UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

GRADO EN INGENIERÍA INFORMÁTICA

Desarrollo de un videojuego serio como herramienta de sensibilización ante el cambio climático

Development of a serious videogame as an awareness tool to climate change

Realizado por
Samuel Espadas Cabezas

Tutorizado por
Antonio José Fernández Leiva

Departamento
Lenguajes y ciencias de la computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2024

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Desarrollo de un videojuego serio como herramienta de sensibilización ante el cambio climático

Development of a serious videogame as an awareness tool to climate change

Realizado por
Samuel Espadas Cabezas

Tutorizado por
Antonio José Fernández Leiva

Departamento
Lenguajes y ciencias de la computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2024

Fecha defensa: septiembre de 2024

Resumen

Este proyecto se enfoca en el desarrollo de un videojuego serio desde la base, usando el motor de creación de videojuegos Unity y el lenguaje de programación C#. El producto final de este trabajo es un videojuego de plataformas 2D con elementos de puzle y acción llamado *EcoDreams* y cuyo objetivo consiste en concienciar al jugador sobre las causas y consecuencias del cambio climático.

Para lograr esta meta, nuestro juego implementa en ciertos puntos pruebas dónde al jugador se le plantean una serie de preguntas y en función de cómo estas se respondan puede suponer una ventaja o un atraso para él.

Para completar el videojuego el jugador debe superar todos los distintos niveles, siendo forzado a responder las preguntas que se le plantean para poder continuar, logrando así el objetivo serio de nuestro proyecto, el de concienciar a los usuarios del problema del cambio climático.

Palabras clave: programación de videojuegos, videojuego serio, cambio climático.

Abstract

This project is focused on the development of a serious video game from zero, using the professional video game development engine Unity and the programming language C#. The final product of this job is a video game that belongs to the 2D platform style with some puzzle elements called *EcoDreams* whose mission is aware the player about causes and consequences of climate change.

To achieve this goal, our game implements at certain points trials where the player needs to answer a set of questions and, depending on the result, could be an advantage or a delay for him.

To complete the videogame, the player have to successfully pass each level, something that force him to overcome all the questions raised, thus achieving our videogame goal, raise the user awareness about the problem of climate change.

Keywords: videogames programming, serious videogame, climate change.

Agradecimientos

Quiero agradecer a todas las personas que me han acompañado en este viaje y jamás permitieron que me rindiera. A mi familia, mi pareja y mis amigos que me han apoyado contra viento y marea, este trabajo no es solo mío, también vuestro, pues sin vuestra presencia y apoyo no hubiera sido posible. Gracias.

Índice de figuras

Índice

1

Introducción

En este capítulo se introduce el problema que vamos a tratar, dando unas pequeñas pinceladas a la solución para dejar entrever algunos temas que se tratarán con mayor profundidad más adelante en este documento.

1.1 Motivación

Uno de los grandes problemas del siglo XXI, si no el mayor, es el cambio climático y como consecuencia de este el calentamiento global. Y ligado a este problema encontramos la desinformación y el desconocimiento de gran parte de la población de cómo se ha originado este fenómeno y qué es exactamente el cambio climático.

Una forma de definir el concepto de “cambio climático” es, según la Convención Marco de las Naciones Unidas sobre el Cambio Climático (ONU, 1992), como "un cambio de clima atribuido directa o indirectamente a la actividad humana que altera la composición de la atmósfera mundial y que se suma a la variabilidad natural del clima observado durante períodos de tiempo comparables".

Debido al desconocimiento de dicho tema, se presenta la necesidad de concienciar a la sociedad sobre qué es y cómo actuar frente a esta amenaza.

Una de las industrias que más público atrae y que cada vez más está cogiendo mucha fuerza es la industria de los videojuegos. Millones de personas juegan videojuegos como entretenimiento y, cada vez más, se están adaptando a diversos objetivos más allá del mero ocio surgiendo así el término de videojuego serio.

En todo este contexto nace la idea de desarrollar un videojuego que, además de entretener al público que lo juegue, consiga educarlo en materia medioambiental, concienciando de un problema muy actual y del cual hay mucha desinformación.

1.2 Objetivos

El objetivo de este TFG es el de desarrollar una versión ejecutable para PC de un videojuego serio que sirva a todo aquel que lo juegue como herramienta de concienciación frente al cambio climático.

Realizaremos toda la parte de diseño y desarrollo del videojuego empleando la metodología SCRUM. El tutor ejercerá de cliente mientras que el alumno ejercerá los diferentes roles del equipo en función de las necesidades del proyecto (SCRUM Master y los diferentes roles dentro del equipo de desarrollo). A medida que avance el proyecto iremos pasando por las distintas fases de desarrollo de software: especificaciones iniciales y requisitos, diseño, implementación y pruebas.

Este juego está pensado para que cualquiera que muestre interés en él pueda jugarlo de forma gratuita, de esta manera estaremos dando voz a un proyecto que busca un bien común y que debido a su formato de videojuego puede atraer a un público considerablemente alto.

En esta versión tendremos una serie de niveles que el jugador deberá superar para devolverle al mundo su esplendor. El orden de los niveles no es relevante, se podrá jugar en función del tema que más desee tratar el jugador.

Para completar los distintos niveles el jugador deberá controlar al personaje mediante teclado y derrotar a los diferentes enemigos que vayan apareciendo mediante saltos, disparos o uso de algún ítem de poder. Además de ítems de poder el jugador podrá valerse de objetos que aparecerán en el nivel para seguir avanzando. Cuando se supere una fase, volveremos a la pantalla de selección de nivel, pero antes de continuar habrá que responder correctamente una o varias preguntas sobre el tema que se acaba de tratar.

En el apartado técnico, emplearemos un motor profesional de desarrollo de videojuegos, Unity, y para la programación de los distintos componentes del juego usaremos el lenguaje C# y el entorno de desarrollo Visual Studio.

1.3 Estructura de la memoria

En este primer capítulo introductorio se ha desarrollado la necesidad y motivación de las que nace este proyecto, así como los objetivos que inicialmente se marcaron para él. En los siguientes capítulos hablaremos de los demás aspectos del proyecto quedando estructurada la memoria de la manera que sigue:

Capítulo 1: Introducción. Se pone en contexto al lector sobre la necesidad de la que nace el proyecto y los objetivos que se persiguen y se introduce el resto de la memoria.

Capítulo 2: Antecedentes. Se presentan los conceptos básicos y las referencias tomadas en cuenta para el desarrollo del trabajo.

Capítulo 3: Diseño de la solución. Se detallará el proceso de diseño del videojuego, para así entender su desarrollo y cómo se ha obtenido el producto final.

Capítulo 4: Implementación de la solución. En este capítulo hablaremos de cómo se ha implementado el código usando Unity y C# y presentaremos los distintos elementos y clases que conforman el juego mediante diagramas.

Capítulo 5: Resultados finales y experiencia de usuario. Tras haber detallado todo el proceso de desarrollo del juego, se presenta el producto de este trabajo y se analizan los resultados obtenidos durante la fase de pruebas por un grupo de usuarios previamente seleccionados.

Capítulo 6: Conclusiones y trabajo futuro. Para cerrar esta memoria se hará una pequeña reflexión sobre todo el proceso de desarrollo de este trabajo y hablaremos de lo que ha supuesto a nivel personal y educativo, así como posibles mejoras y añadidos para el proyecto en el futuro.

2

Antecedentes

Este capítulo introduce los conceptos clave de este proyecto, aquellos que el lector debe conocer para poder entender los temas que en este documento se tratan.

2.1 Terminología

En esta primera sección se explicarán los términos más presentes durante el desarrollo de este trabajo, ya que aparecerán a lo largo de este documento y es necesario que el lector los entienda para poder comprender algunos aspectos del proceso de diseño e implementación del videojuego.

2.1.1 Videojuego serio

El primero en acuñar este término fue el Doctor Clark C. Abt en 1970, proponiendo de manera revolucionaria el uso de juegos y simulaciones como herramientas para ayudar en la toma de decisiones en la industria, las instituciones gubernamentales, la educación o las relaciones personales (U-tad, 2022).

En este caso en concreto queremos desarrollar un juego que ayude a concienciar al jugador, puesto que la idea es que se informe sobre el problema que presentamos a medida que avanza partida a partida.

2.1.2 Desarrollo de software

El desarrollo de software se puede definir como el proceso de diseñar y crear un producto de software aplicando técnicas y habilidades de programación, diseño, pruebas y mantenimiento.

Dentro del ciclo de desarrollo del software encontramos varias fases por las que debe ir pasando el producto hasta llegar a su fase final:

-**Especificación y requisitos**: El cliente define las características y funcionalidades que quiere que estén presentes en el producto final.

-**Diseño**: Se crea un diseño del producto teniendo en cuenta aspectos como las estructuras de datos que se van a utilizar, la arquitectura del sistema, entre otros aspectos que definirán un modelo inicial de nuestro software.

-**Implementación**: Se escribe el código teniendo en cuenta el diseño realizado previamente y empiezan a obtenerse versiones del producto final.

-**Pruebas**: Fase en la que se desarrollan tests exhaustivos para detectar fallos o anomalías en el funcionamiento de nuestro código.

-**Despliegue**: Una vez que se pasan las pruebas y el código está listo, se lanza el software en un entorno de producción que puede estar abierto al público o restringido a un grupo de usuarios.

-**Mantenimiento**: Es la fase en la que se corrigen errores, se agregan nuevas características, se mejora el rendimiento... Esta fase se mantiene durante toda la vida útil del software.

2.1.3 Metodología SCRUM

SCRUM es una metodología ágil de desarrollo de software. Esta metodología se puede entender como "un marco ligero que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptables para problemas complejos" (Schwaber & Sutherland, 2020).

SCRUM es una metodología basada en *sprints*. En cada *sprint* el *Product Owner* marca los objetivos/requisitos que hay que cumplir en dicha fase, el equipo de desarrollo transforma esos requisitos en funcionalidades o cualquier otro tipo de utilidad, se testean y se comprueba que funcionen correctamente y al final del *sprint* se le presentas los resultados al *Product Owner* y este define los nuevos objetivos para el siguiente sprint.

Esta metodología cuenta con varios roles indispensables para poder ser ejecutada: el propietario del producto (o *Product Owner*), es el cliente que nos encarga el trabajo, el encargado de sacar provecho al producto que estamos desarrollando; el *Scrum Master* es la persona que se encarga de aplicar la filosofía de SCRUM, llevando a la práctica todos sus aspectos y encargándose de que todo el mundo los entienda; y por último el equipo de desarrollo, los encargados de transformar las necesidades del cliente en un producto con valor.

En este proyecto se ha trabajado en base a dicha metodología ejerciendo el tutor el rol de *Product Owner* y el alumno tanto de *Scrum Master* como el resto de los roles del equipo de desarrollo.

2.1.4 Motor de videojuegos: Unity

"Los motores de videojuegos son realmente el núcleo en el que se unen todas las piezas que integran un videojuego, desde las más artísticas o creativas a la programación."(Unir,2023)

Hoy en día existe una amplia variedad de motores, cada uno con sus características propias. Por eso es importante conocer bien los requisitos del proyecto y elegir el motor que más se adecúe a las características que queremos implementar en nuestro videojuego.

Entre los principales motores que se usan en la actualidad encontramos: Unity, Unreal Engine, Godot, GameMaker Studio, …

En nuestro caso nos hemos decantado por Unity, el motor de videojuegos por excelencia. En este trabajo se ha usado para desarrollar un videojuego 2D (aunque también permite la creación de juegos 3D) de plataformas puesto que ofrece una amplia variedad de herramientas que permiten obtener un resultado de alta calidad.

Figura 2.1 *Logo de Unity en 2024*

Además, una característica de Unity que se ha explotado bastante en este desarrollo y que se mencionará con frecuencia en el capítulo 4 es su ciclo estándar que inicia con el método *Awake()* que se ejecuta cuando se crea la instancia del objeto, sigue con el método *Start()* cuya ejecución tiene lugar en el primer *frame* activo del objeto, y se actualiza contantemente con el método *Update()* el cual se ejecuta en cada *frame* del juego, permitiéndonos darle un comportamiento al objeto u otro en función de los cambios que están ocurriendo en escena en tiempo real .

2.1.5 C#

C# (Microsoft, 2000) es un lenguaje de programación desarrollado por Microsoft. Es de código abierto, multiplataforma y orientado objetos y en la actualidad se encuentra entre los 5 lenguajes de programación más usados en GitHub.

Entre las principales ventajas que encontramos en C# encontramos una facilidad de lectura que sumada a su tipado estático y la enorme cantidad de librerías que encontramos hace que desarrollar en este lenguaje sea algo realmente rápido y llevadero.

Al ser un lenguaje que nació con la idea de mejorar C y C++ encontramos similitudes con ambos, así como con Java, dando lugar a una sintaxis muy similar a la de Java y C++ pero aportándose entre sí las ventajas el uno del otro.

Todo el código desarrollado en este trabajo está escrito en C#, uno de los 2 lenguajes nativos que utiliza Unity. Además, al incluir la librería UnityEngine y UnityEditor, las cuales se usan durante el desarrollo de código, se simplifica el enlace directo con algunas de las posibilidades que ofrece el motor como por ejemplo acceso al GameObject o a cualquiera de los componentes del objeto padre del script, cosa que sin las librerías sería realmente complicado de hacer.

2.1.6 JSON

JSON (JavaScript Object Notation) (JSON,s.f) es un formato de texto para el intercambio de datos. Es un formato cada vez más usado por la comunidad del desarrollo de software debido a la facilidad de escritura y lectura que proporciona a los humanos y el sencillo procesamiento por parte de la máquina.

Un JSON se puede escribir como:

- Una colección de pares clave/valor

  (p.ej: {"edad": 12 , "nombre":Pedro })

- Una lista ordenada de valores

  (p.ej: [Pedro , José , Mario]

Nosotros hemos usado este formato para almacenar información sobre el estado del juego (archivo de guardado) así como para guardar todas las preguntas que realizaremos al jugador a lo largo de la partida, siendo una de las ventajas que tiene el uso de este formato que el usuario final podrá acceder a este archivo para añadir/modificar/eliminar preguntas en base a la dificultad y el tema que estas tratan.

2.2 Referentes

Ahora que ya conocemos los términos que vamos a manejar, vamos a introducir las diferentes obras que han servido de inspiración para este trabajo tanto a nivel de diseño como a nivel funcional.

2.2.1 Super Mario Bros

Super Mario Bros (Nintendo, 1995) es el videojuego 2D de plataformas por excelencia. En esta serie de videojuegos Mario (a veces junto a su hermano Luigi) debe recorrer el Reino Champiñón derrotando enemigos mediante saltos y algún que otro power-up para poder infiltrarse en las fortalezas del rey koopa Bowser para rescatar a la princesa Peach, la cual siempre es secuestrada por Bowser al principio de la aventura.

Por eso, y como no podía ser de otra forma en este desarrollo, la presencia de esta serie de juegos ha estado presente constantemente, tanto en el apartado artístico como en la forma de entender un videojuego de plataformas a la hora de programar.

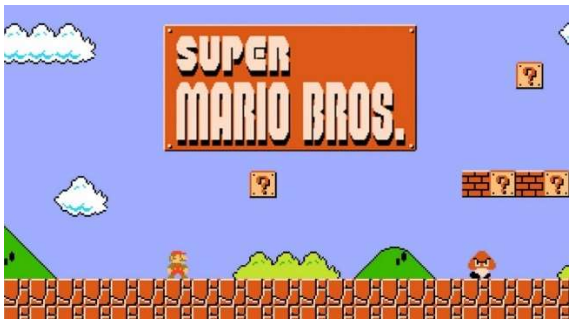


Figura 2.2 *Super Mario Bros* (1985).

2.2.2 Ghosts 'n Goblins

Ghosts 'n Goblins (Capcom, 1985) es otro de los videojuegos de plataformas que han servido de inspiración para realizar este trabajo, estando el ataque y el movimiento del jugador basado en el de Arthur, el protagonista de este juego.

En este videojuego inicialmente para máquinas Arcade y posteriormente lanzado a otras plataformas, Sir Arthur debe recorrer el mapa (formado por 7 niveles diferentes) para rescatar a su amada, la princesa Prin-Prin del malvado Astaroth.

Figura 2.3 *Ghosts 'n Goblins* (1985).

Ghosts 'n Goblins presentaba una dificultad elevadísima para un juego de arcade y en su versión de NES es considerado uno de los juegos más difíciles lanzados para una consola de Nintendo.

2.2.3 The Legend of Zelda

Otro de los juegos más famosos de la empresa Nintendo son los de la *saga The Legend of Zelda* (1986). Aunque la trama es muy similar a la de otros videojuegos, pues consiste en rescatar a una princesa de las manos de un ser malvado, esta saga de videojuegos incluye elementos muy interesantes como, por ejemplo, la variedad de enemigos que presenta, las armas que puede utilizar Link, el personaje principal, a lo largo del juego o mazmorras con diversos desafíos entre otras características.

Figura 2.4 *The Legend of Zelda* (1986)

En este proyecto se han tenido en cuenta algunos aspectos de estos videojuegos como el diseño de las interfaces o el uso de puzles a lo largo de los niveles.

3

Diseño de la solución

En este capítulo se explicará detalladamente el proceso de diseño de nuestro juego, desde la especificación de requisitos, así como la elaboración de los documentos de concepto (*High Concept*) y de diseño (*Game Design Document*). También se presentará la planificación del proyecto y los *sprints* realizados.

3.1 Requisitos y especificaciones

La fase de especificación de requisitos es una de las más importantes a la hora de desarrollar un producto de software. Tiene lugar al principio del desarrollo, antes que la implementación y es aquella que marca las bases sobre las que empezará a erguirse el proyecto. Los requisitos están vivos durante todo el ciclo de desarrollo pudiendo aparecer nuevos, sufrir cambios o incluso eliminarse.

A continuación, se expone como quedó finalmente la lista de requisitos del proyecto tras sufrir algunas modificaciones durante el desarrollo del proyecto:

- **RNF1**: El producto final será un videojuego serio que trate el tema del cambio climático.
- **RNF2**: El motor del videojuego será Unity y el lenguaje de programación usado será C#.
- **RNF3**: El objetivo serio del juego será concienciar al jugador del cambio climático.
- **RNF4**: Los recursos audiovisuales que usemos serán gratuitos, descargados de internet y libres de copyright.
- **RNF5:** La plataforma de juego será PC y el S.O. utilizado será Windows 10 en adelante.
- **RF1**: El videojuego pertenecerá al género plataformas, pudiendo incluir los niveles algunos elementos de tipo puzzle.
- **RF2**: El juego será un juego 2D y la cámara tendrá un desplazamiento lateral.
- **RF3**: Se implementará un sistema de preguntas tipo test para evaluar al jugador con el objetivo de educarlo en la materia que trata el juego.
- **RF4**: Para darle más interés a las preguntas, el jugador será recompensado en función de su puntuación.
- **RF5**: El grado de dificultad de las preguntas variará en función del desempeño del jugador en las pruebas realizadas
- **RF6**: Las preguntas se almacenarán en un fichero que el usuario podrá modificar para añadir/modificar/eliminar las preguntas que aparecerán dentro del juego.
- **RNF6**: Las preguntas se le introducirán al jugador a través de algún *Non-Player-Character* (NPC) evaluador que presente algún elemento narrativo para introducir las pruebas e informar de los resultados.

- **RF7**: El personaje principal deberá tener unos controles sencillos e intuitivos.
- **RF8**: El juego presentará un sistema de guardado con varios perfiles que permitirá a los usuarios reanudar desde donde lo dejaron
- **RF9**: El juego incluirá un nivel-tutorial donde se presentarán las mecánicas principales del juego como los controles y los quizzes.
- **RF10**: El juego incluirá, al menos, un nivel que trate alguna causa del cambio climático.
- **RF11**: Si un nivel presenta algún quizz, el jugador no podrá terminar dicho nivel sin haber contestado las preguntas.

3.2 Concepto

El concepto representa la idea inicial de lo que será el juego y queda plasmado en el *High Concept,* también llamado *Game Concept Document (GCD).* Este documento se suele escribir cuando ya se conocen los requisitos y el juego empieza a coger forma en la mente de los desarrolladores.

Al finalizar el documento de concepto se le presentó al cliente, en este caso el tutor del proyecto, quien le dio el visto bueno para continuar con el diseño.

A continuación, vamos a detallar el concepto de nuestro videojuego tal y como aparece en el *High Concept:*

3.2.1 Título

EcoDreams

3.2.2 High Concept Statement

En un mundo cada día más oscuro, deberás luchar contra el cambio climático para cumplir el deseo de uno de tus seres más queridos de volver a tener un planeta limpio y sostenible.

3.2.3 Características

- Es un juego que además de divertido conlleva una educación medioambiental por parte del jugador.
- Contará con varios niveles del tipo plataformas dónde el jugador deberá enfrentarse a una serie de amenazas para lograr realizar una acción combativa contra el cambio climático.
- Tendremos una especie de lobby donde nos encontraremos con un personaje que, tras realizar un determinado nivel, nos realizará alguna pregunta relacionada con la acción o el problema que acabamos de ver. No podremos pasar al siguiente nivel hasta haber respondido bien la pregunta.
- Nuestro personaje tendrá los controles básicos de un personaje de un juego plataformas (movimiento horizontal, salto, …).
- El jugador encontrará diversos ítems a lo largo de los distintos niveles que le ayudarán a superar la zona en la que se encuentra.
- Los niveles están pensados de manera que cada uno represente una amenaza para el medioambiente (por ejemplo: deforestación-un bosque con árboles talados, contaminación-una ciudad llena de coches …).
- También en los distintos niveles habrá pistas ocultas para responder correctamente a la pregunta posterior, aunque a veces será necesario investigar fuera del juego para encontrar algunas respuestas.

3.2.4 Descripción general

-Motivación del jugador

En este juego el jugador deberá superar una serie de niveles basados en problemas medioambientales reales para lograr un objetivo, en este caso conseguir un mundo más sostenible.

-Género

Plataformas con algunos elementos de puzle y quiz.

-Público objetivo

Cualquier jugador amante de los juegos de plataformas con ganas de aprender más sobre cómo actuar ante un problema real tan serio como es el cambio climático.

-Puntos fuertes

- Conciencia de un problema actual muy serio mientras se juega.
- Mezcla de plataformas con algún puzzle necesario para pasar de nivel.
- Pistas para resolver los cuestionarios escondidas por el mundo.
- Además de las pistas proporcionadas se insta al jugador a buscar información fuera del juego con preguntas menos triviales.

-Hardware necesario

El juego está pensado para jugarse en PC.

3.2.5 Más detalles

-Personajes

Además de los diferentes enemigos que iremos encontrando a lo largo de la aventura, tenemos 2 personajes principales:

Player: Un chico acostumbrado a la vida en la ciudad, un buen día visita a su abuela y su perspectiva del mundo cambia radicalmente, necesita hacer algo para que el planeta que tanto añora su abuela resurja.

Abuela: Es la persona más cercana a nuestro protagonista, cuando le transmite a su nieto su deseo de volver a ver un mundo sano y radiante como antaño, provoca en él un sentimiento de cambio salvaje.

-Objetivo serio

Además del objetivo esencial de un videojuego, el de entretener, este juego se marca como reto adicional el de causar en el jugador un sentimiento de reacción contra una amenaza global real como es la lucha por el medioambiente y frenar los efectos del cambio climático. La misión educativa de este proyecto es enseñar a la gente que lo juegue qué acciones podemos realizar en nuestra vida cotidiana para tener un futuro más sostenible en este nuestro planeta, nuestro hogar.

