

Desarrollo de un videojuego multiplataforma con Phaser3

CFPGS DESARROLLO DE APLICACIONES MULTIPLATAFORMA



X



ALUMNO: JUAN BARRERA CUESTA
CURSO ACADÉMICO: 2022/23



IES POLITÉCNICO
HERMENEGILDO LANZ
GRANADA

ÍNDICE

- [Análisis del problema](#)
 - [Introducción.](#)
 - [Objetivos.](#)
 - [Funciones y rendimientos deseados.](#)
 - [Planteamiento y evaluación de diversas soluciones.](#)
 - [Justificación de la solución elegida.](#)
 - [Modelado de la solución.](#)
 - [Planificación temporal.](#)
- [Diseño e implementación del proyecto](#)
 - [Búsqueda de recursos.](#)
 - [Configuración del entorno.](#)
 - [Menú principal:](#)
 - [Añadido de elementos y lógica de botones.](#)
 - [Control de sonido.](#)
 - [Creación de mapa. Uso de Tiled.](#)
 - [Implementación de mapa y objetos:](#)
 - [Control de cámara.](#)
 - [Moviendo al jugador.](#)
 - [Cambio de zonas.](#)
 - [Movimiento en escaleras.](#)
 - [Escena de combate.](#)
 - [Utilidad de objetos.](#)
 - [Añadiendo NPCS.](#)
 - [Implementando los eventos de historia.](#)
 - [Sistema de guardado.](#)
 - [Exportación multiplataforma del juego.](#)
 - [¿Versión para móviles?](#)
- [Fase de pruebas](#)
- [Documentación de la aplicación](#)
 - [Manual de Instalación.](#)
 - [Manual de usuario.](#)
- [Conclusiones finales](#)
- [Bibliografía](#)

■ ANÁLISIS DEL PROBLEMA

➤ Introducción

Este proyecto busca explicar y mostrar como es el desarrollo de un videojuego partiendo desde cero. Se tratarán temas como el diseño de nivel, la búsqueda de recursos, la configuración del entorno y la exportación multiplataforma del producto final.

➤ Objetivos

La finalidad de este proyecto es desarrollar un videojuego que ofrezca una experiencia de entretenimiento para el usuario, implementando para ello diversas mecánicas que favorezcan dicha experiencia.

Por último, el juego será exportado a diferentes plataformas, tales como Windows, Linux, dispositivos móviles y web (formato original).

➤ Funciones y rendimientos deseados

Para conseguir la experiencia comentada previamente, las mecánicas con las que contará el juego serán las siguientes:

- Eventos de historia.
- Exploración.
- Combate por turnos.
- Combate contra NPCS (non playable characters).
- Guardado de partida.

Además de las ya comentadas, se hará uso de otras mecánicas que aportarán cierto estilo al juego, tales como cambios de zona o control de volumen, tanto de la música ambiente como de los efectos de sonido.

➤ Planteamiento y evaluación de diversas soluciones

A la hora de querer desarrollar un videojuego, es importante elegir el motor adecuado para ello. Un motor de videojuegos es un entorno de desarrollo que proporciona las herramientas necesarias para la creación de videojuegos.

Aunque bien es cierto que a día de hoy existen diversos motores de videojuegos, algunos más complejos que otros, en esta ocasión utilizaremos el motor Phaser, especializado en el desarrollo de videojuegos 2D en plataformas web, aunque también es posible el desarrollo 3D.

➤ Justificación de la solución elegida

He decidido usar el motor Phaser debido a que, durante el curso, he adquirido los conocimientos necesarios que me permitirán elaborar un juego por cuenta propia. Además, se hará uso del SDK Ionic para la conversión multiplataforma del proyecto.

➤ Modelado de la solución

A continuación, se detallan los requisitos correspondientes al entorno de trabajo:

- Un entorno de desarrollo integrado (IDE). En este caso, se usará [Visual Studio Code](#).
- Tener a disposición el programa [Tiled](#), necesario para la creación de mapas.
- Tener instalada la última versión de [Node.js](#).

En lo referente al dispositivo donde se va a llevar a cabo el proyecto, se recomienda contar con un mínimo de 8GB de memoria RAM. Esto se debe a que, como se está desarrollando una aplicación web, esta puede llegar a consumir gran cantidad de recursos.

➤ Planificación temporal

Por último, se indicará la planificación que se tiene con respecto a la elaboración del proyecto:

- Búsqueda de recursos (personaje, enemigos, tilesheet para el mapa): 1 semana
- Diseño e implementación de mapa: semana y media
- Implementación de mecánicas secundarias: 1 semanas
- Implementación de mecánicas principales: 3 semanas
- Corrección de errores y mejoras de código: 2 semanas y media
- Fase de pruebas y correspondiente modificación de código: 1 semana
- Realización de la documentación y presentación: 1 semana

■ DISEÑO E IMPLEMENTACIÓN DEL PROYECTO

➤ Búsqueda de recursos.

Antes de entrar en materia, es necesario buscar los recursos que vamos a necesitar, ya sean

personajes, bandas sonoras, ideas de mapas o de diseño. Con esto, uno puede hacerse una idea del camino a tomar a la hora de desarrollar su videojuego. Para estos casos, existen páginas especializadas que recogen mapas, personajes, objetos y bandas sonoras de multitud de juegos. Entre ellas destacan:

- [The Spriters Resource](#)
- [The Sounds Resource](#)

Sin embargo, también es posible encontrar este tipo de archivos repartidos por internet. Una vez se ha reunido los recursos necesarios, se comenzará con la configuración del entorno.

➤ **Configuración del entorno.**

Como se ha comentado anteriormente, además del uso del motor Phaser, se hará uso del SDK Ionic, el cuál nos ayudará a crear una aplicación híbrida. Ionic, por su parte, está basado en el framework Angular, escrito en typescript.

Una vez comentado esto, se procederá a explicar cómo configurar la carpeta donde se trabajará:

- El primer paso será tener instalada la última versión de [Node.js](#).
- Se creará una carpeta independiente al resto (en este caso, la carpeta será llamada ‘ionic’).
- Se abrirá la consola o terminal y, situándonos en la carpeta creada, ejecutaremos los siguientes comandos:

npm install -g @ionic/cli

npm install -g @angular/cli

Una vez hecho esto, crearemos un nuevo proyecto Ionic. Para ello, ejecutaremos el siguiente comando:

ionic start DreamJourney

Este comando lanzará un asistente que te guiará en el proceso de creación del proyecto. Para esto se seleccionará el framework Angular de entre las opciones disponibles. A continuación, se pedirá el tipo de plantilla que se quiere usar. En este caso, se utilizará la plantilla en blanco. Por último, el asistente preguntará si se quiere construir la aplicación con NgModules o Standalone Components, a lo que se elegirá NgModules.

Una vez seleccionadas estas opciones, el asistente comenzará a construir la aplicación. Este proceso suele tardar en crear la aplicación por completo, por lo que se tendrá que ser paciente.

Cuando el proceso haya finalizado, se podrá visualizar una estructuración de carpetas y archivos dentro de una carpeta llamada ‘DreamJourney’ (nombre que se había indicado a la hora de crear el proyecto).

Con esta parte, ya tenemos el 50% del proyecto estructurado. A continuación, se citarán las

carpetas y archivos a crear para adaptar Phaser en el proyecto generado por Ionic:

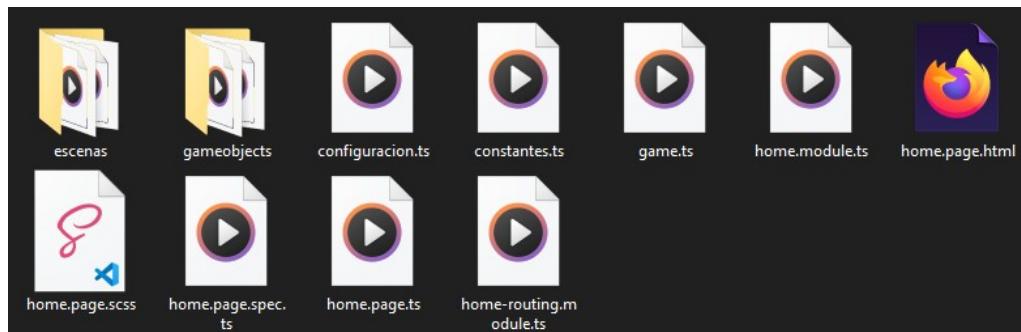
- Dentro de la carpeta del proyecto, se deberá entrar en la carpeta 'src'. Dentro de esta carpeta, se creará la carpeta 'basedatos', en cuyo interior se creará un archivo del tipo 'gestorbd.ts'. Más adelante se explicará la función de dicho archivo.

- Regresando a la estructura de la carpeta 'src', habrá que dirigirse a la carpeta 'assets' y crear las siguientes carpetas: 'audio', 'imágenes' y 'mapas'. Estas carpetas servirán para organizar los recursos que vayamos a usar para el desarrollo del juego.

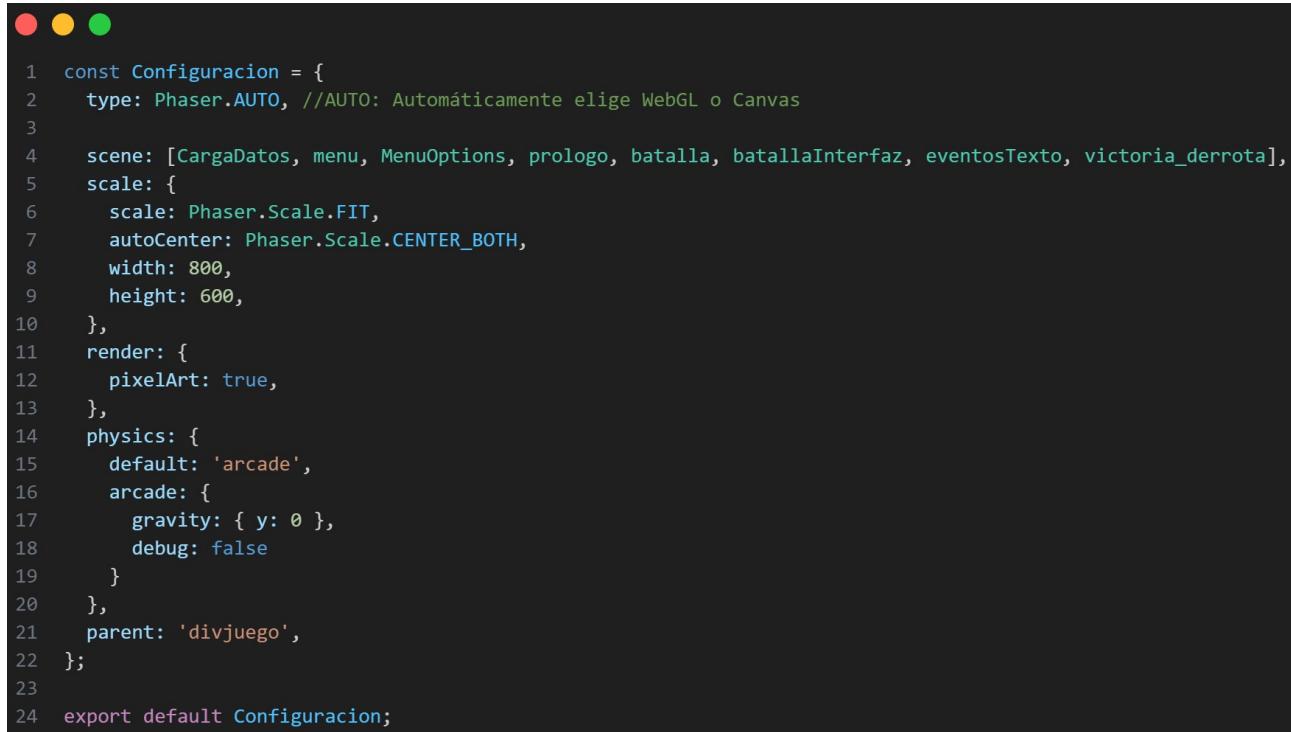
- Desde la carpeta 'src', se accederá a la carpeta 'app' y, dentro de esta, a la carpeta 'home'. Esta carpeta será la que contenga todo el código referente al juego. Empezando con los archivos ya generados, se modificará el archivo **home.page.html**, de tal forma que solo contenga el siguiente código:

```
1 <ion-content>
2   <div id="divjuego" style="margin-left: auto; margin-right: auto;"></div>
3 </ion-content>
```

- Una vez hecha esta modificación, se crearán los archivos 'configuracion.ts', 'constantes.ts' y 'game.ts', además de las carpetas 'escenas' y 'gameobjects'. La estructura debería de quedar de la siguiente manera:



Se comenzará implementando el código del archivo 'configuracion.ts'. Dicho código es el siguiente:



```
1 const Configuracion = {
2     type: Phaser.AUTO, //AUTO: Automáticamente elige WebGL o Canvas
3
4     scene: [CargaDatos, menu, MenuOptions, prologo, batalla, batallaInterfaz, eventosTexto, victoria_derrota],
5     scale: {
6         scale: Phaser.Scale.FIT,
7         autoCenter: Phaser.Scale.CENTER_BOTH,
8         width: 800,
9         height: 600,
10    },
11    render: {
12        pixelArt: true,
13    },
14    physics: {
15        default: 'arcade',
16        arcade: {
17            gravity: { y: 0 },
18            debug: false
19        }
20    },
21    parent: 'divjuego',
22 };
23
24 export default Configuracion;
```

En este archivo se declarará la configuración de Phaser que tendrá el juego. Algunos de los datos a destacar son los siguientes:

- ✗ **scene** → Almacena todas las escenas que se van a usar. De no estar declarada una escena en esta propiedad, dicha escena no se ejecutará. Lo ideal es declarar las escenas de forma cronológica, aunque luego se llamen de forma salteada.
- ✗ **render** → En este caso, como se va a desarrollar un juego 2D, deberemos declarar la propiedad `pixelArt` como `true`.
- ✗ **physics** → Aquí tendremos en cuenta lo siguiente: si se busca un juego de plataformas, habrá que aumentar la variable ‘y’ de la propiedad ‘gravity’ creando así un efecto de gravedad. Por el lado contrario, si lo que se busca es hacer un juego de vista cenital, se dejará el valor ‘y’ a 0. Por último, mientras se está desarrollando el juego, es aconsejable poner la propiedad ‘debug’ a `true`. De esta manera, podremos ver en todo momento las colisiones de cada uno de los objetos que figuran en el mapa.
- ✗ **parent** → Esta propiedad almacena el nombre del div donde se va a mostrar el juego. Es el mismo nombre que figura en el div del `home.page.html`.

