SURVIVOR WARRIOR ZOMBIE

19 DE JUNIO DE 2022

JORGE ARENAS SORIANO

1. Contenido

1. Presentación del proyecto

1.1. Explicación resumida – Game Design Document (GDD).

Un videojuego de supervivencia por rondas en las cuales aparecerán enemigos en función de la ronda en la que se encuentre el jugador y la dificultad escogida por el mismo. Por tanto, el objetivo de este juego es sobrevivir el mayor número de rondas posibles llegando a la ronda número 100 para ganar. El jugador dispone de una barra de salud de 0 a 100 que empezará al completo. Cuando los enemigos golpeen al jugador este perderá vida, en función del daño que cause, que podrá recuperar cogiendo los diferentes botiquines que aparecerán en puntos preestablecidos del mapa. Si la salud del jugador llega a 0, la partida termina automáticamente indicando el número de rondas que has superado. Así mismo, el jugador dispondrá también de una bolsa de granadas. Las granadas se obtienen con una probabilidad del 10% al matar a un enemigo y serán lanzadas por el jugador en la dirección a la que mire el personaje, eliminando a todo enemigo dentro del rango cubierto por la explosión eléctrica.

Además de esto, se pretende añadir una funcionalidad de compra de armas por puntos conseguidos al eliminar enemigos. Se colocarán siluetas de diferentes armas por todo el mapa de forma que, al acercarse el jugador, podrá comprarlas si dispone de los puntos suficientes. Facilitando de esta forma la supervivencia frente a un número de enemigos cada vez mayor.

El juego recibirá el nombre definitivo de Survivor Warrior Zombie (en adelante **swz**).

Pincha aquí para ver un video demo del juego con las funciones principales o usa el siguiente enlace: https://youtu.be/YyNdjc7XYIA

Pincha aquí para acceder al repositorio en GitHub.

Opciones de dificultad.

El juego dispone de 3 modos de dificultad (Fácil, Medio y Difícil) que funcionarán de la siguiente manera:

- **Modo Fácil:** Se generarán enemigos entre el número de ronda y el doble. Por ejemplo: en la ronda 5 se generarán entre 5 y 10 enemigos. El precio de las armas será el preestablecido.
- **Modo Medio:** Se generarán enemigos entre el doble del número de ronda y el cuádruple. Por ejemplo: en la ronda 5 se generarán entre 10 y 20 enemigos. El precio de las armas será un 25% más caro.
- **Modo Difícil:** Se generarán enemigos entre el triple del número de ronda y el séxtuple. Por ejemplo: en la ronda 5 se generarán entre 15 y 30 enemigos. El precio de las armas será un 50% más caro.

En todos los modos de dificultad la salud y el daño de los enemigos serán el mismo, tan solo cambiará el número de enemigos generados y el precio de las armas.

Guardado de datos

SWZ tendrá una opción para guardar partida en cualquier momento accediendo al menú de pausa y seleccionando la opción correspondiente. Esto se hará mediante el uso de objetos serializables y ficheros.

Se almacenarán los puntos acumulados por el jugador, la ronda y escena por la que va, la dificultad escogida y si tiene arma a distancia equipada o no.

De esta forma, al encontrarse en el menú principal con una partida guardada, el jugador podrá escoger la opción de continuar, empezar una nueva partida o incluso cambiar la dificultad de la partida guardada.

Por otro lado, si el jugador obtiene la victoria o muere, los datos guardados se eliminarán y el jugador se verá obligado a comenzar la partida desde cero.

Ranking

Al finalizar una partida, se ofrecerá al jugador la oportunidad de registrar su puntuación para compararse con otros jugadores que se hayan aventurado a disfrutar de las partidas de **swz**.

El jugador podrá escribir su nombre o su alias favorito en un cuadro de texto y, al pulsar en registrar, se almacenará ese nombre junto con la puntuación que ha obtenido en Firebase.

Al acceder a esta ventana de ranking, se mostrarán los 10 mejores clasificados en orden ascendente (del primero al décimo) y, una vez haya registrado sus datos el jugador, el ranking se recargará automáticamente mostrando al jugador si ha alcanzado ese ansiado top 10.

1.2. Estudio de mercado.

Un estudio de mercado consiste en una iniciativa empresarial con el fin de hacerse una idea sobre la viabilidad comercial de una actividad económica. Este estudio tiene un objetivo económico, es decir, su objetivo es generar beneficio económico con la actividad económica cuya validez queremos probar en el mercado.

Esta investigación busca anticipar la respuesta de los clientes potenciales y la competencia ante un producto o servicio concreto para probar los productos o servicios, saber cómo mejorarlos, de qué manera posicionarlos en el mercado, etc.

De esta manera, con un estudio de mercado bien realizado, se conocerá el perfil y el comportamiento de los clientes, la situación del mercado o industria a la que va dedicada el producto, cómo trabaja la competencia, etc. Incluso se pueden llegar a descubrir nuevas necesidades de nuestro público objetivo en las que no se había pensado previamente. Un estudio de mercado se apoya sobre 4 pilares fundamentales:

- **La información del sector:** swz cae directamente en el sector de los videojuegos, concretamente en la parte del sector dedicada a teléfonos inteligentes.
  El sector de los videojuegos es, a día de hoy, uno de los más importantes para la economía mundial debido al amplísimo público que obtiene, desde los más pequeños de la casa hasta, en muchos casos, los no tan pequeños.
  Esta industria ha avanzado a pasos agigantados desde la creación del primer videojuego en el año 1952. Tiempo en el que los videojuegos eran proyectos tan sencillos como puede ser una línea que el jugador puede mover de un lado a otro de la pantalla con el objetivo de bloquear una pelota que va rebotando infinitamente por los bordes de la misma.

Actualmente, en 2022, la industria del videojuego ha crecido tanto, que son capaces de cautivar a cualquier usuario con su calidad de gráficos, sus mecánicas, el increíble realismo que algunos presentan, el sentimiento que desarrollan sus personajes y una capacidad de inmersión cada día más impresionante.
Por desgracia, la mayoría de teléfonos inteligentes no están capacitados para ejecutar, con un rendimiento óptimo, unos juegos de tal calidad. Por tanto, a la hora de desarrollar videojuegos para un teléfono inteligente, existirán una serie de limitaciones que obligará a los desarrolladores a trabajar en proyectos menos ambiciosos. Aun así, con **swz** intentaremos romper un poco esos límites creando un estilo de juego nunca antes visto en estos dispositivos. Para que el proyecto sea rentable, es necesario que funcione en el mayor número de dispositivos posible, por lo que el principal objetivo será optimizar el rendimiento todo lo posible.

- **Conocer al público objetivo (target):** Para crear un público objetivo, normalmente se toman en cuenta su edad, sexo, ubicación, formación educativa, poder adquisitivo, clase social y hábitos de consumo.
  En el caso de **swz**, al tratarse de un videojuego, estará destinado a todos los públicos con la única limitación del código PEGI, un mecanismo de autorregulación diseñado por la industria para dotar a sus productos de información orientativa sobre la edad adecuada para su consumo integrado por dos tipos de iconos descriptores, uno relativo a la edad recomendada y otro al contenido específico susceptible de análisis.
  swz presentará el icono de violencia y miedo, por lo que se empleará el código PEGI-16 para restringir el videojuego a menores de 16 años.

- **Conocer a la competencia actual:** La competencia en la industria del videojuego es muy elevada, puesto que el número de videojuegos que existen a día de hoy roza el infinito. Por suerte, la mayor competencia al estilo de juego de **swz** pertenece al sector de las consolas y el PC, por lo que en teléfonos inteligentes la competencia es mucho menor.
  Un videojuego de consolas muy famoso que podría hacer competencia directa con **swz** es “Call of Duty” en su modo “zombies”, puesto que el estilo de juego es extremadamente similar (supervivencia por rondas con compra de armas y mejora de atributos).
  La idea de **swz** es desarrollar una funcionalidad muy similar a los zombies del call of duty para teléfonos inteligentes, de forma que un jugador aficionado a este estilo de juego, pueda llevar su experiencia y diversión allá donde quiera.

- **Análisis DAFO (Debilidades, Amenazas, Fortalezas y Oportunidades):** Se trata de una sencilla herramienta de análisis estratégico muy extendida en la toma de decisiones de todo tipo de organizaciones y empresas. Sus siglas significan:
  - **Debilidades**: Se trata de las desventajas que tenemos respecto a nuestros competidores, cosas que deberíamos mejorar.
  - **Amenazas**: Entrada de nuevos competidores al mercado o un mercado saturado son amenazas para la empresa.
  - **Fortalezas**: Qué es lo que hacemos mejor que nuestros competidores, en qué nos diferenciamos de ellos.
  - **Oportunidades**: Nichos del mercado no ocupados, estrategias para mejorar la eficiencia y reducir los costes.

Se ha realizado un análisis DAFO sobre **swz** obteniendo los siguientes resultados:

Una gran desventaja es la falta de conocimiento sobre diseño visual tanto de personajes, como de objetos y animaciones. Por lo que se hará uso de Assets públicos creados por diseñadores y publicados en la web de Unity que incluirán diseños y animaciones. Aunque en ocasiones, será necesario diseñar objetos y animaciones que no estén preestablecidas en los assets para hacer uso de ellos.

Debido a que la competencia pertenece al sector de las consolas, las amenazas para **swz** apenas influyen y goza de exclusividad en teléfonos inteligentes por el momento. Lo cual es a la vez su mayor fortaleza y oportunidad. El mayor problema al que se enfrenta **swz** es, sin duda, la variedad de dispositivos móviles que existen y la dificultad que tendrán la mayoría de estos para otorgar un rendimiento óptimo al juego.

1.3. Valor del producto.

El valor de mercado es el valor que un producto (bien o servicio) tiene como consecuencia de la aplicación de la ley de la oferta y la demanda, es decir, lo que normalmente pagaría un comprador por ese producto en condiciones normales de mercado.

Actualmente el mercado de los videojuegos es muy variado entre dispositivos móviles y consolas. El precio de un videojuego de la nueva generación de consolas ronda los 80€, sin embargo, en dispositivos móviles rara vez alcanzan los 10€.

Por otro lado, como se pudo observar en el análisis DAFO, **swz** luchará contra el problema del rendimiento en los diferentes dispositivos móviles, por lo que existirán muchos usuarios que quieran jugarlo pero, sin embargo, no puedan. Por este motivo, **swz** verá reducido su público a usuarios con un poder adquisitivo ligeramente mayor.

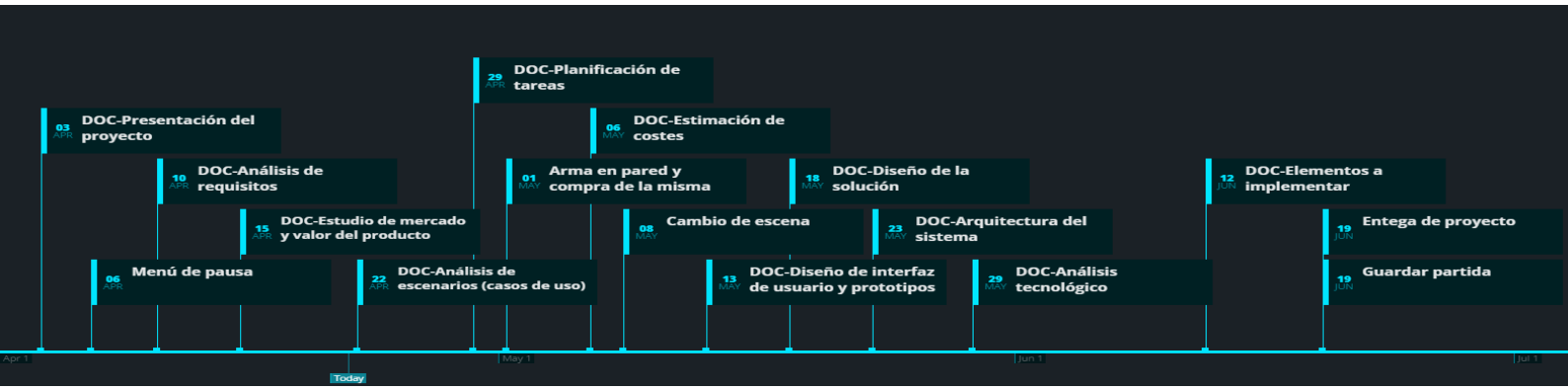
Debido a esto último, el precio que se asignará a **swz** será de 0’99€ con la intención de obtener el mayor número de ventas posible. Es bastante probable que con el precio establecido no se alcance una cantidad suficiente para compensar los gastos, por lo que **swz** tratará de negociar con otras empresas o videojuegos para mostrar su publicidad durante las partidas.