3.2.6 Objetivos del diseño

- Encontrar una buena manera de transmitir el mensaje que queremos a través de la narrativa y de distintas mecánicas como puzles, cuestionarios…
- Controles sencillos: Hacer los controles lo más intuitivos y disfrutables posible. Esto hará que el jugador se familiarice rápido con el personaje y la partida sea más fluida.
- Interfaces amigables y manejables. Nada de menús escondidos o poco entendibles. El usuario deberá ser capaz de encontrar lo que necesita rápido

3.3 Game Design Document

Justo después de terminar el documento de concepto, se empezó a trabajar en el documento de diseño del videojuego, el *Game Design Document (GDD)*.

Este documento está vivo durante todo el desarrollo del juego, pues el diseño puede cambiar durante la implementación y deben quedar reflejadas todas modificaciones que sufra.

En la esta sección se tratarán los aspectos del más importantes del juego reflejados en el GDD, que se podrá encontrar íntegro en los anexos que se encuentran al final de este documento.

3.3.1 Controles del juego

Dispondremos de los controles básicos de un videojuego de plataformas en 2D:

1. Movimiento horizontal: Desplazamiento hacia la izquierda o la derecha utilizando las ***flechas*** o las teclas A o D respectivamente.
2. Salto: Aplicando una fuerza hacia arriba al pulsar la **tecla espaciadora**.
3. **Esc** para pausar/reanudar.
4. **E** para interactuar.
5. **F** para disparar

3.3.2 Interfaces

En esta sección vamos a presentar los bocetos iniciales que se han realizado para imaginar los distintos menús y pantallas que van a aparecer en nuestro juego.

En esta primera versión del GDD se muestran bocetos hechos a mano sobre papel, lo que hace que en la versión final del juego estos puedan variar ligeramente tanto en apariencia como en funcionalidad llegando incluso a aparecer nuevos menús si fueran necesarios.

-INTERFAZ DE JUEGO:

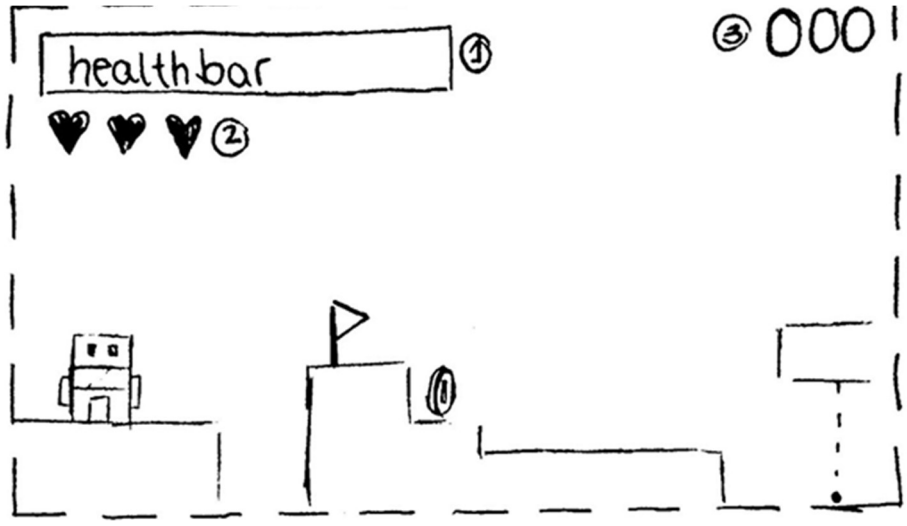


Figura 3.1 *Boceto del HUD de EcoDreams*

Como podemos observar hay varios elementos en la pantalla del jugador que proporcionan información muy útil a la hora de jugar:

1. **Barra de salud**: es un medidor que indica cuánta salud le queda a nuestro personaje antes de que se agote un corazón. Si la barra de salud se vacía el jugador perderá una vida y aparecerá desde el último punto de control.
2. **Corazones**: Indican cuantas vidas le quedan a nuestro jugador antes de que acabe la partida.
3. **Puntuación**: Marcador que registra los puntos que llevamos acumulados por monedas y por derrotar enemigos.

-PANTALLA DE INICIO:

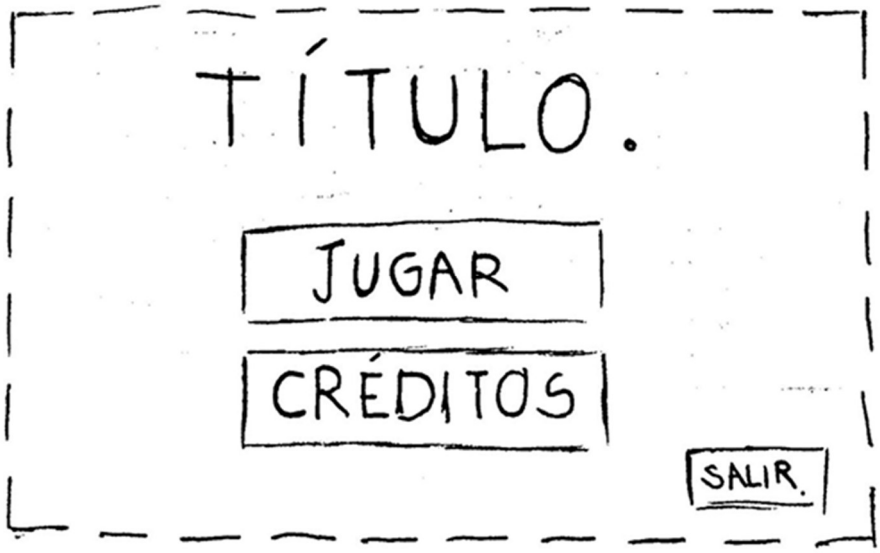


Figura 3.2 *Boceto del menú principal de EcoDreams*

Una pantalla de inicio sencilla donde encontramos el título del juego y algunos botones con distintas funcionalidades: ***Jugar*** para abrir el menú de selección de perfil, ***Créditos*** para ir a la pantalla de información del proyecto y ***Salir*** para cerrar el ejecutable.

A esta pantalla se le podrían añadir algunos menús más como por ejemplo uno de ajustes y quizás alguna pantalla informativa donde poder ver correctamente los controles del juego en un momento dado.

-PANTALLA DE PAUSA:

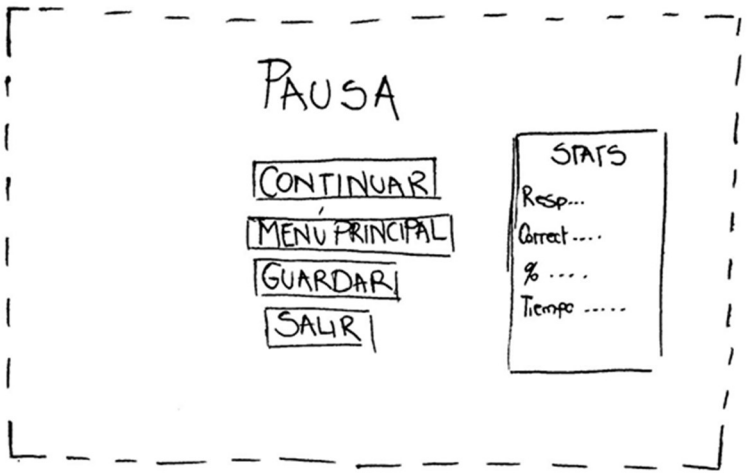


Figura 3.3 *Boceto del menú de pausa de EcoDreams*

Desde este menú tendremos la posibilidad de pausar el juego para después volver a continuar o bien volver al título para operar desde ese menú. También tenemos la posibilidad de cerrar el juego directamente desde esta pantalla o bien de guardar e ir al menú principal.

Además, desde aquí el jugador podrá consultar las estadísticas de su partida en lo que a cuestionarios se refiere y llevar un seguimiento de la dificultad y el progreso de sus datos.

-PANTALLA DE FIN DE PARTIDA:

Figura 3.4 *Boceto de pantalla de GameOver en EcoDreams*

Cuando el jugador gaste todos sus corazones verá esta pantalla desde la que puede:

1) reiniciar el nivel,

2) volver a la pantalla de inicio o

3) salir del juego cerrando el ejecutable.

Si decidimos que queremos reintentar el nivel debemos tener en cuenta que, al haber gastado todas las vidas que nos quedaban, deberemos empezarlo desde el principio.

-PANTALLA DE QUIZ:

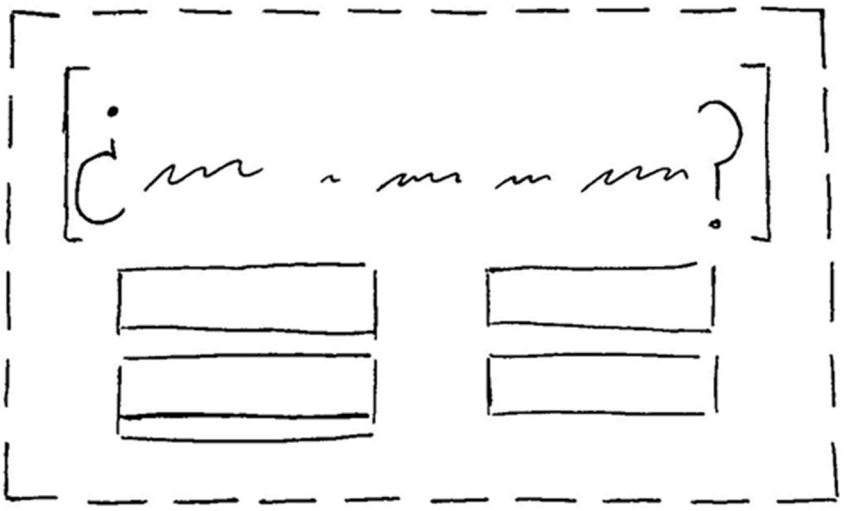


Figura 3.5 *Boceto de panel de quizzes en EcoDreams*

La pantalla de quiz mostrará 1) Un cuadro de texto con la pregunta que tendremos que responder y 2) Una serie de botones con las posibles respuestas a la pregunta formulada. Al pulsar un botón este se marcará en verde si la respuesta es correcta, o rojo si no lo es.

-PANTALLA DE AJUSTES:

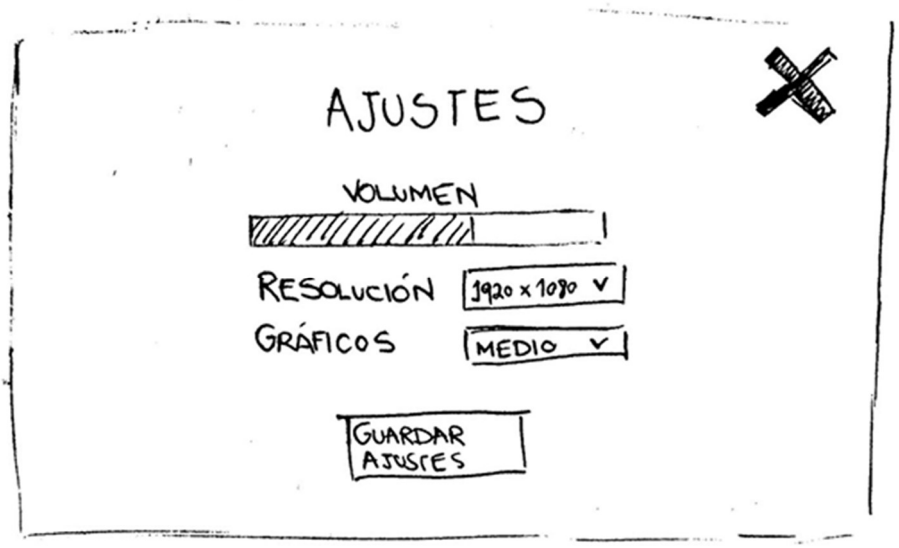


Figura 3.6 *Boceto de menú de configuración de EcoDreams*

A través de esta pantalla podremos modificar los valores de volumen, resolución de la pantalla y calidad gráfica del juego. Si queremos mantener los cambios realizados primero hay que pulsar el botón de guardar cambios, si en su lugar pulsamos salir cerraremos el menú manteniendo la última configuración guardada.

3.3.3 Datos de guardado

Para llevar un registro del estado de la partida de un usuario necesitaremos al menos:

- **Nombre del usuario,** *string*. Para el nombre de perfil e identificación del fichero de guardado.
- **Puntos de salud**, *int*.
- **Número de vidas**, *int*.
- **Puntuación**, *int*.
- **Nombre de la escena** en la que nos quedamos, *string*.
- **Dificultad de la partida**, *int* de 1 a 3. Indica si estamos en dificultad fácil, media o difícil.

- **Tema**, *int*. El tema es un índice que refleja el último nivel que jugamos. Se usa para saber que preguntas hacerle al jugador en función del último nivel que jugó.
- **Estadísticas**, *PlayerStats*. Este campo es un objeto de tipo *PlayerStats*. Esta clase contiene 3 atributos: **questionsAnswered**, *int*; **questionsRight**, *int*; y **rightAnswersPercentage**, *float*.

Estos datos deberían ser suficientes para lograr mantener un registro de partidas de la manera que nosotros queremos. Sin embargo, esto no quiere decir que en un futuro no se puedan añadir más datos de guardado si fueran necesarios, solo habría que añadir la correspondiente variable y hacer los ajustes de código que hicieran falta.

3.3.4 Estructura de las preguntas.

Para almacenar las preguntas que se harán al usuario a lo largo del juego usaremos un fichero JSON que, para cada pregunta, almacenará la siguiente información:

- **Texto de la pregunta**, *string*.
- **Posibles opciones**, array de *Answers*.
- **Índice del array donde se encuentra la respuesta correcta**, *int*.
- Índicador de **dificultad de la pregunta**, *int*.
- Índicador del **tema que trata la pregunta**, *int*.

Estos datos sirven, no solo para definir la pregunta, sino también para poder clasificarlas y acceder a ella de manera más sencilla conociendo su dificultad y el tema que tratan.

3.3.5 Dificultad.

El juego está pensado de manera que cada nivel presente un reto diferente y por tanto una dificultad diferente, sabiendo que es un juego de plataformas y por tanto habrá niveles con saltos y enemigos más complejos que otros.

Sin embargo, para las pruebas didácticas se ha pensado un sistema de *dificultad dinámica* basado en el número de respuestas que ha acertado el jugador a lo largo de la partida. Inicialmente la dificultad será baja y aumentará conforme se acierten más preguntas, pudiendo volver disminuir si el jugador empieza a fallar más a menudo.

3.3.6 Diagrama de flujo para la transición de pantallas

El siguiente diagrama de flujo extraído directamente del GDD indica todas las interacciones que el usuario puede tener con la interfaz de juego a lo largo de la partida.

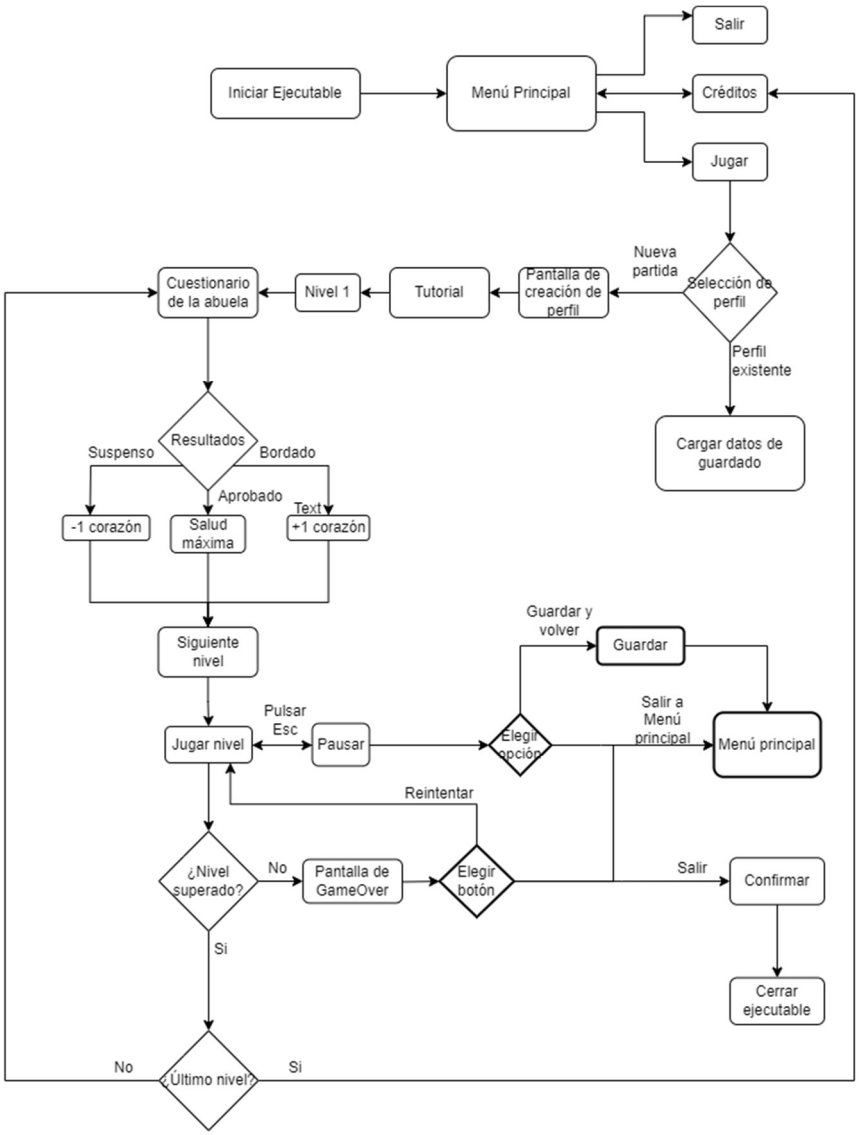


Figura 3.7 *Diagrama de flujo de las interfaces de EcoDreams*

Como aclaración, al cargar datos de guardado llegaremos al último nivel que estábamos jugando y desde ahí el diagrama seguirá su flujo normal al igual que desde el bloque "*Nivel 1*".

3.4 Metodología

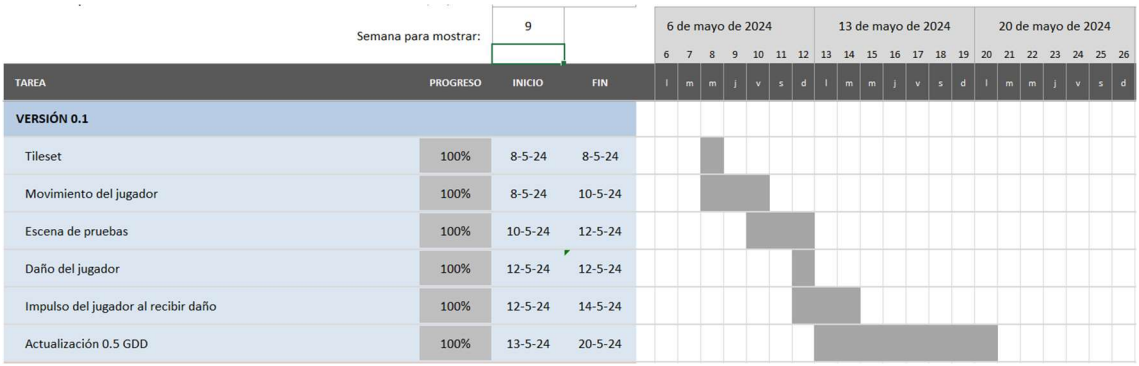
La metodología empleada durante el desarrollo de este proyecto ha sido SCRUM. Como comentábamos en el capítulo 2, SCRUM es una metodología basada en sprints. Esto significa que periódicamente se tienen reuniones con el cliente donde se marcan los objetivos del siguiente sprint y se comprueba que los que se marcaron en el sprint anterior se hayan cumplido satisfactoriamente

3.4.1 Cronograma

Fruto de seguir esta metodología de trabajo, se ha obtenido un diagrama de Gantt, con la planificación del proyecto para cada sprint. De esta manera, no solo tenemos un recurso temporal en el que ver el desarrollo del proyecto, sino también un resumen de todos los sprints, con fecha y tareas realizadas al final de cada uno de ellos.