- A continuación, se añadirá el código correspondiente al archivo ‘constantes.ts’. Realmente, no es un archivo obligatorio, pero si recomendado para que resulte más fácil recordar el nombre de los elementos cargados. Esto suele facilitar el hecho de llamar a un elemento diversas veces. Un extracto de este archivo es el siguiente:



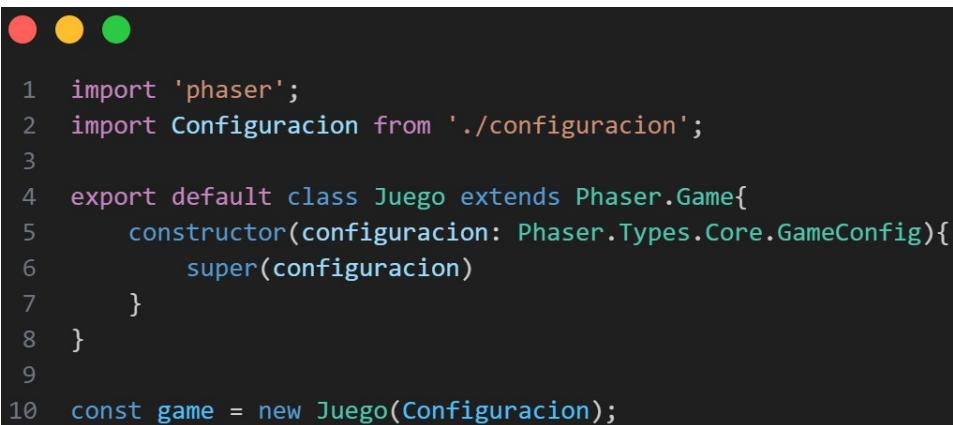
```

1 const Constantes = {
2     TITULO:{
3         TITLE: 'TITLE',
4         BOTTOM: 'BOTTOM'
5     },
6     ESCENAS:{
7         CARGA: 'Carga',
8         MENU: 'Menu',
9         MENUOPTIONS: 'MenuOptions',
10        PANTALLACARGA: 'PantallaCarga',
11        PROLOGO: 'prologo',
12        BATALLA: 'batalla',
13        BATALLAINTERFAZ: 'batallaInterfaz',
14        EVENTOTEXTO: 'eventosTexto',
15        VICTORIA_DERROTA: 'victoria_derrota'
16    }
17 }
18 export default Constantes;

```

Gracias a este archivo, se puede declarar un nombre fijo para los diferentes elementos que se implementen en el juego, pudiendo acceder a dichos nombres de manera más cómoda. Un ejemplo de llamado sería: Constantes.ESCENAS.PROLOGO, el cuál almacena el valor 'prologo'.

- Por último, se añadirá el código correspondiente al archivo 'game.ts'. Dicho código es el siguiente:



```

1 import 'phaser';
2 import Configuracion from './configuracion';
3
4 export default class Juego extends Phaser.Game{
5     constructor(configuracion: Phaser.Types.Core.GameConfig){
6         super(configuracion)
7     }
8 }
9
10 const game = new Juego(Configuracion);

```

Este archivo será el que se ejecute primero al lanzar 'ionic serve', ya que se encarga de cargar la configuración de Phaser y mostrar el contenido en el div que hemos creado.

- Antes de pasar al siguiente apartado, se creará la primera escena, la escena 'cargadatos.ts', la cuál, como su propio nombre indica, cargará todos los recursos que se le indiquen para su posterior uso en el juego. Dicho archivo se creará en la carpeta 'escenas'.

Aunque es cierto que dicha escena puede adornarse con una barra de progreso u otros detalles que indiquen el estado de carga de los recursos, se hará hincapié en el código que realmente

importa. Dicho código es el siguiente:

```
● ○ ●
1  export default class CargaDatos extends Phaser.Scene {
2
3      constructor() {
4          super('CargaDatos');
5      }
6
7      preload (){
8          this.load.on(
9              'complete', () => {
10                  this.scene.start(Constantes.ESCENAS.MENU);
11              },
12          );
13      }
14 }
```

La lógica es muy sencilla: en el momento que el sistema haya cargado todos los recursos especificados, la carga estará completa y, acto seguido, se iniciará la escena ‘Menú’ (en el siguiente apartado se verá dicha escena).

Para cargar un recurso, se deberá hacer dentro del método preload. Los recursos deben ser cargados de la manera correcta, diferenciando unos de otros. Ejemplos:

- Para cargar un mapa, se cargará de la siguiente manera:

```
● ○ ●
1  this.load.tilemapTiledJSON(Constantes.MAPAS.PROLOGO.TILEMAPJSON, 'assets/mapas/Prologo/prologo.json');
2  this.load.image(Constantes.MAPAS.TILESET, 'assets/mapas/Prologo/prologo.png');
```

- Para cargar una imagen, se cargará de la siguiente manera:

```
● ○ ●
1  this.load.image(Constantes.ELEMENTOS_MENU.FONDO_MENU, 'assets/imagenes/menu/menu_back_bottom.png');
```

- Para cargar un elemento el cuál tiene una animación, se hará de la siguiente manera:

```
● ○ ●
1  this.load.atlas(Constantes.JUGADOR.ID, 'assets/imagenes/jugador/spritesheet.png', 'assets/imagenes/jugador/spritesheet.json');
```

- Por último, si se desea cargar un efecto de sonido o una canción, esta se cargará de la siguiente manera:

```
● ○ ●
1  this.load.audio(Constantes.SONIDOS.MENU, 'assets/audio/menu_theme.mp3');
```

Con todo lo explicado, el proyecto ya estaría configurado de forma básica. En el siguiente apartado se hablará de la escena ‘Menú’, en la cuál se hablará de como controlar el volumen dentro del

juego, así como manejar elementos interactivos y en movimiento.

➤ Menú principal.

La primera escena interactiva que se creará será el menú. Aunque bien es cierto que se podría empezar directamente el juego una vez finalizada la carga de datos, he optado por añadir una escena intermedia en la cuál, además de tener el botón para iniciar el juego, también estará disponible el botón de ‘Opciones’.

◆ Añadido de elementos y lógica de botones.

A la hora de crear un botón, puede haber diversas maneras de implementarlo, ya sea una imagen interactiva, un botón como tal, etc.

Para este caso, se van a usar las siguientes imágenes para los botones ‘Jugar’ y ‘Opciones’:



Estos ‘botones’ serán implementados de la siguiente manera:

```
● ● ●
1  this.btnPlay = this.add.image(ancho/2, alto/2, Constantes.ELEMENTOS_MENU.BOTON_JUGAR);
2  this.btnOptions = this.add.image(ancho/2, (alto/2)+90, Constantes.ELEMENTOS_MENU.BOTON_OPCIONES);
```

Es importante que, al añadir una imagen, se hagan pruebas de escalado para tener una mejor presentación de estos.

Una vez hechos estos ajustes, se deberá añadir la propiedad que permitirá a las imágenes ser interactivas. En la siguiente imagen, se puede apreciar como se añade dicha propiedad y los métodos que son llamados cuando uno de los ‘botones’ es pulsado:

```
● ● ●
1  this.btnPlay.setInteractive();
2  this.btnOptions.setInteractive();
3
4  this.comenzarJuego(this.btnPlay, Constantes.ESCENAS.PROLOGO);
5  this.cambiarEscena(this.btnOptions, Constantes.ESCENAS.MENUOPCIONES);
```

Ambos métodos, ‘comenzarJuego’ y ‘cambiarEscena’, comparten el mismo código, con la diferencia del tiempo en el que se produce el fundido en negro de la pantalla. Se va a hacer hincapié en el método ‘cambiarEscena’, ya que este controlará el volumen de todo el juego desde una nueva escena ‘MenuOptions.ts’.

◆ Control de sonido.

Para controlar el volumen, se deben realizar los siguientes pasos:

- Se crearán dos propiedades estáticas en la escena ‘MenuOptions.ts’. Dichas propiedades controlarán, por un lado el volumen general del juego, y por otro lado el volumen de los efectos de sonido. Además, al estar declaradas como estáticas, se podrá acceder a dicho valor desde cualquier escena.

- Se crearán 4 botones, 2 para cada tipo de volumen. La creación de estos varía dependiendo de la idea del creador. En este caso, se ha optado por una imagen animada.

A continuación, se muestra el código de ejemplo para los botones de subir y bajar el volumen general del juego:

```

1 music_volumen_up.setInteractive().on('pointerdown', () =>{
2     music_volumen_up.play('btninteract');
3     if (MenuOptions.ambientSound < 100) {
4         MenuOptions.ambientSound = MenuOptions.ambientSound+5;
5         musicText.destroy();
6         musicText = this.add.text((ancho/2)-15, (alto/2)-65, MenuOptions.ambientSound.toString()+' %', Miestilo);
7     }
8 });
9 music_volumen_down.setInteractive().on('pointerdown', () =>{
10    music_volumen_down.play('btninteract');
11    if (MenuOptions.ambientSound > 0) {
12        MenuOptions.ambientSound = MenuOptions.ambientSound-5;
13        musicText.destroy();
14        musicText = this.add.text((ancho/2)-15, (alto/2)-65, MenuOptions.ambientSound.toString()+' %', Miestilo);
15    }
16 });

```

- Se deberá aplicar la misma estructura a los botones que controlan el volumen de los efectos de sonido. En el ejemplo, se puede observar la propiedad ‘musicText’. Dicha propiedad fue añadida de forma que muestre al usuario en todo momento el porcentaje al que está poniendo el volumen.

Por último, se mostrará la forma en la que se puede reproducir un sonido en general, ya sea música o efecto de sonido. Para el ejemplo, se ha tomado el código que reproduce un efecto de sonido en la escena principal:

```

1 Prologo.efectos = this.sound.add('encounter_effect',{volume: MenuOptions.effectsSound/100});
2 Prologo.efectos.play({
3     loop: false
4 });

```

En caso de que el sonido a reproducir sea una música, se deberá indicar que la propiedad ‘loop’ sea true.

➤ Creación de mapa. Uso de Tiled.

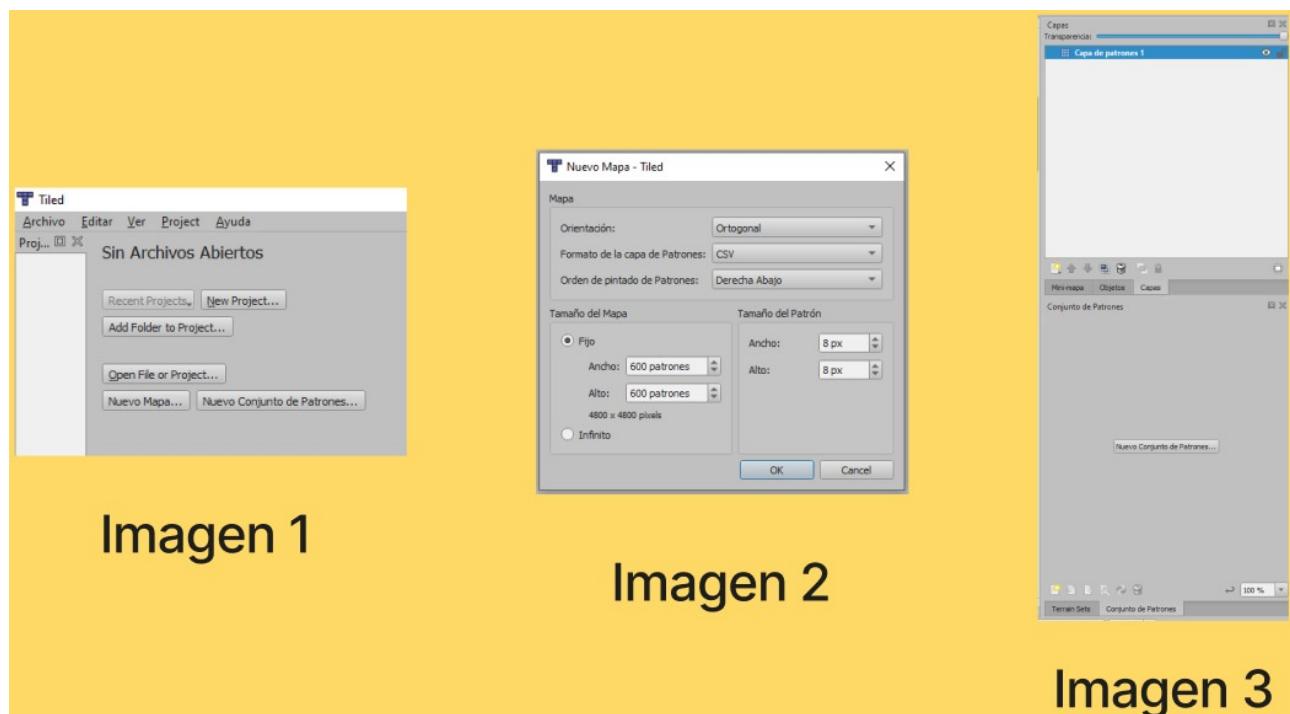
Ya hemos configurado la escena del menú. Sin embargo, si le damos a ‘Jugar’, nos encontraremos con que la escena principal está vacía. Esto se debe a que no hemos creado el mapa. En este apartado se explicará como hacer uso de la herramienta Tiled, con la que podremos crear crear y

exportar un mapa propio.

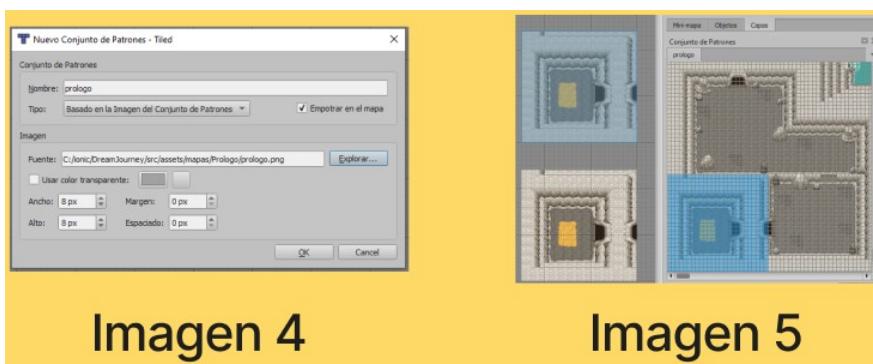
Tiled es un editor 2D de código abierto que se centra en la creación de mapas. Puede descargarse desde el siguiente [enlace](#).

A continuación, se mostrarán una serie de imágenes donde se explicará brevemente el funcionamiento de dicha herramienta.

- Una vez instalado Tiled, al abrirlo deberemos crear un nuevo mapa (Imagen 1 y 2) asignando las dimensiones deseadas para nuestro mapa. Una vez creado, se observará que a la derecha se encuentra situada la zona de capas, objetos y conjunto de patrones (Imagen 3). Esta zona será con la que se trabaje a la hora de añadir elementos.



