Escuela Politécnica Superior

Comparativa de dos desarrollos de un videojuego

Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:
Jorge García Valera

Tutor/es:
Mireia Luisa Sempere

Universitat d'Alacant
Universidad de Alicante

Mayo 2019

JUSTIFICACIÓN Y OBJETIVOS

La idea de este trabajo nace con la visión de la autosuperación y el crecimiento personal dentro del ámbito del desarrollo de videojuegos.

La realización de este trabajo se plantea de tal forma en que se volverá a desarrollar un antiguo proyecto, el cual cuando se desarrolló no se tenía mucha experiencia en el ámbito del desarrollo de videojuegos ni con el lenguaje de desarrollo. Ahora con más experiencia en ambos campos se pretende mejorar el resultado del producto final.

Para conseguir esta mejora se revisará el proyecto antiguo y se extraerán las ideas y mecánicas principales. Una vez extraídas las ideas y mecánicas se entrará en un proceso en el cual se decidirá si esa mecánica que se pensó en el inicio es válida o no, o si necesita una mejora para que se adapte todavía más al producto final que deseamos.

La idea para este proyecto en mi opinión es original, ya que se sale de la típica idea de únicamente desarrollar un videojuego para demostrar tus conocimientos. Esta idea puede servir para que generaciones futuras no le teman al desarrollo, ya que en sus inicios afrontar un desarrollo de un videojuego puede parecer algo imposible, pero como todo, es cuestión de dedicarle tiempo y mejorar en el entendimiento del lenguaje y de la librería que decidas utilizar. Todo esto estará recopilado al final de documento, ya que se va a estudiar el resultado de ambos proyectos.

A su vez, sirve para que el desarrollador vea cómo ha ido cambiado y mejorando su forma de programar y de afrontar los problemas que surgen durante un desarrollo.

Los objetivos por tratar en este trabajo son:

- Extraer las ideas y mecánicas originales y analizarlas para comprobar que son válidas para el nuevo proyecto.
- Con las nuevas mecánicas e ideas desarrollar una nueva versión del videojuego original.
- Comparar ambas versiones y observar el cambio de rendimiento y de limpieza en el código.

Estos objetivos ponen a prueba las capacidades del desarrollador. El primer punto pone a prueba la capacidad de autocrítica y de análisis del programador al tener que buscar, analizar y criticar las decisiones, ideas y mecánicas que se tomaron en el primer proyecto. El segundo punto permite comprobar el nivel de dominio del lenguaje y de los conocimientos que se tienen sobre el desarrollo de un videojuego al tener que desarrollar una nueva versión con las mecánicas e ideas establecidas. El tercer y último objetivo sirve de nuevo para poner de nuevo a prueba la capacidad de análisis del desarrollador, debido a que tiene que comparar el resultado final de ambos proyectos y ver cómo ha mejorado con el lenguaje y con el desarrollo.

AGRADECIMIENTOS

A mis padres
Por haberme apoyado en todo momento

A mi hermano
Por ayudarme siempre que lo necesito

A mi tutora
Por guiarme durante este último tramo

A mi pareja
Por apoyarme siempre de forma incondicional

CITAS

“Cada época tiene su forma de contar historias, y el videojuego es una gran parte de nuestra cultura. Puedes ignorar los videojuegos o aceptarlos y empaparte de su gran calidad artística. Algunas personas están cautivadas con los videojuegos de la misma forma que a otras personas les encanta el cine o el teatro.”

Andy Serkis

INDICE

INDICE DE FIGURAS

1. INTRODUCCIÓN

La industria de los videojuegos es hoy en día una de las más grandes, superando incluso a la del cine [1]. Es por eso por lo que cada día salen cientos de videojuegos, ya sean de grandes, medianas o pequeñas empresas, o hasta incluso de un grupo de gente que se junta para desarrollar y crear contenido en una de sus pasiones.

Ya sea por el interés económico o por crear contenido por una pasión, mucha gente se lanza a la industria del videojuego ya sea como desarrollador o como artista (2D o 3D). Una vez dentro de la industria ven que no es tan fácil llegar a lo más alto de forma rápida, sino que tienes que empezar por el escalafón más bajo y poco a poco ir subiendo e ir ganando experiencia.

Por todas estas razones nace la idea de este trabajo, en el cual se va a comparar cómo ha evolucionado el programador desarrollando el mismo videojuego.

El primer proyecto se desarrolló un año antes de la realización de este trabajo. El proyecto nace como trabajo de la asignatura “Fundamentos de los Videojuegos” del grado de Ingeniería Multimedia. En este momento no se tenía experiencia alguna en el desarrollo de un videojuego y la experiencia que se tenía con el lenguaje de programación C++ era muy escasa.

El segundo proyecto nace con la idea de comparar la experiencia del programador un año después. Durante el periodo de tiempo entre proyectos se ha cursado el itinerario de "Creación y entretenimiento digital” del grado de Ingeniería Multimedia, en el cual se debe desarrollar un videojuego en grupo. Durante el desarrollo de ese proyecto se aprendieron y mejorar muchos conceptos del lenguaje C++ y se obtuvo una gran experiencia en el desarrollo de un videojuego.

2. MARCO TEÓRICO

2.1. ¿QUE ES UN VIDEOJUEGO BEAT 'EM UP?

Un videojuego del género ***Beat 'em up*** (o *brawler*) es aquel en el que se destaca el combate cuerpo a cuerpo entre el protagonista y un gran número de enemigos que suelen venir en forma de oleadas u hordas [2].

Aunque parezca que puede encajar dentro del género de lucha, el género *Beat 'em up* tiene varios rasgos que lo hacen distintivo y característico.

- Los videojuegos *Beat 'em up* se desarrollan sobre un largo escenario en el cual el jugador va avanzando mediante desplazamiento horizontal, mientras que los juegos de lucha se centran en un único escenario que entra en la pantalla.
- Los videojuegos *Beat 'em up* tienen un esquema de control sencillo con pocos movimientos a elegir, mientras que los juegos de lucha tienen gran variedad de ataques y combos para realizar.

Este tipo de videojuegos está pensado para ser jugado en modo cooperativo (de 2 a 4 o 6) más que en modo individual. En el modo cooperativo cada jugador debe de elegir un personaje distinto (normalmente no se puede repetir) y hacen equipos para sobrevivir al mayor número de oleadas posible. Al avanzar cierto número de rondas o al terminar el nivel los jugadores deben de enfrentarse a un poderoso jefe final.

Además del género *Beat 'em up* existen dos subgéneros del que este es padre:

- ***Hack and Slash***: comparten todas las características con su padre, pero estos están más centrados a la lucha cuerpo a cuerpo con armas blancas, de ahí su nombre "cortar y tajar".
- ***Shoot 'em up***: comparten todas las características con su padre, pero estos están más centrados a la lucha con armas de fuego.

2.2. HISTORIA DE LOS VIDEOJUEGOS BEAT 'EM UP

En los inicios, el género era mucho más simple, pero con el paso del tiempo fue evolucionando. Uno de los primeros juegos más conocidos es ***Kung-Fu Master*** (1984). No obstante, uno de los primeros juegos con profundidad de movimiento es ***Renegade*** (1986). *Renegade* incorpora 2 características que sentarán las bases del género.

- El personaje no se mueve solo a izquierda y derecha, sino que también puede moverse arriba y abajo.
- Incorpora un jefe único al final de cada nivel.

Ilustración 1 - Imágenes in-game de Kung-Fu Master (izquierda) y Renegade (derecha)

La edad de oro de los videojuegos *Beat 'em up* llegaría en 1987 con el lanzamiento de ***Double Dragon***. Los niveles se fueron alargando cada vez más y al llegar al final de este el jugador debería de enfrentarse a un jefe final (estas fueron las bases que *Renegade* construyó).

Con cientos de videojuegos de este estilo en el mercado, el género de los *Beat 'em up* se mantuvo como líder de mercado, pero poco a poco comenzó su decadencia. En 1991 con la llegada de ***Street Fighter II*** se producía el cambio de tendencia y los Beat 'em up empezaron a caer.

Ilustración 2 - Imágenes in-game de Double Dragon (izquierda) y Street Fighters II (derecha)

2.3. VIDEOJUEGOS BEAT 'EM UP (2D) ACTUALES

Actualmente existen juegos de este género con gráficos 2D o 2.5D (mayormente creados por estudios indie), pero su cantidad es muy escasa y difícilmente logran destacar frente a los juegos con gráficos 3D de las grandes compañías. Vamos a poner algunos ejemplos de ellos:

- **Hero Siege**: Juego *Hack 'n' Slash* con toques del género *Roguelike*[1] y *RPG*[2] con gráficos 2D en el cual el jugador tiene que sobrevivir a hordas de enemigos. Al derrotar a los enemigos el jugador consigue experiencia para subir de nivel y mejorar a su personaje. Además, al derrotar enemigos estos pueden soltar piezas de equipo para mejorar aún más al jugador. Posibilidad de jugar solo y en modo cooperativo (local u online) hasta 4 jugadores [5].

Ilustración 3 - Imagen in-game del videojuego Hero Siege

- **Final Exam**: Juego *Beat 'em up* con toques del género de plataformas y *RPG* con gráficos 2.5D en el cual el jugador debe de explorar y encontrar la salida mientras lucha con hordas de enemigos. Al derrotar a los enemigos el jugador consigue experiencia para subir de nivel y mejorar a su personaje. Además, al derrotar enemigos estos pueden soltar piezas de equipo para mejorar aún más al jugador. Posibilidad de jugar solo y en modo cooperativo (local u online) hasta 4 jugadores [6].

[1] Género en el cual el jugador debe explorar una mazmorra (o entorno similar) la cual se crea de forma procedimental [3].
[2] Género en el cual el jugador controla las acciones de un personaje inmerso en un mundo detallado [4].

Ilustración 4 - Imagen in-game del videojuego Final Exam

- **Lost Castle**: Juego *Beat 'em up* con toques del género *Roguelike* con gráficos 2D en el cual el jugador debe de superar las 4 fases de una mazmorra que se genera de forma aleatoria en cada partida. Si el jugador muere debe de volver a empezar desde 0. Al derrotar enemigos el jugador puede obtener piezas de equipo nuevo y fragmentos de alma que una vez el jugador muera pueden ser utilizadas para mejorar al personaje. Posibilidad de jugar solo y en modo cooperativo (local u online) hasta 4 jugadores [7].

Ilustración 5 - Imagen in-game del videojuego Lost Castle

2.4. VIDEOJUEGOS BEAT ‘EM UP (3D) ACTUALES

Actualmente, con la potencia que tienen las consolas y los ordenadores la tendencia de los videojuegos ha cambiado y se desarrollan proyectos de grandes dimensiones con grandes gráficos, ambientación e historia. Generalmente, estos proyectos ambiciosos son creados por grandes estudios que tienen ya un hueco en el mercado. Vamos a poner algunos ejemplos:

- **God of War (serie):** Juego *Hack and Slash* mezclado con los géneros de Acción y Aventura en el cual encarnamos a Kratos durante sus aventuras. Al ir avanzando por la historia Kratos ira consiguiendo nuevas armas y poderes para poder derrotar a los enemigos [8].

Ilustración 6 - Imagen in-game del videojuego God of War

- **NieR: Automata**: Juego *Hack and Slash* mezclado con los géneros Acción y Aventura en el cual encarnamos al androide de combate 2B durante sus aventuras [9].

Ilustración 7 - Imagen in-game del videojuego NieR: Automata

- **Devil May Cry (serie):** Juego *Hack and Slash* mezclado con los géneros de Acción y Aventura en el cual encarnamos a Dante durante sus aventuras. El juego consiste en ir superando una serie de misiones y conseguir la máxima nota en ellas, mientras vamos consiguiente mejores armas [10].

Ilustración 8 - Imagen in-game del videojuego Devil May Cry 5

3. METODOLOGÍA

La metodología utilizada para realizar este proyecto se denomina **Kanban**.

La metodología Kanban consiste en un panel en el cual tenemos tantas columnas como fases queramos o necesitemos en nuestro proyecto (Ejemplo: Por hacer, en proceso, en diseño, terminado, etc.).

Las columnas deberán ser rellenadas con tarjetas (normalmente *post-it* de diversos colores para una fácil y rápida interpretación del estado del proyecto), en las cuales se escribe el nombre de la tarea a realizar y una duración estimada de la misma.

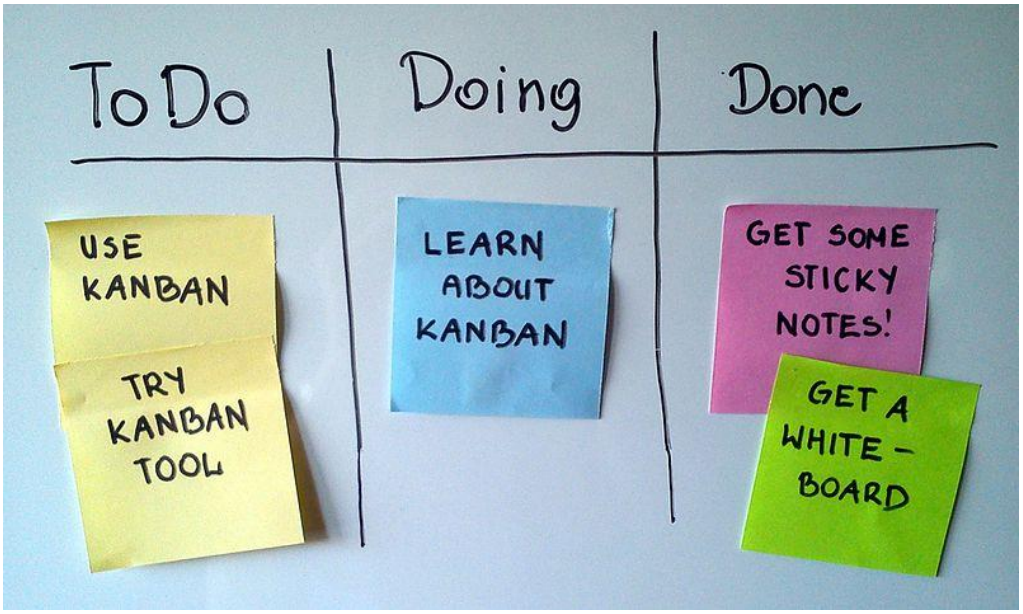

Ilustración 9 - Ejemplo de un panel de la metodología Kanban

Antes de empezar con el desarrollo y teniendo claro que es lo que tenemos que hacer para terminar nuestro producto, tenemos que empezar por rellenar todas las tarjetas necesarias, ordenándolas por orden de prioridad (si fuera necesario) y colocándolas en la columna de "Por hacer". Cuando se empiece con el desarrollo. Lo primero que tenemos que hacer si no tenemos ninguna tarea asignada es seleccionar una tarjeta y ponerla en la columna de "En proceso". Una vez tengamos la tarea terminada, ponemos la tarjeta en la columna de "Terminado". Mientras queden tarjetas en la columna "Por hacer" deberemos de repetir todo el proceso hasta que se acaben las tareas [11].

Si durante el desarrollo nos encontramos que una tarea es demasiado grande se puede subdividir esa tarea en otras más pequeñas añadiendo nuevas tarjetas a la columna de "Por hacer". De esta forma conseguiremos agilizar el flujo de trabajo y conseguir que el producto tenga una calidad superior.

Para poner en marcha esta metodología de desarrollo se ha decidido utilizar **Trello**, que es una aplicación web que nos permite crear tantas columnas como queramos y añadir tarjetas a esas columnas.

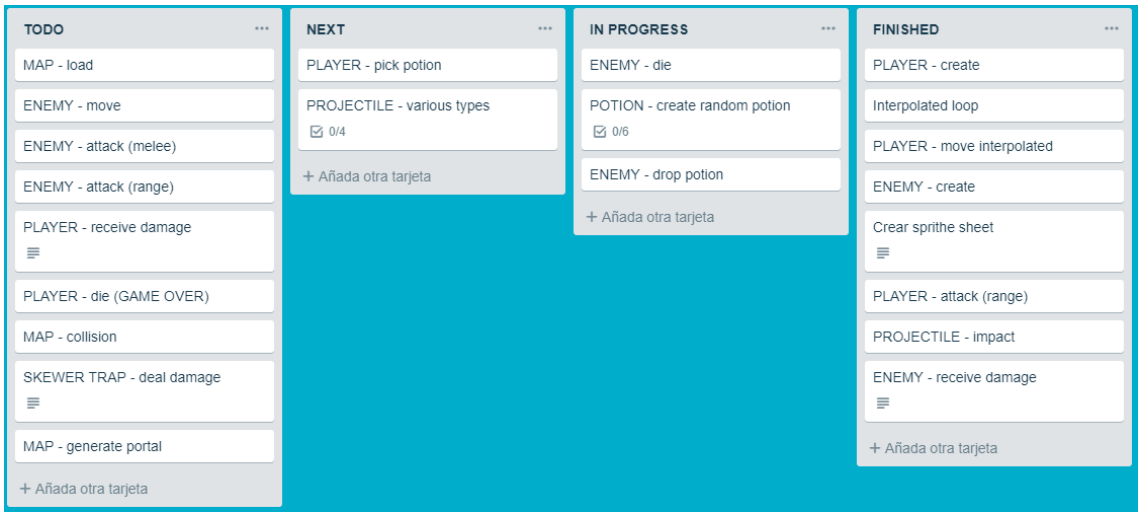

Ilustración 10 - Ejemplo de Trello a mitad de desarrollo

4. HERAMIENTAS

Para poder realizar este proyecto se han utilizado diversos tipos herramientas, cada una especializada en un campo en concreto.