Figura 3.8 *Sprint 1: Requisitos y concepto*

Figura 3.9 *Sprint 2: GDD*



Semana para mostrar: 9

6 de mayo de 2024 | 13 de mayo de 2024 | 20 de mayo de 2024

TAREA	PROGRESO	INICIO	FIN
VERSIÓN 0.1			
Tileset	100%	8-5-24	8-5-24
Movimiento del jugador	100%	8-5-24	10-5-24
Escena de pruebas	100%	10-5-24	12-5-24
Daño del jugador	100%	12-5-24	12-5-24
Impulso del jugador al recibir daño	100%	12-5-24	14-5-24
Actualización 0.5 GDD	100%	13-5-24	20-5-24

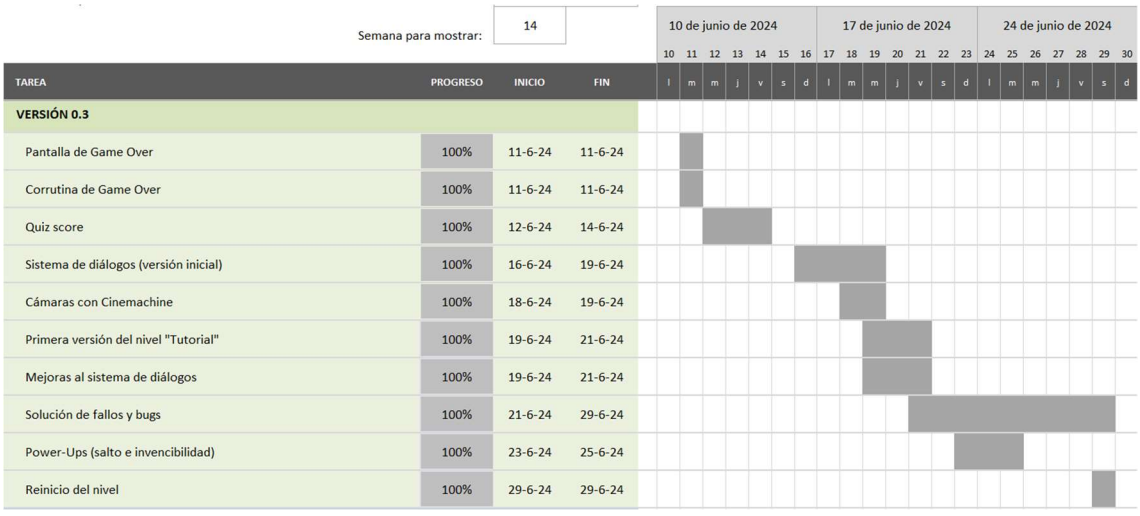
Figura 3.10 *Sprint 3: Versión 0.1*

Semana para mostrar: 12

27 de mayo de 2024 | 3 de junio de 2024

TAREA	PROGRESO	INICIO	FIN
VERSIÓN 0.2			
Menú Principal	100%	24-5-24	28-5-24
Menú de pausa	100%	26-5-24	28-5-24
Pruebas con JSON para carga de datos	100%	29-5-24	31-5-24
Pociones	100%	1-6-24	1-6-24
GameManager	100%	1-6-24	3-6-24
HUD (Head Up Display)	100%	1-6-24	5-6-24
Quiz funcional	100%	6-6-24	7-6-24
Ronda de solución de fallos y bugs	100%	31-5-24	7-6-24

Figura 3.11 *Sprint 4: Versión 0.2*



Semana para mostrar: 14

10 de junio de 2024 | 17 de junio de 2024 | 24 de junio de 2024

TAREA	PROGRESO	INICIO	FIN
VERSIÓN 0.3			
Pantalla de Game Over	100%	11-6-24	11-6-24
Corrutina de Game Over	100%	11-6-24	11-6-24
Quiz score	100%	12-6-24	14-6-24
Sistema de diálogos (versión inicial)	100%	16-6-24	19-6-24
Cámaras con Cinemachine	100%	18-6-24	19-6-24
Primera versión del nivel "Tutorial"	100%	19-6-24	21-6-24
Mejoras al sistema de diálogos	100%	19-6-24	21-6-24
Solución de fallos y bugs	100%	21-6-24	29-6-24
Power-Ups (salto e invencibilidad)	100%	23-6-24	25-6-24
Reinicio del nivel	100%	29-6-24	29-6-24

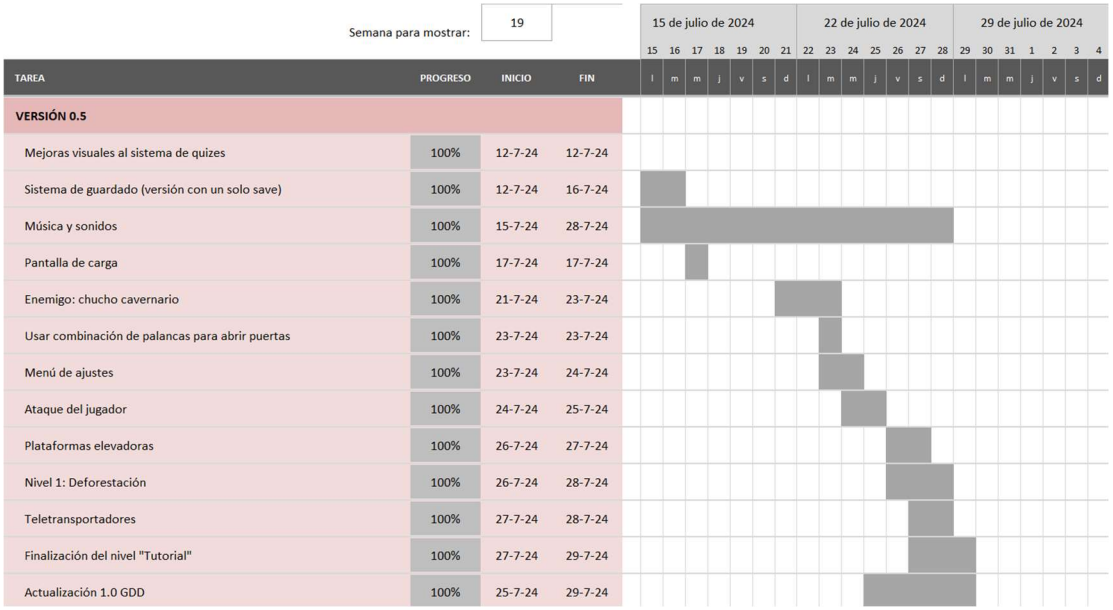
Figura 3.12 *Sprint 5: Versión 0.3*

Semana para mostrar: 17

1 de julio de 2024 — 8 de julio de 2024

TAREA	PROGRESO	INICIO	FIN
VERSIÓN 0.4			
Monedas	100%	2-7-24	2-7-24
Caida al vacío	100%	3-7-24	3-7-24
Checkpoints	100%	3-7-24	5-7-24
Cofres	100%	6-7-24	7-7-24
Puertas y palancas	100%	7-7-24	9-7-24
Sistema de diálogos (versión final)	100%	9-7-24	10-7-24
Pantalla de Controles	100%	10-7-24	10-7-24
Resolución de fallos	100%	9-7-24	11-7-24

Figura 3.13 *Sprint 6: Versión 0.4*



Semana para mostrar: 19

15 de julio de 2024 — 22 de julio de 2024 — 29 de julio de 2024

TAREA	PROGRESO	INICIO	FIN
VERSIÓN 0.5			
Mejoras visuales al sistema de quizes	100%	12-7-24	12-7-24
Sistema de guardado (versión con un solo save)	100%	12-7-24	16-7-24
Música y sonidos	100%	15-7-24	28-7-24
Pantalla de carga	100%	17-7-24	17-7-24
Enemigo: chucho cavernario	100%	21-7-24	23-7-24
Usar combinación de palancas para abrir puertas	100%	23-7-24	23-7-24
Menú de ajustes	100%	23-7-24	24-7-24
Ataque del jugador	100%	24-7-24	25-7-24
Plataformas elevadoras	100%	26-7-24	27-7-24
Nivel 1: Deforestación	100%	26-7-24	28-7-24
Teletransportadores	100%	27-7-24	28-7-24
Finalización del nivel "Tutorial"	100%	27-7-24	29-7-24
Actualización 1.0 GDD	100%	25-7-24	29-7-24

Figura 3.14 *Sprint 7: Versión 0.5*

Semana para mostrar: 21

29 de julio de 2024 — 5 de agosto de 2024 — 12 de agosto de 2024

TAREA	PROGRESO	INICIO	FIN
VERSIÓN 0.6			
Sistema de dificultad progresiva	100%	2-8-24	3-8-24
Mejora del Quiz System (clasificación de preguntas)	100%	2-8-24	3-8-24
Sistemas de partículas para efectos visuales	100%	3-8-24	5-8-24
Respawn de enemigos	100%	4-8-24	4-8-24
Nuevo sistema de guardado con varios perfiles de usuario	100%	5-8-24	7-8-24
Actualización 1.5 GDD	100%	5-8-24	9-8-24
Implementación panel de estadísticas	100%	8-8-24	9-8-24
Preparación fase de pruebas	100%	10-8-24	14-8-24
Solución de fallos y bugs	100%	3-8-24	12-8-24
Redacción de la memoria	100%	1-8-24	4-9-24

Figura 3.15 *Sprint 8: Versión 0.6*

Figura 3.16 *Sprint 9: Pruebas y redacción de la memoria*

4

Implementación de la solución

Este capítulo está dedicado a toda la fase de implementación del proyecto. En él hablaremos de las clases que se han desarrollado para dar vida a nuestro juego y de la integración de los componentes resultantes para obtener el producto final.

4.1 Niveles

Para el diseño de los niveles del juego hubo que considerar algunos aspectos para darle personalidad al nivel y caracterizarlo en base a lo que se quiere transmitir. En esta sección se hablará de todo lo necesario para crear la base de un nivel: mapa, cámara y sonido.

4.1.1 Mapa del nivel

Un *Tilemap* en Unity es un componente que se encarga de almacenar y gestionar las tiles o casillas que conforman nuestro nivel. Para crear un suelo sobre el que se sostenga el jugador, el *Tilemap* le pasa la información de los tiles que lo forman al componente *Tilemap Collider 2D* y este dispone las colisiones en

base a la forma que generan el conjunto de todas ellos.

Pero un *Tilemap* por sí solo no sirve de mucho, siempre es hijo de un objeto *Grid*. Esto es una cuadrícula que dividirá la escena en casillas dónde podremos dibujar diferentes tiles y combinarlos para darle forma al nivel.

En la siguiente figura podemos observar la cuadrícula del *Grid* y resaltado en naranja el *TileMap* que conforma el suelo del nivel.

Figura 4.1 *Grid del nivel de prueba*

Por último, necesitaremos una herramienta para dibujar los tiles y esa es la *Tile Palette,* que nos permite crear nuestras propias paletas de tiles para después usarlas en la creación de nuestros niveles.

Una vez creada la paleta podremos acceder a ella cuándo queramos y utilizar las distintas opciones que proporciona la herramienta para modificar los niveles.

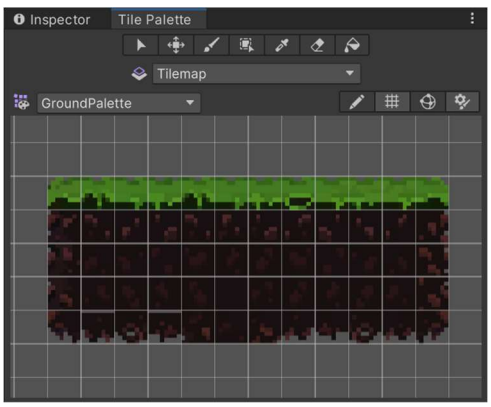


Figura 4.2 *Tile Palette usado en algunos niveles*

4.1.2 Cámara

La cámara es otro de los elementos protagonistas en los niveles de un videojuego. Hay que pensar que tipo de cámara se quiere utilizar y configurarla para que el desplazamiento no se vea raro o poco profesional.

Para este juego se pensó en una cámara lateral en 2D y gracias a otra de las herramientas de Unity, *Cinemachine,* se logró implementar una cámara que siguiera al jugador de manera suave y respetando los límites del nivel.

Cinemachine crea cámaras virtuales en la escena que se usan para controlar objetos del tipo *Camera* de Unity. Es decir, no son cámaras en sí, sino más bien componentes que se encargan de dirigir las cámaras existentes en escena, algo así como un director de cine.

Al introducir una cámara virtual en escena automáticamente se le añade un componente *Cinemachine Brain* a la cámara principal.

Para respetar los límites del nivel y que la cámara no enseñe partes que no deberían verse del nivel se usa una extensión llamada *Cinemachine Confiner 2D* la cual recibe un collider (al cual no podemos olvidar quitarle las colisiones para que nuestro jugador pueda moverse dentro de él) que delimitará el área en la que puede moverse la cámara.

Figura 4.3 *Límites de la cámara virtual*

Como se puede observar en la **Figura** 4.3 la cámara virtual designa unas zonas en las que puede moverse el objetivo, siendo la más exterior una zona de transición dura, la de en medio una zona de transición suave y la más interna una zona sin transición.

4.1.3 Sonido

Para la música de los niveles se creó un GameObject con un componente *AudioSource* exclusivamente para ello. En este componente se puede asignar una pista de audio y reproducirla en bucle controlando parámetros como el volumen con respecto a otros sonidos, la importancia que se le quiere dar a la pista en cuestión...

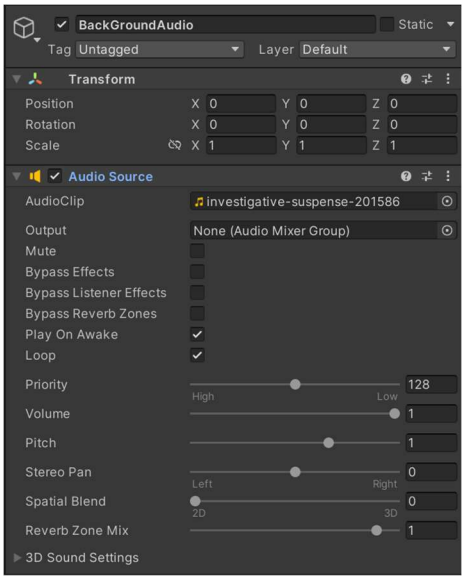


Figura 4.4 *GameObject con Audio Source para música de fondo*

4.2 GameManager

Para entender gran parte de la implementación de este juego, es necesario introducir la clase *GameManager*. Esta clase está implementada basándose en el patrón Singleton, esto nos asegura que solo habrá una instancia de esta clase proporcionando un punto de acceso global a ella ("Singleton", 2023).

Su principal función es mantener la coherencia de datos entre escenas y cualquier modificación que sufran dichos datos, bien desde el propio *GameManager*, bien desde otra clase, la realizará la instancia de *GameManager*.

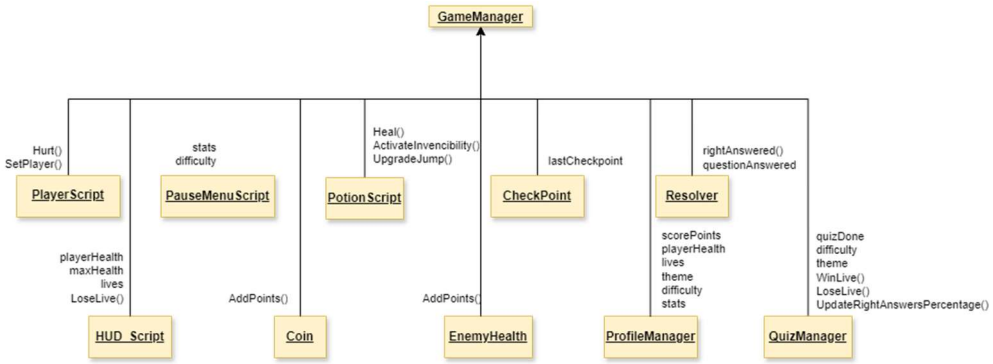


Figura 4.5 *Relación entre clases con el GameManager*

Aunque un *GameManager* se puede usar para muchísimas más cosas como gestión de eventos en el juego, nosotros lo hemos usado exclusivamente para mantener la persistencia de datos entre escenas.

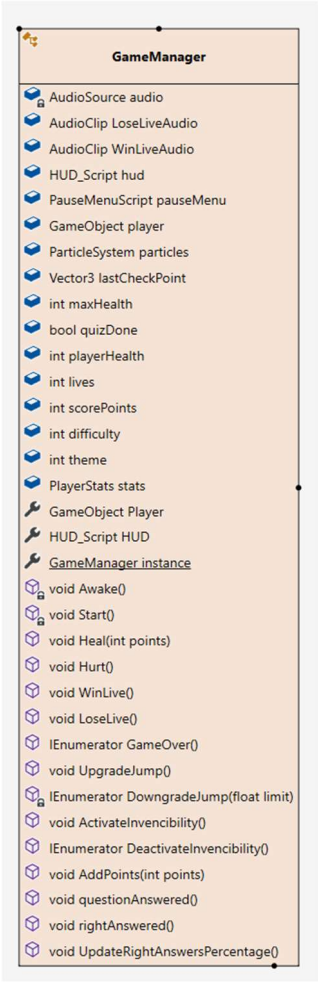


Figura 4.6 *Diagrama UML de la clase GameManager*

Como se puede observar en la **Figura** 4.6 *Diagrama UML de la clase GameManager* esta clase incluye las variables que se definieron en el GDD para llevar el registro de la partida y que son susceptibles de cambiar (desde *playerHealth* hasta *stats*).

De la misma forma encontramos una variable para almacenar una referencia al jugador, otra al HUD y una última al menú de pausa. Esto lo hacemos así que ya que el jugador será quien llame a los métodos para actualizar las variables de estado de la partida y el HUD, junto al menú de pausa, serán quienes se encarguen de mostrar al jugador los datos que administra el *GameManager* y por tanto debemos poder acceder a ellos desde esta clase.

También encontramos la instancia *instance* del GameManager que se inicializa en el método *Awake()* en función de si ya existe o no una instancia. En caso de que ya existiera una instancia viva se llama al método *Destroy()* para eliminar la nueva. La persistencia de los datos entre escenas se consigue usando un método llamado *DontDestroyOnLoad()*, el cual, como su propio nombre indica, añade la instancia a un grupo de objetos que están vivos durante toda la ejecución del programa y que no se destruyen al cambiar de escena.

El resto de los métodos que encontramos son métodos sencillos para actualizar las variables de la partida en momentos dados. Por ejemplo *Hurt()* resta puntos de vida al jugador y el comprueba si pierde un corazón o no, *AddPoints(int points)* suma tantos puntos como se indiquen al marcador…

Desde esta clase se controlan también los efectos de los power-Ups de salto e invencibilidad, así como su duración a través de las corrutinas *DowngradeJump()* y *DeactivateInvencibilty()*.

4.3 Personaje del jugador

A continuación, vamos a hablar del elemento nexo entre el usuario y el videojuego, su personaje. Este elemento es crucial en un videojuego, pues gracias

él se puede interactuar con el entorno y avanzar en la partida.

4.3.1 Movimiento y salud

El comportamiento del jugador se implementó en un script llamado *PlayerScript.* Desde él se controla el movimiento del jugador y las interacciones que tienen que ver con su sistema de salud.

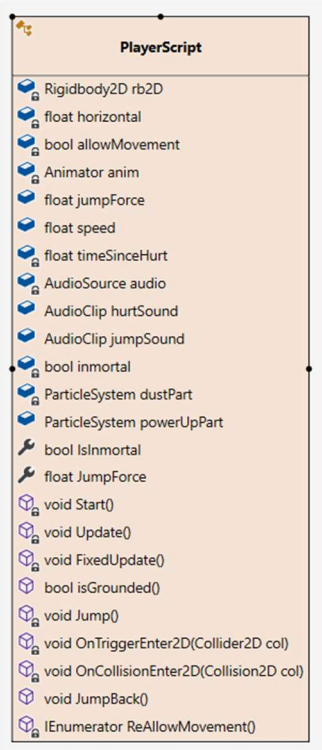


Figura 4.7 *Diagrama UML de la clase PlayerScript*

Hay varias maneras de mover objetos en Unity, sin embargo, aquí se ha optado por mover a nuestro personaje usando el componente *Rigidbody 2D.* Este componente proporciona al objeto que lo implementa un sistema de físicas con varias opciones. Una de ellas es *velocity* la cual aplica una velocidad fija en la dirección que se le indique. En este caso la dirección la indica el usuario final al pulsar las teclas A, D o las flechas.

Para el salto le aplicamos una fuerza de impulso al rigidbody, haciendo uso del método *AddForce()* propio del componente, hacia arriba al pulsar la tecla de espacio. Se debía controlar que el jugador no pudiera saltar de manera indefinida, si no que al dar un salto no se pudiera dar el siguiente hasta estar de nuevo en el

suelo. Para conseguir este efecto, se trazó un Raycast hacía el suelo con una distancia suficiente como para que solo se pudiera dar un salto con el detectar cuando el jugador estaba en el aire o no.

Una vez definido como se iba a mover el jugador hay que juntar todas las piezas. Digamos que el "cerebro" del personaje se implementa en el método *Update().* Este método viene dado por la interfaz *MonoBehaviour* que deben implementar todos los scripts que quieran añadirse como componentes de un objeto en Unity. *Update()* se ejecuta a cada frame del juego y es un método perfecto para cuando hay que estar haciendo comprobaciones todo el rato, como es este caso. Cada vez que el jugador pulsa la tecla de espacio, el método lo detecta y llama al método *Jump(),* siempre y cuando *isGrounded()* devuelva el valor *true.* Aún más notorio es cuando se trata del movimiento horizontal del jugador, pues debe estar todo el rato leyendo cuando se pulsa una de las teclas que activa el movimiento y cuando se deja de pulsar para que el jugador camine o se quede quieto.

Por último, tenemos métodos para detectar cuando el jugador ha entrado en contacto con algún objeto que tiene un efecto sobre él. En este caso usamos *OnTriggerEnter2D* y *OnCollisionEnter2D* para saber cuándo el jugador ha entrado en contacto con algo que le causa daño. Si sufre algún daño se hace una llamada al método *Hurt()* de la clase *GameManager* a través de su instancia para restarle 1 punto de vida al jugador y comprobar si ha agotado su barra de salud para quitarle una vida a través del método *LoseLive()* también perteneciente al *GameManager*. Además, de vuelta en *PlayerScript,* también se llama al método *JumpBack()* que impulsa al jugador hacia atrás, bloquea su movimiento y llama a la corrutina *ReAllowMovement()* que esperará a que *isGrounded()* devuelva *true* para volver a activar el movimiento.

4.3.2 Ataque

Otra de las mecánicas de nuestro juego que está ligada directamente al jugador es el ataque de este.

Está implementada en un script a parte que inicialmente está desactivado pero que se activa cuando el jugador coge un arma.

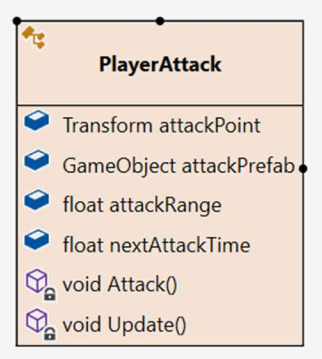


Figura 4.8 *Diagrama UML de la clase PlayerAttack*

Este script tiene 3 variables públicas que sirven al desarrollador para definir:

1. ***attackPoint:*** el punto de origen del que saldrá el disparo.
2. ***attackPrefab:*** el prefab que quiere usar para el disparo.
3. ***attackRange:*** la distancia que recorrerá el proyectil antes de destruirse.
4. ***nextAttackTime:*** el tiempo que tarda el proyectil en destruirse.

Al igual que el movimiento aquí tenemos un *Update()* que se encarga de disparar llamando al método *Attack()* cuando el jugador presione la tecla F, siempre y cuando el tiempo que haya pasado desde la última vez que se disparó sea mayor que *nextAttackTime.*

Por último, el método Attack crea una instancia del *attackPrefab* en el punto *attackPoint*, le aplica una fuerza en la dirección que esté mirando el jugador y destruye la instancia cuando pase el tiempo especificado en *attackRange.*

Pero como comentábamos anteriormente para que este script se active es necesario que el jugador coja un arma entrando en contacto con ella. A continuación, vamos a detallar la implementación del arma y sus componentes.

Lo primero que se debía hacer era fabricar un prefab del proyectil que queríamos lanzar para poder instanciarlo cada vez que se quisiera disparar.

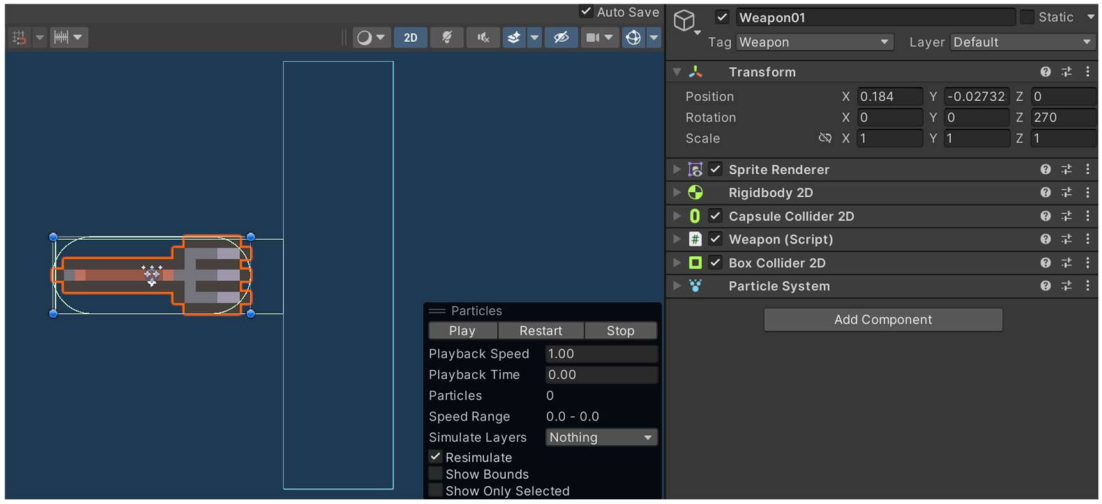


Figura 4.9 *Prefab del arma Weapon01*

Como se observa el prefab contiene un *Rigidbody 2D* usado para añadirle las físicas que permitirán impulsar el proyectil cuando sea necesario, y dos *Colliders 2D,* uno con forma de capsula y que es trigger usado para detectar al jugador y activar su script de ataque; y otro en forma de caja que usamos para proveerle colisiones al proyectil y que detecte cuando ha chocado con algo para activar su efecto de destruirse.