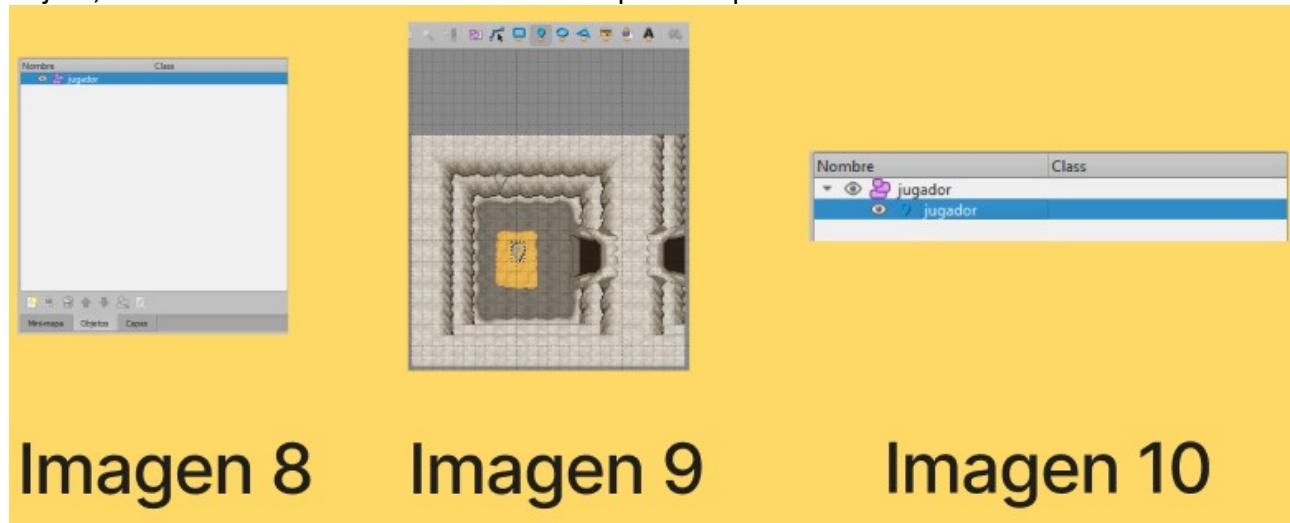
- Para hacer el mapa, necesitamos un conjunto de patrones, por lo que haciendo clic en 'Nuevo Conjunto de Patrones' (Imagen 3), se abrirá una ventana que nos permitirá seleccionar la imagen a usar como conjunto de patrones (Imagen 4). Una vez implementado, bastará con ir diseñando el mapa mediante los elementos del conjunto.



- A continuación, se va a analizar la estructura de capas (Imagen 6 y 7):



- Para añadir un objeto, se seleccionará la capa ‘Objetos’ y, acto seguido, seleccionar el ícono para añadir un nuevo objeto. Esto crea un objeto, pero para crear un elemento que corresponda a ese objeto, se deberán de usar las herramientas superiores para ubicar dicho elemento.



- Por último, se llenará el mapa con los objetos que el usuario quiera implementar. En el caso de este proyecto, la estructura es la siguiente (Imagen 11):



- Por último, se deberá exportar el mapa. Para ello, se necesitarán dos archivos: el png que se ha usado como conjunto de patrones y el mapa en formato json.

Esto puede realizarse desde Archivo → Exportar como imagen y Archivo → Exportar como (en este último, debemos asegurarnos de darle la extensión .json). Se recomienda que el archivo json tenga el mismo nombre que el conjunto de patrones (Imagen 12).



Una vez finalizado el mapa, resta introducirlo en la carpeta de assets y añadir el correspondiente código en la escena 'cargadatos.ts'.

➤ **Implementación de mapa y objetos.**

Una vez se ha completado el mapa y se han añadido todos los objetos que se van a querer usar, es momento de implementar todo eso dentro del juego. Para ello, una vez cargado tanto el mapa, como los diferentes recursos que complementan a este, se procederá al uso de estos en la escena principal. Esto se realizará en el siguiente código:

```
 1  this.mapaNivel = this.make.tilemap({ key: Constantes.MAPAS.PROLOGO.TILEMAPJSON, tileSize: 8, tileHeight: 8 });
 2  this.physics.world.bounds.setTo(0, 0, this.mapaNivel.widthInPixels, this.mapaNivel.heightInPixels);
 3  this.mapaTileset = this.mapaNivel.addTilesetImage(Constantes.MAPAS.TILESET);
 4
 5  this.capasueloMapaNivel = this.mapaNivel.createLayer(Constantes.MAPAS.PROLOGO.CAPACOLISIONES, this.mapaTileset);
 6  this.capaMapaNivel = this.mapaNivel.createLayer(Constantes.MAPAS.PROLOGO.CAPAMAPEADO, this.mapaTileset);
 7  this.capasuperiorMapaNivel = this.mapaNivel.createLayer(Constantes.MAPAS.PROLOGO.CAPASUPERIOR, this.mapaTileset);
 8  this.capasuperiorMapaNivel.setDepth(9);
 9  this.capasueloMapaNivel.setCollisionByExclusion([-1]);
```

Las primeras líneas de código cargan y añaden el Tilemap y Tileset, respectivamente, definiendo entre medias los límites del mundo. Al hacer esto, se debe asegurar que el valor que indicamos en el método 'addTilesetImage' corresponda con el nombre del conjunto de patrones que hemos usado (en este caso 'prologo').

Acto seguido, se crean las distintas capas del mapa, teniendo en cuenta que el valor que le

pasemos en el método ‘createLayer’ sea el correspondiente al nombre de la capa en Tiled.

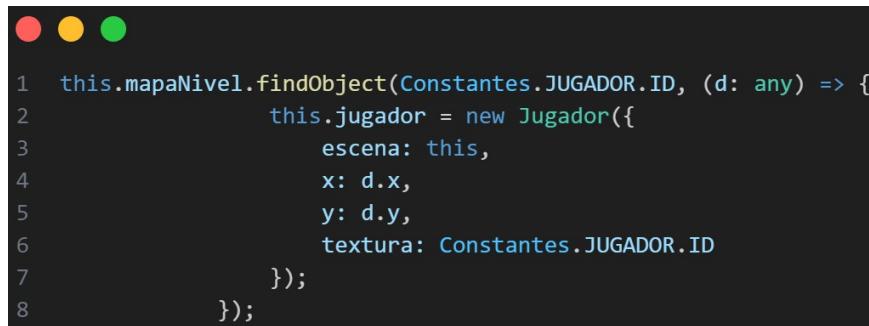
Una vez las capas están añadidas, debemos darle las propiedades que las diferencian entre sí. Para ello, a la capa superior le daremos un alto con valor 9, indicando que será la capa o elemento que esté posicionado por encima de todos los elementos, provocando así el efecto buscado a la hora de desplazarse entre zonas.

Por otro lado, para que la capa de colisiones cumpla su función, se deberá indicar en el método ‘setCollisionByExclusion’ dándole el valor [-1] en el parámetro ‘indexes’.

Con esto configurado, el mapa ya estaría implementado y funcionando de la manera correcta. Aun así, todavía falta por añadir todos los objetos que hemos añadido previamente.

Para ello, se muestran dos ejemplos:

- En el primer ejemplo, se va a añadir al Jugador, ya que es el único objeto en Tiled que tiene un elemento. Su estructura sería la siguiente:



```

1  this.mapaNivel.findObject(Constantes.JUGADOR.ID, (d: any) => {
2      this.jugador = new Jugador({
3          escena: this,
4          x: d.x,
5          y: d.y,
6          textura: Constantes.JUGADOR.ID
7      });
8  });

```

Básicamente, se está pidiendo al mapa que encuentre el objeto que corresponda con el valor que le hemos asignado y, que al encontrarlo, asigne a la propiedad ‘this.jugador’ un nuevo objeto de tipo Jugador. Hablaremos de esta clase en el apartado de movimiento del jugador.

- En el segundo ejemplo, tenemos el caso más común, que es añadir un conjunto de objetos del mismo tipo. Para ello, será necesario el siguiente código:



```

1  var capaObjetos = this.mapaNivel.getObjectLayer('zona');
2      capaObjetos.objects.forEach( (objeto) => {
3          this.datosZonas[objeto.name] = {
4              x: objeto.x,
5              y: objeto.y,
6              width: objeto.width,
7              height: objeto.height
8          };
9      });

```

El proceso es simple: primero, se busca de entre todos los objetos, el objeto en cuestión, almacenando este resultado en una variable. Acto seguido, se hará un bucle foreach para cada

uno de los elementos que contiene este objeto. Dentro de este foreach, la estructura dependerá del tipo de objeto que se esté buscando.

Más adelante se verá qué lógica tendrán dichos objetos.

- ◆ **Control de cámara.**

En la mayoría de juegos, la cámara suele estar enfocada en el jugador, siendo este el punto central de la cámara. Para realizar esto en Phaser, se debe añadir el siguiente código en el mismo bloque de código donde se añade el mapa y sus objetos:



```
1 this.cameras.main.zoom=5;
2 this.cameras.main.startFollow(this.jugador);
```

Ahora mismo, el zoom de la cámara no está justificado. Esto será explicado cuando se hable del cambio de zona.

- ◆ **Moviendo al jugador.**

Se ha añadido al personaje, pero este carece de movimiento. Primero, se añadirá en el update de la escena principal el siguiente código:



```
1 this.jugador.update();
```

Esto hará que cada vez que se haga un update en la escena principal, se llame también al update del jugador. Pero, ¿qué ocurre en dicho update? Antes de entrar en dicho apartado, se deberá indicar al jugador cuales serán sus controles de movimiento. Para ello, usaremos las flechas direccionales. El código a implementar sería el siguiente:



```
1 this.cursores = this.escena.input.keyboard.createCursorKeys();
```

Una vez se han añadido las teclas de movimiento al jugador, es hora de configurar dichas teclas. Para ello, se debe añadir el siguiente código en el update de la clase Jugador:

```
1  if (this.cursores.left.isDown && this.jugador_mueve) {  
2      this.movimiento = true;  
3      this.setVelocityY(0);  
4      this.direccionEsperar = 4;  
5      this.anims.play(Constantes.JUGADOR.ANIMACION.ANDARIZQUIERDA, true);  
6  } else if (this.cursores.right.isDown && this.jugador_mueve) {  
7      this.movimiento = true;  
8      this.setVelocityY(0);  
9      this.flipX = false;  
10     this.direccionEsperar = 2;  
11     this.anims.play(Constantes.JUGADOR.ANIMACION.ANDAR_DERECHA, true);  
12 } else if (this.cursores.down.isDown && this.jugador_mueve) {  
13     this.movimiento = true;  
14     this.setVelocityX(0);  
15     this.direccionEsperar = 3;  
16     this.anims.play(Constantes.JUGADOR.ANIMACION.ANDAR_ABAJO, true);  
17 } else if (this.cursores.up.isDown && this.jugador_mueve) {  
18     this.movimiento = true;  
19     this.setVelocityX(0);  
20     this.direccionEsperar = 1;  
21     this.anims.play(Constantes.JUGADOR.ANIMACION.ANDAR_ARRIBA, true);  
22 } else {  
23     this.movimiento = false;  
24     this.setVelocityX(0);  
25     this.setVelocityY(0);  
26     switch (this.direccionEsperar) {  
27         case 1:  
28             this.anims.play(Constantes.JUGADOR.ANIMACION.ESPERARA, true);  
29             break;  
30         case 2:  
31             this.anims.play(Constantes.JUGADOR.ANIMACION.ESPERARD, true);  
32             break;  
33         case 3:  
34             this.anims.play(Constantes.JUGADOR.ANIMACION.ESPERAR, true);  
35             break;  
36         case 4:  
37             this.anims.play(Constantes.JUGADOR.ANIMACION.ESPERARI, true);  
38             break;  
39         default:  
40             break;  
41     }  
42 }
```

La explicación es muy simple: cuando el update de la clase Jugador es llamado, se comprueba que, en caso de que alguna de las teclas de movimiento está presionada, se realizará el movimiento en la dirección correspondiente junto a su debida animación. En caso de que no se esté presionando

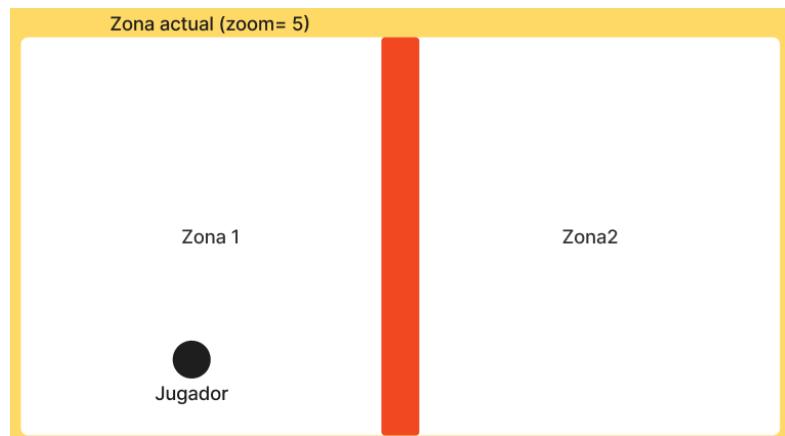
ninguna tecla, el jugador estará en reposo, por lo cuál no tendrá velocidad alguna. Como añadido personal, he configurado el movimiento de modo que si el jugador camina hacia abajo y se para, la animación de reposo será la del jugador mirando hacia abajo.

Con esto, el jugador ya es capaz de moverse libremente por el mapa. Ahora bien, ¿Cómo se puede configurar la cámara para que la transición entre zonas o habitaciones sea más detallada? Este tema será tratado en el siguiente apartado.

◆ Cambio de zonas.

El siguiente apartado se podría considerar específico de este tipo de juegos. Lo que se busca es hacer animación que se produzca cuando el jugador pase por una puerta o zona. Aunque bien es cierto que dicha animación no es algo obligatorio, sí que aporta cierta personalidad al juego.

Para llevar a cabo esta mecánica/animación, se pueden plantear múltiples soluciones. A continuación, se mostrará la solución aplicada:



Partiendo de la imagen mostrada, se explicará el proceso:

- Cuando se añaden los objetos del mapa al comienzo de la escena, el conjunto de objetos de tipo 'zona' representan las habitaciones por donde el jugador se moverá.

- Una vez añadidos estos objetos, se establece cual es la zona inicial, que en este caso, es la zona donde comienza el jugador.

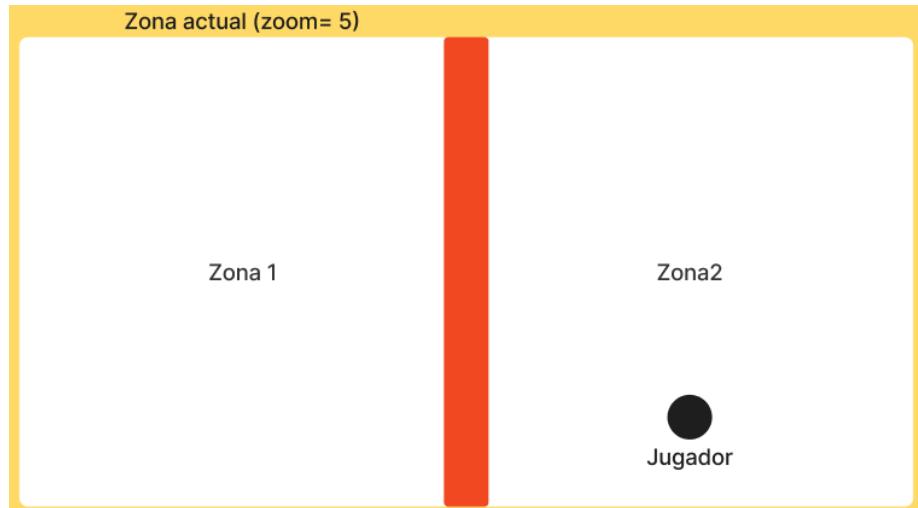
- Cada zona tiene los atributos 'x', 'y', 'width' y 'height', que serán establecidos como los límites de la cámara con el siguiente código:

```
this.cameras.main.setBounds(x, y, width, height);
```

- Cuando se ha hablado del control de la cámara, se ha comentado que el zoom se justificaría en este zona. El motivo por el cuál se aplica zoom a la cámara es para evitar que parte de las zonas contiguas a la zona actual sean vistas. Por otro lado, gracias al zoom de la

cámara se puede ver con mayor claridad el mapa, ya que este, al ser técnicamente grande, si se dejase el zoom por defecto, el resultado sería una vista muy generalizada de todo el mapa, dejando los objetos repartidos por el mapa apenas visibles.