Por otro lado, sabemos que la publicidad puede llegar a ser molesta para algunos usuarios, por lo que trataremos de importar la posibilidad de eliminarlos de forma definitiva mediante un pago extra de 3’99€, quedando de esta manera un precio final del juego completo sin anuncios de 4’98€

2. Planificación de tareas y estimación de costes.

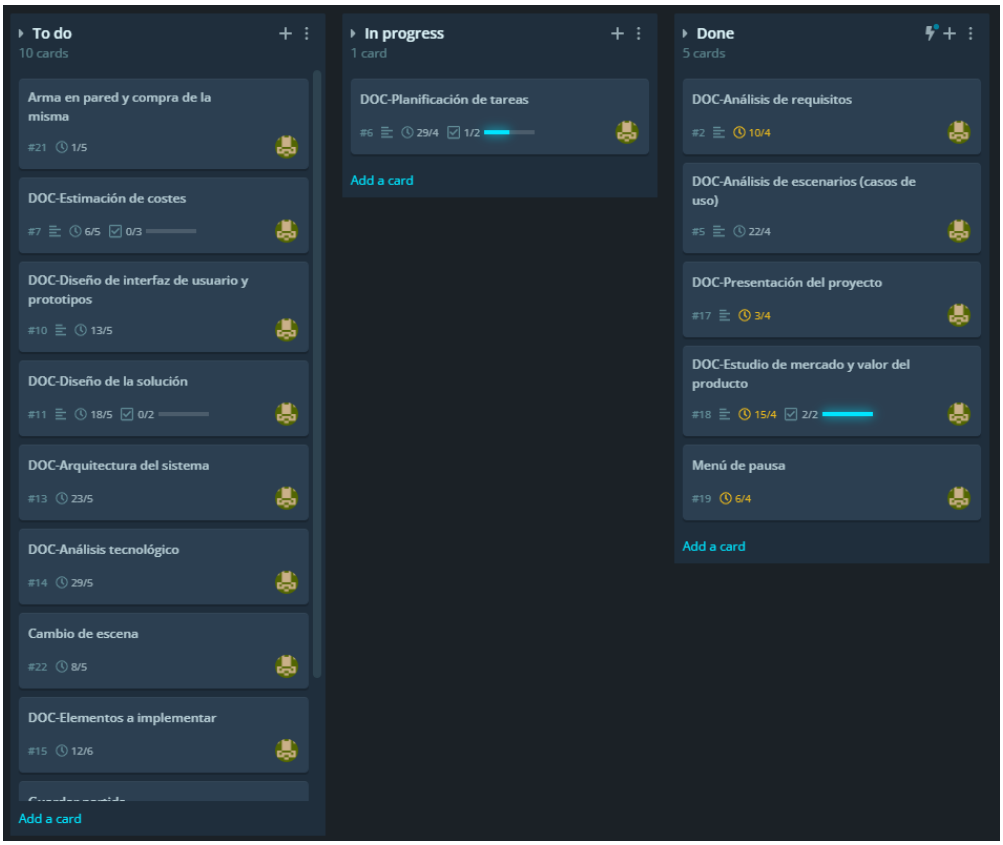
2.1. Planificación y organización de tareas.

Para la planificación y organización de tareas se ha utilizado GitKraken boards, que ofrece la posibilidad de crear un tablero Kanban para poder organizar las tareas del proyecto. Además, estas tareas se pueden descomponer en tareas más pequeñas en modo de CheckList. GitKraken otorga la posibilidad de establecer una fecha en la que terminar cada tarea, pudiendo luego acceder a una vista Timeline en la que se observa de forma muy clara como van a avanzar nuestras tareas en el tiempo. Este es el timeline para el proyecto **swz**:



En la imagen se puede observar como algunas tareas han pasado de su fecha de entrega. Por ahora todas las tareas se han entregado a tiempo y ninguna está retrasada.

Estas tareas se representan además en un tablero dividido en 3 columnas, las tareas pendientes de hacer (To do), las que están en proceso (In progress), y las que ya están realizadas (Done). Presentándose el siguiente tablero:



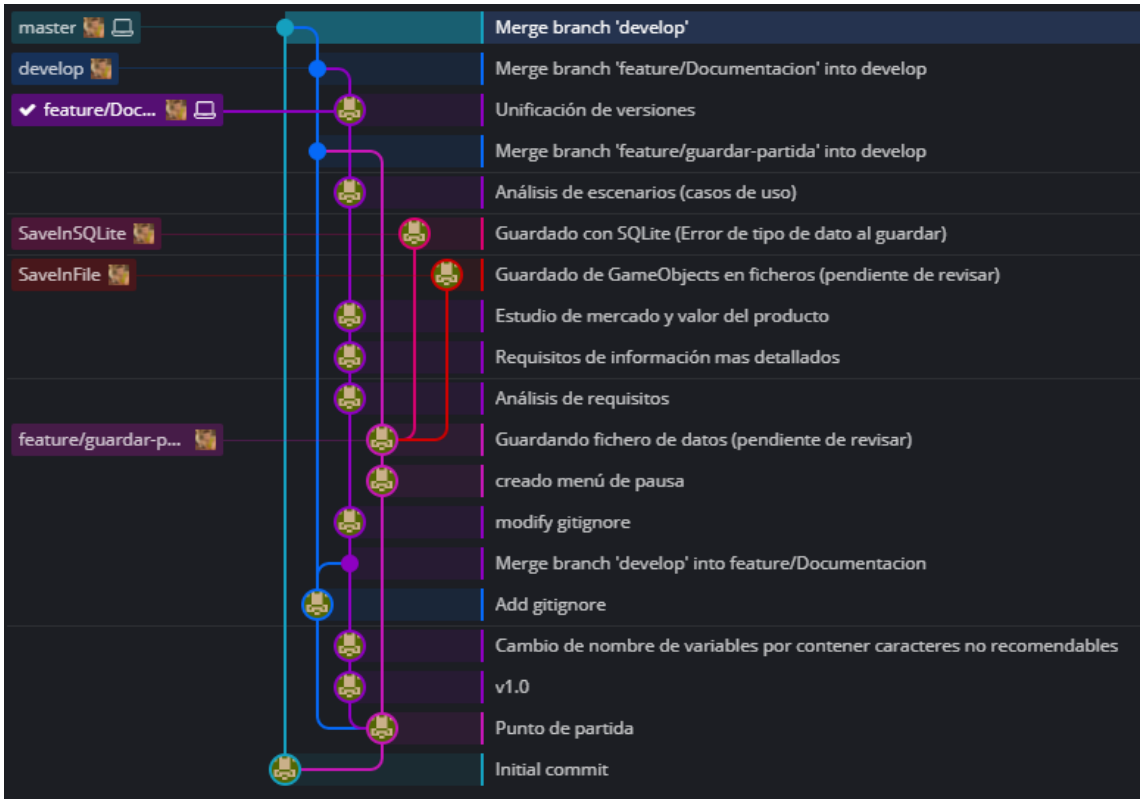
Tambien se puede visualizar el tablero en su versión pública en el repositorio de GitHub del proyecto. https://github.com/Jarenass97/Survivor-Warrior-Zombie/projects/2.

Uso de gitflow como flujo de trabajo.

Gitflow es un modelo alternativo de creación de ramas en Git en el que se utilizan ramas de función y varias ramas principales. En comparación con el desarrollo basado en troncos, Gitflow tiene diversas ramas de más duración y mayores confirmaciones. Según este modelo, los desarrolladores crean una rama de función y retrasan su fusión con la rama principal del tronco hasta que la función está completa. Estas ramas de función de larga duración requieren más colaboración para la fusión y tienen mayor riesgo de desviarse de la rama troncal. También pueden introducir actualizaciones conflictivas.

Gitflow puede utilizarse en proyectos que tienen un ciclo de publicación programado, así como para la práctica recomendada de DevOps de entrega continua. Este flujo de trabajo no añade ningún concepto o comando nuevo, aparte de los que se necesitan para el flujo de trabajo de ramas de función. Lo que hace es asignar funciones muy específicas a las distintas ramas y definir cómo y cuándo deben estas interactuar.

En el repositorio de **swz** se utiliza este flujo de trabajo:



Se pueden observar las ramas master y develop y como principalmente abren o cierran otras ramas, siendo muy limitado el uso de commits.

2.2. Estimación de costes y recursos.

La estimación de costes es un proceso que consiste en desarrollar una aproximación a los recursos monetarios necesarios para completar las actividades del proyecto. Se trata de una predicción basada en la información que tenemos disponible en un momento dado.

En el desarrollo de software es muy complicado determinar el tiempo y esfuerzo que tomará realizar un proyecto. Esto se debe a que la estimación de costes de software no es nada fácil y los seres humanos somos bastante malos prediciendo resultados absolutos. No hay dos proyectos software iguales, por lo que no hay una guía que seguir acerca de cómo realizar estas estimaciones. Hay que tener en cuenta la cantidad de parámetros que conforman la existencia de los proyectos software, frecuentemente un problema que a priori parece sencillo, se vuelve mucho más complejo cuando se aborda desde el punto de vista técnico.

Las variables que se combinan en el dispendio final para la creación de un juego son casi siempre las mismas: desarrollo (incluye herramientas como hardware y software), márketing/publicidad, contratación de personal y distribución. Éstas son, habitualmente, las 4 constantes que suman al desembolso total que han de hacer las compañías para alumbrar un título.

Sin embargo, es muy raro que las mismas nos ofrezcan datos "oficiales" de los costes y, de hacerlo, es poco común que éstos incluyan esas 4 variables. Es por esta razón que es harto complejo el dar una respuesta exacta, contundente e informada a una pregunta tan sencilla como ¿cuánto cuesta hacer un videojuego?

De media, un videojuego puede suponer una partida de 13 a 52 millones de euros para las cuatro variables nombradas anteriormente, dependiendo de la relevancia del producto. Pero esas estimaciones están hechas basadas en equipos de desarrollo profesionales y experimentados. **swz** carece de esa experiencia y, por tanto, no puede apuntar tan alto.

Es por eso que se aprovecha al máximo la disponibilidad de assets gratuitos para desarrollar el videojuego gastando lo menos posible.

swz es desarrollado por una sola persona que estima unos beneficios iguales o superiores a 20€ por cada hora dedicada al desarrollo. Se estima una duración de 90 horas para el desarrollo más 35 horas de documentación e investigación debido a la poca experiencia en el sector, siendo un total de 125 horas. Por tanto, puesto que todos los diseños y software necesarios para el desarrollo serán gratuitos, los costes a cubrir ascienden a 125 horas x 20€ = 2500€

A este valor estimado se añadirán también los costes de despliegue en la Play Store de Google.

Financiación del proyecto

Para financiar **swz**, puesto que el único coste estimado es el sueldo del desarrollador, será financiado a través de las ventas realizadas en la play store de google y de los pagos por publicidad de las empresas interesadas. Este método de financiación puede abocar al fracaso al depender en gran parte de la captación de compradores de manera que es posible no cubrir los costes estimados.

2.3. Herramientas usadas

Las herramientas usadas para el desarrollo de **swz** serán:

- Para el diseño de escenas del videojuego y sus personajes, El motor gráfico de Unity, una herramienta de desarrollo de videojuegos creada por la empresa Unity Technologies.
- Para el desarrollo de código se utilizará visual studio 2022, el mejor IDE completo para desarrolladores de c# en Windows. Completamente equipado con una buena matriz de herramientas y características para elevar y mejorar todas las etapas del desarrollo de software.
- Para las tareas de ofimática, como la documentación, se emplea Microsoft Office 2016, un conjunto de aplicaciones que realizan tareas ofimáticas, es decir, que permiten automatizar y perfeccionar las actividades habituales de una oficina.
- Para la ejecución y prueba del videojuego se utilizará un Xiaomi 11 con Android 12.5 como S.O.
- Para el control de código fuente se hará uso de Git y GitHub. Para gestionar estas herramientas se utilizará Sourcetree y GitKraken free, que además incorpora GitKraken Boards para gestionar las incidencias a través de un tablero kanban donde ver las tareas por hacer, en proceso y finalizadas.

