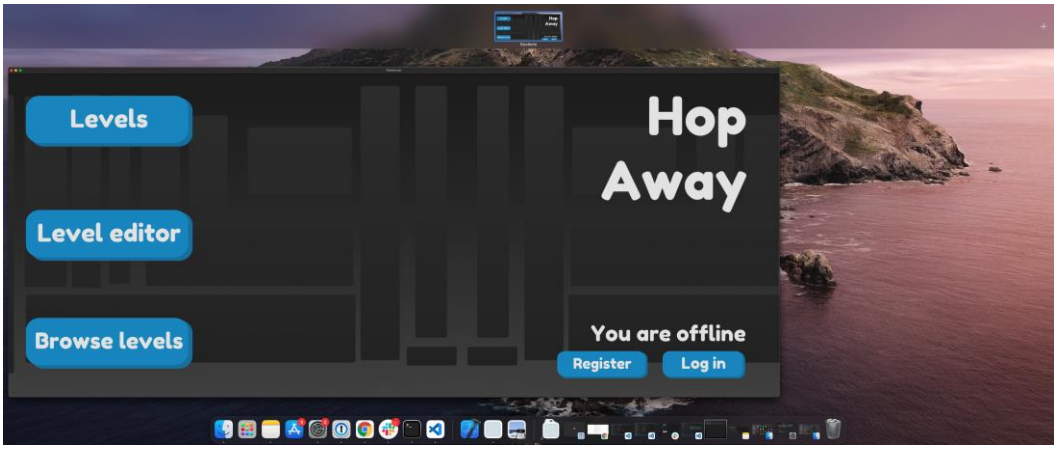

Figura 25. Videojuego corriendo en macOS

-Linux:

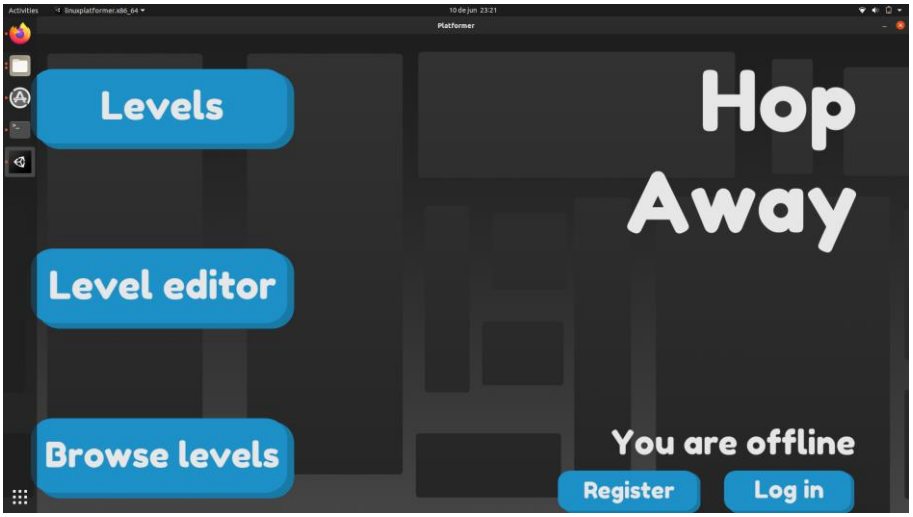

Figura 26. Videojuego corriendo en Ubuntu 20.4

Android:

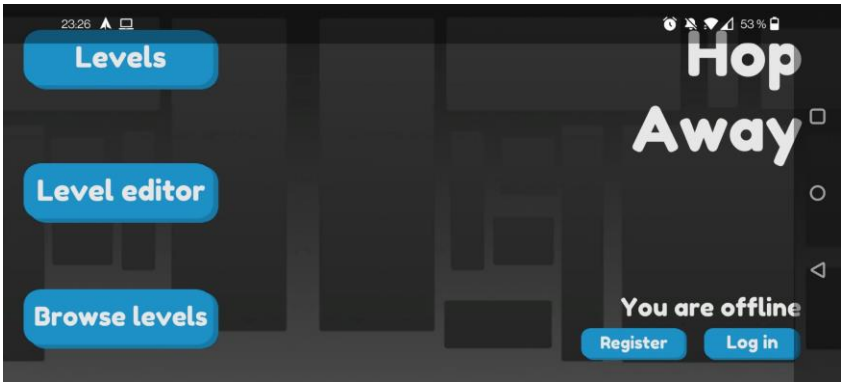

Figura 27. Videojuego corriendo en android

-iOS:

No se ha podido construir la aplicación para iOS ya que no se dispone de un dispositivo iPhone, además, para poder crear un empaquetado y enviárselo a un compañero que disponga de uno, necesitaríamos pagar la licencia de desarrolladores de apple, que supondrían 100$. Si dispusiéramos de un iPhone podríamos construir la aplicación en el dispositivo conectándolo mediante USB a un dispositivo con macOS, del que tampoco se dispone.

-Web:

No se ha podido construir la aplicación para desplegar en una web debido a que no se ha compilado correctamente. Este tipo de construcción utiliza un compilador diferente al resto, debido a esto han surgido problemas con las librerías usadas para el proyecto, así como ciertas funcionalidades del lenguaje c#. Debido a esto, adaptar la aplicación para construirse correctamente como una web se va a proponer como un trabajo futuro.

El despliegue del back end se ha llevado a cabo progresivamente durante el desarrollo, ya que se ha necesitado de su existencia para ir probando la aplicación a lo largo del trabajo.

Para el despliegue del videojuego en PC, MAC y LINUX se plantean las siguientes plataformas de distribución:

-Steam

Steam es la plataforma de distribución de videojuegos más grande del mundo, pero conlleva un coste de 100$ por cada juego que se desea publicar. A pesar de ser un gran coste para una publicación, nos ofrece muchos beneficios, al ser una plataforma tan grande y conocida por millones de usuarios el potencial es muy grande, no estaremos limitados por la plataforma de ninguna manera.

Steam soporta las plataformas: Windows (7+), macOS (10+) y Linux (Ubuntu 12+)

-Itch.io

Itch.io es una web de distribución de juegos gratuita, pero podemos elegir dar cierto porcentaje de nuestras ventas para apoyar a la plataforma. Itch.io no es tan conocida como Steam, está más orientada a desarrolladores independientes que quieren

distribuir su juego sin hacer una gran inversión, aunque no tenemos el potencial de que nuestro juego sea conocido por millones de personas es una opción muy viable.

Itch.io soporta cualquier plataforma debido a su manera de distribuir los juegos, para cada plataforma debemos subir los archivos necesarios para su instalación, y es el usuario el que debe instalar el juego.

-Web personalizada

Diseñar y desplegar una página web dedicada a nuestro juego es una muy buena opción, al igual que itch.io, se alojarán los archivos necesarios para la instalación en cada plataforma. En este caso tenemos control completo sobre cómo se presenta nuestro juego y podemos hacer customizaciones que son imposibles en otras plataformas donde debemos ajustarnos al formato. Esta manera tampoco es perfecta ya que si el juego tiene un precio tenemos que implementar nosotros los métodos de pago seguros para los clientes.

-Conclusión

Aunque no se vaya a realizar para este trabajo, la mejor opción para distribuir nuestro juego es Steam ya que si quisiéramos lanzar este producto es la plataforma que más herramientas y oportunidades nos da a pesar de que sea necesario un pago inicial. Cualquier juego con expectativas de ser conocido o de conseguir un beneficio está disponible en Steam.

El despliegue en las plataformas móviles Android e iOS, debido a su naturaleza solo puede ser mediante las tiendas oficiales o mediante una web personalizada (proporcionando los empaquetados para instalar). Es cierto que existen tiendas de terceros para Android, pero la oficial está muy por delante de estas en cuanto a herramientas, calidad y exposición.