- Ahora bien, ¿qué pasa cuando el jugador cambia de zona?. Veamos la imagen de nuevo:



Cuando el jugador accede a una zona distinta a la actual, se llama al método 'getRoom' de la clase Jugador. Este código es llamado constantemente desde el update de esta clase. Su función es la siguiente:

- Para cada uno de los objetos de tipo zona que obtiene de la escena principal, comprueba si el cuerpo del jugador se encuentra dentro de una zona u otra.

- Cuando se encuentra la zona donde está el jugador, se comprueba si esa zona es la actual o no. En caso de ser una zona distinta a la actual, se activará la condición que permite el cambio de límites de la cámara en la escena principal.

El código que hace estas comprobaciones es el siguiente:

```

1  getRoom() {
2      let num_zona;
3
4      for(let zona in this.escena.datosZonas){
5          let izquierda = this.escena.datosZonas[zona].x!;
6          let derecha = this.escena.datosZonas[zona].x! + this.escena.datosZonas[zona].width!;
7          let top = this.escena.datosZonas[zona].y!;
8          let bottom = this.escena.datosZonas[zona].y! + this.escena.datosZonas[zona].height!;
9
10         //Se comprueba si el jugador se encuentra en los límites de la zona
11         if (this.x > izquierda && this.x < derecha && this.y > top && this.y < bottom){
12             num_zona = zona;
13         }
14     }
15
16     //Se actualiza las variables de zona
17     if (num_zona != this.zona_Actual){
18         this.zona_Anterior = this.zona_Actual;
19         this.zona_Actual = num_zona;
20         this.cambiaZona = true;
21     } else{
22         this.cambiaZona = false;
23     }
24 }
```

- Cuando la propiedad ‘this.cambiaZona’ es true, se ejecuta el siguiente código en el update de la escena principal:

```

1  if (this.jugador.cambiaZona) {
2      this.cameras.main.fadeOut(250, 0, 0, 0, (camera: any, progreso: number) => {
3          this.jugador.jugador_mueve = false;
4          if (progreso === 1) {
5              //Una vez se completa el fadeOut, se cambia las dimensiones de la cámara
6              //con respecto a las dimensiones de la nueva zona
7              let cam_x = this.datosZonas[this.jugador.zona_Actual.toString()].x!;
8              let cam_y = this.datosZonas[this.jugador.zona_Actual.toString()].y!;
9              let cam_width = this.datosZonas[this.jugador.zona_Actual.toString()].width!;
10             let cam_height = this.datosZonas[this.jugador.zona_Actual.toString()].height!;
11
12             this.cameras.main.setBounds(cam_x, cam_y, cam_width, cam_height, true);
13
14             if (this.jugador.zona_Actual == 1){
15                 this.cameras.main.zoom = 5;
16             } else{
17                 this.cameras.main.zoom = 3.5;
18             }
19
20             //Se realiza un fadeIn con la cámara posicionada en la nueva zona
21             this.cameras.main.fadeIn(500, 0, 0, 0, (camera: any, progress: number) => {
22                 if (progress === 1) {
23                     this.jugador.jugador_mueve = true;
24                     this.nuevaHabitacion(this.jugador.zona_Actual);
25                 }
26             }, this);
27         }
28     }, this);
29 }

```

Básicamente, cuando el jugador cruza los límites de la zona actual, se impide su movimiento y se hace un fundido en negro. Una vez se ha completado dicho fundido, se ajustan los límites de la cámara con los valores correspondientes de la nueva zona. Además, si la zona actual es distinta a 1, se aplicará un menor zoom.

Esto se debe a que la zona 1 tiene unas dimensiones menores al resto, por lo que en esta se deberá aplicar un mayor zoom para evitar el error comentado de que se muestren partes de las salas contiguas.

- Por último, se devuelve la capacidad de movimiento al jugador y se llama al método ‘nuevaHabitación’. Este método es opcional, ya que su función es la de controlar, dependiendo de la zona, el tipo de enemigos que aparecerán en combate y la música que debe sonar en cada caso.

◆ Movimiento en escaleras.

Está mecánica puede considerarse la más fácil de implementar. Simplemente debemos añadir el siguiente código al bloque donde se añade el mapa a la escena, teniendo en cuenta que previamente se han obtenido los objetos del tipo escalera:

```

1  this.physics.add.overlap(this.jugador, this.escalerasMano, this.jugador.subeEscalera as ArcadePhysicsCallback, undefined, this);

```

Con este código se está diciendo que, cuando el jugador colisione con un objeto de tipo escalera, se llame al método ‘subeEscalera’ de la clase Jugador. Este método hará lo siguiente:

- Primero, habrá que imaginar una situación, como la que se muestra en la siguiente imagen:



La idea es que, cuando el jugador colisione con ‘Escalera 1’, la posición de este cambie y se sitúe al lado de ‘Escalera 2’.

- Para lograr esto, lo que se hace es obtener la propiedades ‘salida’ y ‘dirección’ que tienen estos objetos. Gracias a esto, se pueden conectar dos objetos entre sí.
- Por último, dependiendo de la propiedad ‘dirección’, se posicionará al jugador en una posición u otra de forma que el jugador, al usar la escalera, aparezca al lado de la otra escalera, evitando así posibles errores.

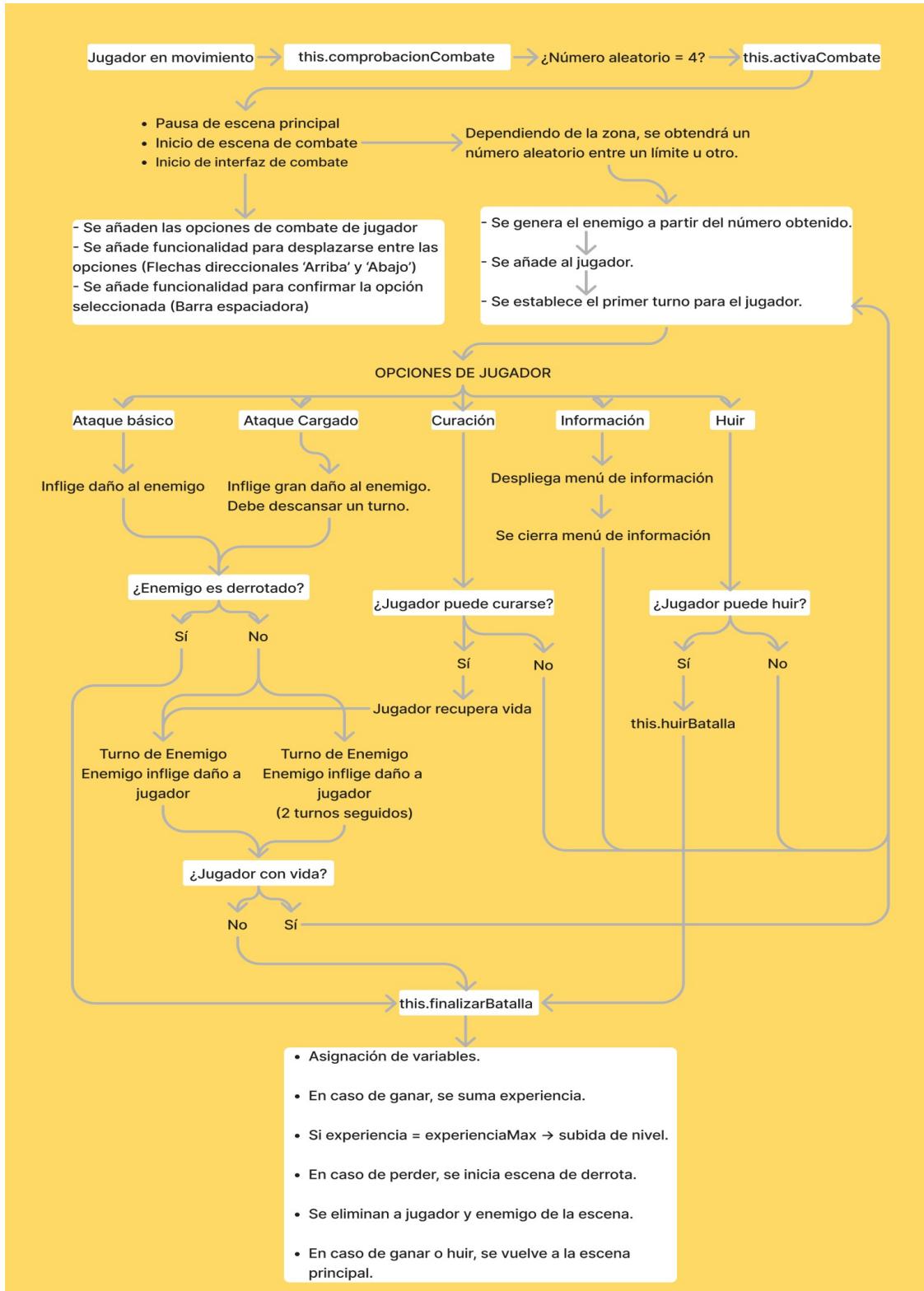
El código del método ‘subeEscalera’ es el siguiente:

```

● ● ●
1  public subeEscalera(jugador: Jugador, objeto: Phaser.Physics.Arcade.Sprite): void{
2      const siguienteZona = jugador.escena.datosEscalerasMano[objeto.name].salida;
3
4      if (siguienteZona != '---'){
5          const direccionSalida = jugador.escena.datosEscalerasMano[objeto.name].direccion;
6
7          const posX = jugador.escena.datosEscalerasMano[''+siguienteZona].x!;
8          const posY = jugador.escena.datosEscalerasMano[''+siguienteZona].y!;
9
10         if (direccionSalida == 'up') {
11             jugador.escena.jugador.x = posX+7;
12             jugador.escena.jugador.y = posY-20;
13         } else if (direccionSalida == 'down') {
14             jugador.escena.jugador.x = posX+10;
15             jugador.escena.jugador.y = posY+30;
16         }
17     }
18 }
```

➤ Escena de combate.

Debido a la longitud del código referente a la escena de combate y su correspondiente lógica, se va a proceder a mostrar un diagrama de su funcionamiento:



➤ Utilidad de objetos.

Anteriormente, se ha visto como, a la hora de implementar el mapa y los diversos objetos que lo conforman, la gran mayoría de estos objetos no han recibido una mecánica. Es por ello que en los siguientes apartados se hablará de la lógica a implementar para cada uno de los objetos del mapa que no han sido explicados con anterioridad. Se comenzará hablando de los objetos.

Dichos objetos serán los que ofrezcan al jugador un aumento de ataque u objetos curativos. Para llevar a cabo esta mecánica, deberemos de implementar el siguiente código en el update de la escena principal:

```
● ● ●
1 this.input.keyboard.on('keydown-SPACE', () => {
2     if (this.teclaPulsada == true){
3         this.teclaPulsada = false;
4         this.time.addEvent({delay: 500, callback: this.comprobarProximidadObjetos, callbackScope: this});
5     }
6 });

```

Este código controla que, cada vez que el usuario pulsa la barra espaciadora (tecla para interactuar), se llame al método ‘comprobarProximidadObjetos’, no sin antes impedir que este código se ejecute más de una vez seguida en un breve periodo de tiempo (uso de booleano para impedir la ejecución repetida de código, debido a que Phaser refresca en milisegundos).

El método ‘comprobarProximidadObjetos’ contiene el siguiente código:

```
● ● ●
1 comprobarProximidadObjetos(){
2     for (let i = 0; i < this.conjuntoObjetos.length; i++) {
3         if (this.conjuntoObjetos[i].body != undefined){
4             if ((Math.abs(this.jugador.body.x - this.conjuntoObjetos[i].body.x)) <= 20 && (Math.abs(this.jugador.body.y - this.conjuntoObjetos[i].body.y)) <= 20) {
5                 Prologo.efectos = this.sound.add('save_effect', {volume: MenuOptions.effectsSound/100});
6                 Prologo.efectos.play({
7                     loop: false
8                 });
9                 this.jugador.jugador_mueve = false;
10                this.conjuntoObjetos[i].recogerObjeto();
11                CargaDatos.objetosRecogidos.push(this.conjuntoObjetos[i].numero);
12                this.conjuntoObjetos[i].visible = false;
13                this.conjuntoObjetos[i].destroy();
14
15                var eleccion = '';
16                var doble_eleccion = false;
17
18                if (this.conjuntoObjetos[i].tipo == 'ataque'){
19                    eleccion = Constantes.MISCELANEOS.SUBIRATAQUE;
20                }
21
22                if (this.conjuntoObjetos[i].tipo == 'cura'){
23                    eleccion = Constantes.MISCELANEOS.SUBIRCURAS;
24                }
25
26                if (this.conjuntoObjetos[i].tipo == 'ambos' || this.conjuntoObjetos[i].tipo == 'ambosx2'){
27                    eleccion = Constantes.MISCELANEOS.SUBIRCURAS;
28                    doble_eleccion = true;
29                }
30            }
31        }
32    }
}

```

Lo que se hace es comprobar, para cada uno de los objetos de este tipo, si la distancia que hay entre el jugador y dicho objeto es mínima. De ser así, el código se ejecutará de la siguiente manera:

- Se lanzará un efecto de sonido para indicar al jugador que ha recogido el objeto.

- Se impedirá el movimiento del jugador.
- Se llamará al método ‘recogerObjeto’ del objeto interactuado. Este método aumentará las propiedades estáticas ‘Cargadatos.jugadorAtaque’ o ‘Cargadatos.jugadorCuración’ dependiendo de la propiedad ‘tipo’ del objeto.
- Por último, este objeto se añadirá al array de ‘objetosRecogidos’ y será eliminado.
 - Una vez realizado este proceso, dependiendo de la propiedad ‘tipo’ del objeto, se mostrará una imagen u otra encima de la cabeza del jugador, indicando que ha obtenido.
 - Para evitar errores de código, se deberá modificar el update de este tipo de objetos en el update de la escena principal, modificando el código de la siguiente manera:

```
 1  for (let i = 0; i < this.conjuntoObjetos.length; i++) {  
 2      if (this.conjuntoObjetos[i].body != undefined && this.conjuntoObjetos[i].visible == true){  
 3          this.conjuntoObjetos[i].update();  
 4      }  
 5  }
```

➤ Añadiendo NPCS.

Siguiendo con la lógica de los objetos, es el turno de los NPCS. Estos tendrán dos tipos de mecánicas: por un lado, tendrán un patrón de movimiento y, por el otro lado, al colisionar contra ellos, dará comienzo un combate contra dicho NPC.