Todo esto se controla desde una clase llamada *Weapon* muy simple. En la siguiente figura se muestra su diagrama UML:

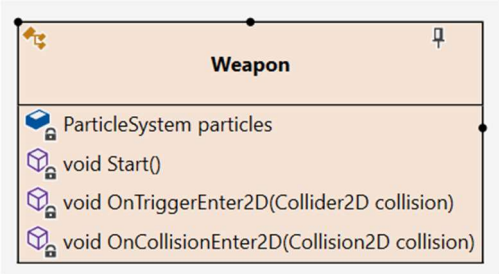


Figura 4.10 *Diagrama UML de la clase Weapon*

Como se observa es una clase muy sencilla la cual implementa los métodos para detectar cuando el jugador entra en contacto con el trigger del arma y cuando colisiona, usando un efecto de partículas para conseguir la sensación de que ha impactado contra algo.

Con todo esto desarrollado el personaje principal queda preparado como se observa en la **Figura** 4.11 y listo para usarlo en nuestros niveles.

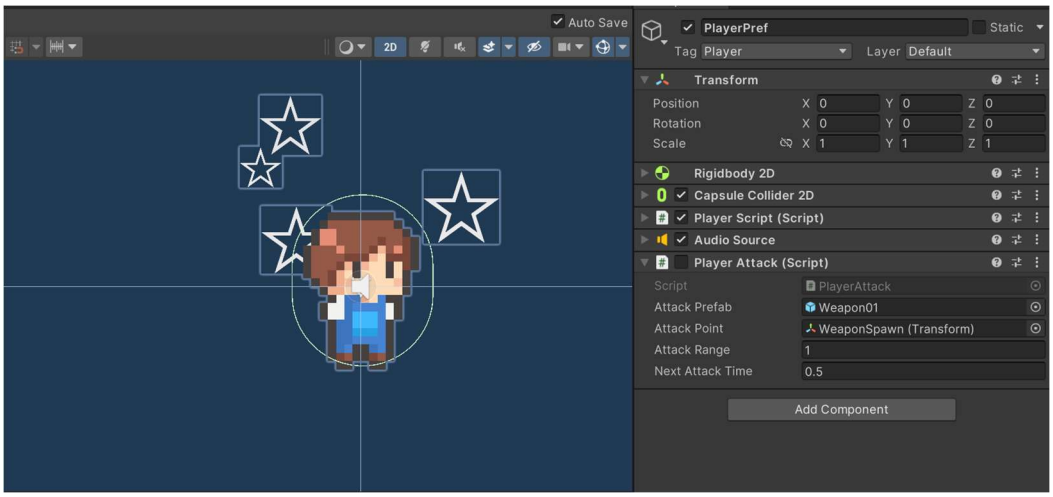


Figura 4.11 *Prefab del personaje del jugador*

4.4 Elementos para la creación de niveles

En esta sección hablaremos de los diferentes objetos que podemos encontrar a lo largo de nuestra aventura y que servirán para actualizar los parámetros del jugador, desbloquear comportamientos especiales y darle mayor dinamismo a los niveles. Esto abarca desde los distintos consumibles que ayudarán al jugador, así como elementos para la gestión de eventos del nivel como las muertes por caída al vacío o el cambio de escenas.

4.4.1 Pociones

Durante la partida podremos encontrar pociones de diferentes colores que causarán diferentes efectos en el jugador:

- Poción roja: regenera vida al jugador
- Poción azul: aumenta la fuerza de salto del jugador permitiéndole saltar más alto durante un breve periodo de tiempo. La fuerza de salto va

disminuyendo a medida que avanza el tiempo y el salto vuelve a la normalidad.

- Poción blanca: poción de invencibilidad. Vuelve al jugador invencible durante unos segundos, dándole ventaja para eliminar a algunos enemigos (o simplemente esquivarlos) más fácilmente.

El comportamiento de las pociones está controlado mediante un script llamado *PotionScript* que hace uso de un tipo enumerado llamado *PotionType* para que el desarrollador pueda elegir el comportamiento de cada poción según sus necesidades.

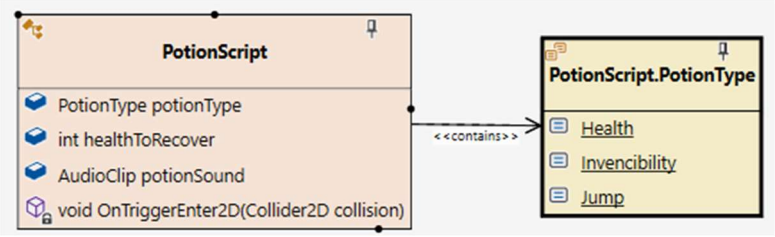


Figura 4.12 *Diagrama UML de la clase PotionScript*

Como podemos observar en la **Figura** 4.12 tenemos una variable que indica el tipo de poción que queremos colocar en nuestra escena y un método *onTriggerEnter2D()* que detectará cuando nuestro jugador ha recogido la poción y en función del tipo actuará de las siguientes maneras:

- ***Health**:* llamará al método *Heal()* de *GameManager* a través de su instancia y sumará al jugador la cantidad de vida especificada en *healthToRecover.*
- ***Invencibility**:* de la misma manera que antes llamará al método *ActivateInvencibility()* que pondrá a *true* la variable *isInmortal* del *PlayerScript,* lo que hará que nuestro jugador no pueda recibir daño hasta que vuelva a activarse al final de la corrutina de *GameManager* llamada *DeactivateInvencibility().*
- ***Jump**:* igual que la de invencibilidad, solo que en este caso se llama al método *UpgradeJump()* para después llamar a la corrutina

DowngradeJump() para ir reduciendo la potencia de salto en función del tiempo de actividad de la poción hasta que vuelve a ser la original.

Tras haber aplicado el efecto de la poción usamos el componente *audioSource* del *GameManager* para aplicar el sonido de recogida y acto seguido destruimos el *GameObject* de la poción para que no pueda ser reutilizada.

Figura 4.13 *Prefabs de las pociones*

4.4.2 Monedas

Otra mecánica de nuestro juego es el sistema de puntos a través de la recolección de monedas. Al igual que la salud del jugador, los puntos son un dato que gestiona el *GameManager* y por ende habrá que acceder a él cuando recojamos alguna moneda.

Las monedas están programadas de una manera bastante similar a las pociones, con la única diferencia de que todas las monedas van a tener el mismo comportamiento y no hay que diferenciarlas.

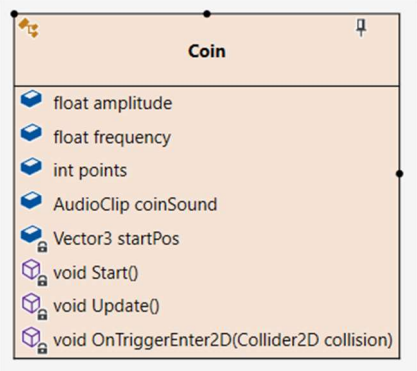


Figura 4.14 *Diagrama UML de la clase Coin*

Otra diferencia con respecto a las pociones es que, al no tener animación, se tuvo que crear mediante programación un efecto de flote para darles algo de

vida. Por eso usamos el método *Start()* para almacenar la posición de referencia de la moneda, y el método *Update()* para generar ese efecto de estar flotando a través de actualizaciones de la posición en cada frame. Se usan los parámetros *amplitude* y *frequency* para determinar la altura y la velocidad a la que queremos que flote la moneda.

Cuando el jugador entra en contacto con la moneda a través de su trigger, se hace una llamada al método *AddPoints(int points)* de *GameManager,* se suman los puntos especificados en la variable *points,* se reproduce el sonido *coinsSound* y se destruye el *GameObject* de la moneda.

4.4.3 Cofres

A medida que avancemos en los distintos niveles el jugador encontrará algún que otro cofre que puede sacarlo de un apuro en un momento dado.

La lógica de los cofres se implementa en la siguiente clase:

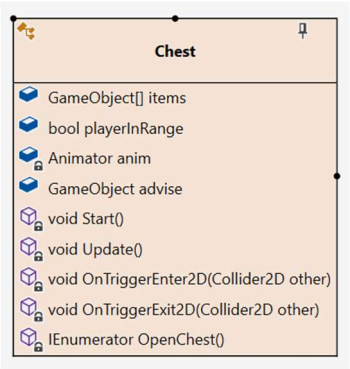


Figura 4.15 *Diagrama UML de la clase Chest*

Como se puede observar, la clase contiene un array de *GameObjects* que definirá los objetos que pueden obtenerse del cofre, un booleano que indica cuando podemos abrir el cofre, el *Animator* que reproducirá la animación de apertura del cofre, y un *advise* que está formado por una imagen que se hará visible cuando el jugador pueda abrir el cofre para indicarle que puede hacerlo mediante la tecla E.

En el método *Start()* inicializamos la variable *anim*, utilizando el componente *Animator* que contiene el objeto *Chest*.

En *Update()* comprobamos continuamente si se abre el cofre o no, es decir, si el jugador se encuentra en la distancia de apertura del cofre (indicado por *playerInRange)* y si ha pulsado E estando dentro de ese rango. Si finalmente el cofre se abre, se inicia la corrutina *OpenChest(),* la cual lanza la animación de apertura del cofre y crea una instancia de un objeto aleatorio del array *ítems* por encima del cofre.

Los métodos *OnTriggerEnter2D() y OnTriggerExit2D()* solamente sirven para comprobar si el jugador está en rango de apertura, dándole el valor *true* a la variable *playerInRange* si el jugador está en el trigger collider del cofre o *false* si se encuentra fuera. De la misma forma también activa o desactiva la visibilidad del aviso de apertura *advise.*

4.4.4 Checkpoints

El juego también implementa un sistema de guardado de posiciones para que el jugador pueda reaparecer desde el último punto de control cuando pierda una vida. Sin embargo, si finalmente el jugador pierde todas sus vidas, los checkpoints recogidos se restaurarán forzando al jugador a iniciar el nivel desde el principio.

Figura 4.16 *Checkpoint desactivado*

Figura 4.17 *Checkpoint activado*

El comportamiento de los checkpoints es muy sencillo y se implementa en el script *CheckPoint.*

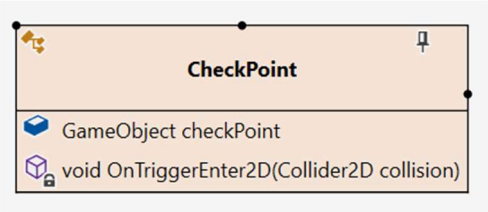


Figura 4.18 *Diagrama UML de la clase CheckPoint*

Lo único que contiene es una referencia al objeto *checkPoint* a través de la cual accede al sprite del círculo para cambiarlo de color cuando cambie de estado.

El cambio de estado se produce cuando el jugador pasa por el checkpoint, activando el método *OnTriggerEnter2D* que cambia el color del círculo a verde y guarda en la variable *lastCheckPoint* de la instancia del *GameObject* la posición del punto de control para que el jugador reaparezca en esa posición cuando pierda una vida.

Los checkpoints solo pueden activarse una vez, por lo que una vez que activemos uno, no podremos volver a reaparecer en el que activamos anteriormente.

4.4.5 Puntos de teletransporte

Los puntos de teletransporte se utilizan cuando el jugador llega al final del nivel y se puede pasar de una escena a otra. Este objeto es imperceptible por el jugador ya que tan solo consta de un *Box Collider 2D* con la propiedad *Trigger* activada y un script para controlar el comportamiento de este elemento.

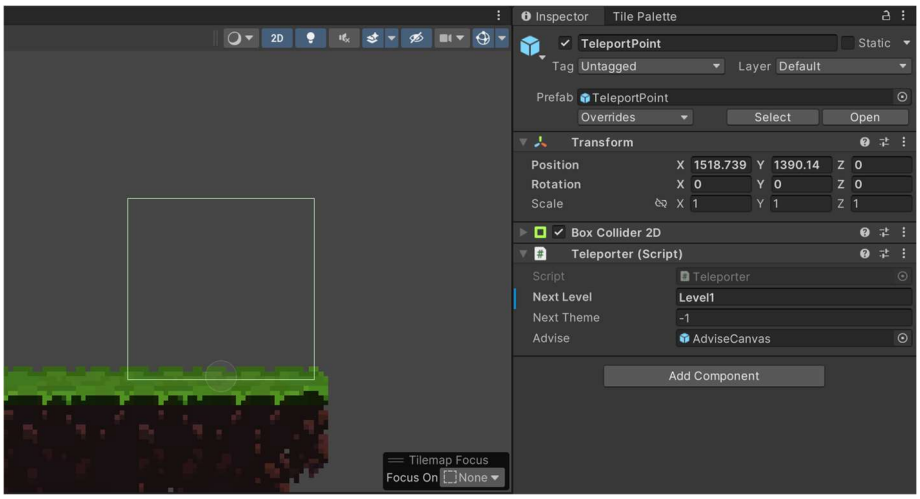


Figura 4.19 *GameObject de un Teleport Point*

El script *Teleporter* se encarga de llevar al jugador al nivel indicado en *NextLevel* y cambiar el tema de las preguntas por el que se especifica en *NextTheme* siempre y cuando se haya superado el quiz necesario para poder pasar de fase (si es que lo hubiese). Si no se cumpliera la condición para pasar de nivel se mostraría por pantalla el aviso indicado en *Advise* mientras el jugador esté dentro del área de teletransporte.

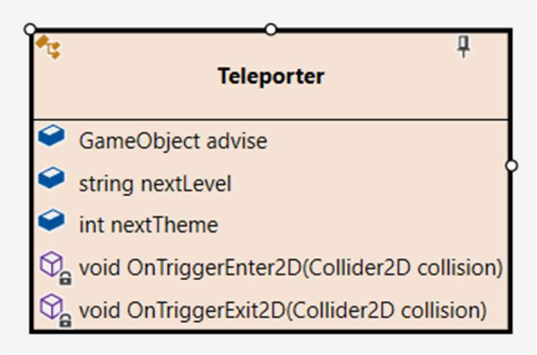


Figura 4.20 *Diagrama UML de la clase Teleporter*

4.4.6 Zonas de muerte por caída

La caída al vacío es uno de los elementos que aportan personalidad a un juego de plataformas, pues hace que el jugador sienta una motivación para llegar a la siguiente plataforma, ya que si no lo consigue puede perder una vida, o peor, perder la partida y tener que empezar el nivel desde el principio.

En este juego se ha implementado la caída al vacío de una manera similar a los puntos de teletransporte: un *Collider2D* y un script que controla lo que sucede si un GameObject con el tag "*Player*" entra en contacto con el *trigger* de la zona de caída.

Si el jugador entra en el *trigger* del colisionador de la zona de muerte por caída, este automáticamente perderá una vida a través de una llamada al método *LoseLive()* de la instancia del *GameManager* y reaparecerá en el último punto de control, que como ya dijimos antes, está almacenado en la variable *lastCheckPoint* también perteneciente a la instancia del *GameManager*.

4.4.7 Palancas

Las palancas inicialmente se pensaron como una herramienta simple para que el jugador tuviera que explorar el nivel para poder abrir alguna puerta y salir, funcionando como llave.

Sin embargo, a medida que avanzaba el proyecto surgieron nuevas formas de utilizar las palancas. A continuación, hablaremos de ellas, y de sus distintas funciones.

- *Palanca como accionador de una plataforma elevadora:* Esta forma de utilizar las palancas nace de la idea de querer ocultar ciertas zonas del nivel a las que solo se pueden llegar haciendo uso de ascensores. Al activar (o desactivar) la palanca el ascensor leerá el valor de la variable *isActivated* e irá a la posición *positionUp* si es verdadero o a *positionDown* si es falso.

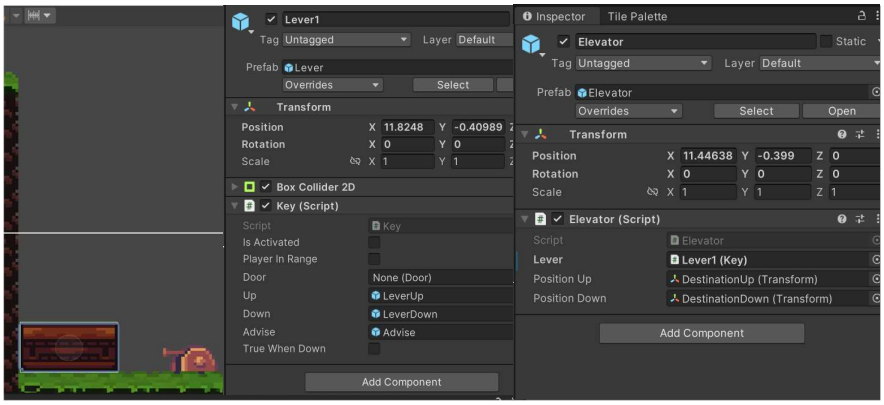


Figura 4.21 *Elevador asociado a una palanca*

- *Palanca como llave de una puerta:* El comportamiento original pensado para las palancas, se puede definir si queremos que la puerta se abra cuando la palanca esté activada. En este caso con solo accionar la palanca se abrirá la puerta asociada hacia arriba o hacia abajo en función del valor de *openUp,* booleano de la clase *Door.*

- *Conjunto de palancas como código para abrir una puerta:* Esta última forma de utilizar las palancas surge como una extensión de la anterior. ¿Por qué no aprovechar este sistema y añadir un puzle que sirva para cumplir el objetivo serio de nuestro juego? En este caso se pensó que teniendo ya una palanca que abría una puerta, ampliar este comportamiento a más de una palanca era tan sencillo como almacenar en un array todas las palancas que abren la puerta, y cada vez que se cambia el valor de una palanca, comprobar si los valores de todas ellas es correcto, usando el método *checkLevers()* de la puerta asociada a la palanca. En caso de que *isActivated* sea true para cada palanca se abrirá la puerta. Para que el creador del nivel pudiera decidir en qué posición la palanca estaba activa se creó un booleano, *trueWhenDown* que permite elegir si la palanca se activa al bajarla (por defecto está activa cuando está arriba).

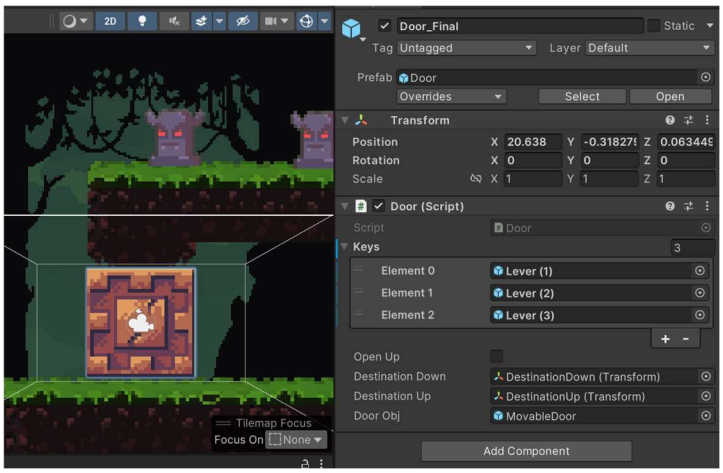


Figura 4.22 *Puerta asociada a varias palancas*

Figura 4.23 *Palanca abajo*

Figura 4.24 *Palanca arriba*

En la siguiente figura podemos observar el diagrama UML que relaciona las palancas (o keys) con las plataformas elevadoras y las puertas:

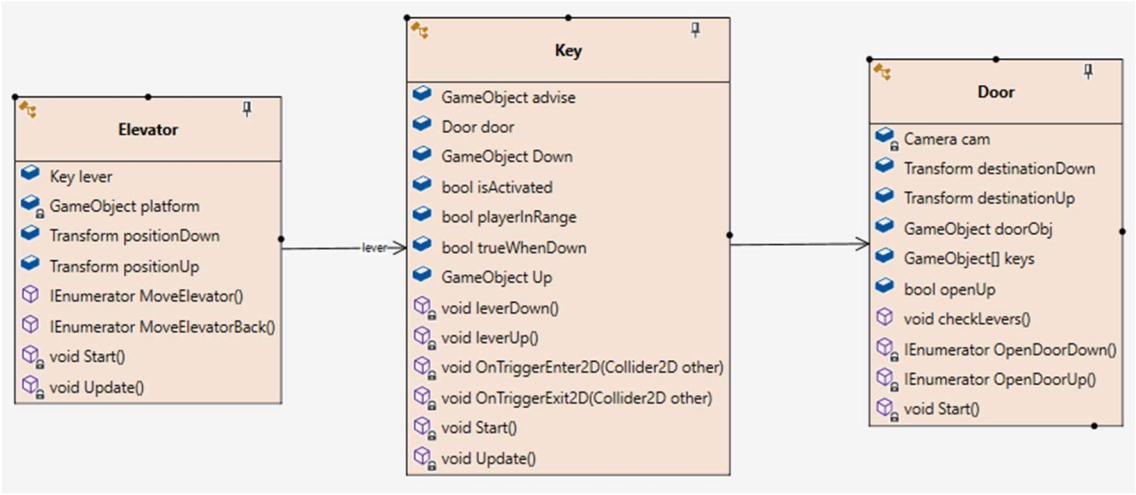


Figura 4.25 *Diagrama UML de las clases Elevator, Key y Door*

4.5 Enemigos

Para darle vitalidad a los niveles, se desarrollaron 2 tipos de *Non-Player-Characters* (NPCs) enemigos, uno estático (actúa como un obstáculo removible) y uno que hace uso de una inteligencia artificial (IA) para manejar los distintos estados que posee.

Con dicho fin se desarrolló un script, común para todos los enemigos, para controlar la salud de estos y como cambia conforme se interactúa con ellos, y otro script para el enemigo inteligente que contiene toda la lógica de los cambios de estado.

4.5.1 Salud de los enemigos

Para poder derrotar a los enemigos y darles también cierta utilidad a estos personajes dentro del juego, es necesario que estos estén dotados de un sistema de salud que permita eliminarlos y que vuelvan a aparecer.

Por eso se desarrolló el script *EnemyHealth.* Este script contiene las variables y los métodos necesarios para manejar el sistema de vidas del enemigo. Por ejemplo, con la variable *health* controlamos la salud actual del enemigo, con *maxHealth* el límite de salud máxima, y también se manejan otros aspectos como

el número de puntos que obtiene el jugador al derrotar al enemigo con *pointsForEnemy* o si puede reaparecer o no uno vez eliminado con *canRespawn.*

Los métodos principales de esta clase son *TakeDamage(int damage)* que resta al enemigo los puntos especificados en la variable que recibe como argumento y comprueba si lo ha eliminado o no para iniciar la corrutina *Die()*, otro de los métodos de esta clase. Con *Die()* se ejecuta la animación de muerte del enemigo y, a continuación, si este no puede reaparecer se elimina el *GameObject* del enemigo. En caso de que *canRespawn* sea *true* se desactiva la colisión, el sprite y el movimiento del enemigo y se inicia la corrutina *Respawn()*, la cual espera un tiempo para volver a activar al enemigo, pero con la salud cargada al máximo.

En la **Figura** 4.26 podemos observar la clase *EnemyHealth* con todas sus variables y métodos.

4.5.2 Movimiento del enemigo "Perro cavernario"

Una vez desarrollado el sistema de salud de los enemigos, nació la idea de crear un enemigo más complejo, capaz de interactuar según su entorno y que pudiera detectar al jugador para de alguna manera planificar su comportamiento.

Se nos ocurrió crear un enemigo patrullero que tuviera la habilidad de perseguir al jugador y atacarlo mientras este estuviera dentro de su campo de visión.