El despliegue del juego en Web se podría llevar a cabo en un contenedor alojado en AWS. De esta manera el uso de nuestra página web también sería dependiendo del uso que esta reciba. Para realizar esto nos aprovecharemos de una característica de S3, gracias a la cual podemos hacer público un contenedor y este actuará como servidor web.

7. Pruebas

Para este proyecto se han realizado pruebas de integración entre las 2 componentes principales, el videojuego y el *back end* en AWS.

Primero se ha comprobado que cada una de las llamadas a la API funcionaba por separado. Para hacer esto hemos hecho uso de la herramienta Postman, esta herramienta nos permite lanzar llamadas HTTP a un *endpoint* elegido, proporcionando cabeceras y parámetros a nuestro gusto. Hemos probado que, para cualquier tipo de entrada, el api no fallaba y nos devolvía un resultado o un mensaje de error adecuado.

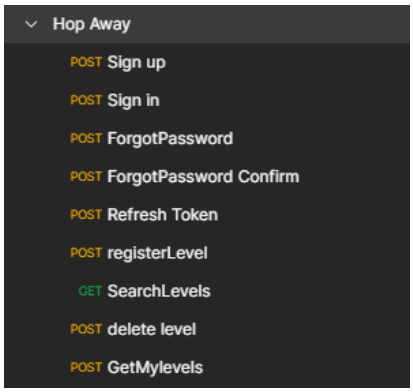

Figura 28. Conjunto de llamadas a la API desde Postman

Aquí tenemos un ejemplo de una llamada desde Postman, la llamada que se utiliza para iniciar sesión.

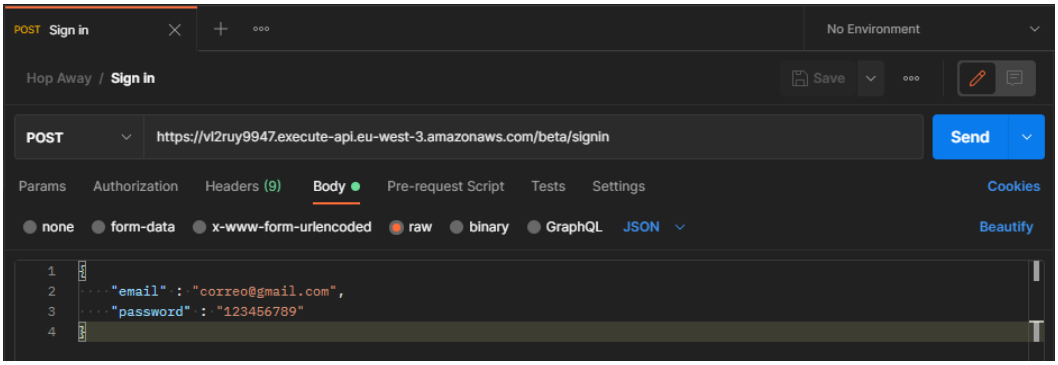

Figura 29. Ejemplo de una llamada en Postman

Por último, hemos comprobado que desde el videojuego las llamadas se realizaban correctamente y se obtenía la respuesta, además nos hemos asegurado de que para cada código de error que nos devolvía el API se le informaba al usuario correctamente de lo ocurrido.

8. Conclusiones

Tras un cuatrimestre de trabajo planificado se ha finalizado correctamente y con las funcionalidades planeadas, se ha probado su funcionamiento y se ha confeccionado este documento. Se han analizado las necesidades y se ha especificado el trabajo a realizar, se ha creado un producto que podría lanzarse al público y dar servicio a millones de usuarios sin problema, suponiéndonos un coste ajustado al pago por uso. Se ha elegido la mejor tecnología para cada caso de uso, analizando las opciones que encontramos a nuestra disposición.

Se ha desarrollado un videojuego con una arquitectura adecuada que nos permitirá mantener la calidad del código y facilitar el mantenimiento futuro. Se ha seguido una planificación ordenada mediante el uso de la herramienta Jira y una metodología ágil por sprints, lo que nos ha permitido mantener un ritmo de trabajo constante y ordenado.