Se comenzará hablando del patrón de movimiento. Para ello, colocaremos en el update de la escena principal el siguiente código:

```
 1  for (let i = 0; i < this.conjuntoNPCs.length; i++) {  
 2      this.conjuntoNPCs[i].update();  
 3  }
```

Ahora bien, dentro de la clase NPC, dependiendo del valor del parámetro identificador, se asignarán el tipo de patrón que puede tener un NPC. Dichos patrones son: arriba/abajo, derecha/izquierda y reposo.

A continuación, se crearán las animaciones necesarias (1 animación por cada dirección + animación de reposo).

Por último, el código que controle las animaciones estará situado en el update y será el siguiente:



```
1 switch (this.patron){
2     case 'Derecha/Izquierda':
3         if (this.body.velocity.x == 0){
4             this.movimientoEnemigo(Phaser.Math.Between(0, 1) ? 'Derecha' : 'Izquierda', this);
5         }
6         if (this.body.touching.right) {
7             this.movimientoEnemigo('Izquierda', this);
8         } else if (this.body.touching.left) {
9             this.movimientoEnemigo('Derecha', this);
10        }
11        break;
12    case 'Arriba/Abajo':
13        if (this.body.velocity.y == 0){
14            this.movimientoEnemigo(Phaser.Math.Between(0, 1) ? 'Arriba' : 'Abajo', this);
15        }
16        if (this.body.touching.up) {
17            this.movimientoEnemigo('Abajo', this);
18        } else if (this.body.touching.down) {
19            this.movimientoEnemigo('Arriba', this);
20        }
21        break;
22    default:
23        this.anims.play('npcAbajo'+this.identi, true);
24        break;
25    }
```

Exceptuando el case default, los otros dos case actúan de la misma manera, solo que en distintos ejes:

- Si el NPC no tiene velocidad, se decidirá aleatoriamente en qué dirección se moverá primero y se llamará al método ‘movimientoEnemigo’ para comenzar su recorrido.

- Si el NPC en su recorrido choca, se llamará al método ‘movimientoEnemigo’ para cambiar el sentido del recorrido.

El código que contiene el método ‘movimientoEnemigo’ es el siguiente:



```

1  movimientoEnemigo(direccion: string, npc: any){
2      switch (this.patron){
3          case 'Derecha/Izquierda':
4              if (direccion === 'Derecha'){
5                  npc.body.setVelocityX(25);
6                  if (this.identi === 'overworld09'){
7                      npc.anims.play('npcAbajo'+this.identi, true);
8                  } else {
9                      this.anims.play('npcDer'+this.identi, true);
10                 }
11             } else if (direccion === 'Izquierda'){
12                 npc.body.setVelocityX(25*(-1));
13                 if (this.identi === 'overworld09'){
14                     npc.anims.play('npcAbajo'+this.identi, true);
15                 } else {
16                     this.anims.play('npcIzq'+this.identi, true);
17                 }
18             }
19             break;
20         case 'Arriba/Abajo':
21             if (direccion === 'Arriba'){
22                 npc.body.setVelocityY(25*(-1));
23                 npc.anims.play('npcArr'+this.identi, true);
24             } else if (direccion === 'Abajo'){
25                 npc.body.setVelocityY(25);
26                 npc.anims.play('npcAbajo'+this.identi, true);
27             }
28             break;
29         }
30     }

```

Este método se encarga de dar la dirección en la que se moverá el NPC dependiendo del patrón asignado. Esta estructurado de la misma manera que el update, es decir, usan el mismo código para alternando los ejes y las animaciones que reproducen. En el caso del NPC 'overworld09', al ser un cangrejo, se decidió darle el patrón Derecha/Izquierda pero que, en todo momento, su animación fuese la misma, simulando el movimiento real de un cangrejo.

Una vez terminada la parte de movimiento de los NPCS, se va a hablar de la otra lógica de estos: el combate por colisión. Para ello, el código correspondiente a la colisiones de este tipo de objetos será el siguiente:



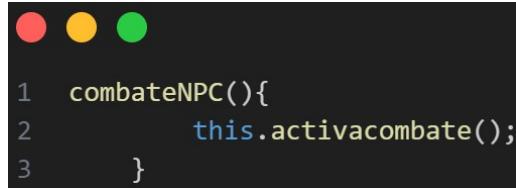
```

1  for (let i = 0; i < this.conjuntoNPCs.length; i++) {
2      this.physics.add.collider(this.conjuntoNPCs[i], this.capasueloMapaNivel);
3      this.physics.add.overlap(this.jugador, this.conjuntoNPCs[i], () => {
4          if (CargaDatos.npcCombate == false && CargaDatos.jugadorPeleaNpc == false){
5              CargaDatos.npcCombate = true;
6              CargaDatos.peleaNpc = this.conjuntoNPCs[i].id;
7              if (this.conjuntoNPCs[i].id == 'boss_fuego'){
8                  CargaDatos.jefeZonaDer = true;
9              }
10             if (this.conjuntoNPCs[i].id == 'boss_hielo'){
11                 CargaDatos.jefeZonaIzq = true;
12             }
13             if (this.conjuntoNPCs[i].id == 'boss_oculto'){
14                 CargaDatos.jefeOculto = true;
15             }
16             this.time.addEvent({delay: 1000, callback: this.combateNPC, callbackScope: this});
17         }
18     });
19 }

```

Este código se puede resumir de la siguiente manera:

- En el momento que el jugador colisiones con un NPC, este asignará a la propiedad ‘CargaDatos.peleaNpc’ el valor de la propiedad ‘id’ del NPC colisionado. Dependiendo del valor de ‘id’, se podrán activar ciertos eventos o no. Al final, se llamará al método ‘combateNPC’, el cuál contiene el siguiente código:



```

1  combateNPC(){
2      this.activacombate();
3  }

```

El llamado de combate se hace de esta manera para evitar que sea llamado múltiples veces en menos de un segundo, pudiendo generar así una cadena de combates infinitos.

Cuando se activa el combate y se llama a la escena de ‘batalla.ts’, se comprueba si la propiedad ‘CargaDatos.peleaNPC’ es distinto a “nada”. De ser así, se el enemigo que se genere para el combate tendrá como identificador el obtenido en ‘CargaDatos.peleaNPC’, cargando el sprite y los atributos correspondientes a este enemigo.

En caso de que el jugador derrote al NPC en combate, la propiedades ‘CargaDatos.borrarNPC’ y ‘CargaDatos.jugadorPeleaNpc’ pasarán a ser true y, una vez regresado a la escena principal, en el update se ejecutarán los siguientes bloques de código:



```

1  if (CargaDatos.borrarNpc == true){
2      CargaDatos.borrarNpc = false;
3      for (let i = 0; i < this.conjuntoNPCs.length; i++) {
4          if (this.conjuntoNPCs[i].id == CargaDatos.peleaNpc){
5              CargaDatos.npcDerrotados.push(this.conjuntoNPCs[i].identi);
6              this.conjuntoNPCs[i].destroy();
7          }
8      }
9      CargaDatos.peleaNpc = 'nada';
10 }
11
12 if (CargaDatos.jugadorPeleaNpc == true){
13     this.time.addEvent({delay: 5000, callback: () => {
14         CargaDatos.jugadorPeleaNpc = false;
15         CargaDatos.npcCombate = false;
16    }});
17 }

```

De esta manera, se elimina al NPC ya derrotado del mapa, a la vez que lo añadimos a la array de ‘npcDerrotados’. Por último, se deberá modificar el update de los NPC para evitar posibles errores, siendo este el nuevo código:

```
1  for (let i = 0; i < this.conjuntoNPCs.length; i++) {  
2      if (this.conjuntoNPCs[i].body != undefined && this.conjuntoNPCs[i].visible == true){  
3          this.conjuntoNPCs[i].update();  
4      }  
5  }
```

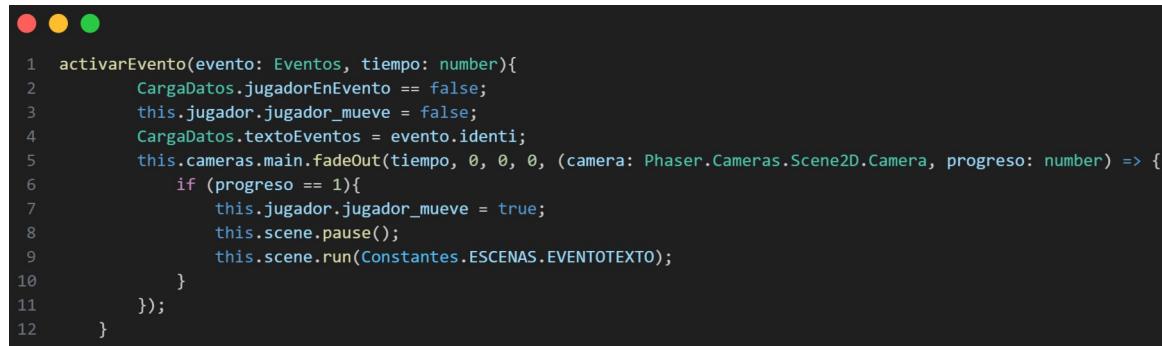
➤ Implementando los eventos de historia.

Para finalizar, se explicará la lógica que hay detrás de los eventos de historia. Las colisiones con respecto a este tipo de objetos es la siguiente:

```
1  for (let i = 0; i < this.conjuntoEventos.length; i++) {  
2      this.physics.add.overlap(this.jugador, this.conjuntoEventos[i], () => {  
3          if (CargaDatos.eventoActivo == false){  
4              if (CargaDatos.jugadorEnEvento == true){  
5                  if (this.conjuntoEventos[i].identi == 'combate'){  
6                      if (CargaDatos.combateInicial == true){  
7                          CargaDatos.combateInicial = false;  
8                          CargaDatos.peleaNpc = 'inicio';  
9                          CargaDatos.eventoActivo = true;  
10                         this.activarcombate();  
11                     }  
12                 }else {  
13                     CargaDatos.eventoActivo = true;  
14                     this.activarEvento(this.conjuntoEventos[i], 2000);  
15                 }  
16             }  
17         }  
18     });  
19 }
```

De este modo, cuando el jugador colisione con un objeto de este tipo, en caso de que las condiciones booleanas permitan la continuidad de código (se hace uso de doble booleano debido a que Phaser no tiene la opción de refrescarse cada segundo, estableciendo un refresco cada milí segundo), se comprobará el identificador del objeto con el que el jugador está colisionando.

Si el jugador ha colisionado con el evento de combate (combate forzado para introducir la escena de batalla), tendrá lugar el combate inicial. En caso de que el identificador no sea el de combate, se llamará al método ‘activarEvento’, el cuál está formado por el siguiente código:



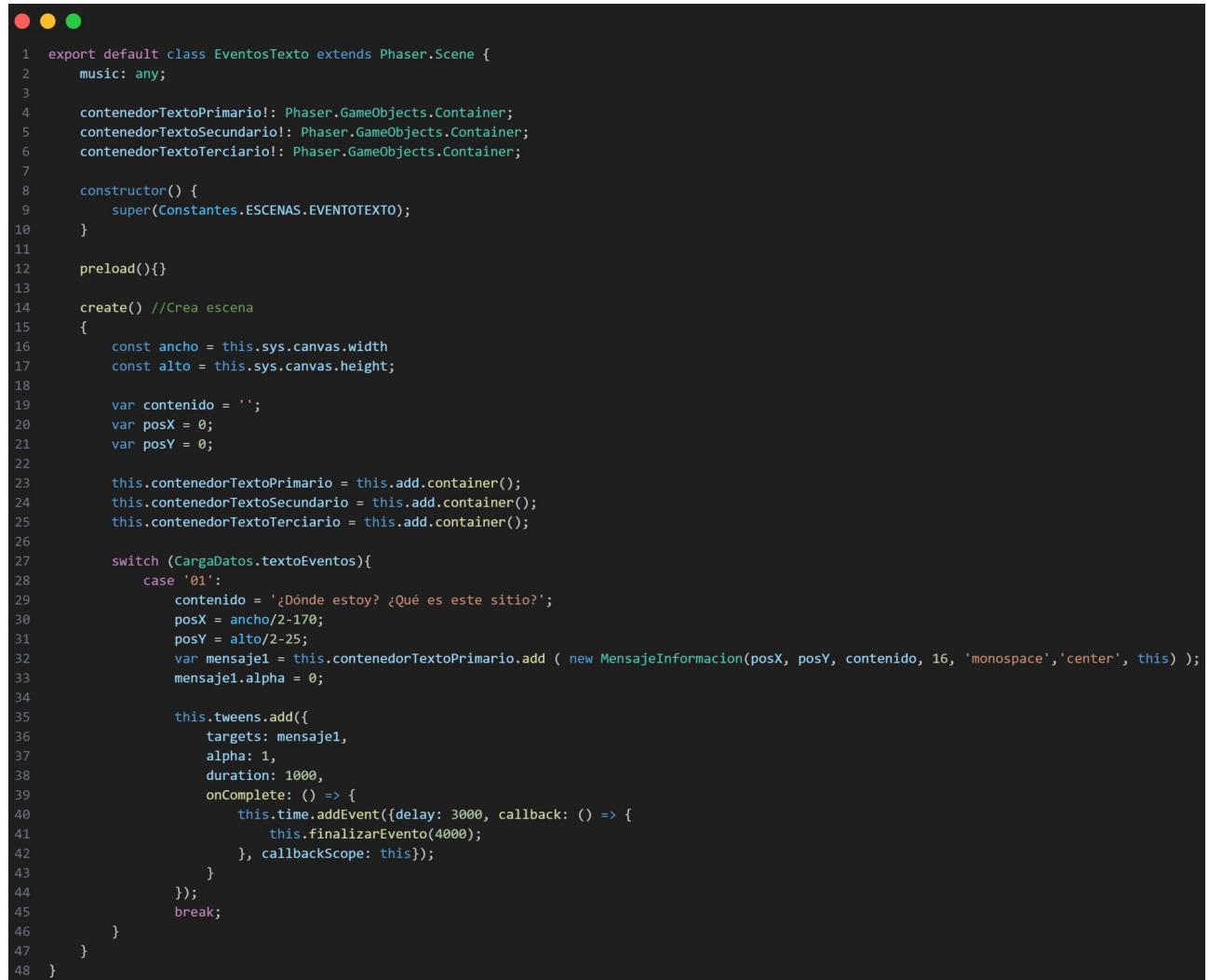
```

1  activarEvento(evento: Eventos, tiempo: number){
2      CargaDatos.jugadorEnEvento == false;
3      this.jugador.jugador_mueve = false;
4      CargaDatos.textoEventos = evento.identi;
5      this.cameras.main.fadeOut(tiempo, 0, 0, 0, (camera: Phaser.Cameras.Scene2D.Camera, progreso: number) => {
6          if (progreso == 1){
7              this.jugador.jugador_mueve = true;
8              this.scene.pause();
9              this.scene.run(Constantes.ESCENAS.EVENTOTEXTO);
10         }
11     });
12 }

```

Dicho método se encarga de lo siguiente: Por un lado, impide el movimiento del jugador, a la vez que se asigna a la propiedad estática ‘CargaDatos.textoEventos’ el identificador del objeto colisionado. A continuación, se produce un función en negro de cámara que, cuando es completado, permite el movimiento del jugador y da inicio a la escena ‘eventosTexto.ts’, dejando en pausa la escena principal.