2.4. Gestión de riesgos

Se entiende por prevención el conjunto de actividades o medidas adoptadas en todas las fases de la actividad de la empresa con el fin de evitar o disminuir los riesgos derivados del trabajo. Estos riesgos del trabajo son riesgos laborales, que es la posibilidad de que un trabajador sufra un determinado daño derivado del trabajo. Estos daños pueden ser enfermedades, patologías o lesiones. Para más información acudir a la Ley 31/1995, de 8 de noviembre de 1995, de prevención de riesgos laborales o LPRL.

En este caso, al ser una empresa de informática hay que prestar especial atención a:

- Dolores de espalda y otros trastornos musculo esqueléticos.
- Fatiga visual.

La prevención de riesgos en **swz** se centrará principalmente en:

- Tener un puesto de trabajo adecuado, silla adecuada, escritorio a altura correcta, teclado separado del monitor, buena iluminación y ventilación. La luz debe ser en su mayoría natural y debe entrar por un lado del trabajador, ya que es perjudicial que la luz de por detrás y se refleje en el monitor
- Utilizar herramientas ergonómicas: pantallas con eye-care, teclados con una franja para apoyar adecuadamente la muñeca.
- Recomendación de levantarse y estirarse al menos una vez cada dos horas.

La gestión de riesgos es una actividad de protección dentro de la gestión de proyectos, encargada de identificar, mitigar y monitorizar los riesgos que pudieran afectar a la ejecución y viabilidad del proyecto.

Estos riesgos pueden ser de 3 tipos:

- **Riesgos del proyecto:** Ponen en peligro al plan, si se producen supondrá un mayor esfuerzo y dinero.
- **Riesgos técnicos:** Ponen en peligro la calidad del producto final.
- **Riesgos del negocio:** Ponen en peligro la realización del proyecto, si se cumplen, el proyecto se cancela.

Las principales causas que incrementan el nivel de riesgo en un proyecto son:

- Caer en alguno de los errores típicos.
- Desarrollar sin metodología.
- No tener una correcta estimación, evaluación y administración de los riesgos.

En **swz**, se ha establecido una política clara en cuanto a los riesgos a los que nos enfrentamos:

RIESGO	ACCIÓN
Falta de organización del equipo	Uso de GitKraken boards para organizar el trabajo y las tareas
Mala elección de herramientas	Se intentará estar al día sobre nuevas herramientas más útiles que las usadas actualmente.
Pérdida de código fuente o archivos	Se hará uso del Sistema de Gestión de Versiones Git para tener el proyecto completo en la nube de forma que siempre tengamos acceso a él a través de la web. También se instará a pushear los cambios de forma reiterada para evitar el mayor número de pérdidas.
Mala calidad del producto final	Se pospone el lanzamiento y se revisan todas las posibilidades de mejora posibles para otorgar el mejor resultado posible.
Retraso en una entrega	Se debe trabajar hasta terminarla. Habrá que descontar al cliente el precio ofertado debido al retraso. **swz** cumplirá siempre su palabra.
Interfaz o código mal implementados	Se vuelve a diseñar e implementar aquello que sea necesario
El producto depende de las normativas del gobierno	Estar siempre al tanto de estas normativas para estar siempre bajo el amparo de la ley

3. Análisis de la solución.

3.1. Análisis de requisitos.

Requisitos funcionales

Un requisito funcional define una función del sistema de software o sus componentes. Una función es descrita como un conjunto de entradas, comportamientos y salidas.

Los requisitos funcionales del usuario de **swz** son los siguientes:

1. Manejar al personaje.
2. Posicionar la cámara.
3. Atacar a los enemigos. Habrá dos tipos de ataque (cuerpo a cuerpo y a distancia) que cambiarán en función del arma equipada.
4. Lanzar objetos arrojadizos.
5. Recoger botiquines dispersos por el mapa.
6. Comprar armas empleando los puntos conseguidos al abatir enemigos.
7. Cambiar entre un total de 2 armas que se guardarán en el inventario. Al comprar un arma nueva, esta ocupará el espacio de la que tenga equipada en ese momento, dejándola caer al suelo.
8. Pasar de un escenario a otro sobreviviendo a un número determinado de rondas (Lo decidiré mas adelante) con la salud, puntos y armas conseguidos hasta el momento, pero perdiendo todas las armas que queden tiradas en el suelo.
9. Recoger las armas que se hayan dejado caer al comprar nuevas, siguiendo la misma normativa de intercambio por el arma equipada.
10. Parar el juego en cualquier momento.
11. Guardar el estado de la partida en cualquier momento. De forma que al volver a abrir el juego, el usuario pueda retomar la partida por donde la dejó, sin perder sus puntos ni armas.

Requisitos no funcionales

Un requisito no funcional o atributo de calidad es, en la ingeniería de sistemas y la ingeniería de software, un requisito que especifica criterios que pueden usarse para juzgar la operación de un sistema en lugar de sus comportamientos específicos, ya que estos corresponden a los requisitos funcionales.

Por tanto, se refieren a todos los requisitos que no describen información a guardar, ni funciones a realizar, sino características de funcionamiento. Por esto, suelen denominarse atributos de calidad de un sistema. Queda entonces el requisito no funcional, que son las restricciones o condiciones que impone el cliente al programa que necesita, por ejemplo, el tiempo de entrega del programa, el lenguaje o la cantidad de usuarios.

Entonces, los requisitos no funcionales de nuestra aplicación serán los siguientes:

1. El Front-End se desarrollará en Unity, utilizando assets públicos prediseñados y animaciones descargadas y diseñadas. Se utilizarán Prefabs con aquellos objetos que lo precisen, como pueden ser los enemigos, las armas o incluso el personaje jugador.
2. El Back-End se desarrollará en visual studio utilizando lenguaje C# apropiado para juegos de Unity, utilizando los métodos heredados de la clase MonoBehaviour. Estos scripts irán asignados a los GameObjects o Prefabs a los que vaya destinado su código.

3. Para guardar el estado de la partida, se generará un fichero de datos que será almacenado en una carpeta interna del móvil. (Estudiable utilizar Firebase)
4. Se utilizará base de datos No-SQL con Firebase para guardar un top de los usuarios que hayan conseguido las mejores puntuaciones.
5. Se pedirá un nombre al usuario al terminar la partida para guardarlo en el top de los mejores indicado anteriormente.

Requisitos de información.

Los requisitos de información son aquellos que representan entidades e información relevante con las que el producto software va a operar.

En el caso de **swz**, Estos requisitos son:

Usuario →

- Nombre otorgado por teclado al terminar la partida.
- Puntuación obtenida durante la partida, contando los puntos con los que cuente al terminar más la suma de todos los gastos que haya hecho el jugador al comprar armas. Por ejemplo, si tiene 6000 puntos acumulados al terminar la partida y ha gastado 24000 puntos en armas, el puntaje final será 6000+24000 = 30000 ptos.

Partida →

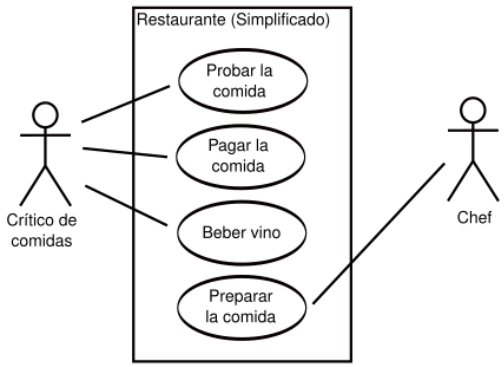
- Se guardará el estado de la escena en el momento de presionar la opción de guardar partida. De forma que, al volver a iniciar el juego, el usuario pueda retomar la partida tal como la dejó.

3.2. Análisis de escenarios (Casos de uso)

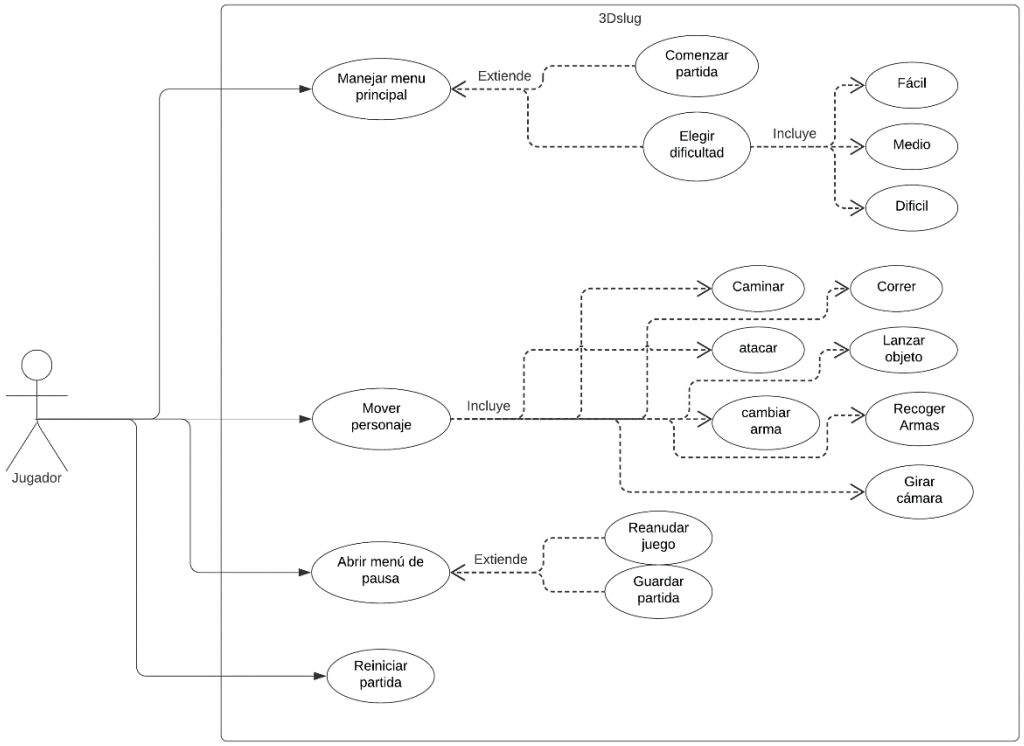
Un caso de uso es la descripción de una acción o actividad. Un diagrama de casos de uso es una descripción de las actividades que deberá realizar alguien o algo con el sistema desarrollado para llevar a cabo algún proceso

En el contexto de desarrollo de videojuegos, un diagrama de casos de uso representa a un sistema o subsistema como un conjunto de interacciones que se desarrollarán entre casos de uso y sus actores en respuesta a un evento que inicia el actor. En este caso, el Jugador.

Estos diagramas de uso son útiles para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios u otros sistemas. A continuación, un ejemplo de un diagrama de casos de uso básico.



A continuación, se expone el diagrama de casos de uso de **swz**, que albergará los requisitos funcionales obtenidos en la fase de análisis de requisitos.



4. Diseño de la solución

A continuación, hablaremos sobre el diseño de la solución. Este proceso juega un papel determinante en el desarrollo, ya que permite a los ingenieros de software producir diversos modelos que:

- Caracterizan la solución a implementar.
- Pueden ser analizados y evaluados con el fin de determinar si se satisfacen los requisitos
- Facilitan el examen y evaluación de alternativas.
- Sirven para planificar las siguientes actividades de desarrollo.

Este proceso es la actividad del ciclo de vida del software en la cual se analizan los requisitos de la solución para producir una descripción de la estructura interna del software que sirva de base para su construcción.

El objetivo de la interfaz de usuario es mantener la interacción con los mismos de la forma más atractiva, centrando el diseño en ellos. Las herramientas principales son los recursos gráficos, los pictogramas, los estereotipos y la simbología.