Adicionalmente, se han analizado las opciones que se tienen para realizar dicha publicación y se ha elegido la opción que más nos beneficiará en el caso de dar este paso.

1. Relación del trabajo con los estudios cursados

La carrera de ingeniería informática ofrece una visión general de muchos temas y tecnologías de la industria, lo que nos permite a los alumnos poder indagar en cada una de las tecnologías que nos sean de interés o necesitemos. Entendiendo los conceptos que yacen en la raíz podemos entender las tecnologías con facilidad.

Esto no quiere decir que la carrera solo nos haya enseñado las bases, también hemos utilizado tecnologías que son punteras hoy en día como Javascript o Java.

Para la búsqueda de niveles se ha utilizado una base de datos relacional MySQL, tecnología que se aprendió a usar en la asignatura de Bases de datos y sistemas de información.

Todo el *back end* está desarrollado en Javascript, el lenguaje que se usa por toda la web y que se aprendió en la asignatura de Tecnologías de sistemas de información en la red.

El videojuego se ha desarrollado usando el motor de Unity, que expone una interfaz en C#, lenguaje orientado a objetos que, a pesar de estar por detrás de Java está muy presente en la industria y que se aprendió a utilizar en la asignatura de Ingeniería del Software.

La asignatura de Diseño de software nos enseñó la importancia de desarrollar código de calidad y nos dio las herramientas y los conocimientos necesarios para conseguirlo.

Las asignaturas de Proceso de software y Proyecto de ingeniería de software nos prepararon para una industria donde la planificación de proyectos ágiles es el camino a seguir. Por esta razón hemos decidido utilizar esta planificación para este trabajo.

El proyecto en grupo realizado en Proceso de software y Proyecto de ingeniería de software se utilizó Unity como tecnología para el desarrollo, por lo que al inicio de este proyecto ya se tenía una base sobre cómo funcionaba Unity.

Las asignaturas de Redes de computadores y Tecnologías de sistemas en red nos han ayudado a entender la comunicación entre ordenadores y a manejar el protocolo HTTP y REST.

Por último, pero no menos importante, la asignatura de Interfaces persona computador nos ha aportado los conocimientos para poder diseñar interfaces intuitivas y agradables a la vista.

9. Trabajos futuros

En este capítulo hablaremos de las ideas o mejoras que debido a las restricciones de tiempo y presupuesto no se han podido abordar en este trabajo.

-Niveles populares, “me gusta”s y número de jugadas

Esta característica daría a los jugadores la posibilidad de ver los niveles que más se han jugado o que más han gustado, así como poder ver estas estadísticas para los niveles que han subido ellos. Esta característica requeriría un rediseño del almacenamiento de los niveles y también se debería analizar la base de datos periódicamente y calculando las jugadas y los “me gusta”s. Debido a la complejidad que requería esta característica se ha dejado fuera de este trabajo.

-Comentarios en niveles

Esta característica permitiría a los jugadores dejar comentarios en los niveles que han jugado, reaccionar al nivel y dar *feedback* al creador. Esta característica requeriría añadir un nuevo almacenamiento a nuestro sistema, así como crear herramientas para moderar comentarios, añadir filtros de lenguaje soez, castigar a los jugadores que no sigan las normas de lenguaje, etc. Esta característica se ha descartado ya que requiere de la realización de muchos detalles a pulir antes de poder estar lista para el público, aunque al principio parezca una característica sencilla de implementar, si la abordamos en su completitud se complica en gran medida.

-Mundos / Conjuntos de niveles

Esta funcionalidad dará a los usuarios la posibilidad de agrupar sus niveles, de esta manera podrán crear niveles con una temática parecida e indicar un orden en el que se deben jugar para ofrecer una experiencia más completa a los jugadores. Para conseguir esto tendríamos que crear otro almacenamiento en nuestros servicios que gestione los mundos creados por los usuarios. Hemos descartado esta característica porque, aunque no parezca muy compleja no añade mucho a la experiencia de juego, se han priorizado otras características que mejoraban la experiencia de los usuarios más directamente.

-Plataforma Web y otras plataformas

Como se ha mencionado anteriormente aparecieron problemas a la hora de construir el juego para la plataforma web, esto se debía a que el compilador utilizado para esta construcción no soportaba correctamente algunas librerías utilizadas para el proyecto, así como algunas funcionalidades del lenguaje C#.

No solo se podría abordar la plataforma Web, sino otras plataformas como Nintendo switch, Xbox One, PS4, tvOS y realidad virtual requerirán de adaptaciones que deberán llevarse a cabo individualmente ya sea por tener que llevar a cabo acuerdos con empresas como Nintendo, tener que adaptar los controles, etc. Debido a esto solo se han abordado para este trabajo las plataformas relativas al PC y dispositivos móviles.

Por estas razones se ha propuesto como trabajo futuro, abordar la resolución de los errores de compatibilidad de nuestro proyecto con las múltiples plataformas.

-Niveles con límite de tiempo y mejores tiempos por nivel

También se propone la creación de un nuevo modo de juego para los niveles, un modo contrarreloj, en el que el creador del nivel pone un tiempo a batir por los jugadores. En este modo los jugadores deben llegar a la meta dentro del tiempo establecido por el creador o si no volverán al inicio. Esta opción se ha descartado porque el desarrollo del modo de juego base y los servicios ya consumían gran parte del tiempo planificado para este trabajo y no aportaría mucho a la experiencia de juego para el tiempo que tendríamos que dedicarle.

-Planificación financiera para futuro lanzamiento

A la hora de lanzar un videojuego al mercado se debe hacer una planificación financiera, en esta se tratarán temas como el precio que tendrá el juego, si se pedirá a los jugadores una subscripción mensual, las vías de pago, etc. Todos estos temas, así como la licencia de desarrolladores de apple que será necesaria adquirir si queremos distribuir nuestro juego para iOS, se deberán abordar en un estudio futuro. Esta opción no se ha incluido en este trabajo ya que no se trata de un TFG de emprendimiento y porque llevaría demasiado tiempo.

-Añadir música al juego

Como toque de calidad final se propone diseñar y añadir efectos de sonido al juego, así como componer distintas pistas de audio como música de fondo. Para abordar esto tendríamos que comisionar artistas, lo que implicaría gastos o buscar recursos gratuitos. Se ha decidido descartar esta característica ya que se ha preferido no introducir sonidos a introducir unos sonidos de baja calidad y que no acompañen al completo la estética con el juego, además del limitado tiempo del que se disponía para el trabajo.

10. Referencias

Videojuegos mencionados:

[1] Hollow Knight. Desarrollo: Team Cherry

[2] Ori and the Blind Forest. Desarrollo: Moon Studios

[3] Donkey Kong. Desarrollo: Nintendo

[4] Super meat boy. Desarrollo: Team Meat

[5] Crash Bandicoot. Desarrollo: Múltiples desarrolladores

[6] Celeste. Desarrollo: Matt Makes Games

[7] Super Mario maker. Desarrollo: Nintendo

[8] Little big planet. Desarrollo: Media Molecule

[9] Geometry dash. Desarrollo: RobTop Games

Bibliografía que ha sido consultada para la realización de este trabajo:

[10] Unity Docs. https://docs.unity3d.com/2019.4/Documentation/Manual/index.html

[11] La imagen 9 pertenece a este artículo. https://ww w.zdnet.com/article/the-top-cloud-providers-of-2021-aws-microsoft-azure-google-cloud-hybrid-saas/#listicle-7a2410f9-314e-4244-8789-d84e900818cf .El resto de las imágenes y capturas de pantalla han sido realizadas por el alumno.