La escena ‘eventosTexto.ts’ tendrá siempre la misma lógica, la cuál puede verse en la siguiente imagen:



```

1  export default class EventosTexto extends Phaser.Scene {
2      music: any;
3
4      contenedorTextoPrimario!: Phaser.GameObjects.Container;
5      contenedorTextoSecundario!: Phaser.GameObjects.Container;
6      contenedorTextoTerciario!: Phaser.GameObjects.Container;
7
8      constructor() {
9          super(Constantes.ESCENAS.EVENTOTEXTO);
10     }
11
12     preload(){}
13
14     create() //Crea escena
15     {
16         const ancho = this.sys.canvas.width;
17         const alto = this.sys.canvas.height;
18
19         var contenido = '';
20         var posX = 0;
21         var posY = 0;
22
23         this.contenedorTextoPrimario = this.add.container();
24         this.contenedorTextoSecundario = this.add.container();
25         this.contenedorTextoTerciario = this.add.container();
26
27         switch (CargaDatos.textoEventos){
28             case '01':
29                 contenido = '¿Dónde estoy? ¿Qué es este sitio?';
30                 posX = ancho/2-170;
31                 posY = alto/2-25;
32                 var mensaje1 = this.contenedorTextoPrimario.add ( new MensajeInformacion(posX, posY, contenido, 16, 'monospace','center', this) );
33                 mensaje1.alpha = 0;
34
35                 this.tweens.add({
36                     targets: mensaje1,
37                     alpha: 1,
38                     duration: 1000,
39                     onComplete: () => {
40                         this.time.addEvent({delay: 3000, callback: () => {
41                             this.finalizarEvento(4000);
42                         }, callbackScope: this});
43                     }
44                 });
45                 break;
46             }
47         }
48     }

```

En este caso, se puede observar que es lo que sucede cuando se activa el objeto de tipo evento con identificador '01', en el cuál sólo se muestra un diálogo y, pasado un tiempo, este termina y llama al método 'finalizarEvento'. Dependiendo del tipo de evento, este puede llamar a un método o a otro, teniendo 3 posibilidades: 'finalizarEvento', 'finalizarJuego', 'finalizarGuardado'. El código de dichos métodos es el siguiente:

```

1 finalizarGuardado(){
2     this.time.addEvent({delay: 1000, callback: () => {
3         this.cameras.main.fadeOut(500, 0, 0, 0, (camera: Phaser.Cameras.Scene2D.Camera, progreso: number) => {
4             if (progreso == 1){
5                 this.contenedorTextoPrimario.removeAll(true);
6                 this.contenedorTextoSecundario.removeAll(true);
7                 this.contenedorTextoTerciario.removeAll(true);
8                 CargaDatos.eventoFinalizado = true;
9                 this.scene.resume(Constantes.ESCENAS.PROLOGO);
10                this.scene.stop();
11            }
12        });
13    }, callbackScope: this});
14 }
15
16 finalizarEvento(delay: number){
17     this.time.addEvent({delay: delay, callback: () => {
18         this.cameras.main.fadeOut(2000, 0, 0, 0, (camera: Phaser.Cameras.Scene2D.Camera, progreso: number) => {
19             if (progreso == 1){
20                 this.skipear = false;
21                 this.contenedorTextoPrimario.removeAll(true);
22                 this.contenedorTextoSecundario.removeAll(true);
23                 this.contenedorTextoTerciario.removeAll(true);
24                 CargaDatos.eventoFinalizado = true;
25                 this.scene.resume(Constantes.ESCENAS.PROLOGO);
26                 this.scene.stop();
27             }
28         });
29    }, callbackScope: this});
30 }
31
32 finalizarJuego(){
33     this.time.addEvent({delay: 5000, callback: () => {
34         this.cameras.main.fadeOut(4000, 0, 0, 0, (camera: Phaser.Cameras.Scene2D.Camera, progreso: number) => {
35             if (progreso == 1){
36                 CargaDatos.jugadorFinaliza = true;
37                 this.scene.start(Constantes.ESCENAS.VICTORIA_DERROTA);
38                 this.scene.stop(Constantes.ESCENAS.PROLOGO);
39                 this.scene.stop();
40             }
41         });
42     }, callbackScope: this});
43 }

```

Exceptuando el método 'finalizarJuego', los otros dos métodos devuelven al jugador a la escena principal, donde, para finalizar con la lógica de los eventos, se deberán de analizar todos los objetos de tipo evento en busca del que se ha realizado, añadirlo a la array de 'eventosRealizados' y eliminar dicho evento del mapa. El código que realiza estos pasos se encuentra en el update de la escena principal y es el siguiente:



```

1 if (CargaDatos.eventoFinalizado == true){
2     CargaDatos.eventoFinalizado = false;
3     if (CargaDatos.textoEventos != 'guardar'){
4         CargaDatos.eventosRealizados.push(CargaDatos.textoEventos);
5         for (let i = 0; i < this.conjuntoEventos.length; i++) {
6             if (CargaDatos.textoEventos == this.conjuntoEventos[i].identi){
7                 this.conjuntoEventos[i].destroy();
8                 CargaDatos.textoEventos = '';
9                 CargaDatos.eventoActivo = false;
10                this.time.addEvent({delay: 2000, callback: () => {
11                    CargaDatos.jugadorEnEvento = true;
12                }});
13            }
14        }
15    }else {
16        CargaDatos.textoEventos = '';
17        CargaDatos.eventoActivo = false;
18    }
19    this.cameras.main.fadeIn(2000, 0, 0, 0);
20 }

```

Durante este proceso, se excluirá el caso en el que el evento que se ha realizado sea el de guardar partida (se hablará de este más adelante).

➤ Sistema de guardado.

El juego está prácticamente terminado, sólo resta implementar un elemento clave para cualquier juego: un sistema de guardado.

Para ello, se va a recurrir a la función **localStorage**, la cuál permite acceder al almacenamiento local y guardar los datos necesarios. A diferencia de la función **sessionStorage**, la cuál almacena los datos mientras la sesión este activa, la función **localStorage** guarda los datos de forma permanente.

A continuación, se verá como implementar este sistema:

Primero, se empezará creando una clase que se encargará de gestionar la base de datos que almacenará la información. La estructura de dicha clase es la siguiente:



```

1 export default class GestorBD {
2     public datos: any;
3
4     constructor(){
5         const datosGuardados = localStorage.getItem(Constantes.BASEDATOS.NOMBRE);
6
7         if (datosGuardados != null){
8             this.datos = JSON.parse(datosGuardados);
9         } else {
10             this.createBD();
11         }
12     }
13 }

```

- Cuando se inicia, se comprueba si existen datos previos en el almacenamiento local (El valor de Constantes.BASEDATOS.NOMBRE es ‘basedatos’).

- En caso de que los haya, se asignará a la variable ‘datos’ un JSON con dichos datos. Esta variable ‘datos’ será la que se utilice para llamar a los diversos elementos de la base de datos.

- En caso de no existir datos previos, se llamará al método ‘crearBD’. Este método es el encargado de crear la estructura de la base de datos en caso de no existir. Una vez creada, esta es asignada a la variable ‘datos’.

Por último, se añade la base de datos al almacenamiento local pasando como una cadena la variable ‘datos’.

```
1  crearBD() {
2      let bdinicial = {
3          datosGuardados: false,
4          musica: {
5              volumenAmbiente: 0,
6              volumenEfectos: 0,
7              musicaFondo: false
8          },
9          jugador: {
10             vida: 0,
11             vidaMax: 0,
12             ataque: 0,
13             curacion: 0,
14             nivel: 0,
15             experiencia: 0,
16             experienciaMax: 0
17         },
18         zonaMapa: '',
19         objetosRecogidos: [] as string[],
20         npcDerrotados: [] as string[],
21         eventosRealizados: [] as string[],
22         posXJugador: 0,
23         posYJugador: 0
24     }
25
26     this.datos = bdinicial;
27     localStorage.setItem(Constantes.BASEDATOS.NOMBRE, JSON.stringify(this.datos));
28 }
```

Para terminar con esta clase, se ha implementado un método ‘actualizarBD’, el cuál será necesario más adelante. Su estructura es la siguiente:

```
● ● ●  
1 actualizarBD(){  
2     localStorage.setItem(Constantes.BASEDATOS.NOMBRE, JSON.stringify(this.datos));  
3 }
```

Una vez la base de datos está preparada, es hora de adaptar aquellos elementos que queremos conservar. Se empezará con el menú principal, en el cuál habrá que añadir el siguiente código:

```
● ● ●  
1 public static bd: GestorBD;
```

Esta será la variable que permitirá acceder a la base de datos. En el constructor, se añadirá lo siguiente:

```
● ● ●  
1 Menu.bd = new GestorBD();  
2     if (Menu.bd.datos.datosGuardados == false){  
3         Menu.music = this.sound.add(Constantes.SONIDOS.MENU,{volume: MenuOptions.ambientSound/100});  
4     } else {  
5         Menu.music = this.sound.add(Constantes.SONIDOS.MENU,{volume: Menu.bd.datos.musica.volumenAmbiente/100});  
6     }
```

Con la implementación de este código, estamos indicando al juego que, en caso de que no hayan datos guardados en la base de datos, use el volumen establecido por defecto. Por otro lado, si hay datos guardados, se usará el volumen que se obtenga de la base de datos.

Para terminar con el menú, se debe hacer una pequeña modificación en el código que controla la subida y bajada de volumen, ya sea del volumen ambiente o de los efectos de sonido. El código será agregado en el bloque correspondiente a cada botón que controla, tanto la música como los efectos de sonido.

En el caso de los botones que controlan la música o volumen ambiente, el código será el siguiente:

```
● ● ●  
1 Menu.bd.datos.musica.volumenAmbiente = MenuOptions.ambientSound;  
2 Menu.bd.actualizarBD();
```

Por otro lado, en los botones que controlan los efectos de sonido, el código será el siguiente:



```
1 Menu.bd.datos.musica.volumenEfectos = MenuOptions.effectsSound;
2 Menu.bd.actualizarBD();
```

Una vez terminada la modificación del menú, es hora de pasar a la escena principal, donde tendrá lugar el guardado de partida.

El primer cambio que se realizará será añadir una comprobación para saber si hay datos guardados en la base de datos o no. Para ello, volveremos a declarar una variable de la clase ‘GestorBD’. Si se encuentran datos, significa que el jugador ya ha guardado partida, lo que implica que ya no se debe mostrar la introducción. En este caso, asignaremos los valores de la base de datos a las propiedades correspondientes, tal y como se muestra en la siguiente imagen:



```
1 if (this.bd.datos.datosGuardados == true){
2     CargaDatos.jugadorVida = this.bd.datos.jugador.vida;
3     CargaDatos.jugadorMaxVida = this.bd.datos.jugador.vidaMax;
4     CargaDatos.jugadorAtaque = this.bd.datos.jugador.ataque;
5     CargaDatos.jugadorExperiencia = this.bd.datos.jugador.experiencia;
6     CargaDatos.jugadorExperienciaMax = this.bd.datos.jugador.experienciaMax;
7     CargaDatos.jugadorNivel = this.bd.datos.jugador.nivel;
8     CargaDatos.jugadorCuracion = this.bd.datos.jugador.curacion;
9
10    CargaDatos.jugadorZona = this.bd.datos.zonaMapa;
11    CargaDatos.npcDerrotados = this.bd.datos.npcDerrotados;
12    CargaDatos.eventosRealizados = this.bd.datos.eventosRealizados;
13    CargaDatos.objetosRecogidos = this.bd.datos.objetosRecogidos;
14
15    this.time.addEvent({delay: 3000, callback: () => {
16        this.cameras.main.fadeIn(1000, 0, 0, 0, (camera: Phaser.Cameras.Scene2D.Camera, progreso: number) => {
17            if (progreso == 1){
18                //Mostrar pantalla de carga
19            }
20        });
21    }, callbackScope: this});
22
23    this.time.addEvent({delay: 7000, callback: this.crearNivel, callbackScope: this});
24}
```

En caso de no haber datos guardados, se entiende que el jugador está empezando una nueva partida, por lo que se tendrá que asignar los valores por defecto y llamar al método que muestra la introducción al juego. El código quedaría de la siguiente manera:

```
1  }else {
2      CargaDatos.jugadorVida = 20;
3      CargaDatos.jugadorMaxVida = 20;
4      CargaDatos.jugadorAtaque = 1;
5      CargaDatos.jugadorExperiencia = 0;
6      CargaDatos.jugadorExperienciaMax = 25;
7      CargaDatos.jugadorNivel = 1;
8      CargaDatos.jugadorCuracion = 0;
9
10     CargaDatos.jugadorZona = '';
11     CargaDatos.npcDerrotados = [];
12     CargaDatos.eventosRealizados = [];
13     CargaDatos.objetosRecogidos = [];
14
15     this.introduccion();
16 }
```

Antes de pasar al propio método de guardado, se deberán de hacer modificaciones de código para terminar de adaptar la escena principal con la base de datos. Por una parte, tendremos al jugador, que en caso de haber datos guardados, se declarará de la siguiente manera:

```
1  if (this.bd.datos.datosGuardados == true){
2      this.jugador = new Jugador({
3          escena: this,
4          x: this.bd.datos posXJugador,
5          y: this.bd.datos posYJugador,
6          textura: Constantes.JUGADOR.ID
7      })
8  }
```

En caso de no haber datos guardados, el jugador será declarado de la forma originalmente descrita, es decir, obteniendo la posición de este en el mapa.

Por otro lado, se deben de controlar aquellos elementos con los que el jugador ha interactuado, es decir, no incluir en mapa aquellos objetos y eventos que han sido activados por el jugador. Para ello, se seguirá la misma estructura en cada uno de los objetos, cambiando las arrays y las propiedades por las correspondientes:

```

● ● ●

1  if (this.bd.datos.datosGuardados == true){
2      if (CargaDatos.npcDerrotados.length != 0){
3          for (let i = 0; i < this.conjuntoNPCs.length; i++) {
4              for (let j = 0; j < CargaDatos.npcDerrotados.length; j++) {
5                  if (CargaDatos.npcDerrotados[j] == this.conjuntoNPCs[i].identi){
6                      this.conjuntoNPCs[i].destroy();
7                  }
8              }
9          }
10     }
11 }

```

Por último, se hablará del sistema para guardar la partida. Para ello, se usarán los objetos de tipo ‘guardado’, acompañados del correspondiente código ubicado en el método ‘comprobarProximidadObjetos’:

```

● ● ●

1  for (let i = 0; i < Prologo.conjuntoGuardados.length; i++) {
2      if ((Math.abs(this.jugador.body.x - Prologo.conjuntoGuardados[i].body.x)) <= 20 && (Math.abs(this.jugador.body.y - Prologo.conjuntoGuardados[i].body.y)) <= 20) {
3          Prologo.efectos = this.sound.add('save_effect', {volume: MenuOptions.effectsSound/100});
4          Prologo.efectos.play({
5              loop: false
6          });
7          CargaDatos.jugadorVida = CargaDatos.jugadorMaxVida;
8
9          this.bd.datos.datosGuardados = true;
10         this.bd.datos.musicaFondo = true;
11         this.bd.datos.jugador.vida = CargaDatos.jugadorVida;
12         this.bd.datos.jugador.vidaMax = CargaDatos.jugadorMaxVida;
13         this.bd.datos.jugador.ataque = CargaDatos.jugadorAtaque;
14         this.bd.datos.jugador.curacion = CargaDatos.jugadorCuracion;
15         this.bd.datos.jugador.nivel = CargaDatos.jugadorNivel;
16         this.bd.datos.jugador.experiencia = CargaDatos.jugadorExperiencia;
17         this.bd.datos.jugador.experienciaMax = CargaDatos.jugadorExperienciaMax;
18         this.bd.datos.zonaMapa = CargaDatos.jugadorZona;
19         this.bd.datos.npcDerrotados = CargaDatos.npcDerrotados;
20         this.bd.datos.eventosRealizados = CargaDatos.eventosRealizados;
21         this.bd.datos.objetosRecogidos = CargaDatos.objetosRecogidos;
22         this.bd.datos.posXJugador = this.jugador.x;
23         this.bd.datos.posYJugador = this.jugador.y;
24
25         this.bd.actualizarBD();
26         var guardado = new Eventos({
27             escena: this,
28             x: 0,
29             y: 0,
30             textura: ''
31         }, 'guardar');
32         this.activarEvento(guardado, 500);
33     }
}

```

¡Ya tenemos el sistema de guardado terminado! Con esto, el juego ya está completo, por lo que en el siguiente apartado se verá como exportarlo a diferentes plataformas.

➤ Exportación multiplataforma del juego.

Una vez que hemos terminado con el desarrollo, toca exportar el proyecto a las diversas plataformas. En este caso, las plataformas a exportar serán: web, Windows y Linux.

Comenzando con la versión web, se deberá abrir la consola y ubicarnos en la carpeta del proyecto.

Acto seguido, se deberá escribir el siguiente comando:

```
C:\TFG\DreamJourney>ionic serve
```

Este proceso suele durar, ya que analiza toda la estructura del proyecto y convierte los archivos typescript en javascript. Una vez finalizado el proceso, se generará una carpeta ('dist' o 'www') cuyo contenido será el juego preparado. Para hacerlo funcionar, se tendrá que configurar un servidor apache y añadir dicha carpeta.

A continuación, se procede a explicar como exportar el proyecto a versiones de escritorio, concretamente Windows y Linux. Este mismo método se puede aplicar para la versión de MacOS, pero para este es necesario estar trabajando en dicho sistema operativo.

Los pasos a seguir son los siguientes:

- Una vez generada la versión web, escribiremos los siguientes comandos:

```
C:\TFG\DreamJourney>npm i electron -g
```

Esto instalará el framework Electron de forma global. En caso de ser necesario, se deberán resolver posibles errores de dependencias (exceptuando el caso de webpack y @angular-devkit).

Una vez hecho esto, se seguirá con los siguientes comandos:

```
C:\TFG\DreamJourney>npm install @capacitor-community/electron@3.0.0
```

```
C:\TFG\DreamJourney>npm install @capacitor/cli@5.0.4
```

Acto seguido, se deberán arreglar dependencias con **npm audit fix -force**. Si se han hecho bien los pasos, sólo deberían quedar dos fallos de dependencias (webpack y @angular-devkit). Dichos fallos serán ignorados.

A continuación, se añadirá la capa de la siguiente manera:

```
C:\TFG\DreamJourney>npx cap add electron
```

Este proceso puede tardar. Una vez finalizado, se lanzará el siguiente comando:

```
C:\TFG\DreamJourney>npx cap open electron
```

Aunque este comando lance un error, es necesario para que se copien archivos en la carpeta 'electron'.

Una vez hecho esto, se puede hacer una comprobación de como quedaría el juego. Para ello, basta con entrar en la carpeta 'electron' y ejecutar '**electron .**'

Para generar las versiones de escritorio, habrá que situarnos dentro de la carpeta 'electron' y ejecutar el siguiente comando:

- Primero, se instalará el paquete necesario para la exportación.

```
C:\TFG\DreamJourney\electron>npm install electron-packager -g
```

- Generar versión de Windows:

```
C:\TFG\DreamJourney\electron>electron-packager . DreamJourney --overwrite --asar --platform=win32 --arch=x64  
--icon=assets/favicon.ico --prune --out=ejecutables
```

- Generar versión de Linux:

```
C:\TFG\DreamJourney\electron>electron-packager . DreamJourney --overwrite --asar --platform=linux --arch=x64  
--icon=assets/favicon.ico --prune --out=ejecutables
```

En caso de estar en un sistema MacOS, habrá que sustituir el contenido de 'platform' por → --platform=darwin.

A continuación, se justifica el uso de las diversas opciones que se encuentran en los comandos:

- overwrite sobrescribirá los archivos de salida en caso de existir.
- prune eliminará aquellos archivos no necesarios.
- asar creará un único fichero.
- out indica la carpeta de destino.
- platform indica la plataforma a la que va a ser exportada la aplicación.

➤ ¿Versión para móviles?

Aunque es cierto que no se ha podido generar la aplicación correspondiente a la versión móvil (se hablará de la situación en profundidad en el apartado de '[conclusiones finales](#)'), se van a comentar los aspectos que diferencian a esta del resto de versiones.

Se comenzará implementando el plugin que permite añadir un joystick en la pantalla. Dentro del archivo 'configuracion.ts', se añadirá el siguiente código:



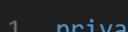
```
 1  plugins: {  
 2      global: [{  
 3          key: 'rexVirtualJoystick',  
 4          plugin: VirtualJoystickPlugin,  
 5          start: true,  
 6          mapping: 'joystick'  
 7      }],  
 8  }
```

Además, se acompañará de su correspondiente import:



```
1 import VirtualJoystickPlugin from 'phaser3-rex-plugins/plugins/virtualjoystick-plugin.js';
```

Una vez implementado el plugin, habrá que dirigirse a la escena principal. En ella, se crearán las siguientes propiedades:



```
1 private joystick: any;
2     private mijoystick: any;
3     public joystickCursors: any;
4     boton_interactuar!: Phaser.GameObjects.Rectangle;
5     texto_interactuar!: Phaser.GameObjects.Text;
```

Estas propiedades serán las que almacenen, por un lado, el joystick junto a sus direcciones de movimiento y, por el otro lado, el botón que se añadirá para poder interactuar con el entorno. Sabiendo esto, se añadirá el siguiente código en el método ‘crearnivel’:



```
1 this.mijoystick = this.joystick.add(this.scene, {
2     x: this.ancho * .43,
3     y: this.alto * .563,
4     radius: 20,
5     base: this.add.circle(0, 0, 15, 0x888888).setAlpha(0.6),
6     thumb: this.add.circle(0, 0, 10, 0xcccccc).setAlpha(0.6),
7     dir: '4dir',
8 });
9 this.joystickCursors = this.mijoystick.createCursorKeys();
```

En mi caso, al estar aplicando una gran cantidad de zoom, hace falta ajustar al más mínimo pixel para que el joystick aparezca en la esquina inferior izquierda. Este joystick se conforma de dos partes: La zona de movimiento o círculo que mueve el jugador (thumb), y la zona que delimita el movimiento del thumb (base).

A continuación, se asignará a la variable ‘joystickCursors’ las direcciones correspondientes (se define con la propiedad dir, en este caso, 4 direcciones: arriba, abajo, izquierda y derecha).

Para terminar con el joystick, se deberá añadir al código de la clase ‘Jugador’ el siguiente código, modificándolo dependiendo de la dirección :



```
1 if (this.cursores.left.isDown || this.escena.joystickCursors.left.isDown)
```

Una vez configurado el joystick, se procederá a añadir el botón con el cuál el jugador podrá

interactuar con el entorno. El código es el siguiente:

```
● ● ●  
1 this.boton_interactuar = this.add.rectangle(this.ancho * .57, this.alto * .575, 40, 15, 0x808080).setAlpha(1).setInteractive().setDepth(9);  
2 this.input.addPointer(1);  
3 this.botonpulsado(this.boton_interactuar);  
4 this.boton_interactuar.setScrollFactor(0);  
5 this.texto_interactuar = this.add.text(this.ancho * .549, this.alto * .57, 'Interactuar', this.miestyle).setDepth(10).setScale(0.35);  
6 this.texto_interactuar.setScrollFactor(0);
```

Esto creará un botón en la esquina inferior derecha, el cuál, al ser presionado, llamará al método 'botonpulsado'. Además, la línea 2 permite que se añada un segundo control a la pantalla, siendo el primero por defecto el ratón.

Los métodos 'setScrollFactor' permiten que estos elementos permanezcan siempre en pantalla y en la misma posición.

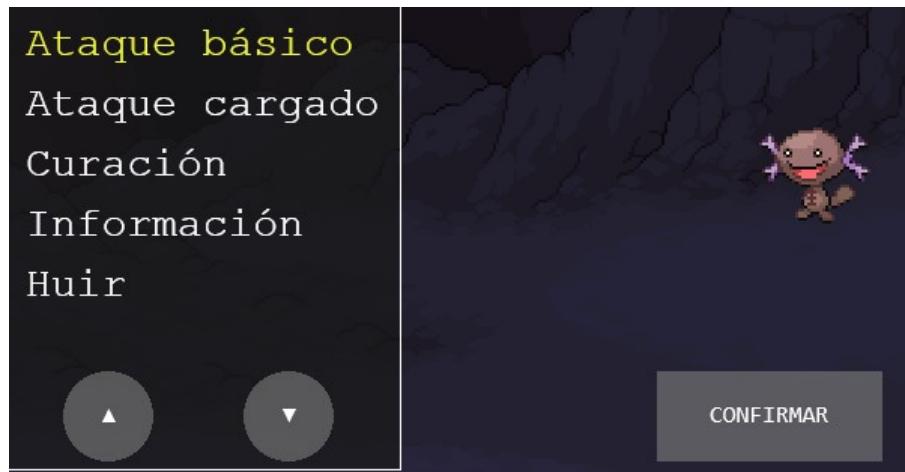
El código correspondiente al método 'botonpulsado' es el siguiente:

```
● ● ●  
1 botonpulsado(boton: any){  
2     boton.on('pointerdown', () => {  
3         this.comprobarProximidadObjetos();  
4     });  
5 }
```

Una vez aplicados estos cambios, el resultado es el siguiente:



Por último, deberemos de aplicar la misma lógica en la escena de batalla. El resultado es el siguiente:



- Cuando el botón 'Arriba' o 'Abajo' es pulsado, se llamará al método 'seleccionarArriba' o 'seleccionarAbajo' de forma correspondiente.
- Cuando el botón 'CONFIRMAR' es pulsado, se llamará al método 'confirmarSeleccion'.

Con estas modificaciones, la versión de móvil estaría finalizada. Sin embargo, tal y como se ha comentado al principio del apartado, más adelante se darán los motivos por los cuáles no ha sido posible generar dicha aplicación.

■ FASE DE PRUEBAS

Una vez terminado el juego, sólo resta hacer pruebas de este y corregir posibles errores. Para ello, se ha pedido a tres personas que probasen el juego. Se recurre a esto de forma que si yo pruebo el juego y no encuentro errores o fallos, otras personas si pueden tener dichos fallos o errores a la hora de jugar.

Una vez recabada la información proporcionada por dichas personas , se procede a la resolución de los posibles fallos.

En este caso, se proporcionaron tanto fallos como posibles mejoras para el juego. Tengamos en cuenta primero los fallos:

- **Cuando se activaba un evento de historia, este entraba en un bucle infinito:** Antes de ver por qué se ocasionaba dicho error, decidí probar en un ordenador distinto al mío para verificarlo; sin embargo, este fallo no ocurrió en ningún momento. Para estar seguro de que el código funcionaba correctamente, añadí una variable que controlaba la activación del evento y, al mismo tiempo, la desactivación de este. De esta forma, el evento se ejecutaría y en el mismo instante dejaría de estar disponible.

Por último, comprobé los cambios en dos ordenadores distintos, en los cuáles funcionaba

correctamente. Se entregó la nueva versión a la persona que notificó el fallo para corroborar esto, pero el fallo persistía. Se intentó arreglar de diversas maneras, pero al final se llegó a la conclusión de que el código se ejecutaba correctamente, impidiendo que el evento se ejecutase más de una vez (el resto de personas que probaron el juego no sufrieron de dicho error).

La solución que aporto esta persona fue que, al cerrar y volver a abrir el juego, este fallo desaparecía y podía continuar.

- **Cuando el jugador es derrotado, la escena principal sigue activa, impidiendo ver con claridad la escena correspondiente al fin de partida:** Este fallo se originaba debido a que, al finalizar el combate, se finalizaban las escenas correspondientes al combate y se iniciaba la escena de derrota, pero la escena principal se encontraba en reposo, por lo que la solución se encontraba en finalizar dicha escena principal en caso de que el jugador fuese derrotado.

- **La curación que recibía el jugador era inferior a lo que el enemigo quitaba:** Este error sucedía debido a que previamente se había modificado el sistema de curación, haciendo que el jugador regenerase más vida con respecto a su nivel. Sin embargo, este sistema no tenía en cuenta cuando el jugador era nivel 1, haciendo que curase solo 2 puntos de salud. Para su solución, simplemente se agregó una excepción a este sistema para que el jugador, estando a nivel 1, pudiera regenerar una cantidad de salud correcta.

- **Si se choca con un NPC de forma diagonal, su animación de movimiento se queda trabada:** Este error no es posible darle una solución directa, ya que el tema colisiones no es muy preciso en Phaser. La única solución viable fue aumentar el tiempo que tarda el jugador en entrar en combate cuando colisiona con un NPC.

- **Algunos efectos de sonido se reproducían con un volumen excesivamente alto. Además, en ciertas zonas no se reproducía audio:** El fallo en este caso reside a la hora de ejecutar el efecto de sonido, ya que en Phaser, el volumen va de 0 a 1, siendo 1 un 100%. Esto causaba que si el volumen se ajustaba al 25%, se reproducía el sonido al 2500%. La solución ha dicho problema fue dividir entre 100 el volumen con el que se ejecutaba estos sonidos.

Con respecto a la zona sin sonido, se añadió la música correspondiente, así como un ajuste en el volumen de esta.

Una vez solucionados los errores en la medida de lo posible, es el turno de las ideas para mejorar el juego. Cabe recalcar que todas las ideas mencionadas a continuación han sido implementadas:

- **Un botón que permita correr al jugador:** Ahora, presionando la tecla 'X' mientras el jugador se mueve, su velocidad aumentará. Al dejar de presionar dicha tecla, la velocidad del jugador volverá a la normalidad.

- **Añadir un botón que permita saltarse los eventos de historia:** Cuando el jugador se encuentre con un evento de historia, podrá usar la barra espaciadora (botón de interacción general) para poder omitir dicho evento.

- **Añadir efectos de sonido en la batalla para indicar cuando se sube de nivel, cuando el jugador se cura o cuando se huye de la batalla:** Dichos efectos se encuentran ya introducidos dentro del juego.