Todo este comportamiento se encuentra dentro del script *EnemyMovement* que actúa como máquina de estados. Dicha máquina de estados tiene la siguiente estructura:

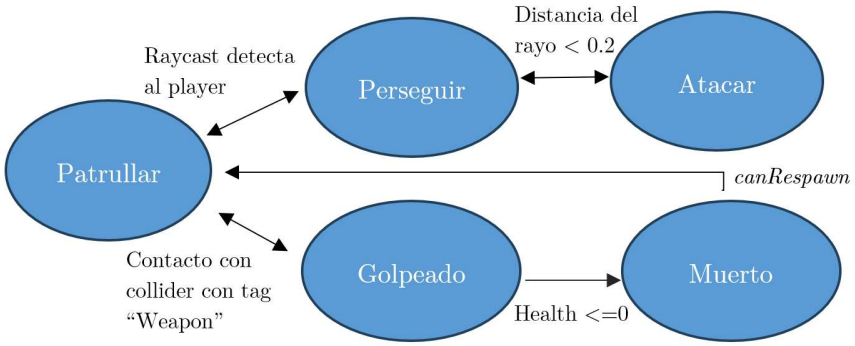


Figura 4.26 *Máquina de estados del enemigo "Perro cavernario"*

Para representar los distintos estados del enemigo hemos creado un tipo enumerado llamado *State* que puede tomar los valores *Patrullando, Persiguiendo, Golpeado, Atacando* o *Muerto.*

El método *Update()* es el que se encarga de manejar los cambios de estado, pues al ejecutarse en cada frame del juego, es ideal para detectar cambios en tiempo real y actuar conforme a ellos según las condiciones para la transición de estado indicadas en la **Figura** 4.26 *Máquina de estados del enemigo "Perro cavernario".*

En función del estado en el que nos encontremos nuestro NPC actuará de una manera u otra:

- *Patrullando*: se hará una llamada al método *Patrol()* que se encarga de planificar cuándo el enemigo deberá de cambiar la dirección de patrullaje, haciendo uso de un cronómetro y llamando al método *Move()* cada vez que el cronómetro llegue a 0 para caminar en dirección contraria.
- *Persiguiendo*: hará una llamada a *FollowPlayer().* Este método es similar a *Move()* con la diferencia de que el enemigo caminará hacia el jugador siempre que este se encuentre en su área de visión, y no dará la vuelta hasta haberlo perdido de vista. Es decir, en esta ocasión no usamos cronómetro.
- *Golpeado*: Solo se puede alcanzar este estado desde la clase *EnemyHealth,* cuando el jugador recibe un golpe. Al ocurrir esto, se ejecuta el método *Hurt()* que inicia la animación de daño y, tras medio segundo, vuelve al estado *Patrullar.*
- *Atacando*: Si la distancia del *RayCast* es menor que 0.2 entonces entraremos en este estado el cual llama al método *Attack().* La forma de implementar el ataque de este enemigo ha sido añadiéndole

un *Collider 2D* en los dientes que, al colisionar con el jugador le hace daño, como si se tratará de un arma.

- *Muerto*: Realmente quien inicia la rutina de *GameOver* es la clase *EnemyHealth* pero en *EnemyMovement* es donde mandamos la orden de reproducir la animación de muerte.

A continuación, se muestra el diagrama UML de las clases que acabamos de exponer:

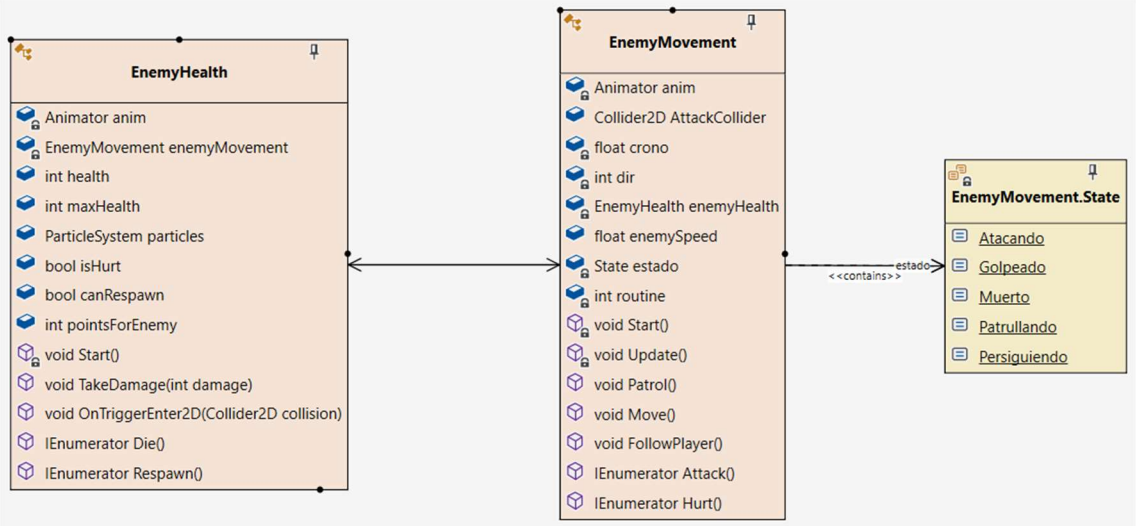


Figura 4.27 *Diagrama UML de las clases EnemyHealth y EnemyMovement*

También, en la siguiente figura se puede observar cómo quedó finalmente el *prefab* de nuestro enemigo "*Perro cavernario*", con sus distintos componentes:

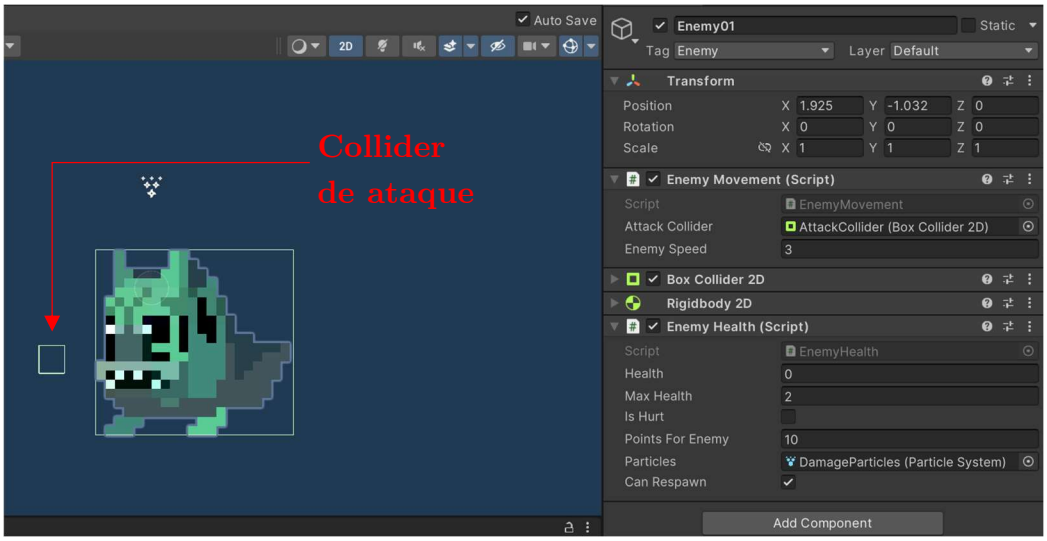


Figura 4.28 *Prefab de Perro cavernario*

4.6 Sistema de guardado

Si queremos desarrollar un juego que mantenga una relación entre las partidas jugadas, ya sea el progreso en la historia principal, o simplemente un único dato que indique cuantas monedas tienes acumuladas para poder gastar en la tienda del juego, debemos implementar una forma de guardar la información para después poder recuperarla.

En este caso hemos querido dar un paso más allá e implementar un sistema de guardado con varios perfiles, de este modo no se restringe una única partida por dispositivo.

Para ello se creó una clase *Profile* que actúa como contenedor de todas las variables de guardado que queremos almacenar, aquellas que se especificaron en la fase de diseño y que ya hemos enumerado anteriormente.

Además, todos los métodos necesarios para la gestión de perfiles se implementan dentro de la clase estática *ProfileManager,* de manera que podemos acceder a ellos fácilmente desde otras clases. Así encontramos *CreateProfile(name)* para crear un archivo JSON de guardado con el nombre que recibe como parámetro, *DeleteProfile(path)* para borrar el archivo que se encuentra en la ruta recibida, *getProfilesList()* para obtener todos los archivos de guardado que tenemos, y *SaveProfile()* junto a *LoadProfile()* para las funciones de guardado y carga de nuestro perfil.

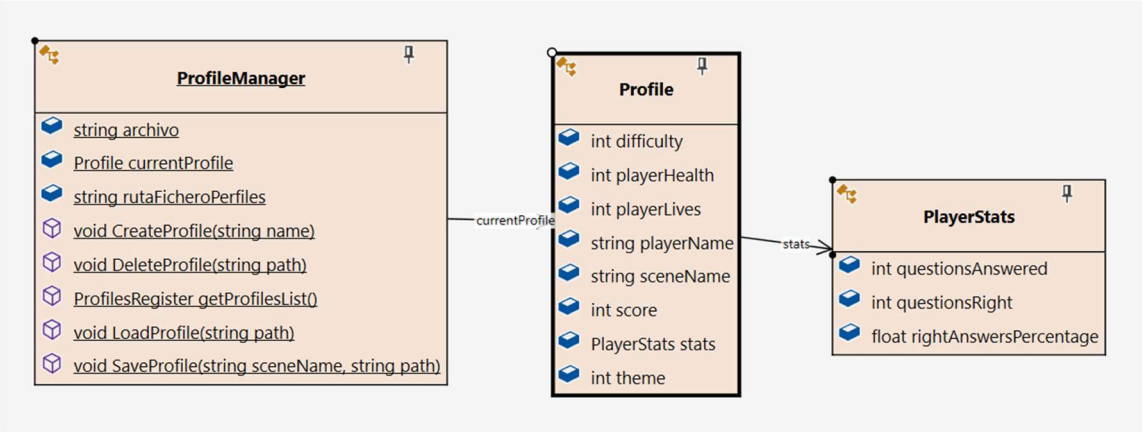


Figura 4.29 *Diagrama UML de las clases para la gestión de perfiles*

Una vez implementados los métodos para gestionar los perfiles, necesitamos poder acceder a ellos desde nuestro juego. Para ello necesitamos un *GameObject* que nos permita visualizar cada perfil y seleccionar cual queremos usar. Para ello, haciendo uso del objeto *Canvas* de Unity que nos permite crear recursos gráficos con distintas utilidades, se creó un *prefab* de lo que sería un botón de perfil para después poder clonarlo todas las veces que fuera necesario según el número de perfiles que hubiera. Se obtuvo una "caja de usuario" como la que vemos en la **Figura** 4.29.

Figura 4.30 *Prefab de ProfileBox*

Una vez creada la estructura que nos permitiría cargar un archivo de guardado, nos faltaba mostrar por pantalla todos los distintos perfiles que había almacenados en el dispositivo. De ahí nace la clase *ProfilesInitializer* cuya función es, en su método *Start(),* recoger todos los perfiles de usuario que tenemos almacenados usando el método *getProfilesList()* de *ProfileManager* para situarlos en la posición de *ProfileBox* que es una ventana con deslizador vertical en la que podremos ver todos nuestros perfiles. Una vez colocados los perfiles, se le asigna la función *LoadProfile()* al botón de carga y *DeleteProfile()* al botón de eliminar.

Por último la función *CreateNewGame()* se utiliza para crear una nueva partida, con el nombre recibido en el *InputField* creado para ello. Para crear una nueva partida se llama al método *CreateProfile()* de *ProfileManager*.

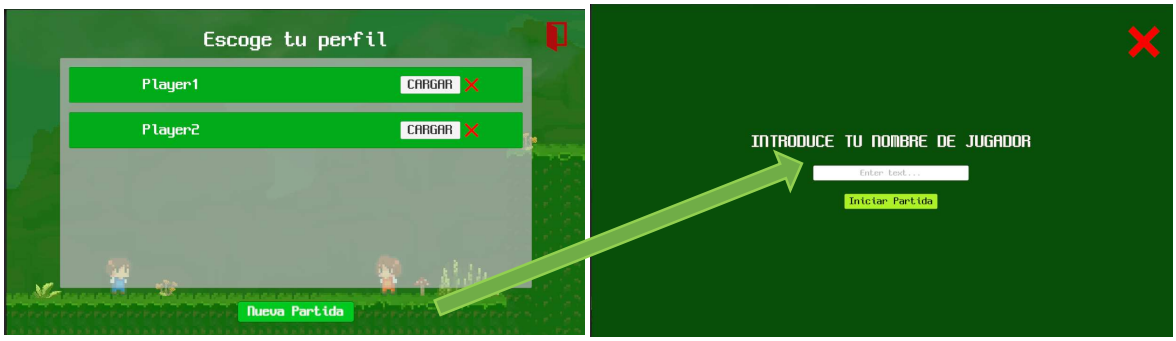


Figura 4.31 *Pantallas finales de carga y creación de partida*

A continuación, se presenta el diagrama UML de la clase encargada de mostrar y crear los perfiles directamente dentro del juego.

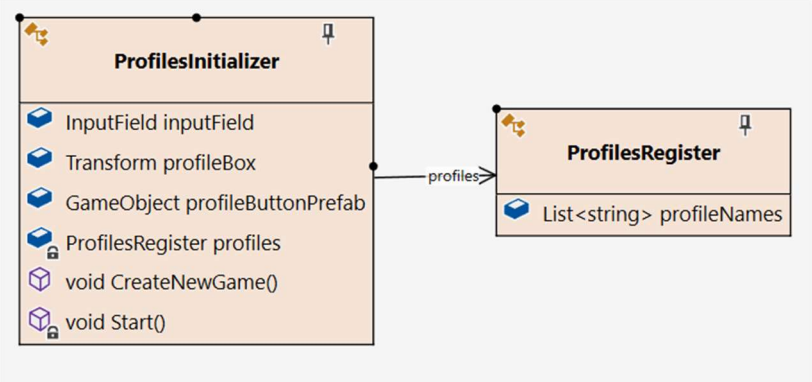


Figura 4.32 *Diagrama UML de la clase ProfilesInitializer*

4.7 Pruebas didácticas. Sistema de quizzes

Este es, posiblemente, uno de los aspectos más importantes de nuestro juego. El que le proporciona esa faceta seria de la que hablábamos al introducir este tipo de juegos. A través del sistema de quizzes se pretende educar al jugador en materia del cambio climático, intentando esclarecer el origen, identificar las consecuencias y aprender a actuar frente a dicho problema.

El desarrollo de esta característica fundamental de nuestro juego abarca no solo programación, si no también diseño de una interfaz donde presentar las cuestiones planteadas, el diseño de la estructura de datos en las que

almacenaremos las preguntas con sus respuestas, la decisión de seleccionar unas preguntas u otras en función de nuestro avance en la partida…

Primero hablaremos de la implementación de la interfaz de quiz. Al igual que en el caso anterior con las pantallas de administración de perfiles, y como con todas las interfaces de usuario de este juego, el sistema de quizzes esta implementado usando el objeto *Canvas. "El Canvas es el área donde todos los elementos UI deben estar. El Canvas es un Game Object con un componente Canvas en él, y todos los elementos UI deben ser hijos de dicho Canvas."* (Unity Technologies, 2016).

Dentro de este *Canvas* encontramos un objeto *Panel* que contiene el fondo del quiz y a su vez incluye un panel secundario con el texto donde irá la pregunta y los 4 botones de respuesta posibles. La creación de un *Canvas* conlleva la creación automática de un *Event System* encargado de controlar los eventos basados en inputs como pueden ser la entrada de texto por teclado, los clics del ratón…

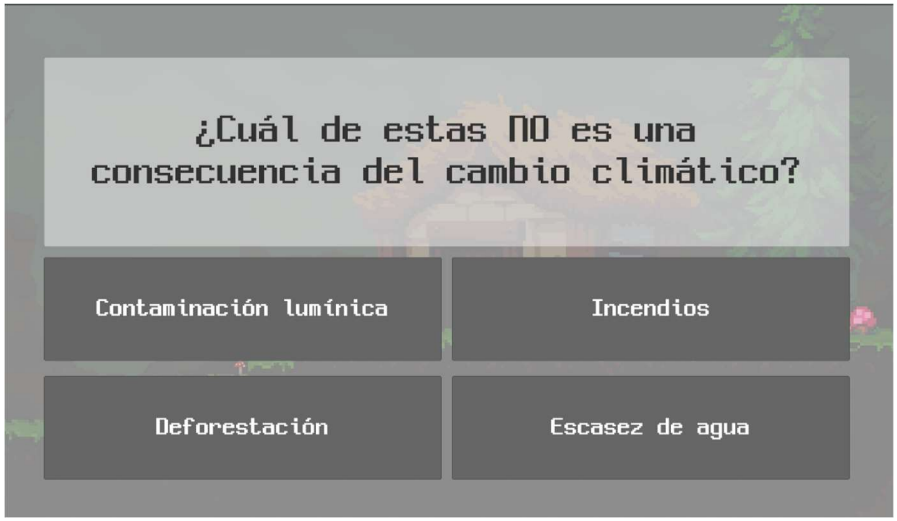


Figura 4.33 *Panel de quiz*

Para la representación de los distintos quizzes se crearon varias clases con el fin de representar los distintos elementos que los componen:

- *Answer:* Creamos una clase para representar las respuestas. Está formada por una cadena de caracteres que representa el texto de la respuesta y un booleano que indica si la respuesta es correcta o no.
- *Question*: Es el elemento fundamental de los quizzes. Incluye un string para el texto de la pregunta y un array de *Answer* para almacenar las respuestas. Pero además incluye un entero con el índice de la respuesta correcta, un índice para marcar el nivel de dificultad que tiene la pregunta y otro índice más para clasificarla por el tema que trata. También tiene un método *SetCorrectAnswerIndex()* que asigna al *correctAnswerIndex* la posición en la que se encuentra la respuesta correcta.
- *Quiz:* Esta es la clase con la que operaremos directamente, consistente en una lista de *Question.*

Una vez implementada la interfaz y teniendo la estructura de nuestros quizzes lista hay que volcar la información en la interfaz y darle un comportamiento.

Esto se hace mediante el *QuizManager*, el cual se encarga de leer un archivo JSON que contiene todas las preguntas que pueden aparecer en el juego (*LoadQuestionsFromJSON()*), seleccionar las preguntas que hay que mostrar en función de los componentes *difficulty* y *theme* de las preguntas, los cuales se comparan con estas mismas variables del *GameManager* (*InitializeQuiz()*), disponerlas en el panel de manera aleatoria y con el orden de las respuestas alterado (*ShowQuestion()*) y, cuando el jugador haya respondido todas las preguntas decidir cual es su recompensa en base a la puntuación obtenida (*ResolveQuiz()*).

Pero aún falta algo crucial, el comportamiento de los botones de respuesta. Pues bien, de esto se encarga la clase *Resolver.* Esta clase contiene un método, *resolve()* que se ejecuta cuando se pulsa uno de los botones de respuesta y se

encarga de colorear el botón en función del resultado, sumar o no puntos a la punuación final, actualizar las estadísticas del jugador y, en caso de fallar, mostrar la respuesta correcta.

Gracias a la clase *Resolver,* podemos saber de antemano si la respuesta que hay en un botón es la correcta o no, de esto se encarga el método *ShowQuestion()* del *QuizManager*, el cual, cuando está asignando una respuesta a un botón, también está asignando *true* o *false* a la variable *isCorrect* del componente *Resolver* de ese botón en función de si la respuesta está marcada como correcta o no.

Una vez finalizado el test, se mostrará por pantalla la puntuación obtenida y se iniciará la corrutina *EndQuiz()* que esperará a que se clique el ratón para cerrar el panel de quiz y continuar la partida.

En la siguiente figura encontramos el diagrama de las clases que conforman el sistema de quizzes.

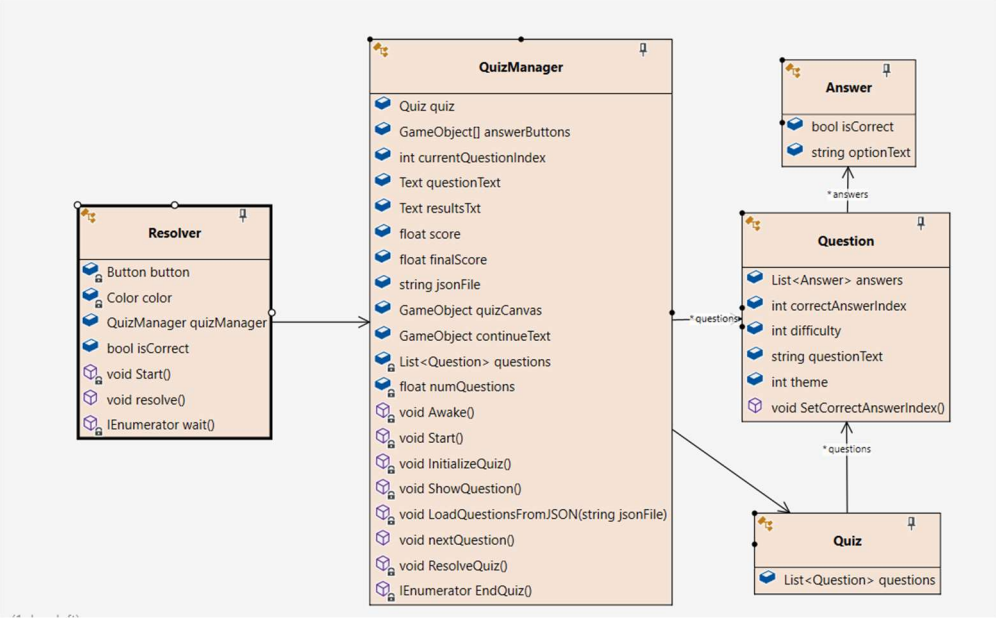


Figura 4.34 *Diagrama UML del sistema de quizzes*

Por último, recordar que el fichero JSON es modificable, anexado a esta memoria encontrará un *Manual de usuario* donde se detalla este aspecto.

4.8 Sistema de diálogos

El sistema de diálogos es también una parte importante del juego. Gracias a él podemos comunicarnos con el jugador para darle información sobre la narrativa del juego, explicar mecánicas e incluso transmitir información útil para lograr nuestro objetivo serio.

Toda la lógica de los diálogos está implementada en un único script llamado *DialogueScript,* pues todos funcionan de la misma manera, pero además se ha incluido la posibilidad de que, tras finalizar el diálogo, el NPC que lo lanza pueda plantearle un quiz al jugador.

Los diálogos pueden iniciarse porque el jugador pulse la tecla E estando en el rango de diálogo o también, si el booleano *showSuddenly* está a *true*, directamente cuando entre en el área de diálogo.

También existen otra opción más para el diálogo llamada *ShowOnce* si queremos que se muestre solo una vez (por ejemplo, para un diálogo eventual al principio de un nivel).

El método *Start(),* al igual que en la mayoría de clases se encarga de inicializar las variables que no dependen del desarrollador del nivel (por ejemplo, si el desarrollador ha asignado un *GameObject* a la variable *quizSystem*, a la variable *hasQuiz* se le asigna el valor *true* en este método).

En *Update(),* se va comprobando si *OnTriggerEnter2D()* u *OnTriggerExit2D()* cambia el valor de la cariable *isOnRange* a *true* para poder iniciar el diálogo. Si *showSuddenly* contiene el valor *true* entonces se mostrará el diálogo automáticamente, sin embargo, si está a *false* deberá cumplirse la condición de que el jugador presione la tecla E. Cuando esto ocurre se llama al método *StartDialogue(),* encargado de pausar el juego, bloquear el menú de pausa y el HUD, activar la pantalla de diálogo y llamar a la corrutina *TypeDialogue()* la cual mostrará cada línea del diálogo almacenada en *dialogueLines* con un efecto

de tecleo. Cuando se muestra una línea entera se espera a que el jugador pulse la tecla derecha del ratón para mostrar la siguiente.