[12] AWS Lambda. https://aws.amazon.com/es/lambda/

[13] AWS Cognito. https://aws.amazon.com/es/cognito/

[14] AWS API Gateway. https://aws.amazon.com/es/api-gateway/

[15] AWS S3. https://aws.amazon.com/es/s3/

[16] AWS Aurora DB. https://aws.amazon.com/es/rds/aurora

[17] AWS Serverless. https://aws.amazon.com/serverless/

[18] JSON.Net Library. https://www.newtonsoft.com/json

[19] C# Documentation. https://docs.microsoft.com/es-es/dotnet/csharp/

11. Anexo

-Figura 12:

```
public static T ExecuteAuthenticatedMethod<T>(Func<T> exec)
{
    try
    {
        return exec();
    }
    catch (BackEndException e)
    {
        if (e.GetErrorCode() != 415) throw e;

        if (e.Message == "Access Token has expired")
        {
            UIManager.Instance.LogOutUser();
            throw e;
        }
        Debug.LogError("Refreshing token");
        LoginService.GetInstance().RefreshTokenNoLoadingScreen();
        Debug.LogError("trying again");
        return exec();

    }
}
```

-Figura 13:

```
public void PopLoadingScreen<T,Q,O>(Func<T,Q> exec1, Func<Q,O> exec2, T input)
    {

        StartCoroutine(LoadingScreen(exec1, exec2,input));
    }

    public IEnumerator LoadingScreen<T,Q,O>(Func<T,Q> exec1, Func<Q,O> exec2, T
input)
    {
        GameObject go = CreateLoading();
        yield return new WaitForEndOfFrame();

        Q result = default(Q);
        bool Exception = false;

        Thread thread = new Thread(() =>
        {
            try
            {
                result = exec1(input);
            }
            catch (Exception e)
            {
                Debug.LogError(e);
                UIManager.Instance.ExceptionHandler(e);
                Exception = true;
            }
        });
        thread.Start();
        while (thread.IsAlive)
        yield return new WaitForEndOfFrame();
```

```
        if(!Exception) exec2(result);
        Destroy(go);
    }
```

-Figura 14:

```
UIManager.Instance.PopLoadingScreen<string, List<LevelFound>, List<Level>>
            ((inputtext) =>{
        List<LevelFound> founds = Service.ExecuteAuthenticatedMethod(
()=>BackEnd.GetInstance().GetSearchLevels(LoginService.GetInstance().GetIDToken(),
inputtext).body.data);
                return founds;
        },
        (founds) =>
        {
            List<Level> levels = new List<Level>();
            foreach (LevelFound level in founds)
            {
                Level l = new Level(level.level_name);
                l.SetLevelAuthor(level.level_nickname);
                l.SetLevelID(level.level_id);
                l.SetLevelDate(level.level_upload_date.Substring(0, 10));
                levels.Add(l);

            }

            ResetPages();
            CleanSearchResults();
            if (levels.Count != 0) InstanciateLevels(levels);
            return levels;
        }
        , input.text
        );
```

-Figura 20:

```
const AWS = require('aws-sdk');

exports.handler = async function handler(event, context, callback) {

    var levelID = event.levelid;
    try {

        var s3 = new AWS.S3({ region: 'eu-west-3' });


        var params = {
            Bucket: "4lfonsohh-untitledgame-database-levels",
            Key: levelID + ".json"
        };

        var response = await makeS3DownloadRequest(s3, params);
        return returnResponse(200, 'Level Downloaded', '');

    } catch (e) {
        return returnResponse(400, e)

    }
}

async function makeS3DownloadRequest(service, params) {
    return new Promise((resolve, reject) => {
        service.getObject(params, (err, data) => {
```

```
            if (err) {
                reject
                    ({
                        error: true,
                        message: err,
                    })
                console.log(err);
            } else {
                resolve({
                    message: data,
                })
                console.log(data);
            }
        });
    });
}

function returnResponse(code, message, data = 'None') {

    return {
        'statusCode': code,
        'body': {
            'message': message,
            'data': data
        }
    }
}
```