4.1. DE GESTIÓN

Las herramientas de gestión son aquellas que nos ayudan durante el desarrollo, ya sea mediante la organización o creando copias de seguridad por si en algún momento tenemos que retroceder debido a algún error o cambio en la estructura. Las herramientas de este tipo utilizadas son:

Trello: Trello es una aplicación web que nos permite organizar información en forma de tarjetas las cuales podemos mover con total libertad entre las múltiples columnas que nos permite crear. Trello ha sido utilizado principalmente para crear el orden de desarrollo mediante la metodología Kanban.

Toggl: Toggl es una aplicación web que nos permite llevar un registro de las horas que hemos empleado para la realización del proyecto. Para utilizarla basta con rellenar un campo con el nombre de la tarea y a continuación darle al botón de "Play" y el tiempo empezará a contarse.

GitKraken: GitKraken es una interfaz gráfica para *Git*[3], lo que nos permite realizar en cuestión de unos pocos clics copias de seguridad de nuestro código a nuestra cuenta de *GitHub*[4].

4.2. DE DESARROLLO

Las herramientas de desarrollo son aquellas que nos han permitido dar forma a nuestro proyecto, ya sea mediante un editor de texto o mediante librerías para añadir funcionalidad. Las herramientas de este tipo que hemos utilizado han sido:

Visual Studio Commuity 2017: Visual Studio ha sido el *IDE* [5]utilizado para desarrollar el proyecto. Se ha decidido utilizar este *IDE* ya que nos facilita mucho el trabajo con atajos de teclado, compilador integrado, autocompletar, texto predictivo, creación de clases con dos clics y muchas otras funcionalidades.

SFML: "*Simple and Fast Multimedia Library*", es la librería que ha hecho posible el desarrollo de este proyecto junto con el lenguaje C++. Esto es gracias a que nos permite crear una ventana en la cual nosotros podemos pintar los *assets*[6] de nuestro proyecto. También nos permite recoger los *inputs* tanto de teclado y ratón como los de un joystick para así poder interaccionar con los *assets*.

---

[3] Software de control de versiones, pensado en la eficiencia y confiabilidad del mantenimiento de versiones de aplicaciones [12].

[4] Plataforma de desarrollo colaborativo que utiliza el sistema de control de versiones Git [13].

[5] Entorno de Desarrollo Integrado. Aplicación que proporciona servicios integrales para facilitar el desarrollo de software [14].

[6] Elementos de un videojuego como el arte, efectos sonoros, música, texto y cualquier otro elemento que se le presente al usuario.

TinyXML-2: TinyXML-2 es una librería que nos permite leer de forma rápida y sencilla el contenido de un fichero XML. Esta librería la usaremos para leer el fichero XML que nos generará Tiled con toda la información del mapa del juego.

4.3. GRÁFICAS

Las herramientas gráficas son aquellas que nos han permitido crear todos los *assets* visuales necesarios para poder realizar el proyecto, ya sean bocetos, imágenes sueltas (*sprites*[7]) o "*Sprite sheets*[8]" (conjunto de imágenes o animaciones).

Photoshop: Photoshop es un programa de creación y edición de imágenes. Este programa nos permite crear los múltiples *assets* necesarios para nuestro proyecto. Más en concreto me ha permitido crear los *sprite sheets* necesarios para luego convertirlos en texturas y así poder aplicarlos a los *sprites* dentro del videojuegos.

Tiled: Tiled nos permite crear de forma rápida y simple mapas en 2D con múltiples capas para luego utilizarlos en nuestros proyectos. Para poder usar Tiled primero tenemos que crear un *tileset*[9]. Una vez tenemos creado el *tileset* lo cargamos en Tiled y con unos cuantos clics podremos crear nuestro mapa.

[7] Mapa de bits que se dibuja en pantalla [15].
[8] Conjunto de sprites y/o animaciones en una única imagen para mejorar el rendimiento.
[9] Similar a un sprite sheet pero formado por tiles (pequeñas imágenes cuadradas).

5. PRIMEROS PASOS

Antes de empezar con el desarrollo y una vez establecidas las herramientas principales para poder llevar a cabo el proyecto se empezó con la búsqueda y recuperación de las ideas y mecánicas principales que formaron el juego original. En esta búsqueda también se recuperaron los *assets* del juego (principalmente se recuperaron los *sprites* de los personajes, los enemigos y los *tiles* que forman el mapa), ya que este trabajo está enfocado a la parte de la programación de código y no a la parte del arte.

5.1. BUSCAR Y ANALIZAR CARACTERISTICAS Y MECANICAS

Una vez recuperada la versión final del proyecto original se empezó por estudiar el documento que resumió el desarrollo y el estado final del primer proyecto. Gracias a este documento se pudieron rescatar de forma rápida todas las características y mecánicas básicas que contenía el juego original, implementadas o no.

Una vez identificadas todas las características y mecánicas se procede con el análisis de todas y cada una de ellas para ver si encajan de forma correcta con el nuevo proyecto o necesitan ser descartadas, o si por el contrario necesitan una mejora para poder encajar mejor en el nuevo proyecto.

5.1.1. BUSQUEDA DE CARACTERISTICAS

Las características encontradas y extraídas del juego original fueron las siguientes:

- Modos de juego
  - Modo individual.
  - Modo cooperativo local (no implementado).
- Personajes
  - Personaje jugable azul.
  - Personaje jugable verde.
  - Personaje jugable amarillo (no implementado).
  - 1 enemigo.
  - 1 enemigo jefe.
- Mapa
  - Mapa finito predefinido.
  - Minijuego predefinido.

5.1.2. ANALISIS DE CARACTERISTICAS

En cuanto a los **modos de juego** no se ha encontrado ningún contra por lo que ambos modos de juego seguirán presentes en la nueva versión del juego.

Respecto a los **personajes** se van a mantener los 3 personajes jugables. Con respecto a los personajes enemigos encontramos que el número incrementa de ser únicamente 1 a 3. Se conserva el enemigo original que ataca cuerpo a cuerpo y se añaden dos nuevos:

- Enemigo a distancia: lanza una bola viscosa que hace daño al jugador si impacta con él.

- Enemigo con carga: se inmoviliza y carga un ataque rabioso. Cuando termina de cargar el ataque este sale corriendo en línea recta hacia el jugador, si impacta contra el jugador este recibe daño.

Respecto al **mapa** se descarta el mapa con el minijuego que aparece cada cierto tiempo porque rompe un poco con el frenesí del juego y también porque no se puede recompensar de forma correcta al jugador.

5.1.3. BUSQUEDA DE MECANICAS

Las mecánicas encontradas fueron las siguientes:

- Modo de juego infinito basado en oleadas.
    - Cada oleada aumenta la cantidad de enemigos y resistencia.
    - Cada 5 oleadas aparece el jefe enemigo.
- Personajes jugables.
    - Libertad de movimiento 2D (arriba y abajo, izquierda y derecha).
    - Atacan a los enemigos con diversos ataques.
        - Ataque básico infinito.
        - Habilidades especiales con tiempo de reutilización (no implementado).
    - Pierden/ganan vida.
    - Ganan experiencia y suben de nivel (no implementado).
- Personajes enemigos.
    - Persiguen a los jugadores.
    - Atacan a los jugadores cuerpo a cuerpo.
    - El jefe ataca cuerpo a cuerpo y a distancia.
    - Al morir el personaje jugable que lo derrota recibe experiencia (no implementado).
    - Al morir pueden soltar una bola de vida.
- Mapa.
    - Contiene trampas de pinchos que al ser pisadas infligen daño a los personajes.
    - Contiene elementos que bloquean el paso a los personajes y enemigos.
    - Genera un portal con un minijuego cada cierto tiempo.
        - Al superar este minijuego el jugador recibe vida.

5.1.4. ANALISIS DE MECANICAS

Respecto a las mecánicas del **modo de juego** no se ha encontrado ninguna contra, por lo cual el sistema de oleada seguirá siendo utilizado.

Respecto a los **personajes jugables** el movimiento seguirá siendo en 2D con las mismas direcciones. En los ataques seguirá habiendo 1 ataque básico, pero ahora los ataques especiales aumentan a 3 para dar un poco más de variedad al rol del jugador. También se añadirán puntos de magia al jugador que será el limitante para poder lanzar más o menos habilidades especiales. A todo esto, se le suma el nivel del personaje, lo que hará que el jugador se haga más fuerte a medida que derrota más enemigos.

Los **personajes enemigos** seguirán actuando de la misma forma, es decir, perseguirán a los jugadores hasta derrotarlos.

Se cambian las **bolas de vida** que pueden soltar los enemigos por **pociones**. Existirán 6 tipos de pociones: vida, magia, daño, armadura, velocidad de ataque y velocidad de movimiento. Las dos primeras siempre tendrán efectos positivos. El resto pueden tener efectos positivos o negativos.

En cuanto al **mapa** como en las características se ha eliminado el mapa con el minijuego se elimina también la mecánica relacionada con este tipo de mapa.

5.2. BUSCAR Y CREAR ASSETS

Una vez teniendo todos los *assets* localizados se empezó por seleccionar cuáles iban a estar presentes en esta nueva versión. A su vez se crearon algunos nuevos que no estaban en la versión original del juego y otros fueron ligeramente modificados para que encajaran mejor en esta nueva versión.

5.2.1. SPRITE SHEETS

Con todos los *sprites* localizados y actualizados se procedió con la creación de las plantillas. Se creó un *sprite sheet* que contenía los personajes jugables, los enemigos y las pociones. También, se creó un *tileset* que contenía todos los *tiles* que componen el mapa de juego.

Ilustración 11 - Sprite sheet de entidades

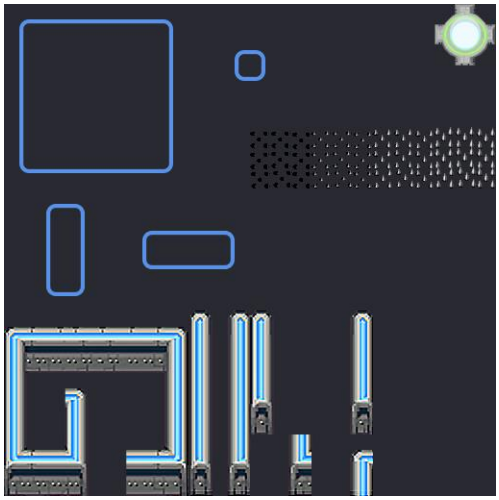

Ilustración 12 - sprite sheet de mapa

5.2.2. PALETA DE COLORES

Tras observar como quedaron las plantillas, se continuó creando una paleta de colores con los principales colores que componen la mayoría de los elementos para así poder ser usados en el menú y en la interfaz del juego.

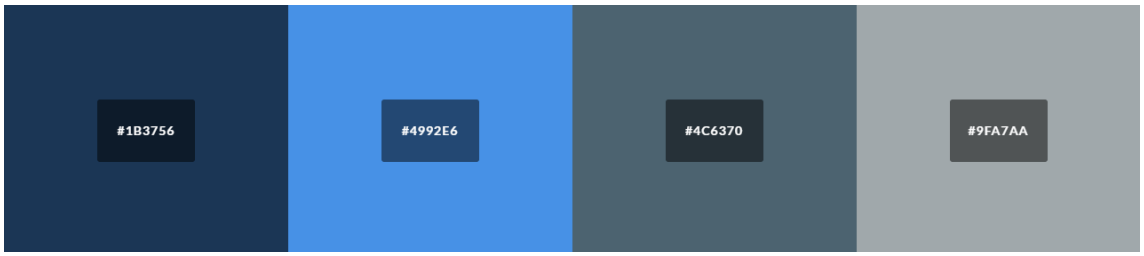

Ilustración 13 - Paleta de colores

5.2.3. PANTALLAS

Una vez creada la paleta de colores se procedió con el bocetado de las pantallas del menú y con el bocetado de las interfaces para los diferentes modos de juego. Se decidió crear en este momento los bocetos para no parar el desarrollo del proyecto una vez se alcancé el punto en el cual se deba de implementar el menú o la interfaz.

Se bocetan unas pantallas simples con el menor número de elementos posibles y con un *feedback visual*[10] eficiente para facilitar su entendimiento por parte del usuario.

5.2.3.1. PANTALLA PRINCIPAL

Esta será la pantalla que se verá nada más se ejecute el proyecto. En esta pantalla podemos observar cómo se encuentran todos los colores de la paleta que hemos mostrado anteriormente. En la parte superior en el centro observamos se encuentra el nombre del videojuego "ALPHAS". En la parte izquierda encontramos 3 botones con las posibles acciones que podemos realizar. Si la opción que seleccionamos es la primera "JUGAR" aparecerá la pantalla mostrada en la Ilustración 14. Si seleccionamos "CONTROLES" aparecerá la pantalla mostrada en la Ilustración 16. Si por el contrario seleccionamos "SALIR" el juego se cerrará.

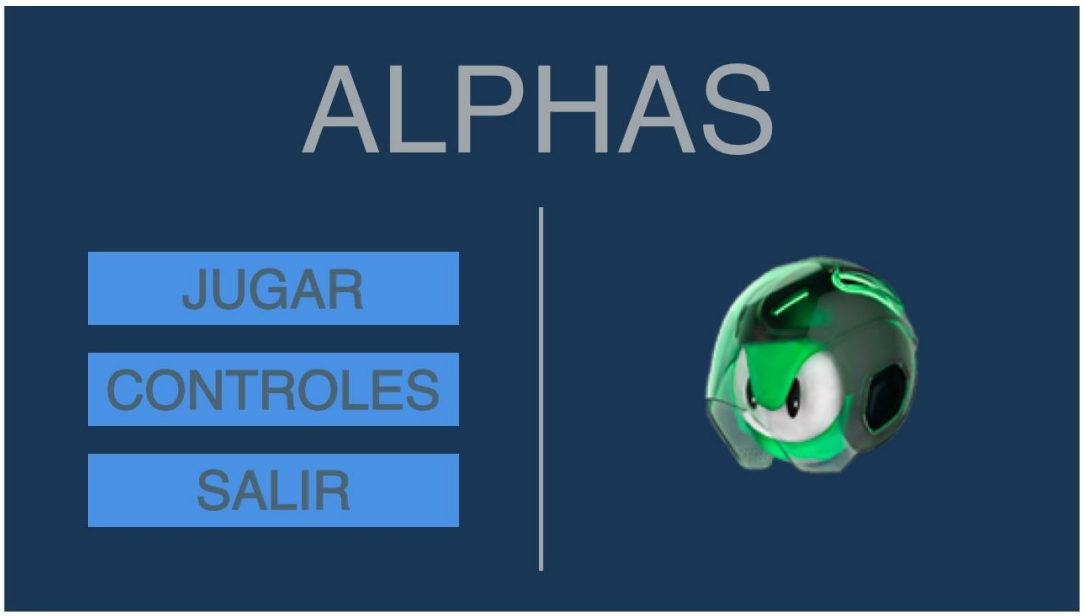

Ilustración 14 - Boceto de la pantalla principal

5.2.3.2. PANTALLA SELECCION MODO DE JUEGO

En la pantalla de **selección de modo de juego** podemos seleccionar el modo de juego que queremos, ya sea jugar solo o con un amigo de forma cooperativa. Una vez seleccionada la opción deseada pasaremos a la pantalla de selección de personaje (solo) si hemos seleccionado "SOLO" o a la pantalla de selección de personaje (cooperativo) si hemos seleccionado "COOPERATIVO".

[10] Información que el juego le ofrece al usuario.

Ilustración 15 - Boceto de la pantalla selección modo de juego

5.2.3.3. PANTALLA CONTROLES

En la pantalla de **controles** podremos observar cómo se controlan los personajes dependiendo de si jugamos con teclado o con un joystick.

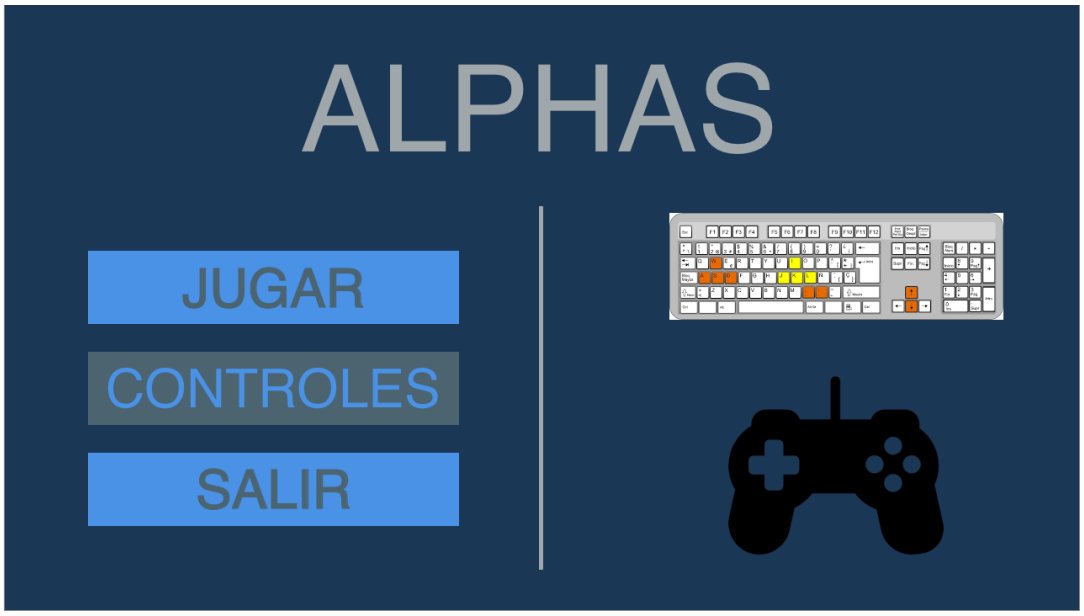

Ilustración 16 - Boceto de la pantalla opciones

5.2.3.4. MENU SELECCION PERSONAJE (SOLO)

La pantalla de **selección de personaje (solo)** consta en el inferior izquierdo de los 3 personajes disponibles para ser usados. El jugador podrá desplazarse sobre ellos y seleccionar el que más les guste. Mientras va desplazándose sobre ellos va cambiando la imagen en grande del seleccionado junto con la información de los ataques.

Una vez el jugador seleccione un personaje el botón de “JUGAR” estará desbloqueado.

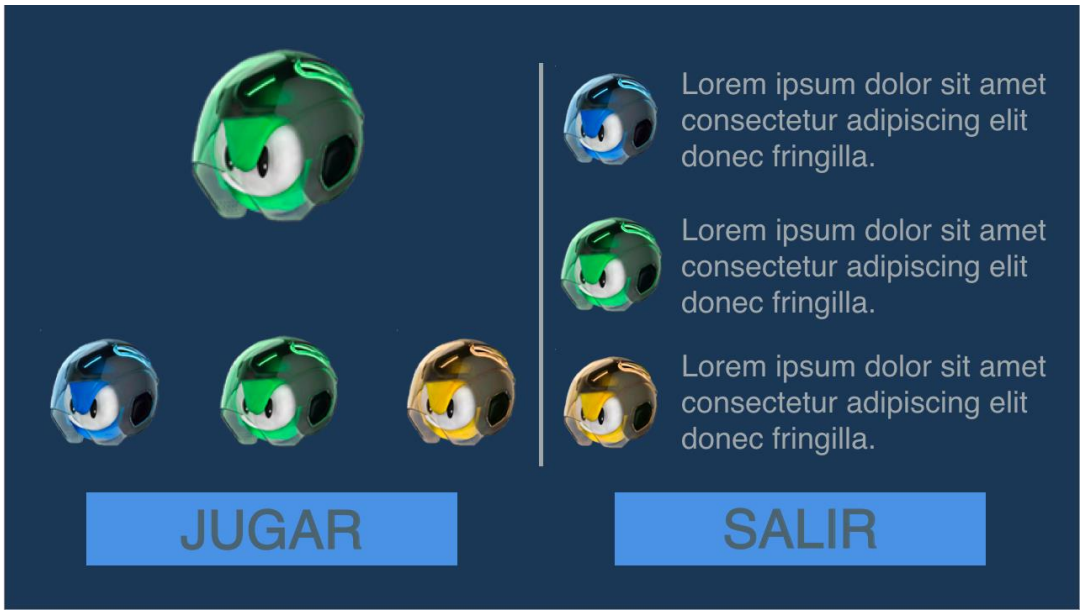

Ilustración 17 - Boceto de la pantalla selección personaje (solo)

5.2.3.5. PANTALLA SELECCION PERSONAJE (COOPERATIVO)

La pantalla de **selección de personaje (cooperativo)** es igual a la pantalla de selección de personaje (solo) con la pequeña diferencia de que ahora no podemos observar las habilidades de los personajes, sino que en su lugar aparecerá la imagen del personaje que ha seleccionado el segundo jugador.

Una vez los dos jugadores seleccionen un personaje distinto al del otro el botón de "JUGAR" estará desbloqueado.

Ilustración 18 - Boceto de la pantalla selección personaje (cooperativo)

5.2.3.6. PANTALLA PAUSA

La pantalla de **pausa** solo aparecerá dentro de una partida cuando el jugador presione la tecla "ESC". Al entrar en la pantalla de pausa el juego se detiene mientras esta pantalla esté activada.

Mientras la pantalla este activada al jugador le aparecerán dos opciones. “CONTINUAR” cerrará la pantalla de pausa y se retomará la partida que estaba en curso. “SALIR” terminará la partida y hará que el jugador vuelva a la pantalla principal.

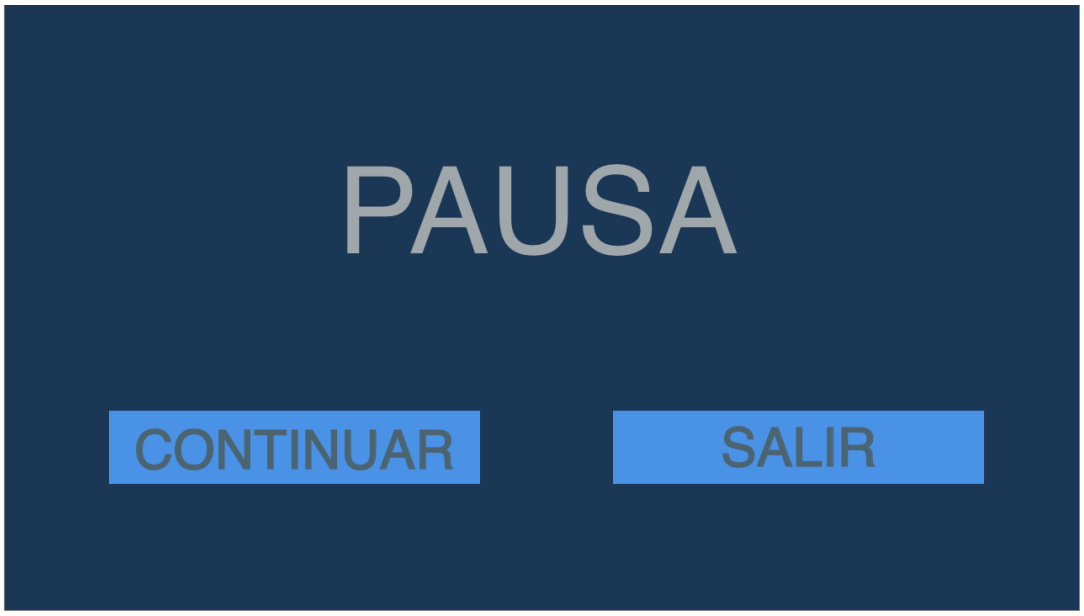

Ilustración 19 - Boceto de la pantalla de pausa

5.2.3.7. PANTALLA GAME OVER

La pantalla de **Game Over** aparece cuando el jugador es derrotado. En esta pantalla el jugador tiene dos opciones: volver a jugar con el mismo personaje otra partida o salir a la pantalla principal.

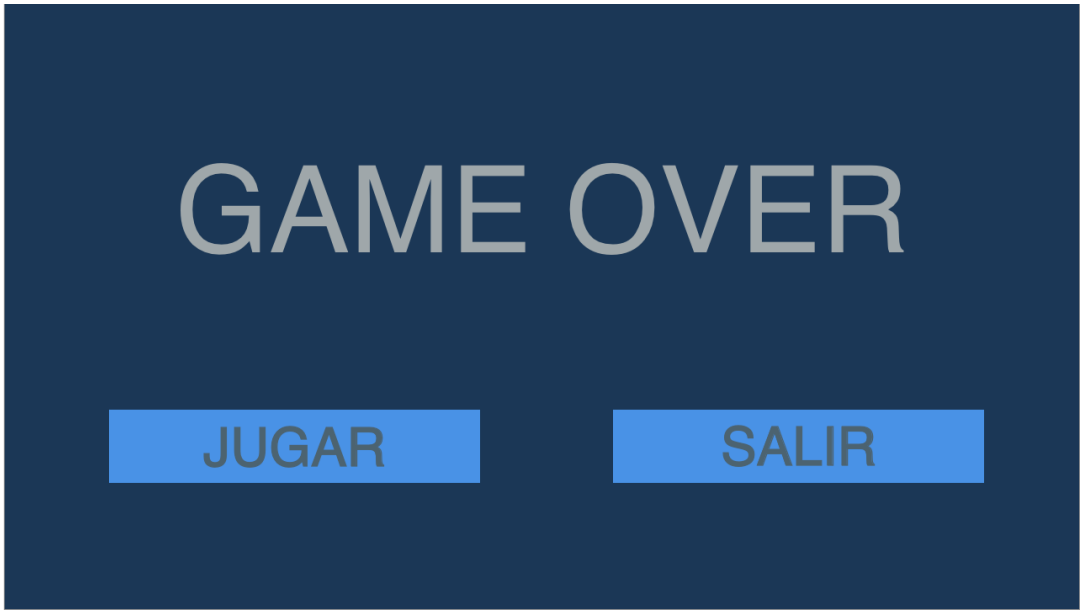

Ilustración 20 - boceto de la pantalla de Game Over

5.2.4. INTERFACES

Una vez bocetados las pantallas del menú procedemos con el bocetado de las interfaces del juego. Como tenemos dos modos de juego (un solo jugador o dos jugadores) tendremos que crear dos interfaces con la máxima similitud posible para que los jugadores no se confundan independientemente del modo de juego en el que estén.

En la interfaz de 1 jugador observamos como en la parte inferior en el centro tenemos toda la información relevante sobre el personaje que está siendo utilizado. Encontramos:

- Tres barras: una roja (vida), otra azul (magia) y una amarilla (experiencia y nivel).
- Entre medias de las dos primeras barras encontramos los iconos de las 3 habilidades especiales del personaje junto con su respectivo *feedback visual*.
    - Si la habilidad esta con colores vivos es que podemos usar la habilidad.
    - Si la habilidad tiene el color apagado es que tenemos que esperar un tiempo para poder volver usar esa habilidad.
    - Si la habilidad tiene un tono azul significa que la habilidad está cargada pero que no tenemos la magia necesaria para lanzarla.
- Debajo de las barras encontramos 4 iconos que hacen referencia a los puntos de daño, armadura, velocidad de ataque y velocidad de movimiento.

Por otro lado, en el centro de la parte superior encontramos en que número de oleada estamos y cuantos enemigos quedan para que termine la oleada actual.

Ilustración 21 - Boceto de la interfaz en modo 1 jugador

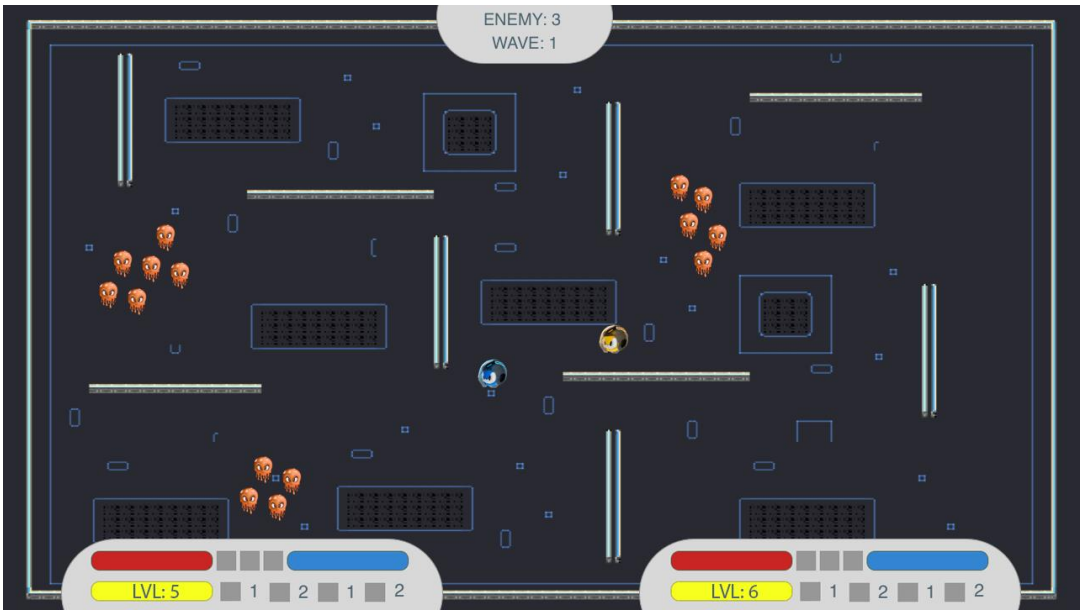

Ilustración 22 - Boceto de la interfaz en modo 2 jugadores

La interfaz de dos jugadores es exactamente idéntica a la de 1 jugador, pero ahora en vez de tener abajo en el centro la información de un jugador tenemos la información de los dos jugadores, una a la izquierda (jugador 1) y otra a la derecha (jugador 2).

6. DESARROLLO NUEVA VERSION

Teniendo ya clara la idea junto con las características principales y las mecánicas básicas se procede al desarrollo de la nueva versión de ALPHAS.

A continuación, se procederá a explicar cómo se han ido creando las múltiples entidades que componen la nueva versión de ALPHAS, cuáles son sus características y cuál es su finalidad dentro del videojuego.

6.1. CLASE ENGINE MANAGER

La clase **Engine Manager** es una clase que implementa el patrón de diseño ***Facade***[11] junto con un ***Singleton***[12].

Esta clase nos permite desacoplar por completo el motor que usará y moverá nuestro juego (SFML) de tal forma que todas las llamadas que se realizan a SFML se invocan desde esta clase. Esta estructura nos facilitaría el cambio de motor en un futuro si fuera necesario, ya que solamente deberíamos de modificar esta clase y no el resto de los ficheros.

6.2. CLASE ENTIDAD (ENTITY)

La clase entidad es la clase base y clave para el desarrollo del juego. Esta clase contiene los parámetros y métodos base para modificar la gran mayoría de los elementos que formarán el videojuego.

```
protected:
    EngineManager*  m_engineManager;
    Entities    m_entity;

    const char* m_texturePath;

    int     m_spriteID;
    int     m_spriteSheetRow;

    double  m_posX;
    double  m_posY;
    double  m_lastPosX;
    double  m_lastPosY;

    float   m_width;
    float   m_height;
};
```

Ilustración 23 - Variables de la clase Entidad

Como se puede observar en la imagen, la clase Entidad contiene el identificador del *sprite*, el tamaño del *sprite*, la posición actual y la última posición de la entidad a la que haga referencia. Además, contiene una variable que nos permite saber qué tipo de entidad es en concreto.

En la parte de los métodos observamos que la mayoría son getters[13] de las variables nombradas anteriormente. No es necesario utilizar setters[14] debido a que los valores son modificados en las clases derivadas gracias a que las variables las hemos puesto en la parte protegida de la clase, lo que permite que sus derivadas puedan acceder a estas variables.

Los únicos métodos que no son getters son *update()* y *draw()*. Estos dos métodos son virtuales debido a que no es necesario que sean implementados en la clase base, sino que deben ser implementados en las clases derivadas.

[11] Patrón de diseño que busca minimizar la comunicación y dependencia entre clases. Este patrón proporciona una interfaz simple para un subsistema complejo (la librería SFML en este caso) [16].
[12] Patrón de diseño que permite restringir la creación de objetos de una clase a un único objeto [17].
[13] Métodos de una clase que devuelven el valor de una variable.
[14] Métodos de una clase con el cual se le da valor a una variable.

```
public:
    Entity(const char* p_path, Entities p_entity);
    Entity(const char* p_path, int p_textureLeft, int p_textureTop, int p_textureWidth, int p_textureHeight, float p_posX, float p_posY);
    virtual ~Entity();

    virtual void update();
    virtual void update(double p_time, double p_deltaTime);
    virtual void draw();

    double  getPositionX()  { return m_posX; };
    double  getPositionY()  { return m_posY; };
    float   getWidth()      { return m_width; };
    float   getHeight()     { return m_height; };
    int     getSpriteID()   { return m_spriteID; };

    const char* getTexturePath()    { return m_texturePath; };
    Entities    getEntity()         { return m_entity; };
```

Ilustración 24 - Métodos de la clase Entidad

6.3. CLASE JUGADOR (PLAYER)

La clase Jugador es derivada de la clase Entidad. Esta clase no es la clase final, sino que es una clase intermedia antes de llegar al verdadero personaje que manejará el jugador.

```
protected:
    double  m_deltaTime;
    double  m_time;

    bool    m_alive;

    float   m_maxHealth;
    float   m_health;
    float   m_maxMana;
    float   m_mana;
    float   m_baseVelocity;
    float   m_velocity;
    float   m_baseDamage;
    float   m_damage;
    float   m_baseArmor;
    float   m_armor;
    float   m_baseAtackSpeed;
    float   m_atackSpeed;

    bool    m_pasiveActive;

    Entities    m_bulletColor;

    Direction   m_faceDirection;
```

Ilustración 25 - Variables clase Jugador

Como los 3 personajes jugables que hay en ALPHAS tienen las mismas mecánicas y variables esta clase nos sirve para implementar las mecánicas y no tener que repetir el mismo código 3 veces, 1 por cada personaje.

Como podemos observar, el jugador tiene los parámetros de vida, magia, velocidad de movimiento, daño, armadura y velocidad de ataque.

Junto a las variables de vida y magia encontramos las mismas, pero con un "*max*" delante. Esto quiere decir que no pueden sobrepasar ese valor, es decir, es el valor máximo al que pueden llegar.

Junto a las variables de velocidad de movimiento, daño, armadura y velocidad de ataque encontramos las mismas, pero con un 'base' delante. Esto quiere decir que es el valor base al que deben de volver cuando se acabe el efecto de una poción o una habilidad.

Estas variables serán luego diferentes dependiendo del personaje final que elijamos, lo que nos permitirá que cada personaje se adapte mejor a un rol (agresivo, defensivo o supervivencia).

En la parte de los métodos, podemos observar que al igual que en la clase base Entidad solo existen métodos getter debido a que los valores se actualizan en esta clase o en la derivada que será ya el propio jugador.

Además de estos métodos encontramos los métodos para recibir daño ya sea desde un enemigo o desde una trampa. Se ha creado esta distinción ya que si se recibe daño de un enemigo entra en juego el factor de la armadura, pero si es una trampa la que inflige daño el factor armadura no juega ningún papel, por lo que recibe todo el daño que inflige esa trampa.

Debajo de estos métodos encontramos los correspondientes de cada poción. Como el efecto es el mismo independientemente de que personaje la coja se han creado en esta clase y no en las derivadas.

```
public:
    Player(float p_posX, float p_posY, const char* p_path, Entities p_playerEntity);
    virtual ~Player();

    bool    getAlive()      { return m_alive; };
    float   getHealth()     { return m_health; };
    float   getMaxHealth()  { return m_maxHealth; };
    float   getMana()       { return m_mana; };
    float   getMaxMana()    { return m_maxMana; };
    float   getVelocity()   { return m_velocity; };
    float   getDamage()     { return m_damage; };
    float   getArmor()      { return m_armor; };
    float   getAtackSpeed() { return m_atackSpeed; };

    void    receiveDamage(float p_damage);
    void    receiveDamage(float p_damage, Projectile* p_projectile);
    void    receiveTrapDamage(float p_damage);

    void    increaseHealth(float p_health);
    void    increaseMana(float p_mana);
    void    increaseSpeed(float p_duration, float p_speedIncrease);
    void    increaseDamage(float p_duration, float p_damageIncrease);
    void    increaseArmor(float p_duration, float p_armorIncrease);
    void    increaseAtackSpeed(float p_duration, float p_atackSpeedIncrease);

    void    move();
    void    moveBackwards();
    void    rangeAtack();
    bool    enoughMana(float p_mana);
    virtual void hability1() = 0;
    virtual void hability2() = 0;
    virtual void hability3() = 0;
```

Ilustración 26 - Métodos de la clase Jugador

Debajo de los métodos de las pociones encontramos el método para poder mover a los jugadores por el mapa.

Los últimos métodos que encontramos son los relacionados con los ataques. El primero es el ataque básico (*rangeAtack()*) el cual lanza una aro en línea recta en la dirección a la que está apuntando la cabeza del personaje. Debajo encontramos el método *enoughMana()* el cual sirve para saber si tenemos suficiente magia para poder lanzar una habilidad especial. Los 3 últimos son los correspondientes a las 3 habilidades que tendrá cada jugador y por lo tanto son virtuales puras, lo que significan que deben de ser implementadas en la clase derivada sí o sí.

6.3.1. CLASE JUGADOR FINAL (PLAYER BLUE/GREEN/YELLOW)

Esta clase es derivada de la clase Jugador. La clase Jugar final solo contiene como métodos los 3 métodos virtuales puros de la clase Jugador ya que deben de implementarse sí o sí en esta clase ya que es la derivada. Estos 3 métodos están formados por diversos tipos de aros, dependiendo del rol del personaje que sea.

```
PlayerBlue::PlayerBlue(float p_posX, float p_posY, const char * p_path) : Player(p_posX, p_posY, p_path, Entities::PLAYER_BLUE)
{
    m_bulletColor = Entities::BULLET_BLUE;

    m_baseDamage     = 10.f;
    m_damage         = m_baseDamage;
    m_maxHealth      = 100.f;
    m_health         = m_maxHealth;
    m_maxMana        = 225.f;
    m_mana           = m_maxMana;
    m_baseArmor      = 3.f;
    m_armor          = m_baseArmor;
    m_baseVelocity   = 100.f;
    m_velocity       = m_baseVelocity;
    m_baseAtackSpeed = 0.25f;
    m_atackSpeed     = m_baseAtackSpeed;
}
```

Ilustración 27 - Constructor Jugador Azul (Jugador Final)

Como se puede observar en la imagen anterior el constructor del Jugador final (el azul en este caso) implemente todas las variables que se nombraron en la clase Jugador. De esta forma podemos crear jugadores con características diferentes y modificarlas de forma fácil.

6.4. CLASE ENEMIGO (ENEMY)

La clase Enemigo es derivada de la clase Entidad. Al igual que Jugador esta es una clase intermedia que nos permitirá configurar los enemigos antes de darles su verdadero rol.

Como podemos observar, los enemigos tienen al igual que los jugadores vida, daño, velocidad de ataque y velocidad de movimiento, pero no tiene armadura.

```
protected:
    bool    m_stunned;
    bool    m_sticky;

    float   m_maxHealth;
    float   m_health;
    float   m_damage;
    float   m_atackSpeed;
    float   m_velocity;
    float   m_baseVelocity;

    Player* m_objectivePlayer;
    double  m_distanceToObjective;
    float   m_directionMoveX;
    float   m_directionMoveY;
```

Ilustración 28 - Variables clase Enemigo

Las dos primeras variables de tipo *bool* son efectos de estados que les pueden causar los jugadores con sus diferentes ataques especiales.

Para terminar, encontramos que cada enemigo tiene una variable para guardar cual es el jugador objetivo al que tiene que perseguir dependiendo de varios factores. Debajo encontramos la distancia que los separa (se utiliza para saber si el jugador está en rango para poder inicializar un ataque). Las ultimas variables corresponden con la dirección de movimiento que deben de seguir para poder llegar hasta el personaje objetivo.

En la parte correspondiente a los métodos de la clase Enemigo encontramos dos métodos que son parecido a los de la clase Jugador, estos métodos son los de recibir daño. La diferencia es que el enemigo guarda cual ha sido el último proyectil que le ha causado daño para que no le haga daño durante una pequeña cantidad de tiempo. Al mismo tiempo que el enemigo comprueba si ha recibido daño o no, este comprueba también si está muerto o no.

A continuación, encontramos los métodos que nos permiten mover al enemigo por el mapa. Para mover al enemigo lo primero que hacemos es identificar cual es el jugador que está más cerca o cual es el que menos vida tiene para poder ir a derrotarlo. Una vez el jugador objetivo es identificado el enemigo lo perseguirá.

Para terminar, encontramos los métodos correspondientes con los ataques. Estos métodos son virtuales puros ya que cada tipo de enemigo tiene uno diferente con unos requisitos de activación y actualización distintos. Es por estas razones que deben ser implementados en la clase derivada sí o sí.

```
public:
    Enemy(float p_posX, float p_posY, const char* p_path, Entities p_entity);
    virtual ~Enemy();

    void receiveDamage(float p_damage, Projectile* p_projectile);
    void receiveTrapDamage(float p_damage);
    bool isDead() { return (m_dead && !m_sticky); };

    void checkObjective();
    void move();
    virtual void moveBackwards();
    double getDistanceToObjective() { return m_distanceToObjective; };

    virtual void atack() = 0;
    virtual void updateAtack() = 0;
```

Ilustración 29 - Métodos de la clase Enemigo

6.4.1. CLASE ENEMIGO GUERRERO (ENEMY WARRIOR)

Clase derivada de Enemigo. Es el enemigo más básico de todo el juego y más equilibrado. Es el enemigo que menor daño hace, pero se compensa con mayor velocidad de movimiento y resistencia. Su función es perseguir al jugador objetivo y atacarle cuerpo a cuerpo.

6.4.2. CLASE ENEMIGO A DISTANCIA (ENEMY RANGER)

Clase derivada de Enemigo. La característica de este enemigo es que su ataque es a distancia a gran velocidad y que tiende a mantener la distancia con el jugador objetivo. Este tipo de enemigo cuenta con una daño de ataque superior al del guerrero, pero tiene la menor resistencia de todos los tipos de enemigos.

6.4.3. CLASE ENEMIGO CON CARGA (ENEMY CHARGER)

Clase derivada de Enemigo. La característica de este enemigo es que cuando detecta que tiene al jugador en rango se queda inmóvil y vulnerable, pero empieza a cargar un ataque lleno de ira. Una vez cargado el ataque se lanza en línea recta sobre su objetivo. Este tipo de enemigo cuenta con el mayor daño de todos los enemigos y su resistencia es normal.

6.5. CLASE PROYECTIL (PROJECTILE)

La clase Proyectil es derivada de la clase Entidad. Al igual que las anteriores es una clase intermedia que nos permitirá configurar los valores de los múltiples proyectiles existentes.

Todos los proyectiles tienen una velocidad de movimiento, lo que influye en lo rápido que avanzan. También cuentan con un tiempo de vida, lo que significa que cuando su tiempo se utilización se termina el proyectil es destruido.

```
protected:
    int     m_velocity;
    float   m_moveX;
    float   m_moveY;

    float   m_lifeTime;
    float   m_dieTime;
    float   m_damage;

    bool    m_readyToDelete;

    bool    m_crossEnemy;
    bool    m_makeDamage;

    Entities        m_entityOwner;
```

Ilustración 30 - Variables de la clase Proyectil

Pero la velocidad y el tiempo de vida no son los únicos factores que afectan a la vida útil del proyectil. También cuentan con 2 propiedades muy importantes y que nos permiten crear muchas combinaciones de proyectiles:

- **Atravesar enemigos**(*m_crossEnemy*): esta propiedad nos indica si cuando el proyectil impacta en un enemigo este debe de desaparecer o debe de continuar su trayectoria y desaparecer una vez su tiempo de vida termina.
- **Hacer daño**(*m_makeDamage*): esta propiedad nos permite saber si un proyectil tiene que hacer daño o no. Si no hace daño es porque provoca un efecto de estado como puede ser causar *stun* (inmovilizar) al enemigo o aumentar la velocidad de ataque o movimiento del jugador.

```
public:
    Projectile(const char* p_texturePath, Entities p_ent, Direction p_dir, float p_playerPosX, float p_playerPosY, float p_damage);
    virtual ~Projectile();

    bool    getReadyToDelete() { return m_readyToDelete; };

    void    update(double p_time, double p_deltaTime);
    void    update();

    void    draw();

    void    setLifeTime(float p_lifeTime);
    void    setReadyToDelete(bool p_readyToDelete) { m_readyToDelete = p_readyToDelete; };
```

Ilustración 31 - Métodos de la clase Proyectil

En la parte de los métodos observamos que son métodos muy genéricos: saber cuándo tiene que ser eliminado, actualiza y dibujar. Esto es así ya que dependiendo de qué tipo de proyectil sea tendrá una funcionalidad u otra, entonces su función de *update()* será diferente.

6.5.1. CLASE PROYECTIL RECTO (PROJECTILE STRIGHT)

Clase derivada de Proyectil. Este tipo de proyectil es el más básico de todos los existentes debido a que es el ataque básico de los jugadores y el ataque a distancia de los enemigos. Dependiendo de quien lo lance tiene unas características u otras, pero en ambos casos si impacta contra otra entidad objetivo este es destruido causando daño o no.

- **Jugador**: Si es lanzado por un jugador el proyectil es lanzado sobre el eje X o sobre el eje Y, dependiendo de donde está apuntando el jugador.
- **Enemigo**: Si es lanzado por un enemigo el proyectil se mueve en línea recta hacia el jugador objetivo, es decir, no sigue una línea recta sobre un eje como el del jugador. Esto es así para aumentar ligeramente la dificultad del juego.

Existe una variedad de este ataque para los jugadores la cual en vez de causar daño causa *stun* a todos los enemigos con los que impacta.

6.5.2. CLASE PROYECTIL RECTO CON GIRO (PROYECTILE STRAIGHT SPIN)

Clase derivada de Proyectil. Este tipo de proyectil es lanzado en línea recta. Si no impacta en ningún enemigo es destruido, pero si por el contrario impacta en algún enemigo este empieza a girar durante un periodo de tiempo con centro de giro en el lugar en el cual ha impactado. Mientras el aro está girando causa daño a todos los enemigos con los que colisiona.

6.5.3. CLASE PROYECTIL RECTO PEGAJOSO (PROYECTILE STRAIGHT STICKY)

Clase derivada de Proyectil. Este tipo de proyectil es igual al anterior, pero con una ligera diferencia, ya que causa diferentes efectos dependiendo sobre qué entidad colisiona:

- **Jugador**: si impacta sobre un jugador este ve aumentada su armadura de forma considerable durante un cierto tiempo. Además, durante ese tiempo el aro gira a su alrededor causando daño a todos los enemigos con los que colisiona.
- **Enemigo**: si impacta sobre un enemigo este recibe un gran daño. Además, durante un periodo de tiempo el aro se pega en él y empieza a girar a su alrededor causando daño a los demás enemigos con los que colisiona. El portador del aro es inmortal mientras tiene el aro pegado.

6.5.4. PROYECTIL CON GIRO (PROYECTILE SPIN)

Clase derivada de Proyectil. Este proyectil al ser lanzado empieza a girar alrededor del jugador, causando daño a los enemigos con los que colisiona y aumentando de forma temporal la armadura del jugador que lo porta. El aro gira alrededor del jugador en todo momento, incluso si él se mueve.

Existe otra versión de este ataque que en vez de causar daño aumenta considerablemente la velocidad de movimiento del portador.

6.5.5. PROYECTILE CON GIRO FIJO (PROYECTILE SPIN FIXED)

Clase derivada de Proyectil. Este proyectil al ser lanzado empieza a girar con centro fijo en la posición en la cual el jugador lo activa. El proyectil empieza a girar durante un periodo de tiempo, durante el cual si un aliado entra dentro del área que dibuja el proyectil este regenera salud lentamente, pero si es un enemigo el que entra este recibe daño.

6.5.6. PROYECTIL EN CONO (PROYECTILE CONUS)

Clase derivada de Proyectil. Este proyectil al ser lanzado lanza 3 proyectiles en forma de cono invertido desde la posición del jugador. Los 3 proyectiles atraviesan a los enemigos causando daño a todos con los que colisionan.

Existe otra versión de este proyectil el cual en vez de hacer daño causa *stun* a los enemigos.

6.6. CLASE POCION (POTIONS)

La clase Poción es derivada de la clase Entidad. Al igual que las anteriores es una clase intermedia que nos permitirá configurar los valores de las diversas pociones.

```
protected:
    bool m_efectUsed;

    float   m_effectIncrease;
    float   m_effectDuration;
```

Como podemos observar la clase Poción es muy pequeña y solo contiene unos pocos datos los cuales nos dicen si el efecto de la poción ha sido usado y durante cuánto tiempo el efecto de esa poción estará activado.

Ilustración 32 - Variables de la clase Poción

Las pociones funcionan de tal manera que una vez muere un enemigo existe una probabilidad de que estos suelten una poción de entre 6 tipos distintos. La poción se mantendrá de forma indefinida en el mapa hasta que un jugador la recoja y automáticamente reciba su efecto.

```
public:
    Potion(const char* p_path, float p_posX, float p_posY, PotionType p_potionType);
    virtual ~Potion();

    virtual void setEffect(Player* p_player) = 0;
    bool getEffectUsed() { return m_efectUsed; };
```

Ilustración 33 - Métodos de la clase Poción

6.6.1. CLASE POCION VIDA (POTION HEALTH)

Clase derivada de Poción. Esta poción tiene un 3% de probabilidad de generarse al derrotar a un enemigo. Al ser recogida el jugador verá aumentados sus puntos de salud restantes de forma considerable.

6.6.2. CLASE POCION MAGIA (POTION MANA)

Clase derivada de Poción. Esta poción tiene un 3% de probabilidad de generarse al derrotar a un enemigo. Al ser recogida el jugador verá aumentados sus puntos de magia restantes de forma considerable.

6.6.3. CLASE POCION DAÑO (POTION DAMAGE)

Clase derivada de Poción. Esta poción tiene un 1% de probabilidad de generarse al derrotar a un enemigo. Al ser recogida el jugador verá como sus puntos de ataque aumentan o disminuyen durante un cierto periodo de tiempo.

6.6.4. CLASE POCION ARMADURA (POTION ARMOR)

Clase derivada de Poción. Esta poción tiene un 1% de probabilidad de generarse al derrotar a un enemigo. Al ser recogida el jugador verá como sus puntos de armadura aumentan o disminuyen durante un cierto periodo de tiempo.

6.6.5. CLASE POCION VELOCIDAD MOVIMIENTO (POTION SPEED)

Clase derivada de Poción. Esta poción tiene un 1% de probabilidad de generarse al derrotar a un enemigo. Al ser recogida el jugador verá como su velocidad de movimiento aumenta o disminuye durante un cierto periodo de tiempo.

6.6.6. CLASE POCION VELOCIDAD DE ATAQUE (POTION ATACK SPEED)

Clase derivada de Poción. Esta poción tiene un 1% de probabilidad de generarse al derrotar a un enemigo. Al ser recogida el jugador verá como su velocidad de ataque aumenta o disminuye durante un cierto periodo de tiempo.

6.7. CLASE MAPA (SCENE MAP)

La clase Mapa nos permite crear un mapa que anteriormente hayamos creado en Tiled. El único requisito para que esta clase funcione correctamente es que cuando se cree el mapa en Tiled este tiene que tener el "Formato de la Capa de Patrones" en formato XML para que la librería TinyXML-2 pueda funcionar de forma correcta al leer el fichero del mapa.

Para poder crear un mapa debemos indicarle cual es el fichero del mapa y cuál es el fichero que contiene las texturas de los tiles del mapa.

```
private:
    EngineManager* m_engineManager;

    Tile**** m_mapMatrix4D;

    tinyxml2::XMLDocument*  m_mapDocument;
    tinyxml2::XMLElement*   m_rootElement;
    tinyxml2::XMLElement*   m_firstLayerElement;
    tinyxml2::XMLElement*   m_layerElement;
    tinyxml2::XMLElement*   m_tileElement;

    int m_width;
    int m_height;
    int m_tileWidth;
    int m_tileHeight;
    int m_totalLayers;
```

Ilustración 34 - Variables de la clase Mapa

Como podemos observar el mapa está formado por una matriz de 3 dimensiones de Tiles que son el número de capas que contiene el mapa y el ancho y alto de cada capa.

A continuación, tenemos varias variables que sirven para guardar elementos importantes que encontramos en el documento XML del mapa como el propio documento, elemento raíz o la primera capa.

Para terminar, encontramos variables que extraemos del fichero como el alto, ancho y capas totales del mapa para saber el tamaño que debe de tener la matriz. También encontramos el alto y ancho de los tiles para así poder crearlos con el tamaño adecuado.

```
public:
    SceneMap(const char* p_urlXML, const char* p_urlTexture);
    ~SceneMap();

    void draw();

    int getWidth() { return (m_width*m_tileWidth); };
    int getHeight() { return (m_height*m_tileHeight); };
```

Ilustración 35 - Métodos de la clase Mapa

En el apartado de los métodos además del constructor que explicaremos a continuación encontramos el método de dibujado y dos getters que nos indican el ancho y alto del mapa en pixeles.

Como se dijo unos párrafos arriba, para crear el mapa se necesita el ficho del mapa creado en Tiles y el fichero con la texturas del mapa. Nada más entrar en el constructor se comprueba que el fichero del mapa sea correcto si no, nos daría un error. Si todo es correcto se entra en un bucle en el cual se buscan cuantas capas existen dentro del documento para poder crear la matriz con las dimensiones correctas.

Una vez se tiene el número de capas se crea la matriz y acto seguido se empieza a recorrer línea a línea cada capa para ir rellenando la matriz. Las líneas solo contienen un dato que es el gid (identificador que les da Tiled para saber que tile son). Al recoger ese dato y saber cuántos líneas

hemos leído y el tamaño total del mapa, calculamos en qué posición está en el mapa y es entonces cuando creamos el tile y lo guardamos en la matriz.

6.8. CLASE TILE

La clase Tile es derivada de la clase Entidad. Al igual que las anteriores es una clase intermedia que nos permitirá configurar los valores de los diversos tiles siempre y cuando tengan un efecto, sino serán simples tiles.

```
protected:
    TileTypes    m_tileType;
    int m_gid;
```

Ilustración 36 - Variables de la clase Tile

Como podemos observar, los únicos datos que tienen en común todos los tiles son su propio identificador que les proporciona Tiled y el tipo de Tile que son.

En el apartado de los métodos encontramos que existen dos getters: uno para saber el *gid*[15] y otro para saber qué tipo de tile es.

```
public:
    Tile(const char* p_urlTexture, int p_textureLeft, int p_textureTop, int p_textureWidth, int p_textureHeight, float p_posX, float p_posY, int p_gid);
    virtual ~Tile();

    virtual void applyEffect(Entity* p_entity);
    virtual void update(double p_time, double p_deltaTime);

    int         getGID() { return m_gid; };
    TileTypes   getTileType() { return m_tileType; };
```

Ilustración 37 - Métodos de la clase Tile

Además, encontramos dos métodos virtuales que serán implementados en las clases derivadas de Tile, debido a que cada Tile tiene un efecto en concreto dependiendo de su tipo.

6.8.1. CLASE TILE BLOQUE (TILE BLOCK)

Clase derivada de Tile. Estos tiles están distribuidos por todo el mapa en forma de muros y también son lo que delimitan el mapa, lo que impide que los jugadores puedan salirse del mapa.

La funcionalidad de este Tile es que cuando un enemigo o un jugador colisiona con él, los enemigos o jugadores son parados y no pueden avanzar hacia esa dirección. Si por el contrario es un proyectil el que colisiona con un Tile de este tipo el proyectil es destruido.

6.8.2. CLASE TILE PINCHO (TILE SKEWER)

Clase derivada de Tile. Estos tiles están distribuidos por todo el mapa en grupos de varios pinchos.

La funcionalidad de este Tile varia con el tiempo.

- Si los pinchos están escondidos o a medio salir actúan como un tile normal.
- Si por el contrario los pinchos han salido en su totalidad causaran daño a toda entidad con la que colisione.

[15] Identificador que genera Tiled para identificar en qué posición del tileset se encuentra el tile representado.

6.9. MENU / PANTALLAS

Para la creación de los menús que fueron bocetados antes de empezar con el desarrollo se implementó una máquina de estados para conseguir una transición fluida entre las diferentes pantallas que componen el menú.

Para ello se ha utilizado una clase base Pantalla con un patrón de diseño de *Singleton* para que la información básica y común de las pantallas no tenga que ser repetida entre todas las pantallas del menú. Posteriormente se crearon las diversas clases derivando de Pantalla.

6.9.1. CLASE SCREEN MANAGER

Esta clase es la que nos permite poder hacer los cambios entre las diversas pantallas de forma sencilla. Al ser implementado con un patrón de diseño *State*[16], es necesario que le digamos en que pantalla nos encontramos.

```
protected:
    ScreenManager();

    EngineManager*  m_engineManager;

    Screen* m_currentScreen;
    Screen* m_overlayScreen;
```

Ilustración 38 - Variables de la clase Screen Manager

Como podemos observar, esta clase solo contiene la información de la pantalla en la que se encuentra actualmente y la de una pantalla superpuesta.

La pantalla actual es el estado general y la superpuesta es un estado que puede aparecer en diversas ocasiones y tiene un mayor peso que el estado general, pero lo necesita para poder existir.

En la parte de los métodos encontramos dos setter, uno para el estado general y otro para el estado superpuesto. Además, existe un método para saber si el estado superpuesto está abierto y otro para eliminarlo para cuando ya no lo necesitemos.

```
public:
    ~ScreenManager();
    static ScreenManager& p();

    void setCurrentScreen(Screen* p_newScreen) { m_currentScreen = p_newScreen; };
    void setOverlayScreen(Screen* p_newScreen) { m_overlayScreen = p_newScreen; };

    void deleteOverlayScreen();

    bool overlayOpened();

    void init();
    void update(double p_time, double p_deltaTime);
    void draw();
    void changeScreen(Screen* p_newScreen);
```

Ilustración 39 - Métodos de la clase Screen Manager

El método más importante de esta clase es el último, *changeState()*. Este método nos permite cambiar entre las diversas pantallas (estados) que existen en nuestro videojuego. Cuando

[16] Patrón de diseño que permite alterar el comportamiento de un objeto dependiendo de su estado interno [18].

invocamos este método le pasamos el estado al cual queremos cambiar y el solo gestiona la creación de este y el borrado del estado anterior.

6.9.2. PANTALLAS FINALES

A continuación, se van a mostrar el resultado final de las pantallas del menú una vez implementadas siguiendo lo más fielmente posible los bocetos que se realizaron antes de empezar con el desarrollo.

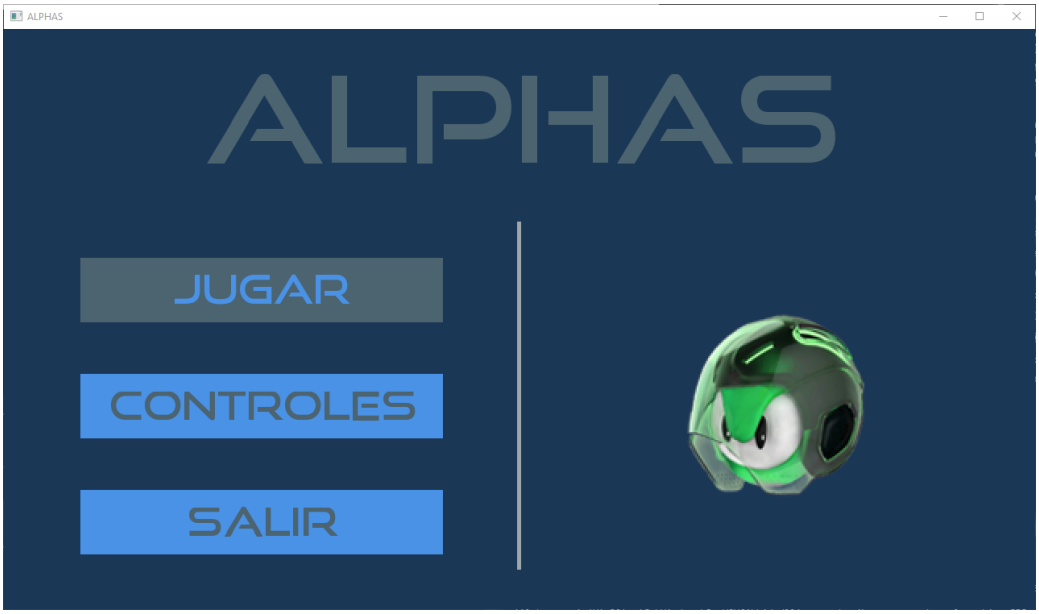

Ilustración 40 - Pantalla inicial del menú in-game

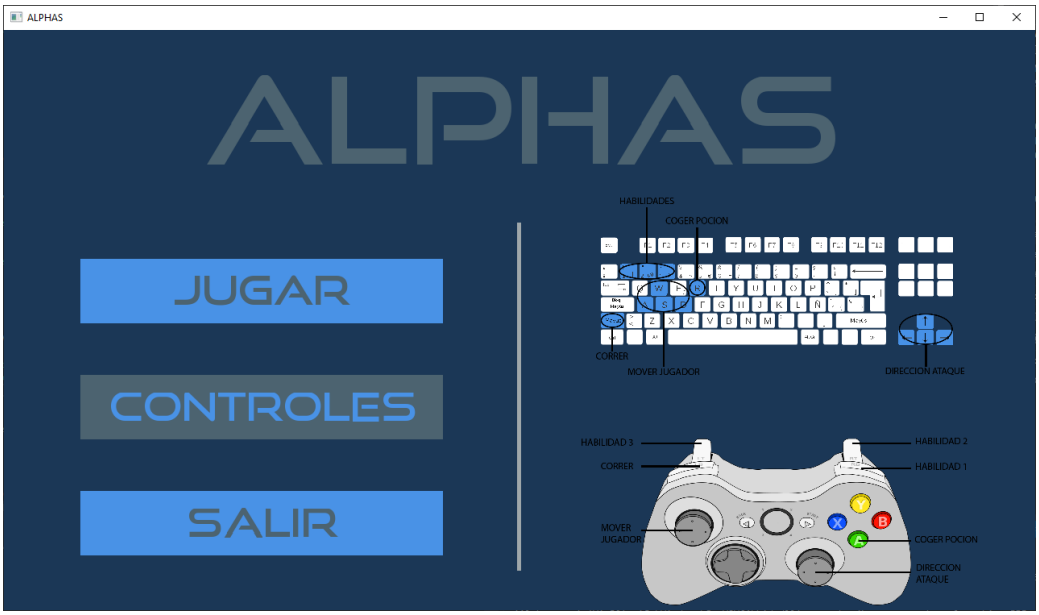

Ilustración 41 - Pantalla de controles in-game

Ilustración 42 - Pantalla de selección de modo de juego in-game

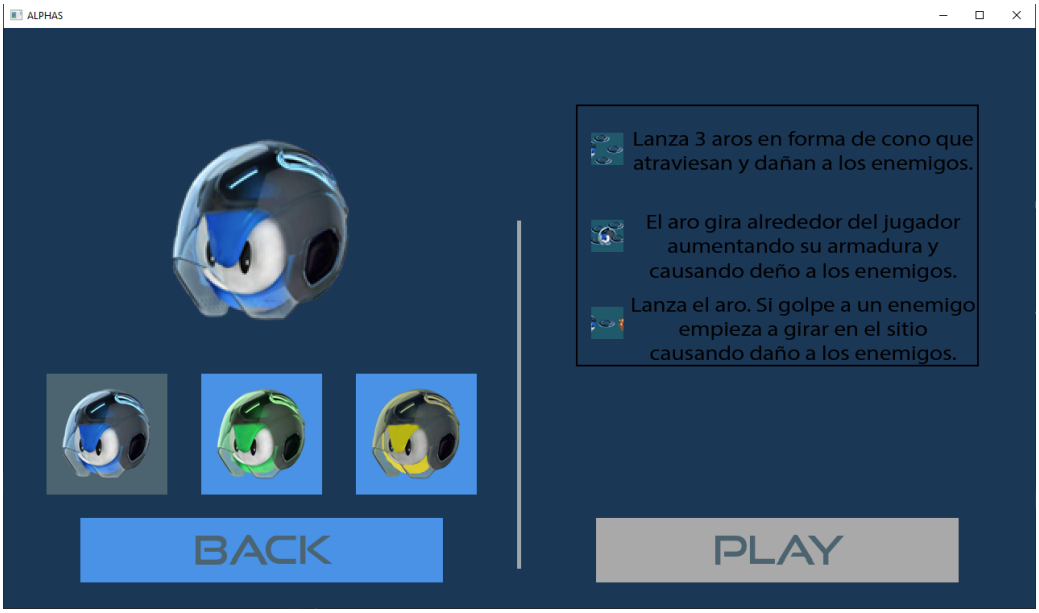

Ilustración 43 - Pantalla de selección de jugador (solo) in-game

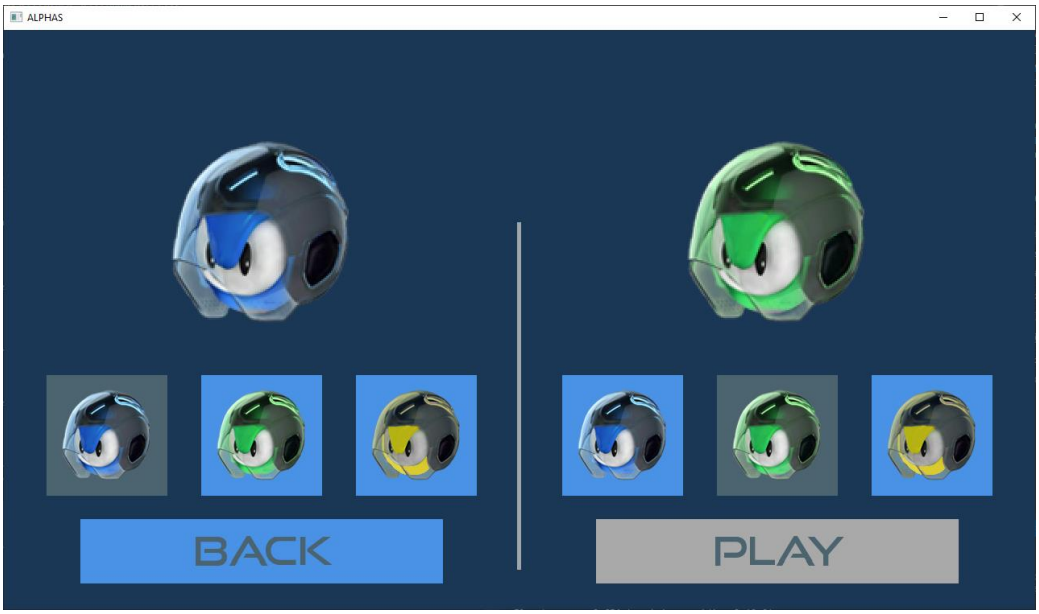

Ilustración 44 - Pantalla de selección de personaje (cooperativo) in-game

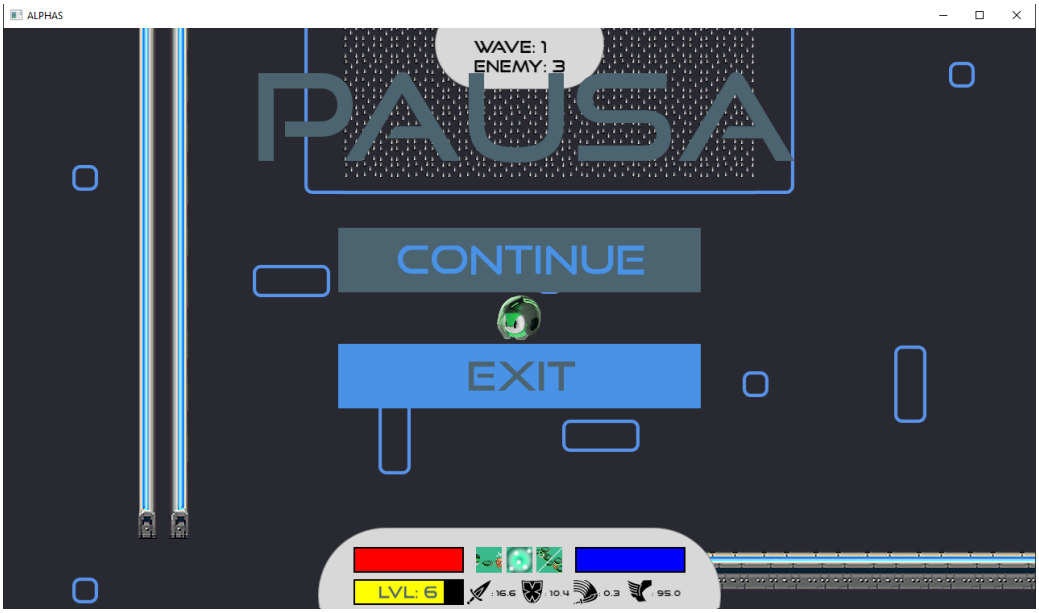

Ilustración 45 - Pantalla de pausa in-game

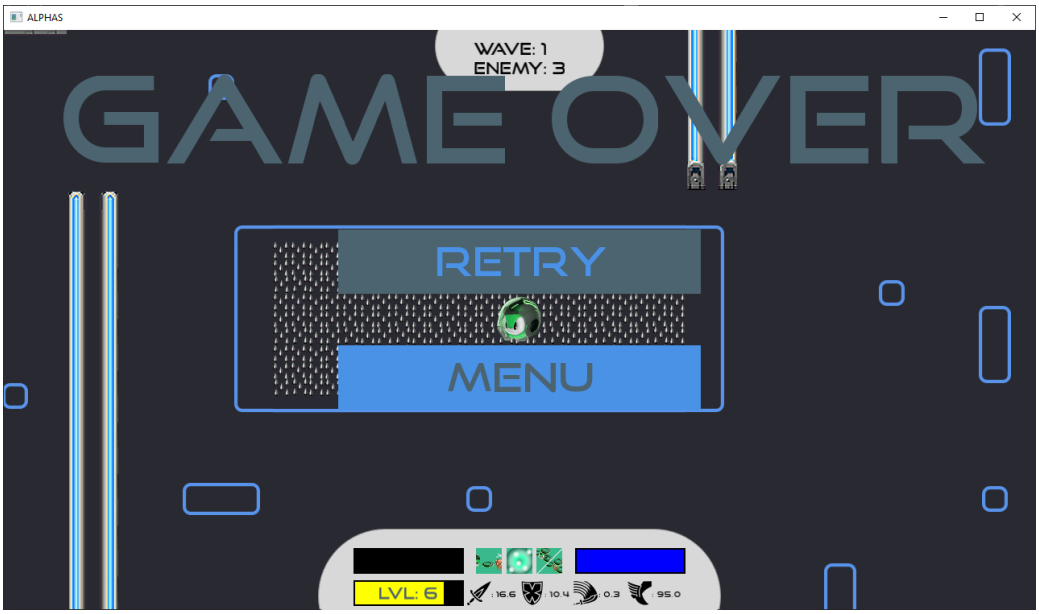

Ilustración 46 - Pantalla de Game Over in-game

6.10. INTERFAZ

A continuación, se mostrarán las dos interfaces una vez implementadas dentro del juego. Como se puede observar se ha seguido el boceto lo más fielmente posible.

Ilustración 47 - Interfaz de modo solo in-game

Ilustración 48 - Interfaz del modo cooperativo in-game

6.11. COLISIONES

Durante la explicación de las múltiples clases ha habido una palabra que se ha ido repitiendo en muchas de ellas, aunque más en unas que en otras, esta palabra es 'colisión'. Las colisiones son una parte fundamental de un videojuego ya que nos permiten que todas las entidades puedan interaccionar con el mundo o con el resto de las entidades.

En el caso de la nueva versión se ha implementado un sistema de colisiones simple debido a que la librería SFML nos facilita mucho el trabajo.

Al crear un *sprite* con SFML a este se le asigna automáticamente un rectángulo en el cual está contenido, independientemente de la forma que tenga el *sprite*. Este rectángulo es el que permite posicionar el *sprite* en el mundo ya que contiene la información del punto más a la izquierda y arriba, y la altura y la anchura del cuadrado. Con estos 4 datos calcularía los 3 puntos restantes.

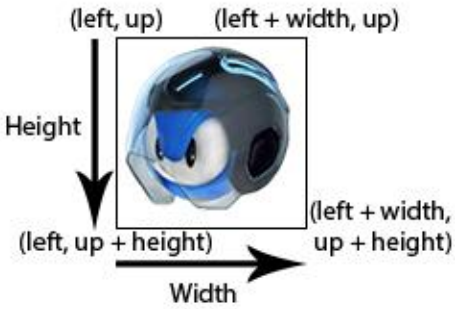

Ilustración 49 - Ejemplo del rectángulo limitante

Como cada entidad tiene uno de estos rectángulos, debemos de ver si entre dos entidades sus rectángulos intersectan o no, para saber con certeza si hay colisión o no. Este es el método más común y sencillo, pero como se puede observar en la imagen de ejemplo puede quedar espacio entre el rectángulo y el Sprite, lo que haría que visualmente los sprites no colisionen, pero si lo hagan sus rectángulos. Para evitar ese error, se optó por comprobar la colisión de una forma diferente.

La nueva forma de comprobar una colisión era viendo si el *sprite* 1 contiene el punto central del *sprite* 2 con el que se está comprobando la colisión. Si el *sprite* 1 contiene dicho punto del *sprite* 2 entonces existe una colisión. De esta forma minimizamos el impacto visual y las colisiones se ven más reales.

Cuando se manejan unas pocas entidades, el cálculo de colisiones no supone ningún impedimento, pero cuando el número de entidades crece se convierte en un problema debido a que es una operación costosa comprobar todas las entidades con todas. Es por este problema por el que se debe optimizar el sistema de colisiones con algún tipo de estructura.

6.11.1. HASH GRID

La estructura utilizada para optimizar el sistema de colisiones es un *Hash Grid*. Pero ¿en qué consiste esta estructura y como nos ayuda en el cálculo de las colisiones?

La estructura *Hash Grid* divide el mapa del juego en cuadriculas de una dimensión determinada. Una vez tenemos el mapa dividido en cuadriculas debemos de insertar todas las entidades que nos interesen en una cuadricula que irá determinada por su posición.

De esta forma, al tener todas las entidades ordenadas en cuadriculas sabemos cuáles son las entidades con las que tienes más posibilidades de colisionar, por lo que solo comprobamos si colisiona con esas y no con las que están en una cuadricula más lejana.

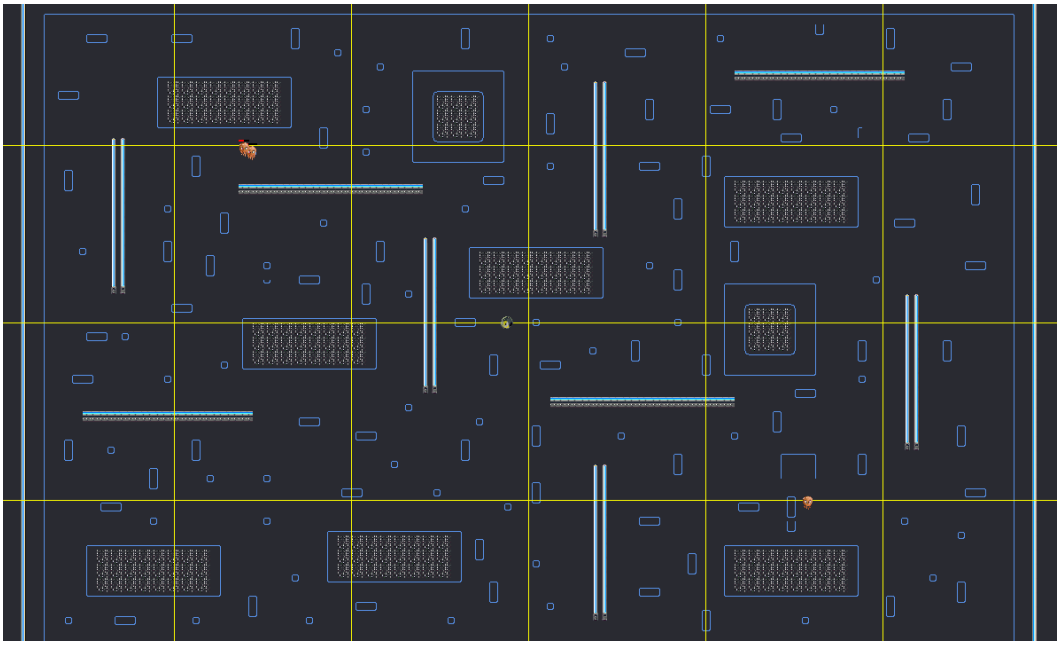

Ilustración 50 - Ejemplo de Hash Grid

En la imagen anterior se puede observar como el mapa está dividido en cuadriculas de igual tamaño. En cada cuadricula se insertarían como se dijo anteriormente las entidades que pueden causar colisión que en este caso serían los jugadores, los enemigos y algunos tiles como las barreras o las trampas de pinchos, el resto de los tiles no se insertarían ya que no tienen ninguna función que no sea más que la visual.

6.12. SISTEMA DE OLEADAS Y NIVEL

ALPHAS tiene implementado un sistema de oleadas en el cual cada oleada de enemigos es más grande que la anterior y los enemigos son también más fuertes y resistentes. Además, cada 5 rondas aparece el jefe de los enemigos para ponerle las cosas difíciles al jugador.

Al mismo tiempo que el jugador derrota enemigos y va pasando de ronda, el jugador va obteniendo experiencia por cada enemigo derrotado. A más fuerte el enemigo, más cantidad de

experiencia obtiene el jugador. Con la experiencia obtenida el jugador va subiendo de nivel y con cada nivel ganado aumentan sus estadísticas y cada ciertos niveles desbloquea una nueva habilidad para aumentar las posibilidades de supervivencia.

6.12.1. PROGRESION ENEMIGOS

En la siguiente imagen podemos observar cual va a ser la progresión de los enemigos conforme se van incrementando las oleadas.

Para empezar, observamos que se ha hecho un cálculo con 20 oleadas para tener una idea general de la progresión (el resto de las oleadas seguiría el mismo formato). Al lado de las oleadas podemos ver cuántos enemigos existirían en cada oleada.

Base enemigo	4												
Factor	0,8	1,1	1,2	1,1	1,4	1,1	1,05	1,25	1,25	1,15	2,5	20	5
		WARRIOR			RANGER			CHARGER			BOSS		
OLEADA	ENEMIGOS TOTALES	DAÑO	VIDA	EXPERIENCIA	DAÑO	VIDA	EXPERIENCIA	DAÑO	VIDA	EXPERIENCIA	DAÑO	VIDA	EXPERIENCIA
1	3	5	150	10	10	75	8	20	100	12			
2	6	7,2	152,4	12,2	12,8	77,2	10,1	22,5	102,5	14,3			
3	10	8,3	153,6	13,3	14,2	78,3	11,15	23,75	103,75	15,45			
4	13	9,4	154,8	14,4	15,6	79,4	12,2	25	105	16,6			
5	16	10,5	156	15,5	17	80,5	13,25	26,25	106,25	17,75	42,5	600	125
6	19	11,6	157,2	16,6	18,4	81,6	14,3	27,5	107,5	18,9			
7	22	12,7	158,4	17,7	19,8	82,7	15,35	28,75	108,75	20,05			
8	26	13,8	159,6	18,8	21,2	83,8	16,4	30	110	21,2			
9	29	14,9	160,8	19,9	22,6	84,9	17,45	31,25	111,25	22,35			
10	32	16	162	21	24	86	18,5	32,5	112,5	23,5	55	700	150
11	35	17,1	163,2	22,1	25,4	87,1	19,55	33,75	113,75	24,65			
12	38	18,2	164,4	23,2	26,8	88,2	20,6	35	115	25,8			
13	42	19,3	165,6	24,3	28,2	89,3	21,65	36,25	116,25	26,95			
14	45	20,4	166,8	25,4	29,6	90,4	22,7	37,5	117,5	28,1			
15	48	21,5	168	26,5	31	91,5	23,75	38,75	118,75	29,25	67,5	800	175
16	51	22,6	169,2	27,6	32,4	92,6	24,8	40	120	30,4			
17	54	23,7	170,4	28,7	33,8	93,7	25,85	41,25	121,25	31,55			
18	58	24,8	171,6	29,8	35,2	94,8	26,9	42,5	122,5	32,7			
19	61	25,9	172,8	30,9	36,6	95,9	27,95	43,75	123,75	33,85			
20	64	27	174	32	38	97	29	45	125	35	80	900	200

Ilustración 51 - Tabla con la progresión de los enemigos en cada ronda

A continuación, se pueden ver separados por colores todos los tipos de enemigos existentes en ALPHAS con sus correspondientes parámetros que irán incrementando con el paso de las oleadas. Estos parámetros son: el daño que infligen, la vida que tienen y la experiencia que recibirá el jugador si derrota a ese enemigo.

6.12.2. PROGRESION DE LOS JUGADORES

En la siguiente imagen podemos observar cual va a ser la progresión de los jugadores conforme van incrementando su nivel tras ganar experiencia derrotando a los enemigos en cada oleada.

Base Exp	25																					
Factor	2,5	1,2	10	12	0,7	1,1	0,002	1,1	15	10	0,9	1,1	0,002	1,3	10	12	0,5	1,1	0,002			
		BLUE						GREEN						YELLOW								
NIVEL	EXPERIENCIA TOPE	DAÑO	VIDA	MAGIA	ARMADURA	VELOCIDAD	VEL. ATAQUE	DAÑO	VIDA	MAGIA	ARMADURA	VELOCIDAD	VEL. ATAQUE	DAÑO	VIDA	MAGIA	ARMADURA	VELOCIDAD	VEL. ATAQUE			
1	30,66	10	100	225	3	100	0,25	10	125	200	5	95	0,3	15	100	200	1	120	0,15			
2	40,59	12,4	120	249	4,4	100	0,246	12,2	155	220	6,8	95	0,296	17,6	120	224	2	120	0,146			
3	57,00	13,6	130	261	5,1	100	0,244	13,3	170	230	7,7	95	0,294	18,9	130	236	2,5	120	0,144			
4	80,90	14,8	140	273	5,8	100	0,242	14,4	185	240	8,6	95	0,292	20,2	140	248	3	120	0,142			
5	113,18	16	150	285	6,5	100	0,24	15,5	200	250	9,5	95	0,29	21,5	150	260	3,5	120	0,14			
6	154,64	17,2	160	297	7,2	100	0,238	16,6	215	260	10,4	95	0,288	22,8	160	272	4	120	0,138			
7	206,02	18,4	170	309	7,9	100	0,236	17,7	230	270	11,3	95	0,286	24,1	170	284	4,5	120	0,136			
8	268,00	19,6	180	321	8,6	100	0,234	18,8	245	280	12,2	95	0,284	25,4	180	296	5	120	0,134			
9	341,23	20,8	190	333	9,3	100	0,232	19,9	260	290	13,1	95	0,282	26,7	190	308	5,5	120	0,132			
10	426,31	22	200	345	10	100	0,23	21	275	300	14	95	0,28	28	200	320	6	120	0,13			
11	523,83	23,2	210	357	10,7	100	0,228	22,1	290	310	14,9	95	0,278	29,3	210	332	6,5	120	0,128			
12	634,34	24,4	220	369	11,4	100	0,226	23,2	305	320	15,8	95	0,276	30,6	220	344	7	120	0,126			
13	758,36	25,6	230	381	12,1	100	0,224	24,3	320	330	16,7	95	0,274	31,9	230	356	7,5	120	0,124			
14	896,42	26,8	240	393	12,8	100	0,222	25,4	335	340	17,6	95	0,272	33,2	240	368	8	120	0,122			
15	1.049,00	28	250	405	13,5	100	0,22	26,5	350	350	18,5	95	0,27	34,5	250	380	8,5	120	0,12			
16	1.216,58	29,2	260	417	14,2	100	0,218	27,6	365	360	19,4	95	0,268	35,8	260	392	9	120	0,118			
17	1.399,62	30,4	270	429	14,9	100	0,216	28,7	380	370	20,3	95	0,266	37,1	270	404	9,5	120	0,116			
18	1.598,56	31,6	280	441	15,6	100	0,214	29,8	395	380	21,2	95	0,264	38,4	280	416	10	120	0,114			
19	1.813,85	32,8	290	453	16,3	100	0,212	30,9	410	390	22,1	95	0,262	39,7	290	428	10,5	120	0,112			
20	2.045,92	34	300	465	17	100	0,21	32	425	400	23	95	0,26	41	300	440	11	120	0,11			

Ilustración 52 - Tabla con la progresión de los jugadores tras ir subiendo de nivel

En la primera columna encontramos el nivel y en la columna de la derecha encontramos la cantidad de experiencia necesaria para pasar al siguiente nivel. La experiencia es calculada con la siguiente función exponencial:

$$expSiguienteNivel = experienciaBase + \left((nivel + 1)^{factor}\right)$$

Donde:

- **expSiguienteNivel:** experiencia necesaria para subir al siguiente nivel.
- **experienciaBase:** constante (25), que indica el punto de corte con el eje Y.
- **nivel:** nivel actual en el que se encuentra el personaje.
- **factor:** constante (2.5) que indica como crece la cantidad de experiencia necesaria.

Ilustración 53 - Gráfica con la curva de experiencia para subir de nivel

Continuando con la table de los personajes, a continuación, encontramos los 3 personajes jugables con los valores de sus estadísticas (daño, vida, magia, armadura, velocidad y velocidad de ataque) en cada nivel.

Si nos fijamos un poco más en la tabla podemos observar 3 comportamientos diferentes respecto a las estadísticas.

- Los valores de ataque, vida, magia y armadura **incrementan** con cada nivel ganado.
- El valor de la velocidad **es igual** independientemente del nivel en el que nos encontremos.
- El valor de la velocidad de ataque **decrementa** con cada nivel ganado. Esto es así debido a que a menor sea el valor de velocidad, más rápido lanzará los ataques básicos nuestro personaje. Además, este es el único valor que una vez alcanzado al nivel 20 no se modificará más.

6.13. MODO COOPERATIVO

El modo cooperativo permite jugar con otro jugador desde la misma pantalla, lo cual aumenta las horas de juego debido a que jugar con amigos suele ser más divertido.

Este modo ha estado presente desde que se empezó con el desarrollo, por ello durante el mismo mientras que se desarrollaba el modo de un jugador se implementaba a la vez el modo cooperativo para agilizar las cosas.

La clase sobre las que más recae el peso del cooperativo es en los enemigos, ya que estos deben de elegir a que jugador seguir para derrotarlo. Mientras que en el modo de 1 jugador el cálculo era fácil e iban siempre tras el único jugador existente, en el modo de 2 jugadores deben de elegir a cuál seguir. Para ello se calcula cuál de los dos jugadores está más cerca de ese enemigo y cuál es la vida de cada uno. Con estos dos datos de distancia y vida el enemigo selecciona cual es el objetivo más viable para perseguir.

Otra clase que también tiene un peso extra es la clase jugador, ya que debe de saber a qué jugador mover. El jugador 1 siempre se moverá con el teclado y el jugador 2 con un joystick. Para poder hacer esta distinción nada más crear a un jugador en el modo cooperativo se le asigna teclado o ratón dependiendo de si es el 1 o el 2.

7. COMPARACION ENTRE DESARROLLOS

En esta parte vamos a comparar como han cambiado los hábitos y la forma de programar tras casi 2 años de experiencia con la programación y el lenguaje C++.

Para poder hacer esta comparación se utilizará el proyecto ALPHAS original el cual fue desarrollado un año atrás (a este proyecto le llamaremos “proyecto original” de ahora en adelante) y el proyecto ALPHAS actual que ha sido desarrollado a día de hoy (a este proyecto le llamaremos “proyecto actual” de ahora en adelante).

Principalmente nos vamos a centrar en cómo han mejorado los siguientes aspectos:

- Organización de ficheros.
- Conocimientos del lenguaje (clases, estructuras, …).
- Organización y limpieza del código.
- Optimización.

7.1. ORGANIZACIÓN DE FICHEROS

El primer aspecto por comparar entre los dos proyectos es la organización de los ficheros de código. Aunque parezca irrelevante, una buena organización de los ficheros de forma jerárquica ahorra mucho tiempo de trabajo cuando el proyecto crece. Si sabemos lo que buscamos y sabemos su jerarquía, localizar un archivo es solo cuestión de unos pocos clics.

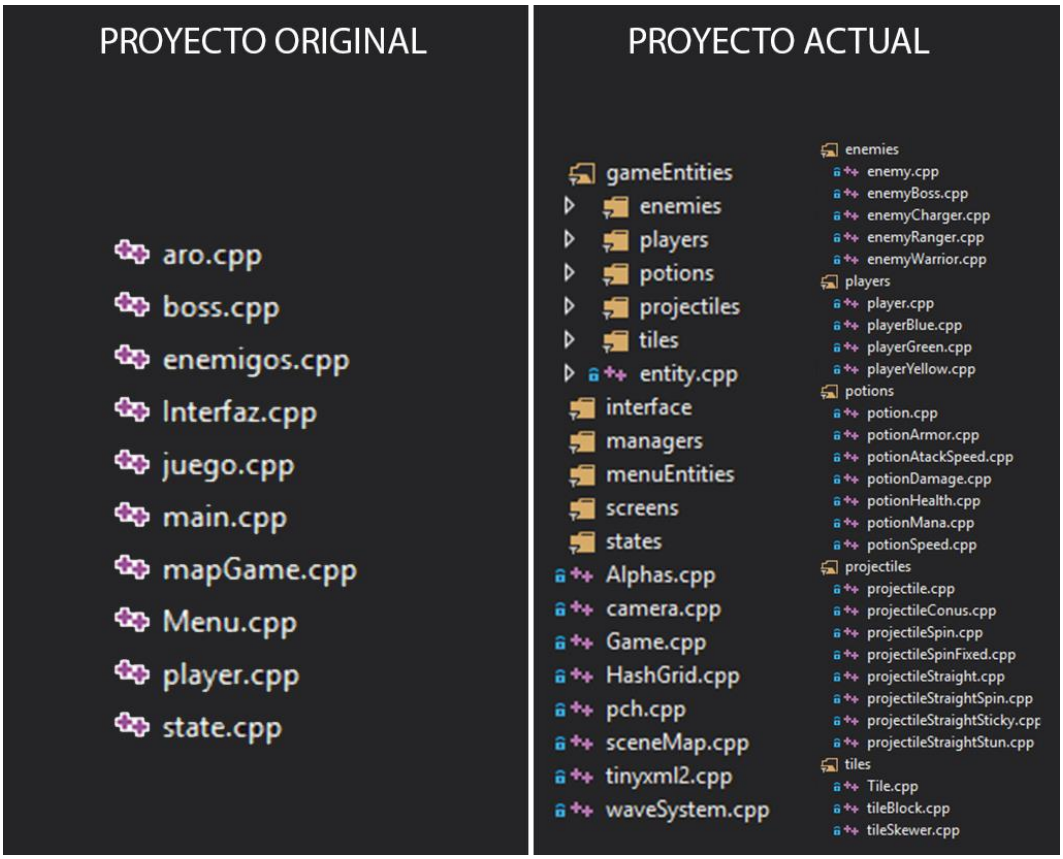

Ilustración 54 - Comparación de la organización de ficheros

Como podemos observar en la imagen anterior, en el proyecto original no existía ninguna organización de ficheros en carpetas, sino que estaban todos los archivos juntos en un una misma carpeta. Esto se debe a que como no se conocía a fondo el lenguaje de desarrollo C++, no se aprovechó su principal punto fuerte (hablaremos de esto en el siguiente apartado).

No obstante, en el proyecto actual podemos observar como la organización de los archivos tiene un mayor peso. Todas las entidades del juego están dentro de una misma carpeta (gameEntities), la cual tiene dentro las carpetas con el nombre de la entidad y dentro de esta están todos los tipos diferentes de esa entidad. Además de la carpeta de las entidades del juego encontramos la carpeta "interface" que contiene todos los elementos de la interfaz, la carpeta "menuEntities" que contiene todo lo relacionado con las entidades de los menús (botones, imágenes, etc.) y la carpeta "screens" que contiene las múltiples pantallas que se pueden encontrar en el proyecto.

Además de la organización en carpetas, el nombre de los archivos también es muy significativo, lo que facilita aún más localizar un archivo si se sabe que es lo que se busca o se sabe en qué parte hay que buscar.

7.2. CONOCIMIENTO DEL LENGUAJE

El segundo aspecto por comparar es el conocimiento del lenguaje tras casi dos años utilizándolo. Durante este tiempo se ha ido perfeccionando el conocimiento sobre los punteros, las herencias, las clases, las estructuras, las enumeraciones, el uso de la memoria dinámica y el uso de patrones de diseño.

7.2.1. CLASES Y HERENCIAS

En el proyecto original no se utilizaban estas funcionalidades, lo que conllevaba a repetir mucho código o a escribir clases muy grandes que tenían muchas funcionalidades.

Esto se observa sobre todo en las clases "player" y "aro", las cuales vamos a desglosar un poco viendo sus fallos y posteriormente veremos cómo se han subsanado en el proyecto actual ("player" y "proyectil" en el proyecto actual respectivamente).

7.2.1.1. CLASE PLAYER

La clase player en el proyecto original se utiliza para crear los dos tipos de personajes disponibles (azul y verde), indicando en el constructor el color del personaje que queremos jugar.

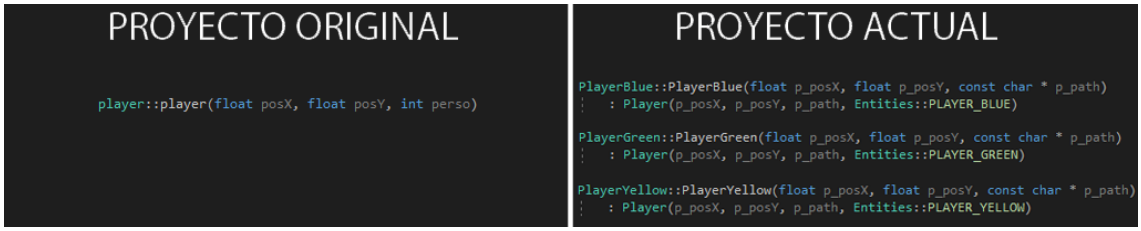

Ilustración 55 - Comparación creación jugadores

En el proyecto actual, cada personaje tiene su propia clase y dependiendo del color que sea se llama a una clase o a otra. Además, los tres personajes indistintamente del color que sea tienen como padre la clase Player, la cual contiene todos los datos en común de los jugadores.

7.2.1.2. CLASE ARO/PROYECTIL

La clase aro (proyectil en el proyecto actual), al igual que la clase player se utiliza para crear los dos tipos de ataques existentes en el juego del proyecto original.

En el proyecto original, al crear los personajes se inicializan todos los aros disponibles y se ocultan hasta que son llamados. Una vez se llama un aro este se activa y se le dice cuál va a ser su función. Al terminar su función el aro no es destruido, sino que vuelve al estado inicial en el cual está quieto y oculto.

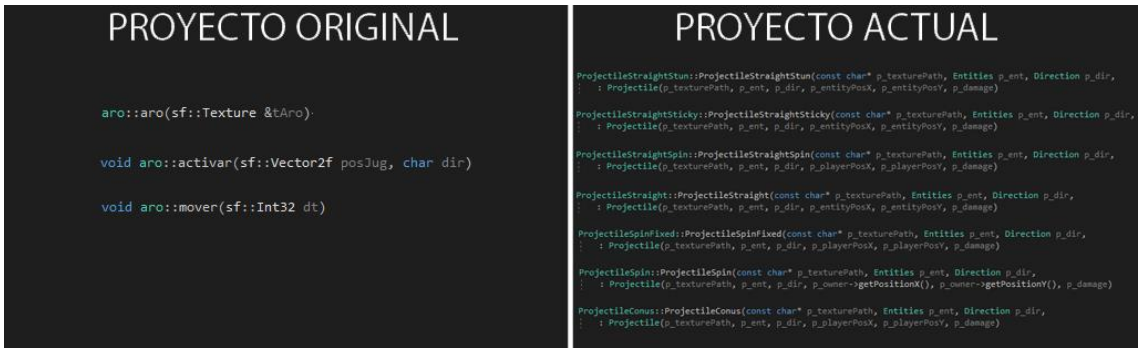

Ilustración 56 - Comparación creación proyectiles

En el proyecto actual, cada proyectil tiene su propia clase por lo que tenemos varias dependiendo de lo que haga ese proyectil. Cuando realizamos un ataque, se crea ese proyectil y se actualiza dependiendo de su comportamiento que esta detallado en la función update(). Una vez termina su función el proyectil es destruido.

7.2.1.3. CLASE JUEGO

Además de las clases player y proyectil, en el proyecto original el juego se desarrollaba por completo en el fichero main.cpp, lo cual daba como resultado lo siguiente:

```
void inputSiempre(sf::RenderWindow &window,state &state);
void updateGame(sf::RenderWindow &window, sf::Int32 dt, player &jugador, std::vector<enemigos> &enemigo, mapGame &mapa, state &state, sf::Texture
void render(sf::RenderWindow &window, float percentTick, player jugador, std::vector<enemigos> enemigo, mapGame mapa, sf::View vista, state &stat
void updateMenu (sf::RenderWindow &window,Menu &menu,int &jugadorSeleccionado,int width,int height,state &state,juego &juego, player &jugador);
void renderMenu(sf::RenderWindow &window, Menu menu);
void GameOver(sf::RenderWindow &window, state &state, sf::View &vista,Menu &menu);
void Oleadas(sf::RenderWindow &window,juego &juego,int &oleada,std::vector<enemigos> &enemigo,Interfaz &interfaz, boss* boss,player &jugador,stat
void Pausa(sf::RenderWindow &window, state &state, sf::View &vista,int &select,Menu &menu,boss* &boss);
void Ganar(sf::RenderWindow &window, state &state, sf::View &vista,int &select,Menu &menu,boss* &boss);
```

Ilustración 57 - Métodos del juego en el proyecto original

El resultado de usar el fichero main como elemento central da como resultado tener que crear funciones a las cuales hay que pasarle todos y cada uno de los elementos y/o parámetros necesarios para hacer funcionar el juego. Esto hace que un simple cambio, prueba o error se convierta en un gran quebradero de cabeza para el programador.

Este gran fallo de programador inexperto tiene una fácil solución que esta implementada en el proyecto actual. La solución es crear una clase Game que gestione todo el proyecto y al hacerlo el fichero main se quedaría de la siguiente forma:

```
int main()
{
    Game game;

    game.run();

    return 0;
```

Ilustración 58 - Fichero main del proyecto actual

La clase Game gestiona el bucle principal del juego, el cual contiene el estado actual en el que se encuentra el proyecto una vez está en

ejecución. Gracias a estos estados se simplifica el entendimiento del bucle facilitando la tarea del desarrollador.

7.2.2. ESTRUCTURAS

Las estructuras son grupos de datos agrupados bajo un mismo nombre. Este tipo de datos no es usado en ningún momento en el proyecto original. No obstante, en el proyecto actual si han sido utilizados en diversas ocasiones, facilitando así el paso de datos.

La estructura más usada es Point (Punto), la cual contiene solo dos datos que son dos enteros (x, y) que sirven para indicar la posición de un elemento o entidad. Esta estructura es usada en todas las entidades que tienen que estar posicionadas en el espacio. Esta estructura facilita la lectura de los métodos al pasar un dato de tipo Punto y no dos enteros que hacen referencia a la posición.

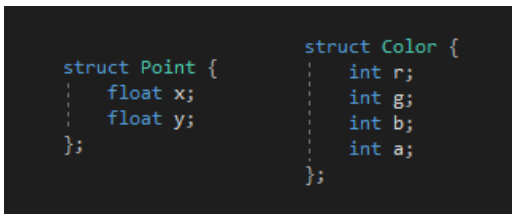

Ilustración 59 - Estructuras más utilizadas

Otra estructura que se utiliza mucho es Color, la cual es usada principalmente para el menú y la interfaz. Esta estructura contiene 4 valores de enteros que hacen referencia a los valores RGBA de un color. Gracias a esta estructura conseguimos pasarle el valor del color a SFML con solo 1 tipo de dato y no con 4 datos por parámetro mediante la fachada.

7.2.3. ENUMERACIONES

Las enumeraciones son otro tipo de datos que no fueron utilizados en el proyecto original, pero si en el proyecto actual. Las enumeraciones son un tipo especial de datos las cuales asignan datos enteros a nombres, lo que facilita la lectura del código.

La enumeración más utilizada e importante es Entities (Entidades). Esta enumeración tiene un doble uso. Su principal utilidad es que permite identificar en que fila del Sprite sheet se encuentra la entidad que queremos utilizar. Su segundo uso viene cuando queremos comprobar si una entidad debe colisionar con otra.

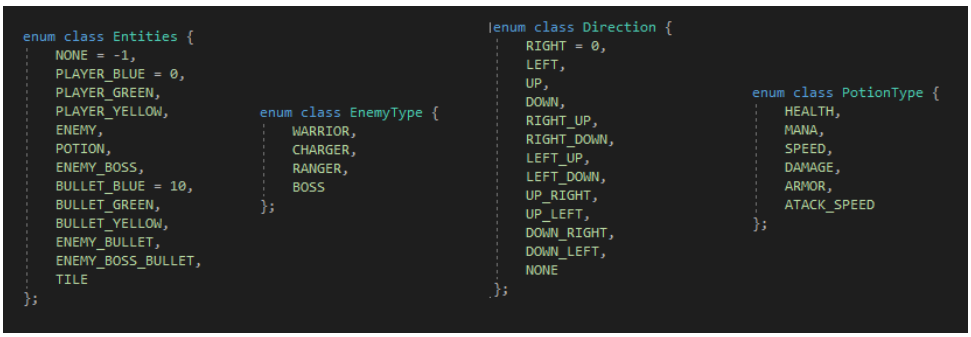

Ilustración 60 - Enumeraciones principales

Además de esta, existen otras 3 enumeraciones más.

- **EnemyType,** contiene todos los tipos de enemigos existentes. Al saber el número total de tipos de enemigos que existen podemos sacar un número aleatorio entre 0 y ese número total de tipos de enemigos y generar enemigos de forma aleatoria.
- **Potion,** contiene todos los tipos de poción existentes. Esta enumeración complementa a la de Entities para saber cuál es la poción que hay que pintar.
- **Direction,** contiene todas las direcciones en las que el jugador puede lanzar un ataque.

7.2.4. PATRONES DE DISEÑO

En el proyecto original se "implemento" el patrón de diseño *State*. Implemento va entrecomillado debido a que no está bien implementado, sino que es una aproximación a lo que el verdadero patrón *State* propone. No obstante, en el proyecto actual si se implementa el patrón *State* de forma correcta.

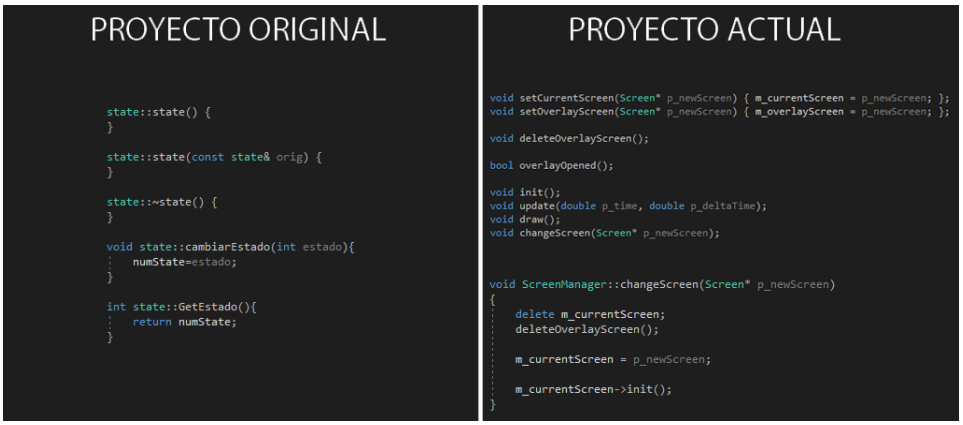

Ilustración 61 - Implementación original y actual del patrón State

En el proyecto original hay implementada una clase *state* pero es en realidad un intento de máquina de estados, puesto que la única función que tiene es guardar un valor entero para posteriormente devolverlo.

La máquina de estados (pantallas en este caso) implementada en el proyecto actual gestiona el cambio de estado borrando el anterior y creando el nuevo estado, además de inicializarlos, actualizarlos y dibujarlos si fuera necesario.

En el proyecto original, al no tener los estados implementados en clases aparte se implementan directamente en el bucle del juego. En el bucle del juego obtenemos de la máquina de estados el valor y mediante una serie de *if-else* creamos la funcionalidad y la transición entre pantallas y estados del juego.

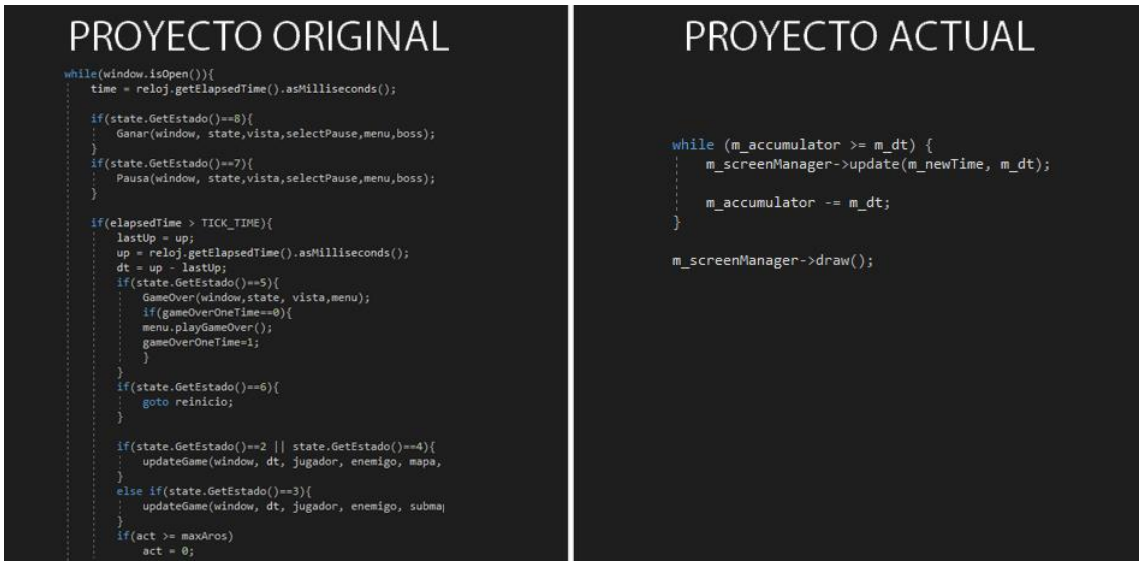

Ilustración 62 - Comparación de la implementación de los estados

Como podemos observar en el proyecto actual, el bucle del juego queda muy sencillo resumido a 4 líneas, no como en la anterior versión donde se necesitaban muchas más líneas. En esta versión, la máquina de estados es la encargada de saber cuál es el estado actual e invocar la función de *update()* correspondiente para actualizar el estado del juego.

```
void ScreenGame::update(double p_time, double p_deltaTime) {
    //UPDATE HASH GRID

    //UPDATE TILES

    //UPDATE PLAYERS

    //UPDATE ENEMIES

    //UPDATE POTIONS

    //UPDATE INTERFACE
}
```

Ilustración 63 - Ejemplo de pseudo-código de actualización del estado game

7.3. ORGANIZACION Y LIMPIEZA DEL CODIGO

Otro aspecto por comparar entre ambas versiones es la organización y limpieza del código. Teniendo una buena organización y un código limpio ganaremos en legibilidad. Con el paso del tiempo y al ir ganando experiencia se empiezan a seguir una serie de pautas ya sean estándares o reglas creadas por el propio desarrollador.

Entre estas pautas se suelen encontrar las tabulaciones y el espaciado, los comentarios, la nomenclatura o idioma de desarrollo, entre otras.

El proyecto original carece de cualquier normal o pauta a la hora de desarrollar código. La única pauta que más o menos se sigue es la relacionada con las tabulaciones, aunque no se sigue en la totalidad de las clases o métodos.

No obstante, el proyecto actual sí que sigue una serie de pautas a la hora de escribir código, lo que facilita su entendimiento por personas ajenas al proyecto. La principal característica es que todo el código está escrito en inglés, ya sean clases, variables, métodos o comentarios.

En cuanto a las variables se sigue un regla para saber cuál es el origen de la variable y saber cuál es el ámbito de esta.

- **m_xxx**: es una variable de una clase.
- **p_xxx**: es una variable que se ha pasado por parámetro en un método.
- **t_xxx**: es una variable que se ha creado en un método y solo existe dentro del método.

Además, si alguna variable o método hace referencia a una palabra de gran longitud, esta palabra no puede acortarse o simplificarse para que no pierda su significado y así de esta forma no pueda causar confusión. También, si son 2 o más palabras no se puede usar ninguna separación, sino que cada nueva palabra se identificará con la letra en mayúsculas. Por ejemplo, si queremos hacer referencia a la velocidad de ataque como variable de clase, la variable se escribirá de la siguiente manera: *m_velocidadDeAtaque*.

7.4. OPTIMIZACION

La optimización es otro aspecto por comparar entre ambas versiones, en concreto se va a comparar las colisiones de ambos proyectos.

En el proyecto original, las colisiones se centraban en una única capa del mapa creado por Tiled, por lo cual solo se debía de hacer la comparación con esa capa. A simple vista parece óptimo, pero el problema es que se comprobaba cada entidad con todas las entidades de esa capa, estuvieran cerca o lejos. Además, si el mapa era de 10x10 y solo había 20 entidades colisionables en el mapa, se comprobaban las 100 posiciones de la matriz del mapa, cosa que no era óptima.

No obstante, en el proyecto actual se ha implementado un Hash Grid para optimizar el cálculo de colisiones. Gracias a esta estructura no debemos de comparar una entidad con el resto de las entidades del mapa, sino que solo comprobamos las que están en su misma celda. Además, para agilizar el rellenado del Hash Grid se ha creado un contenedor el cual contiene todas las entidades del mapa colisionables. Esta optimización permite no tener que comprobar las entidades que no causan efecto ninguno salvo el efecto visual (el suelo del mapa).

Ilustración 64 - Comparación optimización

8. CONCLUSIONES

Llegados a este punto se puede afirmar que se ha conseguido completar con éxito los objetivos marcados al inicio de este mismo documento.

En primer lugar, se analizó el documento del proyecto original. De este documento se pudieron obtener los objetivos cumplidos y no cumplidos del proyecto, lo cual sentaba unas bases para la creación de la nueva versión. Al terminar de analizar el documento se procedió a analizar la versión final del juego. De esta versión se obtuvieron detalles que podrían haber sido omitidos en el documento y que tenían un peso relevante para la nueva versión.

Con todos estos datos se procedió con el análisis de estos. Durante la fase del análisis algunas mecánicas se mantuvieron iguales, otras se modificaron y otras fueron descartadas que no terminaban de encajar del todo en la nueva versión.

En segundo lugar, y con el primer paso terminado se comenzó con el desarrollo de la nueva versión del videojuego con las mecánicas y funcionalidades establecidas. Durante el desarrollo fueron surgiendo varios contratiempos que fueron subsanados de forma eficiente con la mayor brevedad posible.

Para terminar, se han comparado las dos versiones del mismo producto. Del análisis de ambas versiones se han conseguido obtener varios errores a la hora de desarrollar el mismo producto desde dos perspectivas diferentes, una sin experiencia y otra con experiencia. Los errores en los que nos hemos fijado han sido los relacionados con el conocimiento del lenguajes y la optimización de un videojuego.

Han pasado 4 años desde que entré en la carrera y conocí el mundo de la programación. A los 3 años nos plantearon el crear un videojuego con los conocimientos que teníamos en ese momento. A día de hoy habiendo superado todos los cursos se me plantea el ultimo reto, el cual acepto con entusiasmo y con ganas de probar todos los conocimientos adquiridos, por lo que decido rehacer el primer videojuego que desarrolle.

Al inicio del trabajo y observar el código del primer proyecto no podía creer lo que veía y aún más no sabía cómo podía funcionar ese proyecto. Ahora que he terminado la nueva versión me he dado cuenta de cuánto he mejorado en el ámbito de la programación y cuánto se puede mejorar en retos de este tipo, en retos de superación personal. No obstante, seguro que dentro de 2 años vuelvo a mirar este proyecto y me pasará lo mismo que me pasó con su versión original, ya que nunca se para de aprender y mejorar en el ámbito de la programación.

Para finalizar, voy a volver a citar la frase que en mi opinión resume el alma de este trabajo:

“La idea de este proyecto puede servir para que generaciones futuras no le teman al desarrollo, ya que en sus inicios afrontar un desarrollo de un videojuego puede parecer algo imposible, pero como todo, es cuestión de dedicarle tiempo y mejorar en el entendimiento del lenguaje y de la librería que decidas utilizar.”

9. BIBLIOGRAFIA

[1] Videojuegos:
https://es.wikipedia.org/wiki/Industria_de_los_videojuegos

[2] Beat 'em up:
https://es.wikipedia.org/wiki/Beat_%27em_up

[3] Roguelike:
https://es.wikipedia.org/wiki/Roguelike

[4] Rol-Playing Game (RPG):
https://es.wikipedia.org/wiki/Videojuego_de_rol

[5] Hero Siege:
https://herosiege.fandom.com/wiki/HeroSiege_Wiki

[6] Final Exam:
http://finalexam.wikia.com/wiki/Final_Exam_(Game)

[7] Lost Castle:
http://es.indie.wikia.com/wiki/Lost_Castle

[8] God of War:
https://es.wikipedia.org/wiki/God_of_War_(serie)

[9] Nier: Automata:
https://es.wikipedia.org/wiki/Nier:_Automata

[10] Devil May Cry:
https://es.wikipedia.org/wiki/Devil_May_Cry_(serie)

[11] Kanban:
https://es.wikipedia.org/wiki/Kanban_(desarrollo)

[12] Git:
https://es.wikipedia.org/wiki/Git

[13] GitHub:
https://es.wikipedia.org/wiki/GitHub

[14] IDE:
https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado

[15] Sprite:
https://es.wikipedia.org/wiki/Sprite_(videojuegos)

[16] Patrón Facade:
https://es.wikipedia.org/wiki/Facade_(patr%C3%B3n_de_dise%C3%B1o)

[17] Patrón Singleton:
https://es.wikipedia.org/wiki/Singleton

[18] Patrón State:
https://es.wikipedia.org/wiki/State_(patr%C3%B3n_de_dise%C3%B1o)