Una vez mostrado todo el diálogo, se llama a la corrutina *FinishDialogue()*. Esta se encarga de esconder la pantalla de diálogo y de proceder en función de:

- *showQuiz* tiene el valor *true*: si este es el caso, entonces se llama al método *ShowQuiz()* que activa el objeto *quizSystem* que contiene la pantalla de quiz y el script del *QuizManager*. Tras esto espera a que la variable *quizDone* del *GameManager* sea *true* para continuar la ejecución de la corrutina. Dicha variable es marcada como *true* al final del método *EndQuiz()* del *QuizManager*.
- *showQuiz* tiene el valor *false*: Entonces se sustituyen las líneas de diálogo por las *extraLines* en caso de que estas existan. Si no existen, el diálogo se repetirá cada vez que se inicie la conversación. Si existen entonces la próxima vez que hablemos con el NPC con diálogo nos dirá el texto que está especificado en *extraLines*.

Una vez se da alguno de los escenarios anteriores se reanuda el juego y si *showOnce* está a *true*, se destruye el objeto que contiene el script de diálogo, si no el objeto se mantiene y el jugador puede iniciar diálogo con el NPC siempre que quiera.

En este juego incluyen diálogo: Los textos de ayuda, los carteles, los NPC del nivel 1 y el NPC de la abuela que además también incluye un quiz.

Figura 4.35 *Prefab NPC*

Figura 4.36 *Prefab Señal*

A continuación, se puede ver cómo está diseñada la clase *DialogueScript* a través de su diagrama de clase.

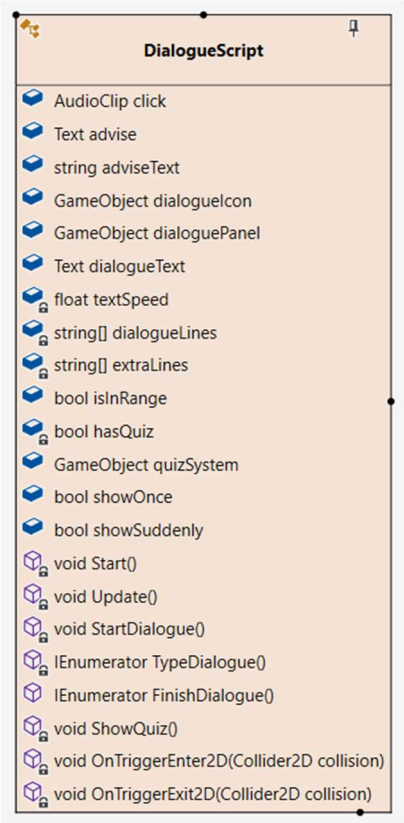


Figura 4.37 *Diagrama UML de la clase DialogueScript*

4.9 Otros aspectos de la implementación

En este último apartado vamos a introducir algunos aspectos que no se han explicado anteriormente pero que sin embargo son interesantes y afectan a la experiencia final del usuario.

4.9.1 Sistema de dificultad dinámica

Uno de los requisitos que surgieron durante el desarrollo del proyecto fue el de que las preguntas que se realizan al usuario pudieran tener distintas dificultades y presentarse en función del desempeño del jugador en los quizzes anteriores.

Ya hemos hablado anteriormente de la clase *PlayerStats* que almacena el número de preguntas respondidas, el de preguntas acertadas y un ratio de preguntas acertadas en comparación con el total de respuestas, así como del parámetro *difficulty* presente tanto en los datos de la partida como en las distintas preguntas. Este parámetro sirve para indicar el nivel de dificultad al que debe

someterse el jugador la próxima vez que responda un cuestionario y para saber que preguntas presentarle en función de ese dato.

A través de las clases *Resolver* y *QuizManager* se actualizan las estadísticas de la partida en *GameManager* para que la próxima vez que haya que recuperar preguntas del fichero JSON, se recuperen en base a la dificultad establecida.

La dificultad se establece en base al porcentaje de preguntas acertadas sobre preguntas respondidas almacenado en *PlayerStats:* si el 70% o más de las preguntas se han respondido correctamente la dificultad será de tipo 3, difícil; si el porcentaje de acertadas es menor al 30% la dificultad será de tipo 1, fácil; sin embargo, si el porcentaje de aciertos se encuentra entre 30% y 70% la dificultad será de tipo 2, media. Los cambios de dificultad los maneja el *GameManager* al actualizar el dato del porcentaje de acertadas.

4.9.2 Comportamiento de las interfaces

Una interfaz sin comportamiento tan solo es una pantalla estática sin utilidad alguna. Se desarrollaron scripts para controlar que ocurriría tras pulsar los botones de las distintas interfaces. Sin embargo dedicarle una sección entera a dichos scripts no es muy interesante puesto que los métodos desarrollados son métodos de una sola línea que llaman a funciones de la librería *UnityEngine*, como el método *Exit()* para cerrar el juego, o a métodos del sistema de guardado que ya hemos presentado anteriormente.

Quizás es interesante mencionar el script del menú de pausa que hace uso del método *Update()* para comprobar cuando el jugador pulsa la tecla Escape para pausar el juego y abrir la interfaz de pausa.

También es destacable la interfaz del HUD que se encuentra en comunicación constante con el *GameManager* para reflejar en pantalla los cambios que se producen en el estado de la partida en tiempo real.

Por último, destacar la pantalla de carga entre escenas, la cual, de manera asíncrona, carga el siguiente nivel mientras se guarda el progreso de la partida y el usuario esta viendo la pantalla con la indicación de carga y guardado.

Figura 4.38 *Pantalla de carga*

4.9.3 Efectos visuales. Sistemas de partículas

Con el fin de darle más vida al juego se implementaron algunos efectos en el movimiento y el ataque del jugador que dan una sensación de mayor dinamismo a la hora de jugar: cuando corremos levantamos polvo del suelo, cuando el arma choca con algo se produce una pequeña explosión de color, si un enemigo recibe daño expulsa unas bolas con tonos rojizos para indicar que ha sufrido daño...

Para ello se usó la herramienta *Particles System* de Unity, la cual presenta muchísimas posibilidades para crear efectos muy llamativos y con distintas formas y colores.

Un ejemplo de lo que permite hacer es: indicar el número de partículas que pueden producirse, el tiempo de vida de estas partículas, si se quiere que tengan colisión o no, si queremos que les afecte la gravedad, el rango de colores en el que queremos que se generen las partículas, el sprite de la partícula, como queremos

que cambie la partícula durante el tiempo que se muestre en pantalla, y un larguísimo etcétera.

En la siguiente imagen podemos ver las configuraciones básicas de las partículas de daño de los enemigos:

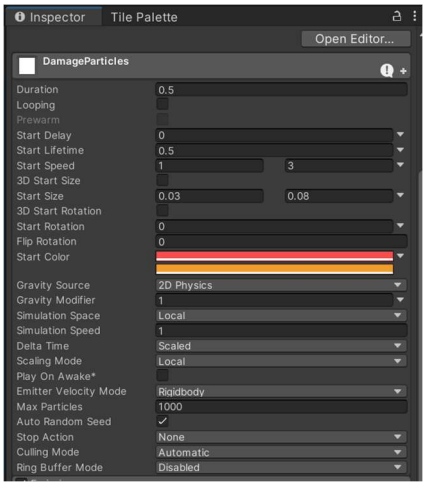


Figura 4.39 *Sistema de partículas de daño a enemigos*

Figura 4.40 *Particulas de movimiento del jugador*

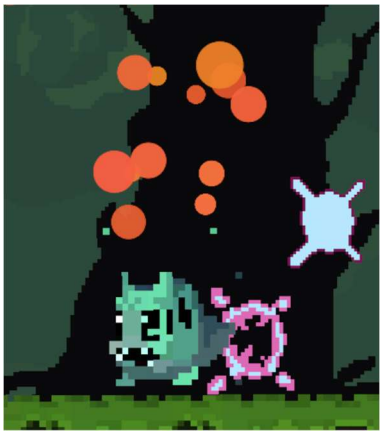


Figura 4.41 *Partículas de daño a enemigos*

5

Resultados finales y experiencia de usuario

Tras finalizar toda la fase de diseño e implementación del juego, este capítulo está dedicado a analizar los resultados obtenidos y estudiar las diferentes opiniones de los usuarios que han podido probar este juego en su fase de pruebas.

5.1 Resultado final: EcoDreams

Llegado el final del desarrollo de este proyecto, podemos decir que hemos conseguido obtener un producto que cumple con los requisitos marcados al principio del capítulo 2 y que se asemeja en prácticamente todos sus aspectos a lo establecido en el GDD a lo largo del proceso de desarrollo.

EcoDreams es un videojuego de plataformas 2D que se mezcla con elementos de puzle y preguntas que hacen que aprender jugando sea algo posible y además entretenido. Se ha logrado obtener un nivel de tutorial y otro parte de la narrativa

del juego que además implementa un puzle basado en la mecánica de palancas como código para abrir una puerta.

Figura 5.1 *Nivel Tutorial de EcoDreams*

Figura 5.2 *Puzzle con palancas del Nivel 1*

También se pudo desarrollar uno de los puntos fuertes del proyecto, el sistema de preguntas pensado para las pruebas didácticas. Este consta de preguntas de 4 opciones clasificadas por tema y dificultad. Como se especificó en los requisitos iniciales, las preguntas las realiza un NPC en una especie de lobby principal donde poder elegir qué nivel se quiere jugar. Pues bien, se desarrolló un

nivel del videojuego donde el jugador llega siempre que supera alguna fase anterior. Este nivel intermedio es la casa de la abuela y es donde tienen lugar las pruebas didácticas, las cuales deben ser respondidas para poder jugar el siguiente nivel.

Figura 5.3 *Casa de la abuela*

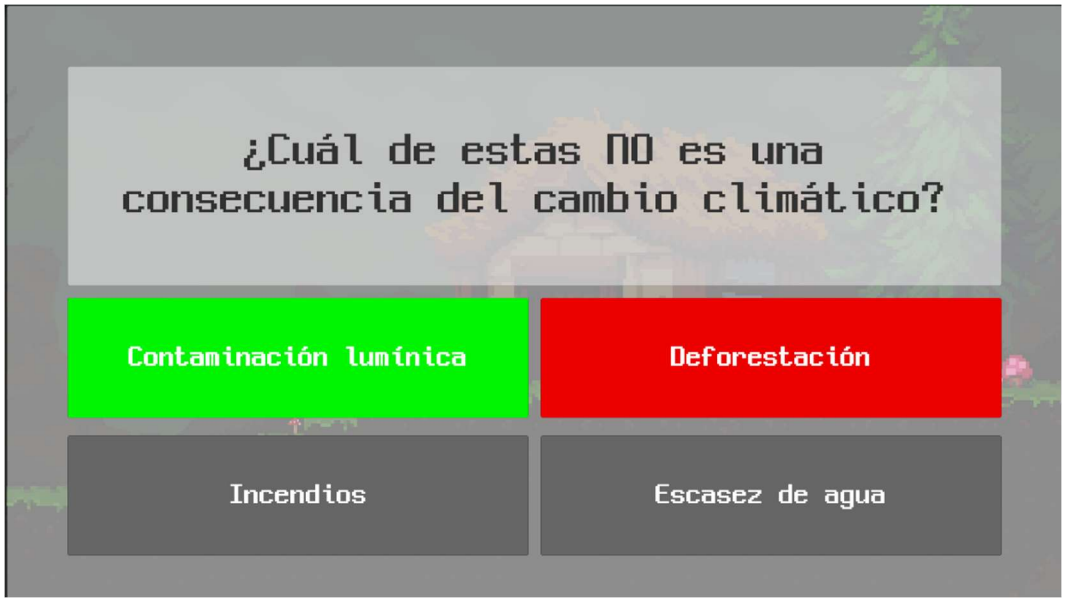


Figura 5.4 *Prueba didáctica*

A lo largo de esta memoria se han ido exponiendo varios puntos que cubrían el resto de los requisitos como por ejemplo el sistema de guardado para

múltiples perfiles, o los controles simples del jugador. Pero para apreciar la importancia de tener un diseño bien marcado y conciso a continuación se exponen las interfaces finales de nuestro juego, completamente funcionales y realizadas en base a los bocetos que veíamos en el capítulo 3:

Siguiendo el boceto representado en la **Figura** 3.1 Boceto del *HUD de EcoDreams*, el HUD quedó de la siguiente manera.

Figura 5.5 *HUD de EcoDreams*

De la misma manera el boceto representado en la **Figura** 3.2 *Boceto del menú principal de EcoDreams*, el menú principal se ve como sigue.

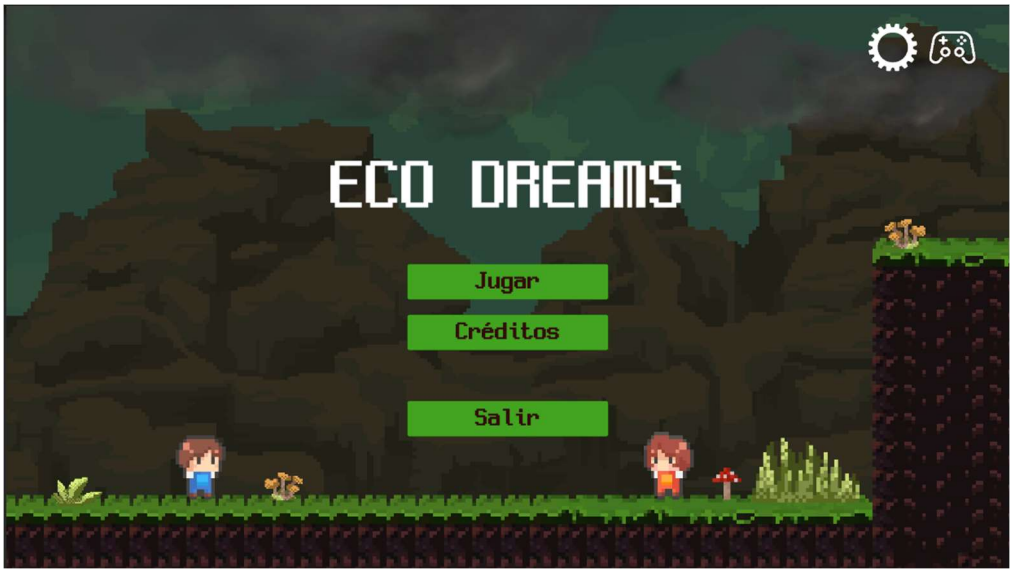


Figura 5.6 *Menú principal de EcoDreams*

Igualmente, el resto de las interfaces que se presentaron en el capítulo 3, son puestas en práctica de los bocetos del GDD, pudiéndose apreciar la clara influencia que han tenido sobre el producto final

5.2 Experiencia de los usuarios.

Cuando ya se obtuvo una versión funcional de nuestro juego, se inició la fase de pruebas con usuarios, proporcionando una copia del juego a un grupo de 7 personas cercanas al equipo de desarrollo las cuales, tras jugar, rellenaron una encuesta donde expresaban sus opiniones y su valoración sobre varios aspectos del proyecto.

En esta sección vamos a analizar las respuestas más relevantes de la encuesta para tratar de analizar el alcance y la calidad de nuestro proyecto.

Uno de los aspectos que más nos interesaba conocer era si el juego cumplía bien sus objetivos, tanto el de entretener como su objetivo serio de concienciar al jugador sobre el cambio climático. Se realizaron preguntas de evaluación del 1 al 10 donde 1 es "Nada" y 10 es "Muchísimo".

Para evaluar el entretenimiento que ofrece nuestro juego se formuló la siguiente pregunta:

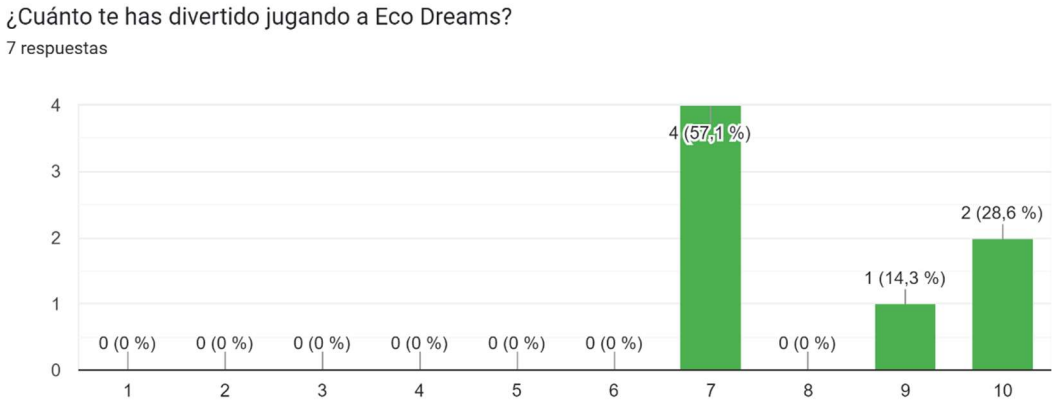


Figura 5.7 *Gráfico de barras: Diversión de los usuarios*

Podemos observar que el juego parece cumplir su objetivo de entretener, aunque vemos que hay algunos usuarios que podrían aumentar este grado de

satisfacción. Esto nos indica que, para futuras versiones, se puede invertir tiempo en desarrollar características que aumenten el nivel de entretenimiento del juego.

De la misma manera quisimos evaluar la capacidad de EcoDreams de concienciar al público, para ello realizamos la siguiente cuestión:

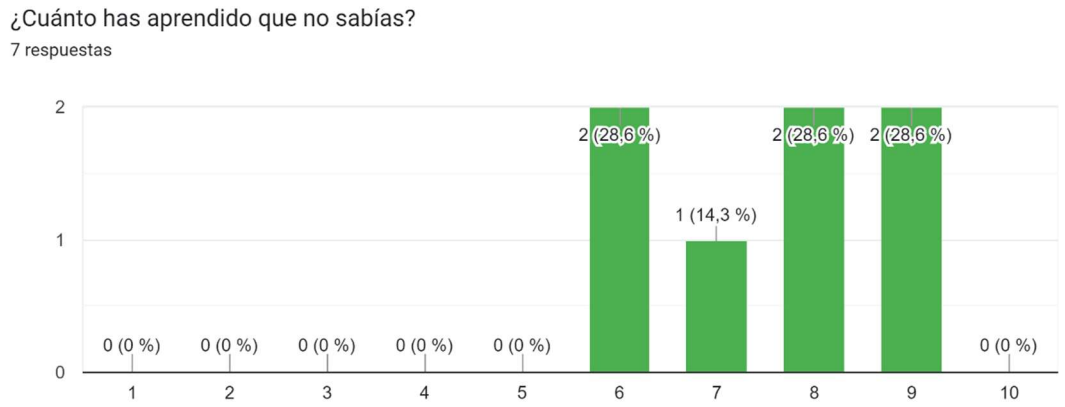


Figura 5.8 *Gráfico de barras: ¿Cuánto has aprendido?*

En este caso también parece que el juego cumple su objetivo de acercar al jugador la problemática del cambio climático. Todos los usuarios coinciden en que, en mayor o menor grado, han aprendido algo en esta primera versión de EcoDreams. Es un dato bastante esperanzador puesto que se pretende seguir desarrollando niveles que traten diferentes aspectos del problema y haga que los datos de esta gráfica se muevan a valores más altos.

También se quiso estudiar la calidad de algunos aspectos del videojuego como pueden ser la calidad de los controles o de las interfaces.

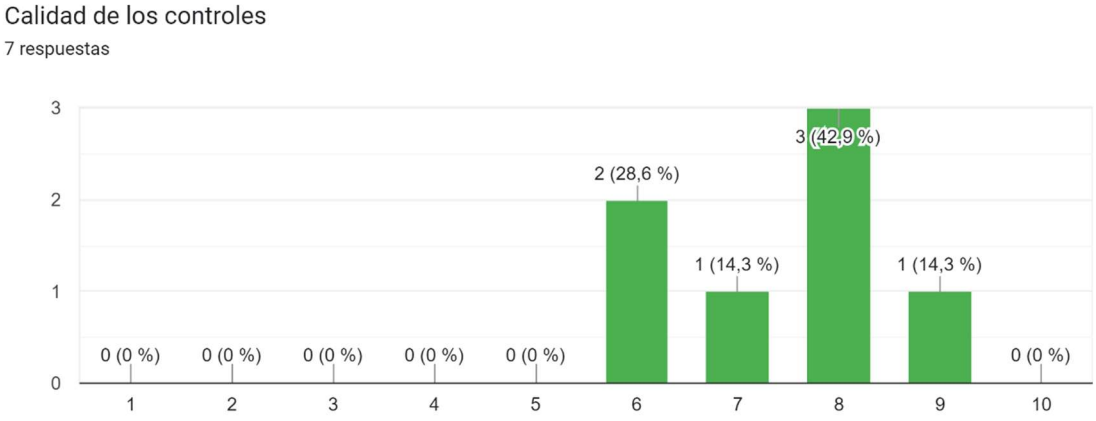


Figura 5.9 *Gráfico de barras: Calidad de controles*

Aunque la mayoría de los usuarios piensan que los controles son aceptablemente buenos, algunos, a través del cajón de opiniones nos hizo llegar que a veces el personaje parecía no reaccionar bien a la tecla de salto, es un dato que debemos tener en cuenta a futuro y solucionarlo ya que es algo que puede afectar negativamente a la experiencia final de usuario.

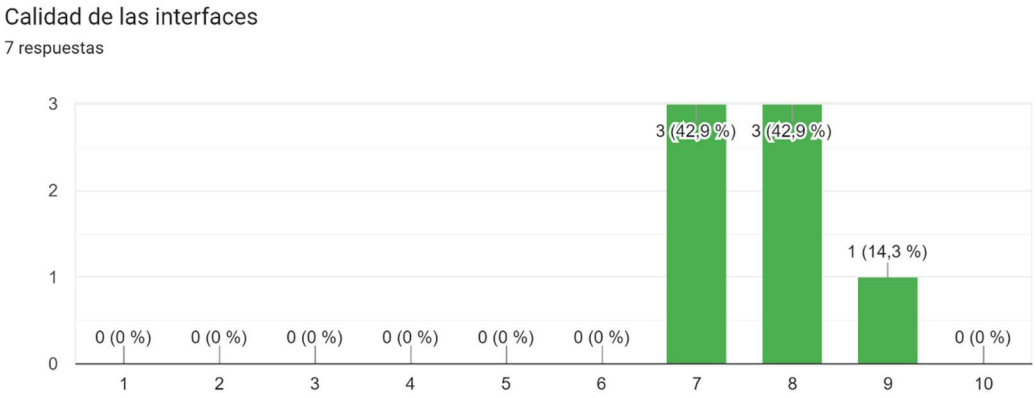


Figura 5.10 *Gráfico de barras: Calidad de interfaces*

Las interfaces también parecen ser del agrado de los usuarios, aunque algunos comentaban que podrían mejorar estéticamente, por lo demás todos parecen coincidir en que cumplen correctamente su función y son accesibles facilmente.

Nos interesaba mucho saber si el usuario había encontrado fallos durante su partida. Para ello se pidió que contestaran si/no a la pregunta: ¿Has encontrado algún fallo o bug?