- **Añadir más de una canción para los combates:** Ahora, dependiendo de la zona en la que el jugador se encuentre, la música de combate cambiará (este cambio no aplica a los jefes, ya que estos cuentan con su propia música de combate).

- **Idea para mejorar un evento de historia y su correspondiente ajuste.**

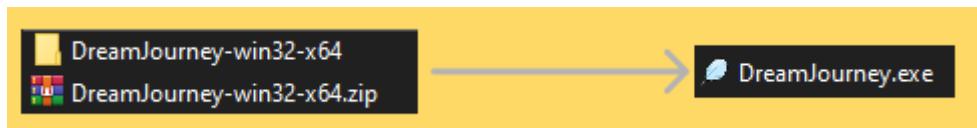
■ DOCUMENTACIÓN DE LA APLICACIÓN

A continuación, se va a explicar cómo poder instalar y abrir el juego, tanto en plataforma web, como en escritorios Windows y Linux.

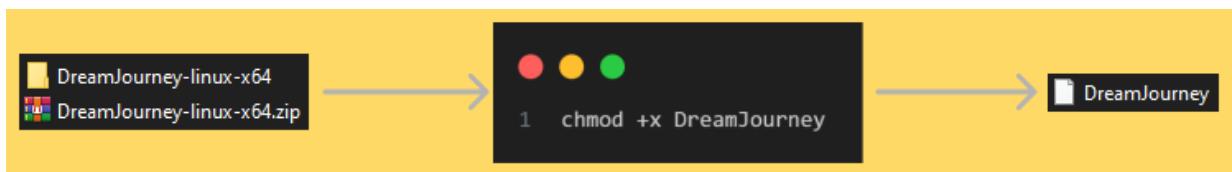
➤ Manual de instalación.

Lo primero que se tendrá que hacer es dirigirse a la carpeta “ejecutables”, la cuál se puede encontrar dentro de la carpeta del proyecto. Dentro de dicha carpeta se encuentran las diversas versiones del juego, por lo que se deberá de seleccionar la versión que se quiera usar. Dependiendo de la versión elegida, el procedimiento de instalación será diferente:

- Si se ha descargado la versión de Windows, se deberá extraer el contenido del Zip. Acto seguido, entrar en la carpeta y hacer doble clic en la aplicación “**DreamJourney.exe**” para abrir el juego.



- Si se ha descargado la versión de Linux, se deberá extraer el contenido del Zip. Una vez hecho esto, se deberá abrir la terminal dentro de la carpeta donde se localizan todos los archivos y ejecutar el comando: “**chmod +x DreamJourney**”. Por último, bastará con hacer doble clic en el archivo “**DreamJourney**” para abrir el juego.



- Si se ha descargado la versión web, esta deberá ser alojada en un servicio web. Para ello, se puede disponer de herramientas como **Docker**, el cuál nos despliega un entorno virtualizado donde almacenar el proyecto; o sistemas de gestión de servicios, tales como

XAMPP o Laragon, los cuales brindan los servicios necesarios para la versión web. Cuando se haya configurado dichos servicios, se situará la versión web en la carpeta correspondiente. Por último, se deberá entrar al juego mediante la URL del servicio.

➤ **Manual de usuario.**

(Recomendado: Para mejor experiencia se recomienda jugar con la ventana maximizada, o ajustar la ventana de modo que se muestre todo el contenido)

A continuación, se detallan algunos consejos e información referentes a los diversos aspectos del juego:

- **Menú principal:** Nada más abrir el juego, se mostrará el menú principal, el cuál cuenta con dos botones. Seleccionando el botón 'Jugar' comenzará el juego inmediatamente. Por el contrario, si se selecciona el botón 'Opciones', se podrá encontrar una ventana que permite configurar el volumen ambiente y el volumen de los efectos de sonido propios del juego.

- **Controles de jugador:** El jugador dispone de una serie de controles que le permite interactuar y desplazarse por el entorno. Estos son:

+ Flechas direccionales: Controlan el movimiento del jugador.

+ Tecla 'X': Al dejar pulsada esta tecla mientras el jugador se mueve, su velocidad aumentará. Al dejar de pulsar esta tecla, la velocidad del jugador volverá a la normalidad.

+ Barra Espaciadora: Estando cerca de un objeto y presionando esta tecla, el jugador interactuará con dicho objeto. Además, esta tecla permitirá omitir los eventos de historia durante el transcurso de estos.

- **Objetivo del juego:** El objetivo principal de este juego consiste en averiguar el motivo por el cuál Goldie, la protagonista, se encuentra en un lugar totalmente desconocido para ella, a la vez que lucha con ciertas amenazas. La beta 1.1 ofrece la posibilidad de jugar el prólogo, donde se introduce a la protagonista y se relata el comienzo de la historia.

- **Combates:** Durante el transcurso del juego, el jugador se encontrará con diversos enemigos, ya sea de forma aleatoria o de forma obligatoria. A continuación, se dará información acerca de cómo manejar el escenario de combate. Una vez el combate haya comenzado podremos observar una interfaz con los siguientes elementos:



+ Cuadro Acciones: En este apartado se encuentran las diversas opciones que el jugador puede elegir a la hora de combatir, entre las que se encuentran:

- * *Ataque básico*: El jugador provocará cierto daño al enemigo.
- * *Ataque cargado*: El jugador provocará mayor daño al enemigo, pero deberá descansar en el siguiente turno.
- * *Curación*: Siempre y cuando el jugador tenga objetos curativos, este podrá regenerar vida. La cantidad curada variará dependiendo del nivel del jugador.
- * *Información*: Mostrará al jugador una leyenda con los controles y una breve descripción de las opciones de combate.
- * *Huir*: Permite al jugador huir del combate siempre y cuando las circunstancias lo permitan.

+ Cuadro de jugador: Apartado donde se muestra la información referente al jugador (nivel, vida, objetos curativos y experiencia).

+ Cuadro de enemigo: Apartado donde se muestra la información referente al enemigo (nombre y vida).

+ Cuadro de información: Cuadro que se actualiza conforme se realizan acciones, notificando al jugador de lo que está pasando en cada momento.

- Guardar partida: El jugador podrá encontrar varios objetos con forma de bloc de notas. Si el jugador interactúa con dicho objeto, podrá guardar su progreso, así como restaurar su vida por completo.

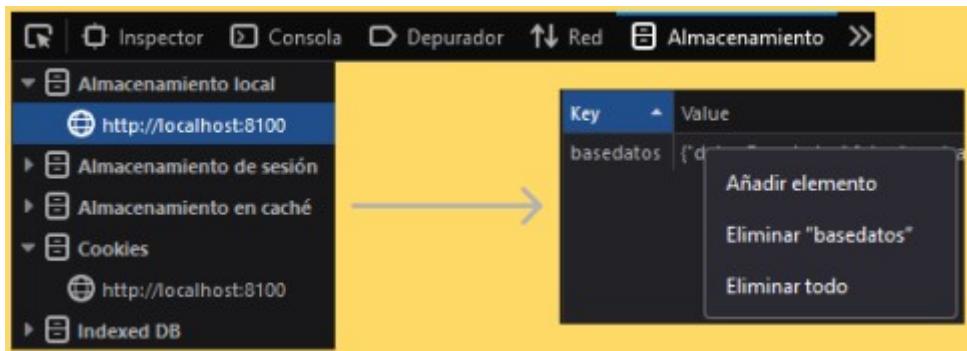
En caso de que el usuario quiera borrar la partida, deberá seguir una serie de pasos dependiendo de la versión utilizada:

+ Si el usuario está usando la versión de Windows, deberá acceder a la siguiente ruta:
“ **C:\Users\NombreDeUsuario\AppData\Roaming\dreamjourney\Local Storage** ” y borrar el contenido de dicha carpeta.

+ Si el usuario está usando la versión de Linux, deberá acceder a la siguiente ruta:
“ **/home/NombreDeUsuario/.config/dreamjourney/Local Storage** ” y borrar la carpeta **leveldb**.

+ Si el usuario está usando la versión web deberá, dentro de la propia página, hacer clic derecho y seleccionar ‘Inspeccionar elemento’. Acto seguido, debe dirigirse a las herramientas de desarrollo y entrar en la sección ‘Storage o Almacenamiento’ (esta ubicación puede variar dependiendo del navegador usado).

Una vez localizado, se deberá buscar la base de datos y eliminarla. A continuación, se muestra un ejemplo de cómo se debería ver esta operación (en el ejemplo se ha usado el navegador Mozilla Firefox):



Por último, cabe recalcar que existe la posibilidad de que, a la hora de mostrar los diferentes eventos de historia, estos se vuelvan infinitos y no terminen. A pesar de varias correcciones de código y pruebas de este, no se ha encontrado el motivo por el que esto sucede, ya que no es un error que suceda en el 100% de los casos.

En caso de que suceda este error, basta con cerrar y volver a abrir el juego.

■ CONCLUSIONES FINALES

Para concluir con el proyecto, se hablará de los objetivos propuestos principalmente, se hayan cumplido o no, así como una serie de propuestas para mejorar dicho proyecto en un futuro.

Comenzando con los objetivos propuestos, bien es cierto que se han encontrado ciertas dificultades a la hora de implementar algunas de las mecánicas.

- Empezaremos con la mecánica del cambio de zona ya que, aunque bien es cierto que dicha mecánica proviene de un proyecto anterior, la forma en la que se usaba era incompatible. Debido a eso, se busco la forma de modificar el código con el fin de ser una mecánica útil para el

proyecto.

- Continuamos con la mecánica de los NPC. Cuando se hizo esta mecánica, su uso era el de un juego de obstáculos donde, al colisionar con un NPC, el jugador perdía vida. Se intentó hacer que, en este caso, al colisionar con el jugador, se activase un combate con el enemigo correspondiente al NPC. Se intentó añadir colisiones entre jugador y NPC, buscando de esta manera que, al chocar ambas colisiones, se activase el combate. Sin embargo, esta forma no funcionaba, dejando como última opción la actual y, por consiguiente, el pequeño fallo de intentar huir del NPC pero entrar en combate continuamente.

- Terminando con las mecánicas, tenemos los eventos de historia. En este caso se debe diferenciar dos eventos de historia: la introducción junto a la pantalla de carga; y el resto de eventos.

Esto se debe a que la introducción se sitúa en la propia escena principal antes de que el mapa sea agregado, por lo que las dimensiones son las normales. Sin embargo, si se intentaba hacer un evento de este tipo después de haber sido creado el mapa, esto no era posible debido a las nuevas dimensiones de la cámara y al zoom aplicado, haciendo que los textos que se introducían se vieran borrosos.

Es por eso que se decidió implementar estos eventos en una escena aparte que se superponía a la principal cuando se activaba un evento.

En lo que respecta al resto de objetivos/mecánicas, fue más tiempo de adaptación que dificultad en sí.

A continuación, se hablará de aquellos objetivos que no se han podido cumplir. El primero ellos está relacionado con la falta de una continuación al prólogo, ya que en un principio el juego estaría formado por un prólogo + capítulo 1, pero debido al tiempo empleado para el prólogo, era imposible añadir una continuación a este.

Por otro lado, está el caso de la versión móvil. Tal y como se ha comentando en el apartado de ‘Diseño e implementación del proyecto’, la versión móvil del juego se ha desarrollado de la misma manera que la versión web y la versión de escritorio, adaptando los controles a los correspondientes para móvil. A pesar de ello, surgieron varios errores a la hora de exportar el proyecto y convertirlo en una apk:

- Aunque se indicaba que las dimensiones del juego debían ajustarse a la pantalla del dispositivo, al final de cada prueba este ajuste nunca se producía.

- Por otro lado están los eventos de historia, ya que a pesar de que ambos tuvieran el mismo tamaño y la misma fuente, cuando se generaba la aplicación, cada texto se mostraba con un tamaño, fuente y posición distinta a la establecida.

- Por último, estaba el problema del mapa, y es que, a pesar de estar todo correcto, el mapa no cargaba dentro de la versión móvil. Se intentó generar la aplicación mediante Electron (mismo método que para versión de escritorio) y con el uso de Cordova, pero en esta caso,

tampoco se llegaba a mostrar el mapa.

Estos fueron los motivos por los cuales no se pudo presentar una versión de móvil a pesar de estar preparada.

Una vez finalizado el apartado correspondiente a los objetivos, es hora de comentar ciertas propuestas y mejoras que se pueden aplicar al proyecto a futuro.

La primera propuesta es clara: Llevar el proyecto a un motor de videojuegos más completo. Esto se debe a que, a pesar de que Phaser también es un motor de videojuegos, este solo está limitado a versiones web, ya que de no ser por la implementación de IONIC, no se podría haber generado el resto de versiones. Es por ello que la idea de traspasar el proyecto a otro motor, Unity por ejemplo, sería una idea más que acertada.

Gracias a esta propuesta, pienso que se pueden mejorar gran parte de los aspectos que conforman el juego, ya sean cambios visuales, cambios de código de mecánicas con el fin de mejorar estas, añadido de mayor contenido, e incluso la posibilidad de una versión móvil estable y funcional.

En mi opinión, durante el periodo de realización del proyecto, he podido observar lo complejo que puede llegar a ser el desarrollar un videojuego, controlar que todo funcione al unísono y no produzca errores, el diseño de nivel y de historia, etc. Desde pequeño he sido un aficionado por los videojuegos, y pensar que he sido capaz de desarrollar uno yo mismo, es algo que todavía no termino de creerme. Aun así, esta experiencia me ha proporcionado ciertos fundamentos y entusiasmo para seguir aprendiendo todo lo relacionado a este sector, y aquellos sectores que lo rodean, ya sea inteligencia artificial o diseño 3D, sin llegar a olvidarme de las bases que ya poseo.

■ BIBLIOGRAFÍA

A continuación, se nombran las fuentes de donde se ha sacado información:

- Temario de la asignatura ‘Diseño de Interfaces’ : Tema 8-1 Introducción a IONIC (Necesario para la configuración del entorno de trabajo).
- Temario de la asignatura ‘Diseño de Interfaces’: Tema 8-2 Phaser (Se ha sacado poca información, debido a que el tipo de juego desarrollado y el tipo de juego impartido en el tema son distintos).
- Reutilización de código de un proyecto anterior realizado en la asignatura ‘Diseño de Interfaces’. Se han reusado las siguientes mecánicas:

- + Menú de opciones.
- + Mecánica de cambiar de zona.
- + Mecánica de obtención de objetos.
- + Mecánica de movimiento de NPCS.

- [Leshy SpriteSheet Tool](#) (Usado para crear las animaciones de objetos, enemigos y jugador).
- [Tiled](#) (Usado para la creación de mapas)
- [Phaser3-JoystickPlugin](#) (Plugin utilizado para la implementación de un joystick funcional para la versión móvil)
- [The Sprites Resource](#) (Utilizado para obtener los diversos elementos del juego, tales como enemigos, objetos, pantalla de menú, jugador, etc).
- [The Sounds Resource](#) (Utilizado para los efectos de sonidos que se reproducen durante la escena de combate).
- Youtube (Utilizado para obtener las diferentes bandas sonoras que hay durante el juego)