Es por ello, que toda interfaz de usuario sigue estos 6 principios:

- **Familiaridad con el usuario:** Se utilizan términos y conceptos que se toman de la experiencia de las personas que más utilizan el sistema
- **Consistencia:** La interfaz debe ser consistente en el sentido de que las operaciones comparables se activan de la misma forma
- **Mínima sorpresa:** El comportamiento del sistema no debe provocar sorpresa en los usuarios
- **Guía al usuario:** la interfaz debe dar feedback significativo al usuario
- **Diversidad de usuarios:** La interfaz debe proveer características de interacción apropiada para los diferentes tipos de usuarios

4.1. Diseño de la interfaz de usuario

El diseño tomado para los controles de **swz** se basa en el uso de un joystick digital izquierdo para mover al personaje, un joystick digital derecho para girar la cámara y una serie de botones para las diferentes acciones posibles como: saltar, correr, cambiar de arma, golpear o disparar y lanzar objetos. Además, dispondrá de una parte del HUD informativos sobre el estado de la partida como puede ser el número de enemigos que quedan vivos, la ronda en la que se encuentra el jugador, su barra de vida, el número de granadas que tiene, etc.

4.2. Diseño gráfico

El diseño gráfico tiene un papel fundamental en el proceso, ya que, a través de programas y un previo proceso de bocetaje, se plasman las ideas de manera precisa y clara, por lo que debe de comunicarse con los miembros de los diferentes departamentos para saber lo que han hecho en su trabajo, ligarlo a su invención creativa y reflejar la estética del tema para que todo esté en concordancia.

El diseño gráfico aporta los conocimientos necesarios para:

- que el usuario pueda tener una interactividad funcional con el producto a través de la creación de íconos, menús, etc.

- llevar a cabo la correcta creación o selección de una tipografía adecuada que cumpla con los criterios técnico-visuales y que sea del agrado y comprensión del usuario.

- dar un orden visual en formas que interactúen correctamente y den como resultado algo estético y fácil de digerir, visualmente hablando.
- proporcionar una paleta de colores que comunique armonía, coherencia y que sea atractiva.
- trabajar en el tratamiento de imágenes.
- crear ilustraciones internas y externas.
- dar seguimiento en medios impresos y digitales.

Por ello, es importante que el diseñador tenga los conocimientos básicos sobre la profesión y que tenga un lenguaje visual evolucionado para dar una solución funcional al problema de comunicación visual.

Y es que el diseño gráfico viste aquellos códigos de programación, guía a la animación para que visualmente sea atractiva, comunica visualmente lo que los guiones o historias nos dictan y da funcionalidad para que el mensaje sea recibido.

En **swz** casi todo lo relacionado con diseño gráfico se ha descargado de la web de assets de Unity de forma gratuita en forma de paquetes como puede ser el jugador, los enemigos, las escenas, armas, etc.

Aun así, no todo se ha utilizado tal y como venía descargado. Se han añadido luces, efectos o pequeños detalles según han ido siendo necesarios, como por ejemplo la animación de ataque del jugador, que no estaba prediseñada y ha tenido que diseñarse desde cero.

A la hora de escoger los assest a utilizar, **swz** se ha decantado por un estilo tétrico que genere una inmersión del jugador en un ambiente de survival horror. Se han empleado un estilo de luces anaranjadas y oscuras para lograr con aun más éxito este efecto.

Además de esto, se han combinado diferentes assets como puede ser la granada y el efecto de electricidad que genera para dar más sensación de realismo al explotar.

4.3. Impacto del diseño en los usuarios

Los diseños y modelos escogidos para un videojuego influyen directamente en la percepción del usuario del mismo. No es lo mismo un videojuego de guerra como call of duty que uno de plataformas como el Mario Bross. Esto significa que no todos los juegos están dirigidos necesariamente a todos los públicos, estableciendo límites de edad para los niños en juegos susceptibles para su edad o informando sobre posibles riesgos en personas con algún tipo de problema que pueda verse afectado por el videojuego, como puede ser:

- la epilepsia por exceso de cambios de luz
- problemas de corazón por sustos inesperados o tensión
- dolor de cabeza por exceso de tiempo de juego, etc.

swz es un videojuego principalmente violento y con algunas situaciones de terror debido al ambiente tétrico generado en alguna escena, por lo que puede no ser apto para cierto público con enfermedades de corazón o personas que no superen una edad mínima establecida. Esta edad mínima se ha establecido en los 16 años basándose en las reglas de edad PEGI.

4.4. Diagrama de clases

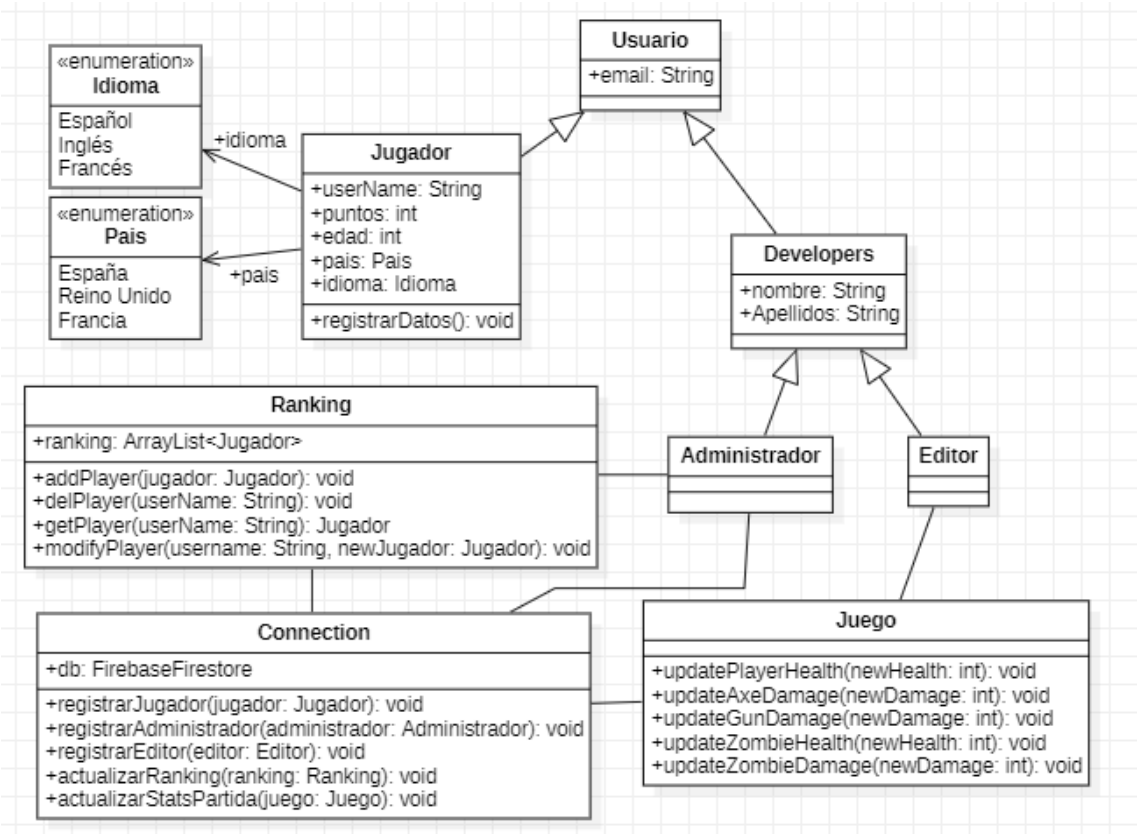
Un diagrama de clases en Lenguaje Unificado de Modelado (UML) es un tipo de estructura estática que describe la estructura de un sistema mostrando las clases de este, sus atributos, operaciones (o métodos) y las relaciones entre los objetos.

En cuanto a las clases, podemos definir las mismas como una plantilla para la creación de objetos de datos según un modelo predefinido. Cada clase es un modelo que define un conjunto de variables (estado) y métodos para operar con dichos datos (comportamiento). Cada objeto creado a partir de la clase se conoce como instancia de la clase.

En cuanto a las relaciones entre los objetos, son el tercer pilar fundamental del diagrama de clases. Pueden ser binarias o de orden superior. Si dos clases están relacionadas significa que esas clases tienen algo que ver entre sí.

Puesto que hablamos de un videojuego, las clases son asignadas a diferentes objects que ejercen las funciones designadas en la clase. Todas estas clases asignadas heredan de MonoBehavior y adquieren 2 métodos esenciales para el funcionamiento del videojuego: **Start y update.** Start se ejecuta solo en el primer fotograma en que se instancie el object mientras que update se ejecuta en cada uno de los fotogramas, pudiendo desarrollar en ese método, por ejemplo, el movimiento del jugador.

Además del juego, se propone de cara a una futura ampliación el diseño de una app también para móvil en la que podrán entrar los desarrolladores del juego bajo los roles de administración de jugadores o edición de las stats del juego como la salud del jugador o el daño producido con cada ataque, quedando como resultado el siguiente diagrama de clases:



4.5. Persistencia de la información

Se denomina persistencia a la capacidad de guardar la información de un programa para poder volver a utilizarla en otro momento. Esto puede significar guardar los datos en un fichero (de texto, binario, csv) o guardar los datos en una base de datos (y sus distintas alternativas)

Para el desarrollo de swz, utilizaremos la base de datos NoSQL de Google Firebase y, además, un guardado en ficheros para almacenar los datos de la partida en el propio almacenamiento del teléfono.

NoSQL significa “not only SQL”, es decir, no sólo SQL. No es un modelo antagónico, si no un enriquecimiento y complemento útil de las tradicionales bases de datos SQL relacionales.

NoSQL plantea modelos de datos específicos de esquemas flexibles que se adaptan a los requisitos de las aplicaciones más modernas. Este tipo de bases de datos surgió debido a las limitaciones y problemas de las bases de datos relacionales, que no son capaces de hacer frente a las exigencias del desarrollo moderno. En cambio, las bases de datos NoSQL utilizan novedades, como los servidores en la nube y estructuras de datos muy potentes y flexibles.

El funcionamiento de estas bases de datos parte de la premisa de no usar tablas tradicionales y rígidas para almacenar los datos. En su lugar, organizan grandes volúmenes de datos con técnicas más flexibles como por ejemplo documentos y pares de clave valor. Una de las particularidades de los sistemas NoSQL es el escalamiento horizontal. Para entender este concepto es necesario saber que las bases de datos tradicionales escalan de manera vertical, es decir, toda su capacidad de rendimiento se basa en un solo servidor, por lo que para aumentar su capacidad hay que invertir en un servidor más potente (una opción muy cara en el largo plazo). El escalamiento horizontal supone que las soluciones NoSQL distribuyen sus datos en varios servidores, por lo que si necesitamos manejar más volumen de datos con un servidor austero sería suficiente. De esta manera, pueden almacenar grandes cantidades de datos a un precio menor.

Decisión del tipo de almacenamiento de datos.

En primer lugar, se ha elegido Firebase por su facilidad para almacenar datos desde teléfonos inteligentes y su flexibilidad a la hora de crear colecciones. Por otro lado, se ha escogido utilizar almacenamiento en ficheros puesto que los datos de la partida guardada solo le interesan al propio usuario, de forma que no es necesario un uso de base de datos y se aprovecha una metodología más rápida y menos dependiente de una red a internet.

Para ello se analizan una serie de conceptos sobre la persistencia de datos:

- **Integridad:** Es la garantía de que los datos almacenados mantendrán su exactitud y consistencia en el tiempo. En NoSQL no es necesario definir el tipo de dato a almacenar, de forma que permite almacenar datos de una forma extremadamente flexible. Se da prioridad especialmente al acceso a los datos.
- **Escalabilidad:** La capacidad de crecimiento de la base de datos, especialmente ágil en las bases de datos NoSQL facilitando la expansión de la base de datos de una forma rápida y barata.
- **Velocidad:** Capacidad de escribir y leer en una base de datos en un periodo de tiempo. En NoSQL se suele contar con mecanismos de búsqueda sumamente rápida para conseguir un dato específico entre millones. La principal ventaja en la velocidad es que puedes diseñar la base de datos en función de las consultas esperadas.
- **Consistencia vs Redundancia:** En SQL, consiste en asegurarse de que un único dato esté una única vez en toda la base de datos. Mientras que en NoSQL, la redundancia es repetir adrede los datos a conveniencia en varias partes de la base de datos. De forma que se pueden almacenar los mismos datos de un usuario dentro de cualquier colección de forma que pueda pertenecer a distintos objetos, como por ejemplo al administrador.

En definitiva, la justificación de la elección de NoSQL se basa en lo siguiente:

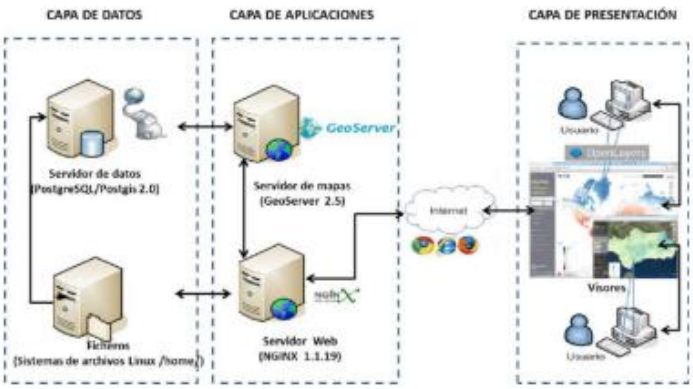
- Facilidad de acceso a los datos y manipulación de los mismos mediante Firebase
- Puesto que el proyecto aún está en las primeras etapas de desarrollo, NoSQL facilitará la modificación de los datos a la hora de incluir al administrador de datos y editor de funcionalidad.
- No requerimos de operaciones atómicas que cambien muchas entidades, En principio el jugador hará un insert de sus datos con su puntuación al terminar la partida, mientras que el administrador solo podrá modificar datos de estos usuarios y el editor podrá modificar valores dentro del juego.
- Si el videojuego es bien recibido por el público, será necesario manejar un volumen de datos mayor que facilitará la escalabilidad horizontal de NoSQL.

4.6. Arquitectura del sistema

Introducción

En los inicios de la informática, la programación era bastante difícil para la mayoría de las personas, pero con el tiempo se han ido descubriendo y desarrollando formas y guías generales para resolver problemas. A las mismas se les ha denominado arquitectura de software, por la semejanza con los planos de un edificio, debido a que la arquitectura indica la estructura, funcionamiento e interacción entre las distintas partes del software. Abordada desde el punto de vista técnico, la arquitectura de software es el diseño de más alto nivel de la estructura de un sistema:

- También denominada arquitectura lógica, integra un conjunto de patrones y abstracciones coherentes que proporcionan un marco definido y claro para interactuar con el código fuente del software.
- Se selecciona y diseña con base en requisitos y restricciones.
    - **Objetivos**: Prefijados por el sistema, no solo son de tipo funcional, sino que también incluyen otros aspectos como el mantenimiento, la auditoría, flexibilidad e interacción con otros sistemas.
    - **Restricciones**: Limitaciones derivadas de las tecnologías disponibles.
- Define de manera abstracta los componentes que llevan a cabo tareas de cómputo, sus interfaces y la comunicación entre los mismos. La arquitectura general del sistema especifica las distintas particiones físicas del mismo, su descomposición lógica en subsistemas de diseño y la ubicación de cada subsistema en cada partición.



Arquitectura del proyecto

La arquitectura empleada en **swz** será una arquitectura en tiempo real debido a la utilización de Firebase. Una arquitectura en tiempo real es un software cuyo correcto funcionamiento depende de los resultados producidos por el mismo y del instante de tiempo en el que se producen esos resultados. Hay dos tipos:

- **Soft**: se degrada si los resultados no se producen correctamente de acuerdo con los requisitos especificados.
- **Hard**: su funcionamiento será incorrecto si los resultados no se producen de acuerdo con la especificación temporal.

swz no depende estrictamente de estos cambios, por lo que se podrá definir su arquitectura como tipo **soft**. Ya que el único cambio que presentará en tiempo real sin necesidad de refrescar la pantalla ni volver a cargar la ventana será el ranking de las mejores puntuaciones (algo no esencial para el funcionamiento del juego)

Para la aplicación móvil prevista para una ampliación futura, se utilizará una arquitectura orientada a servicios. Cada servicio es una representación lógica de una actividad de negocio que tiene un resultado de negocio específico como puede ser consultar los datos bancarios de un cliente. En Firebase, si un usuario quiere autenticarse lo hará contra el servicio de autenticación de Firebase. Si quiere subir una imagen como foto de perfil, esta se subirá al servicio de almacenamiento **Storage.** Mientras tanto los datos necesarios para el ranking de puntuaciones serán subidos al servicio de base de datos **Realtime database.**

swz está conformado por dos clientes (videojuego y app móvil), válidos en exclusividad para el sistema operativo Android utilizando Firebase como backend. El backend está dividido entre los diferentes servicios que se utilizarán durante la vida de **swz**.

- **Autenticación:** la base de datos tendrá dos tipos de autenticación o registros al mismo tiempo. Uno de ellos será el del videojuego en el que el usuario indicará su nombre y correo electrónico y será almacenado en la base de datos automáticamente. Este jugador realmente no estaría realizando una acción de autenticación, puesto que no se le generará una cuenta para su uso, sino más bien realizará una acción de inscripción en la que se le asignará su puntuación máxima durante la partida a su cuenta de uso exclusivo por administradores.
  Por otro lado, estaría la autenticación de la app en la cual se registrarían y autenticarían administradores y editores (los jugadores tendrán acceso a la app con una sesión de invitado en la que podrán consultar el ranking a tiempo real).
  Esta serie de autenticaciones se realizan contra el servicio de Firebase con un email y contraseña además de la posibilidad de utilizar el servidor de google para registrarse e iniciar sesión.
- **Firestore Database:** hace referencia a la base de datos en tiempo real. Cuando un jugador finalice su partida, añadirá sus datos y su puntuación a la base de datos NoSQL. Firebase almacenará un número de registros ilimitado para la colección del ranking que contendrá los nombres de los jugadores que hayan decidido registrar su puntuación y la puntuación propiamente dicha. Aun así, al no ser datos excesivamente relevantes (sobre todo a partir de los 10 mejores), se procurará eliminar cada cierto tiempo los registros cuya posición en el ranking sea superior a 1000.
- **Storage:** Es el servicio de almacenamiento de ficheros. En él se almacenarán las fotos de perfil de los administradores y editores de la aplicación en lugar de guardarlas en la propia base de datos como una cadena de bytes.

La comunicación utilizada tanto por el juego como por la aplicación para conexión con base de datos será el protocolo HTTP.

5. Implementación de la solución.

5.1. Análisis tecnológico.

Para realizar un análisis de las distintas tecnologías tanto utilizadas como alternativas, En primer lugar, se hará una división de las mismas entre la parte del videojuego y la parte de la aplicación administradora del mismo.

Videojuego

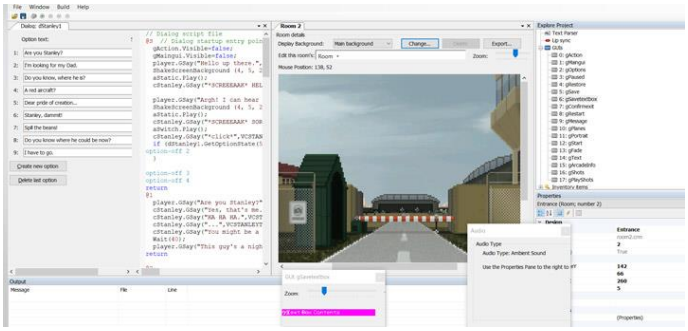
A la hora de elegir la herramienta más adecuada para el diseño de un videojuego, existen varias opciones que se deben analizar previamente para tomar la mejor decisión posible. Las opciones más destacadas son:

- **GameMaker Studio:** una potente suite de desarrollo de software propietario usada principalmente para juegos 2D. Utiliza una interfaz intuitiva para arrastrar y soltar, y su propio lenguaje de programación basado en C, GML.
  Se pueden crear la mayoría de **tipos de juegos** en GM, desde carreras con vista cenital y juegos de rol hasta aventuras de apuntar y hacer clic, y plataformas clásicas. Es compatible con sombreadores, iluminación en tiempo real, física, partículas y más, todo a través de GML.
  Puedes crear juegos simples sin ingresar una línea de código. De igual forma, es posible lograr puntuaciones, puntos de experiencia, menús e IA arrastrando y soltando íconos simplemente etiquetados. Tampoco necesitarás ningún software de gráficos externo, ya que tiene su propio software integrado y completo con herramientas de animación.

- **Unity:** Si quieres crear un juego en 3D, este es un buen lugar para comenzar. Es mucho más complicado que GameMaker, pero tiene una gran cantidad de soporte en línea. Los juegos integrados en el motor se pueden ejecutar en PC, iPhone, Android, navegadores web y otras plataformas.
  Unity es la forma más sencilla de **crear juegos en 3D** para principiantes. Asimismo, es lo suficientemente versátil como para crear cualquier juego en él, incluso en 2D. Tanto desarrolladores independientes como editores de triple A utilizan el motor. La versión básica es gratuita y puedes vender juegos hechos en ella sin regalías, siempre que ganen menos de $100,000.

- **Adventure Game Studio:** permite crear juegos de aventuras en el molde de los juegos populares de LucasArts, como Monkey Island y Full Throttle. Esto significa que no es ideal para juegos con mecánicas que no sean rompecabezas de apuntar y hacer clic, pero es una gran plataforma para mostrar historias, diálogos y diseñar rompecabezas.
  No se necesitan conocimientos de codificación, pero es una herramienta opcional para hacer que tu juego sea más sofisticado. Puedes preparar un juego de aventuras simple un fin de semana con AGS.

Maneja el diálogo y la búsqueda de caminos para que puedas concentrarte en la escritura y el arte. La mayor limitación es que solo puedes crear juegos para Windows, por lo que no podrás obtener tus juegos en plataformas móviles. Esta herramienta utiliza licencia artística.



- **RPG Maker:** se diseñó teniendo en cuenta los juegos de rol de estilo japonés y viene con conjuntos de gráficos básicos, por lo que puedes armar algo sin crear ningún activo en absoluto.
  Aun así, algunas personas han logrado extender el motor para diseñar juegos de disparos y aventuras. Pero es más adecuado para jugadores de rol con vista cenital al estilo de los juegos de SNES, como Chrono Trigger y Final Fantasy.
  RPG Maker tiene su propio lenguaje de secuencias de comandos basado en Ruby, pero no es esencial para crear un videojuego. Puedes diseñar diálogos, combates, exploración y cinemáticas utilizando la sencilla interfaz basada en Windows. Está diseñado pensando en los principiantes, por lo que es ideal para el desarrollo de nivel de entrada. Emplea una licencia de software privado.
- **Unreal Engine:** Es probable que Unreal Engine (UE) tenga la curva de aprendizaje más alta de estas **herramientas de diseño de videojuegos** debido a que está repleta de funciones avanzadas de vanguardia. No obstante, quizás también sea la más profesional.
  El aspecto más exclusivo de UE es su sistema Blueprint, que permite a los usuarios desarrollar rápidamente una lógica de juego compleja y juegos completos sin interactuar con una sola pieza de código. Pero, no te preocupes; aquellos que prefieren codificar también tienen esa opción.
  Unreal Engine facilita la exportación a plataformas populares a través de dispositivos móviles, computadoras, consolas de juegos y sistemas de realidad virtual. Su modelo de pago también se basa en el éxito de los juegos que crea, por lo que puedes usar el software de forma gratuita. No obstante, cada vez que ganes $3,000 en un trimestre, pagas una regalía del 5% de tus ingresos totales.

En conclusión, Para el desarrollo del videojuego, swz se decanta por la opción de **Unity**, a pesar de que Unreal Engine parece presentar unas ventajas y profesionalidad mayores. Esto se debe a que el conocimiento del equipo de desarrollo acerca de Unity es mucho mayor y, además como se describe en el punto de Unity, está mejor preparado para principiantes aun presentando una alta versatilidad a la hora de desarrollar.

Aplicación administradora.

La intención de swz es diseñar una aplicación multiplataforma para poder usarla desde cualquier dispositivo móvil, ya sea Android o IOS. Para ello, la mejor opción es dejar de lado las opciones de desarrollo nativo y acudir a tecnologías que permitan realizar un desarrollo de una solución multiplataforma. En este tipo de tecnologías, las que más destacan son las siguientes:

- **React Native:** Es un framework de Javascript, creado por Facebook, y que permite crear aplicaciones reales nativas para iOS y Android, basado en la librería de Javascript React. En vez de realizar una aplicación web híbrida o en HTML lo que se obtiene con este framework es una aplicación real nativa. Utiliza el paradigma fundamental de construcción de bloques de interfaz de usuario (componentes visuales con los que interacciona el usuario) y gestiona la interacción entre los mismos utilizando las capacidades de Javascript y React. Las características principales de este framework son:
    - Compatibilidad: Permite crear aplicaciones que pueden ser ejecutadas tanto en iOS como en Android con el mismo código fuente.
    - Funcionalidad nativa: Las aplicaciones creadas con este framework funciona de la misma manera que una aplicación nativa real sin el uso de un WebView.
    - Hot-reload: Podemos ver los cambios que ejecutamos al instante, debido a la gran velocidad de compilación.
    - Sencillo de aprender: Muy fácil de leer y comprender tanto si conoces Javascript como si eres principiante en ese lenguaje de programación.
    - Experiencia positiva para el desarrollador: Funcionalidades como el hot-reload y el flexbox layout engine (nos permite abstraernos de los detalles de cada layout en iOS y Android) hacen que el desarrollo sea más satisfactorio para el programador.

Otra de las cosas a destacar de este framework es el gran soporte que tiene y la extensa documentación que facilita.

Su principal desventaja es que utiliza un bridge de Javascript que es el que se encarga de "convertir" nuestro código Javascript en código nativo de la plataforma de ejecución, lo que lo hace un poco menos veloz que otras alternativas.

Utiliza una licencia que permite reutilizar software dentro de software propietario conocida como licencia MIT

- **Ionic:** Se trata de un framework de código abierto que se utiliza destinado al desarrollo de aplicaciones híbridas, es decir, en la que se utilizan HTML, Javascript y CSS para generar interfaces y se "envuelven" en una aplicación nativa que la muestra en un WebView. Incluso el despliegue se realiza en plataformas nativas (Play Store y App Store).
  Lo bueno, es que, junto a este framework, puedes utilizar otro framework de desarrollo front-end web como es Angular, que agilizará el desarrollo mucho más que si se realiza con Vanila Javascript (Javascript puro). Como principales desventajas, destaca que su rendimiento puede ser ligeramente menor en comparación a una aplicación nativa.

- **Kotlin Native:** Se trata de una tecnología que permite compilar el código escrito en Kotlin directamente a archivos binarios nativos, por lo que puede ser utilizado sin la necesidad de una máquina virtual (como sería el caso de Java). Actualmente ofrece soporte para plataformas como iOS, MacOS, Android, Windows, Linux y WebAssembly.
  Funciona bajo la licencia apache 2.0.

- **Native Script:** Se trata de una alternativa bastante similar a Ionic, que permite construir aplicaciones para Android e iOS utilizando lenguajes de programación independientes del dispositivo (Javascript o Typescript). También soporta desarrollo directamente con Angular y con Vue.js mediante un complemento desarrollado por la comunidad. Además, se pueden utilizar bibliotecas de terceros como Maven y npm en las aplicaciones móviles
  Como desventajas de esta herramienta encontramos:
  - No tiene una curva de aprendizaje sencilla.
  - No se puede reutilizar buena parte del código de la web.
  - No tiene un conjunto de componentes de interfaz muy destacable, siendo superado por el de Ionic.

  Al igual que kotlin native, funciona bajo una licencia Apache.

- **Xamarin:** Es una plataforma de código abierto para crear aplicaciones multiplataforma (iOS, Android y Windows) con .NET. Se ejecuta en un entorno administrado que proporciona ventajas como la asignación de memoria y la recolección de elementos no utilizados. Permite a los desarrolladores compartir un promedio de un 90% del código entre plataformas.
  Una de sus principales desventajas es su coste para uso profesional empresarial. Por ejemplo, es necesario comprar una licencia de Visual Studio Professional que cuesta 1.199 dólares el primer año y 799 por renovación.

- **Flutter:** Es un framework de código abierto desarrollado por Google parar crear aplicaciones nativas de forma fácil, rápida y sencilla. Su principal ventaja radica en que el código que genera es 100% nativo para cada plataforma, consiguiendo un rendimiento y una UI idénticos a las aplicaciones nativas tradicionales.
  Pese a las promesas de otros frameworks con el mismo propósito, se trata del único en generar código 100% nativo y ejecutable en ambas plataformas. Además presenta otras ventajas como:
  - **Experiencia de usuario.** Flutter incluye Material Design de Google y Cupertino de Apple por lo que la experiencia de usuario es óptima y las interfaces idénticas a las de las aplicaciones desarrolladas de forma nativa.
  - **Tiempo de carga.** Con Flutter se experimentan tiempos de carga por debajo de un segundo en cualquiera de las plataformas.
  - **Desarrollo ágil y rápido:** Gracias al hot-reload.
  - Se puede programar desde cualquier sistema operativo.
  - Rendimiento superior a su máximo competidor, React Native, ya que no utiliza un bridge.

  Además, este framework es cada vez más conocido y utilizado por desarrolladores multiplataforma y tiene un gran futuro por delante, debido a que google está desarrollando su sistema operativo Fucsia, que si finalmente sale al mercado sería un salto enorme para Flutter.
  Este framework utiliza un lenguaje llamado Dart, que también es creado por Google y sencillo de aprender debido a sus similitudes con C++, c# o Java.
  Al igual que Java, Dart se ejecuta sobre una máquina virtual (Dart VM) que permite a Flutter evitar la necesidad de tener otra capa de lenguaje declarativo como CML para realizar las interfaces, porque el propio layout es definido en Dart y es fácil de entender y rápido de desarrollar.

En conclusión, Para el desarrollo de la aplicación administradora, swz se decanta por la opción de **Flutter** debido a su gran fama entre los desarrolladores, su facilidad y agilidad de desarrollo y el prestigio que le otorga tener detrás al mismísimo Google.

Además, estos son otros de los motivos:

- utilizando el propio Dart, no será necesario el uso y la definición de vistas html ni xml.
- Muestra unas interfaces increíbles, totalmente iguales a las de aplicaciones nativas.
- Al ser un producto de Google como Firebase tiene una buenísima integración con esta última tecnología.
- Posibilidad de utilizar el Hot Reload para ver cambios y realizar pruebas al momento.

5.2. Elementos a implementar.

A continuación, se mostrarán las implementaciones más relevantes en el desarrollo de swz como pueden ser el manager del juego, los movimientos y acciones del jugador y los enemigos, etc.

Game Manager

En primer lugar, el script encargado del funcionamiento del juego. Se encarga de aumentar las rondas generando los enemigos correspondientes, además de instanciar cada x tiempo los botiquines que permitirán al jugador recuperar parte de su vida. Y otras cosas más.

Al arrancar el juego, el GameManager se encarga de comprobar si hay datos guardados, cargar el script encargado de controlar al jugador para poder acceder a sus funciones, y establecer la dificultad al modo fácil.

```
void Start()
    {
        comprobarDatosGuardados();
        tpc =
GameObject.Find("PlayerArmature").GetComponent<ThirdPersonController>();
        txtButtonDificultad.text = "Dificultad: Fácil";
    }
```

Cada vez que corresponde pasar de nivel se lanzará la ejecución del método nextLevel que, en caso de que no se haya terminado el juego, generará un numero de enemigos según la dificultad escogida, que oscilará entre 1 y 3.

```
private void nextLevel()
    {
        if (!isGameOver && isInGame)
        {
            if (firstScene && ronda == rondaCambioEscena)
StartCoroutine(pasarEscena());
            else
            {
                if (ronda <= rondaFinal)
                {
                    ronda++;
                    numEnemies = Random.Range(dificultad * ronda, (2 *
dificultad * ronda) + 1);
                    contadorEnemigos.text = "Enemigos: " + numEnemies;
                    contadorRondas.text = "Ronda: " + ronda;
                    if (firstScene)
respawnEnemies(enemyRespawnPointsGraveyard);
                    else respawnEnemies(enemyRespawnPointsCastle);
                }
                else finGame();
            }
        }
    }
```

Si al avanzar de nivel, el jugador se encuentra en la ronda del cambio de escena, se iniciará una corrutina que guardará los object necesarios y establecerá los cambios necesarios para progresar en el juego. Finalmente se vuelve a llamar al método nextLevel, ya que se salta la generación de enemigos al crear la escena para no crear conflictos.

```
IEnumerator pasarEscena()
    {
        firstScene = false;
        //desactivación del control para evitar problemas a la hora del
transporte
        tpc.enabled = false;

        tpc.mostrarMensaje("Ronda completada");
        yield return new WaitForSeconds(2);
        GameObject player = GameObject.Find("PlayerArmature");
        GameObject respawnPlayer = GameObject.Find("respawnPlayer");

        //transporte del jugador al punto de inicio de la siguiente
escena establecido mediante un object en el apartado gráfico del cual
toma referencia
```

```
        player.gameObject.transform.position = new
Vector3(respawnPlayer.transform.position.x, player.transform.position.y,
respawnPlayer.transform.position.z);
        player.gameObject.transform.rotation =
respawnPlayer.transform.rotation;
        yield return new WaitForSeconds(0.1f);

        //reactivación del control del jugador
        tpc.enabled = true;

        //Carga de la nueva escena manteniendo el jugador y el propio
GameManager
        DontDestroyOnLoad(GameObject.Find("Player"));
        DontDestroyOnLoad(this);
        SceneManager.LoadScene("Castle");
        yield return new WaitForSeconds(0.1f);

        //Carga de los nuevos respawns de enemigos y botiquines
        cargarNewRespawns();
        nextLevel();
    }

private void cargarNewRespawns()
    {
        enemyRespawnPointsCastle = new GameObject[7];
        for (int i = 0; i < enemyRespawnPointsCastle.Length; i++)
        {
            enemyRespawnPointsCastle[i] = GameObject.Find("RespawnCastle
(" + (i + 1) + ")");
        }
        healthyRespawnPointsCastle = new GameObject[8];
        for (int i = 0; i < healthyRespawnPointsCastle.Length; i++)
        {
            healthyRespawnPointsCastle[i] =
GameObject.Find("HealthyRespawnCastle (" + (i + 1) + ")");
        }
        healthyRespawnPoints = healthyRespawnPointsCastle;
        healthyRespawnsUtilizados.Clear();
    }
```

Para respawnear los enemigos se lanza un método que recorre los puntos de respawn disponibles colocando un enemigo en cada uno de ellos cada vez que pasa, dando vueltas hasta llegar al número de enemigos que se deben generar en esta ronda.

```
private void respawnEnemies(GameObject[] respawnPoints)
    {
        ArrayList respawns = new ArrayList(respawnPoints);
        GameObject respawnPoint;
        for (int i = 0; i < numEnemies; i++)
        {
            if (respawns.Count == 0) respawns = new
ArrayList(respawnPoints);
            int enemyIndex = Random.Range(0, enemies.Length);
            respawnPoint = (GameObject)respawns[Random.Range(0,
respawns.Count)];
            respawns.Remove(respawnPoint);
            Instantiate(
                enemies[enemyIndex],
                respawnPoint.transform.position,
                respawnPoint.transform.rotation
                );
        }
    }
```

Cuando un enemigo muere, se decrementa en 1 el número de enemigos que hay hasta llegar a 0. Momento en el que se llama a la función nextLevel para avanzar en el progreso.

```
public void enemyDeath()
    {
        numEnemies--;
        if (numEnemies == 0) nextLevel();
        else contadorEnemigos.text = "Enemigos: " + numEnemies;
    }
```

Cuando el jugador comienza la partida con un simple clic, se lanza la función comenzar partida, encargada de iniciar la primera ronda del juego e iniciar una corrutina encargada de generar botiquines cada x tiempo.

```
public void comenzarPartida()
    {
        GameIntroPanel.SetActive(false);
        GamePanel.SetActive(true);
        isInGame = true;
        nextLevel();
        healthyRespawnPoints = healthyRespawnPointsGraveyard;
        StartCoroutine(generateHealthy());
    }
```

La corrutina encargada de los botiquines generará un número de botiquines aleatorio entre 1 y el número de respawns no utilizados.

```
IEnumerator generateHealthy()
    {
        int seconds = 30;
        yield return new WaitForSeconds(seconds);
        while (!isGameOver)
        {
            int numResp = 0;
            if (healthyRespawnsUtilizados.Count !=
healthyRespawnPoints.Length)
            {
                numResp = Random.Range(1, (healthyRespawnPoints.Length -
healthyRespawnsUtilizados.Count) + 1);
            }
            for (int i = 0; i < numResp; i++)
            {
                GameObject rp;
                do
                {
                    rp = healthyRespawnPoints[Random.Range(0,
healthyRespawnPoints.Length)];
                } while (healthyRespawnsUtilizados.Contains(rp));
                healthyRespawnsUtilizados.Add(rp);
                Instantiate(healthyObject, rp.transform);
            }
            yield return new WaitForSeconds(seconds);
        }
    }
```

Cuando el jugador colisiona con uno de estos botiquines, se lanza desde el script del botiquín, una llamada a la función encargada de dejar libre el punto de respawn utilizado por ese botiquin.

```
public void cogerHealthy(GameObject healthy)
    {
        foreach (GameObject rp in healthyRespawnsUtilizados)
        {
            if (rp.transform.position.x == healthy.transform.position.x
                && rp.transform.position.z ==
healthy.transform.position.z)
            {
                healthyRespawnsUtilizados.Remove(rp);
                break;
            }
        }
    }
```

Cuando el jugador muere, se lanza la función de gameOver que muestra un panel indicando la ronda a la que has llegado y un botón de restart para recargar la escena. (En este punto se implementará la introducción de un correo para subir la puntuación del jugador a Firebase de cara a la entrega final)

```
public void gameOver()
    {
        isGameOver = true;
        GamePanel.SetActive(false);
        GameOverPanel.SetActive(true);
        resultadoPartida.text = "Has llegado a la ronda " + ronda + ",
¡Vuelve a intentarlo!";
        tpc.enabled = false;
    }

    public void RestartGame()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
```

PlayerManager

A continuación, es el turno del manager de la vida, las armas y los puntos.

Al cargar el jugador, se establece la vida del jugador al valor máximo, además de cargar componentes necesarios como el GameManager, animator, audio, lista de armas y el propio script controlador ThirdPersonControler.cs

```
private void Start()
    {
        vidaActual = vidaMaxima;
        gameManager =
GameObject.Find("GameManager").GetComponent<GameManager>();
        anim = GetComponent<Animator>();
        audio = GetComponent<AudioSource>();
        armas = new List<GameObject>();
        tpc = GetComponent<ThirdPersonController>();
    }
```

Por cada frame del juego, se lanza la función update, que se encarga de comprobar la salud del jugador y establecerla correctamente en la barra de vida, aumentando o reduciendo la misma según sea necesario mientras no esté muerto.

```
void Update()
    {
        if (!dead)
        {
            if (vidaActual <= 0)
            {
                vidaActual = 0;
                dead = true;
                gameManager.gameOver();
            }
            if (vidaActual > vidaMaxima) vidaActual = vidaMaxima;
            healthBar.fillAmount = vidaActual / vidaMaxima;
            healthPoints.text = vidaActual.ToString();
        }
    }
```

Cuando el jugador se acerca a una tienda, si acepta la opción de comprar se lanzarán las funciones encargadas de comprobar si puede pagar el precio establecido y, en caso afirmativo, se lanzará la función de venta que le restará los puntos y le cederá el arma comprada.

```
internal bool puedePagar(int precio)
    {
        return precio <= puntos;
    }

    internal void venderArma(GameObject arma, int precio)
    {
        addPuntos(-precio);
        armas.Add(arma);
        tpc.equiparArma(arma);
        puntoMira.SetActive(true);
    }
```

Estas funciones serán llamadas desde el script de la tienda al pulsar el botón comprar, que lanzará la función.

```
public void comprar()
    {
        if (playerManager.puedePagar(precio))
        {
            playerManager.venderArma(arma, precio);
            armaComprada = true;
            mostrarMensaje("Arma comprada");
            panelTienda.SetActive(false);
        }
        else
        {
            mostrarMensaje("No tienes suficientes puntos para adquirir
este arma");
            panelTienda.SetActive(false);
        }
    }
```

ZombieMovement

Script dedicado al comportamiento de los enemigos de tipo zombi. Por cada frame se modificará la rotación del zombi haciendo que mire en todo momento al jugador, esto hará que, al avanzar hacia delante, persiga al jugador. Además, si está suficientemente cerca del jugador, iniciará un ataque.

```
void Update()
    {
        if (!isDead)
        {
            Vector3 playerPosition = new
Vector3(Player.transform.position.x, 0, Player.transform.position.z);
            transform.LookAt(playerPosition);
            if (!isAttacking && !isGrowling || isRunning)
            {
                float distanciaActual = (Player.transform.position -
transform.position).magnitude;
                if (distanciaActual > distanciaCuerpoACuerpo)
                {
                    transform.Translate(Vector3.forward * movementSpeed *
Time.deltaTime);
                }
                else
                {
                    StartCoroutine(attack());
                }
            }
        }
```

```
    }

IEnumerator attack()
    {
        isAttacking = true;
        animator.SetBool("isAttacking", isAttacking);
        yield return new WaitForSeconds(0.8f);
        isDamaging = true;
        yield return new WaitForSeconds(1.4f);
        isDamaging = false;
        isAttacking = false;
        animator.SetBool("isAttacking", isAttacking);
        contDamages = 0;
    }
```

Si al atacar, alcanza al jugador, le hará una cantidad de daño preestablecida.

```
private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            if (contDamages == 0 && isDamaging)
            {
                contDamages++;
                playerManagement.pierdeVida(damage);
            }
        }
    }
```

También, al instanciar al enemigo se iniciará una corrutina que hará que el zombi se enfurezca cada x tiempo, lanzando un gruñido y acelerando su paso durante 5 segundos.

```
IEnumerator growl()
    {
        yield return new WaitForSeconds(Random.Range(10, 30));
        while (!isDead)
        {
            isGrowling = true;
            animator.SetBool("isGrowling", isGrowling);
            audioGrowl.Play();
            yield return new WaitForSeconds(2);
            isGrowling = false;
            animator.SetBool("isGrowling", isGrowling);
            yield return StartCoroutine(run());
            yield return new WaitForSeconds(Random.Range(10, 30));
        }
    }
```

```
    IEnumerator run()
    {
        isRunning = true;
        animator.SetBool("isRunning", isRunning);
        animator.SetBool("isWalking", !isRunning);
        movementSpeed = RUN_SPEED;
        yield return new WaitForSeconds(5);
        movementSpeed = WALK_SPEED;
        isRunning = false;
        animator.SetBool("isWalking", !isRunning);
        animator.SetBool("isRunning", isRunning);
    }
```

Al morir, se manda al jugador una cantidad de puntos preestablecida.

```
      public void morir()
    {
        playerManagement.addPuntos(puntosPorBaja);
    }
```

Resto de scripts

Si el jugador realiza un ataque cuerpo a cuerpo y choca con algo, se lanzará la siguiente función que, en caso de alcanzar a un enemigo, le restará salud.

```
private void OnTriggerEnter(Collider other)
    {
        if (!health.isDead())
        {
            if (other.gameObject.CompareTag("enemy"))
            {
                if (attacking)
                {
                    EnemyHealth enemyHealth =
other.gameObject.GetComponent<EnemyHealth>();
                    enemyHealth.recibeDamage(20);
                }
            }
        }
    }

public void recibeDamage(int damage)
    {
        if (!isDead)
        {
            vidaActual -= damage;
            if (vidaActual < 0) vidaActual = 0;
            else if (vidaActual > vidaMaxima) vidaActual = vidaMaxima;
            StartCoroutine(damageAnim());
```

```
        }
    }

IEnumerator damageAnim()
    {
        animator.SetBool("isTakingDamage", true);
        if (vidaActual > 0)
        {
            yield return new WaitForSeconds(0.5f);
            animator.SetBool("isTakingDamage", false);
        }
        else
        {
            animator.SetBool("isDead", true);
            enemyMovement.morir();
            gameManager.enemyDeath();
            isDead = true;
            aprovisionar();
            yield return new WaitForSeconds(5);
            Destroy(gameObject);
        }
    }
```

Al morir un enemigo, se otorgará una granada al jugador en una probabilidad del 10%

```
private void aprovisionar()
    {
        int prob = Random.Range(1, 101);
        if (prob <= 10)
        {
            tpc.addGranadas();
        }
    }
```

Cuando el jugador lanza una granada, suponiendo que la tiene, se instanciará una granada en la mano del jugador que será lanzada con una fuerza hacia delante y hacia arriba predeterminada, además de una rotación aleatoria para hacerlo mas realista.

```
void Start()
    {
        Player = GameObject.Find("PlayerArmature");
        granadeRb = GetComponent<Rigidbody>();
        colliderExplosion = GetComponent<SphereCollider>();
        audio = GetComponent<AudioSource>();
        lanzar();
    }
```

```
void lanzar()
    {
        granadeRb.AddForce(fuerzaUp(), ForceMode.Impulse);
        granadeRb.AddForce(fuerzaFront(), ForceMode.Impulse);
        granadeRb.AddTorque(RandomNumber(-10, 10), RandomNumber(-10, 10),
RandomNumber(-10, 10));
    }

    private int RandomNumber(int min, int max)
    {
        return Random.Range(min, max+1);
    }

    private Vector3 fuerzaUp()
    {
        return Vector3.up * 4;
    }
    private Vector3 fuerzaFront()
    {
        return Player.transform.forward * 10;
    }
```

Cuando la granada choca con cualquier objeto que no sea un enemigo, está explotará lanzando rayos en todas las direcciones llevándose por delante a todos los enemigos dentro del alcance de la misma.

```
private void OnCollisionEnter(Collision collision)
    {
        if (!collision.gameObject.CompareTag("enemy"))
        {
            audio.Play();
            StartCoroutine(explotar());
        }
    }

    IEnumerator explotar()
    {
        explosion.Play();
        colliderExplosion.enabled = true;
        yield return new WaitForSeconds(2);
        Destroy(gameObject);
    }
```

```
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("enemy"))
        {
            EnemyHealth enemyHealth =
other.gameObject.GetComponent<EnemyHealth>();
            enemyHealth.recibeDamage(50);
        }
    }
```

Los botiquines poseen unas propiedades generadas aleatoriamente en el momento en que se instancia cada uno de ellos

```
void Start()
    {
        int prob = Random.Range(0, 101);
        if (prob <= 20) healthPoints = Random.Range(50, 101);
        else healthPoints = Random.Range(20, 51);
    }
```

Cuando el jugador colisiona con el botiquín, se lanza la función que aumenta la vida del jugador según la propiedad healthPoints del botiquín y elimina el botiquín de la escena.

```
private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            int vida = healthPoints;
            other.GetComponent<PlayerManager>().addVida(vida);
            GameObject.Find("GameManager").GetComponent<GameManager>().co
gerHealthy(other.gameObject);
            Destroy(this.gameObject);
        }
    }
```

Se ha establecido un límite en el mapa que hará que todo objeto que salga de ese límite sea automáticamente eliminado de la escena, con la intención de eliminar posibles bugs, como que un enemigo atraviese el suelo en el momento en que se generan al superponerse unos encima de otros cuando son un número elevado. Esta eliminación se notifica al gameManager para que reste a dicho enemigo del contador que lleva de los mismos para controlar las rondas.

```
private void OnTriggerExit(Collider other)
    {
        if (other.gameObject.CompareTag("enemy"))
        {
            gameManager.enemyDeath();
        }
        Destroy(other.gameObject);
    }
```

Por otro lado, se ha generado una clase Mensaje, encargada de mostrar por pantalla de forma animada un texto enviado por parámetro.

```
public class Mensaje : MonoBehaviour
{
    public TextMeshProUGUI msg;
    private Animator animText;
    IEnumerator animacionTexto()
    {
        animText = msg.GetComponent<Animator>();
        msg.enabled = true;
        animText.SetBool("mostrar", true);
        yield return new WaitForSeconds(2);
        animText.SetBool("mostrar", false);
        msg.enabled = false;
    }

    internal void mostrar(string mensaje)
    {
        msg.text = mensaje;
        StartCoroutine(animacionTexto());
    }
}
```

Se han generado puntos de teletransporte que transmitirá al jugador y a los enemigos de un punto a otro de forma instantánea. Al colisionar con uno de estos puntos, el jugador o enemigo serán transportados al punto de destino. En caso de ser el jugador, se reproducirá un sonido característico del teletransporte.

```
private void OnCollisionEnter(Collision other)
    {
        if (!other.gameObject.CompareTag("ground"))
        {
            if (other.gameObject.CompareTag("Player"))
StartCoroutine(transportPlayer(other));
            else other.gameObject.transform.position =
teleportDestino.transform.position;
        }
    }

    IEnumerator transportPlayer(Collision player)
    {
        audio.Play();
        playerScript.enabled = false;
        player.gameObject.transform.position =
teleportDestino.transform.position;
        player.gameObject.transform.rotation =
teleportDestino.transform.rotation;
        yield return new WaitForSeconds(0.1f);
```

```
        playerScript.enabled = true;
    }
```

Guardar partida

Para guardar el progreso del juego y poder continuar cuando al jugador le apetezca, swz utiliza un sistema de serialización de datos a un fichero almacenado en la memoria del teléfono. En dicho fichero se almacena un objeto Partida, que contiene los datos necesarios para recuperar el estado de la partida por donde se dejó, como son la escena, la ronda, los puntos acumulados, la dificultad escogida, si tiene arma equipada y el número de granadas que posee.

```
[System.Serializable]
public class Partida
{
    public static Partida current;
    private bool hasArma;
    private int scene;
    private int ronda;
    private int puntos;
    private int dificultad;
    private int granadas;

    public Partida()
    {
        this.hasArma = false;
        this.scene = 0;
        this.ronda = 0;
        this.puntos = 0;
        this.dificultad = 0;
        this.granadas = 0;
    }
}
```

Esta clase también incluye sus getters & setters.

Por otro lado, se ha creado una clase SaveLoad encargada de la gestión del fichero, tanto en lectura como en escritura. Además, se han implementado de forma adicional dos funciones. Una para eliminar el fichero al morir o al terminar la partida, y otra para comprobar la existencia de ese fichero y, de esa forma, saber si se deben cargar datos o no.

```
public static class SaveLoad
{
    private static string fileName = Application.persistentDataPath +
"/datosGuardados.swz";

    public static void save()
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = File.Create(fileName);
        bf.Serialize(file, Partida.current);
        file.Close();
    }

    public static void Load()
    {
        if (existeFichero())
        {
            BinaryFormatter bf = new BinaryFormatter();
            FileStream file = File.Open(fileName, FileMode.Open);
            Partida.current = (Partida)bf.Deserialize(file);
            file.Close();
        }
    }

    public static void borrarDatos()
    {
        File.Delete(fileName);
    }

    public static bool existeFichero()
    {
        return File.Exists(fileName);
    }
}
```

Una vez vistas las clases necesarias para gestionar los datos guardados, es el turno del object encargado de llamar a estas clases y aplicar los cambios, el GameManager.

En primer lugar, al iniciar el juego, se instancia el atributo estático 'current' como un nuevo objeto Partida. Se comprueba si existen datos y, en caso afirmativo, se prepara el menú principal para que el jugador pueda continuar o, si lo prefiere, comenzar una partida nueva.

```
void Start()
    {
        Partida.current = new Partida();
        . . .
        comprobarDatosGuardados();
    }

private void comprobarDatosGuardados()
    {
        SaveLoad.Load();
        if (Partida.existenDatos())
        {
            btnNuevaPartida.SetActive(true);
            btnComenzar.text = "Continuar";
            dificultad = Partida.getDificultad();
            switch (dificultad)
            {
                case 1:
                    txtButtonDificultad.text = "Dificultad: Fácil";
                    break;
                case 2:
                    txtButtonDificultad.text = "Dificultad: Medio";
                    break;
                case 3:
                    txtButtonDificultad.text = "Dificultad: Difícil";
                    break;
            }
        }
    }
```

Tras esto, el jugador podrá decidir si continua con la partida por donde la dejó, o empezar una nueva desde el principio. Si decide continuar, se recogerán los datos del objeto Partida 'current' previamente cargado y se asignarán a las variables oportunas. Tras ello, comenzará el juego.

```
public void continuar()
    {
        comprobarArma();
        scene = Partida.getScene();
        pm.addPuntos(Partida.getPuntos());
        ronda = Partida.getRonda() - 1;
        tpc.cargarGranadas(Partida.getGranadas());
    }
```

```
    private void comprobarArma()
    {
        if (Partida.HasArma())
        {
            tpc.equiparArma(arma);
            pm.activarMira();
        }
    }
```

En cambio, si decide empezar una partida desde el principio, en primer lugar, se eliminará el fichero que almacena los datos para comenzar desde cero y tras ello, comenzará el juego.

```
public void nuevaPartida()
    {
        SaveLoad.borrarDatos();
        GameIntroPanel.SetActive(false);
        GamePanel.SetActive(true);
        isInGame = true;
        nextLevel();
        healthyRespawnPoints = healthyRespawnPointsGraveyard;
        StartCoroutine(generateHealthy());
    }
```

Finalmente, para guardar la partida el jugador podrá pulsar el botón de pausa en cualquier momento y seleccionar la opción de guardar progreso.

Una vez pulsado el botón de guardar progreso, se almacenarán todos los datos necesarios en el objeto Partida 'current' y se serializará al fichero.

```
public void guardarPartida()
    {
        Partida.setHasArma(GameObject.Find("M1911") != null);
        Partida.setScene(scene);
        Partida.setPuntos(pm.puntos);
        Partida.setRonda(ronda);
        Partida.setDificultad(dificultad);
        Partida.setGranadas(tpc.numGranadas);
        SaveLoad.save();
    }
```

Ranking

Al finalizar la partida, ya sea que resulte en una gran victoria o en una aplastante derrota, se ofrecerá al usuario la oportunidad de registrar su puntuación en un ranking bajo un nombre que el mismo escoja, sin necesidad de registro. Este ranking se podrá consultar en la futura aplicación prevista para Android y Apple. En la pantalla del ranking del juego se podrán consultar las 10 mejores puntuaciones hasta la fecha.

En cuanto a código, el apartado del ranking no es demasiado exigente, tan solo ha sido necesario inicializar firestore, hacer el insert y recoger los datos almacenados.

En primer lugar, para iniciar Firebase se ha utilizado el código establecido en la documentación de Firebase para los juegos Unity.

```
private FirebaseFirestore db;
private string collection = "ranking";
private string idNombre = "nombre";
private string idPuntos = "puntos";
```

```
async void firebaseInit()
    {
        FirebaseApp.CheckAndFixDependenciesAsync().ContinueWith(task =>
        {
            var dependencyStatus = task.Result;
            if (dependencyStatus == Firebase.DependencyStatus.Available)
            {
                db = FirebaseFirestore.DefaultInstance;
            }
            else
            {
                UnityEngine.Debug.LogError(System.String.Format(
                  "Could not resolve all Firebase dependencies: {0}",
dependencyStatus));
            }
        });
    }
```

Una vez inicializado firestore, estará preparado para el registro del jugador y la carga de los datos registrados.

Como se menciona al principio de este punto, al morir el jugador se le ofrece la oportunidad de registrar su puntuación ofreciéndole un cuadro de texto en el que poner su nombre o su alias favorito. Tras ello, al darle al botón registrar se lanzará la función de registro que crea un diccionario con su nombre y los puntos que ha acumulado y lo sube directamente a firestore.

```
public void registrarEnRanking()
    {
        string nombre = txtIdentificadorRanking.text;
        if (!string.IsNullOrEmpty(nombre))
        {
            Dictionary<string, object> reg = new Dictionary<string,
object>
            {
                {idNombre, nombre},
                {idPuntos, pm.puntos}
            };
            db.Collection(collection).Document(nombre + "-" + pm.puntos)
                .SetAsync(reg).ContinueWithOnMainThread(task =>
                {
                    cargarRanking();
                    btnRegistrar.SetActive(false);
                    txtBtnCancelConfirmRanking.text = "Confirmar";
                });
        }
    }
```

Como se puede observar, al terminar la tarea de subir los datos a firestore, se hace una llamada a la carga del ranking. Esto se hace con la intención de añadir el registro del jugador en directo y pueda ver si ha conseguido una plaza en ese ansiado top 10.

Para esta carga del ranking se hace una llamada a firestore para descargar todos los registros en el ranking de forma descendente. Tras terminar de descargar todos los registros, se generan dos strings que completan la muestra por pantalla del ranking. Uno de los strings contiene la posición y el nombre del jugador (este string será colocado en un TextArea en la parte izquierda de la pantalla), y el otro string contendrá la puntuación de cada jugador en el mismo orden que se muestran a la izquierda (este string será colocado en un TextArea en la parte derecha de la pantalla).

```
private void cargarRanking()
    {
        db.Collection(collection).OrderByDescending(idPuntos).Limit(10).G
etSnapshotAsync().ContinueWithOnMainThread(task =>
        {
            string rankingNombres = "";
            string rankingPuntos = "";
            int contador = 1;
            QuerySnapshot qSnap = task.Result;
            Debug.Log(task.Result.ToString());
            foreach (DocumentSnapshot snap in qSnap)
            {
                Dictionary<string, object> reg = snap.ToDictionary();
                rankingNombres += contador++ + ". " + reg[idNombre] +
"\n";
                rankingPuntos += reg[idPuntos] + "\n";
            }
            txtRankingNombres.text = rankingNombres;
            txtRankingPuntos.text = rankingPuntos;
        });
    }
```

Con esto concluiría la parte del ranking hasta que se diseñe la aplicación de Android y apple, en la que se hará uso de estos mismos datos.

Bibliografía/webgrafía

- Estudio de mercado
- Público objetivo, cliente ideal y buyer persona: ¿cuáles son las diferencias?
- Guía fundamental del Análisis DAFO
- Gestión de costos del proyecto
- Estimación de Costos de Software en Gestión de Proyectos Ágiles
- ¿Cuánto cuesta crear una app?
- Consulta un ejemplo de precios de Cloud Firestore
- Visual Studio Code
- Microsoft Office 365
- GitKraken: un excelente cliente de Git multiplataforma para tu escritorio
- ¿Qué es la Prevención de Riesgos Laborales?
- Riesgos laborales en el sector de la informática
- Gestión de riesgos de proyectos software
- Requisito funcional
- Requisitos no funcionales
- Caso de uso
- Diseño de software
- Diseño de interfaz de usuario
- Diagrama de clases
- UML – Diagramas de Clases – Relaciones
- Persistencia de datos
- Bases de Datos NoSQL | Qué son, marcas, tipos y ventajas
- Funciones y ventajas de las bases de datos NoSQL
- ¿Cómo saber si necesitas una Base de Datos NoSQL?
- Arquitectura de software
- Definición de la Arquitectura del Sistema
- Ciclo de vida de un sistema de información: fases y componentes
- ¿Qué es Ionic?
- Kotlin Native
- NativeScript
- ¿Qué es Xamarin?
- ¿Qué es Flutter?
- ¿Es Flutter el framework del futuro?
- Las mejores alternativas a Firebase
- Firebase, qué es y para qué sirve la plataforma de Google
- 5 softwares para el diseño de videojuegos
- Documentación de Instadroid proporcionada por el profesorado