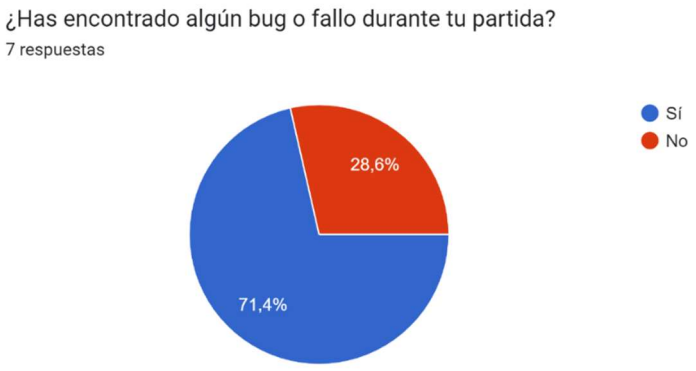


Figura 5.11 *Gráfico circular: Fallos y bugs*

La mayoría de los jugadores encontraron algún fallo durante su partida. Justo debajo de esta pregunta se le pidió al jugador que expresara los fallos que habían encontrado. Los que más se repetían eran: fallo de las colisiones que hacía que en cierto punto del mapa el jugador quedara enterrado en el suelo y sin poder moverse, y también el bloqueo del jugador al caer entre 2 plataformas que no tenían el espacio suficiente para hacer que el avatar del jugador cayera. Tener conocimiento de esto es algo muy útil para el desarrollo de futuras versiones puesto que sabemos qué es lo que tenemos que corregir en primera instancia.

Relacionado con el objetivo serio del juego quisimos conocer que opinaban los usuarios sobre la cantidad de preguntas que aparecen a lo largo del juego, si están equilibradas con el resto de las mecánicas o si por el contrario son demasiadas o muy pocas.

Figura 5.12 Gráfico circular: *Cantidad de preguntas*

La mayoría coinciden en que las preguntas están equilibradas, sin embargo, hay a quien le gustaría que hubiera algunas preguntas más.

Por último, al finalizar la encuesta se le lanzó al usuario la pregunta posiblemente más relevante a la hora de saber si continuar o no con el desarrollo, una pregunta relacionada con su interés en jugar futuras versiones de EcoDreams.

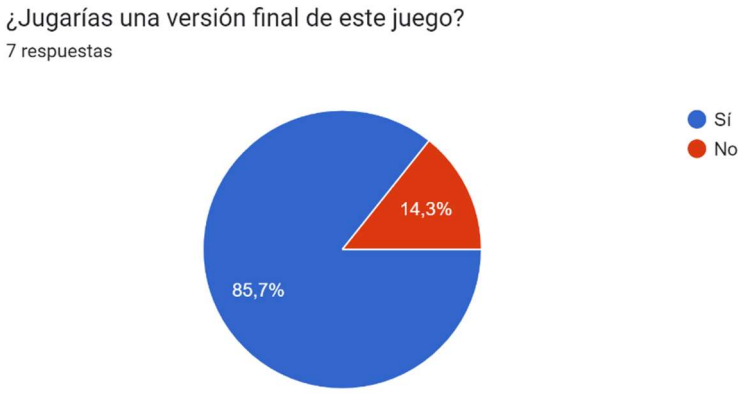


Figura 5. 13 *Gráfico circular: Interés en futuras versiones*

Este dato es uno de los más esperanzadores de toda la encuesta. A los usuarios parece haberles gustado el producto y estarían dispuestos a jugar otra versión actualizada del juego. Esto hace que el producto tenga más posibilidades de continuar su desarrollo y poder salir adelante en algún momento.

6

Conclusiones y líneas futuras

El objeto de este Trabajo de Fin de Grado ha consistido en el desarrollo de un videojuego serio desde cero cuyo objetivo es concienciar al usuario del cambio climático. Se ha desarrollado usando la metodología ágil SCRUM y ha pasado por todas las etapas de desarrollo de un producto software antes de llegar a su puesta en producción: especificación y análisis de requisitos, diseño de la solución, implementación y pruebas.

En este capítulo se reflexionará sobre las mejoras a realizar en el futuro, el aprendizaje personal que ha supuesto el desarrollo de este trabajo, así como de las dificultades y problemas que hemos encontrado a la hora de trabajar.

6.1 Mejoras y trabajo futuro

La principal mejora que necesitaría nuestro proyecto sería a nivel audiovisual. Aunque los sprites que utilizamos combinan bien entre ellos, a la larga puede ser un trabajo difícil encontrar assets que combinen bien y corremos

el riesgo de que el juego acabe siendo un batiburrillo de elementos sin sentido. Lo mismo ocurre con el audio, es difícil encontrar piezas de audio que combinen bien entre ellas, por tanto, una posible mejora sería aprender de generación de audio y diseño de sprites para poder abastecer de personalidad al juego.

Claramente el juego dista de ser una versión final, una mejora significativa sería la creación de nuevos niveles, enemigos, puzles y elementos interactivos que enganchen al jugador y lo animen a continuar la partida.

También hemos tenido muy en cuenta la opinión de los usuarios que han jugado el juego y, aunque ya se han añadido algunas mejoras propuestas como por ejemplo que al marcar una respuesta incorrecta se mostrara cual era la correcta, aún nos quedan algunos fallos que solucionar y una buena lista de mejoras y adiciones propuestas para tener en cuenta.

6.2 Aprendizaje personal

Este Trabajo de Fin de Grado ha supuesto una oportunidad enorme para iniciarse en un área que me apasiona y que no he podido desarrollar tanto dentro de la carrera más allá de unas cuántas asignaturas.

Aun así, se han puesto en práctica muchísimos conocimientos adquiridos en mi paso por el Grado de Ingeniería Informática, sobre todo aquellos relacionados con la ingeniería del software y la programación en general.

Finalmente se ha obtenido un producto del que me puedo sentir orgulloso ya que, al iniciar este camino, no se tenía prácticamente ningún conocimiento sobre desarrollo de videojuegos, y ahora, mirando hacia atrás he aprendido una cantidad inmensa de conceptos, prácticas y habilidades que seguramente de otra forma no hubiera conseguido.

6.3 Problemas técnicos y dificultades

A continuación, vamos a exponer los principales problemas que hemos encontrado durante el desarrollo y cómo se solucionaron los que se pudo.

-Falta de animaciones: Este ha sido uno de los problemas a nivel estético más grande que hemos tenido, pues al faltarnos animaciones por ejemplo de salto o de ataque, ha habido que experimentar con distintos efectos visuales para que no pareciera todo muy estático.

-Colisiones del jugador al chocar con ciertos objetos: Un problema que se ha arrastrado hasta bastante avanzado el proyecto tiene que ver con las colisiones y el sistema de físicas. Ocurría que, cuando el jugador saltaba y chocaba con algo como una pared, este se quedaba pegado a ella. Finalmente, esto pudo solucionarse añadiéndole al jugador un material físico sin fricción que eliminaba ese efecto de pegarse a los objetos.

-Problemas al impulsar hacia atrás al jugador: Este es otro de los problemas que se pudieron solucionar pero que han estado presentes durante gran parte del desarrollo. Lo que sucedía era que, al sufrir daño, el jugador en lugar de salir impulsado hacia atrás, solamente se impulsaba hacia arriba. Esto ocurría porque al estar controlando por teclado, se leía como movimiento horizontal el que se recibía de teclado que normalmente era en dirección contraria al impulso. Se solucionó bloqueando el movimiento del jugador al hacerse daño y volviéndolo a activar cuando tocase el suelo tras el impulso.

-Diseño de interfaces: Costó un poco arrancar a hacer interfaces porque estaba usando un tipo de elementos UI más complicados de usar y que no nos servían. Se solucionó usando los elementos simples y dejando los *TextMeshPro* solo para cuando fuera necesario.

-Fase de pruebas: La fase de pruebas tuvo lugar durante unas fechas un tanto complicadas porque tuvo lugar al mismo tiempo que las vacaciones de gran parte de la gente que se tenía pensada para la fase de pruebas. Costó un tiempo encontrar un grupo dispuesto a probar el juego y dar su opinión, aunque finalmente se ha conseguido hacer un pequeño estudio de experiencia de usuario y, en general, la gente que pudo probar el juego acabo satisfecha con el resultado.

Referencias

Capcom. (1985). *Ghosts 'n Goblins* [Videojuego]. (Accedido el 5 de agosto de 2024). En *Wikipedia* https://es.wikipedia.org/w/index.php?title=Ghosts_%27n_Goblins&oldid=161320835

JSON (s.f). *Introducing JSON.* (Accedido el 1 de agosto de 2024). https://www.json.org/json-en.html

Microsoft. (2000). *C#.* (Accedido el 1 de agosto de 2024). https://dotnet.microsoft.com/es-es/languages/csharp

Naciones Unidas. (1992). *Marco de las Naciones Unidas sobre el Cambio Climático*, Artículo 1º. https://unfccc.int/resource/docs/convkp/convsp.pdf

Nintendo. (1985). *Super Mario Bros* [Videojuego]. (Accedido el 5 de agosto de 2024). En *Wikipedia* https://es.wikipedia.org/w/index.php?title=Super_Mario_Bros.&oldid=161687365

Nintendo. (1986). *The Legend of Zelda* [Videojuego]. (Accedido el 5 de agosto de 2024). En *W*ikipedia https://es.wikipedia.org/w/index.php?title=The_Legend_of_Zelda&oldid=161462504

Schwaber K. [Ken], & Sutherland J. [Jeff]. (2020). *La Guía Scrum. La Guía Definitiva de Scrum: Las Reglas del Juego.* (Accedido el 31 de julio de 2024). https://objetivoscrum.com/wp-content/uploads/2021/01/2020-Scrum-Guide-Spanish-European-2.0_objetivoScrum.pdf

Singleton. (11 de octubre de 2023). (Accedido el 15 de agosto de 2024). En *Wikipedia*. https://es.wikipedia.org/w/index.php?title=Singleton&oldid=154544338#Enlaces_externos

StarkCloud. (2024): *¿Qué es el desarrollo de software?* (Accedido el 30 de julio de 2024). https://www.starkcloud.com/starkcloud-blog/cloud/que-es-el-desarrollo-de-software#:~:text=El%20desarrollo%20de%20software%20es,funcional%20y%20de%20alta%20calidad.

U-tad. (2022). *Serious games: ¿qué son y para qué sirven?* (Accedido el 30 de julio de 2024).https://u-tad.com/que-son-los-serious-games/

Unir. (2024). *Los motores de videojuegos: componentes y principales plataformas.* (Accedido el 1 de agosto de 2024). https://www.unir.net/ingenieria/revista/motores-videojuegos/

Unity Technologies. (2005). *Unity* (v2022.3.18f1). (Accedido el 7 de agosto de 2024). https://unity.com/es

Unity Technologies. (2016). *Canvas.* (Accedido el 28 de agosto de 2024). https://docs.unity3d.com/es/530/Manual/UICanvas.html

Unity Technologies. (2016). *Event System.* (Accedido el 28 de agosto de 2024). https://docs.unity3d.com/es/2018.4/Manual/EventSystem.html

Unity Technologies. (2016). *Particle Systems.* (Accedido el 28 de agosto de 2024). https://docs.unity3d.com/es/530/Manual/ParticleSystems.html

Unity Technologies. (2016). *Unity Learn.* (Accedido el 20 de agosto de 2024). https://learn.unity.com/

Apéndice A

Game Design Document

ECO-DREAMS

VERSIÓN DEL DOCUMENTO 1.5

Escrito por: Samuel Espadas Cabezas

En este documento vamos a desarrollar todo el diseño de un videojuego serio que servirá como Trabajo de Fin de Grado del Grado en Ingeniería Informática. La temática del juego será la lucha contra el cambio climático y su finalidad será concienciar al jugador de este problema tan alarmante.

A1. PERSONAJES

Personaje principal.

Eco es el protagonista de nuestra historia. Partió hace mucho tiempo de su hogar dejando su vida atrás. Tras un sueño revelador aparece en casa de su abuela, la persona que lo vio crecer y con la que más recuerdos felices guarda nuestro jugador, pero todo está muy diferente. En el sueño que tuvo Eco su abuela le pedía ayuda desesperadamente, por lo que al llegar a casa y ver todo devastado y sombrío Eco decide empezar una aventura para recuperar el antiguo esplendor de su hogar.


Eco

Abuela.

Vive en el pueblo del protagonista y es la persona que despierta en nosotros ese espíritu de reacción por recuperar la salud de su mundo, cuando nos ve después de tanto tiempo se emociona y nos pide ayuda desesperadamente. Necesita recabar toda la información posible para concienciar al mundo del daño que le está haciendo a la naturaleza, pero su avanzada edad no le permite explorar todo lo que quisiera.


Abuela

Enemigos.

A lo largo de la aventura encontraremos diferentes enemigos según la dificultad y la temática del nivel, estos enemigos representan algunos de los problemas que ha ocasionado el cambio climático.

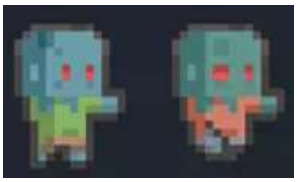
ROOTENS

Estos pequeños tocones mutaron como consecuencia del odio de la naturaleza por la deforestación. Están estáticos en el lugar donde un día fueron hermosos árboles. Son fáciles de eliminar, pero como vayas desarmado mejor no acercarse mucho.

NUBLINOS

Debido a las altas emisiones contaminantes que se emite en la gran ciudad, estos pequeños fantasmas nacen como la fusión de dichas emisiones con las almas de los que mueren atropellados por culpa de los conductores irresponsables.

NUCLEANTES



Estos zombis son el terrible efecto de beber agua contaminada. Son muy agresivos y tienen muy buena vista, deshacerse de varios a la vez puede ser una tarea dura de pelar, aunque al derrotarlos suelen soltar una sustancia revitalizadora, curioso.

PERROS CAVERNARIOS

Esta especie ha sido descubierta recientemente en el subsuelo del bosque. Se cree que es una mutación de algún tipo de canino que se vio obligado a esconderse bajo tierra cuando empezaron a destrozar su hogar.

A2. HISTORIA

Un buen día, después de muchísimos años viviendo en la ciudad sin contacto con su familia, Eco tiene un sueño revelador y decide volver a su aldea a ayudar a una de las personas que más extraña en su nueva vida, su abuela, ya que en su sueño le pedía ayuda desesperadamente.

En el camino a casa de la abuela Eco se encuentra con una escena totalmente diferente a los paisajes verdes y llenos de vida que recordaba. La abuela se alegra muchísimo de tener a Eco con ella y hablando recuerdan los tiempos donde todo era bonito y jugaban juntos en los campos de alrededor de la casa. Tras esa charla, la abuela tan nostálgica le pide ayuda para volver a conseguir que el pueblo rebose vitalidad y algo se mueve dentro de Eco, ¡tiene que hacer algo para recuperar el esplendor de su hogar!

La abuela le empieza a contar todo lo que ha ocurrido en los últimos años y cómo la aldea se ha visto afectada sin si quiera tener la culpa los aldeanos: *hace unos cuantos años abrieron una autopista que conectaba la ciudad con el pueblo, todos los gases contaminantes de los coches y vehículos que atravesaban por allí a diario han hecho que se forme una nube contaminante que apenas deja pasar la luz del sol. A esto se sumó una tala masiva de árboles en el bosque del pueblo, los aldeanos por más que se quejaron a las autoridades no consiguieron impedir la deforestación casi total de aquel espacio. Pero sin duda el suceso más reciente es el que más problemas nos está trayendo, los vertidos contaminantes en el lago nos están desabasteciendo de agua, pues la que cogíamos de allí ya no se puede utilizar.*

Tras oír todo esto Eco inicia su aventura, decidido a investigar que ocurre y tratar de solucionar todos los problemas que recaen sobre el pueblo, ¿será capaz de conseguirlo? ¿No parece como si todo fuera en contra del pueblo? ¿Habrá alguien o algo detrás de todo esto?

A3. TEMÁTICA

Este juego pretende concienciar al jugador del problema que es el cambio climático mientras aprende como combatirlo y crece a la vez que el personaje, en cierto modo, ambos aprenden a la vez y el jugador puede llegar a sentirse conectado de alguna manera a su avatar.

Una vez finalizada la aventura el jugador tendrá una visión más amplia sobre el problema y como ayudar en esta lucha que es de todos.

A4. DESARROLLO DE LA HISTORIA

La acción comienza cuando Eco, después de tener un sueño revelador, decide marchar para combatir todos esos problemas. Sin embargo, la abuela no está muy convencida y no nos dejará marchar hasta estar completamente segura de que sabemos con qué estamos lidiando. En este momento nos hará unas preguntas de dificultad baja que deberemos responder correctamente para avanzar al siguiente nivel.

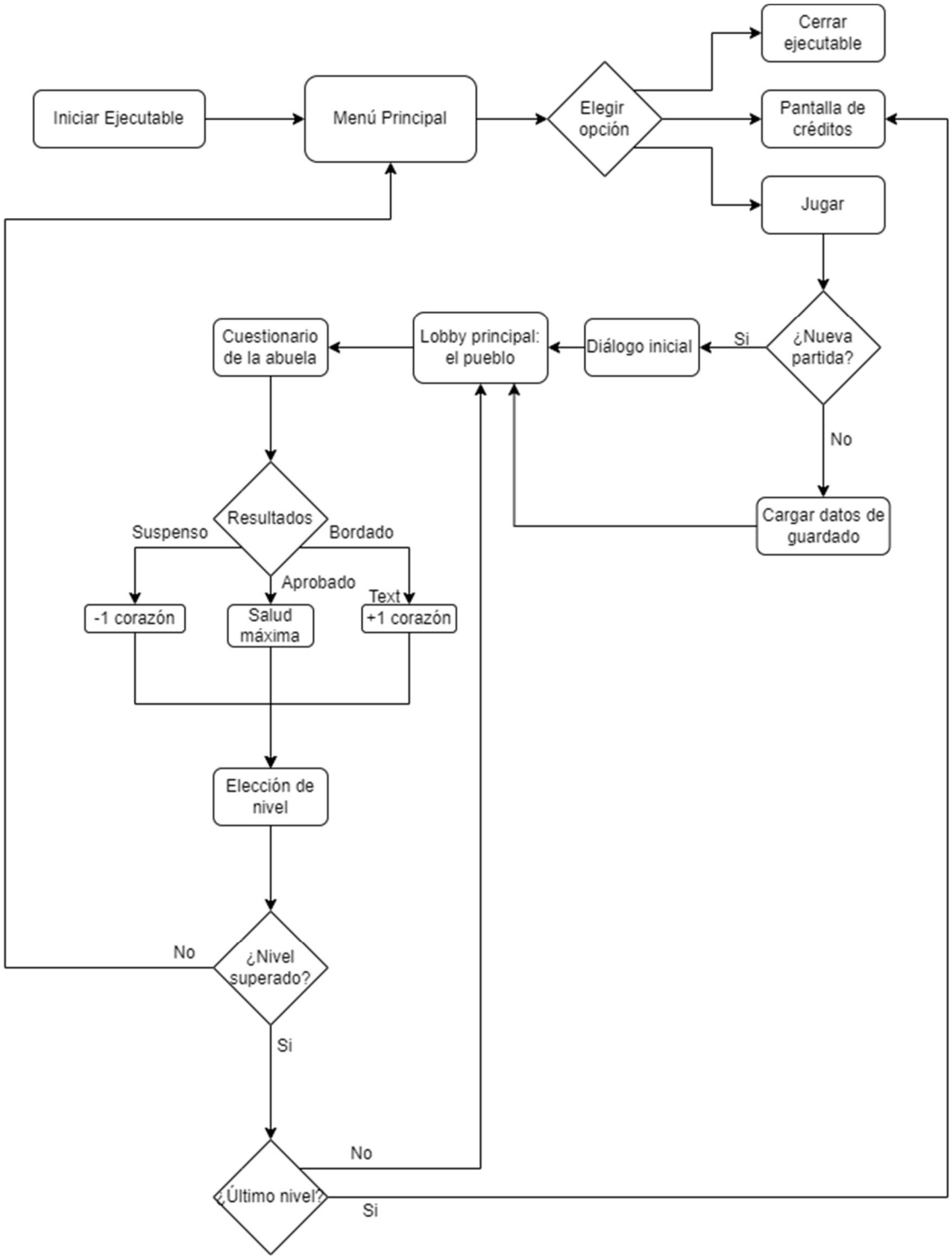
Cuando respondamos la pregunta correctamente podremos desplazarnos hacia un punto de salida de la aldea que nos llevará a un nuevo nivel. Si llegamos a ese punto sin responder la pregunta nos saltará una ventana que no nos dejará avanzar. Los distintos niveles para elegir son:

- El Bosque Maldito (trata la deforestación)
- NYC: Nube York City (trata la emisión de gases por el uso masivo de vehículos)
- Smoke Industries (trata la emisión de gases por parte de la industria y las grandes fábricas)
- Green Lake (trata contaminación del agua)
- ???? (Nivel secreto: desbloquea una lucha contra el boss final si se han superado todas las fases)

Al finalizar cada fase, se volverá a la aldea, donde la abuela nos esperará para volver a ponernos a prueba antes de partir. Una vez aprobada la prueba, el proceso es el mismo que con el primer nivel.

La finalización de cada nivel causa una mejora medioambiental en la aldea por lo que la idea es que cuando se completen todos, el pueblo haya recuperado

-DIAGRAMA DE FLUJO



A5. OBJETIVOS

Si hablamos del objetivo del jugador en esta historia, entonces debemos remarcar que nuestro personaje lo que busca es acabar con las amenazas que se ciernen sobre su hogar y devolverle la vida y el color a su pueblo mientras supera una serie de niveles derrotando enemigos y saltando entre plataformas que podrán presentar una mayor o menor dificultad en función del nivel en el que nos encontremos.

Pero además del objetivo esencial de un videojuego, el de entretener, este juego se marca como reto adicional el de concienciar al jugador de que estamos viviendo una amenaza global real que es el cambio climático. La misión educativa de este proyecto es enseñar a la gente que lo juegue qué causas han provocado el cambio climático, los efectos que ocasiona este problema y las posibles acciones que podemos realizar en nuestra vida cotidiana para tener un futuro más sostenible en este nuestro planeta.

A6. CONTROLES

Dispondremos de los controles básicos de un videojuego de plataformas en 2D:

1. Movimiento horizontal: Desplazamiento hacia la izquierda o la derecha utilizando las ***flechas*** ( y) o las teclas o respectivamente.
2. Salto: Aplicando una fuerza hacia arriba al pulsar la **tecla espaciadora**  .
3. Esc para pausar/reanudar.
4. E para interactuar.
5. F para disparar

Además de los controles básicos, en ciertos niveles podremos recoger objetos (más detallados en la sección *7. Objetos)* que nos permitirán disparar o atacar de distintas maneras (a distancia, cuerpo a cuerpo, en área…).

A7. OBJETOS (Potenciadores y armas)

A lo largo de la aventura podremos encontrar una variedad de objetos que pueden ayudarnos a superar los distintos niveles a los que nos vamos a enfrentar. Estos objetos que pueden decidir el rumbo de la partida son los siguientes:

-POTENCIADORES Y COLECCIONABLES

Sustancia revitalizante

Este pequeño frasco contiene una sustancia que regenera la salud del jugador. Puede encontrarse en cofres, esparcidos por el mundo o al derrotar a un nucleante.

Poción de salto

Esta poción alberga un hechizo a base de pata de conejo que nos permite alcanzar alturas de vértigo. Muy útil para llegar a lugares que a priori parecían inalcanzables, pero date prisa, que no dura para siempre.

Pócima de invencibilidad

Cuenta la leyenda que quien beba el líquido que contiene esta botella gozará de una fuerza inconmensurable y una resistencia inigualable. ¿Será cierto? Habrá que encontrarla para comprobarlo.

Monedas

Estas pequeñas monedas de oro se encuentran esparcidas por todo el mundo. Suman puntos al contador y si se consigue llegar a 100 se arrojará un potenciador al jugador.

Cofres del tesoro.

Escondidos en los rincones más insospechados del mapa podrás encontrar estos cofres con sorpresas que te ayudarán a superar la fase en la que te encuentres.

-ARMAS:

Todos estos potenciadores suenan muy bien, pero… ¿dónde están las armas? Bueno, las armas irán apareciendo exclusivamente en los niveles dónde se las necesite y serán las siguientes:

Mini pistola.

Pequeña pero matona. Permite ejecutar ataques a distancia, aunque es cierto que el tiempo ha hecho que vaya perdiendo fuerza lo que hace que su daño no sea tan mortífero como en los buenos tiempos.

Rastrillo.

No es lo más ideal si no te gusta exponerte al peligro, pero está bien afilado. Al lanzarlo da golpes tan fuertes que podría enviar rápidamente al otro barrio a los enemigos más débiles.

Hoz.

El arma más letal de todas. Golpea de arriba abajo y causa un daño impresionante. Ideal para acabar con los enemigos más duros. Es difícil de encontrar, pero cuando te hagas con ella superar el nivel será coser y cantar.

A8. EVOLUCIÓN DE LA HISTORIA

A continuación, vamos a desarrollar el progreso de la historia a medida que el jugador irá avanzando por los distintos niveles.

Aunque se puedan jugar los niveles en el orden que desee el jugador, el punto de partida es el mismo para todo el mundo. El juego iniciará con el protagonista llegando al pueblo y encontrándose con la abuela. Tras esto se iniciará una serie de diálogos que culminarán con el primer quiz de nuestro juego con preguntas sencillas para demostrarle a la abuela que estamos listos para iniciar la aventura.

Tras haber superado correctamente el primer desafío llega el momento de la acción. El jugador deberá escoger el nivel que más prefiera pudiendo o no tener en cuenta las recomendaciones de dificultad que se indican junto al título del

nivel. Por lo tanto, un criterio para elegir en qué orden jugar los niveles puede ser la dificultad que presenta cada uno.

Cada nivel presentará desafíos únicos como puede ser: enemigos más fuertes, saltos más complicados, una combinación de ambas… La idea es que el jugador se encuentre con diferentes retos en cada nivel que le animen a querer seguir descubriendo el juego.

Al final de cada nivel volveremos al pueblo a contarle a la abuela lo que hemos descubierto, esta nos hará una serie de preguntas en función del nivel que hayamos escogido. Esta parte es muy importante, pues en función de nuestra resolución en la prueba obtendremos castigos o recompensas:

-**Suspender** (solo 1 o ninguna pregunta correcta): La abuela nos regañará y nos quitará un contenedor de vida.

-**Aprobar** (de 2 a 4 preguntas correctas): La abuela nos transmitirá su tranquilidad para que continuemos nuestra aventura y nos llenará el medidor de salud al máximo.

-**Bordarlo** (todas las preguntas correctas): La abuela se quedará atónita y de tal impresión nos premiará con un contenedor de vida.

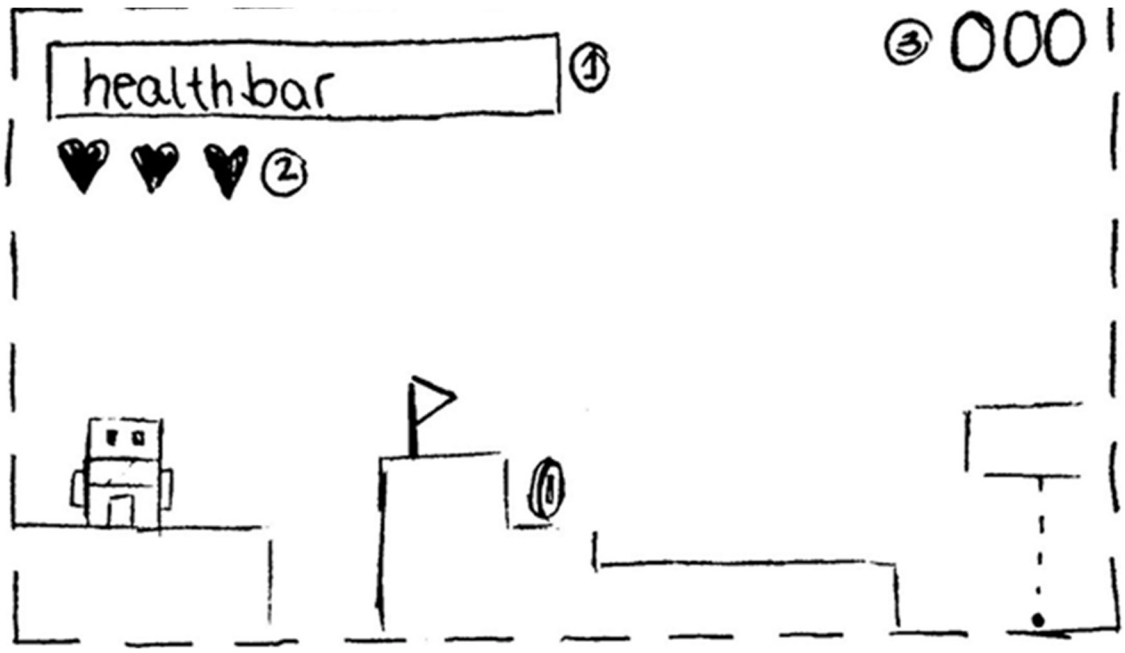
Al final de cada quiz, los efectos de la prueba (tanto positivos como negativos) caerán sobre el jugador y se le permitirá avanzar al próximo nivel.

A9. INTERFACES

En esta sección vamos a presentar los bocetos iniciales que se han realizado para imaginar los distintos menús y pantallas que van a aparecer en nuestro juego.

En esta primera versión del GDD se muestran bocetos hechos a mano sobre papel, lo que hace que en la versión final del juego estos puedan variar ligeramente tanto en apariencia como en funcionalidad llegando incluso a aparecer nuevos menús si fueran necesarios.

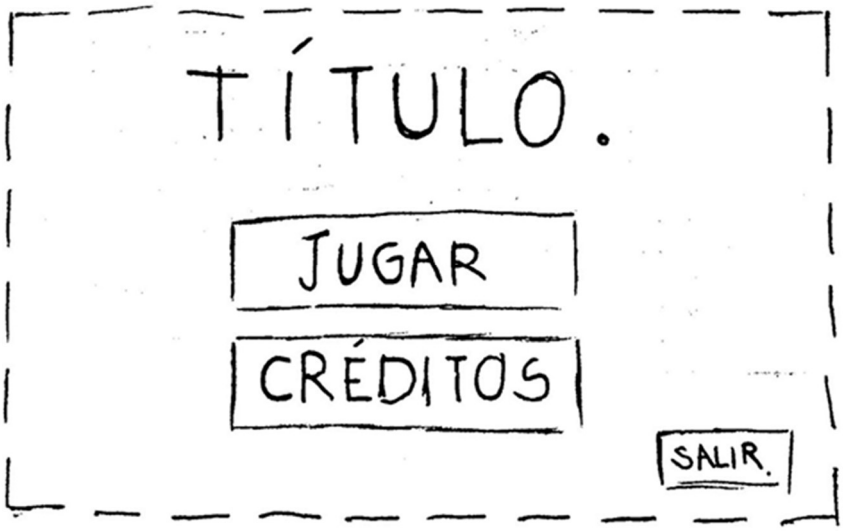
-INTERFAZ DE JUEGO:



Como podemos observar hay varios elementos en la pantalla del jugador que proporcionan información muy útil a la hora de jugar:

1. **Barra de salud**: es un medidor que indica cuanta salud le queda a nuestro personaje antes de que se agote un corazón. Si la barra de salud se vacía el jugador perderá una vida y aparecerá desde el último punto de control.
2. **Corazones**: Indican cuantas vidas le quedan a nuestro jugador antes de que acabe la partida.
3. **Puntuación**: Marcador que registra los puntos que llevamos acumulados por monedas y por derrotar enemigos.

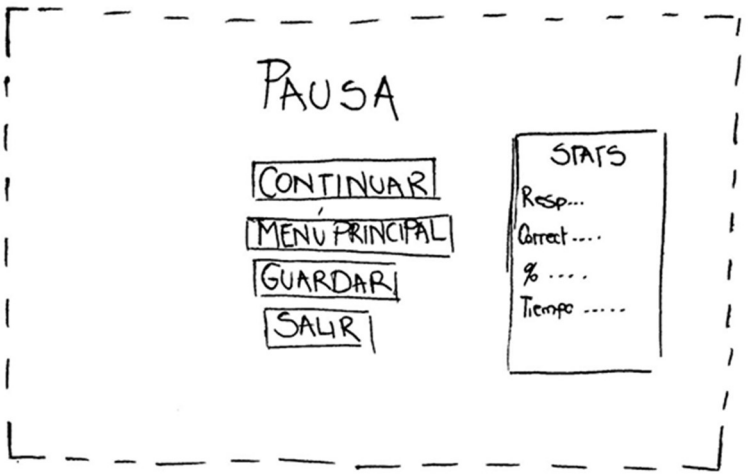
-PANTALLA DE INICIO:



Una pantalla de inicio sencilla donde encontramos el título del juego y algunos botones con distintas funcionalidades: ***Jugar*** para abrir el menú de selección de perfil, ***Créditos*** para ir a la pantalla de información del proyecto y ***Salir*** para cerrar el ejecutable.

Además, en la esquina superior derecha encontramos 2 iconos: el **engranaje** para abrir el menú de ajustes y el **mando** para abrir la pantalla de información de los controles

-PANTALLA DE PAUSA:



Desde este menú tendremos la posibilidad de pausar el juego para después volver a continuar o bien volver al título para operar desde ese menú. También

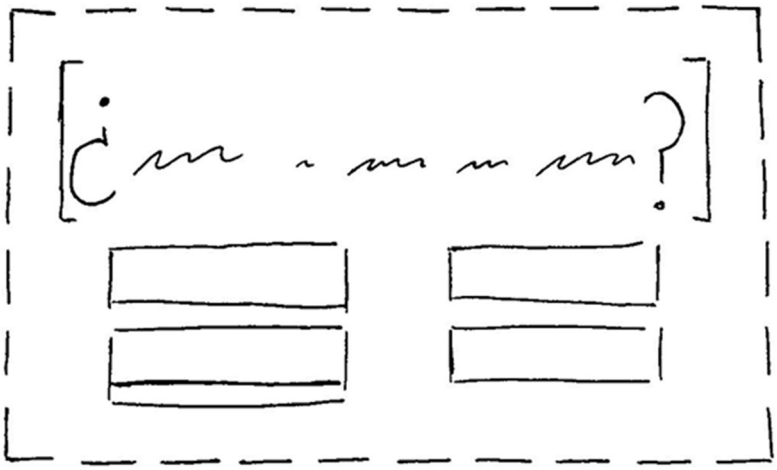
tenemos la posibilidad de cerrar el juego directamente desde esta pantalla o bien de guardar e ir al menú principal.

Además, desde aquí el jugador podrá consultar las estadísticas de su partida en lo que a cuestionarios se refiere y llevar un seguimiento de la dificultad y el progreso de sus datos.

-PANTALLA DE FIN DE PARTIDA:

Cuando el jugador gaste todos sus corazones verá esta pantalla desde la que puede 1) reanudar el nivel desde el último punto de guardado (checkPoints no cuentan), 2) volver a la pantalla de inicio o 3) salir del juego cerrando el ejecutable.

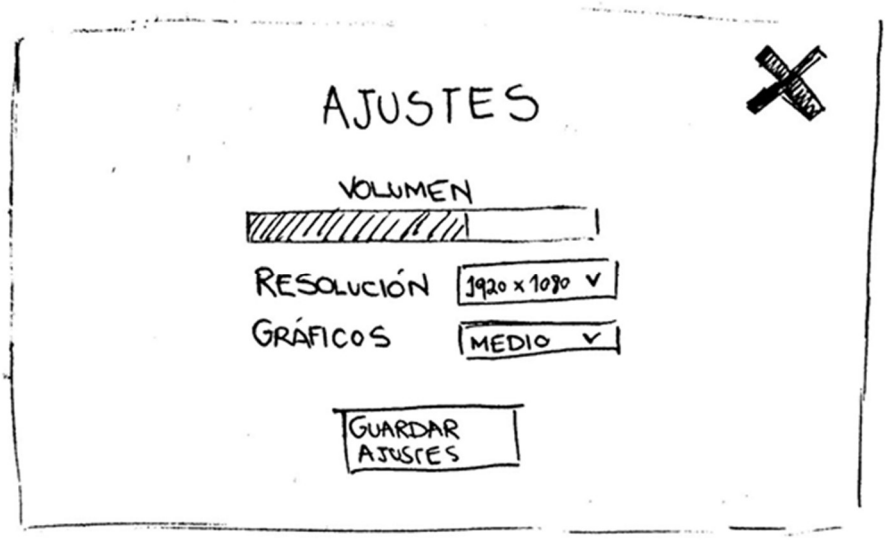
-PANTALLA DE QUIZ:



La pantalla de quiz mostrará 1) Un cuadro de texto con la pregunta que tendremos que responder y 2) Una serie de botones con las posibles respuestas a la pregunta formulada. Al pulsar un botón este se marcará en verde si la respuesta es correcta, o rojo si no lo es.

Además, cuando hayamos respondido todas las preguntas, el cuadro de preguntas marcará el resultado obtenido en la prueba y los 4 botones se deshabilitarán para mostrar un mensaje al jugador:

-PANTALLA DE AJUSTES:



A través de esta pantalla podremos modificar los valores de volumen, resolución de la pantalla y calidad gráfica del juego. Si queremos mantener los cambios realizados primero hay que pulsar el botón de guardar cambios, si en su lugar pulsamos salir cerraremos el menú manteniendo la última configuración guardada.

Inicialmente los valores se establecerán a la mitad del volumen y resolución por defecto de la pantalla en la que se esté jugando.

A10. LOGROS

En este juego habrá una serie de sucesos que el jugador irá descubriendo que desencadenarán algún efecto sobre la partida. Estos efectos en concreto son los siguientes:

-**Resultados de los quiz**: Como ya se explicó anteriormente el resultado en las pruebas teóricas supondrá una ventaja o un castigo para el jugador según el desarrollo en estas.

-Finalización de los niveles: El final de cada nivel supondrá el haber solucionado el problema que se planteaba. Esto conllevará un cambio en la apariencia del lobby inicial, que cada vez será menos oscuro y estará más lleno de vida.

-**Recolección de puntos:** A lo largo de la partida podremos ir sumando puntos a un marcador mediante la recogida de monedas y eliminando enemigos. Cuando lleguemos a 100 puntos el jugador recibirá una vida extra.

A11. QUIZZES

Uno de los aspectos principales de nuestro juego será los quizzes. A través de ellos el jugador demostrará sus conocimientos sobre el tema tratado en el último nivel que jugó. Sin este elemento jamás podríamos cumplir el objetivo serio de nuestro juego.

Una de las características principales de este sistema de quiz es la posibilidad de que el usuario pueda acceder al archivo de preguntas para eliminar/añadir/editar las preguntas que aparecen en el juego en base a su dificultad y el tema que tratan.

La dificultad de las preguntas se irá calculando a medida que avance la partida y se definirá en base al porcentaje de preguntas acertadas sobre el total de preguntas que se han respondido.

Para esto debemos ser capaces de almacenar información y recuperarla cuando la necesitamos, por lo que nos surge la necesidad de implementar un sistema de guardado que registre las variables necesarias para guardar el estado de la partida.

A12. SISTEMA DE GUARDADO

La carga y el guardado de datos son un elemento crucial de los videojuegos. Pueden darse el caso de juegos con un único archivo de guardado o sin embargo juegos que permiten el almacenamiento de varios archivos con partidas diferentes.

Nuestro juego opta por la segunda opción permitiendo la existencia de más de una partida en un mismo dispositivo.

También hay que pensar en cómo se guardarán los datos. En nuestro caso los datos se guardan de dos maneras diferentes:

1. Desde el menú de pausa: en el que encontraremos un botón de guardado
2. Al cambiar de nivel la partida se guardará automáticamente.

La carga se realizará desde el menú de selección de perfil o bien desde el menú de GameOver al pulsar el botón de reintentar.

A13. DATOS DE GUARDADO.

Para llevar un registro del estado de la partida de un usuario necesitaremos al menos:

- Nombre del usuario, string. Para el nombre de perfil e identificación del fichero de guardado.
- Puntos de salud, int.
- Número de vidas, int.
- Puntuación, int.
- Nombre de la escena en la que nos quedamos, string.
- Dificultad de la partidad, int de 1 a 3. Indica si estamos en dificultad fácil, media o difícil.
- Tema, int. El tema es un índice que refleja el último nivel que jugamos. Se usa para saber que preguntas hacerle al jugador en función del último nivel que jugó.

- Estadísticas, PlayerStats. Este campo es un objeto de tipo PlayerStats. Esta clase contiene 3 atributos: questionsAnswered, int; questionsRight, int; y rightAnswersPercentage, float.

Estos datos deberían ser suficientes para lograr mantener un registro de partidas de la manera que nosotros queremos. Sin embargo, esto no quiere decir que en un futuro no se puedan añadir más datos de guardado si fueran necesarios, solo habría que añadir la correspondiente variable y hacer los ajustes de código que hicieran falta.

A14. ESTRUCTURA DE LAS PREGUNTAS

Para almacenar las preguntas que se harán al usuario a lo largo del juego usaremos un fichero JSON que, para cada pregunta, almacenará la siguiente información:

- Texto de la pregunta, string.
- Posibles opciones, array de *Answers.*
- Índice del array donde se encuentra la respuesta correcta, int.
- Índicador de dificultad de la pregunta, int.
- Índicador del tema que trata la pregunta, int.

Estos datos sirven, no solo para definir la pregunta, sino también para poder clasificarlas y acceder a ella de manera más sencilla conociendo su dificultad y el tema que tratan.

Este fichero de preguntas será accesible para el usuario final, el cual podrá modificarlo y añadir, editar o eliminar preguntas a su antojo.

A15. DIFICULTAD

El juego está pensado de manera que cada nivel presente un reto diferente y por tanto una dificultad diferente, teniendo en cuenta que es un juego de plataformas y por tanto habrá nivele con saltos y enemigos más complejos que otros.

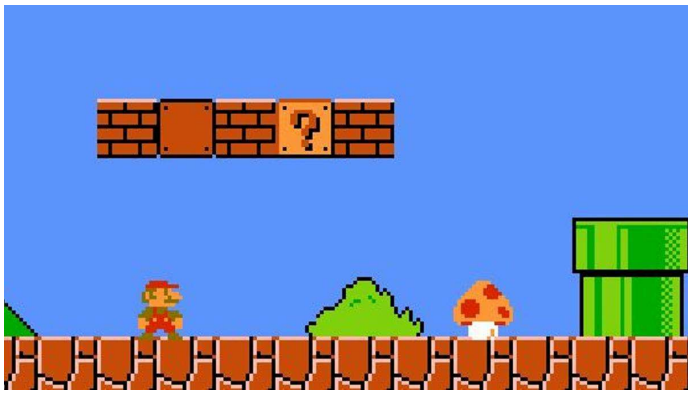
Sin embargo, para las pruebas didácticas se ha pensado un sistema de dificultad dinámica basado en el número de respuestas que ha acertado el jugador a lo largo de la partida. Inicialmente la dificultad será baja y aumentará conforme se acierten más preguntas, pudiendo volver a caer si el jugador empieza a fallar más a menudo.

A16. ESTILO VISUAL

Para este juego se ha elegido el estilo retro como estilo artístico que nos recuerda al de otros videojuegos muy famosos que han servido como inspiración para este proyecto. Alguno de estos juegos puede ser:

-Ghosts 'n Goblins:

-Super Mario Bros:



-The Legend of Zelda:

Como se puede apreciar, en el apartado artístico nuestro proyecto presenta varias similitudes con estos y muchos más juegos super famosos que han revolucionado la industria del videojuego. Además, este estilo artístico tan usado nos permite combinar distintos assets que, aunque no pertenecen al mismo paquete, encajan muy bien juntos.

A17. RECURSOS AUDIOVISUALES

Todos los recursos audiovisuales usados en este proyecto, sprites, tiles, fondos, sonidos y música son gratuitos y libres de derechos de autor.

Las principales fuentes de las que se han obtenido son:

- Sprites, tiles y fondos:
    - https://assetstore.unity.com/
    - https://itch.io/
- Música y sonidos:
    - https://assetstore.unity.com/
    - https://mixkit.co/
    - https://www.jamendo.com/start

Apéndice B

Manual de usuario

A continuación, se detallan algunos aspectos que pueden servir de utilidad al usuario en su primera toma de contacto con EcoDreams.

B1. Instalación

Para instalar EcoDreams, basta con descargar el archivo *EcoDreams.zip,* descomprimirlo en la carpeta donde desee almacenar el juego, entrar a la carpeta *Builds* y abrir el ejecutable *TFG_UnityProject.exe.*

B2. Controles del juego

EcoDreams se juega en PC, y se controla mediante teclado y ratón. Los controles básicos del juego son los siguientes:

1. Movimiento horizontal: Desplazamiento hacia la izquierda o la derecha utilizando las ***flechas*** o las teclas A o D respectivamente.
2. Salto: **Tecla espaciadora**.
3. **Escape** para pausar/reanudar.
4. **E** para interactuar.
5. **F** para disparar

B3. Modificaciones al fichero de preguntas.

Una de las características principales de este juego es la posibilidad de añadir/modificar/eliminar las preguntas que conforman los distintos *quizzes* accediendo al fichero de almacenamiento de preguntas.

Para ello es necesario saber dónde se encuentra dicho fichero. En la carpeta *Builds* donde se encuentra el ejecutable del juego, abra *TFG_UnityProject_Data,* dentro de esa carpeta abra *StreamingAssets* y ahí dentro encontrará el fichero *quiz.json* que contiene todas las preguntas del juego.

Para añadir preguntas al juego es recomendable estar familiarizado con el formato JSON y saber que este fichero contiene un objeto de la clase *Quiz* que está formada por una lista de objetos *Question.*

La estructura de estos *Question* es la que sigue:

- *questionText:* el enunciado de la pregunta que se desea formular.
- *answers:* una lista de objetos *Answer* formada por:
  - *optionText:* el texto de la respuesta que estamos definiendo.
  - *isCorrect:* variable booleana: *true* si es correcta, *false* si no.
- *difficulty:* índice que define la dificultad de la pregunta. Puede tomar los valores 0 para dificultad *Tutorial,* 1 para *Fácil,* 2 para *Medio,* 3 para *Difícil.* Otro valor no mostrará la pregunta en el juego.
- *theme:* índice que define el tema que trata la pregunta. En esta primera versión solamente existen 0 para tema *Básico* y 1 para tema *Deforestación.*

Si eliminando preguntas el juego no puede leer nada relativo al tema y dificultad actuales de la partida el jugador será avisado por pantalla a través del texto *"No hay preguntas para la dificultad y el tema dados"*

B4. Pérdida de datos de guardado

Si se borra la carpeta *saves* por error, volver a crearla vacía dentro de la carpeta *StreamingAssets.* Si no existiera la carpeta *saves* podría haber errores para crear/cargar/guardar una partida.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga