Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

GRADO EN INGENIERÍA INFORMÁTICA DE GESTIÓN Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

PHOENIX

Alumno/Alumna: Ruiz Martínez, Alesander
Director/Directora (1): Blanco Jauregi, Begoña
Director/Directora (2):

Curso: 2017-2018

Fecha: Lunes, 18-06-2018

Índice de contenidos

Índice de tablas

Índice de ilustraciones

INTRODUCCIÓN

Hace tiempo que la sociedad ha cambiado. El entretenimiento en el ámbito tecnológico se ha movido en nuevas direcciones, en varios sentidos, y en eso se va a basar el proyecto que aquí se plantea.

En primer lugar, la forma de entender la diversión mediante los videojuegos ha dado un giro radical. En los años 90, época del "boom" de este tipo de ocio, primaban los videojuegos de características similares: dificultad, alta duración, énfasis en una historia y progresión del jugador, entre otras. El género de *rol*, dónde el jugador asume el papel de un personaje que se embarcará en una aventura en la que recorrerá mundos enfrentándose a enemigos y desafíos y desarrollando sus habilidades, con especial atención a los elementos mencionados, era el género que dominaba el panorama mundial.

Sin previo aviso y con la adición de nuevos y más jóvenes jugadores, el mercado comenzó a cambiar a principios del siglo XXI. El énfasis de los videojuegos dejó de estar en la aventura, duración y dificultad y se comenzaron a ver tendencias que se aplican a muchos otros sectores del mundo tecnológico actual. Se empezó a buscar la gratificación instantánea del usuario, con juegos cada vez más sencillos y con menos costes de desarrollo, pero que incorporaban elementos multijugador o adictivos. Lejos quedó la complejidad de una historia o aventura, para dar lugar a la sencillez, tanto técnica como jugable, que a tantos jugadores ha conseguido atraer.

Ilustración 1 Xenosaga III

***Xenosaga Episode III: Also Sprach Zarathustra**, lanzado al mercado en 2006, es uno de los últimos videojuegos "de la vieja escuela" que vio la PlayStation 2. Centrado en narrar una historia larga y compleja, acompañado de una gran banda sonora, fue un fracaso y un punto de referencia para el resto de desarrolladoras. Aunque no era la última entrega de la saga, la compañía que lo desarrolló decidió terminarla aquí y replantearse su filosofía.*
http://egameboss.com/wp-

Ilustración 2 Candy Crush

***Candy Crush**, uno de los ejemplos más claros y conocidos del cambio de filosofía detrás del desarrollo de videojuegos. Buscando una estética sencilla pero atractiva, su jugabilidad se basa en elementos de gratificación instantánea y adictivos. Coincidiendo además con el auge de los smartphones, es un juego extendido por todo el mundo con más de 314 millones de usuarios y reportando beneficios de más de 200 millones de dólares entre 2016 y 2017(1).https://lh3.googleusercontent.com/rV*

De forma paralela a esta nueva tendencia se dio el auge de la tecnología móvil y los *smartphones,* dispositivos móviles inteligentes que aunque no llevan más de una década con nosotros han cambiado radicalmente la forma de vivir el día a día. En lo relacionado con los videojuegos, el jugador medio ha pasado de jugar en la tranquilidad de su salón o con otras personas a, por un estilo de vida atareado y estresante, jugar sesiones cortas en sus viajes al trabajo o descansos. Es por esto que las características de estos nuevos videojuegos son tan atractivas; proporcionan diversión o distracción inmediata sin tener que prestar excesiva atención al contenido de lo que se está jugando.

Es aquí donde entra Phoenix. Este proyecto busca coger lo mejor de ambos mundos y juntarlo con el fin de llegar a la mayoría de usuarios, rescatando la pasión que antes destilaban los videojuegos con las nuevas evoluciones tecnológicas y sociales. Se pretende realizar un videojuego con todas las características de un juego de rol clásico, además de su estética, evocando el sentimiento con el que muchos de nosotros crecimos y aplicándolo a los Smartphones. Es por esto que el juego será desarrollado para PC y Android, con posible futura ampliación a iOS.

Obviamente, es muy difícil encontrar en 2017 algo que no se haya hecho previamente, y esto no es una excepción. Como ya se indicará más adelante hay numerosos videojuegos, tanto para móvil como para PC, que trabajan sobre esta idea. ¿Qué hace a Phoenix especial entonces? Se buscará algo que no he podido ver aún en ningún juego de estas características: la mezcla de estilos de juegos. Los juegos antiguos, en parte debido a las limitaciones técnicas, solían emplear uno de dos enfoques: *side-scroll* o *top-down*. El primero consistía en una cámara que seguía al jugador desde su perfil, pudiendo moverse solamente por el eje X o el eje Y si se implementaba la posibilidad de saltar. El enfoque *top-down* consistía en seguir al jugador con una cámara sobre su cabeza, pudiendo moverse este, por lo tanto, en 4 direcciones.

Ilustración 3 Side Scroll vs Top Down

A la izquierda (2) un ejemplo de side-scroll, con la cámara desde el perfil. A la derecha (3) uno de top-down, visto desde arriba.

Gracias a las posibilidades que ofrecen LibGDX y Box2D (herramientas que se describirán en profundidad más adelante), se pretende que en según qué sección del juego se encuentre el jugador se emplee uno u otro modo, aportando esto variedad y mayor entretenimiento a la jugabilidad, pudiendo mezclar así secciones de *platforming* con la jugabilidad de mundo abierto top-down clásica.

El uso e implementación de estas herramientas va más allá de lo visto en la carrera, y esta es otra de

las razones por las que se han escogido. No solamente son herramientas empleadas en numerosos juegos hoy en día, sino que además supone un reto personal aprender a usar un framework y motor de físicas totalmente desconocidos.

Por lo tanto y en resumen, ¿qué es *Phoenix*? **Es un videojuego de rol, inspirado por aquellos que vieron la luz y el éxito en los años 90, que pretende combinar el encanto perdido de aquella época con las tecnologías y las tendencias sociales actuales, aportando además una mezcla de estilos de juego novedosa y que pretende entretener al jugador desde donde él lo quiera jugar**. El jugador recorrerá un mundo de fantasía siguiendo una historia, dónde podrá ganar experiencia, aprender y mejorar sus habilidades y disfrutar tanto de secciones de combate como de *platforming*. Además, y como se verá en la sección de *Valor Añadido*, a medida que se ha avanzado con el proyecto se ha logrado crear, no solo una experiencia jugable, sino un creador/editor de RPGs. Usuarios sin ninguna experiencia en el campo de la programación podrán crear sus propios mapas customizados dentro de las opciones existentes, compartirlos entre ellos, y hacer de Phoenix una experiencia sin fecha de caducidad y en constante expansión.

PLANTEAMIENTO INICIAL

Objetivos

El objetivo principal de este proyecto es desarrollar un videojuego de rol que, como ya se ha explicado, sea una mezcla de estilos característica de los juegos de los años 90 adaptada al mundo y al usuario actual. De ahí parte el segundo objetivo del proyecto; desarrollarlo para PC y sobre todo para Android con posible ampliación a iOS.

Como objetivo a título personal, se pretende aprender a usar las novedosas herramientas (descritas en profundidad más adelante) que se usarán en la implementación de Phoenix y que son usadas en el desarrollo de numerosos juegos encontrados en la *Play Store* de Google para dispositivos móviles. En la misma línea, se tiene el objetivo de entender en profundidad la complejidad y dificultades que supone el desarrollo de un videojuego, así como las diferencias técnicas y de programación que entraña respecto a lo visto en la carrera.

Arquitectura

La arquitectura del juego se basará en la programación en Java de la aplicación y el uso de archivos XML para los métodos de guardado y carga de datos, entre otros. Esto es así por varias razones.

En primer lugar, la cantidad de información a guardar no es lo suficientemente elevada como para justificar el uso de una BBDD convencional, ya que por cada partida se guardan elementos como los atributos del jugador, los objetos o habilidades que ha obtenido o su posición en el mapa. Estos elementos pueden ser fácilmente guardados en archivos XML, que son además menos pesados y harán que la aplicación se ejecute de forma más ligera y eficiente.

Además, y como se explicará con más detalle en la sección de *Diseño*, LibGDX cuenta con soporte para el uso de archivos XML, por lo que se pretende aprovechar esta característica del framework.

Estas ideas se desarrollarán en más profundidad a medida que se vayan describiendo las

herramientas que se emplearán, así como en las secciones de *Diseño* e *Implementación*.

La arquitectura contará, por lo tanto, con un **modelo** que gestionará toda la lógica interna del juego, y que almacenará la información de la mayoría de objetos (elementos como los estados del jugador/enemigos en caso de empezar una nueva partida, estados de mapas, etc), **vistas** que se encargarán de representar el apartado visual y menús, y **archivos XML** con los que se realizará la carga de información de objetos o elementos de textos muy largos (por ejemplo, todas las características, descripciones, precios de armas y armaduras, todos los diálogos de los NPC, etc.). Como se ha dicho, en las secciones de *Diseño e Implementación* todas estas ideas son expandidas.

Alcance

Para la realización de este proyecto se ha escogido un desarrollo iterativo e incremental, junto con la metodología SCRUM. Este tipo de desarrollo consiste en planificar distintos bloques temporales, que serán las iteraciones. En cada iteración se implementa una funcionalidad, siendo normalmente las primeras iteraciones las más relevantes al proyecto, creándose por lo tanto después de cada iteración un nuevo **prototipo**, cada vez más completo que el anterior. Se irán sucediendo los ciclos hasta llegar al resultado planificado.

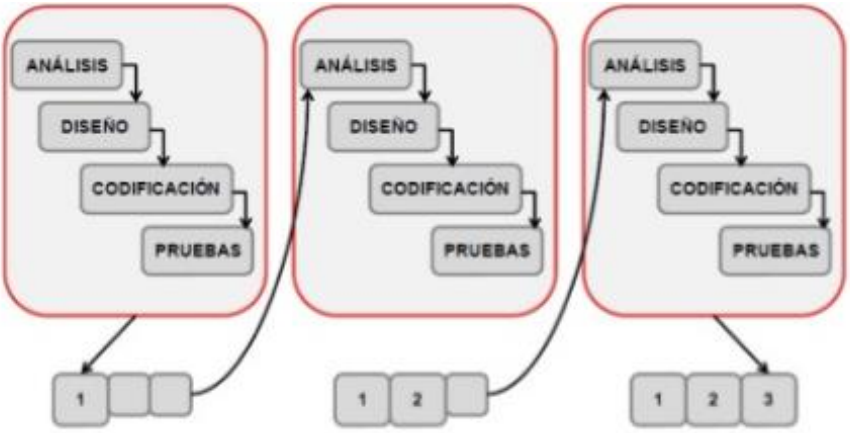


Ilustración 4 Desarrollo iterativo e incremental

(4)

Se ha escogido este tipo de desarrollo por varias razones:

- En primer lugar, se adecúa a la manera en la que se pretende desarrollar el juego; se parte de bases más simples para ir añadiendo funcionalidades cada vez más ricas
- Permite priorizar los elementos claves a implementar, dejando lo menos relevante para el final
- Se asegura calidad del software; si en algún momento se da un error, este solo afectará a la última iteración y será por lo tanto más fácil de solucionar
- Permite que haya lugar a cambios en el proyecto, algo que se prevé que pueda ocurrir
- Permite conocer el progreso real del proyecto, y ver por lo tanto si es posible llegar a los

objetivos planteados a tiempo y si se podrá o no realizar las ampliaciones planteadas

La metodología SCRUM es una metodología ágil especialmente indicada para proyectos con entornos complejos, con necesidad de resultados y "feedback" tempranos y constantes, con requisitos cambiantes y de naturaleza innovadora. Consiste en realizar entregas parciales y regulares del producto final, dando prioridad a aquellas que se consideran más importantes o el cliente demanda en ese momento. Se realizan además reuniones de equipo diarias para ver cómo va el desarrollo del proyecto, obtener "feedback" y poder así adaptarse rápidamente a los cambios o nuevos requisitos.

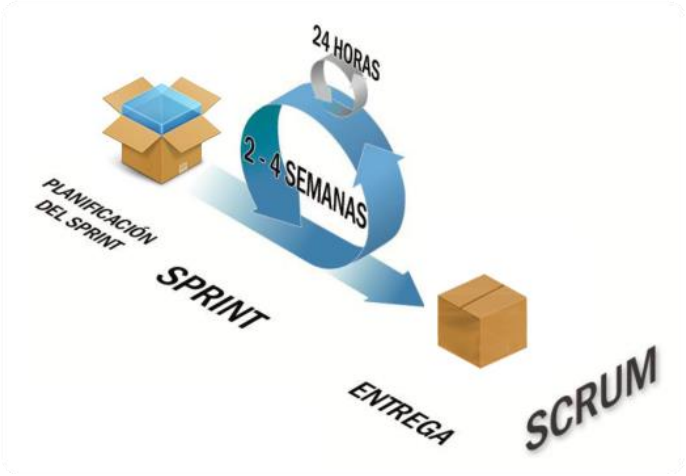


Ilustración 5 Metodología SCRUM

(5)

Por otra parte, se utilizará una metodología SCRUM para los plazos por las siguientes razones:

- La naturaleza flexible de SCRUM permite adaptarse a cambios en el software empleado
- Se hace más fácil asegurar un producto de calidad en el tiempo definido
- Gracias a los sprints cortos se adapta mejor a posibles cambios
- Con las "reuniones" frecuentes se puede ver como progresa el proyecto y si se va a llegar con soltura o no a los objetivos planificados
- Se adapta, por estas razones, al desarrollo iterativo e incremental

Los sprints se dividirán en base a las secciones que se describirán en la *Carga de Trabajo*. Serán de una duración de entre 15 y 20 días aproximadamente, y las reuniones se simularán a diario con uno mismo, analizando el progreso en el trabajo realizado y anotando las mejoras que se deben realizar, ya sea de carga de trabajo o de ritmo. También se pretende tener reuniones frecuentes (cada 15 días) con la tutora del proyecto. Los sprints se plantean de la siguiente manera:

- **Documentación**
  - *15 de Diciembre – 1 de Enero*
    - Introducción
    - Planteamiento inicial
      - Objetivos
      - Arquitectura
      - Alcance
      - Planificación temporal

    - *1 de Enero – 15 de Enero*
        - Planteamiento inicial
            - Herramientas
            - Gestión de riesgos
            - Evaluación económica
            - Antecedentes

- **Captura de requisitos**
    - *15 de Enero – 1 de Febrero*
- **Implementación**
    - *1 de Febrero – 20 de Febrero*
        - Diseño e implementación de mundos
        - Diseño e implementación de enemigos
    - *20 de Febrero – 11 de Marzo*
        - IA de enemigos e interacciones del jugador
        - Diseño e implementación de ciudades y NPC
    - *11 de Marzo – 3 de Abril*
        - Funcionalidades necesarias
        - Diseño e implementación de menús
    - *3 de Abril – 15 de Abril*
        - Port a Android

A continuación se mostrará la estructura de descomposición del trabajo. Constará de tres apartados principales: documentación, captura de requisitos e implementación. Coinciden con las tareas mencionadas en los sprints.

La **documentación** se dividirá en dos apartados:

- Elaboración del Documento de Objetivos del Proyecto (DOP), dónde se describirá los objetivos del proyecto y lo que se pretende conseguir.
- Memoria del TFG, en la que se recoge el resto de información del mismo.

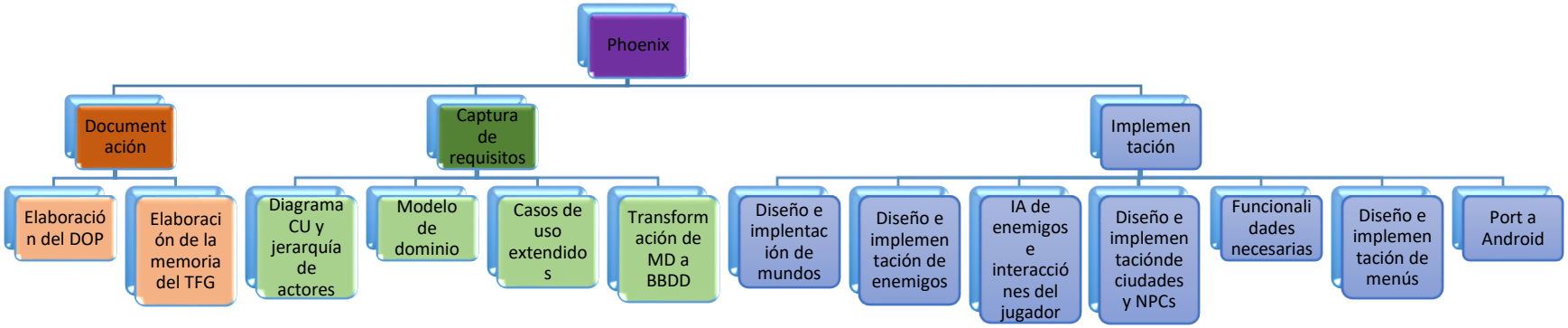
En segundo lugar, la **captura de requisitos** se dividirá en:

- Diagrama de casos de uso y jerarquía de actores
- Modelo de dominio
- Casos de uso extendidos
- Transformación del modelo de dominio a BBDD

Por último, la **implementación** constará de varias iteraciones o "paquetes" en los que se irán añadiendo nuevas funcionalidades al juego, haciéndolo de esta forma más completo en cada iteración. Los objetivos de cada iteración se han distribuido de la siguiente manera:

- Diseño e implementación de mundos
- Diseño e implementación de enemigos
- IA de enemigos e interacción del jugador con enemigos y mundos
- Diseño e implementación de ciudades y NPCs
- Funcionalidades necesarias: cargar, guardar y cálculo de puntuación
- Diseño e implementación de menús
- Port a Android

DIVISIÓN DE LA CARGA DE TRABAJO



Todos las sub-tareas de implantación constarán de:

- Análisis
- Diseño
- Implementación y pruebas

Ilustración 6 Carga de trabajo

A continuación se describirán con detalle todas las tareas en la carga de trabajo:

Elaboración del DOP
Duración estimada: 45 horas
Descripción: Elaboración del documento en el que se encuentran recogidas las intenciones y objetivos del proyecto.
Entradas: ----------
Salidas/Entregables: Documento de Objetivos del Proyecto (DOP)
Herramientas necesarias: Un ordenador y Microsoft Word
Precedencias: ----------

Tabla 1 Elaboración del DOP

Elaboración de la Memoria del TFG
Duración estimada: 60 horas
Descripción: Elaboración de la memoria del proyecto, en la cual se describirá todo lo referente al TFG.
Entradas: Documento de objetivos del proyecto.
Salidas/Entregables: Memoria del TFG
Herramientas necesarias: Un ordenador, Microsoft Office, Visual Paradigm, Android Studio
Precedencias: Realización de la parte de Análisis, Diseño e Implementación y Pruebas de los módulos de software.

Tabla 2 Elaboración de la Memoria del TFG

DIAGRAMA DE CASOS DE USO Y JERARQUÍA DE ACTORES
Duración estimada: 8 horas
Descripción: Elaboración del diagrama de casos de uso, que representará las funcionalidades de la aplicación así como los posibles actores de cada una de ellas.
Entradas: ----------
Salidas/Entregables: Diagrama de Casos de Uso
Herramientas necesarias: Visual Paradigm
Precedencias: ----------

Tabla 3 Diagrama de Casos de Uso y Jerarquía de Actores

MODELO DE DOMINIO
Duración estimada: 8 horas
Descripción: Elaboración del modelo de dominio, el cual representa las entidades cuyos datos se van a almacenar, así como las relaciones entre ellas.
Entradas: ----------
Salidas/Entregables: Modelo de Dominio
Herramientas necesarias: Visual Paradigm
Precedencias: Diagrama de Casos de Uso

Tabla 4 Modelo de Dominio

CASOS DE USO EXTENDIDOS
Duración estimada: 20 horas
Descripción: Elaboración del documento en el que se encuentran los casos de uso extendidos, en los que se detallan a fondo las funcionalidades y diagramas de flujo de los Casos de Uso.
Entradas: Diagrama de Casos de Uso
Salidas/Entregables: Casos de Uso Extendidos
Herramientas necesarias: Microsoft Word, Visual Paradigm
Precedencias: Diagrama de Casos de Uso

Tabla 5 Casos de Uso Extendidos

TRANSFORMACIÓN DE MD A BBDD
Duración estimada: 10 horas
Descripción: Elaboración del documento en el que se convierte el Modelo de Dominio realizado a una Base de Datos Relacional. En este caso se convertirá a XMLs, mapas y objetos.
Entradas: Modelo de Dominio
Salidas/Entregables: Archivos XML.
Herramientas necesarias: Microsoft Word y Visual Paradigm
Precedencias: Modelo de Dominio

Tabla 6 Transformación de MD a BBDD

DISEÑO E IMPLEMENTACIÓN DE MUNDOS
Duración estimada: 20 horas
Descripción: Elaboración de los mundos en los que se desarrollará el juego, así como de la programación para su futura interacción con el jugador. En primer lugar se desarrollará la parte gráfica con Tiled, para luego añadir las propiedades pertinentes a cada "tile" (por ejemplo, un cofre contiene o no un objeto). Después se procederá a cargar estos mundos con Android Studio y a programar sus colisiones e interacciones.
Entradas: Tilesets de texturas, Casos de uso Extendidos de la funcionalidad.
Salidas/Entregables: Mundos completos
Herramientas necesarias: Android Studio, LibGDX, Box2D, Tiled, TexturePacker
Precedencias: Captura de requisitos

Tabla 7 Diseño e Implementación de Mundos

DISEÑO E IMPLEMENTACIÓN DE ENEMIGOS
Duración estimada: 30 horas
Descripción: Elaboración de los enemigos que el jugador se encontrará en su sesión de juego. La parte de diseño consistirá en pensar las mecánicas sobre las que actuarán los enemigos (por ejemplo, que habilidades usan y con qué patrones, cuándo es posible o no atacarlos, cuánta vida tienen…) y en el diseño gráfico de estos mediante el uso de sprites gratuitos. La implementación consistirá en la programación de estos enemigos en los mundos previamente creados.
Entradas: Casos de uso Extendidos de la funcionalidad.
Salidas/Entregables: Enemigos implementados y visibles en los mundos
Herramientas necesarias: Android Studio, LibGDX, Box2D
Precedencias: Diseño e Implementación de Mundos

Tabla 8 Diseño e Implementación de Enemigos

IA DE ENEMIGOS E INTERACCIONES DEL JUGADOR
Duración estimada: 40 horas
Descripción: Elaboración de la inteligencia artificial de los enemigos. Se basará en la frecuencia con la que atacan, los distintos ataques y posicionamiento que muestran y su agresividad y forma de perseguir al jugador (si es que la tienen).
Entradas: Casos de uso Extendidos de la funcionalidad.
Salidas/Entregables: Los enemigos actúan de forma inteligente.
Herramientas necesarias: Android Studio, LibGDX, Box2D
Precedencias: Diseño e Implementación de Enemigos

Tabla 9 IA de Enemigos e Interacciones del Jugador

DISEÑO E IMPLEMENTACIÓN DE CIUDADES Y NPCs
Duración estimada: 30 horas
Descripción: Elaboración de las ciudades y los NPCs (non-playable characters). En primer lugar se diseñarán desde un punto de vista estético, utilizando sprites gratuitos y Tiled. Después se implementarán mediante Android Studio y se programará todo lo relacionado con las colisiones en el caso de las ciudades, y las distintas posibilidades y conversaciones en lo referente a los NPC.
Entradas: Tilesets de NPCs y ciudades, Casos de uso Extendidos de la funcionalidad.
Salidas/Entregables: Ciudades y NPCs completadas, con colisión y posibilidad de interacción con el jugador.
Herramientas necesarias: Android Studio, LibGDX, Box2D, Tiled, TexturePacker
Precedencias: IA de Enemigos e Interacciones del Jugador

Tabla 10 Diseño e Implementación de Ciudades y NPCs

FUNCIONALIDADES NECESARIAS
Duración estimada: 30 horas
Descripción: Elaboración de funcionalidades necesarias para una correcta experiencia a la hora de disfrutar del juego. Estas incluirán “Guardar Partida”, “Cargar Partida”, una selección de “Ajustes” y el cálculo de la puntuación a medida que se avanza en el juego. El guardado de partida será accesible en cualquier momento, mientras que la carga y selección de ajustes solo lo serán desde el menú principal.
Entradas: Casos de uso Extendidos de la funcionalidad.
Salidas/Entregables: Juego con posibilidad de guardar y cargar partida
Herramientas necesarias: Android Studio, LibGDX
Precedencias: Diseño e Implementación de Ciudades y NPCs

Tabla 11 Funcionalidades Necesarias

DISEÑO E IMPLEMENTACIÓN DE MENÚS
Duración estimada: 30 horas
Descripción: Elaboración de los menús que serán accesibles antes de comenzar la partida y durante la partida. Estos incluirán el “Menú de Inicio” (el primero que se ve al iniciar la partida), “Menú de Pausa” (mostrado durante la partida al pausar el juego) y “Menú de Fin de Partida” (mostrado cuando la partida llega a su fin). También se implementarán todos los menús referentes al inventario y estados del jugador, accesibles durante la partida.
Entradas: Casos de uso extendidos de la funcionalidad.
Salidas/Entregables: Juego con los menús implementados, mostrando su contenido y ejecutando las funcionalidades correctamente.
Herramientas necesarias: Android Studio, LibGDX
Precedencias: Funcionalidades Necesarias

Tabla 12 Diseño e Implementación de Menús

PORT A ANDROID
Duración estimada: 30 horas
Descripción: Port del juego a Android con el fin de que sea jugable en dispositivo móviles. Se añadirá el soporte necesario, que incluirá necesariamente flechas direccionales y botones táctiles para mover y ejecutar acciones del jugador.
Entradas: Casos de uso extendidos de la funcionalidad.
Salidas/Entregables: Port jugable en dispositivos Android.
Herramientas necesarias: Android Studio, LibGDX, Box2D
Precedencias: Diseño e Implementación de Menús.

Tabla 13 Port a Android

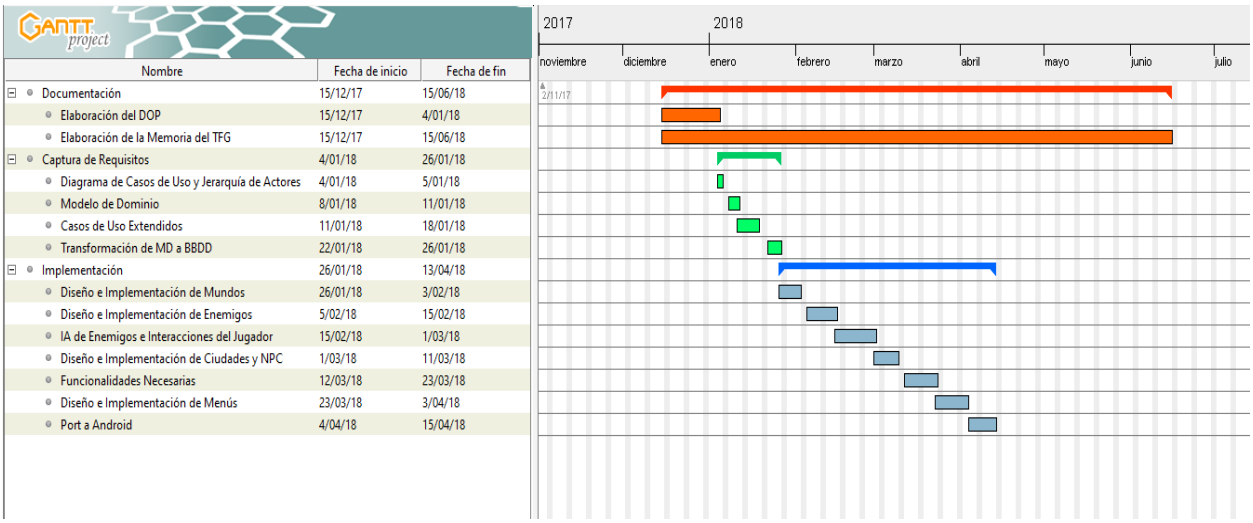
Planificación temporal

Ahora que se conocen las tareas que se van a llevar a cabo, se va a proceder a decidir cómo serán distribuidas en el tiempo. Teniendo en cuenta que:

- La carga total estimada de trabajo es de 361 horas
- La fecha de la realización de esta sección es 30 de Enero, y quedan por lo tanto 136 días para el 15 de Junio, la cual se considerará como fecha límite de realización del proyecto.
- Eliminando los sábados y domingos como día de trabajo, la carga diaria de trabajo debería ser de 3 horas/día o 18 horas/semana.
- Se empezó a realizar este documento el 15 de Diciembre

Para la realización de la planificación temporal se realizará un diagrama de Gantt semanal, dividido de la misma forma que la *División de la Carga de Trabajo*. De esta forma, encontraremos a la izquierda las 3 secciones globales: Documentación, Captura de Requisitos e Implementación, y cada una de ellas dividida en sus sub-tareas correspondientes. La duración de cada uno de estos elementos se representará a la derecha mediante el color empleado previamente en la *Descripción de Tareas.* La primera tarea será la *Elaboración del DOP,* y será seguida por el resto de tareas, las cuales se realizarán de forma individual. De esta forma, se espera a terminar una tarea para pasar a la siguiente, excepto en el caso de la *Elaboración de la Memoria del TFG*, la cual se extenderá durante todo el desarrollo del proyecto y será la última en finalizar.

La distribución se hará indicando las fechas de inicio y final estimadas y la distribución mensual que conlleva cada tarea. En el diagrama de Gantt se verán la fecha de inicio y final.



Nombre	Fecha de inicio	Fecha de fin
Documentación	15/12/17	15/06/18
Elaboración del DOP	15/12/17	4/01/18
Elaboración de la Memoria del TFG	15/12/17	15/06/18
Captura de Requisitos	4/01/18	26/01/18
Diagrama de Casos de Uso y Jerarquía de Actores	4/01/18	5/01/18
Modelo de Dominio	8/01/18	11/01/18
Casos de Uso Extendidos	11/01/18	18/01/18
Transformación de MD a BBDD	22/01/18	26/01/18
Implementación	26/01/18	13/04/18
Diseño e Implementación de Mundos	26/01/18	3/02/18
Diseño e Implementación de Enemigos	5/02/18	15/02/18
IA de Enemigos e Interacciones del Jugador	15/02/18	1/03/18
Diseño e Implementación de Ciudades y NPC	1/03/18	11/03/18
Funcionalidades Necesarias	12/03/18	23/03/18
Diseño e Implementación de Menús	23/03/18	3/04/18
Port a Android	4/04/18	15/04/18

Ilustración 7 Diagrama de Gantt

En el diagrama se observa que la fase de *Documentación* se extiende hasta el 15 de Junio, mientras que la de *Implementación* termina en Abril. Esto se debe a dos razones:

- Aunque se cumplan los plazos y se termine en la implementación en Abril, se seguirá modificando y puliendo la documentación hasta la fecha límite. De esta forma, el total de horas de esta fase se distribuirá a lo largo de toda la duración del proyecto.
- Es posible que haya retrasos en una o varias fases del proyecto. Esas semanas extra se verán como el retraso temporal que se puede asumir.

Al ser el diagrama mensual no se aprecia con detalle la duración de cada tarea. Por esta razón y con el fin de ver el detalle de cada tarea y el total de la carga de trabajo de forma resumida, se ha generado la siguiente tabla:

TAREA	DURACIÓN (Horas)
DOCUMENTACIÓN	
Elaboración del DOP	45
Memoria del TFG	60
CAPTURA DE REQUISITOS	
Diagrama de Casos de Uso y Jerarquía de Actores	8
Modelo de Dominio	8
Casos de Uso Extendidos	20
Transformación de MD a BBDD	10
MÓDULOS	
Diseño e Implementación de Mundos	20
Diseño e Implementación de Enemigos	30
IA de Enemigos e Interacciones del Jugador	40
Diseño e Implementación de Ciudades y NPCs	30
Funcionalidades Necesarias	30
Diseño e Implementación de Menús	30
Port a Android	30
DURACIÓN TOTAL ESTIMADA	**361**

Tabla 14 Resumen de duración total de tareas

Herramientas

A continuación se listarán y explicarán en detalle las herramientas que se van a usar en la elaboración de las distintas partes del proyecto. La sección se dividirá en las herramientas que se van a usar para la parte de *Documentación, Captura de Requisitos* y para la de *Implementación*:

Herramientas para la Documentación

- **Microsoft Office 2013**: Paquete de software ofimático para Windows. De él se usarán Microsoft Word 2013 para la realización de la memoria, Microsoft Excel 2013 para la realización de elementos como diagramas y gráficos, y Microsoft PowerPoint 2013 para la realización de la presentación del proyecto.

- **GanttProject**: Software de código abierto con licencia GPL empleado para la elaboración de diagramas de Gantt.

- **Dropbox**: Servicio de alojamiento de archivos en la nube. Se usará para guardar con seguridad el archivo sobre el que se va realizando la memoria, así como para su control de versiones.

Herramientas para la Captura de Requisitos

- **Visual Paradigm 13.2:** Herramienta UML empleada para la realización de distintos diagramas y esquemas que representan distintos modelos de la aplicación. Con él se realizará la *Captura de Requisitos* y *Análisis y Diseño*, generando así los Diagramas de Casos de Uso, Diagramas de Clases, Diagramas de Secuencia y Modelo de Dominio.

- **Microsoft Office 2013**

- **Dropbox**

Herramientas para la Implementación

- **Android Studio 2.3.1.:** Entorno de desarrollo integrado oficial para la plataforma Android. Basado en Intelli J IDEA y programado en Java, se usará para llevar a cabo el desarrollo de la aplicación en Java y su posterior port a Android. Gracias a su compilación en tiempo real y su herramienta de visualización de la aplicación en un dispositivo móvil virtual, permitirá realizar las pruebas de manera visual y rápida.

- **LibGDX:** Framework de código libre, programado mayoritariamente en Java empleado para el desarrollo de juegos multiplataforma usando el mismo código base. En este proyecto se desarrollará para PC y Android.

  Esta herramienta fue escogida con varios aspectos en mente. En primer lugar, y como se verá con más profundidad en la sección de *Antecedentes*, es una herramienta que es globalmente usada en la actualidad para el desarrollo de juegos y tiene, por lo tanto, una extensa documentación y comunidad online, con las facilidades que esto supone. Se analizó el uso de otras herramientas similares como Slick2D, LWJGL o Unity 3D, estando las dos primeras ligeramente desactualizadas y con falta de soporte, además de carecer de la versatilidad y potencia de LibGDX. En el caso de Unity 3D, a pesar de su popularidad, su lenguaje principal (C++) y el hecho de que no se centrara en el diseño bidimensional fueron las razones por las que se descartó su uso.

  LibGDX ofrece una serie de características que se ajustan ampliamente a las necesidades de este proyecto. Aunque la implementación y detalle de todas estas se verá con más detalle en

secciones posteriores, las más relevantes son:

  - Asistencia a la hora de generar y tener en cuenta las *physics* (elementos físicos como la gravedad, rozamiento o densidad, entre otros) de los mundos generados.
  - Posibilidad de añadir con facilidad música y sonidos a eventos particulares.
  - Generación de animaciones en base a *sprites*.
  - Transición entre estados (pantallas).
  - Compatibilidad con la herramienta visual de diseño de mapas *Tiled*.
  - Posibilidad de usar diferentes extensiones que añaden funcionalidades adicionales a la biblioteca. Para este proyecto, se usarán las siguientes:
    - *GdxAI*, framework que antes pertenecía a la biblioteca principal. Se usa principalmente para diseñar la inteligencia artificial de los personajes del juego, como la toma de decisiones, movimiento y búsqueda de la mejor ruta.
    - *Box2D*, motor de *physics* que gestionará todos los elementos y magnitudes físicas que se vayan a implementar, además de ser el encargado de dibujar las líneas de *debugging* en el mapa a la hora de realizar pruebas y búsqueda de errores.

- **Tiled:** Herramienta visual que se usa como editor de mapas para videojuegos. Tomando una seria de *spritesheets* (colección de *sprites*) de uno o varios archivos, permite crear mapas arrastrando y manipulado estos *sprites*. Dividiendo el mapa en cuadrados o rectángulos de la dimensión deseada, permite añadir a cada *sprite* valores ocultos, numéricos o de texto, que se usarán luego en la programación para identificar propiedades de este (por ejemplo, si un cofre tiene o no un objeto en su interior). Gracias a la compatibilidad con LibGDX, permite cargar y manipular los mapas desde el código tal y como han sido creados de manera visual. Esta facilidad de uso y conveniencia fue la razón principal por la que se escogió esta herramienta.

- **TexturePacker:** Software empleado para la creación de *spritesheets*. Su funcionamiento consiste en la selección de dos o más *sprites*, contenidos en archivos diferentes, para su posterior unión en un nuevo archivo individual. Tener muchos *sprites* juntos en un mismo archivo y una menor cantidad de archivos es vital para que, al cargarlos, el rendimiento de la aplicación no se vea comprometido.

- **GitHub:** Forja de alojamiento de proyectos y control de versiones los mismos. Se usará con estos dos fines para este proyecto.

Gestión de riesgos

Con el fin de evitar pérdidas durante la realización del proyecto, tanto económicas como de tiempo, se va a generar un plan de gestión de riesgos. Se entenderá un riesgo como un evento que tendría un impacto sobre la planificación temporal del proyecto, retrasándola. El objetivo de este plan no es solo analizar los efectos que un riesgo puede tener sobre el trabajo, sino también como prevenirlos. Se usarán los siguientes porcentajes para describir la probabilidad de que se dé un riesgo:

Probabilidad	Porcentaje
Improbable	0 - 25 %
Poco probable	25 – 50 %
Probable	50 – 75 %
Muy probable	75 – 100 %

Tabla 15 Probabilidad de Riesgos

De la misma forma, se valorará también el impacto en días que tendría un riesgo. De nuevo, lo visualizamos mediante la siguiente tabla:

Gravedad	Impacto (Días)
Muy leve	Menos de un día
Leve	1 día
Medio	2 – 5 días
Grande	5 – 10 días
Muy grande	10 días o más

Tabla 16 Impacto de Riesgos

Ahora que se tienen en cuenta la probabilidad y el impacto que se suponen los riesgos, se ha de generar una prevención y un plan de contingencia para cada uno de ellos. En lo referente al desarrollo y en general a la parte tecnológica de este, los riesgos pueden ser de dos tipos: de *software* y de *hardware*. Al estar en muchos casos relacionados, se tratarán en un solo apartado al que se le llamará *Riesgos de Desarrollo*. Existe también la posibilidad de que se modifiquen elementos de la planificación realizada, como podría ser la planificación temporal, sprints, o especificaciones del proyecto. Este tipo de riesgos se recogerán bajo la categoría *Riesgos de Planificación*.

Sin embargo, no todos los riesgos son de naturaleza tecnológica o técnica. El propio desarrollador del proyecto puede quedar indispuesto por una o varias razones, y estas serán analizadas en la sección *Riesgos Personales*. Por último, no se debe olvidar que se debe ofrecer soporte y garantías al usuario una vez el desarrollo ha terminado, y existen una serie de riesgos que se pueden dar una vez el producto ha sido ya generado. Estos pueden incluir la desaparición del soporte a las herramientas con las que se ha desarrollado el producto o riesgos de seguridad del producto final. A estos se les llamará *Riesgos del Producto Generado*.

Riesgos de Planificación	Riesgos de Desarrollo
Pl 1. Planificación Temporal Incorrecta	*Des 1. Cambio de versiones*
Pl 1.1. Gantt	*Des 1.1. Android Studio*
Pl 1.2. Sprints SCRUM	*Des 1.1.2. LibGDX*
Pl 2. Cambio de las especificaciones del proyecto	*Des 2. Infección por virus*
	Des 3. Rotura o pérdida del equipo
Riesgos Personales	**Riesgos del Producto Generado**
Per 1. Enfermedad	*Pg 1. Tecnología sin soporte*
Per 2. Empleo	*Pg 2. Acceso y modificación de archivos*

Tabla 17 Conjunto de Riesgos

Ahora que se han identificado los riesgos, se procederá a analizarlos de forma individual. Como se ha mencionado anteriormente, el análisis constará de cuatro partes: prevención, plan de contingencia, probabilidad e impacto. Como es lógico, se intentará que los riesgos que se describen nunca lleguen a producirse, por lo que un buen plan de prevención es de suma importancia. Sin embargo, es probable que aun así en algún caso termine por darse algún riesgo, por lo que tener un plan de contingencia adecuado tiene la misma relevancia con el fin de tratar de que el impacto se reduzca en la medida de lo posible. Los cuatro apartados del análisis se han adecuado a la naturaleza del proyecto, especificando con detalle las medidas que se llevarán a cabo.

Riesgos de planificación

Pl 1. Planificación Temporal Incorrecta	
Pl 1.1. Gantt	
Prevención	Realizar un diagrama de Gantt lo más preciso posible. Dado que la probabilidad de que se dé el riesgo es muy alta, sería conveniente planificar con un colchón de tiempo de unos días para cada tarea.
Plan de contingencia	Replantear la planificación temporal lo antes posible.
Probabilidad	Muy probable
Impacto	Medio
Pl 1.2. Sprints Scrum	

Prevención	Planificar los sprints con la mayor precisión posible y siguiendo la línea del diagrama de Gantt.
Plan de contingencia	Ajustar los sprints posteriores en base a los días perdidos en el anterior.
Probabilidad	Probable
Impacto	Bajo

Tabla 18 Planificación Temporal Incorrecta

Pl 2. Cambio de las especificaciones del proyecto	
Prevención	Realizar el proyecto de forma modular y seguir el plan de desarrollo iterativo lo máximo posible.
Plan de contingencia	Replanteamiento de las modificaciones.
Probabilidad	Probable
Impacto	Medio - Grande

Tabla 19 Cambio de las especificaciones del proyecto

Riesgos de desarrollo

Des 1. Cambio de versiones	
Des 1.1. Android Studio	
Prevención	Teniendo en cuenta que LibGDX no funciona con versiones posteriores a AndroidStudio 3.0, no actualizar Android Studio a dichas versiones.
Plan de contingencia	*Downgrade* de Android Studio e importación de una copia de seguridad de la última versión del proyecto.
Probabilidad	Improbable
Impacto	Medio
Des 1.2. LibGDX	
Prevención	Prestar atención a los cambios en las nuevas versionesde LibGDX. Si las modificaciones fueran positivas para el proyecto, asegurarse de que no surgirán conflictos y actualizar LibGDX. Si las nuevas funcionalidades de LibGDX no fueran útiles para el proyecto, continuar con la misma versión que se haya estado usando.

Plan de contingencia	Si se han dado conflictos, realizar un *downgrade* de LibGDX e importar una copia de seguridad del proyecto.
Probabilidad	Improbable
Impacto	Leve - Medio

Tabla 20 Cambio de Versiones

Des 2. Infección por virus	
Prevención	Realización periódica de copias de seguridad del sistema, además de las copias del proyecto. Además, se mantendrá el anti-virus actualizado.
Plan de contingencia	Formateo y restauración desde la copia de seguridad.
Probabilidad	Improbable
Impacto	Leve

Tabla 21 Infección por virus

Des 3. Rotura o pérdida del equipo	
Prevención	Almacenar todo el trabajo realizado en la nube, ya sea en Dropbox o GitHub. Tomar precauciones con el equipo de trabajo, no exponiéndolo a elementos nocivos como líquidos, lugares altos…
Plan de contingencia	Emplear otro ordenador. Si no se puede disponer de uno en casa, usar los ordenadores de la universidad. Restaurar las copias de seguridad.
Probabilidad	Improbable
Impacto	Medio

Tabla 22 Rotura o pérdida del equipo

Riesgos personales

Per 1. Enfermedad	
Prevención	Evitar exponerse a agentes de riesgo en lo referente a la salud.
Plan de contingencia	Centrarse en recuperarse lo antes posible, y compensar las horas pérdidas con horas extra.
Probabilidad	Probable
Impacto	Medio

Tabla 23 Enfermedad

Per 2. Empleo	
Prevención	Escoger un empleo que permita compaginar las horas de trabajo con las de realización del proyecto.
Plan de contingencia	Compensar el tiempo dedicado a diario, ya sea trabajando un día más horas o cambiando el momento de trabajo del día.
Probabilidad	Muy probable
Impacto	Leve

Tabla 24 Empleo

Riesgos del producto generado

Pg 1. Tecnología sin soporte	
Prevención	Prestar atención a la frecuencia con la que se realizan actualizaciones de la tecnología utilizada, así como del tamaño de la comunidad de desarrolladores que la usa.
Plan de contingencia	Actualizar a herramientas más modernas y con más soporte.
Probabilidad	Improbable
Impacto	Muy grande

Tabla 25 Tecnología sin soporte

Pg 2. Acceso y modificación de archivos	
Prevención	Este riesgo se tratará de forma excepcional. Al tratarse de un juego de un solo jugador, modificar los archivos para "hacer trampas" no supone una ventaja competitiva ni un problema de seguridad para nadie, simplemente afectaría a la experiencia del propio jugador que está modificando los archivos. Por esta razón, toda la prevención que se llevará a cabo consistirá en usar la mayor cantidad posible de métodos y atributos privados, pero no se encriptarán los archivos, además de por lo mencionado, para no perjudicar el rendimiento del juego.
Plan de contingencia	-
Probabilidad	Improbable
Impacto	Muy leve

Tabla 26 Acceso y modificación de archivos

Valoración de riesgos

Ahora que se han analizado la probabilidad e impacto de cada uno de los riesgos, hay que ver, teniendo ambos factores en consideración, que valoración global le vamos a dar a cada uno. Esta valoración reflejará la importancia general que se le da al riesgo, y en base a ella se decidirá el seguimiento y consideración que se le dará. Este seguimiento se hará en forma de días, de la siguiente manera:

Valoración	Seguimiento
Muy leve	Cada mes
Leve	Cada 15 días
Normal	Cada 10 días
Importante	Cada semana
Muy importante	Cada 3 días o menos

Tabla 27 Valoración de riesgos

Valoramos los riesgos en la siguiente tabla:

Riesgos de Planificación	Probabilidad	Impacto	Valoración
Pl 1.	*Muy probable*	*Medio*	*Importante*
Pl 2.	*Probable*	*Medio - Grande*	*Importante*
Riesgos de Desarrollo	Probabilidad	Impacto	Valoración
Des 1.	*Improbable*	*Medio*	*Leve*
Des 2.	*Improbable*	*Leve*	*Muy leve*
Des 3.	*Improbable*	*Medio*	*Muy leve*
Riesgos Personales	Probabilidad	Impacto	Valoración
Per 1.	*Probable*	*Medio*	*Normal*
Per 2.	*Muy probable*	*Leve*	*Normal*
Riesgos de Producto Generado	Probabilidad	Impacto	Valoración
Pg 1.	*Improbable*	*Muy grande*	*Normal*
Pg 2.	*Improbable*	*Muy leve*	*Muy leve*

Tabla 28 Valoración de riesgos global

Se ha observado que las copias de seguridad de diferentes elementos del proyecto son un plan de contingencia que se repite en numerosas ocasiones. Por ello, se realizará una copia de seguridad del sistema y proyecto cada 15 días, el cual se verá complementado por la seguridad que ofrecen Dropbox y GitHub.

Evaluación económica

Para esta sección se procederá a calcular tres aspectos económicos del proyecto: cuánto dinero se va a gastar en su realización, en su mantenimiento posterior y qué estrategias se van a emplear para la recuperación de esos gastos y la generación de ingresos.

En primer lugar, se tendrá cuenta el coste del transporte. Aproximadamente la mitad de las horas empleadas en la realización del proyecto se dan en casa, mientras que la otra mitad lo hacen en la universidad. Supongamos que se acude a la universidad una media de 3 días semanales, habiendo que hacer uso de los servicios de tren y metro:

Descripción	Coste Mensual	Coste proyecto
Coste del servicio de Euskotren	21,60 €	129,60 €
Coste del servicio del Metro de Bilbao	16,80 €	100,80 €
Total transporte	**38,40 €**	**230,40 €**

Tabla 29 Gastos de transporte

En segundo lugar, se valorará el coste de las licencias de software. Para este cálculo se tendrá en cuenta la amortización, además de que la vida media del software es de unos 3 años. Se realizan los cálculos de la siguiente manera:

Descripción	Coste	Amortización	Coste proyecto
Windows 10	119 €		19,83 €
Microsoft Office 2013	220 €	3 años	36,66 €
*Visual Paradigm 13.2**	0 €		0 €
Total software	**339 €**		**56,49 €**

Tabla 30 Gastos de licencias software

**La licencia de Visual Paradigm 13.2 se obtiene a través de la ofrecida a alumnos por la UPV*

También debemos valorar el valor del hardware que se emplea en la realización del proyecto. En ella se usará el siguiente equipo: un ordenador de sobremesa valorado en 2.000 euros, un Smartphone (Samsung Galaxy Note 4) valorado en 600 euros, y un ordenador portátil valorado en 800 euros. Se vuelven a realizar los cálculos con las amortizaciones medias correspondientes:

Descripción	Coste	Amortización	Coste proyecto
Ordenador de sobremesa	2.000 €	2 años	500 €
Ordenador portátil	800 €	3 años	133,33 €
Samsung Galaxy Note 4	600€	3 años	100 €
Total hardware	**3.400 €**		**733,33 €**

Tabla 31 Gastos hardware

Según la planificación temporal, un total de 105 horas serán dedicadas a tareas de *documentación*, mientras que 256 lo serán a tareas de *análisis, diseño e implementación*. Suponiendo que la hora de *documentación* cuesta 25 €/hora, y la de *análisis, diseño e implementación* 40€/hora, el coste de desarrollo sería el siguiente:

Descripción	Horas	Coste/Hora	Coste proyecto
Documentación	105 Horas	25 €/Hora	2.625 €
Análisis, diseño e implementación	256 Horas	40€ / Hora	10.240 €
Total hardware	**3.400 €**		**12.865 €**

Tabla 32 Gastos de desarrollo

Además de todos estos gastos, se deben de tener en cuenta los gastos diarios indirectos como luz, agua o electricidad. Se ha calculado que estos gastos suponen en total unos 100€ mensuales, por lo que el gasto en los 6 meses de proyecto ascendería a 600€. Ahora, los gastos totales del proyecto se verían de la siguiente manera:

Gasto	Coste
Gastos de transporte	230,40 €
Gastos de licencias software	56,49 €
Gastos de hardware	733, 33 €
Gastos de desarrollo	12.865 €
Gastos indirectos	600 €
Gastos totales	**14.485,22 €**

Tabla 33 Gastos totales

Ahora que sabemos a cuánto ascienden los gastos totales del proyecto, debemos plantearnos cómo vamos a no solo recuperar esta inversión, sino también a generar beneficios.

Es importante tener en cuenta que, por encima de una aplicación móvil, se está tratando con un videojuego, y por lo tanto el modelo de negocio debe ser adecuado a esto. Para *Phoenix*, se plantean las siguientes formas de generar ingresos:

- Publicidad
- Cash Shop
- Precio de venta del propio juego

En muchas aplicaciones móviles nos encontramos con publicidad, ya sea cada vez que se abre la app o durante su uso. Estas apariciones de anuncios, o *impresiones*, pueden ser muy lucrativas, generando de media 1,50€ por cada 1000 impresiones mediante el uso de la plataforma publicitaria para iOS y Android *adMob,* teniendo otras números similares [6]. Sin embargo, irían contra la filosofía de Phoenix. Como ya se ha explicado, este proyecto está basado en la valoración de los videojuegos tal y como se entendían antes, dando énfasis a una experiencia completa, rica e ininterrumpida. El uso de anuncios invasivos y frecuentes dañaría la inmersión y experiencia del usuario, y queda por lo tanto desechado.

Otro modelo de negocio muy extendido es el uso de una *Cash Shop*. Consiste en implementar una "tienda online" dentro del propio juego, en la que se venderán por dinero real elementos relacionados con este (intentos para superar una pantalla, objetos que mejoren en características a los obtenibles dentro del propio juego, elementos cosméticos, etc.). Volviendo de nuevo a la filosofía que se intenta seguir, esta idea es muy controvertida. Por una parte se pretende ofrecer la experiencia completa e ininterrumpida, pero por otra estas compras son opcionales y es posible disfrutar del 100% del juego sin ni siquiera haber visto la *Cash Shop*. Por lo tanto, se buscará encontrar lo mejor de ambos mundos de la siguiente manera: se ofrecerá una *Cash Shop* en la que tan solo se vendan elementos:

- Cosméticos
- De *QoL* (Quality of Life), es decir, que reduzcan el tiempo total requerido para realizar una tarea que pueda ser repetitiva (por ejemplo, que permita subir de nivel un 20% más rápido).

Esta tienda irá por lo tanto dirigida a auténticos fans del juego, además de a jugadores que no dispongan del suficiente tiempo diario para jugar. Es importante entender que estos elementos supondrían un problema si hubiera algún elemento competitivo online, ya que pondrían en desventaja al jugador que no paga. Sin embargo, al no haber elementos competitivos, es algo que afecta únicamente a la experiencia que el propio usuario desea tener, y por lo tanto es aceptable.

Teniendo en cuenta que se desecha la idea de la publicidad y que se va a implementar una *Cash Shop* muy filtrada y limitada, se le pondrá un precio base al juego. Se considera que, dada la calidad de este y los precios actuales del mercado (lo cual se analiza en profundidad en la sección de *Antecedentes*), un precio aceptable para Phoenix sería de 0,99€.

Por lo tanto, ¿cuántos usuarios hacen falta para recuperar la inversión realizada? Supongamos que de cada 100 usuarios, 5 hacen uso de la *Cash Shop*. Supongamos además que, de forma orientativa, el precio medio de los elementos de la tienda es de 1,50 €, y los usuarios que hacen uso de ella lo hacen una vez por semana durante un mes. Es importante entender que, por la naturaleza del juego, no se va a jugar de forma constante durante un periodo largo de tiempo (por ejemplo, años). Se considerará que el periodo de vida del juego será de un mes, por lo que los beneficios mensuales de cada usuario serán también los beneficios totales que ese usuario ha aportado:

5 de cada 100 usuarios gastan dinero en 10 elementos de la Cash Shop 4 veces al mes de media
1 usuario → 1,50 € x 10 x 4 = 60 € / mes
5 usuarios (equivalente a 100 usuarios) → 60 € x 5 = 300 € / mes
100 usuarios generan 300 € / mes mediante la Cash Shop

Además, esos 100 usuarios habrán comprado el juego → 100 x 0,99 = 99 € / mes en ventas

Por lo tanto, ¿cuántos usuarios hacen falta para recuperar la inversión?
Si cada 100 usuarios suponen 399 € de beneficio total →
*→ (14.485,22 x 100) / 399 = **3.630 usuarios***

Ilustración 8 Usuarios necesarios

De esta forma, a partir de los 3.630 usuarios la inversión habría sido recuperada y Phoenix comenzaría a generar beneficios.

ANTECEDENTES

Como ya se ha visto en la *Introducción*, el mercado de los videojuegos ha cambiado en numerosos sentidos. Lo ha hecho, en primer lugar, orientándose hacia ser jugado en un dispositivo móvil, y en segundo lugar, en el género de los videojuegos. Estas son las perspectivas desde las que se van a enfocar los antecedentes.

En primer lugar, podemos encontrar varios tipos de compañías lanzando juegos de rol al mercado móvil, haciéndolo cada una desde enfoques distintos, aunque con elementos comunes:

- **Square Enix**: Gigante de la industria desde los años 80, es una de las empresas que más videojuegos de rol ha desarrollado y ha servido como inspiración de muchísimos juegos posteriores. La mayoría de los que se han desarrollado y se desarrollan en dos dimensiones (como Phoenix) imitan las bases de su juego estrella, *Final Fantasy*, especialmente de sus ediciones más exitosas, que vieron la luz en los años 90.

  Gracias al cambio de mercado que se ha descrito, Square Enix vio hace unos años una oportunidad: el relanzamiento de sus viejas glorias en Android e iOS, con interfaces y gráficos ligeramente mejorados. Esta idea resultó exitosa, colocando a la empresa entre los desarrolladores *top* en la *Play Store* de Android y generando un 49% de crecimiento en los beneficios del año fiscal que terminó en Marzo de 2015, por ejemplo [7]. De esta forma, la compañía no solo ha repetido esta práctica con la mayoría de sus juegos exitosos, sino que decidió destinar aún más recursos a este enfoque, haciendo el desarrollo para móviles tanto o más importante que el desarrollo convencional para consolas.

  Aunque criticado por algunos, el modelo de negocio de Square Enix ha resultado fructífero. Sus primeros juegos en la tienda, aunque desarrollados en los 90 y principios del siglo XXI, son de alta calidad y no contemplan la idea de compras *in-game*. Es por esto que el precio de estos es considerablemente elevado en comparación con los precios de otras compañías: oscilan entre los 10 y los 20 euros en su mayoría. Sus desarrollos más recientes siguen el enfoque moderno, siendo más cortos, de naturaleza adictiva, y con precios más bajos (entre 1 y 10 euros) y opción de *Cash Shop*.

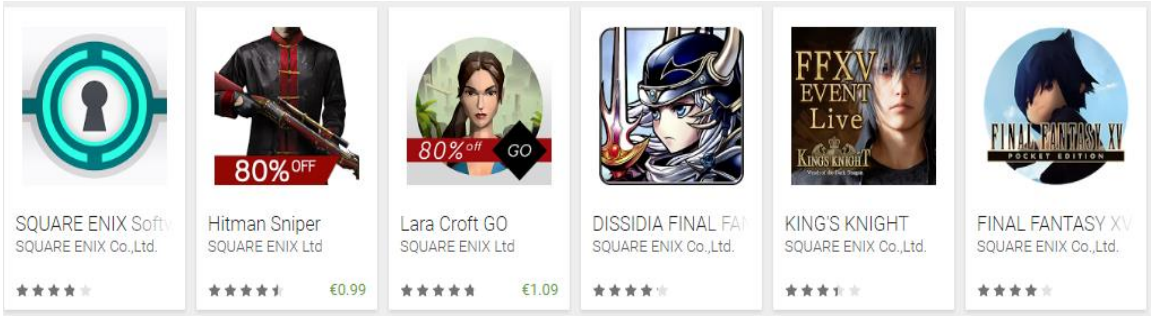


Ilustración 9 Square Enix - Enfoque Moderno

Se observan en la Ilustración 9 algunos de los juegos de Square que siguen el enfoque moderno. Juegos cortos, sencillos y de precios bajos que complementan a juegos ya existentes. Todos ellos tienen la posibilidad de realizar compras in-game.

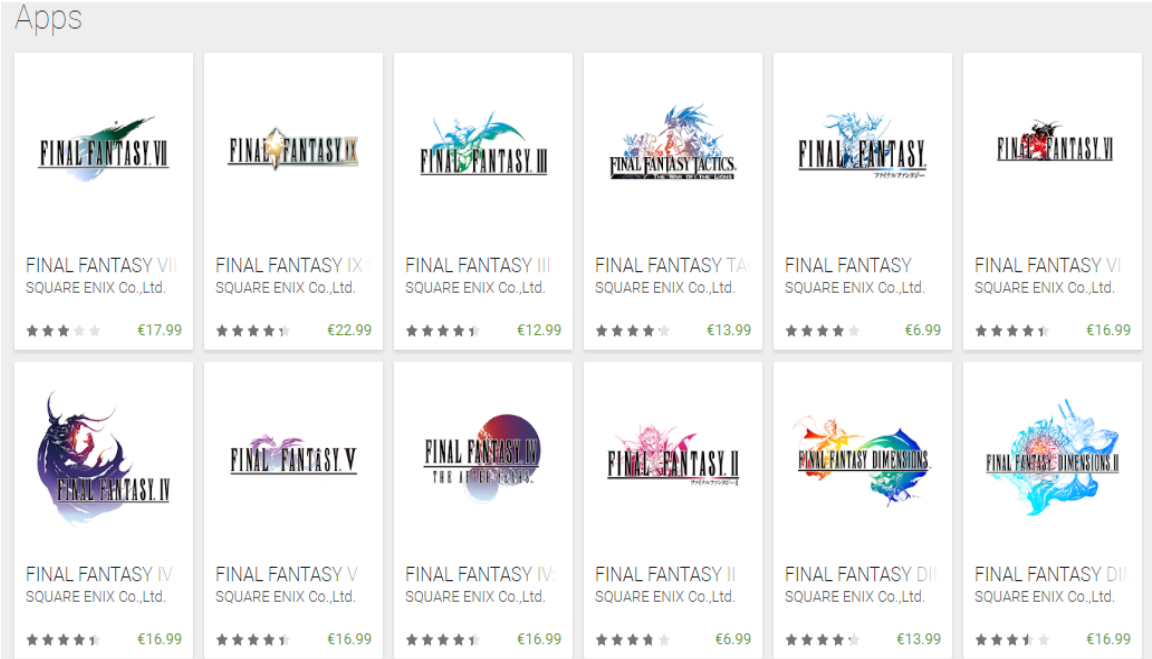


Ilustración 10 Square Enix – Relanzamientos

Enfoque muy distinto al descrito en la Ilustración 9. Al no haber Cash Shop en estos juegos y ser de más calidad, los precios oscilan entre los 10 y 20 euros en su mayoría.

- **Nexon**: De nuevo, otro gigante de la industria, centrado sobre todo en el desarrollo de juegos online. Al igual que Square Enix, Nexon centra cada vez más su desarrollo en títulos para Android e iOS, especialmente en los mercados que más los demandan (Asia en particular), pero no es esta la razón por la que se ha decidido analizar esta empresa. Se ha elegido porque esta es una compañía que, en su mayoría, aplica un modelo *Free to Play* o gratuito a la gran mayoría de sus juegos, teniendo todos ellos un fuerte énfasis en la *Cash Shop.* Aunque sea un modelo mal visto por la comunidad de jugadores, indudablemente genera beneficios, habiendo generado un crecimiento estable anual en el caso de Nexon, especialmente en el último cuatrimestre de 2017, reportando los mayores beneficios de cualquier cuatrimestre en su historia [8] [9].

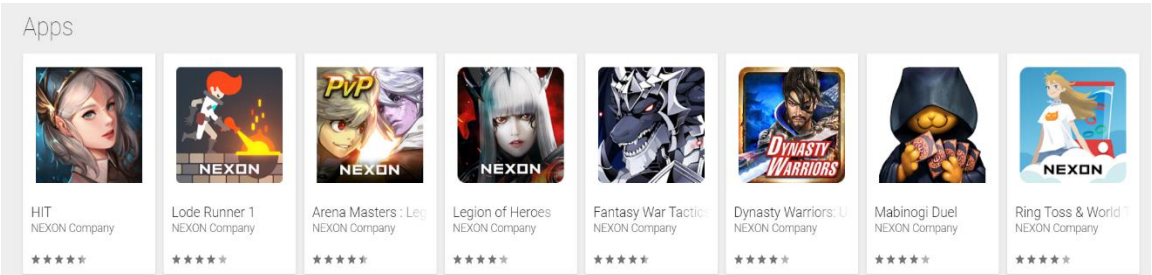


Ilustración 11 Nexon - Juegos Free to Play

Como se ve en la Ilustración 11, los juegos más relevantes en la búsqueda de "Nexon" siguen todos un modelo Free to Play.

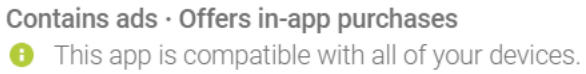


Ilustración 12 Nexon - Anuncios y Cash Shop

Sin embargo, todos ellos hacen uso de anuncios y compras in-game de forma bastante más intrusiva.

- **Kemco**: Contrastando con los compañías previamente mencionadas, Kemco es más pequeña y orientada hacia un nicho de mercado distinto. Aunque fue formada en los años 80 como desarrolladora de juegos para consolas, desde la aparición de los Smartphone se les conoce únicamente por sus videojuegos para estos.

  El análisis de esta compañía resulta especialmente interesante, ya que sus productos son los que más similitud guardan con lo que se pretende hacer en Phoenix. Realizan juegos mayoritariamente en 2D (con algún elemento en 3D), basados en el uso de *sprites* e intentando emular el estilo empleado en los años 90. Se centran en una historia y en la experiencia del jugador, por lo que sus anuncios y *Cash Shop* no son frecuentes o intrusivos, pero los incluyen de forma sutil para generar beneficios. Por esta razón, los precios de sus productos oscilan entre 0,99 € y 8 €, habiendo alguna excepción gratuita. Es por estas razones que Kemco ha servido como modelo a la hora de plantear la *Evaluación Económica* de Phoenix, ajustándose su filosofía a lo que se pretende hacer.

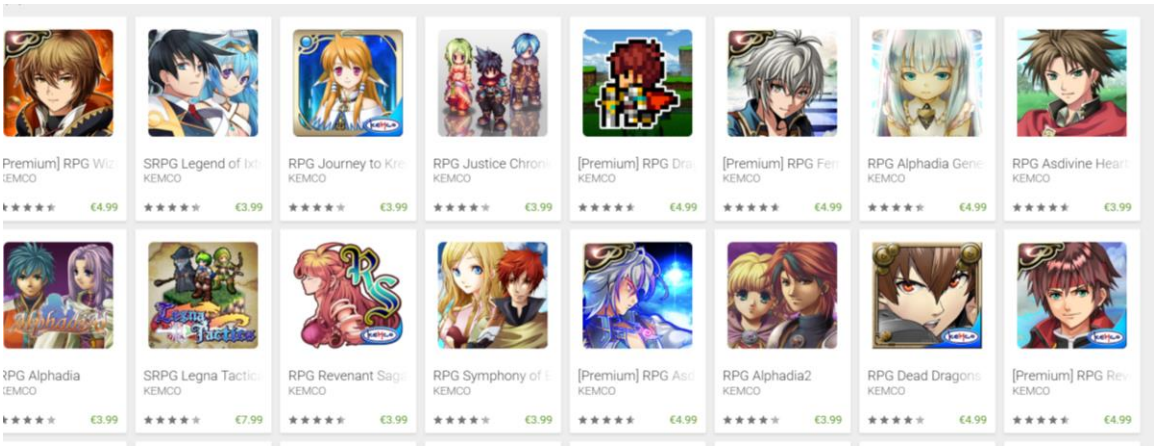


Ilustración 13 Kemco - Rango de Precios

En la Ilustración 13 se ve el rango de precios mencionado. Esto sirvió, de forma orientativa, para decidir el precio final de Phoenix.

CAPTURA DE REQUISITOS

Modelo de Casos de Uso

En esta sección se representará el modelo de casos de uso. Es importante destacar que, al ser *Phoenix* un juego únicamente jugado por el usuario final, la jerarquía de actores es bastante simple: tan solo existe el *usuario*. El siguiente diagrama representará las distintas funcionalidades del juego que el usuario debe ser capaz de llevar a cabo:

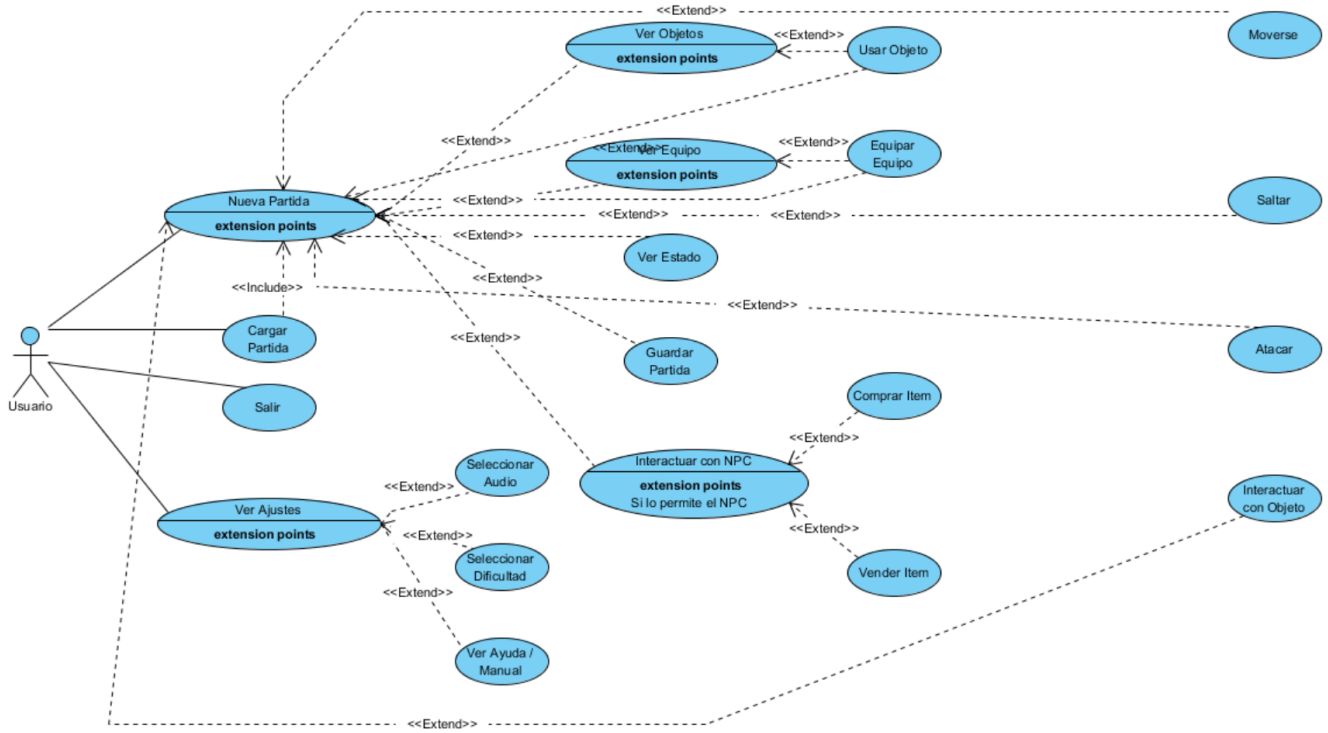


Ilustración 14 Modelo de Casos de Uso

Casos de Uso Extendidos

En los casos de uso extendidos se presentarán cada una de las funcionalidades mostradas arriba de forma mucho más precisa, detallando sus características, flujo de eventos e interfaz gráfica. Ya que el tamaño de esta sección es notablemente alto, se añadirá en otra sección al final del documento a la que se ha llamado *Anexo I*.

Modelo de Dominio

En esta sección se mostrará el modelo de dominio, el cual reflejará el tratamiento de los datos de la aplicación.

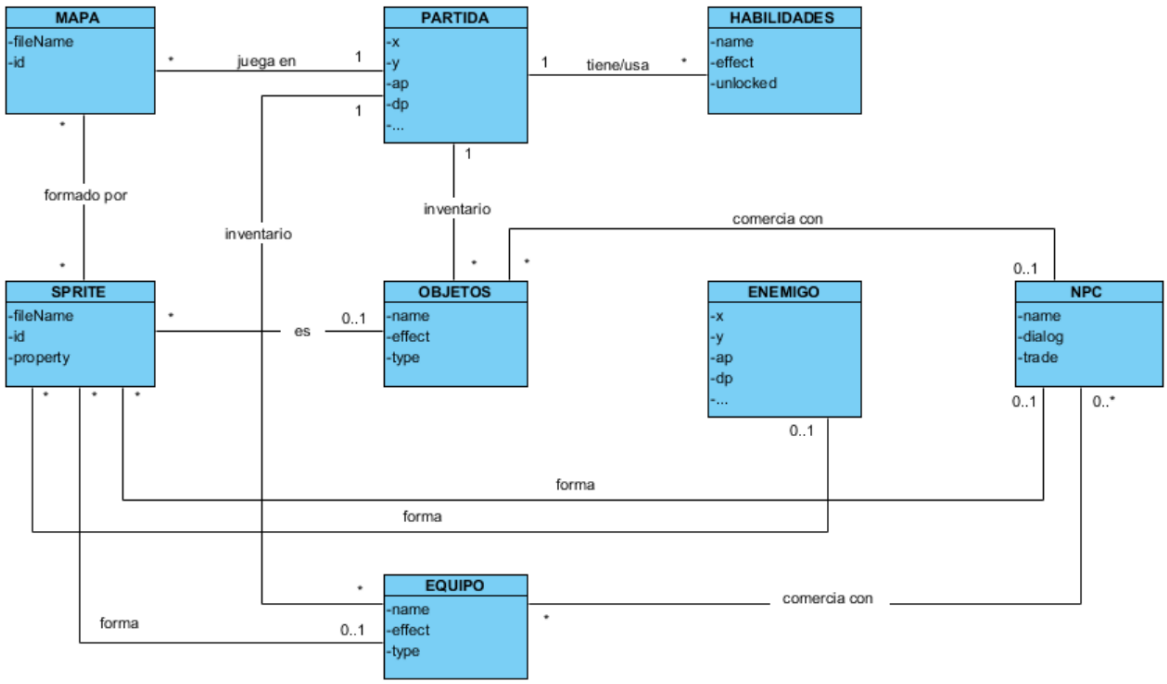


Ilustración 15 Modelo de Dominio

Entidades

A continuación se describirá cada entidad. Se destacarán los atributos más relevantes:

- **PARTIDA**: Representa al jugador que se está controlando en la partida actual. Guarda toda la información referente a este y al progreso de la partida.

- **MAPA:** Mapa en el que se está controlando al jugador. Está formado por varias colecciones de Sprites, y cada mapa tiene un identificador único.

- **SPRITE:** Pequeña imagen que se usará como texturas del mapa o parte de las animaciones de los elementos animados y dinámicos del juego. Cada uno tiene un identificador único, además de un atributo *Property* que indicará cualidades únicas del sprite (por ejemplo, si contiene un cofre un objeto o no)

- **OBJETO:** Objeto usable que guardará el jugador en su inventario. Tendrá una descripción, un efecto, y una efectividad (representada de forma numérica). Su animación se hace mediante sprites.

- **EQUIPO:** Pieza de equipo que el jugador se puede equipar. Sus atributos serán los estados que le sube al jugador, además de un atributo *type* que indicará dónde se equipa. Su animación se hace mediante sprites.

- **ENEMIGO:** Enemigo con el que el jugador pelea. Se almacenan todos sus estados, que influenciarán el desarrollo de la batalla con el jugador, además de sus coordenadas de aparición inicial. Su animación se hace mediante sprites.

- **NPC:** Non-Playable-Character, personaje con el que el jugador puede interactuar. Se almacenará su *Diálogo*, si *Compra* o *Vende,* y qué *Objetos* y *Equipo* vende. Su animación se hace mediante sprites.

- **HABILIDAD:** Habilidad que puede usar el jugador. De ella se almacena su *Efectividad*, y si ha sido *Desbloqueada* o no.

Relaciones

Se detallan a continuación las relaciones que se representan en el modelo de dominio:

- **JUEGA EN:** Relación de 1 a N entre PARTIDA y MAPA.

    - El jugador puede jugar en varios mapas, y un mapa solo puede ser jugador por un jugador.

- **TIENE/USA:** Relación de 1 a N entre PARTIDA y HABILIDAD.

    - El jugador puede tener y usar varias habilidades, y una habilidad solo puede ser usada por un jugador.

- **INVENTARIO_OBJ:** Relación de 1 a N entre PARTIDA y OBJETO.

    - Un jugador puede tener varios objetos en el inventario, y un objeto solo puede ser poseído por un jugador.

- **INVENTARIO_EQU:** Relación de 1 a N entre PARTIDA y EQUIPO.

    - Un jugador puede tener varias piezas de equipo en su inventario, y una pieza de equipo solo puede ser poseída por un jugador.

- **FORMA_OBJ:** Relación de 0..1 a N entre SPRITE y OBJETO.

    - Un sprite puede formar un objeto (o ninguno), y un objeto está formado por varios sprites que forman una animación.

- **FORMA_EQU:** Relación de 0..1 a N entre SPRITE y EQUIPO.

    - Un sprite puede formar una pieza de equipo (o ninguna), y una pieza de equipo está formada por varios sprites que forman una animación.

- **FORMA_NPC:** Relación de 0..1 a N entre SPRITE y NPC.

    - Un sprite puede formar un NPC (o ninguno), y un NPC está formado por varios sprites que forman una animación.
- **COMERCIA CON_OBJ:** Relación de 0..1 a N entre NPC y OBJETO.
    - Un NPC comercia con varios objetos, y un objeto es vendido por uno o ningún NPC.
- **COMERCIA CON_EQU:** Relación de 0..1 a N entre NPC y EQUIPO.
    - Un NPC comercia con varias piezas de equipo, y una pieza de equipo es vendida por uno o ningún NPC.
- **FORMADO POR:** Relación de N a M entre MAPA y SPRITE.
    - Un mapa está formado por varios sprites, y un sprite puede formar y estar presente en varios mapas.

ANÁLISIS Y DISEÑO

TRANSFORMACIÓN DE MODELO DE DOMINIO A OBJETOS, XML Y MAPAS

Como ya se adelantaba en el *Planteamiento Inicial, Phoenix* empleará un modelo de base de datos distinto al convencional, usando archivos XML como base de datos. Por esta razón, en esta sección se describirá el formato general de la plantilla general que se usará así como las razones para tomar esta decisión.

En primer lugar, al plantear el diseño de *Phoenix* se vio que se necesitarían almacenar en bases de datos los siguientes elementos:

- Información referente a los objetos usables y equipables
- Diálogos de los NPC
- Información referente a las tiendas
- Información referente a la partida guardada

Aunque no son pocos, realmente no forman parte de los elementos gráficos y "pesados" que están en constante actualización en la pantalla y/o con los que se puede interaccionar, siendo la mayoría de ellos de naturaleza situacional y complementaria (por ejemplo, el juego podría ser jugable sin diálogos, simplemente sería menos completo e intuitivo). Por esta razón se decidió que la base de datos tenía que ser lo más ligera y sencilla posible, y como ya se ha mencionado, se procedió a emplear archivos XML que contendrían todos estos elementos.

Aunque el uso de estos y cómo se relacionan con *Tiled* y el código se verá más en profundidad en el apartado de *Desarrollo*, su funcionamiento básico es el siguiente:

Cada XML se dividirá en secciones que se correspondan a la naturaleza de lo que se desea almacenar, y a cada una de estas secciones se le dará un nombre identificador. Por ejemplo, si queremos almacenar espadas se creará una sección a la que llamaremos "swords". Para cada sección de este tipo se crearán subsecciones en las que se indicará el tipo del elemento dentro de la sección. Por

ejemplo, en el juego se pueden obtener tres tipos de espadas, por lo que dentro de “swords” tendríamos tres elementos: “type_1”, “type_2” y “type_3”. Cada uno de estos tipos almacenará sus detalles, como el nombre, efectividad, etc. Cómo todo esto se relaciona con el código se indicará en la sección de *Desarrollo*, y a continuación se mostrarán las plantillas XML para los elementos a almacenar:

Objetos usables y equipables

Este XML contendrá la información referente a los objetos que pueden ser obtenidos o comprados que pueden ser usados por el jugador. El formato es el siguiente:

```
<equipment>
    <swords>
        <type_1>
            <name>Espada de bronce</name>
            <description>Espada de bronce. No parece muy afilada</description>
            <effect>1</effect>
            <sellprice>50</sellprice>
            <buyprice>100</buyprice>
        </type_1>
        <type_2>
            <name>Espada de plata</name>
            <description>Espada de plata. Parece en buen estado.</description>
            <effect>2</effect>
            <sellprice>300</sellprice>
            <buyprice>600</buyprice>
        </type_2>
        <type_3>
            <name>Espada de oro</name>
            <description>Espada de oro. Está afilada y un estado excelente.</description>
            <effect>3</effect>
            <sellprice>2000</sellprice>
            <buyprice>4000</buyprice>
        </type_3>
    </swords>

    <spears>

    </spears>

    <chests>
        <type_1>
            <name>Armadura de bronce</name>
            <description>Armadura de bronce. No parece en buen estado.</description>
            <effect>1</effect>
            <sellprice>30</sellprice>
            <buyprice>60</buyprice>
        </type_1>
        <type_2>
            <name>Armadura de plata</name>
            <description>Armadura de plata. Parece en buen estado.</description>
            <effect>2</effect>
            <sellprice>200</sellprice>
            <buyprice>400</buyprice>
        </type_2>
        <type_3>
            <name>Armadura de oro</name>
            <description>Armadura de oro. Está cuidada y un estado excelente.</description>
            <effect>3</effect>
            <sellprice>1550</sellprice>
            <buyprice>3100</buyprice>
        </type_3>
    </chests>
```

Ilustración 16 XML - Objetos y Equipo

Se pueden observar en la ilustración los elementos mencionados. En primer lugar, la primera *tag* “*equipment*” nos indica que nos encontramos en la sección de objetos equipables. Vemos después que sus elementos hijo son “*swords*”, “*spears*” y “*chests*” entre otros. Estos hacen referencia a diferentes piezas de equipo que se podrán encontrar en el juego, como espadas, lanzas o armaduras de pecho. Se observa que “*spears*” aún no incluye información. Esto se debe a que las lanzas no han sido implementadas aún en el juego, y serían un arma encontrada en versiones futuras.

Diálogos de NPC

Almacenando esta vez los diálogos de los diferentes NPC encontrados en el juego, este XML sigue un diseño muy similar al anterior:

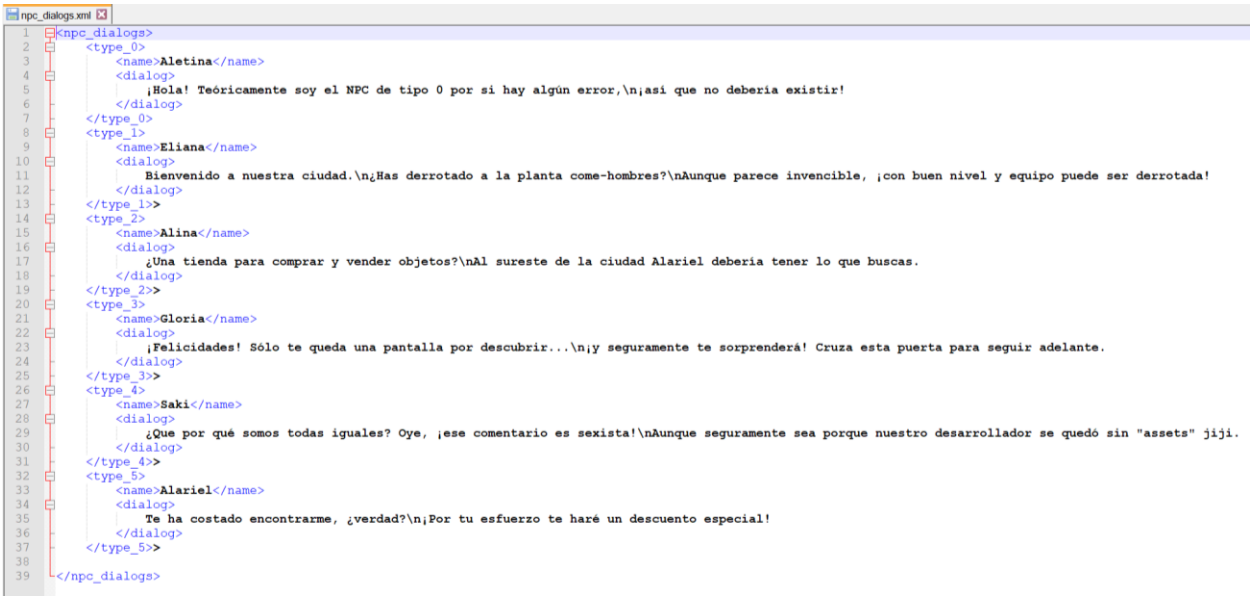


Ilustración 17 XML - Diálogos de NPC

Nos encontramos de nuevo con una primera *tag* que nos indica en qué sección nos encontramos. En este caso, no es necesario hacer ninguna división más por lo que pasamos directamente a indicar cada tipo de NPC con su diálogo correspondiente.

Tiendas NPC

Este XML almacenará los elementos que se venden en las diferentes tiendas NPC. En este caso, solo el NPC “type_5” puede vender objetos, por lo que el formato es el siguiente:

```
<shops>
    <type_5>
        <name>Alariel</name>
        <usable>
            <item> mpPotion </item>
            <item> hpPotion </item>
        </usable>
        <equipment>
            <swords>
                <item> type_1 </item>
                <item> type_2 </item>
                <item> type_3 </item>
            </swords>
            <chests>
                <item> type_1 </item>
                <item> type_2 </item>
                <item> type_3 </item>
            </chests>
        </equipment>
    </type_5>
</shops>
```

Habiendo visto el diseño de los anteriores XML, se ve que este funciona de la misma manera. Este NPC vende pociones, espadas y armaduras de pecho de 3 tipos.

Ilustración 18 XML - Tiendas NPC

Partida guardada

La funcionalidad de guarda y cargar partida no ha sido implementada y queda como trabajo futuro, como ya se detallará más adelante, y por esta razón no se ha diseñado el XML pertinente. Sin embargo, el diseño seguiría la misma línea que los vistos y se almacenarían los siguientes elementos:

- Atributos del personaje (nivel, fuerza, defensa, etc)
- Inventario del personaje
- Mapa actual
- Diálogos y secuencias vistas hasta el momento (punto de la historia en el que se encuentra)

Objetos

Como es lógico, existe mucha más información que debe ser almacenada durante la partida en *Phoenix*. Ejemplos de alta relevancia serían:

- Estado del mapa
- Estado de sprites específicos del mapa (por ejemplo, si un cofre está abierto o cerrado, tanto para su colisión como *sprite* a representar)
- Estado de los enemigos
- Estadísticas y nivel del jugador
- Etc

Todos estos elementos, a no ser que se esté cargando la partida, son almacenados y gestionados por los propios objetos de Java en tiempo real a medida que se va avanzando en la partida. Muchos de ellos, por razones de diseño, se resetean al reanudar o empezar una nueva partida (por ejemplo, la posición y número de enemigos se resetea al cambiar de mapa o cargar una partida, ya que el jugador puede querer volver a repetir la zona o adquirir más nivel), así que es lógico que lo gestionen los

propios objetos. Algunos objetos, como las armas, armaduras o ítems, interactúan con los XMLs para cargar sus descripciones y características, y luego modificarlas en tiempo real si fuera necesario (un arma desgastada, por ejemplo, ve su valor de venta reducido, aunque este en un principio es siempre el mismo). De aquí se ve que, de un modo general, la información se almacena tanto en XMLs persistentes, cuyo contenido no se ve alterado (excepto el XML referente a la partida guardada), como en objetos, que pueden ver su información modificada a lo largo de la partida, y que además, en muchos casos, interactúan entre ellos.

Mapas

Por último, también se almacenan los mapas, como ya se ha descrito, mediante archivos .tmx. Estos, al igual que los XMLs, no se ven modificados en ningún momento y sirven para la carga inicial de los mapas. Si algún *sprite* de los mapas se viera modificado (por ejemplo, se abre un cofre), esto sería de nuevo gestionado por los propios objetos del código y las variables en las que se está almacenando el mapa cargado. Se ve, de esta forma, una interacción similar a la de los XMLs, en la que se carga un objeto inalterable para que después los objetos del juego lo modifiquen, si es necesario. Estas modificaciones, por diseño, son temporales y no deben ser almacenadas, ya que se pretende ofrecerle al jugador la opción de experimentar los mapas igual que en la primera carga, incluso en una misma partida.

DIAGRAMA DE CLASES

En esta sección se mostrará el diagrama de clases de *Phoenix*. El proyecto cuenta con 50 clases, por lo que para una clara representación de estas se mostrarán únicamente las variables y métodos más relevantes. Además, se mostrará el diagrama de la siguiente manera, de forma que sea más comprensible para un proyecto de esta magnitud:

- El proyecto ha sido dividido en paquetes, los cuales incluyen clases con características similares, y que además suelen incluir herencia. En primer lugar, se mostrará cómo estos paquetes interactúan entre sí, mostrando un “diagrama de paquetes”. Ya que todos los paquetes contienen clases de naturaleza muy similar, ofrece una visión clara y general del proyecto.
- Después, se harán sub-diagramas de clases para cada uno de estos paquetes. En ellos se verán elementos más precisos como la herencia o la relación entre clases con un mismo objetivo. De esta forma, se espera hacer un diagrama de clases más modular y comprensible, aprovechando la división en paquetes del código. Esta misma división se hará en el apartado de *Desarrollo*.
- Por último, se describirá de forma breve qué representa cada clase y si tiene alguna característica general particular.

Diagrama de paquetes

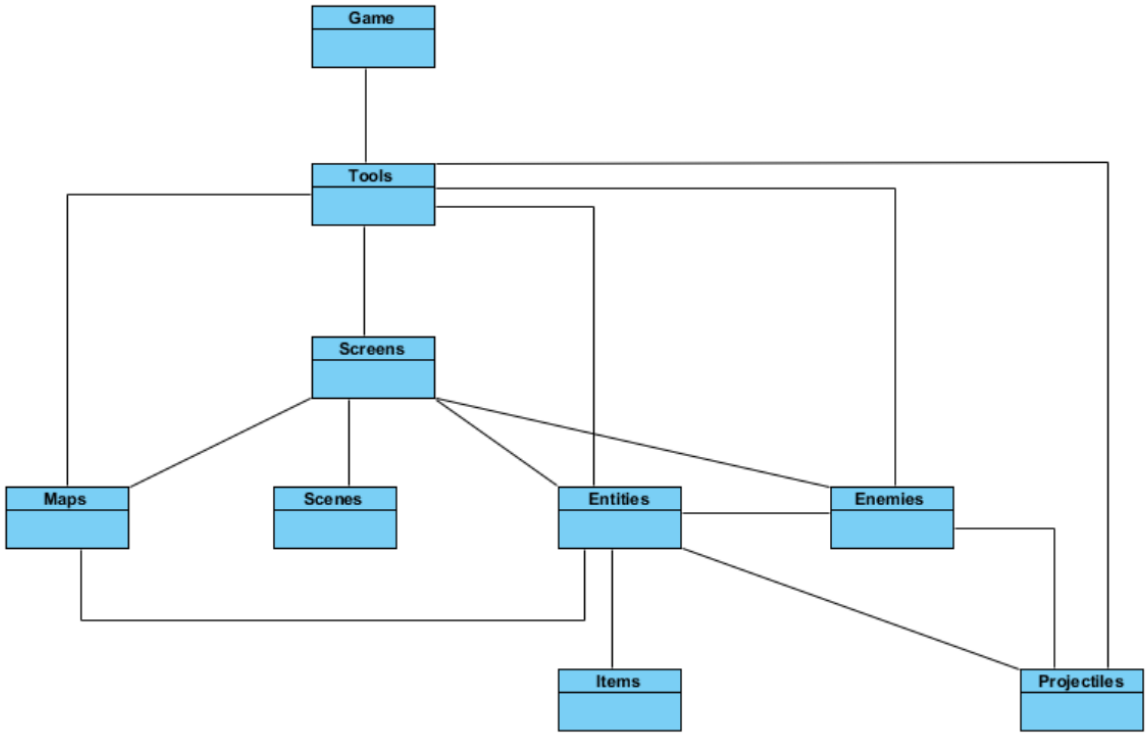


Ilustración 19 Diagrama de paquetes

En esta ilustración se aprecian las relaciones entre los distintos paquetes del proyecto. A continuación se describirá de forma breve cada uno de ellos (esta explicación se expande en el apartado de *Desarrollo*):

- **Game**: Consiste únicamente de la clase Game. Esta se encarga de realizar las primeras llamadas.
- **Tools**: Consiste de clases que se usan como herramientas en la creación del juego, pero no tienen efecto visual alguno. Entre ellas están clases como la encargada de crear los mundos, gestionar la colisión o el *input* del jugador, entre otras. La mayoría de paquetes hacen uso de estas herramientas.
- **Screens**: Contiene todas las pantallas. La clase más destacada, GameScreen, es en la que se desarrolla el *gameplay* del juego. El resto se corresponden a menús o interfaces de naturaleza más estática.
- **Maps**: Contiene todos los elementos relacionados con los mapas, como árboles, rocas, escaleras, etc.
- **Scenes**: Contiene las interfaces estáticas que se ven durante el *gameplay*.
- **Entities**: Contiene todas las entidades del juego que no sean enemigos. En el estado actual del juego, contiene al jugador principal y a los NPCs.
- **Enemies**: Contiene todos los tipos de enemigos.
- **Items**: Contiene todos los tipos de objetos que se pueden obtener en el juego, tanto equipables (armas, armaduras…) como usables (pociones).

- **Projectiles**: Contiene todos los proyectiles del juego, tanto del propio jugador como de los enemigos.

Tools

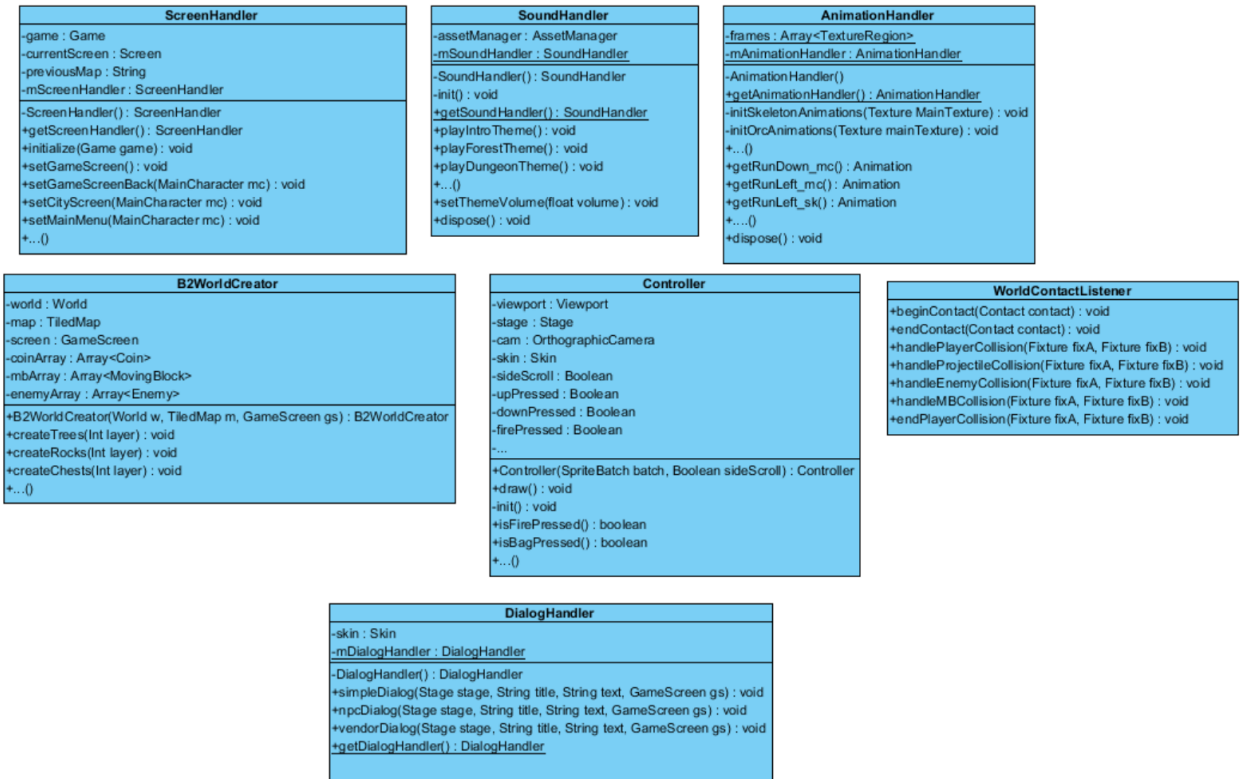


Ilustración 20 Diagrama de clases - Tools

Descripción de clases:

- **ScreenHandler:** Gestiona las pantallas. Indica cuando eliminar la pantalla actual y mostrar una nueva. Se encarga de mantener pantallas que no se desean perder y de limpiar la memoria cuando una pantalla se descarta.
- **SoundHandler:** Encargado de gestionar el sonido. Gestiona tanto sonidos aislados como las pistas de audio (temas musicales).
- **AnimationHandler:** Encargado de cargar todas las animaciones que se van a usar. Su presencia es clave para el rendimiento óptimo del juego, ya que cargar estas animaciones en más de una ocasión causaría grandes problemas de optimización.
- **B2WorldCreator:** Encargado de crear el mundo Box2D a partir de mapas .tmx. Crea también nuevos elementos en las coordenadas indicadas en el mapa.
- **Controller:** Encargado de gestionar los controles del usuario. En la versión Android, es también el encargado de crear la interfaz visual de controles.
- **WorldContactListener:** Encargado de gestionar la colisión de todos los elementos del juego.
- **DialogHandler:** Se encarga de gestionar los distintos tipos de diálogos del juego.

Screens

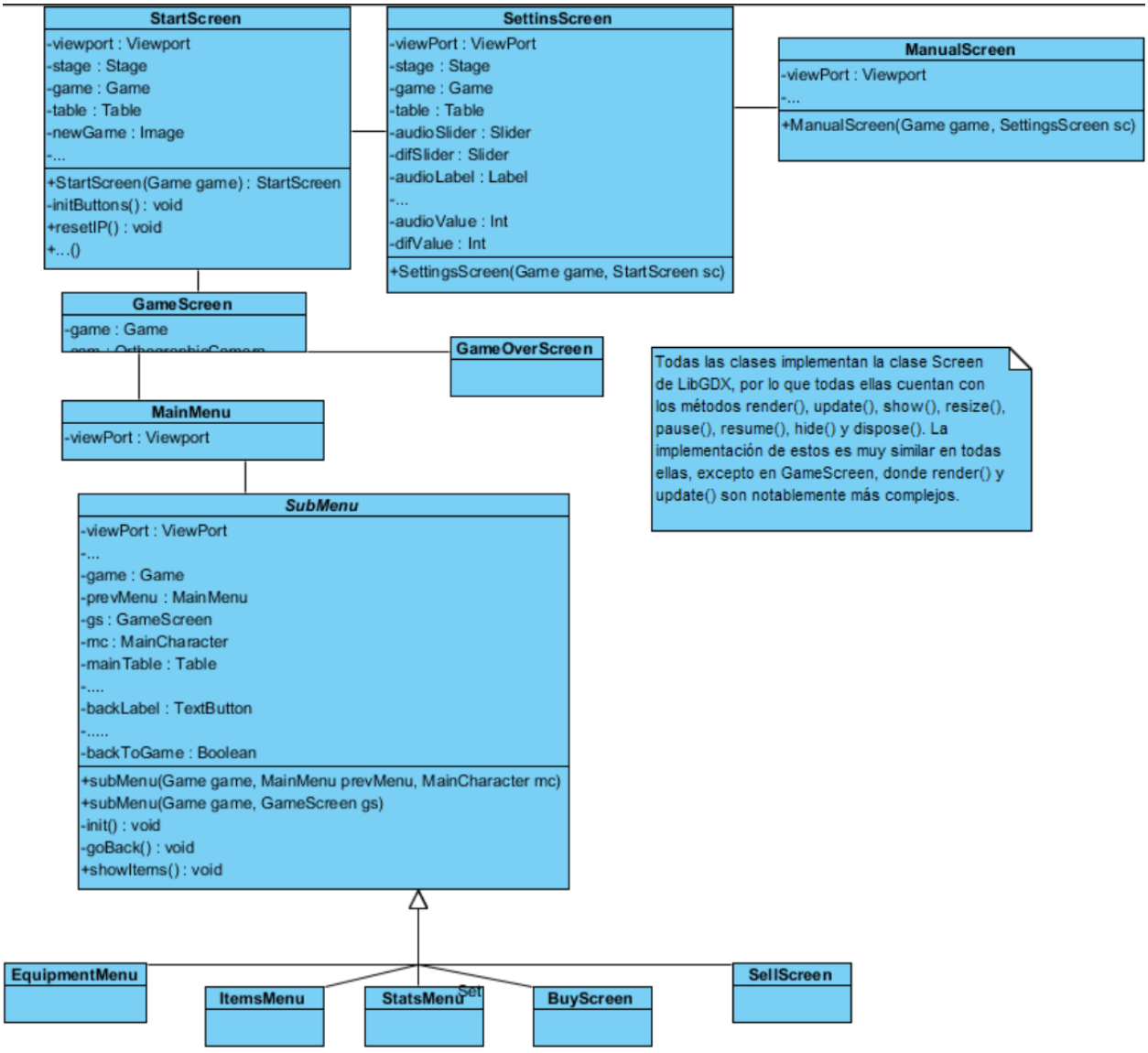


Ilustración 21Diagrama de clases – Screens

Descripción de clases:

- **StartScreen:** Pantalla inicial del juego. La primera que se ve al iniciar la aplicación.
- **SettingsScreen:** Pantalla de ajustes. Se accede a ella desde la pantalla de inicio.
- **ManualScreen:** Manual de usuario en accesible desde la sección de ajustes. Guía al usuario en sus primeros pasos en el juego.
- **GameScreen:** Pantalla en la que se desarrolla el gameplay del juego. Es con diferencia la más larga y compleja. En el diagrama se ha minimizado por razones de claridad, por lo que se mostrará más adelante su diseño completo.
- **GameOverScreen:** Pantalla que se muestra cuando el jugador muere. No está implementada en el juego final.

- **MainMenu:** Menú principal del jugador. Lleva a todos los submenús con la información del jugador. Al igual que GameScreen, está minimizada en el diagrama y se mostrará entera al final de esta sección.
- **SubMenú:** Clase abstracta que indica la apariencia estándar que tendrán todos los submenús accesibles en el juego.
- **EquipmentMenu, ItemsMenu, StatsMenu, BuyScreen, SellScreen:** Submenús a los que se accede cuando se quiere ver el equipo del jugador, los estados del jugador, los objetos del jugador, comprar o vender objetos.

Muchas de estas clases cuentan con un número muy elevado de variables y métodos (hay clases con 20+ Labels o TextButtons). Por ello, en el diagrama se muestran a veces puntos suspensivos, indicando que hay más variables de un estilo muy similar a la mostrada con anterioridad. Se busca, de esta forma, hacerlo más conciso y legible. Se han mostrado las que se consideraban más relevantes.

Sin embargo, tanto GameScreen como MainMenu son clases con mucho contenido único y complejo, y por lo tanto se muestran a continuación en su totalidad:

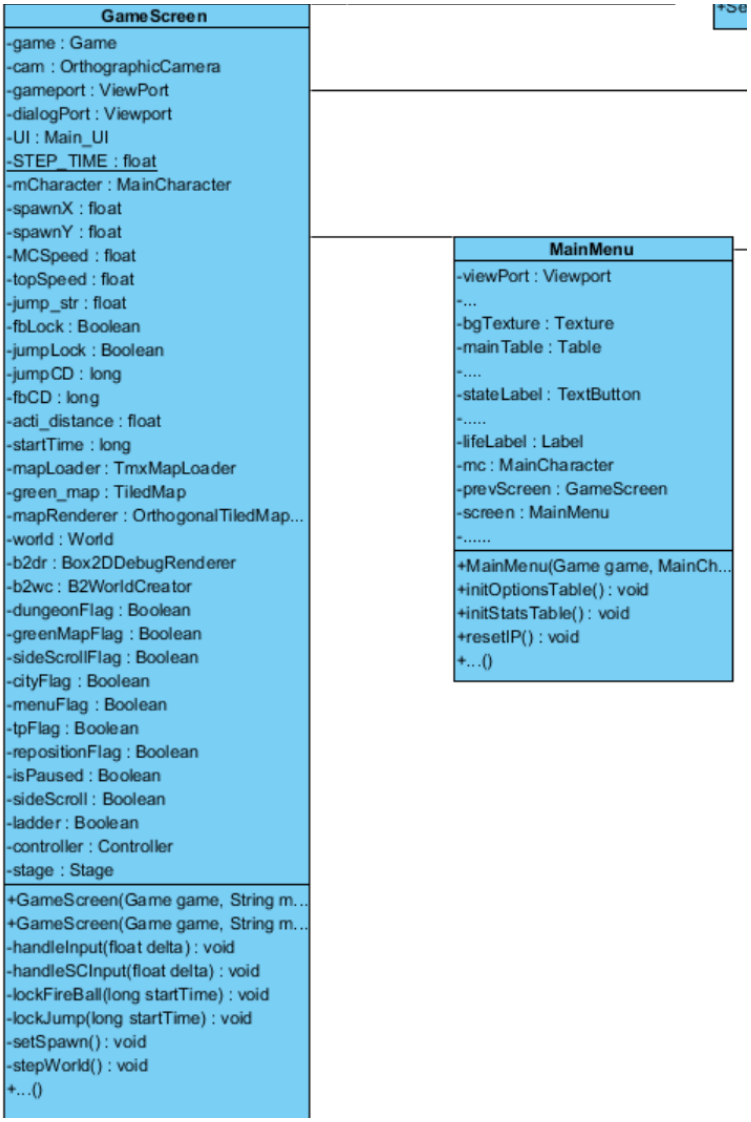


Ilustración 22 Diagrama de clases - GameScreen y MainMenu

Maps

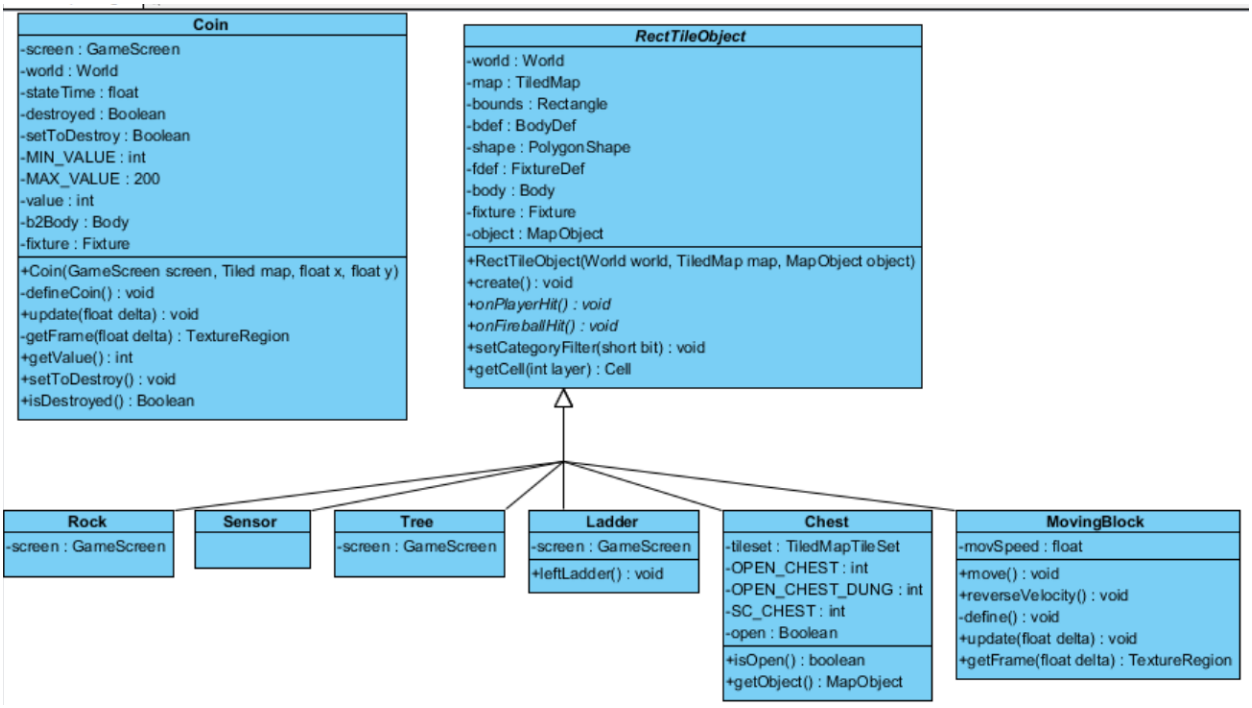


Ilustración 23 Diagrama de clases – Maps

Descripción de clases:

- **RectTileObject:** Representa todo objeto que se compone en el juego por polígonos cuadrados. Todos excepto MovingBlock usan *sprites* extraídos del mapa generado, mientras que este (y los cofres cuando se abren) extraen sus *sprites* de un *spritesheet* aparte.
- **Rock:** Todo elemento formado por rocas en el juego.
- **Tree:** Todo árbol del juego.
- **Ladder:** Escaleras del juego.
- **Chest:** Cofres del juego
- **MovingBlock:** Bloques en movimiento del juego.
- **Sensor:** Sensores "invisibles". Sirven para el correcto funcionamiento de los bloques. Su función se explica mejor en la sección de Desarrollo.
- **Coin:** Monedas que se encuentran por todos los mapas.

Projectiles

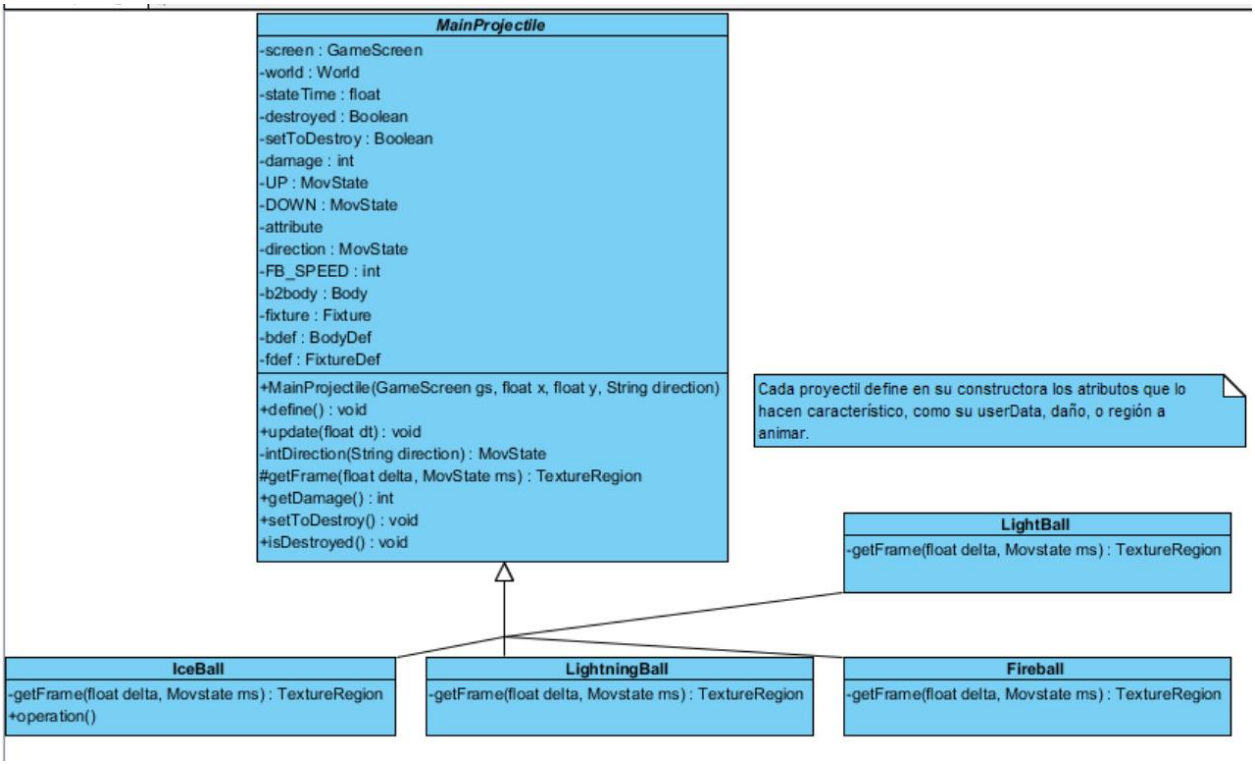


Ilustración 24 Diagrama de clases – Projectiles

Descripción de clases:

- **MainProjectile:** Clase abstracta que representa todo proyectil del juego, tanto lanzado por el jugador como por enemigos.
- **Iceball:** Bola de hielo lanzada por el jugador.
- **LightningBall:** Bola de rayo lanzada por el jugador.
- **FireBall:** Bola de fuego lanzada por el jugador.
- **LightBall:** Bola de luz lanzada por algunos tipos de enemigos.

Items

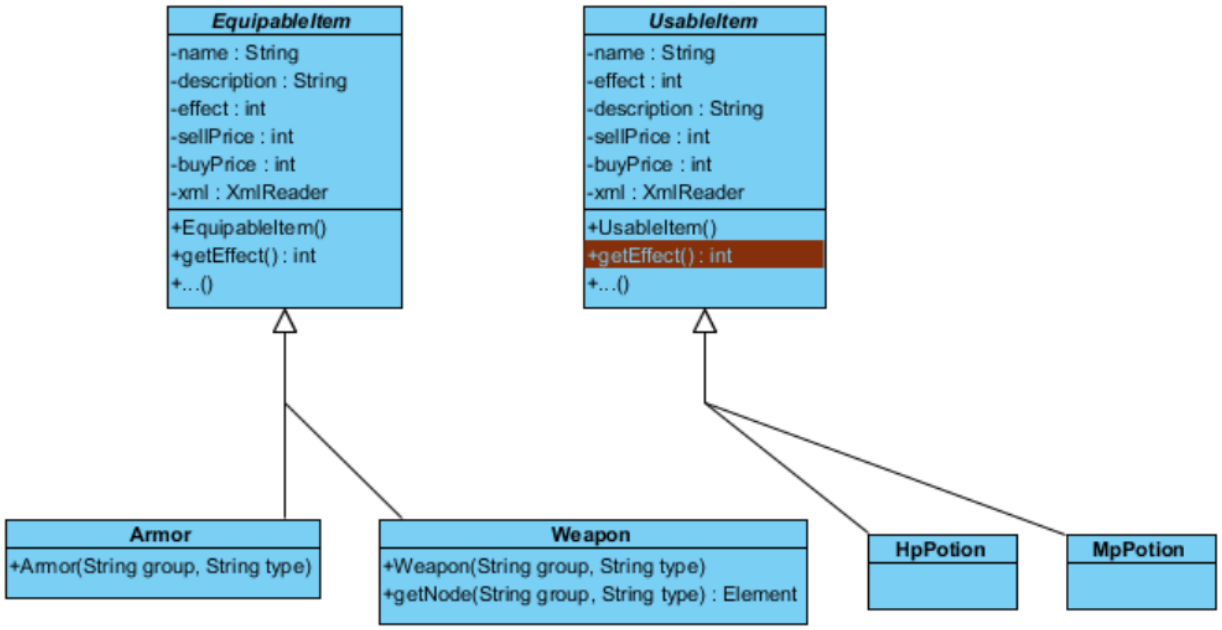


Ilustración 25 Diagramas de clases – Items

Descripción de clases:

- **EquipableItem:** Cualquier objeto que el jugador compre o se encuentre que después se pueda equipar.
- **Armor:** Pieza de armadura que el jugador se puede equipar.
- **Weapon:** Arma que el jugador se puede equipar.
- **UsableItem:** Cualquier objeto que el jugador compre o se encuentre que después pueda usar.
- **HpPotion:** Poción que recupera la vida del jugador.
- **MpPotion:** Poción que recupera el maná del jugador.

Enemies

Al igual que en casos anteriores, la clase abstracta Enemy es demasiado grande y se mostrará en una ilustración aparte:

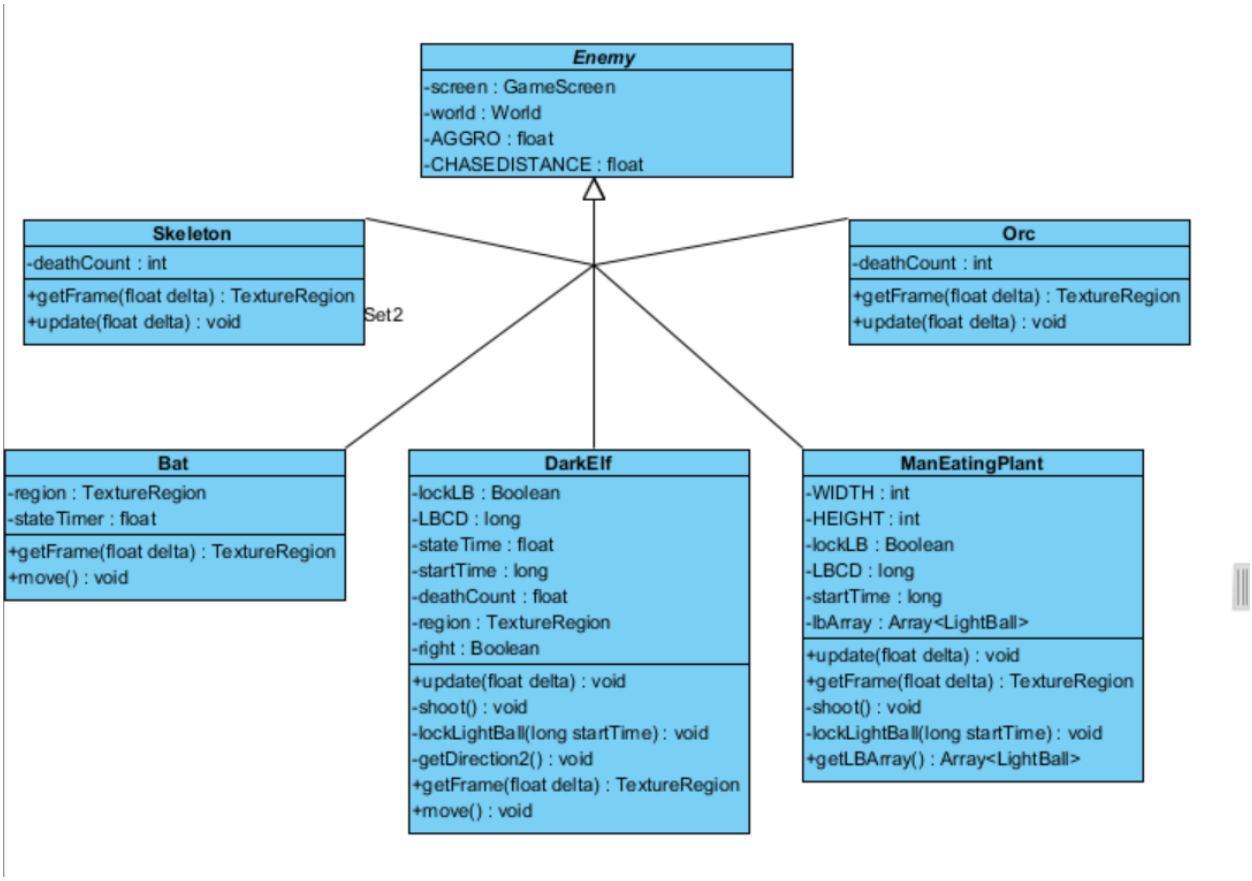


Ilustración 26 Diagrama de clases – Enemies

Descripción de clases:

- **Enemy:** Clase abstracta que representa cualquier enemigo del juego.
- **Skeleton:** Enemigo de tipo esqueleto.
- **Orco:** Enemigo de tipo orco.
- **Bat:** Enemigo de tipo murciélago
- **DarkElf:** Enemigo de tipo elfo oscuro.
- **ManEatingPlant:** Enemigo de tipo planta come-hombres.

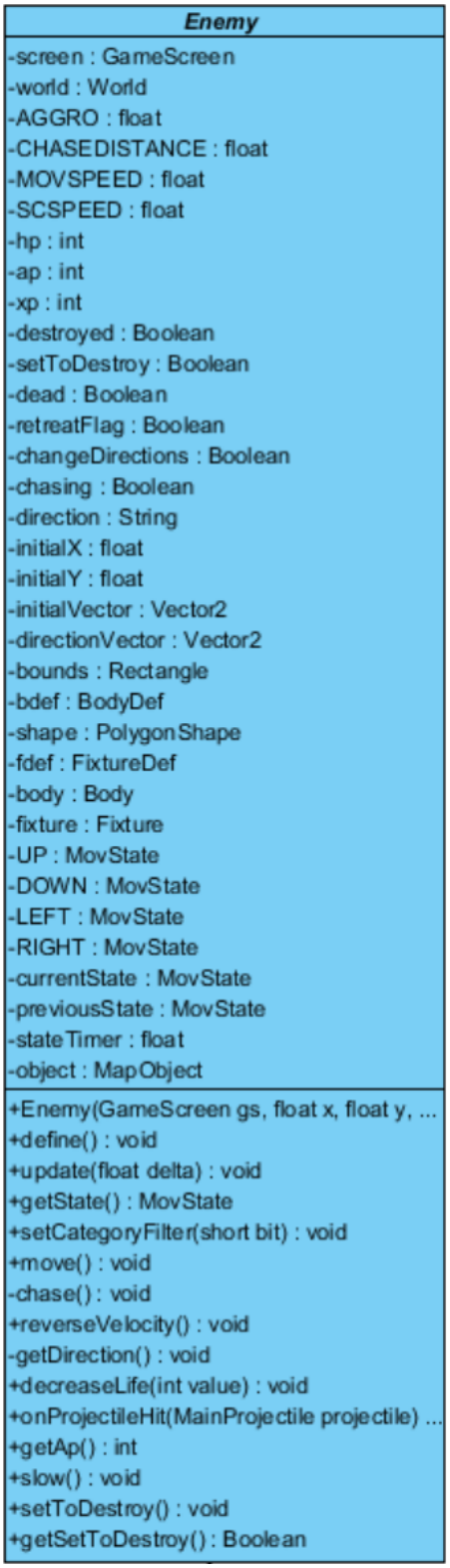


Ilustración 27 Diagramas de clases – Enemy

Scenes

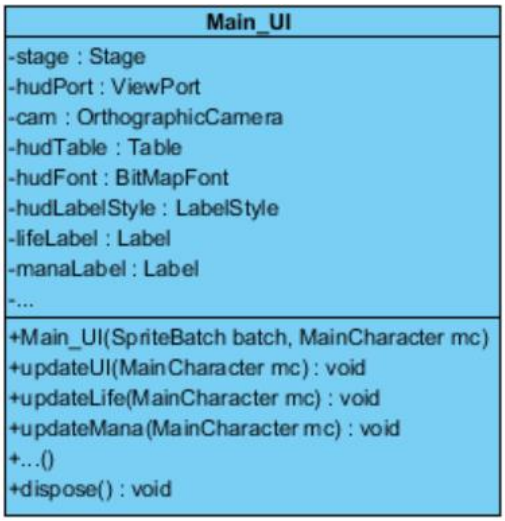


Ilustración 28 Diagrama de clases - Main_UI

Descripción de clases:

- **Main_UI:** Interfaz gráfica informativa que se muestra de forma permanente durante el gameplay, en la parte superior izquierda de la pantalla.

Entities

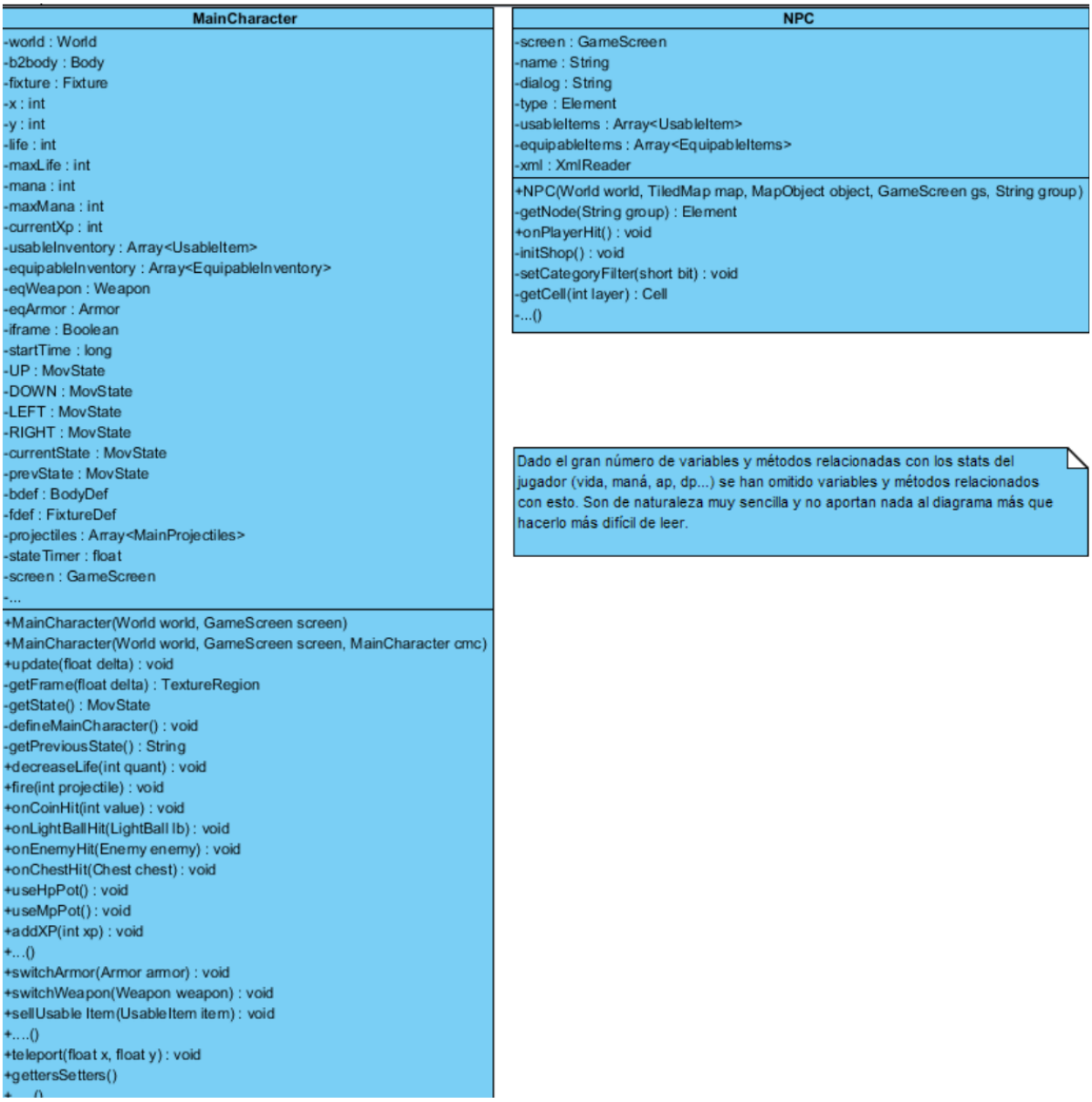


Ilustración 29 Diagrama de clases – Entities

Descripción de clases:

- **MainCharacter:** Clase que hace referencia al personaje principal controlado por el jugador. Tiene un gran tamaño, ya que gestiona sus estados, colisión, equipo, progreso…
- **NPC:** Cualquier NPC del juego. Algunos de ellos tendrán objetos en su tienda, los cuales también varían de NPC a NPC.

DIAGRAMAS DE SECUENCIA

Al igual que con los casos de uso, los diagramas de secuencia se encuentran en una sección aparte al final del documento – *Anexo II*.

DESARROLLO

En esta sección se describirá todo el desarrollo de *Phoenix* en lo referente a la implementación del código, así como las explicaciones más detalladas del funcionamiento de LibGDX, Box2D, Tiled y las medidas tomadas para la optimización de memoria RAM, uso de CPU y GPU. Muchos de estos elementos son complejos y solo tienen sentido habiendo visto pasos previos, por lo que se intentará ofrecer una visión cronológica del desarrollo yendo en el orden en el que se ha realizado.

Muchos de los elementos que se van a tratar comparten características entre sí. Por esta razón se explicará cada comportamiento destacable de los elementos del juego, y en caso de que se repita con variaciones insignificantes no se entrará en detalle en estas variaciones (por ejemplo, distintos proyectiles que en esencia son lo mismo). Dadas estas similitudes, las clases con naturalezas similares se han agrupado en carpetas, a las cuales se referenciará en los títulos de las siguientes secciones:

Tiled

El primer paso en el desarrollo de Phoenix fue el diseño y programación de mundos. Este se dividió en dos fases: diseño puramente gráfico e implementación. Para poder entender estas dos fases mejor, es importante ver el funcionamiento del editor gráfico *Tiled.* Como ya se adelantaba en secciones anteriores, esta herramienta nos permitirá emplear una serie de *tilesets* para crear nuestros mapas con la siguiente interfaz:

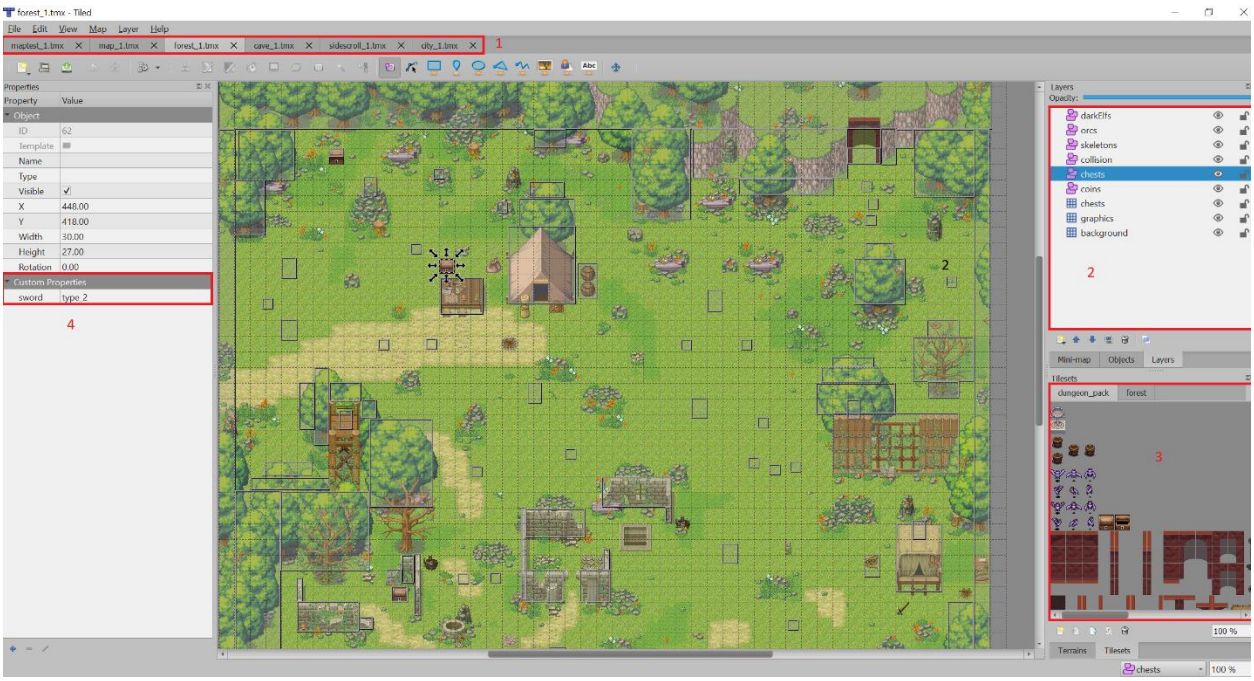


Ilustración 30 Tiled - Creación del mapa "bosque"

Los elementos que encontramos destacados en la ilustración 19 son los siguientes:

1. Los distintos mapas realizados. Todos ellos tienen la extensión .tmx, propia de *Tiled*, y se guardarán en la carpeta assets del proyecto para poder ser accesibles mediante el código, al igual que el resto de elementos como sprites, música, etc.

2. La creación gráfica de los mapas se hace por capas (recuerda, por ejemplo, al conocido *Photoshop*). En este elemento se aprecian las nueve capas que se han creado, las cuales tienen dos iconos distintos. Los iconos con un *grid azul* representan elementos puramente gráficos con los cuales no se pretende interactuar en el juego en lo que a la colisión se refiere. Aun así, conocer la posición de esta capa nos ayudará a, por ejemplo, cambiar un *sprite* por otro mientras se juega al juego. Las capas representadas por iconos de polígonos rosas representan elementos con los que se pretende colisionar. Supongamos que en una de las primeras capas hemos colocado un árbol, y se quiere que exista colisión con ese árbol. El primer paso para que esto ocurra es crear un polígono alrededor de este árbol (de forma que se ajuste a su forma lo máximo posible), creando por lo tanto la capa de colisión "*collision*" y creando manualmente el polígono. Ya que la capa "*collision*" representa todo tipo de colisión, se puede realizar el mismo procedimiento para todos los elementos con los que se pretende chocar. Todos los polígonos que se aprecian en el mapa ejemplo (la mayoría son rectángulos) forman parte de las distintas capas de colisión.

3. Este elemento representa el *spritesheet* con el que se está trabajando. El proceso de creación gráfica del mapa consiste en, seleccionando sprites individuales, juntarlos de forma que se forme el mapa. El *spritesheet* que se muestra contiene elementos como distintos tipos de puertas, cofres y algún tipo de enemigo.

4. Por último, este elemento representa propiedades personalizadas de cada uno de los

polígonos. Por defecto este campo está vacío, pero como se verá en la siguiente sección, la creación de estas propiedades es básica para la realización de ciertas tareas y la distinción de polígonos. En este caso, se ha seleccionado el rectángulo que rodea un cofre, el cual tiene una propiedad personalizada llamada "*sword – type_2*".

Ahora que ya conocemos los elementos más relevantes de Tiled, podemos pasar a entender cómo se relacionan estos con nuestro código.

Tools

La carpeta *Tools* contiene aquellas clases que sirven para la realización de alguna función peculiar que no tiene impacto gráfico en el juego pero es imprescindible en su creación. Ya que en la sección anterior se ha hablado de Tiled, se procederá ahora a hablar sobre la segunda fase de la creación de los mapas y su comunicación con el código: la clase *B2WorldCreator*.

B2WorldCreator

Los polígonos que se han creado en el mapa previamente visto no son visibles por defecto y no tienen efecto alguno en la jugabilidad, y son, por lo tanto, inútiles. El renderizado del mapa (mediante elementos que se verán posteriormente) tan solo mostrará las capas gráficas e ignorará el resto, así que se debe poder adquirir la información sobre la posición y tamaño de estos polígonos y así poder trabajar con ellos. Para poder llevar esto a cabo, se necesitan 3 elementos:

- Un mapa (archivo con extensión .tmx) bien formado con *sprites*, capas y propiedades.
- Un mundo Box2D. Nos referimos a "mundo Box2D" como una instancia de la clase *World* propia de Box2D, la cual tendrá la habilidad de gestionar las físicas de este.
- La pantalla que se esté renderizando y actualizando.

```
//Crea nuestro mundo Box2D, que de momento usamos para los Bodies, que no los dibujos de Sprites
public class B2WorldCreator {
    private World world;
    private TiledMap map;
    private GameScreen screen;

    private Array<Coin> coinArray;
    private Array<MovingBlock> mbArray;
    private Array<Enemy> enemyArray;

    public B2WorldCreator(World world, TiledMap map, GameScreen screen) {
        this.world = world;
        this.map = map;
        this.screen = screen;
        coinArray = new Array<Coin>();
        mbArray = new Array<MovingBlock>();
        enemyArray = new Array<Enemy>();
```

Ilustración 31 Tools - Atributos básicos de B2WorldCreator

El siguiente paso será usar *map* para encontrar los rectángulos y hacer lo que se desee con ellos. En este caso se mostrará la creación de algunos elementos básicos:

```
private void createTrees(int layer){
    //Por cada objeto de tipo Object en Tiled cogemos aquellos objetos con ID = 11, hacemos un rectángulo y creamos el objeto
    for (MapObject object : map.getLayers().get(layer).getObjects().getByType(RectangleMapObject.class)) {
        new Tree(world, map, object, screen);
    }
}

private void createRocks(int layer){
    for (MapObject object : map.getLayers().get(layer).getObjects().getByType(RectangleMapObject.class)) {
        new Rock(world, map, object, screen);
    }
}

private void createChests(int layer){
    for (MapObject object : map.getLayers().get(layer).getObjects().getByType(RectangleMapObject.class)) {
        new Chest(world, map, object);
    }
}

private void createCoins(int layer){
    for (MapObject object : map.getLayers().get(layer).getObjects().getByType(RectangleMapObject.class)) {
        Rectangle rect = ((RectangleMapObject) object).getRectangle();

        Coin coin = new Coin(this.screen, map, rect.x / Game.PPM, rect.y / Game.PPM);
        coinArray.add(coin);
    }
}
```

Ilustración 32 Tools - Creación de elementos

En esta ilustración se observan dos tipos de creaciones distintas y la creación de cuatro elementos: árboles, rocas, cofres y monedas. En los tres primeros casos, el procedimiento es sencillo; se cogen todos los polígonos de la capa indicada con la forma indicada, y por cada uno de ellos se procede a crear el objeto pertinente. Estos métodos reciben por parámetro el número de la capa, el cual está indicado por el orden ascendente de las capas en *Tiled* de 0 a X. Hemos visto, por lo tanto, que a un elemento gráfico estático como una roca o un árbol se le puede añadir un polígono con el que se chocará.

¿Pero qué ocurre con los elementos que requieren animación y no pueden por lo tanto ser creados en *Tiled*? Se puede observar que el método *createCoins()*, el cual crea las monedas (objetos animados) no solamente realiza los pasos anteriores, sino que además obtiene las coordenadas X e Y del rectángulo dado y las manda como parámetros en la creación de una moneda. Lo que se consigue con esto no es crear un rectángulo como en el caso anterior, sino dar además el punto de creación de forma que podamos crear objetos en un punto deseado sin necesidad de contar con el *sprite* en *Tiled*, necesario para los objetos que cuentan con animación. Existen 14 métodos de creación de elementos y todos ellos funcionan de una de estas dos maneras, aunque algunas tienen aún un paso más en consideración:

```
private void createNPCs(int layer){
    String type = "type_0";
    for (MapObject object : map.getLayers().get(layer).getObjects().getByType(RectangleMapObject.class)) {
        if(object.getProperties().containsKey("type")){
            if(object.getProperties().get("type").equals("type_1")){
                type = "type_1";
            }
            else if(object.getProperties().get("type").equals("type_2")){
                type = "type_2";
            }
            else if(object.getProperties().get("type").equals("type_3")){
                type = "type_3";
            }
            else if(object.getProperties().get("type").equals("type_4")){
                type = "type_4";
            }
            else if(object.getProperties().get("type").equals("type_5")){
                type = "type_5";
            }
        }
        new NPC(world, map, object, screen, type);
    }
}
```

Ilustración 33 Tools - Creación de NPCs

En el caso de los NPCs, era necesario distinguirlos a unos de otros, ya que cada uno debe tener diálogos diferentes. Es aquí donde entran en juego las propiedades personalizadas de los elementos de colisión. Tal y como se ve en la imagen, se podrán obtener las propiedades personalizadas de cada polígono y poder crear así cada NPC de distinta manera. Las propiedades personalizadas se usan de maneras muy distintas a lo largo de todo el proyecto, y no su existencia no se restringe únicamente a los polígonos; los propios mapas también pueden tenerlos. Con el fin de hacer el proceso de creación de mapas lo más eficiente posible, la constructora de B2WorldCreator llama únicamente a los métodos que necesita para el mapa en cuestión, los cuales se diferencia, obviamente, por sus propiedades personalizadas.

```
public B2WorldCreator(World world, TiledMap map, GameScreen screen) {
    this.world = world;
    this.map = map;
    this.screen = screen;
    coinArray = new Array<Coin>();
    mbArray = new Array<MovingBlock>();
    enemyArray = new Array<Enemy>();

    if (map.getProperties().containsKey("name")) {
        if (map.getProperties().get("name").equals("forest_1")) {
            createRocks(5);
            createChests(4);
            createCoins(3);
            createSkeletons(6);
            createOrcs(7);
            createElfs(8);
        } else if (map.getProperties().get("name").equals("dungeon_1")) {
            createTrees(5);
            createChests(6);
            createBats(7);
            createMep(8);
        } else if (map.getProperties().get("name").equals("sidescroll_1")) {
            createWalls(2);
            createChests(3);
            createCoins(4);
            createLadders(6);
            createMovingBlocks(5);
            createSensors(7);
        } else if (map.getProperties().get("name").equals("city_1")) {
            createWalls(1);
            createNPCs(2);
        }
    }
}
```

Se observa como los mapas también cuentan con properties.

Ilustración 34 Tools - Creación de Mapas

Gestores

Uno de los primeros problemas que se encontró en las primeras ejecuciones del juego fue que, cuando se ejecutaba con pocos objetos animados el rendimiento era bueno, pero a medida que aumentaban el rendimiento (en particular el consumo de memoria RAM) disminuía notablemente. Aunque se explicará más adelante la creación de enemigos en detalle, esta se debía a que se estaban creando animaciones y sonidos para cada uno de los enemigos creados. Este proceso era altamente ineficiente, por lo que se decidió crear una serie de gestores que ayudasen con él. La idea detrás de los gestores era que fuesen clases que siguieran un Patrón Singleton, y que se encargasen de inicializar todas las animaciones, sonidos y diálogos una sola vez al principio del juego. Además, de esta forma, podían ser accesibles en todo momento por los enemigos o elementos que los necesitaran. Con esta filosofía en mente se crearon las siguientes clases:

- **AnimationHandler:** Encargado de gestionar todas las animaciones del juego, así como algunos *sprites,* texturas y apariencias de tablas y botones. Su creación fue uno de los factores que mayor impacto ha tenido en el proceso de optimización del juego.

- **SoundHandler:** Encargado de gestionar toda la música y sonidos del juego. Gestiona que los sonidos suenen una sola vez, mientras que los temas musicales lo hagan de forma continua. Sigue el mismo formato que *AnimationHandler*.

- **DialogHandler:** Encargado de gestionar los diálogos del juego. Lo hace tanto como con

diálogos simples (como los diálogos informativos cuando se abre un cofre) como con diálogos que dan lugar a nuevas pantallas (como el diálogo que da la opción de comprar o vender objetos).

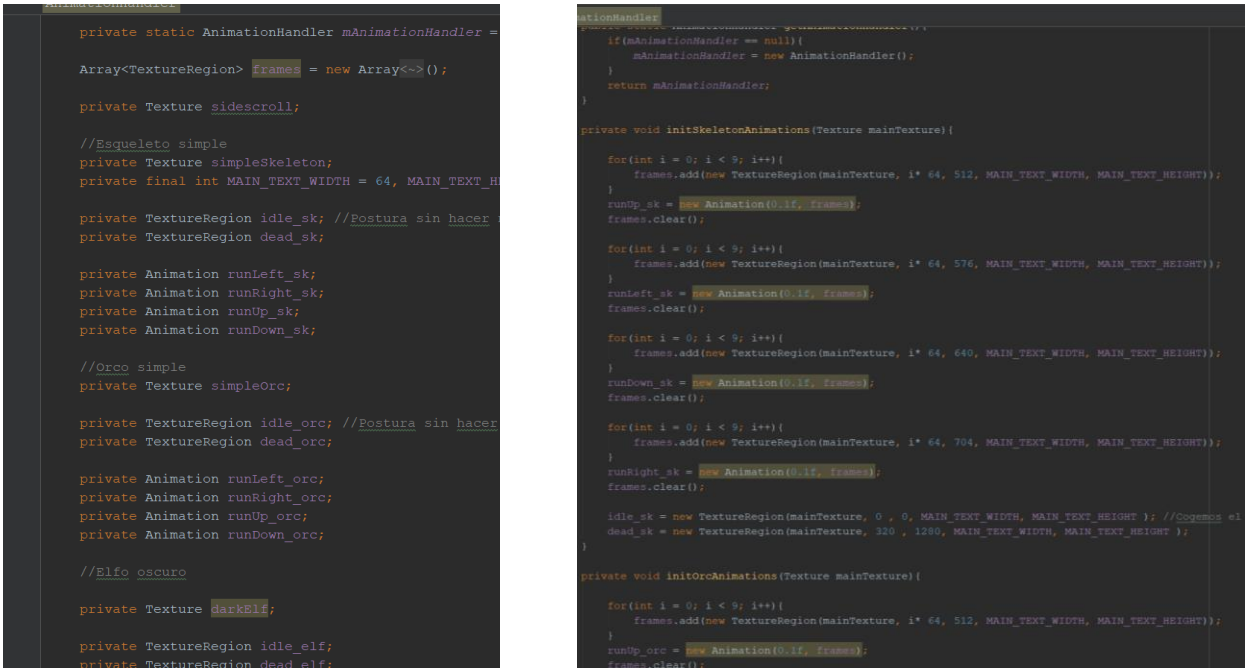


Ilustración 35 – Gestores

Ejemplos de la creación de animaciones en AnimationHandler

Existe otra clase gestora de índole ligeramente distinta. Otro de los problemas que se han encontrado con frecuencia es la eliminación de elementos que no se están usando para la liberación de recursos y memoria. Este proceso es especialmente importante a la hora de eliminar pantallas que no se estén usando, ya que debido a su gran volumen consumen una gran cantidad de recursos. Sumada a esta necesidad, también se encontró la necesidad de poder cambiar de pantalla con facilidad, ya que los eventos que provocan un cambio de pantalla pueden ser muy variados. Por estas razones se creó un gestor de pantallas, el cual se encarga de almacenar la pantalla de la que se procede, en la que se está y de cambiar la pantalla a una nueva cuando sea necesario. A este gestor se le llamó **ScreenHandler**.

Ejemplo de métodos de ScreenHandler. Todos funcionan de forma similar. Sus características más relevantes son la delegación de la instancia del personaje principal (para que no se pierda su información) y la llamada al Garbage Collector. El método dispose() se encarga de sacar de memoria las pantallas que ya no tienen uso.

Ilustración 36 Tools – ScreenHandler

Como se observa en la ilustración, el método *setScreen()* propio de la clase *Game* de LibGDX nos permite sustituir la pantalla actual por la deseada, añadiendo toda la gestión extra en los distintos métodos de *ScreenHandler*.

Detección y gestión de colisión

Uno de los elementos más importantes de cualquier juego es la detección de colisión, es decir, hacer que dos cuerpos puedan chocar entre sí y gestionar qué ocurre cuando esto se da. El primer caso es simplificado gracias al funcionamiento interno de LibGDX. Sin entrar mucho en detalle (ya que se entrará en una sección posterior), existen tres tipos de cuerpos con los que podemos tratar en este framework:

- Estáticos (*Static*)
- Dinámicos (*Dynamic*)
- Cinéticos (*Kinematic*)

Cada uno tiene diferentes características, pero en lo referente a la colisión todos ellos por defecto van a poder chocar entre sí. De esta forma, cuando se cree un cuerpo Box2D (todos los elementos excepto el fondo gráfico van a tener un cuerpo Box2D asignado) por defecto ya van a poder colisionar entre ellos, y solo tendríamos que gestionar esta colisión si quisiéramos que no lo hicieran. En este caso, los cuerpos dinámicos poseen un método *setSensor()*, el cual convierte al cuerpo en un "sensor" de forma que no pueda chocar con otros cuerpos. Existe además otra forma de conseguir que dos cuerpos no puedan colisionar: mediante *mask filters*. A cada cuerpo creado se le puede asignar una máscara, la cual será un bit que siempre será una potencia de 2. La creación de cuerpos se detallará en secciones posteriores, pero a los cuerpos, además de su propia máscara, se les puede asignar con qué máscaras colisionar. De esta forma, si por ejemplo una roca tuviera un bit de máscara 2, y el jugador solo pudiese colisionar con máscaras de bit 4, estos elementos no colisionarían.

```
package com.phoenix.game;

import ...

public class Game extends com.badlogic.gdx.Game {
    public SpriteBatch batch;
    public static final int WIDTH = 800, HEIGHT = 480; //Resolución del juego

    public static final float PPM = 100; //Pixels por metro en box2d
    public static float volume = 0.5f;
    public static int difficulty = 1;

    public static final short DEFAULT_BIT = 2;
    public static final short MC_BIT = 4;
    public static final short ROCK_BIT = 8;
    public static final short TREE_BIT = 16;
    public static final short CHEST_BIT = 32;
    public static final short SENSOR_BIT = 64;
    public static final short MAIN_PROJ_BIT = 128;
    public static final short COIN_BIT = 256;
    public static final short ENEMY_BIT = 512;
    public static final short LADDER_BIT = 1024;
    public static final short MB_BIT = 2048;
    public static final short LIGHTBALL_BIT = 5096;

    @Override
```

Bits de la mayoría de elementos del juego. Se usan sobre todo para la gestión fácil de colisión en la que no es necesario que ocurra nada. En general no se han usado demasiado, ya que se ha usado la gestión que se detalla más adelante.

Ilustración 37 Tools - Mask Filter Bits

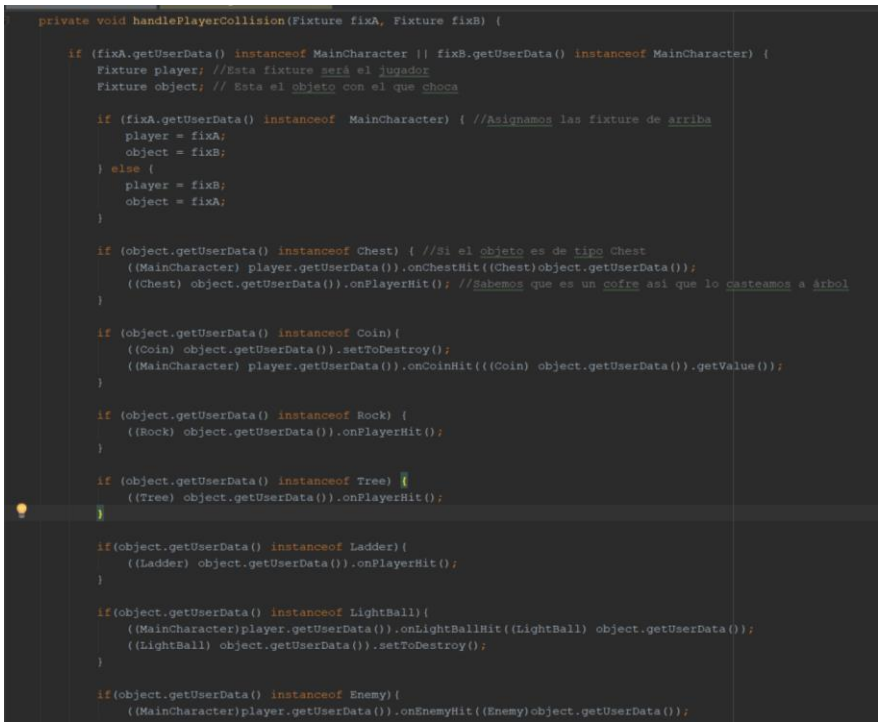
Con la colisión base ya gestionada por Box2D una vez asignado el tipo de cuerpo de los elementos, queda por gestionar qué ocurre cuando se dan distintas colisiones, y para ello se creó la clase **WorldContactListener**. Esta clase recibirá un parámetro de tipo *Contact*, o cuerpos de cualquier tipo que hayan chocado entre sí. De ese contacto se extraerán las dos *fixtures* que han colisionado y se filtrarán para conocer su clase, y por lo tanto, poder gestionar la colisión.

```
//Esta clase permite saber que fixture (b2body) colisiona con qué fixture
public class WorldContactListener implements ContactListener {

    //Qué pasa cuando entran en contacto
    @Override
    public void beginContact(Contact contact) {
        // Las dos fixtures que colisionan
        Fixture fixB = contact.getFixtureB();
        Fixture fixA = contact.getFixtureA();

        handlePlayerCollision(fixA, fixB); //El jugador colisiona con algo
        handleProjectileCollision(fixA, fixB); // Una bola de fuego colisiona con algo
        handleMovingBlockCollision(fixA, fixB);
        handleEnemyCollision(fixA, fixB);

    }

    //Qué pasa cuando termina el contacto
    @Override
    public void endContact(Contact contact) {
        Fixture fixB = contact.getFixtureB();
        Fixture fixA = contact.getFixtureA();
        endPlayerCollision(fixA, fixB);
    }

    @Override
    public void preSolve(Contact contact, Manifold oldManifold) {

    }

    @Override
    public void postSolve(Contact contact, ContactImpulse impulse) {

    }
```

beginContact() y endContact() gestionan qué ocurre cuando comienza y cuando termina una colisión. En este ejemplo, en beginContact() se gestiona qué ocurre cuando el jugador, un proyectil, un bloque en movimiento o un enemigo chocan con algo. Aún es necesario identificar de qué tipo es aquello con lo que chocan, proceso que se detallará a continuación.

Ilustración 38 Tools - Comienzo y final de colisión

El siguiente paso consiste únicamente en decidir de qué tipo es cada *fixture* y actuar en consecuencia. Aunque esto podría hacerse con los bits mencionados, se ha decidido usar el operador *instanceOf* para conocer el tipo de clase de cada elemento:



```
private void handlePlayerCollision(Fixture fixA, Fixture fixB) {

    if (fixA.getUserData() instanceof MainCharacter || fixB.getUserData() instanceof MainCharacter) {
        Fixture player; //Esta fixture será el jugador
        Fixture object; // Esta el objeto con el que choca

        if (fixA.getUserData() instanceof  MainCharacter) { //Asignamos las fixture de arriba
            player = fixA;
            object = fixB;
        } else {
            player = fixB;
            object = fixA;
        }

        if (object.getUserData() instanceof Chest) { //Si el objeto es de tipo Chest
            ((MainCharacter) player.getUserData()).onChestHit((Chest)object.getUserData());
            ((Chest) object.getUserData()).onPlayerHit(); //Sabemos que es un cofre asi que lo casteamos a árbol
        }

        if (object.getUserData() instanceof Coin){
            ((Coin) object.getUserData()).setToDestroy();
            ((MainCharacter) player.getUserData()).onCoinHit(((Coin) object.getUserData()).getValue());
        }

        if (object.getUserData() instanceof Rock) {
            ((Rock) object.getUserData()).onPlayerHit();
        }

        if (object.getUserData() instanceof Tree) {
            ((Tree) object.getUserData()).onPlayerHit();
        }

        if(object.getUserData() instanceof Ladder){
            ((Ladder) object.getUserData()).onPlayerHit();
        }

        if(object.getUserData() instanceof LightBall){
            ((MainCharacter)player.getUserData()).onLightBallHit((LightBall) object.getUserData());
            ((LightBall) object.getUserData()).setToDestroy();
        }

        if(object.getUserData() instanceof Enemy){
            ((MainCharacter)player.getUserData()).onEnemyHit((Enemy)object.getUserData());
```

Ilustración 39 Tools - Gestión de colisión

De esta manera se podrá conocer la el tipo de instancia con la que, en este caso, el jugador está chocando. Cada clase contiene métodos que dictarán como actuar en caso de distintas colisiones. Por ejemplo, los enemigos cuentan con los métodos onPlayerHit() o onProjectileHit(), en los que se describirá qué ocurre cuando chocan con un jugador o con algún tipo de proyectil. Lo que se hace desde esta clase es simplemente identificar la colisión y llamar a esos métodos.

La gran mayoría de las colisiones siguen este esquema, pero se encontró un caso problemático que hubo que tratar de forma distinta: la colisión con una escalera. Esta colisión no solo fue problemática porque no debía haber colisión real (solo detección), sino porque había que tratar con precisión cuándo terminaba para no poder seguir ascendiendo. Esto se solucionó añadiendo un *boolean leftLadder* en la gestión de controles del juego, y gestionando esta variable desde tanto el método *beginContact()* (que la pondría a false) y *endContact()* (que la pondría a true).

Controller

Para terminar con la sección de herramientas, se procederá a explicar la clase **Controller**. Esta clase es la que, en esencia, gestiona el *input* del jugador. En un principio el input se gestionaba de forma directa. Es decir, "si el jugador pulsa la tecla 1 → se dispara una bola de fuego". Este formato solamente tenía valor para los tests iniciales, ya que obviamente no iba a poder funcionar cuando el juego se jugara en un dispositivo móvil. Por esta razón se decidió introducir un elemento intermedio, que además aportaría la interfaz gráfica para el *input* en Android.

Planteamiento Inicial:

Input del jugador ➡ Reacción en la pantalla

Planteamiento final:

Input del jugador ➡ Gestión del controlador ➡ Reacción en la pantalla

Ilustración 40 Tools - Elementos que gestiona Controller

Se aprecian en rojo los elementos que gestiona Controller, gráficamente visibles en la versión Android. En PC estos elementos son invisibles.

De esta forma, se consigue realizar dos tareas en un solo paso. Si el jugador está jugando en un dispositivo Android, el input se realizará de forma táctil. El controlador gestionará esto de forma directa y se generará el *feedback* visual. Por otro lado, si se está jugando en PC, el input será "traducido" a los mismos *boolean* que el controlador emplea cuando se juega desde Android, y de esta forma se generará la misma respuesta en pantalla.

```
private boolean upPressed, downPressed, leftPressed, rightPressed, firePressed, lig
```

Ilustración 41 Tools - Variables de Controller

Se aprecian en la ilustración algunos ejemplos de las variables con las que trabaja **Controller**. Estas tomarán valores tanto si se juega desde Android como si se hace desde PC.

```
@Override
public boolean keyDown(InputEvent event, int keycode) {
    switch(keycode){
        case Input.Keys.UP:
            upPressed = true;
            break;
        case Input.Keys.DOWN:
            downPressed = true;
            break;
        case Input.Keys.LEFT:
            leftPressed = true;
            break;
        case Input.Keys.RIGHT:
            rightPressed = true;
            break;
        case Input.Keys.NUM_1:
            firePressed = true;
            break;
        case Input.Keys.NUM_2:
```

```
Image upImg = new Image(new Texture("flatDark25.png"));
upImg.setSize(70, 70);
upImg.addListener(new InputListener() {

    @Override
    public boolean touchDown(InputEvent event, float x,
        upPressed = true;
        return true;
    }

    @Override
    public void touchUp(InputEvent event, float x, float
        upPressed = false;
    }
});
```

Ilustración 42 Input Android vs PC

De esta forma, con una simple comprobación mediante *getters* desde donde se realiza el movimiento y las acciones del jugador de estas variables se conseguirá el control del jugador.

Aunque resultó muy útil, el controlador dio lugar a una serie de problemas. Los más relevantes fueron:

- La pérdida del *focus*
- El no-reseteo de ciertas variables al cambiar de pantalla

Para que el controlador pueda recibir e interpretar input, debe tener el *focus* del *stage* (el contenedor de elementos visuales) que se esté usando. Al cambiar de pantalla y volver más adelante este *focus* se perdía, por lo que el controlador dejaba de funcionar. El método *resetIP()* devuelve el *focus* al controlador y se resuelve el problema. El cambio de pantalla también resultaba problemático cuando, al ser el cambio instantáneo al darse una colisión, las variables de movimiento se quedaban a true sin poder pasar a false. Esto resultaba en que, al volver a la pantalla principal, el movimiento no funcionaba correctamente. Se solucionó con un método similar, *disableMovement(),* el cual pone a false las variables de movimiento para que al volver a la pantalla principal funcionen correctamente.

```
    public void resetIP() { Gdx.input.setInputProcessor(stage); }

    public void disableMovement(){
        upPressed = false;
        downPressed = false;
        leftPressed = false;
        rightPressed = false;
    }

    public void resetBag() { bagPressed = false; }
}
```

Reseteo de variables al cambiar de pantalla.

Ilustración 43 Tools - Reseteo de variables

Maps

RectTileObject

En esta sección se hablará de todos los elementos que crean y componen los distintos mapas del

juego, y sobre todo sus características comunes más relevantes.

Como ya se había adelantado previamente, todos los elementos que no sean *sprites* del fondo gráfico van a tener un cuerpo Box2D adjunto, por lo tanto es importante entender qué es un cuerpo Box2D. De forma general, los cuerpos Box2D son cuerpos que el propio *framework* nos ayuda a crear con varias clases propias, y que van a ser cuerpos que van a contar con magnitudes físicas y los cálculos que estas conllevan. Para la creación de un cuerpo, se necesitan tener dos elementos básicos en consideración:

- Su definición abstracta (*BodyDef*)
- Su definición tangible o de *Fixture* (*FixtureDef*)

La definición abstracta será la encarga de definir elementos como la posición del cuerpo en el mapa mediante sus coordenadas, el ángulo en el que se encuentra o el tipo de cuerpo que es, entre muchas otras posibilidades que no se han usado en este proyecto. La más importante de estas es el tipo de cuerpo que es, pudiendo estos tipos ser, como ya habíamos adelantado, *dynamic*, *static* o *kinematic*:

- **Dynamic**: Cuerpo dinámico, es decir, que tiene la capacidad de estar en movimiento. Cuando este cuerpo se encuentra activo (por defecto lo está) se calculan de forma constance las magnitudes que provoca y que lo rodean, pudiendo así reaccionar a impulsos, momentos, rotaciones, colisiones, etc. Designado por recuadros naranjas en Box2D.

- **Static**: Cuerpo estático. Este cuerpo no tiene la capacidad de estar en movimiento y no reacciona a magnitudes físicas. Útil para elementos inertes como, por ejemplo, paredes. Sí tiene capacidad de colisión y es mucho más ligera en cuanto a coste computacional. Designado por recuadros verdes en LibGDX.

- **Kinematic**: Una mezcla entre los dos anteriores. Este cuerpo tiene la capacidad de estar en movimiento, pero actúa como si su masa fuera infinita (aunque LibGDX la almacena como 0). A efectos prácticos, es útil para elementos que deben estar en constante movimiento pero no se quiere que sean afectados por fuerzas. Al igual que los tipos anteriores, implementa colisión. Útil, por ejemplo, para plataformas, teniendo además un coste computacional bajo. Designado en LibGDX por recuadros azules.

Una vez se ha definido la *BodyDef* de un cuerpo, se procederá a definir su *FixtureDef*. Esta definición es más intuitiva que la anterior, habiéndose definido en *Phoenix* elementos como:

- Forma (*shape*)
- Densidad (*density*)
- Restitución (*restitution* o *bounciness*, es decir, cuánto rebota un objeto)
- Bits de máscara (visto en el apartado anterior)
- …

En general, todos los cuerpos creados tienen una forma rectangular, densidad y restitución baja y sus bits de máscara correspondientes.

El último elemento a tener en cuenta a la hora de crear cuerpos, magnitudes o distancias en LibGDX, es el *PPM* o *Pixels per Meter*. Por defecto, todas las magnitudes en LibGDX son bastante grandes. Esto se debe a que un pixel equivale a un metro, y en la mayoría de resoluciones esto resulta en cuerpos

de dimensiones muy grandes, y por lo tanto, que generan magnitudes muy altas (por ejemplo una fuerza de atracción muy alta, o requieren impulsos muy grandes para ser movidos y no lo hacen con soltura). Por lo tanto, es muy importante definir los PPM con los que queremos que trabaje Box2D. En el caso de *Phoenix*, se encontró que *PPM* = 100 (100 píxeles = 1 metro) era el número óptimo para trabajar con las magnitudes de fuerzas deseadas, y por lo tanto, todos los cuerpos creados y sus coordenadas deben ser re-escalados entre este número.

```
public RectTileObject(World world, TiledMap map, MapObject object){
    this.object = object;
    this.world = world;
    this.map = map;
    this.bounds = ((RectangleMapObject) object).getRectangle();

    bdef = new BodyDef(); //Definición del cuerpo
    shape = new PolygonShape(); //Qué tipo de polígono rodea al Body
    fdef = new FixtureDef(); //Definición de las fixtures

    create();
}

//Crea el cuerpo Box2D con colisión de polígono rectangular
private void create(){

    bdef.type = BodyDef.BodyType.StaticBody; //Cuerpo estático, no se mueve
    bdef.position.set((bounds.getX() + bounds.getWidth() / 2) / Game.PPM, (

    body = world.createBody(bdef); //Crea el Body en el mundo

    shape.setAsBox(bounds.getWidth() / 2 / Game.PPM, bounds.getHeight() / 2
    fdef.shape = shape;
    fdef.filter.maskBits = Game.SENSOR_BIT;
    fixture = body.createFixture(fdef);
}
```

Definición de la mayoría de cuerpos que formarán parte de los mapas (árboles, rocas, paredes…), mediante la clase RectTileObject. Cuenta con todos los elementos mencionados.

Ilustración 44 Maps – Definición

Se encontraron dos casos problemáticos al implementar esta definición: las escaleras y las plataformas. La escalera debía ser un cuerpo estático sin colisión, que además contara con *flags* que nos dijeran si se estaba en contacto directo con ella o no. Como ya se ha explicado previamente, se resolvió transformándola en un sensor y mediante WorldContactListener. Sin embargo, la plataforma presentaba un problema más complejo.

En la sección *sidescroll* del juego, existen plataformas que están en constante movimiento. El personaje principal debe poder chocar con ellas (para poder montarse), pero estas no deben ser afectadas por el impulso de su caída. Es lógico, por lo tanto, que deben ser de tipo *kinematic*. Sin embargo, esto presentaba un problema. Estas plataformas tenían que rebotar en distintas paredes (cuerpos estáticos) de forma que describieran una ida y venida constante, pero al contar el cuerpo cinético con masa infinita, el cuerpo no rebotaba al chocar, sino que se quedaba pegado a la pared. Para solucionar este problema se dicidió crear una nueva clase propia: **Sensor**. Esta clase no sería más que un delgadísimo (casi imperceptible) cuerpo dinámico que estaría estratégicamente colocado allí donde las plataformas deban rebotar, de forma que en vez de chocar contra la pared choquen contra el sensor y sí puedan rebotar. Esta idea resultó muy efectiva y se pudo completar correctamente la implementación de las plataformas.

Ilustración 45 Maps - Tipos de cuerpos

Se observan en la ilustración: 1. Cuerpo cinemático, 2. Cuerpo estático, 3. Sensor (clase propia), 4. Cuerpo dinámico.

Screens

En esta sección se hablará de los distintos tipos de pantallas que se verán en *Phoenix*. El concepto de pantalla, en este caso, es distinto al de nivel. Es decir, se entenderá una pantalla como un rectángulo en el que se están dibujando 60 veces por segundo una serie de elementos, y no como un nivel que se supera alcanzando un objetivo. Para dar la explicación de todas las pantallas, se va a distinguir entre dos tipos de estas:

- *GameScreen*, o la pantalla en la que transcurren eventos con acción, movimiento y dónde se ve al jugador principal.
- El resto de pantallas, que típicamente serán menús formados por fondos, tablas y elementos como *sliders* o botones.

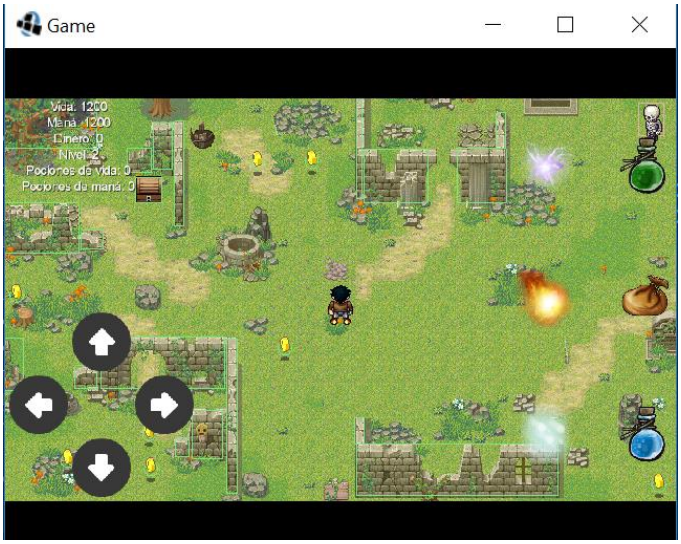
GameScreen

En primer lugar, todas las pantallas del juego implementan la clase *Screen* de LibGDX. Esto conlleva que todas ellas deben contar al menos con los siguientes métodos:

- **Show():** Inicializa las variables con las que se va a trabajar
- **Update()**: Actualiza elementos como posiciones, *sprites* o la creación/eliminación de cuerpos X veces por segundo. En el caso de *Phoenix*, se hace a 60 veces por segundo.
- **Render()**: Dibuja los elementos indicado. También lo hace 60 veces por segundo.
- **Resize():** Redimensiona los elementos de la pantalla cuando esta cambia de tamaño.
- **Pause()**: Pausa el juego.
- **Resume():** Resume el juego.
- **Hide()**: Hace la pantalla invisible.
- **Dispose():** Elimina los elementos que se puedan eliminar de la pantalla para el ahorro de recursos.

Estos métodos por defecto no contienen nada y deben ser todos ellos implementados desde cero, y en esta sección se mostrarán los elementos más característicos de la implementación que se ha hecho de cada uno de ellos.

Lo primero que hay que tener en cuenta cuando se crea una nueva pantalla es como se va a manejar la relación de aspecto o *aspect ratio*, lo cual también influirá en como escalan los elementos cuando se redimensione la ventana. LibGDX ofrece la solución mediante el uso de *ViewPorts*, con la clase *ViewPort*. Distintos *ViewPorts* emplean distintas estrategias (como por ejemplo, llenar toda la pantalla aunque se pierda el aspect ratio), y en el caso de *Phoenix* se ha decidido usar un *FitViewPort*. Este se caracteriza por mantener la relación de aspecto constante en la resolución dada, y si es necesario para que eso se cumple, crear barras negras en la parte superior e inferior de la pantalla. En monitores de ordenador más cuadrados estas barras pueden ocupar un espacio perceptible de la pantalla, pero sin embargo dan un resultado excelente en dispositivos Android con un aspecto más panorámico.



Se observan las barras negras mencionadas. Estas son prácticamente imperceptibles en Android.

Ilustración 46 Screens - ViewPorts y Cámaras

En segundo lugar, todo *ViewPort* debe tener una cámara asociada. Al igual que en el caso anterior, existen distintos tipos de cámara para diferentes situaciones, pero en este caso se busca una cámara que pueda seguir al jugador desde una posición estática. Para ella usaremos la clase *OrthographicCamera*, cuya posición X e Y se actualizará la par que la del jugador para generar una ilusión de seguimiento en el método *update()*. Por otro lado, hay elementos, como los de la interfaz gráfica o los diálogos, que se desea tener en la pantalla de forma estática. Para estos se creará un nuevo *ViewPort* y una nueva cámara, que en todos los casos, será del mismo tipo que las anteriores.

```
private Stage stage;

public GameScreen(Game game, String map, boolean gravity){
    this.game = game;
    //La cámara que seguirá a nuestro jugador
    this.cam = new OrthographicCamera();
    //FitViewPort se encarga de que el juego funciona en todas las resoluciones
    this.gamePort = new FitViewport(Game.WIDTH / Game.PPM, Game.HEIGHT / Game.PPM, cam);
    this.dialogPort = new FitViewport(Game.WIDTH, Game.HEIGHT, new OrthographicCamera());
    this.stage = new Stage(dialogPort, game.batch);
```

Ilustración 47 Viewports y Cámaras II

El siguiente elemento que se debe tener en cuenta es el *step time*. Como ya se ha mencionado, LibGDX gestiona internamente el complejo "loop principal" del juego, el cual genera los *frames* que van a ser dibujados en la pantalla y su frecuencia. Además, en cada uno de estos "pasos" o frames generados se hará la simulación de todas las físicas actuando sobre el mundo. A estos pasos se les llama *steps,* y la definición que se haga de la variable *step time* dictará la frecuencia con la que se hace este proceso por segundo. En términos más conocidos, *step time* determinará los *FPS* o *Frames per Second* del juego. En el ámbito de los videojuegos se considera que 60 *FPS* es la frecuencia de refresco ideal para que el jugador tenga una experiencia fluida, sin embargo, el número de *FPS* es directamente proporcional al consumo de recursos y posible pérdida de rendimiento. En nuestro caso se plantearon 3 posibilidades: *30, 45 y 60 FPS*. Después de realizar pruebas y monitorizar el impacto en el rendimiento (como se detallará en la sección de *Pruebas*) se llegó a la conclusión de que 60 *FPS* era un número perfectamente viable, y se implementó de esa manera. Sin embargo, esto dio lugar a una nueva preocupación. En base a la potencia de un ordenador o un dispositivo móvil, diferentes números de FPS (los cuales además pueden fluctuar en base a la exigencia técnica del momento del juego) resultaban en distintas velocidades de renderizado y procesamiento en distintos dispositivos. En otras palabras, el juego iba más o menos rápido en según qué dispositivo. Con todo esto en cuenta, se procedió a definir cada *step* dentro de un método propio que contaría con un acumulador, con el tiempo *delta* de actualización del juego, el cual se aseguraría de que la velocidad del juego fuese independiente de los *FPS.* Este método se colocó, obviamente, dentro de update():

```
private void stepWorld() {
    float delta = Gdx.graphics.getDeltaTime();

    accumulator += Math.min(delta, 0.25f);

    if (accumulator >= STEP_TIME) {
        accumulator -= STEP_TIME;

        world.step(STEP_TIME, 6, 2);
    }
}
```

Ilustración 48 GameScreen - StepTime y FPS

Como se había adelantado en la sección de mapas, el siguiente paso es la carga del mapa de la pantalla. Esta resulta muy sencilla gracias a la clase *TmxMapLoader* y la ya vista *TiledMap*, las cuales únicamente necesitarán el archivo .tmx en cuestión ya creado. Se hace uso, además, de una tercera clase relacionada con el renderizado del mapa: *OrthogonalTiledMapRenderer*, la cual fue escogida en base a la cámara empleada y el tipo de vista que se le desea dar al juego.

```
//Carga el mapa hecho en Tiled
mapLoader = new TmxMapLoader();
green_map = mapLoader.load(map);
mapRenderer = new OrthogonalTiledMapRenderer(green_map, 1 / Game.PPM);
//Centra la cámara cuando se ejecuta el juego al principio
cam.position.set(gamePort.getWorldWidth()/2, gamePort.getWorldHeight()/2, 0);
```

Ilustración 49 GameScreen - Carga de mapa

Existe una última variable relacionada con la carga de mapas de extremada importancia en el desarrollo. Ya se ha visto cómo se crean los cuerpos que se van a encontrar en el mapa, pero estos por defecto son invisibles (y deberían serlo, ya que en la versión final del juego no se debería poder más que el elemento gráfico). La clase *B2WorldCreator* se encargará un mundo de simulaciones físicas *Box2D*, mientras que *Box2DDebugRenderer* creará bordes de distintos colores para los cuerpos creados, de forma que no solo sean visibles, sino identificables por su tipo, una funcionalidad clave a la hora de realizar *Pruebas*.

```
//Inicializa variables box2D
world = new World(new Vector2(0,0), true); //No hay gravedad, los cuerpos están "dormidos"
b2dr = new Box2DDebugRenderer(); //Esto nos dibujará los cuadrados verdes de colisión

//Crea el mundo Box2D con el mapa en cuestión
 b2wc = new B2WorldCreator(world, green_map, this);
```

Ilustración 50 - Box2DDebugRenderer

Solamente queda un último aspecto a tener en cuenta en la creación de la pantalla. *Phoenix*, por la filosofía que se le desea aplicar, tiene dos modos de juegos distintos: *top-down* y *sidescroll*. Como se aprecia en la ilustración 39, en el proceso de creación de un mundo se debe indicar si este, en su simulación de magnitudes físicas, debe contar o no con gravedad. En nuestro caso, el modo *top-down* sí debería, mientras que el modo *sidescroll* no. Ya que únicamente contamos con una sola clase y pantalla para mostrar el juego (que no los menús), se optó por introducir un nuevo parámetro llamado *gravity* (ilustración 36) en la constructora de *GameScreen*. Este sería un *boolean* que indicaría si el mundo debe contar con gravedad o no, y la creación del mundo del mapa en cuestión se haría de distinta manera en base a él.

```
//Inicializa variables box2D
if (!gravity){
    world = new World(new Vector2(0,0), true); //No hay gravedad, los cuerpos están "dormidos"
}
else{
    world = new World(new Vector2(0, -10), true);
    sideScroll = true;
}
```

Ilustración 51 GameScreen – Gravedad

Una vez están las bases de la pantalla creada, solo queda por entender la gestión de los dos métodos más complejos de esta*: update() y render()*. Ya se ha visto parte de lo que los compone, y la siguiente funcionalidad destacable es la relacionada con el input y el movimiento y realización de acciones del jugador: *handleInput()* y *handleSCInput()* (encontradas ambas dentro de *update()*). Estos métodos únicamente alteran la posición X e Y del jugador en base a la tecla de movimiento que esté pulsando, y ejecuta acciones en base al mismo criterio. Como se ya explicó en la sección *Controller*, estas acciones están ligadas al controlador de acciones encontrado en *Tools*, el cual es el encargado de gestionarlas.

```
public void handleInput(float delta){

    if(Gdx.input.isKeyPressed(Input.Keys.ANY_KEY)|| Gdx.input.isTouched()) {//Maneja el movimiento de las 4 flechas de dirección
        if (controller.isRightPressed() && mcharacter.b2body.getLinearVelocity().x < TOP_SPEED) {
            mcharacter.b2body.applyLinearImpulse(new Vector2(MCSpeed, 0), mcharacter.b2body.getWorldCenter(), true);
        }
        if (controller.isLeftPressed() && mcharacter.b2body.getLinearVelocity().x > -TOP_SPEED) {
            mcharacter.b2body.applyLinearImpulse(new Vector2(-MCSpeed, 0), mcharacter.b2body.getWorldCenter(), true);
        }
        if (controller.isUpPressed() && mcharacter.b2body.getLinearVelocity().y < TOP_SPEED) {
            mcharacter.b2body.applyLinearImpulse(new Vector2(0, MCSpeed), mcharacter.b2body.getWorldCenter(), true);
        }
        if (controller.isDownPressed() && mcharacter.b2body.getLinearVelocity().y > -TOP_SPEED) {
            mcharacter.b2body.applyLinearImpulse(new Vector2(0, -MCSpeed), mcharacter.b2body.getWorldCenter(), true);
        }
        if (controller.isFirePressed() && !fbLock) {
            mcharacter.fire(1); //Dispara una bola de fuego
            SoundHandler.getSoundHandler().getAssetManager().get("audio/sounds/fireball.wav", Sound.class).play(Game.volume);
            startTime = TimeUtils.nanoTime();
            fbLock = true;
        }
        if (controller.isIcePressed() && !fbLock) {
            mcharacter.fire(2); //Dispara una bola de hielo
            SoundHandler.getSoundHandler().getAssetManager().get("audio/sounds/iceball.wav", Sound.class).play(Game.volume);
            startTime = TimeUtils.nanoTime();
            fbLock = true;
        }
        if (controller.isLightningPressed() && !fbLock) {
```

Ilustración 52 GameScreen – Movimiento

El método *handleSCInput()* resulta extremadamente similar a este, y es el empleado en las secciones sidescroll del juego, por lo que únicamente elimina la posibilidad de realizar ciertas acciones.

En esta sección se encontraron dos de los problemas que más impacto tuvieron tanto en el diseño como en el rendimiento del juego:

- Gestión de destrucción de cuerpos
- Activación de enemigos

Es obvio que cuando, por ejemplo, un enemigo es derrotado este tiene que desaparecer del mapa. Ocurre lo mismo con proyectiles que han impactado a su objetivo o que llevan demasiado tiempo activos. Todos los cuerpos Box2D están almacenados y renderizados desde *arrays* (*en render()*) para poder gestionarlos con más facilidad, por lo tanto el paso lógico para eliminarlos parecía ser eliminarlos del *array* correspondiente. Sin embargo, en todos los casos esto llevaba a una interrupción automática del juego y un error fatal. Esto se debía a que estos cuerpos estaban siendo eliminados dentro de los ya mencionados *steps*, es decir, cuando el mundo estaba realizando cálculos y simulaciones físicas; momento en el cual no se debe eliminar cuerpos. Este error no estaba únicamente reservado a la eliminación de cuerpos, sino también a cambios de pantalla o reposicionamiento instantáneo y anti-natural de cuerpos (como el teletransporte del jugador). La solución que se encontró consistió en la creación de *flags* para cada una de estas actividades problemáticas, de forma que cuando se dieran, solamente activaran el *flag*, y el método *render()* hiciera comprobaciones de estos cuando su *batch* ya estaba cerrado (y por lo tanto la simulación terminada):

```
game.batch.end();

UI.stage.draw();
stage.act();
stage.draw();

controller.draw();

if(dungeonFlag){
    b2wc.getEnemyArray().clear();
    ScreenHandler.getScreenHandler().setDungeonScreen(mcharacter);
}
if(greenMapFlag){
    b2wc.getEnemyArray().clear();
    ScreenHandler.getScreenHandler().setGameScreenBack(mcharacter);
}
if(sideScrollFlag){
    ScreenHandler.getScreenHandler().setSideScrollScreen(mcharacter);
}
if(cityFlag){
    b2wc.getEnemyArray().clear();
    ScreenHandler.getScreenHandler().setCityScreen(mcharacter);
}
if(menuFlag){
    ScreenHandler.getScreenHandler().setMainMenu(mcharacter);
    menuFlag = false;
```

Flags para el cambio de mapas. Existen también para la eliminación de cuerpos y el reposicionamiento de estos. Se encuentran al final del método render().

Ilustración 53 GameScreen – Flags

Por otro lado, ya se ha comentado como los cuerpos activos y dinámicos consumen muchos más recursos que los dormidos, debido a sus cálculos físicos. Este comportamiento no es el deseado para enemigos que, por ejemplo, se encuentran en algún lugar muy distante del jugador, especialmente si hay una gran cantidad de enemigos. Con el fin de optimizar el rendimiento del juego, especialmente de cara a futuras ampliaciones, se decidió "dormir" a estos enemigos lejanos, de forma que el único cálculo que tuvieran que hacer es el de la distancia respecto al jugador para decidir cuándo activarse. El coste de este cálculo es notablemente menor al de su movimiento (como se verá en la sección de enemigos), por lo que la realización de esta implementación compensa y es más eficiente. Actualmente esto se ha realizado de forma que los enemigos se activen aproximadamente cuando entren en la pantalla visible del jugador, calculando la distancia entre dos puntos (sus coordenadas). Esta comprobación se realiza en el método *update()*, tanto de *GameScreen* como de cada enemigo.

```
public void update(float delta) {
    if (!isPaused) {
        //Maneja lo que pulsa el usuario
        if (sideScroll)
            handleSCInput(delta);
        else
            handleInput(delta);

        stepWorld();

        //La cámara sigue al jugador
        cam.position.x = mcharacter.b2body.getPosition().x;
        cam.position.y = mcharacter.b2body.getPosition().y;

        //Actualiza las coordenadas de la cámara
        cam.update();
        //Solo dibujamos en la pantalla la parte del mundo que podemos ver
        mapRenderer.setView(cam);

        //Actualizamos objetos del juego
        for (Coin coin : b2wc.getCoinArray()) {
            coin.update(delta);
            if (coin.isDestroyed()) {
                b2wc.getCoinArray().removeValue(coin, true);
            }
        }
        for (MovingBlock mb : b2wc.getMbArray()) {
            mb.update(delta);
        }

        for (Enemy enemy : b2wc.getEnemyArray()) {
            enemy.update(delta);
            if (!enemy.getBody().isActive()) {
                if (enemy.getBody().getPosition().dst2(mcharacter.b2body.getPosition()) < ACTIVATE_DISTANCE) {
                    enemy.getBody().setActive(true);
                }
            }
            if (enemy.getSetToDestroy()) {
                world.destroyBody(enemy.getBody());
                b2wc.getEnemyArray().removeValue(enemy, true);
            }
        }
```

Ilustración 54 GameScreen - Update()

Método update(), con todos los elementos vistos en esta sección. La comprobación en la condición "if(¡enemy.getBody.isActive()) calcula la distancia a la que se encuentra el jugador, y si es mayor a la deseada el cuerpo permanece dormido para ahorrar recursos.

Resto de pantallas (menús)

Existen muchas clases realizadas para el resto de pantallas. Estas van a representar distintos menús, tales como el menú de inicio, de estado del jugador o de compra/venta de objetos. La implementación de estos es conceptualmente mucho más sencilla que la de *GameScreen*, ya que no se trata con físicas, cuerpos o elementos problemáticos, sino por lo general con tablas, botones y sliders que implementan *Listeners*.

Cada una de estas clases, como era caso en el apartado anterior, implementan la clase *Screen*, por lo que tienen todos los métodos propios de ella. Sin embargo, esta vez tan solo se tocará el método *render()*, el cual se encargará de dibujar los elementos gráficos del *stage* y la imagen de fondo:

```
@Override
public void render(float delta) {
    game.batch.begin();
    Gdx.gl.glClearColor(0,0,0,1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    game.batch.draw(bgTexture, 0, 0, Game.WIDTH, Game.HEIGHT);
    game.batch.end();

    stage.act();
    stage.draw();
}

@Override
```

Stage.act() y stage.draw() activan los listeners y dibujan los elementos respectivamente.

Ilustración 55 Menús – Render

Para el resto de los elementos que compondrán el stage, se han empleado las siguientes clases propias de Scene2D:

- **TextButton**: Botón rectangular.
- **Table**: Tabla de las dimensiones deseadas.
- **Label**: Etiqueta con texto. Usada generalmente dentro de las tablas.
- **Slider**: Control deslizante. Se utiliza para la selección de dificultad y de audio. Scene2D permite elegir cuantas divisiones realizar sobre el Slider.

En la creación de la mayoría de estos elementos se les ha añadido un *ClickListener.* Este *listener* ha resultado especialmente útil para la gestión de *input,* ya que gestiona igualmente los controles en PC como en Android gracias a sus métodos *touchUp()* y *touchDown()*, los cuales funcionan tanto para *clicks* con ratón como para *taps* con los dedos. Adicionalmente, en aquellas etiquetas en las que se debe mostrar su información en la sección de "Descripción" al pasar el ratón por encima, se han implementado los métodos *enter()* y *exit()*. El primero se ejecutará cuando el ratón esté encima del elemento gráfico en cuestión, y el segundo cuando se deje de estar en contacto con él.

Con el fin de cuidar la estética del juego, se ha hecho uso además de la clase *Skin*. Mediante el uso de tres archivos .txt, .json y .atlas que pueden ser encontrados en la sección *GitHub* de *Scene2D,* se podrá customizar la apariencia de los elementos gráficos que se están empleando con más profundidad, además de contar con una fuente y diseño más elaborado. En el caso de Phoenix, se empleó la apariencia por defecto de la *Skin* principal de *Scene2D*.

Por último, aunque la implementación de menús es más sencilla que pasos anteriores, es fácil que la distribución de tablas y elementos sea muy confusa y difícil de "cuadrar". Por esta razón, siempre se usó la opción de todos los elementos *Scene2D setDebug()*. Similar al *Box2DDebugRenderer*, este método colorea los bordes de los elementos de distintos colores para poder así ver su distribución y posición real. Otro de los problemas encontrados fue que, en muchos casos, los elementos no solamente no aparecían en la posición especificada, sino que a veces no salían ni siquiera dentro de la pantalla. Esto se debía a la necesidad de emplear el método *setFillParent()*, el cual le indica al elemento que debe estar dentro de su elemento padre y llenarlo. Además, y como nota final, es importante no olvidar que, ya que toda pantalla contiene un *stage* distinto, para poder hacer uso de él este debe tener el *focus* de la aplicación. Como ya se ha visto previamente, el método *resetIP()* nos permite solucionar este problema también en los menús.

```
descriptionLabel = new Label("Descripción", skin);

initStatsTable(mc);
initOptionsTable();

optionsTable.add(stateLabel).expandX().padBottom(40).width(130).height(60);
optionsTable.row().padBottom(40).width(130).height(60);
optionsTable.add(itemsLabel);
optionsTable.row().padBottom(40).width(130).height(60);
optionsTable.add(equipmentLabel);
optionsTable.row().padBottom(40).width(130).height(60);
optionsTable.add(saveLabel);

statsTable.add(lifeLabel).padBottom(20).padLeft(20);
statsTable.row().padBottom(20).padLeft(20);
statsTable.add(manaLabel);
statsTable.row().padBottom(20).padLeft(20);
statsTable.add(lvlLabel);
statsTable.row().padBottom(20).padLeft(20);
statsTable.add(xpLabel);

mainTable.add(new Image(new Texture(Gdx.files.internal("mc.jpg")))).height(150).padLeft(100).padBottom(100);
mainTable.add(statsTable).left().expand().padBottom(60);
mainTable.add(optionsTable).width(Game.WIDTH / 4);
mainTable.row();
mainTable.add(descriptionLabel).height(Game.HEIGHT / 8).colspan(2);
mainTable.add(backLabel).height(60).width(130);
mainTable.setFillParent(true);

stage.addActor(mainTable);
```

Ilustración 56 Menús - Uso de tablas, botones y labels

```
private void initOptionsTable(){
    backLabel = new TextButton("Atrás", skin);
    backLabel.setColor(Color.DARK_GRAY);
    backLabel.addListener(new ClickListener(){
        @Override
        public void touchUp(InputEvent event, float x, float y, int pointer, int button) {
            super.touchUp(event, x, y, pointer, button);
            screen.dispose();
            ScreenHandler.getScreenHandler().setScreen(previousScreen);
            SoundHandler.getSoundHandler().getAssetManager().get("audio/sounds/back.wav", Sound.class).play(Game.volume);
            previousScreen.getController().resetIP();
            previousScreen.getController().resetBag();
        }

        @Override
        public void enter(InputEvent event, float x, float y, int pointer, Actor fromActor) {
            super.enter(event, x, y, pointer, fromActor);
            descriptionLabel.setText("Vuelve a la pantalla de juego");
        }

        @Override
        public void exit(InputEvent event, float x, float y, int pointer, Actor toActor) {
            super.exit(event, x, y, pointer, toActor);
            descriptionLabel.setText("Descripción");
        }
    });
    stateLabel = new TextButton("Estado", skin);
    stateLabel.setColor(Color.DARK_GRAY);
    stateLabel.addListener(new ClickListener(){
```

Ilustración 57 Menús - Listeners y sus métodos

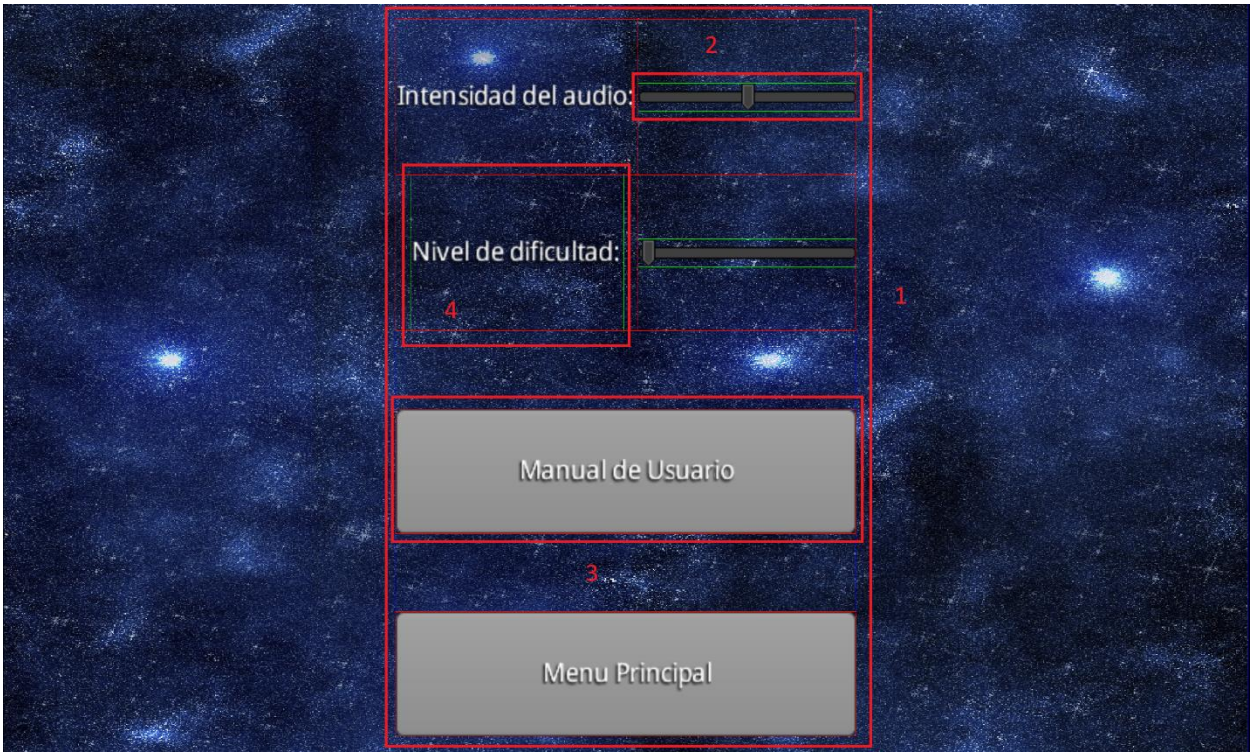


Ilustración 58 Menús - Elementos Scene2D

Se observan en la ilustración los elementos Scene2D con los que se han trabajado: 1. Table, 2. Slider, 3. TextButton, 4. Label. Se pueden ver también las líneas de debugging pintadas de diferentes colores.

Enemies

En esta sección de describirá la creación y comportamiento de los enemigos. Llegados a este punto, la gran mayoría de los elementos de diseño de los enemigos son ya conocidos. A modo de resumen, estos serían:

- Son cuerpos *Box2D* dinámicos y rectangulares.
- Algunos de ellos son sensores, de forma que no exista colisión (como el murciélago).
- Sin embargo, aún en estos casos, todos reaccionan al contacto con el jugador y proyectiles.
- Cada uno tiene su propio *mask bit*.
- Sus polígonos en Tiled tienen propiedades personalizadas, las cuales indicarán en qué dirección patrullarán.
- Su actualización y renderizado se incluyen en los métodos *update()* y *render()* de *GameScreen* respectivamente.
- Algunos de ellos son capaces de lanzar proyectiles.
- Sus cuerpos están dormidos hasta que el jugador se acerca a una distancia predefinida.

Todas estas características ya han sido explicadas con detalle en secciones anteriores (excepto los proyectiles, lo cual se hará en la siguiente), por lo que esta sección se centrará en la característica fundamental y más compleja de los enemigos: su **algoritmo de movimiento**. El movimiento del enemigo constará de dos fases:

- Fase de patrulla. El enemigo patrulla una zona de forma horizontal o vertical.
- Fase de persecución. El jugador ha entrado dentro del *aggro* (distancia a partir de la cual el enemigo empieza a perseguir) del enemigo, y este le persigue. Si el enemigo se aleja una distancia X de su punto de *spawn* o creación inicial, vuelve a este punto, dejando por lo tanto de perseguir al jugador.

En ambos casos, el concepto base es el siguiente: el cálculo de la distancia entre dos puntos y la creación de un vector de fuerza bidimensional que indique cuanta fuerza aplicar a un cuerpo en un punto para que se mueva a otro punto con velocidad constante.

En primer lugar, y como ya se vio en la sección anterior, el cálculo de la distancia entre dos puntos viene dado por la función *dst()* de la clase *Vector2* (vector bidimensional) en Box2D. Sin embargo, es conocido que este cálculo puede resultar particularmente costoso ya que incluye cuadrados y raíces cuadradas. Para los casos, como este, en los que el cálculo preciso de la distancia no es tan relevante como el hecho de tener un número de referencia de distancia, Box2D nos ofrece la función *dst2()*, la cual elimina el cálculo de la raíz cuadrada. Esto resulta en una optimización de uso de CPU notable, y se ha empleado en todos los cálculos de distancia.

Mientras el enemigo está patrullando, únicamente está calculando su distancia actual con el jugador. El método de patrulla simplemente hace que el enemigo se mueva desde su X o Y original (dependiendo de su propiedad individual en *Tiled* que indica de qué manera debe hacerlo), a *X/Y + la distancia deseada*. Cuando llega a este punto límite, cambia el signo a su componente X/Y de su vector de velocidad y vuelve a moverse hasta *X/Y – la distancia deseada*. De esta forma, patrulla o bien de forma horizontal o vertical. La cuestión más compleja es: ¿cómo calcular el vector de fuerza necesario para que el enemigo se mueva de un punto a otro? Una vez más, recurrimos al álgebra lineal para ello. El proceso consiste en:

- Obtener los vectores de posición iniciales de ambos cuerpos.
    - Como son necesarios, se almacenan en la creación de cada enemigo.
- Restar al vector inicio (enemigo) el vector destino (punto destino o jugador).
- Normalizar el resultado de esta operación, generando un vector normalizado.
- Darle al enemigo una velocidad lineal dada por este vector resultado.
    - Dependiendo de la velocidad del enemigo, debe moverse más o menos rápido. Esto se conseguirá escalando el vector resultado por la velocidad del enemigo, gracias al método *scl().*

De esta forma, el enemigo se mueve hacia el punto deseado con velocidad constante. Como este punto destino no cambia, no es necesario volver a realizar los cálculos. Cuando este mismo método se aplica para perseguir al jugador, hay que tener en cuenta que su posición puede (y suele) estar constantemente cambiando, por lo que se llamará en el método *update()* del enemigo, haciendo por lo tanto que se ejecute 60 veces por segundo y esté constantemente corrigiendo el punto destino. El procedimiento para volver al punto inicial es el mismo.

Hay momentos del juego en el que una gran cantidad de enemigos está activa, y muchos de ellos persiguiendo al jugador, lo cual supone la ejecución de este método un gran número de veces por segundo, por lo que el uso de *dst2()* tuvo un gran impacto en el proceso de optimización, consiguiéndose que la experiencia no se viese ralentizada en ningún dispositivo móvil independientemente de su antigüedad.

```
public void move(){
    if(this.body.getPosition().dst2(initialVector) < CHASEDISTANCE && !retreatFlag) {
        if (direction.equals("horizontal") && (body.getPosition().dst(screen.getMcharacter().b2body.getPosition())) > AGGRO) {
            this.body.setLinearVelocity(movSpeed, 0);
            if(body.getPosition().dst2(initialVector) > 5 && !changeDirections){
                reverseVelocity();
                changeDirections = true;
            }
            if((int) body.getPosition().x == (int) initialX && (int)body.getPosition().y == (int) initialY) {
                changeDirections = false;
            }
        } else if (direction.equals("vertical") && (body.getPosition().dst(screen.getMcharacter().b2body.getPosition())) > AGGRO) {
            this.body.setLinearVelocity(0, movSpeed);
            if(body.getPosition().dst2(initialVector) > 5 && !changeDirections){
                reverseVelocity();
                changeDirections = true;
            }
            if((int) body.getPosition().x == (int) initialX && (int)body.getPosition().y == (int) initialY) {
                changeDirections = false;
            }
        }
        else{
            chase(screen.getMcharacter());
            chasing = true;
        }
    }
    else{
        if(!retreatFlag){
            retreatFlag = true;
            chasing = false;
        }
        else{
            directionVector = initialVector.sub(body.getPosition());
            directionVector.nor();
            body.setLinearVelocity(directionVector);
            initialVector.x = initialX;
            initialVector.y = initialY;
            if((int) body.getPosition().x == (int) initialX && (int)body.getPosition().y == (int) initialY) {
                retreatFlag = false;
            }
        }
    }
}
```

Ilustración 59 Enemies - Algoritmo de movimiento

```
private void chase(MainCharacter mc){
    body.setLinearVelocity((mc.b2body.getPosition().sub(body.getPosition())).nor().scl(SCSpeed));
}
```

Ilustración 60 Enemies - Persecución

Uno de los elementos problemáticos que se encontró con los enemigos fue su animación de muerte. Cuando un enemigo es derrotado, cae al suelo con una animación característica y distinta a las de movimiento. Las animaciones de movimiento consisten, generalmente, de nueve *sprites* distintos, mientras que la animación de derrota tiene tres. Esta disparidad de *sprites* hacía que el método de animación estándar reprodujese una animación demasiado lenta (este problema también se daba, en menor medida, con las animaciones de disparo de proyectiles de los elfos). Para solucionarlo, se re-escaló (en la medida necesaria) la variable *statetimer*, uno de los parámetros que toma la clase *Animation* y que usa el parámatro *delta* del propio juego.

Projectiles

Como el propio nombre indica, en este paquete se encuentran los distintos tipos de proyectiles implementados, lanzados tanto por el jugador como por los enemigos. Como era el caso con los enemigos, todos los elementos característicos de los proyectiles han sido ya descritos. Un resumen de estos es:

- Son cuerpos dinámicos y circulares.
- Están animados.
- Tienen su propio *mask* bit.
- Desaparecen al cabo de 3 segundos de haber sido creados, de forma que no afecten al rendimiento una vez se encuentren fuera de la visión del jugador.
- Su eliminación se gestiona con flags para evitar que se haga dentro de la simulación de físicas.
- Su actualización y renderizado se realiza en *GameScreen*.

Sin embargo, se encontró un problema de altísima relevancia a la hora de realizar pruebas extensivas en lo relacionado con los proyectiles. Tal y como se programó en un principio, la creación de proyectiles consistía en primer lugar en la creación del cuerpo, al cual se le asignaba una animación. El problema era que, en los casos en los que la creación del proyectil se creaba sobre otro cuerpo (por ejemplo, cuando el jugador y el enemigo están literalmente pegados el uno al otro), la colisión eliminaba el cuerpo demasiado rápido y la animación se intentaba reproducir sobre un cuerpo que no existía. Aunque técnicamente no es un problema difícil de solucionar (se solucionó añadiendo una textura sin animación por defecto en la creación de cuerpos), fue un problema difícil de diagnosticar y que ocasionaba *crashes* de forma inconsistente, ya que no se daba el problema en el 100% de los casos.

Items

En esta sección se encuentra todo lo relacionado con los objetos del juego, tanto consumibles como equipables. Estos no tienen ningún tipo de elemento gráfico o de animación, y su desarrollo se realizó prácticamente sin errores. Para una mayor facilidad a la hora de mostrar estos objetos en los menús, se crearon dos tipos de inventarios para el jugador: el de objetos *consumibles* y el de objetos *equipables.* Estos se implementaron con los *Arrays* propios de LibGDX, los cuales tienen métodos internos de búsqueda y eliminación optimizados en cuanto a rendimiento, y desde el conocimiento de que el tamaño de estos *Arrays* siempre sería pequeño. Todas las descripciones y característicos de los objetos vienen dados por archivos XML. El proceso de lectura de estos viene especificado en la sección de *Análisis y Diseño – Modelo de Dominio a Base de Datos*.

```
public class Weapon extends EquipableItem {

    public Weapon(String group, String pType){
            XmlReader.Element type = getNode(group, pType);

            name = type.get("name");
            description = type.get("description");
            effect = type.getInt("effect");
            sellPrice = type.getInt("sellprice");
            buyPrice = type.getInt("buyprice");
    }

    private XmlReader.Element getNode(String group, String pType){
        try {
            XmlReader.Element root = xml.parse(Gdx.files.internal("xml/equipment.xml"));
            return root.getChildByName(group).getChildByName(pType);
        }
        catch (Exception e){
            e.printStackTrace();
            return null;
        }
    }
}
```

Ilustración 61 Items - Atributos y lectura

Characters

En este último paquete se encuentra las clases *MainCharacter* y *NPC*. Todos sus elementos característicos han sido ya descritos previamente.

VALIDACIÓN Y PRUEBAS

Toda aplicación debe contar con un proceso que permita validar correctamente su funcionamiento, y que además permita comprobar que, ante posibles cambios, las funcionalidades ya implementadas siguen funcionando sin nuevos errores. Esto es especialmente importante en el desarrollo de videojuegos, ya que con la cantidad de campos interconectados existentes, es muy fácil que un nuevo cambio genere errores en el sitio menos esperado. Además de este tipo de pruebas, hay que ser conscientes de que la aplicación se podría ejecutar en dispositivos con hardware limitado, y por lo tanto la simulación de esto y las pruebas relacionadas con el rendimiento son también de vital importancia. En esta sección se hablará de las pruebas desde dos ángulos distintos:

- Pruebas propias de las funcionalidades del juego.
    - Realizadas por el propio desarrollador y *beta testers*,
- Pruebas de rendimiento.

Funcionalidades

En esta sección se hablará sobre cómo se han probado las funcionalidades propias y básicas del juego. En estas se buscará, sobre todo, comprobar que no hay lugar a *crashes* y que las funcionalidades son sólidas y generan exactamente el comportamiento deseado.

Pruebas del desarrollador

En lo referente a las funcionalidades, muchas de las que fueron problemáticas (así como sus soluciones) han sido detalladas en el apartado de *Desarrollo*, ya que estas soluciones requerían una explicación técnica más detallada. Además, es importante destacar que las pruebas de las funcionalidades del juego no están automatizadas. Las razones de esta decisión son:

- Falta de tiempo
- Aunque el juego es complejo, en su estado actual es corto (tiene una duración media de una hora aproximadamente). No hay un número muy elevado de funcionalidades a probar, y se estimó que se podría probar "a mano".

Por ello, el procedimiento que se siguió en la gran mayoría de los casos era el de implementar las funcionalidades, y en cada paso ejecutar el juego y ver si se representaba en la pantalla la funcionalidad buscada. Ya que la cantidad de errores que se han dado es demasiado alta como para detallarlos todos en esta sección (y los más relevantes han sido ya descritos en la sección de *Desarrollo*), se procederá a mostrar las pruebas finales y globales que se han realizado, y que se realizarían de nuevo si se cambiasen funcionalidades:

Funcionalidad – Movimiento del jugador	Resultado
El jugador se mueve en las 4 direcciones – Mapa de Bosque	Correcto
El jugador se mueve en las 4 direcciones – Mapa de Cueva	Correcto
El jugador se mueve en las 4 direcciones – Mapa de Ciudad	Correcto
El jugador se mueve en las 2 direcciones y salta– Mapa Sidescroll	Correcto
El jugador se mueve correctamente después de disparar una bola de fuego (Todos los mapas excepto Sidescroll)	Correcto
El jugador se mueve correctamente después de disparar una bola de hielo (Todos los mapas excepto Sidescroll)	Correcto
El jugador se mueve correctamente después de disparar una bola de rayo (Todos los mapas excepto Sidescroll)	Correcto
El jugador se mueve correctamente después de usar una poción de vida	Correcto
El jugador se mueve correctamente después de usar una poción de maná	Correcto
El jugador se mueve correctamente después de colisionar con cuerpos estáticos	Correcto
El jugador se mueve correctamente después de colisionar con enemigos	Correcto
El jugador se mueve correctamente durante la colisión con enemigos	Correcto

El jugador se mueve correctamente después de colisionar con proyectiles	Correcto
El jugador se mueve correctamente después de cerrar el menú principal	Correcto
El jugador se mueve correctamente después de la interacción con un NPC	Correcto
El jugador se mueve correctamente después de la compra/venta de objetos	Correcto
El jugador se mueve correctamente después de caer al agua y reaparecer – Solo Sidescroll	Correcto
El jugador escala las escaleras, y si deja de estar en contacto con ellas cae	Correcto

Tabla 34 Pruebas - Movimiento del jugador

El movimiento del jugador es absolutamente básico para poder jugar y disfrutar del juego, por lo que en su versión final todas las funcionalidades son sólidas y correctas. El mayor de los problemas en lo referente al movimiento es la pérdida de *focus*, gestionado por el *InputProcessor*. Los problemas que se han dado, así como sus soluciones, ya han sido detallados previamente.

Funcionalidad – Movimiento de los enemigos	Resultado
Los enemigos están dormidos hasta que el jugador entra en su rango de detección	Correcto
Los enemigos patrullan sobre el eje indicado en las propiedades de Tiled	Correcto
Los enemigos cambian su dirección cuando chocan con un cuerpo estático	Correcto
Los esqueletos, orcos y plantas come-hombres detectan y persiguen al jugador cuando este entra en el aggro indicado	Correcto
Los esqueletos, orcos y plantas come-hombres persiguen al jugador actualizando su ruta constantemente	Correcto
Cuando los esqueletos, orcos y plantas come-hombres están lo suficientemente lejos de su spawn point, vuelven a este	Correcto
Cuando el jugador se aleja lo suficiente del enemigo que le persigue, este último vuelve a su spawn point	Correcto
Los esqueletos, orcos y plantas come-hombres se mueven de un punto a otro esquivando obstáculos, aunque haciendo esto último de forma más lenta	Semi - Correcto
Los murciélagos y elfos oscuros no persiguen al jugador	Correcto
Los murciélagos y elfos oscuros no patrullan	Correcto

Los esqueletos se mueven con velocidad alta; igual a la del jugador	Correcto
Los orcos se mueven con velocidad lenta	Correcto
La planta come-hombres se mueve con velocidad media	Correcto

Tabla 35 Pruebas - Movimiento de los enemigos

Como ya se ha explicado, el algoritmo de movimiento de los enemigos es seguramente el más complejo de todo el juego. Todas las funcionalidades funcionan correctamente, excepto un caso en particular (el cual es muy difícil que se dé). Cuando un enemigo choca desde cualquier ángulo que no sea totalmente perpendicular con un elemento estático, el enemigo rodea este cuerpo, esquivándolo (de forma lenta para dar tiempo a escapar al jugador). Sin embargo, si el ángulo con el que colisiona con el elemento es totalmente perpendicular a este, no lo esquiva correctamente. Esto no es un problema a la hora de perseguir al jugador, ya que el constante movimiento del jugador hace que sea imposible que este ángulo sea constante, pero sí es problemático cuando está volviendo a su punto inicial. Como ya se ha dicho, es extremadamente improbable que el enemigo choque contra un cuerpo estático de forma absolutamente perpendicular (fue muy costoso reproducir el error), pero en el caso de que se dé, se quedaría andando contra el elemento de forma indefinida. La solución de este error queda como trabajo futuro.

Funcionalidad – Colisión del jugador	Resultado
El jugador colisiona con cuerpos estáticos	Correcto
El jugador colisiona con cuerpos dinámicos	Correcto
El jugador colisiona con cuerpos cinéticos	Correcto
El jugador pierde vida al colisionar con un enemigo	Correcto
El jugador adquiere iframe (inmortalidad) durante 3 segundos después de colisionar con un enemigo	Correcto
El jugador sigue recibiendo daño al colisionar con un enemigo después del iframe	Correcto
El daño sigue sucediendo si la colisión no cambia (el enemigo no se ha separado del jugador)	Correcto
El jugador pierde la vida correspondiente al chocar con un proyectil enemigo	Correcto
Se dispara el diálogo correspondiente cuando el jugador colisiona con un cuerpo estático de tipo NPC	Correcto
Cuando el jugador colisiona con un cofre, este se abre y muestra un mensaje indicando que hay en su interior (si no hay mensaje quiere decir que no hay nada)	Correcto

Cuando el jugador colisiona con una moneda, esta desaparece y se suma una cantidad aleatoria dentro de un rango predeterminado al dinero total del jugador	Correcto
El juego no se interrumpe cuando un enemigo lanza bolas mientras colisiona con el jugador y estas no llegan a ser visibles	Correcto
El jugador se transporta correctamente a la nueva localización al chocar con una puerta, yendo a las coordenadas y mapa especificadas	Correcto
El jugador vuelve al comienzo del nivel cuando colisiona con el agua – Solo Sidescroll	Correcto

Tabla 36 Pruebas - Colisión del jugador

Funcionalidad – Colisión de los enemigos y plataformas	Resultado
El murciélago es un sensor; no hay colisión real con el jugador, pero sí detección	Correcto
Cuando el enemigo choca con el jugador, se le hace un daño proporcional a sus puntos de ataque y se cumple el cálculo de la fórmula de daño	Correcto
Cuando el enemigo colisiona con un proyectil del jugador, sufre un daño proporcional a sus puntos de defensa y se cumple el cálculo de la fórmula de daño	Correcto
Cuando dos enemigos chocan entre ellos, estos cambian la dirección del eje sobre el que se estaban moviendo a la contraria	Correcto
Cuando dos enemigos chocan entre ellos, estos no sufren daño	Correcto
Cuando una plataforma choca con un sensor, cambia su dirección	Correcto
Cuando el jugador se monta en una plataforma, esta no sufre efectos de fuerza ni de gravedad y mantiene su ruta	Correcto

Tabla 37 Pruebas - Colisión de los enemigos

Funcionalidad – Funcionalidades del jugador	Resultado
El jugador adquiere la experiencia correspondiente al derrotar a un enemigo	Correcto
Cuando el jugador llega a la experiencia necesaria, sube de nivel (y se refleja en el menú)	Correcto
Cuando el jugador sube de nivel, se aumentan sus estados (y se reflejan en su menú)	Correcto

El daño de los proyectiles del jugador se calcula correctamente en base a sus estados y la fórmula de daño	Correcto
Cuando la vida del jugador llega a cero, este muere	Error
Cuando el jugador usa una poción de vida, su vida aumenta en la cantidad en cuestión	Correcto
Cuando el jugador usa una poción de maná , su maná aumenta en la cantidad en cuestión	Correcto
Si la vida o maná están en su cantidad máxima, el uso de pociones no modifica el maná o vida actual	Correcto
Si el resultado de la recuperación de vida o maná excede el maná o vida máximo, el resultado es el maná o vida máximo	Correcto

Tabla 38 Pruebas - Funcionalidades el jugador

En la versión final del juego, el jugador no puede morir. La funcionalidad está implementada, pero desactivada. Esto es así por dos razones:

- Mayor facilidad a la hora de realizar pruebas para el desarrollador
- Para que, si alguien está probando el juego, lo pueda hacer de manera más sencilla. Es posible morir incluso en la dificultad más sencilla si el jugador no es cuidadoso, y para que la experiencia del jugador sea tranquila y continuada, se ha decidido desactivar la muerte. Como es obvio, esta funcionalidad se reactivaría para la versión comercial.

Funcionalidad – Proyectiles	Resultado
El uso de proyectiles es posible después de: • *El uso de pociones* • *El uso de menús* • *El uso de otro proyectil* • *Mantener otras teclas pulsadas* • *Cualquier tipo de colisión* • *Cualquier tipo de movimiento*	Correcto
El uso de proyectiles no es posible cuando: • *Se está en un mapa Sidescroll* • *La habilidad está en cooldown*	Correcto
El uso de los distintos proyectiles resulta en el consumo de maná apropiado	Correcto
El daño de la bola de fuego y rayo es el indicado, y respeta el cálculo de la fórmula de daño en base a su daño base y estados del jugador	Correcto
La bola de hielo no hace daño	Correcto
Cuando la bola de hielo impacta sobre un enemigo, la velocidad de este se ve reducida a la	Correcto

mitad	
Los proyectiles del jugador no colisionan con los cuerpos estáticos de tipo Roca	Correcto
Los proyectiles colisionan con el resto de elementos del juego	Correcto
Todos los proyectiles, tanto del jugador como de los enemigos, desaparecen a los 3 segundos de ser creados	Correcto
Los proyectiles del jugador tienen un cooldown de 1 segundo	Correcto
El daño de los proyectiles enemigos es el indicado, y respeta el cálculo de la fórmula de daño en base al daño base de los enemigos y la defensa del jugador	Correcto
Los proyectiles del jugador solo pueden ser usados una vez la habilidad ha sido desbloqueada	Error
El impacto de los proyectiles sobre cuerpos dinámicos genera una ligera reacción física sobre estos, empujándolos ligeramente hacia atrás	Correcto
El impacto de los proyectiles sobre cuerpos estáticos o cinéticos no resulta en ningún tipo de reacción física	Correcto
La creación de proyectiles que no llegan a visualizarse en la pantalla (debido a la colisión instantánea con algún cuerpo) no resulta problemática	Correcto

Tabla 39 Pruebas – Proyectiles

Al igual que con la muerte del jugador, la funcionalidad de desbloqueo de habilidades ha sido implementada pero desactivada. El razonamiento detrás de esta decisión es el mismo que en el caso anterior. Sin embargo, en vista de la velocidad a la que se sube de nivel y la variedad de enemigos que existen desde el primer mapa, es probable que esta funcionalidad no se encuentre en la versión comercial del juego (este razonamiento se detallará más en la sección de *Beta Testers*).

Funcionalidad – Interfaces	Resultado
Para todas las interfaces se cumple que: • *Su botón "atrás" o "volver" lleva a la pantalla anterior* • *La creación de un nuevo menú tiene el focus*	Correcto
En la pantalla de Settings se cumple que: • *El movimiento del slider de audio resulta en la alteración en tiempo real de la intensidad del audio*	Correcto

• *Los valores elegidos para "audio" y "dificultad" son correctamente llevados a la nueva partida*	
Se puede usar pociones desde la pantalla de "Objetos"	Correcto
Cuando se usa una poción, su efecto se refleja en los estados del jugador y se elimina de su inventario (esto se refleja también en la interfaz principal de GameScreen)	Correcto
Se puede equipar armas y armaduras desde el menú de "Equipo"	Correcto
Los cambios en el equipo se ven reflejados en: • *Los estados del jugador* • *El propio menú de equipo* • *El menú de estados del jugador*	Correcto
Cuando se interactúa con un cofre aparece el diálogo informativo correspondiente	Correcto
Cuando se interactúa con un NPC, aparece el diálogo de convesación correspondiente	Correcto
Cuando se vende un objeto, este desaparece del inventario y se suma el dinero de la venta al del jugador. Este cambio se refleja en la interfaz de GameScreen	Correcto
No es posible comprar un objeto si no se tiene el dinero suficiente	Correcto
Cuando se compra un objeto, se le resta al jugador el dinero correspondiente. Este cambio se refleja en la interfaz de GameScreen	Correcto
Cuando se compra un objeto, este no desaparece de la lista de venta del NPC	Correcto
Cuando se compra un objeto, este pasa al inventario del jugador. Esto se refleja en la interfaz "Equipo"	Correcto

Tabla 40 Pruebas - Interfaces

Funcionalidad – Sonidos y música	Resultado
Existen sonidos cortos distintos para los siguientes elementos del juego: • *Selección de menús* • *Vuelta atrás en menús* • *Uso de pociones* • *Bola de fuego* • *Bola de rayo* • *Bola de hielo* • *Daño recibido* • *Apertura de un cofre*	Correcto

• *Adquisición de dinero* • *Salto – Solo Sidescroll* • *Comienzo de la partida* • *Selección de menú erróneo o sin implementar* • *Uso incorrecto de objetos o habilidades* • *Impacto de proyectiles en enemigos*	
Todos los sonidos del apartado anterior esperan a que termine el anterior sonido para reproducirse	Correcto
Existen temas musicales para los siguientes mapas o menús: • *Menú de inicio* • *Mapa Bosque* • *Mapa Cueva* • *Mapa Ciudad* • *Mapa Sidescroll*	Correcto
El cambio de mapa resulta en el cese del tema actual y el comienzo del nueva tema musical	Correcto
La apertura de menús in-game no afecta al tema musical que está sonando	Correcto

Tabla 41 Pruebas - Sonidos y música

Funcionalidad – Animaciones	Resultado
Los siguientes elementos del juego cuentan con animaciones: • *Movimiento del jugador en 4 direcciones* • *Movimiento de los enemigos en 4 direcciones* • *Proyectiles del jugador en 4 direcciones* • *Ataque de elfos oscuros* • *Muerte de los enemigos* • *Muerte del jugador* • *Monedas*	Correcto
Los siguientes elementos del juego cuentan con texturas/sprites: • *Proyectiles enemigos* • *Mapas* • *Fondos de menús* • *Controlador – Solo Android*	Correcto

Tabla 42 Pruebas – Animaciones

Uno de los problemas que se encontró a la hora de realizar pruebas ágiles fue el salto entre mapas. Cuando el desarrollo del juego alcanzó un punto avanzado y se buscaba probar algo del último mapa, se debía pasar por todos los anteriores. Esto, dado el alto número de pruebas que se debía realizar,

suponía una gran pérdida de tiempo. Por esta razón se decidió implementar "portales" en el primer mapa. Estos portales serían cuerpos estáticos invisibles que actuarían a modo de puertas cercanas. Entrando en contacto con estos portales, los cuales tenían en su atributo *Tiled* a qué mapa apuntaban, se logró acortar en gran medida el tiempo requerido para las pruebas en la fase final del desarrollo. Estos portales han sido eliminados en la versión final.

Beta Testers

Es importante que, una vez se ha concluido el desarrollo de cualquier aplicación, esta sea probada por otras personas. Esto es especialmente relevante cuando se trata con videojuegos, ya que no solamente querremos que otras personas puedan descubrir errores técnicos en la aplicación, sino también elementos que hacen que la experiencia de usuario, sensaciones o progresión (elementos clave en un juego videojuego) se vea perjudicada. Por estas razones, 12 *beta testers* fueron escogidos para probar *Phoenix*. Con su participación se buscaba:

- Buscar errores técnicos. Se les pidió:
    - Que buscasen errores en las colisiones de forma exhaustiva. Esto incluía forzar situaciones como estar totalmente rodeado de enemigos, intentar "caerse" de algún mapa, buscar huecos en elementos sólidos (como rocas o árboles), etc.
    - Que buscasen errores en las interfaces y menús. Esto incluía hacer click sobre los bordes de los elementos gráficos y ver que funcionaba correctamente, ir hacia adelante y hacia atrás en las interfaces en repetidas ocasiones y comprobar que no afectaba al rendimiento ni a las funcionalidades del juego, hacer compra/venta/uso de pociones de forma exhaustiva, etc.
    - Que buscasen errores en las funcionalidades del juego. Esto incluía subir al máximo nivel posible, usar y equipar objetos en una gran variedad de situaciones, intentar "confundir" a los enemigos en su persecución y observar su comportamiento, etc.
    - En general, que jugasen al juego y probasen de forma exhaustiva todos los elementos que se les ocurriera. Estos beta testers, como es lógico, no eran profesionales, por lo que su conocimiento técnico y tiempo para realizar las pruebas era limitado. Por esto, el objetivo era sobre todo comprobar que el usuario medio puede jugar una partida sin dificultades técnicas, y que buscasen errores en la medida de lo posible con las directrices dadas.
- Buscar elementos perjudiciales para la experiencia de usuario. Este es el objetivo principal de las pruebas mediante *beta testers*. Al no haber un enorme número de funcionalidades técnicas y no ser larga la duración del juego, las dificultades técnicas, aunque siempre pueden estar presentes, pueden ser controladas en cierta medida. Sin embargo, el desarrollador suele perder la percepción de como equilibrar la experiencia del usuario, de cómo hacerla disfrutable para el usuario medio. Por esto, se les pidió que respondieran a las siguientes cuestiones, las cuales se encuentran exactamente de la misma forma en el archivo que se les envió:
    - ¿Es Phoenix claro en su presentación? ¿Da lugar a confusiones en un principio?
    - ¿Son los controles intuitivos?
    - ¿Es la interfaz de Android intuitiva y fácil de usar?
    - ¿Has usado el manual de usuario del menú de ajustes? Si es así, ¿consideras que ha sido útil?
    - ¿Resulta Phoenix visualmente atractivo?
    - ¿Resulta familiar la temática y la apariencia del juego?
    - ¿Es la progresión de Phoenix satisfactoria?

        - ¿Se sube lo suficientemente rápido de nivel?
        - ¿Es el número de habilidades satisfactorio?
        - ¿Tienes sensación de progreso y satisfacción?
        - ¿Es el número de objetos y armaduras disponibles suficiente?
    - ¿Son los mapas completos, amplios y explorables? ¿Dan lugar al deseo de explorarlos y descubrir lo que hay en los cofres ocultos?
    - ¿Disfrutas de la mezcla de distintos tipos de *gameplay* en los distintos mapas? (Mazmorra, Sidescroll…)
    - ¿Consideras alguno de estos tipos de *gameplay* difícil o aburrido?
    - ¿Están los modos de dificultad equilibrados? (Por ejemplo, ¿es el modo fácil demasiado fácil o el difícil demasiado difícil?)
    - ¿Es el modo Sidescroll demasiado difícil?
    - ¿Es el sonido, tanto efectos de sonido como temas musicales, agradable? ¿Consideras que es un punto fuerte de Phoenix?
    - ¿Son los menús intuitivos y fácilmente navegables?
    - ¿Es fácil la navegación de menús en dispositivos móvil?
    - Si tu dispositivo móvil tiene un tamaño de pantalla inferior a 5", ¿es fácil jugar en ella?
    - Realiza, si lo deseas, algún comentario sobre tu experiencia o cuestiones que no hayan sido incluidas arriba.
- Observar el rendimiento en su dispositivo móvil. Por suerte, cada uno de los *beta testers* contaba con un dispositivo móvil diferente, siendo estos desde dispositivos de última generación hasta dispositivos con 5 años de antigüedad. Se les pidió que dieras sus impresiones del rendimiento que presentaba el juego en sus dispositivos, y que monitorizaran, de forma sencilla, su uso de CPU y memoria RAM. Este proceso está explicado con detalle en la sección de *Rendimiento*.

Una vez se recogieron las respuestas de los *beta testers*, las conclusiones fueron las siguientes: En general, las impresiones fueron altamente positivas. La progresión, estética, variación de tipos de gameplay, apartado sonoro, menús y usabilidad en Android y rendimiento recibieron notas extremadamente altas.
Algunas cuestiones a mejorar que fueron destacadas fueron las siguientes:

- La mayoría de usuarios no leyeron el menú de usuario, por lo que no conocían el esquema de controles. Esto resultó en confusión al comenzar el juego. Es necesario añadir una serie de diálogos iniciales, al comenzar el juego, que guíen al usuario en estos primeros pasos.
- El nivel de dificultad "normal" resultó el más satisfactorio para los usuarios, mezclando la sensación de progresión con la superación de retos. El modo "fácil" resultó demasiado fácil, y el difícil demasiado difícil. Estos dos modos de dificultad deben ser ajustados.
- Algunos usuarios reportaron haberse perdido en los dos primeros mapas y no saber continuar. Los dos primeros (en especial el primero), no son tan lineales ni pequeños como los dos siguientes y buscan incentivar la exploración del usuario. Aun así, se añadirán elementos gráficos a modo de guía para alcanzar con mayor facilidad, si el usuario lo desea, el siguiente mapa (por ejemplo, un camino de tierra que señale el camino directo al siguiente mapa).
- Algunos usuarios no llegaron a encontrar la tienda de objetos por su cuenta. Actualmente, esta tienda se encuentra en la esquina inferior derecha del tercer mapa. Las conversaciones con distintos NPC dan a entender su localización, pero aun así, su presencia no es 100% clara. Como se busca mantener la sensación de sorpresa y de exploración de mapas, se ajustarán los diálogos para ser ligeramente más explicativos, pero no se mostrará explícitamente su localización.

- La primera sección de plataformas resultó difícil a prácticamente todos los usuarios. Se ajustará la distancia entre plataformas de forma que las plataformas superiores no supongan un obstáculo para estos saltos.
- El rendimiento, cuestión de alta importancia, fue satisfactorio en todos los dispositivos. El consumo de recursos era el esperado y mantenía los 60 FPS en todos ellos.
- Los usuarios no encontraron errores técnicos ni *crashes*, lo cual resultó sorprendente y muy positivo. Sin embargo, se descubrió un *bug* en el cierre de la aplicación. Debido al funcionamiento del *assetManager* y la carga de *assets* y al hecho de que, cuando en un dispositivo Android se le da hacia "atrás" para cerrar una aplicación esta en realidad no si cierra, la carga de *assets* en una segunda ejecución del juego da lugar a *assets* totalmente opacos y negros. Esto se debe a que no se ha llegado a eliminar completamente el *assetManager* de la sesión de juego anterior y se genera un error en la carga. Aunque se planea solucionar este bug, la solución actual pasa por cerrar el juego correctamente, mediante el administrador de aplicaciones de Android accesible desde cualquier menú en la mayoría de los móviles.

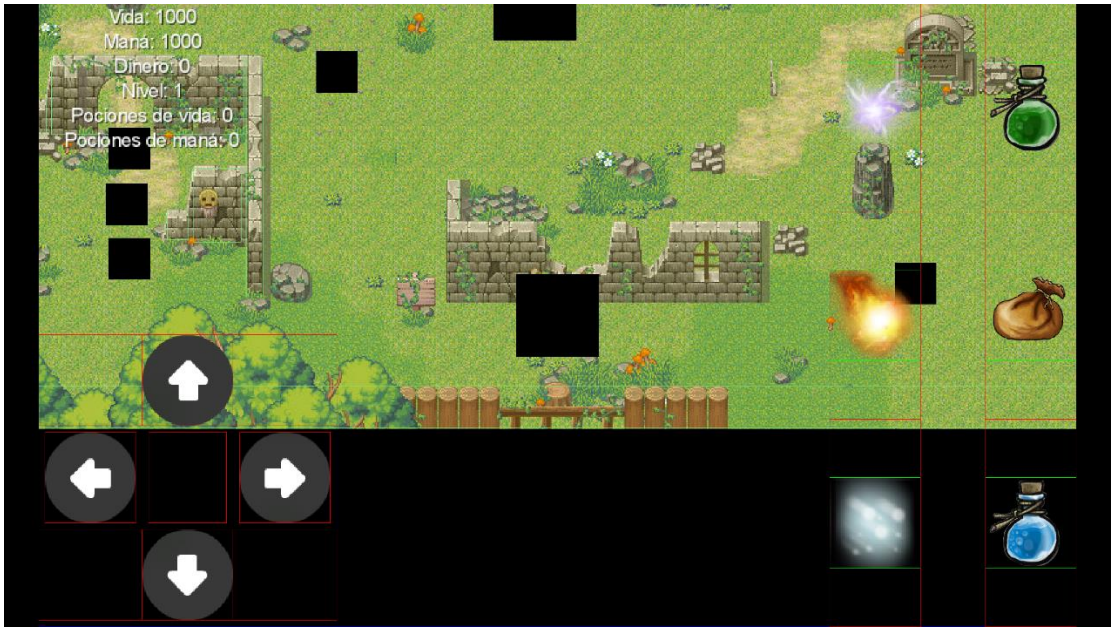


Ilustración 62 Pruebas - Bug de Carga de Texturas

Rendimiento

Ya se ha adelantado en la sección anterior por qué asegurar el rendimiento óptimo de la aplicación, en todos los dispositivos, es de vital importancia. Ya se ha hablado de todas las medidas que se han adoptado para mejorar el rendimiento de distintos elementos del juego, y ahora se procederá a explicar cómo se monitorizaron.

PC

La monitorización de recursos en PC resultó notablemente más sencilla que en dispositivos Android. Sobre todo, se hizo uso del administrador de tareas y su capacidad de monitorización aproximada.

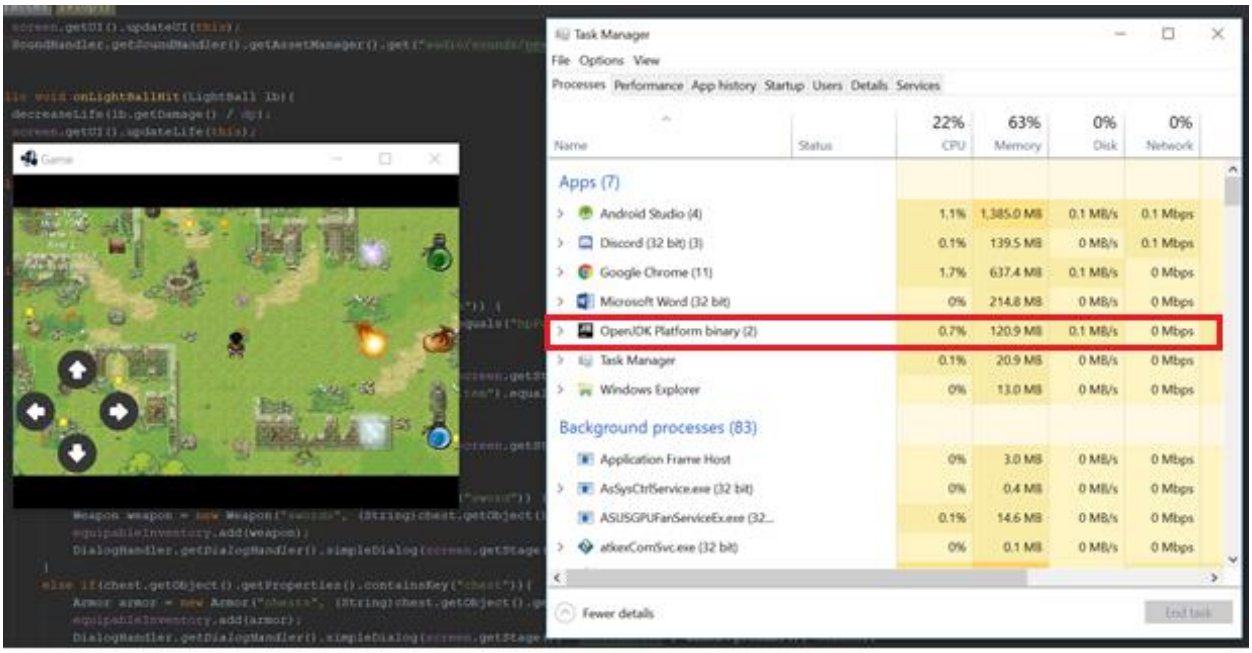


Ilustración 63 Pruebas - Administrador de tareas

Como se observa en la ilustración, el consumo de memoria es sorprendentemente alto. Esto se debe a que, cuando se ejecuta el juego desde *AndroidStudio*, este consume mucha más memoria RAM de lo que consumiría ejecutando únicamente al archivo .jar. Siendo conscientes de esto, esta cifra se utilizó sobre todo como referencia, para ver las variaciones de consumo que había mientras se ejecutaba el juego y para comprobar que, cuando se dejaba ejecutado durante largos periodos de tiempo, no había *memory leaks* o filtraciones de memoria. Por ejemplo, se observó que prácticas como modificar el canal alpha de las texturas utilizadas o emplear archivos .wav / .ogg para los sonidos, resultaron en una mejoría notable de consumo de memoria. Otro ejemplo notable fue como, antes de realizar el proceso de optimización de carga de animaciones, se consumía aproximadamente el triple de memoria. El resto de procesos de optimización técnica están indicados en la sección de *Desarrollo*.

Para la monitorización del consumo de GPU en particular, se usó la aplicación ASUS GPU Tweak II, la cual monitoriza en tiempo real todos los elementos referentes a este consumo.

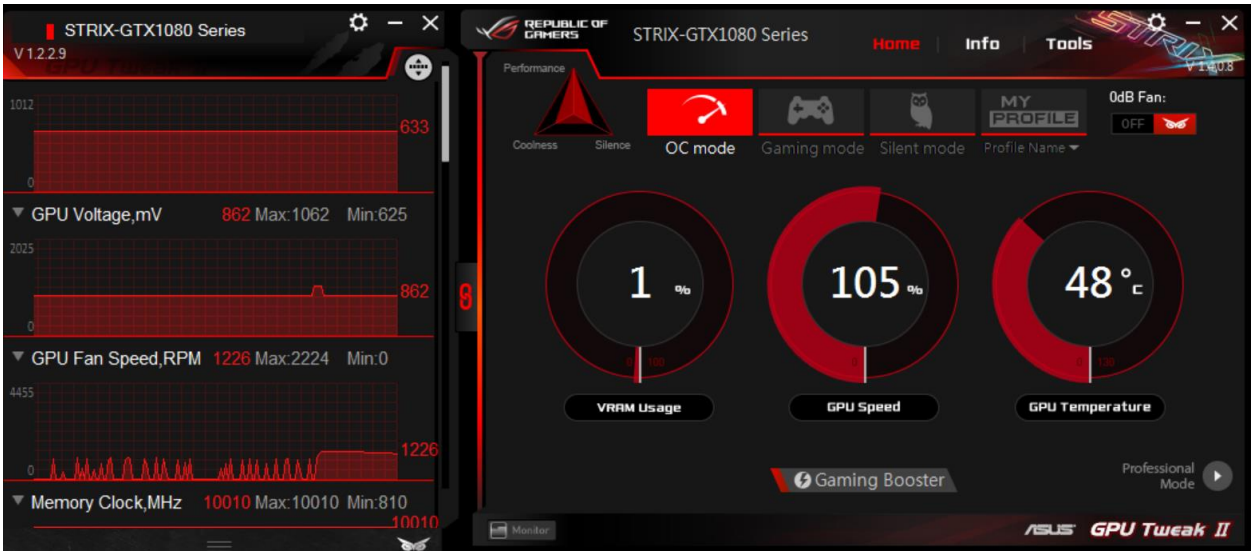


Ilustración 64 Pruebas - Monitorización de GPU

Android

La optimización en Android resultó más problemática desde el primer momento. En primer lugar, se encontró un problema que dio lugar a enfoques erróneos en el proceso de optimización. Las primeras pruebas en Android se realizaban de dos formas:

- Desde la instalación automática de la *app* mediante USB que el propio AndroidStudio aporta en un dispositivo móvil real conectado.
- Con el emulador *Virtual Device* de AndroidStudio, el cual permite ejecutar aplicaciones en un entorno Android, simulando además el modelo de móvil en el que se realiza.

Ambas resultaron problemáticas. La instalación de la aplicación mediante USB, aunque permitía ver que la aplicación se ejecutaba en un dispositivo móvil, resultaba en ralentizaciones extremas en ciertos dispositivos que no deberían darse en un juego de estas características técnicas (sin embargo, de esto se sacó una mayor motivación para optimizar aún más la aplicación, aunque el problema no fuera este). El simulador, por su parte, no presentaba estos problemas de rendimiento, sino que presentaba problemas en la carga de *assets* sonoros. Esta disparidad dio lugar a sospechas al cabo de unos días, por lo que se procedió a generar un APK de prueba para ver si, instalándolo simulando una instalación de la aplicación final real (y no por USB), se solucionaban los problemas. Resultó ser la decisión acertada, ya que mediante esta instalación no se presentaban ninguno de los problemas anteriores, y la ejecución era perfecta e igual a la de PC.

Con esto ya en mente, el resto de pruebas se realizaron en su mayoría en el simulador, ya que se buscaba hacer uso de una herramienta de este: el monitor de recursos. Con él, se buscaba controlar de forma más precisa el uso de RAM, CPU y GPU, y sobre todo ver en qué momento ocurrían (si ocurrían) las filtraciones de memoria.

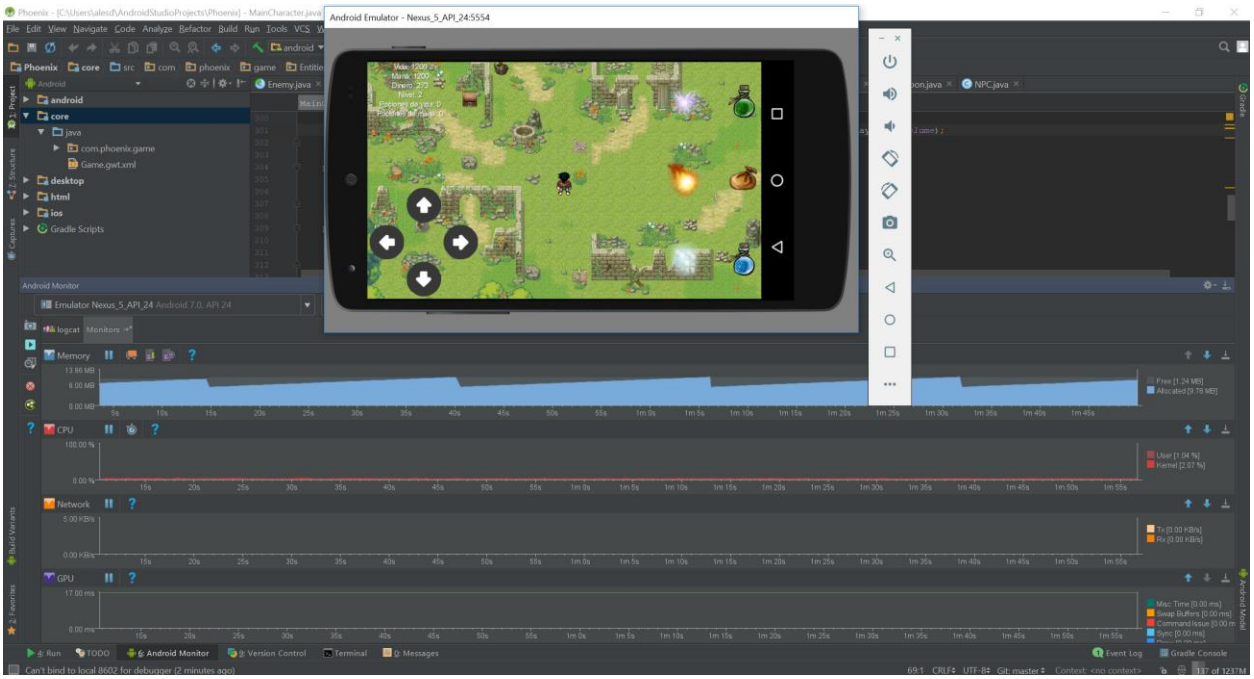
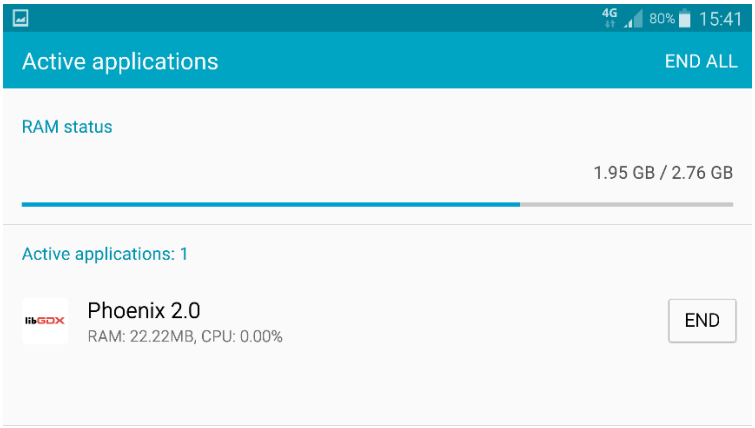


Ilustración 65 Puebas - Monitorización de recursos en Android

Gracias a esta herramienta, se pudo observar que no solamente era el uso de RAM el esperado (y bajo), sino que además se liberaban los *assets* que no se emplean de forma correcta (como se observa en los picos y bajadas del primer gráfico). Para confirmar el funcionamiento del juego en sesiones largas, se dejó el juego corriendo en la zona de más enemigos durante varias horas. Se encontró que la memoria empleada por la aplicación se mantenía estable, por lo que se dio el test como satisfactorio.

Claro que toda esta simulación se realizó sobre un teórico Nexus 5. El siguiente paso era realizar las pruebas sobre móviles físicos reales, en los cuales la monitorización se realizaría de 3 formas:

- **Memoria RAM**: Administrador de aplicaciones. Desde este se puede ver la memoria que está ocupando una aplicación, así como los cambios que se van dando en tiempo real.



Uso de memoria de la aplicación en ejecución. Durante la partida, fluctúa entre los 20 y los 30 MB, lo cual se consideró un éxito.

Ilustración 66 Pruebas - Uso de memoria en Android

- **CPU**: *Show CPU Usage* de las *Developer Options*. Esta opción muestra en la esquina superior derecha de la pantalla el uso que se está haciendo de los diferentes núcleos, así como sus cambios.

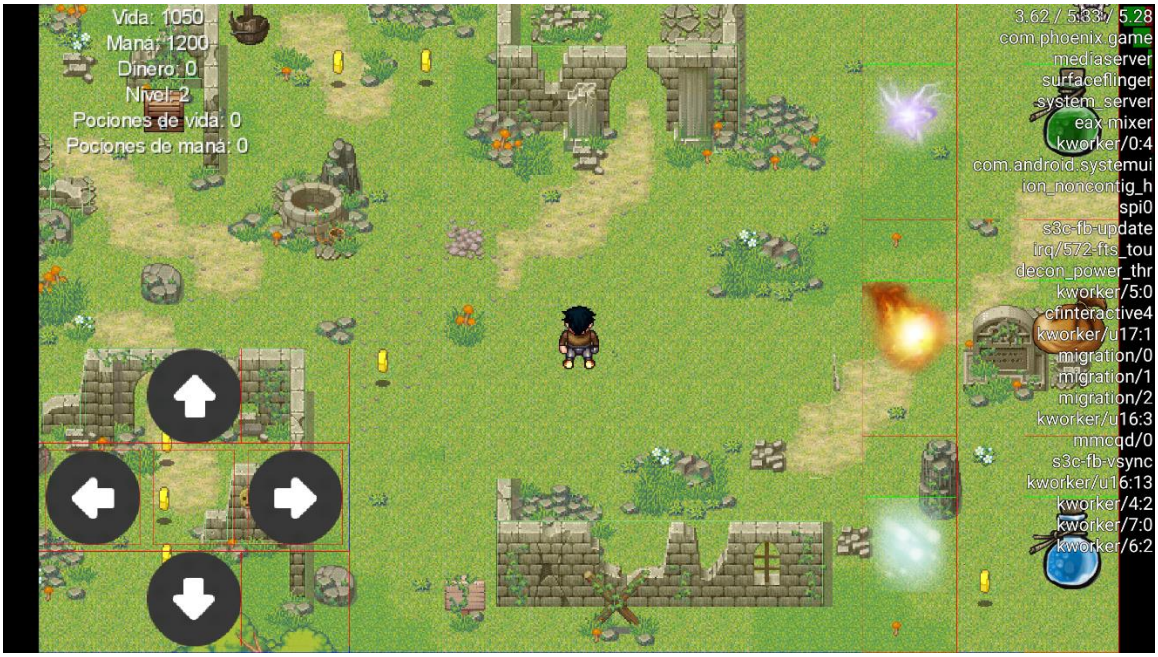


Ilustración 67 Pruebas - Uso de CPU en Android

- **GPU**: *GPU Rendering Profile* de las *Developer Options*. Para cada aplicación visible se generan unas barras (en un gráfico de barras) en la pantalla del dispositivo, las cuales representan fotogramas y la velocidad a la que se dibujan. Se usó para comprobar que se mantenían los 60 FPS estables. El funcionamiento completo de esta herramienta se puede encontrar en el siguiente enlace: https://developer.android.com/studio/profile/inspect-gpu-rendering .

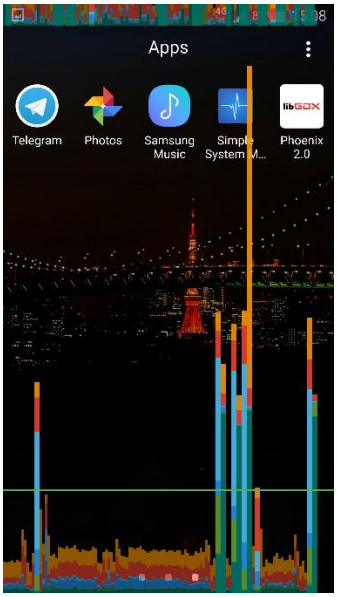


Gráfico de barras mencionado. La línea verde horizontal representa la frontera de los 60 FPS.

Ilustración 68 Pruebas - Uso de GPU en Android

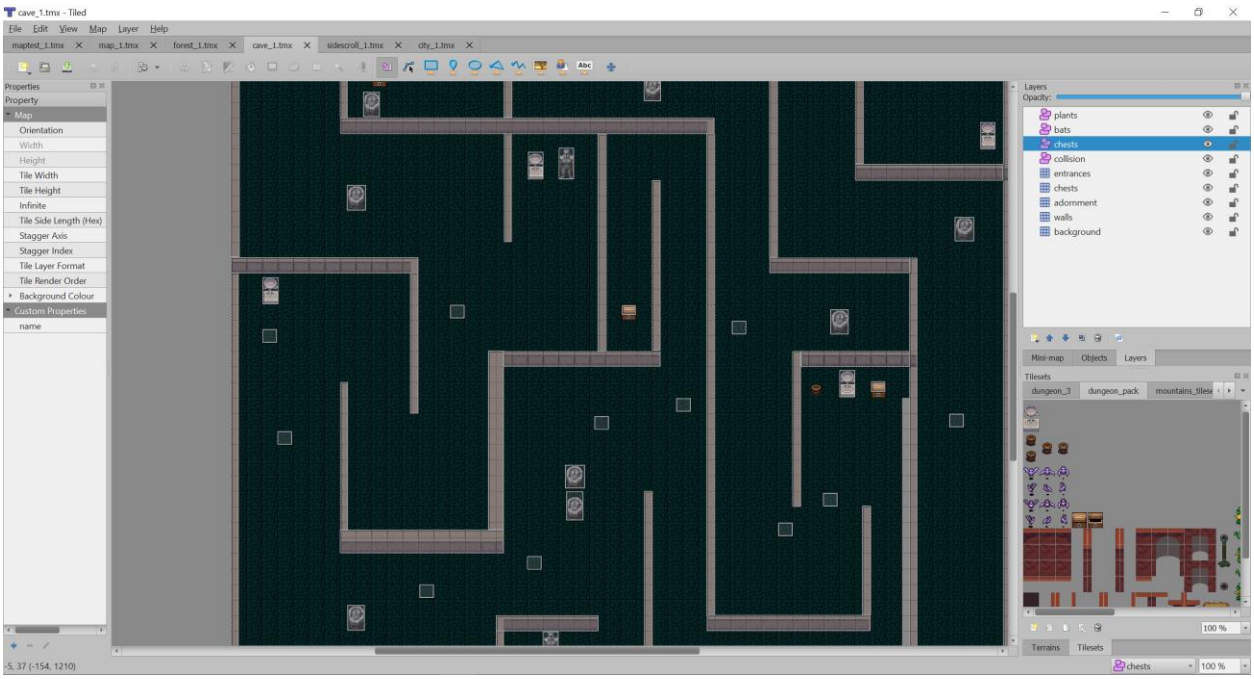
Por último, se tomó una última medida en lo referente al rendimiento que no ha sido mencionada. Debido a los falsos problemas de rendimiento que se encontraron con las pruebas por instalación USB, se encontró otro elemento que más adelante se tendría en cuenta: la *hardware acceleration*. Activando esta característica, que en algunos dispositivos móviles viene desactivada por defecto, se buscaba ver si se podían solucionar los problemas de rendimiento iniciales, ya que aprovecha más recursos y aumenta el rendimiento de la GPU a cambio de un mayor uso de memoria RAM (10). Aunque el final resultó no ser necesario en ningún dispositivo (y por lo tanto se dejó con su valor *default*), se puede forzar desde el archivo AndroidManifest.xml de nuestra aplicación.

VALOR AÑADIDO – EDITOR RPG

Cuando se concibió la idea de lo que sería *Phoenix*, se pensó en una experiencia tradicional en dispositivos modernos, y sobre todo, en una experiencia para un único jugador. Sin embargo, a medida que se fueron empleando las herramientas designadas para su creación, y aprendiendo sobre las posibilidades que estas daban, no tardó en surgir una nueva idea: *Phoenix*, tal y como ha sido programado, no debía limitarse a ser un RPG estático, sino dinámico. Gracias a las posibilidades que ofrece el editor de mapas *Tiled* y al enfoque con el que se ha realizado el desarrollo, se ha creado, además de una experiencia individual, un editor de mapas (y de mundos de fantasía) que cualquier persona, sin necesitar conocimientos técnicos ni de programación, puede llevar a cabo. En otras palabras, se ha creado un editor de RPGs, similar (en menor medida) al archiconocido *RPG Maker*.

Muchos usuarios disfrutan en mucha mayor medida del proceso de creación artístico de mundos, y no del hecho de jugar en sí. Con Phoenix, estos jugadores podrán crear mundos nuevos utilizando sus propios *spritesheets* en *Tiled*, asignar propiedades a los elementos del mapa (siempre dentro de lo permitido por la aplicación, no se puede inventar cosas que no están programadas), utilizar nuevas pistas de audio, etc. Lo único que se requiere para esto es conocimiento sobre el funcionamiento básico de *Tiled*, con elementos ya vistos como las capas de dibujo, los polígonos de colisión o la asignación de propiedades a estos polígonos. Estos mundos serán luego perfectamente jugables por otros jugadores, con las mismas propiedades físicas y funcionales que el juego original.

De esta forma, se pretende aprovechar las posibilidades que nos ofrece Internet y hacer *Phoenix* más grande y un juego en constante expansión siempre que la comunidad que lo juegue esté activa. Ya que los mapas en cuestión son únicamente archivos .tmx, estos podrían ser compartidos entre los usuarios vía foros dedicados, plataformas online de videojuegos e incluso plataformas artísticas. Por lo tanto, *Phoenix* consigue de nuevo mezclar lo tradicional con lo moderno, y apelar no solo a los jugadores tradicionales, sino también a los artistas y los apasionados de la creación.



Manteniendo el nombre y orden de las capas (layers, arriba a la derecha), un usuario podrá usar sus propios Tilesets (abajo a la derecha) para crear mapas nuevos y totalmente funcionales. A sus nuevos cuerpos podrá asignarles las propiedades ya vistas, customizando así los comportamientos de enemigos, colisiones, etc.

Ilustración 69 Valor añadido - Creación de mundos

CONCLUSIONES

Ahora que el desarrollo del proyecto ha llegado a su fin, es hora de echar la vista atrás y analizar los pasos dados, los objetivos alcanzados, los cambios de dirección y el futuro de *Phoenix*.

En primer lugar, si nos remontamos a la sección de objetivos, dos de los más relevantes eran claros: la realización de un proyecto a gran escala de principio a fin y el aprendizaje de una herramienta totalmente desconocida para realizar un juego, tarea en la que no se contaba con ninguna experiencia previa. A pesar de la dificultad que estos conllevaban (siempre se fue consciente de que los proyectos grandes tienen tendencia a quedarse cortos o a eliminar funcionalidades por falta de tiempo), se puede decir que su cumplimiento ha sido un éxito. No solamente se han implementado todas las funcionalidades planeadas excepto una, sino que se ha hecho excediendo la calidad inicial planteada. Además, se ha adquirido una enorme cantidad de conocimiento en lo referente, no solo a las herramientas empleadas, sino a la programación de videojuegos y su filosofía y metodología.

En relación a esto, se ha superado otro reto planteado inicialmente: los recursos como programador con tecnología desconocida. Prácticamente el 100% de las funcionalidades implementadas, más allá del lenguaje, eran en un principio totalmente desconocidas en lo que a su programación se refiere. Se sabía que la búsqueda eficiente y eficaz en internet de esto, así como la capacidad de aprenderlo por cuenta propia, sería clave para el desarrollo del proyecto. Y así fue, y de nuevo, se consiguió realizar con éxito, habiendo empleado numerosas fuentes de conocimiento oficiales, más o menos técnicas, que han contribuido notablemente a mi desarrollo como programador. Siguiendo con el desarrollo personal, *Phoenix* ha llevado a numerosos quebraderos de cabeza con las funcionalidades más complejas, llevando algunas horas de únicamente pensar teóricamente en su funcionamiento, sin estar siquiera seguro de si al final se podrían llevar a cabo. Estas "peleas" han resultado al final victoriosas, y se considera un logro lo suficientemente grande como para ser mencionado.

Phoenix, por otra parte, tenía un objetivo muy claro como proyecto y como medio de entretenimiento; apelar a fans "de la vieja escuela" de los videojuegos, mezclando lo antiguo con lo moderno y llevar a cabo un juego que respete de nuevo al jugador. Se consideraba que este sería el objetivo más complicado de llevar a cabo, ya que va más allá de lo objetivamente técnico, teniendo que "imprimir" al juego con una sensación subjetiva. Además, se tenía la limitación en cuanto a *assets* se refiere, ya que no se contaba con un equipo artístico de desarrollo. A pesar de esto, el resultado es satisfactorio. Se ha conseguido imprimir una sensación de la vieja escuela, que de alguna forma tiene toques de modernidad y atractivo visual. Obviamente, con un equipo dedicado a esto se podría haber realizado un trabajo aún más atractivo, y se podría haber añadido un escenario narrativo, elemento que, aunque diseñado, no ha entrado en la versión final de *Phoenix*. Aunque no se ha podido comprobar su aceptación con grupos grandes usuarios, esta ha sido muy alta en los círculos cercanos y entre los *beta testers*, por lo que se considera un nuevo objetivo conseguido.

En cuanto a la parte más técnica del proyecto, se considera, en general, que se ha hecho un buen trabajo. El diseño inicial era sólido y no ha variado en exceso a medida que se iba avanzando, lo cual se considera un éxito en un proyecto de esta magnitud. Sin embargo, existe una ligera sensación agridulce en cuanto a la modularidad alcanzada. Es cierto que el diseño es modular, especialmente para el juego en su estado actual, pero a medida que la idea iba madurando y adquiriendo una visión más clara del proyecto y del desarrollo de videojuegos en general, se vio que se podía ir un paso más allá en el diseño modular y eficiente, especialmente de cara a expansiones futuras. Desgraciadamente, ya que había una fecha límite, estos cambios habrían requerido demasiado tiempo y no se han podido realizar. Sin embargo, y como se mencionará en el siguiente apartado, quedarían como trabajo futuro

y necesario para una aplicación de la mayor calidad técnica posible. En lo referente al apartado técnico que mayor desafío suponía, la optimización y el correcto rendimiento del juego en distintos dispositivos, el resultado final es excelente. El rendimiento es, en general, mejor que el planteado inicialmente (como se detalla en el apartado de *Pruebas*), y *Phoenix* corre a 60 FPS en todos los dispositivos en los que se ha probado, asegurando en todos una experiencia óptima. Este se considera el mayor de los logros, ya que más allá de la dificultad técnica de cualquier proyecto similar a lo visto en la carrera, este requería ir un paso más allá en cuanto a la optimización y diseño eficiente.

En general, no ha habido prácticamente elementos planificados que no se hayan podido llevar a cabo, y la mayoría de los riesgos planteados no se dieron. Sin embargo, hay algo que sí difiere en gran medida de la planificación inicial: la planificación temporal. A continuación se muestran las diferencias entre lo inicialmente planteado y la realidad:

TAREA	DURACIÓN (Horas)	DURACIÓN REAL (Horas)
DOCUMENTACIÓN		
Elaboración del DOP	45	50
Memoria del TFG	60	150
CAPTURA DE REQUISITOS		
Diagrama de Casos de Uso y Jerarquía de Actores	8	3
Modelo de Dominio	8	3
Casos de Uso Extendidos	20	4
Transformación de MD a BBDD	10	4
MÓDULOS		
Diseño e Implementación de Mundos	20	100
Diseño e Implementación de Enemigos	30	60
IA de Enemigos e Interacciones del Jugador	40	40
Diseño e Implementación de Ciudades y NPCs	30	20
Funcionalidades Necesarias	30	10
Diseño e Implementación de Menús	30	30
Port a Android	30	20
DURACIÓN TOTAL ESTIMADA	**361**	**494**

Tabla 43 Conclusiones - Diferencia de planificación temporal

En esta tabla se pueden apreciar distintos factores. En primer lugar se ha dedicado bastante más tiempo a la documentación del inicialmente planificado. Esto no se debe tanto al cambio de contenidos en sí, sino a la dedicación empleada en que la documentación sea fácil de leer y visualmente atractiva. Herramientas como *Visual Paradigm*, aunque potentes y necesarias, conllevan una inversión de tiempo mayor para la representación de diagramas. Algo similar ocurrió con las tablas, índices, gráficos o ilustraciones, cuyo formateo llevó también más tiempo del esperado. Aun así, y a pesar de la diferencia de horas, no se considera que haya sido un cálculo con excesivo impacto de retraso temporal, ya que se preveía que esto podía ocurrir y se planteó la documentación como proceso que se extendería hasta la entrega del TFG.

La *Captura de Requisitos*, en general, se hizo más rápido de lo inicialmente pensado. Aunque Phoenix es un proyecto técnicamente complejo, su este fase resultó ser más sencilla de lo esperado. Elementos como la base de datos formada por archivos .xml, un solo actor, o casos de uso claros simplificaron en gran medida esta sección.

Es en el siguiente apartado, en el desarrollo, en el que se dedicó mucho más tiempo del planeado y donde los cálculos realmente se alejan mucho de la realidad. En un principio, se planteó el diseño (tanto artístico como técnico) de mundos, mapas y enemigos como una tarea sencilla. Nada más lejos de la realidad. El mero apartado visual y la creación de los mapas ya llevaron más tiempo de lo que se había planteado inicialmente para todo el proceso, encontrando muchas dificultades para la búsqueda y creación de *assets* que, no solamente tenían que ser técnicamente compatibles entre ellos, sino además juntarse para formar mapas atractivos y con un diseño lógico. No solo esto fue problemático. Una vez diseñados visualmente, su creación técnica fue igualmente compleja. En esta fase se tuvieron que establecer las bases de lo que sería el juego, el "motor" que lo moviera. Esto incluía elementos como magnitudes físicas, colisiones, y sobre todo cómo todo esto se compatibilizaba con todos los módulos posteriores. En resumen, lo que en general se planteó como una fase más bien transitoria, resultó al final ser la más compleja y duradera de todo el proyecto, además de la gran culpable de la diferencia de horas final.

Es muy destacable, además, una tendencia que se observa en las horas empleadas en el desarrollo. Se ve como, en los primeros módulos, la diferencia de horas es muy grande. Sin embargo, a medida que se avanzando, la diferencia va siendo menor o se iguala, haciendo los últimos módulos incluso en menos tiempo que el planeado. Esto se debe, sobre todo, a la experiencia adquirida y como al final el desarrollo resultaba mucho más sencillo por esta. Además, desde un punto de vista técnico, los últimos módulos eran más sencillos, basándose estos en menús, funcionalidades no-visuales o el port a Android.

De lo que no hay ninguna duda es de que, si se tuviese la oportunidad de empezar el proyecto de nuevo, este sería más modular y su planificación se realizaría de forma mucho más precisa. Además, con los conocimientos adquiridos la implementación se realizaría en un tiempo mucho menor, dando tiempo a un desarrollo más relajado y que con toda seguridad generaría un proyecto final de aún más calidad. En general, y resumiendo, se considera que Phoenix ha sido un éxito tanto en lo personal como en lo técnico, yendo más allá de los conocimientos adquiridos en la carrera, creando un producto que cumple las altas expectativas de las que se partía, y haciendo de mí un mucho mejor programador.

TRABAJO FUTURO

Dada la pasión con la que se ha realizado el proyecto y el satisfactorio resultado final, se pretende, con bastante seguridad, seguir trabajando en *Phoenix* en el futuro cercano. Ya se ha hablado en diferentes apartados del proyecto de las partes que quedan abiertas a trabajo futuro, siendo estas:

- Refactorización de ciertos elementos del código de cara a expansiones
    - Modularidad de las clases
    - IA de los enemigos en casos críticos (y muy improbables)
    - Funcionalidad → Guardar partida
    - Funcionalidad → Cargar Partida
- Revisión de dos niveles de dificultad
- Implementación de la Cash Shop como elemento de monetización de *Phoenix*
- Solución de bugs
    - Texturas negras en Android cuando el juego no se cierra desde el administrador de tareas
    - Gravedad ligeramente por debajo de lo normal cuando se salta en el momento en el que se sale de una escalera
- Adición de tutoriales en los primeros compases del juego
- Adición de una historia, en forma de diálogos ya implementados, durante todo el juego

Aunque no se hayan mencionado durante la documentación, si se siguiese trabajando en el juego los pasos lógicos a realizar serían:

- Creación de nuevos mapas con elementos característicos y distintos a los ya vistos
    - Mazmorras con nuevas y más difíciles mecánicas
    - Zonas con diferentes climas y atractivo visual: desiertos, nieve…
    - Nuevas ciudades con nuevos NPCs
- Creación de nuevos enemigos con nuevas mecánicas
- Creación de nuevos objetos, cuyos estados deben corresponderse al progreso que el jugador va experimentando
    - Nuevos tipos de armas (lanzas, guantes…)
    - Nuevos tipos de armaduras
    - Objetos curativos más potentes
    - …
- Adición de nuevas habilidades a ser desbloqueadas
- Progreso de la historia inicial planteada

Lo bueno de *Phoenix* es que la adición de todos estos elementos llevaría un tiempo muchísimo menor al desarrollo que se ha hecho. El motor lógico que mueve todos los elementos del juego ya ha sido desarrollado, y la creación de nuevos elementos visuales o ciertas mecánicas requeriría un tiempo bastante pequeño. Donde la creación de los cuatro mapas actuales ha llevado meses, la adición de, por ejemplo, 10 nuevos mapas, con enemigos, mecánicas nuevas y objetos llevaría seguramente un tiempo inferior a un mes. Desde este punto de vista, sería una pena no aprovechar la parte ya desarrollada y poder crear una experiencia aún más completa y satisfactoria para el jugador. Con todo esto y el hecho de que, si se crea una comunidad de jugadores apasionados, estos pueden crear una infinidad de mundos, *Phoenix* es un proyecto en constante expansión y futuro infinito.

RECONOCIMIENTOS

En esta sección se procederá a nombrar a los autores de los diferentes recursos que se han empleado para el desarrollo del juego. Principalmente se nombrará a artistas cuyos *assets* se han empleado, aunque también se nombrará a creadores de contenido cuyas lecciones han tenido mucho valor a la hora de aprender todo lo relacionado con la programación de videojuegos:

- **Daniel Cook, yd, Jetrel, Zabin** → Spritesheets de elementos visuales de mapas.
  - https://opengameart.org/content/2d-lost-garden-zelda-style-tiles-winter-theme-with-additions
- **Hyptosis** → Diseño de mapa de ciudad
  - https://opengameart.org/content/mage-city-arcanos
- **Flowly** → Iconos
  - https://opengameart.org/content/bag-icon
- **Warlock's Gauntlet team** → Iconos de habilidades
  - https://opengameart.org/content/spell-icon-collection-part-4
- **www.flaticon.com** → Icono de salto
  - https://www.flaticon.com/free-icon/happy_157776#term=jump&page=1&position=1
- **Satur9** → Fondo de menús
  - https://opengameart.org/content/space-background-01
- **ViRiX** → Efectos sonoros
  - https://opengameart.org/content/ui-sound-effects-pack
- **DoKashiteru** → Efectos sonoros
  - https://opengameart.org/content/ui-sound-effects-pack
- **Kalez** → Spritesheets para la animación de proyectiles
  - https://kalez.deviantart.com/art/Elementals-Set-204503634
- **lpc.opengameart.org** → Personaje principal, enemigos, elementos visuales en mapas
  - http://lpc.opengameart.org/static/lpc-style-guide/assets.html
  - http://gaurav.munjal.us/Universal-LPC-Spritesheet-Character-Generator/
- **Rmn** → Temas musicales
  - https://rpgmaker.net/musicpack/
- **Brent Aurelli** → Series sobre el uso de LibGDX y Box2D
  - https://www.youtube.com/watch?v=a8MPxzkwBwo&list=PLZm85UZQLd2SXQzsF-a0-pPF6IWDDdrXt
  - https://www.youtube.com/watch?v=rzBVTPaUUDg&list=PLZm85UZQLd2TPXpUJfDEdWTSgszionbJy

Todos los *assets* empleados forman parte del *Dominio Público* o están sujetos a licencias que permiten su distribución sin que estos sean modificados. Algunos de ellos están sujetos a la licencia CC BY-NC-ND 3.0 (11), la cual permite su distribución fuera de usos comerciales. Si Phoenix fuera a salir a la venta en un futuro, estos *assets* en particular deberían ser cambiados.

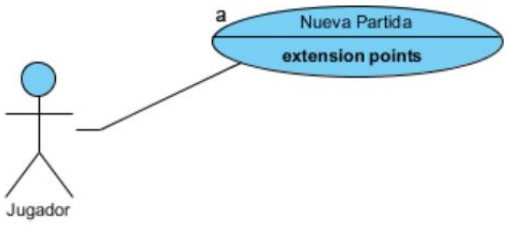
BIBLIOGRAFÍA

(1) **Stephanie Chan (2017)** – *King's players numbers are down, but they're spending more in Candy Crush and other games* - https://venturebeat.com/2017/08/03/kings-player-numbers-are-down-but-theyre-spending-more-in-candy-crush-and-other-games/
(2) https://img.wonderhowto.com/img/32/34/63475339844556/0/walkthrough-side-scrolling-video-game-sonic-hedgehog-4.1280x600.jpg
(3) https://i.stack.imgur.com/ryYMo.jpg
(4) https://image.slidesharecdn.com/desarrolloiterativoeincremental-120829050505-phpapp02/95/desarrollo-iterativo-e-incremental-6-728.jpg?cb=1346216786
(5) http://www.sii-group.es/pfw_files/cma/Website/Site_ESP/nuestra_actividad/sprint.png
(6) **MonetizePros** – *Mobile Ad CPM Rates* - https://monetizepros.com/cpm-rate-guide/mobile/
(7) **James Plafke (2015)** – *Square Enix shifts focus to mobile games, but it's for the best* - https://www.geek.com/games/square-enix-shifts-focus-to-mobile-games-but-its-for-the-best-1622767/
(8) **Dan Gallagher (2012)** – *Nexon rides free-to-play model to strong gains* - https://www.marketwatch.com/story/nexon-pushes-free-to-play-to-strong-gains-2012-08-09
(9) **NEXON co., Ltd. (2017)** – *Investor Presentation Q4 2017* - https://ir.nexon.co.jp/en/library/pdf/20180208_2.pdf
(10) **Android Developers** – *Hardware Acceleration* - https://developer.android.com/guide/topics/graphics/hardware-accel
(11) **Creative Commons** – *Attribution – NonCommercial – NoDerivs 3.0 Unported* - https://creativecommons.org/licenses/by-nc-nd/3.0/

ANEXO I

CASOS DE USO
CASOS DE USO EXTENDIDOS

CASO DE USO NUEVA PARTIDA



Nombre: Nueva Partida

Descripción: El usuario comienza a jugar una nueva partida desde el menú principal.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario ejecuta el juego y accede al menú principal. (*Ilustración 1*)
2. Pulsa sobre "Nueva Partida".
3. Se muestra en la ventana la interfaz gráfica del juego, con el jugador ya controlable. (*Ilustración 2*)

[Si pulsa sobre la "I" en PC o el botón táctil designado en el dispositivo móvil]

4a. Se mostrará su inventario (*Ilustración 3*)

[Si pulsa sobre Objetos]

4aa. EXTEND VER OBJETOS

[Si pulsa sobre Equipo]

4ab. EXTEND VER EQUIPO

[Si pulsa sobre Estado]

4ac. EXTEND VER ESTADO

[Si pulsa sobre Guardar Partida]

4ad. EXTEND GUARDAR PARTIDA

[Si colisiona con un NPC]

4b. EXTEND INTERACTUAR CON NPC

[Si pulsa sobre ("Flecha arriba", "Flecha abajo", "Flecha derecha" o "Flecha izquierda") o las flechas táctiles en dispositivo móvil]

4c. EXTEND MOVERSE

[Si pulsa sobre "Espacio" o el botón designado para saltar en dispositivo móvil]

4d. EXTEND SALTAR
[Si pulsa sobre "1", "2" o "3" o los botones táctiles designados en la pantalla táctil]
4e. EXTEND ATACAR
[Si colisiona con un objeto]
4f. EXTEND INTERACTUAR CON OBJETO
[Si pulsa sobre "4" o "5" o los botones táctiles designados en la pantalla táctil]
4g. EXTEND USAR OBJETO

Poscondiciones: Ninguna

Interfaz gráfica:

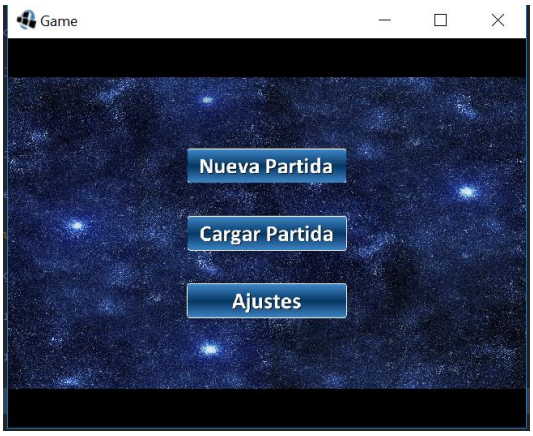


Ilustración 1

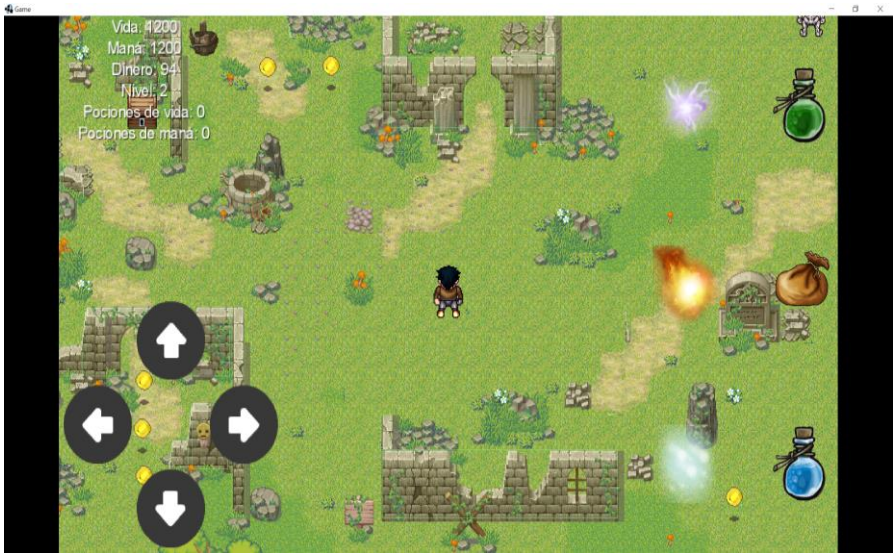


Ilustración 2

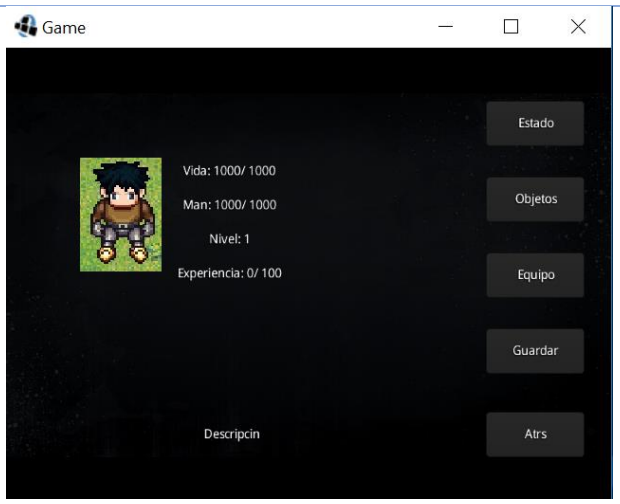
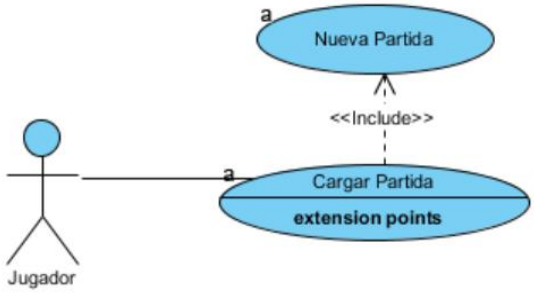


Ilustración 3

CASO DE USO CARGAR PARTIDA



Nombre: Cargar Partida

Descripción: El usuario carga una partida previamente guardada, para resumirla en el punto en el que la había dejado.

Actores: Usuario

Precondiciones: Hay una partida guardada.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario ejecuta el juego y accede al menú principal (Ilustración 1)
   [Si pulsa sobre Cargar Partida]
   2a. INCLUDE NUEVA PARTIDA

Poscondiciones: Ninguna

Interfaz gráfica:

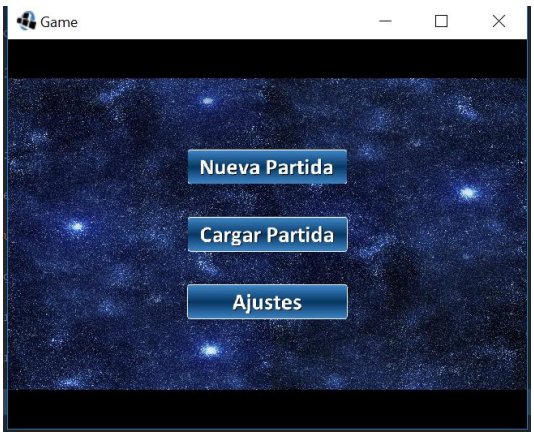
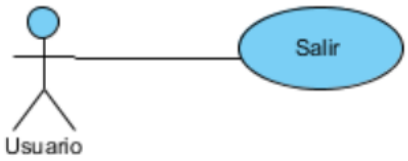


Ilustración 1

CASO DE USO SALIR



Nombre: Salir

Descripción: El usuario sale de la partida, concluyéndola.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Se pulsa sobre la "X" encontrada en la parte superior derecha de la pantalla en cualquier momento del juego.
2. Se cierra la ventana del juego

Poscondiciones: Ninguna

Interfaz gráfica:

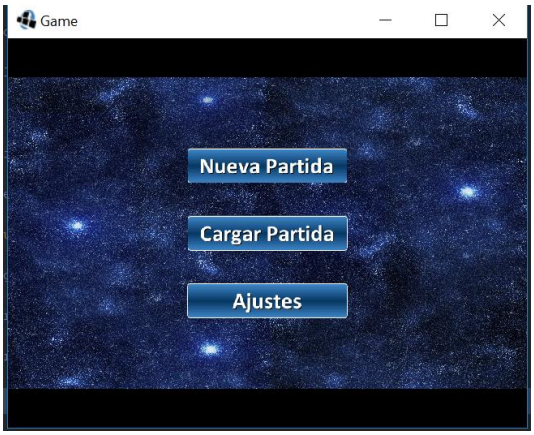
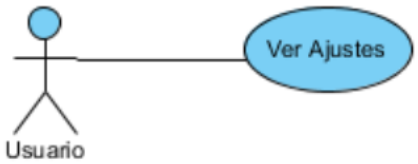


Ilustración 1

CASO DE USO VER AJUSTES



Nombre: Ver Ajustes

Descripción: El usuario puede visualizar las diferentes opciones que se le muestran para personalizar la partida.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Se le muestra al usuario el menú principal (*Ilustración 1*)
   [Si pulsa sobre "Ver Ajustes"]
   2a. Se le muestra un nuevo menú con los posibles ajustes que puede modificar *(Ilustración 2)*
   [Si pulsa sobre el slider de "Intensidad del Audio"]
   2aa. EXTENDS SELECCIONAR AUDIO
   [Si pulsa sobre el slider de "Nivel de Dificultad"]
   2ab. EXTENDS SELECCIONAR DIFICULTAD
   [Si pulsa sobre "Manual de Usuario"]
   2ac. EXTENDS VER AYUDA / MANUAL

Poscondiciones: Ninguna

Interfaz gráfica:

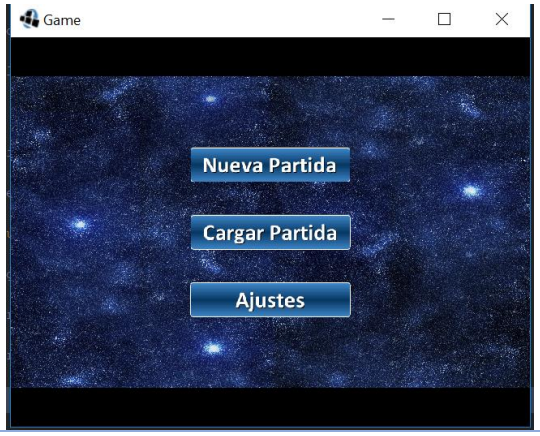


Ilustración 1

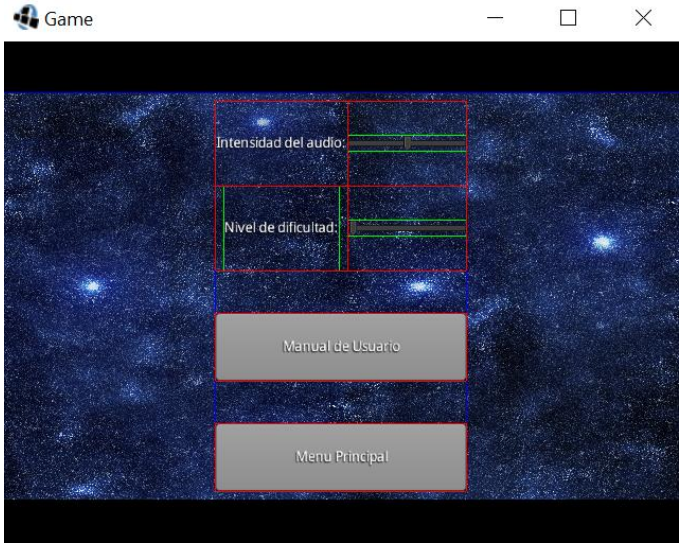
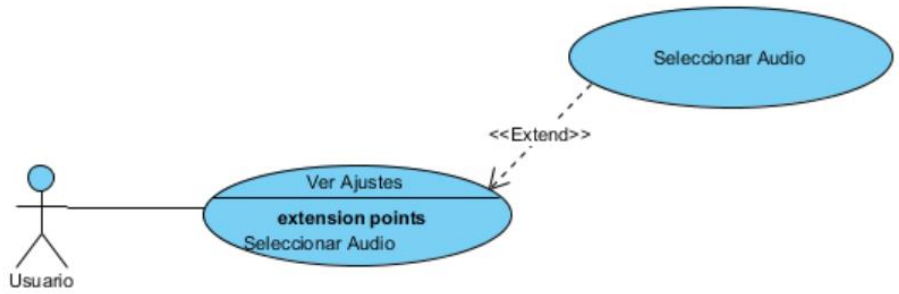


Ilustración 2

CASO DE USO EXTENDIDO SELECCIONAR AUDIO



Nombre: Seleccionar Audio

Descripción: El usuario selecciona la intensidad del nivel de audio con la que quiere jugar.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Se le muestra al usuario un menú con un *slider* para que seleccione el nivel de audio con el que quiere jugar. (Ilustración 1)
   [Si mueve el slider]
   2a. Se recoge y aplica al juego el nuevo valor de la intensidad del audio

Poscondiciones: Se almacena el nuevo valor del audio

Interfaz gráfica:

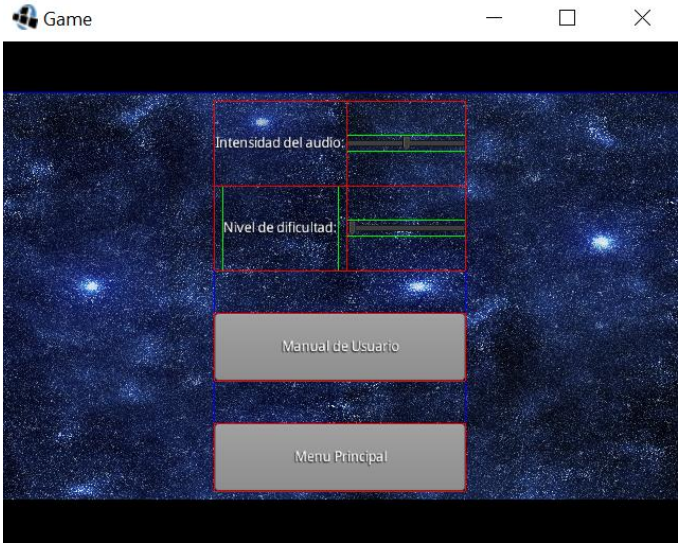
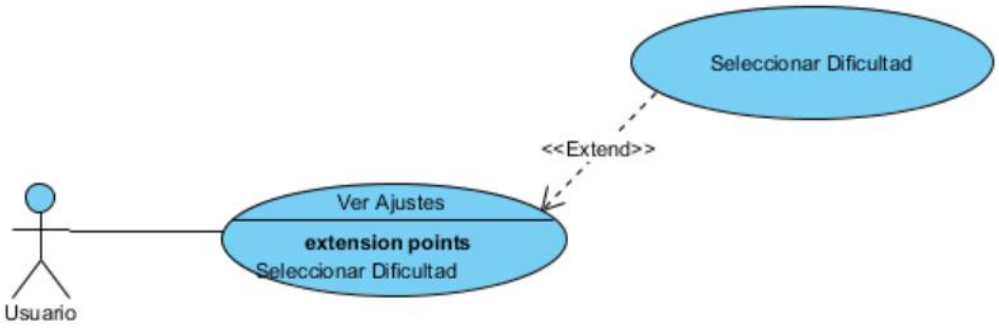


Ilustración 1

CASO DE USO EXTENDIDO SELECCIONAR DIFICULTAD



Nombre: Seleccionar Dificultad

Descripción: El usuario selecciona la dificultad con la que quiere jugar la partida.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Se le muestra al usuario un slider con 3 dificultades distintas.
   [Si mueve el slider]
   2a. Se recoge la nueva dificultad y se aplica al juego.

Poscondiciones: Se almacena el nuevo valor de la dificultad

Interfaz gráfica:

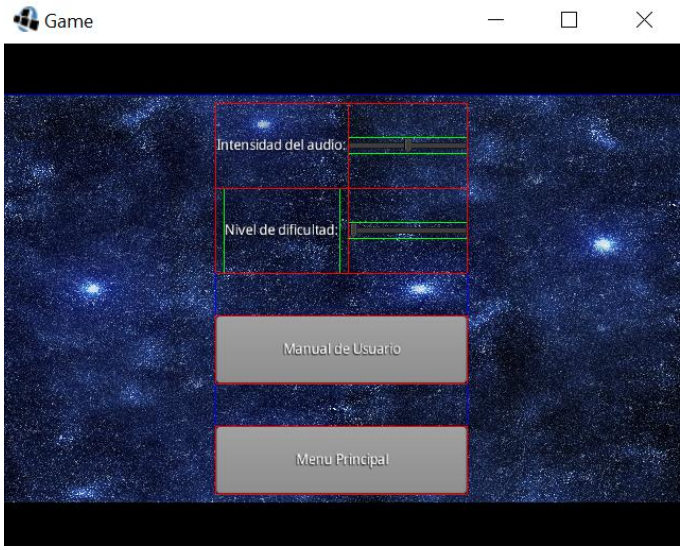
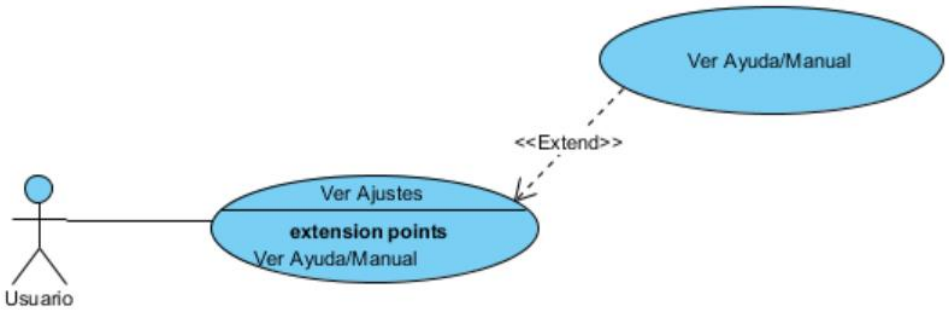


Ilustración 1

CASO DE USO EXTENDIDO VER AYUDA/MANUAL



Nombre: Ver Ayuda / Manual

Descripción: El usuario puede ver un texto a modo de ayuda o manual, en el que se explicarán las bases del juego, funcionamiento y controles básicos.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. Se le muestra al usuario un menú con las explicaciones de las mecánicas y elementos básicos del juego (Ilustración 1)

Poscondiciones: Ninguna

Interfaz gráfica:

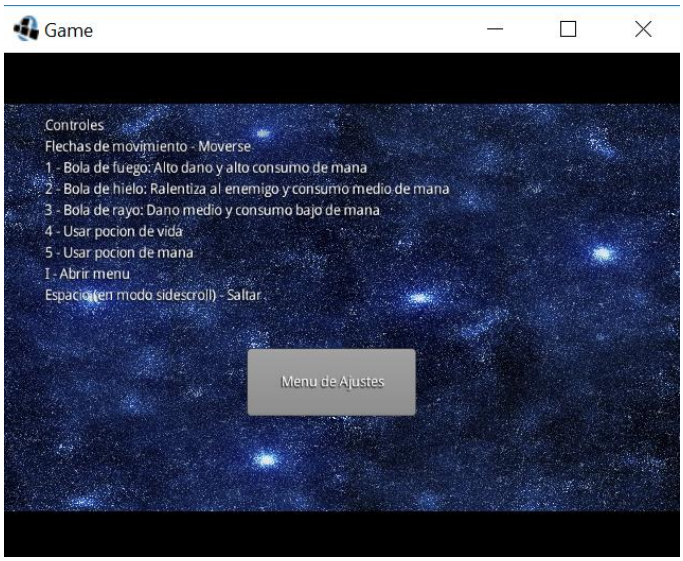
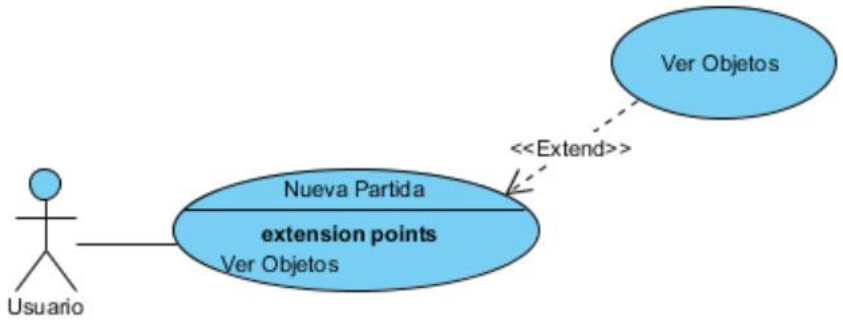


Ilustración 1

CASO DE USO EXTENDIDO VER OBJETOS



Nombre: Ver Objetos

Descripción: El usuario puede ver, en su menú, los objetos que posee actualmente en su inventario.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario está jugando la partida
   [Si pulsa sobre la "I"]
   2a. Se abre su menú principal in-game, con todas las opciones. (*Ilustración 1*)
   [Si pulsa sobre "Objetos"]
   3aa. Se le muestra un nuevo menú con la lista de objetos que posee. (*Ilustración 2*)

Poscondiciones: Ninguna

Interfaz gráfica:

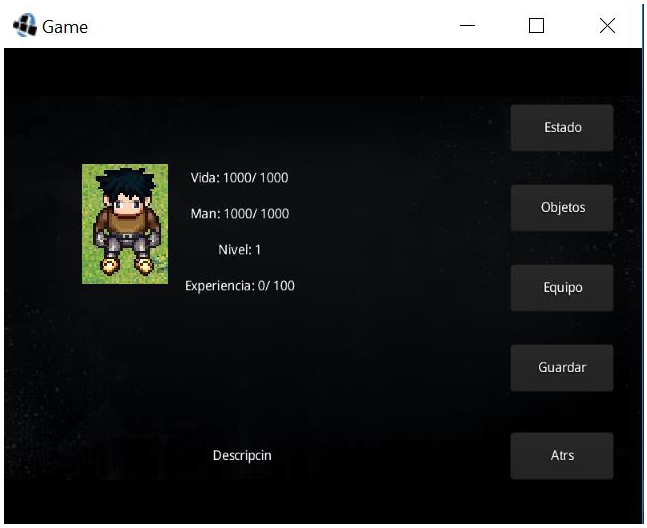


Ilustración 1

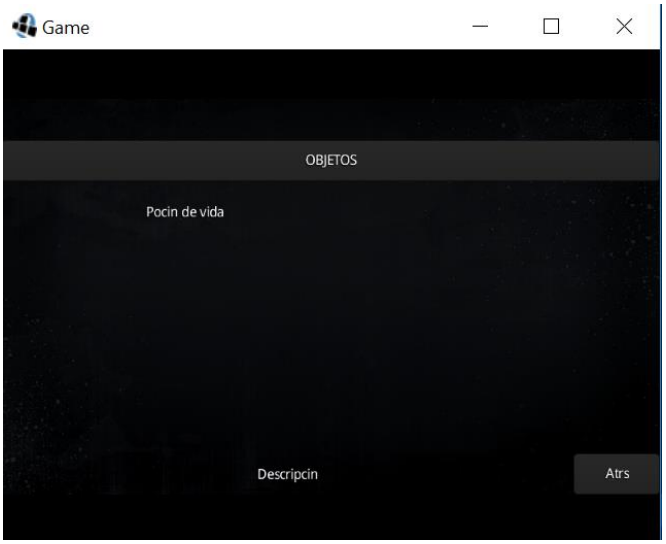
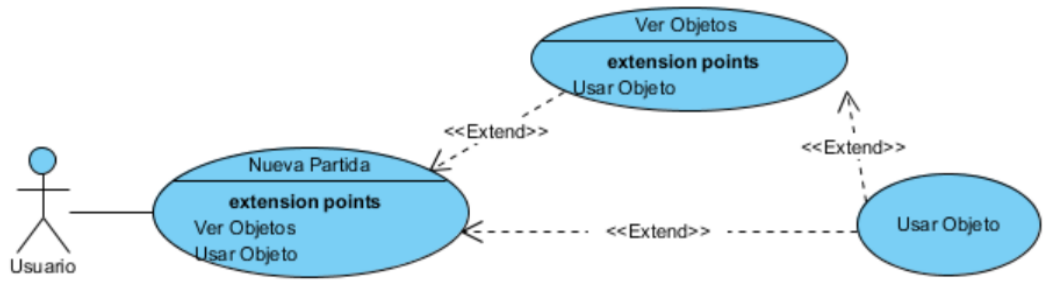


Ilustración 2

CASO DE USO EXTENDIDO USAR OBJETO



Nombre: Usar Objeto

Descripción: El usuario usa un objeto de su inventario, consumiéndolo y generando el efecto de dicho objeto.

Actores: Usuario

Precondiciones: Hay al menos un objeto en el inventario.

Requisitos no funcionales: Ninguno

Flujo de eventos:

[Si el usuario ha pulsado la "I"]

1. El usuario se encuentra en su menú de Objetos. (Ilustración 1)
   [Si pulsa sobre un objeto]
   2a. Se genera el efecto pertinente
   3a. Se actualiza el menú de Objetos, restando una unidad a la cantidad de objetos del mismo tipo o eliminando la fila si solo había uno.
   4a. Se actualiza la IU de la pantalla principal con el número actual de pociones.

[Si el usuario ha pulsado "4" o "5"] (*Ilustración 2*)

1b. Se genera el efecto pertinente
2b. Se actualiza el menú de Objetos de la misma manera que en el caso 3a.
3b. Se actualiza la IU de la misma manera que en el caso 4a.

Poscondiciones: Ninguna

Interfaz gráfica:

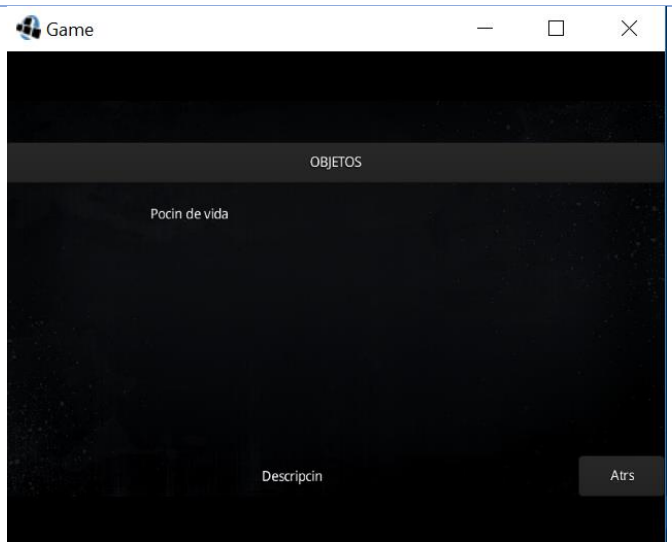


Ilustración 1

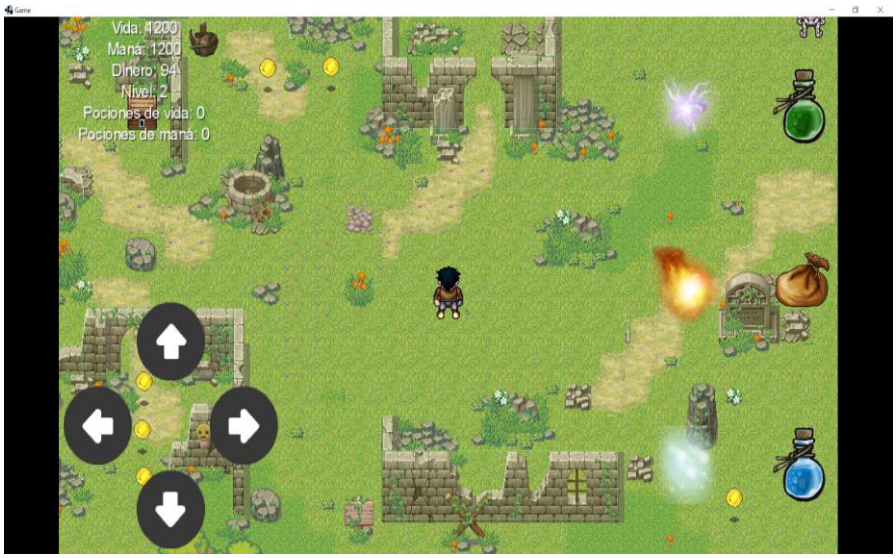
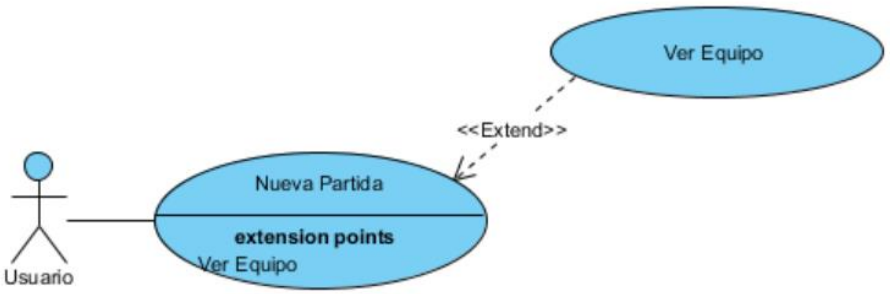


Ilustración 2

CASO DE USO EXTENDIDO VER EQUIPO



Nombre: Ver equipo

Descripción: El usuario puede ver, en su menú, el equipo que posee actualmente en su inventario.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario está jugando la partida
   [Si pulsa sobre la "I"]
   2a. Se abre su menú principal in-game, con todas las opciones.
   [Si pulsa sobre "Equipo"]
   3aa. Se le muestra un nuevo menú con el equipo que posee y el que lleva equipado. (Ilustración 1)

Poscondiciones: Ninguna

Interfaz gráfica:

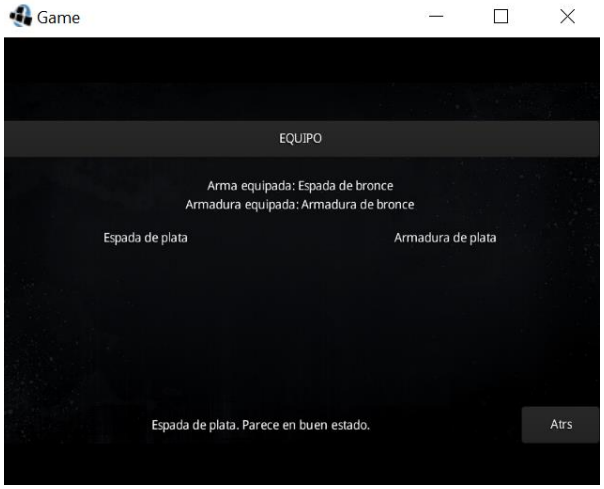
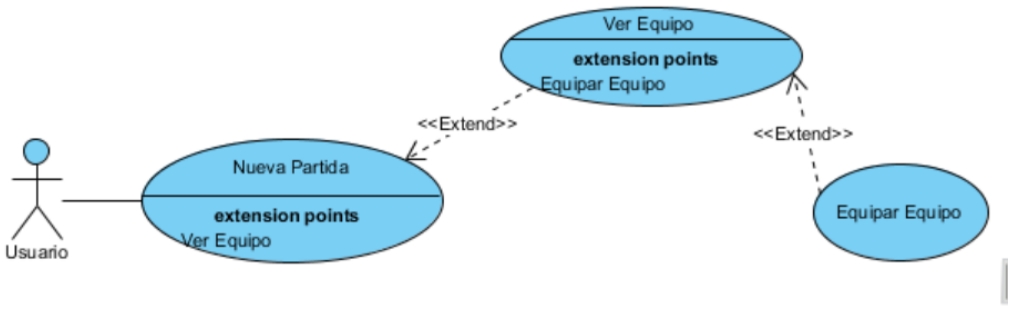


Ilustración 1

CASO DE USO EXTENDIDO EQUIPAR EQUIPO



Nombre: Equipar Equipo

Descripción: El usuario se equipará una pieza de equipo de su inventario.

Actores: Usuario

Precondiciones: Hay al menos una pieza de equipo en el inventario.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario se encuentra en su menú de Equipo.
   [Si pulsa sobre una pieza de equipo no equipada]
   2. El jugador se equipa la pieza, mostrándose el nombre de la pieza equipada en la sección de piezas actualmente equipadas del jugador en el menú de Estado y en el propio menú de Equipo.

Poscondiciones: Ninguna

Interfaz gráfica:

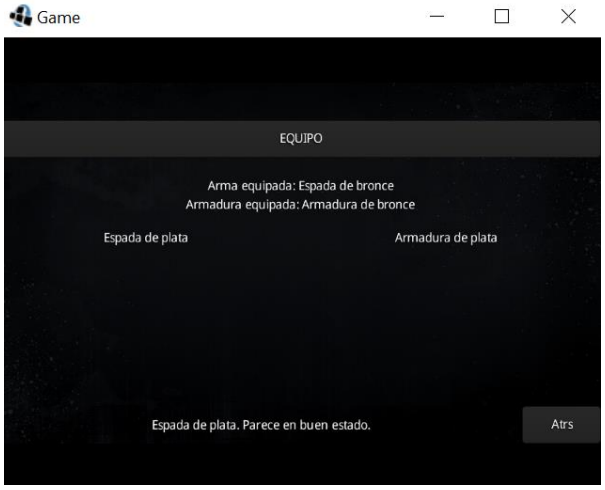
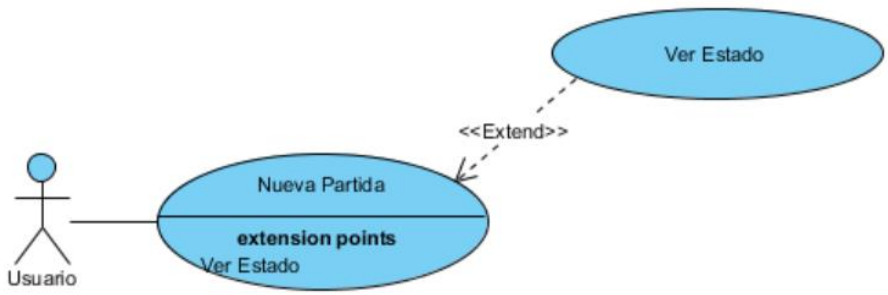


Ilustración 1

CASO DE USO EXTENDIDO VER ESTADO



Nombre: Ver Estado

Descripción: El usuario podrá ver el estado actual del jugador, el cual es un resumen de sus atributos, equipo y habilidades actuales.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario está jugando la partida
   [Si pulsa sobre la "I"]
   2a. Se abre su menú principal in-game, con todas las opciones.
   [Si pulsa sobre "Estado"]
   3aa. Se le muestra un nuevo menú con sus atributos actuales, afectados por el equipo que lleva equipado.

Poscondiciones: Ninguna

Interfaz gráfica:

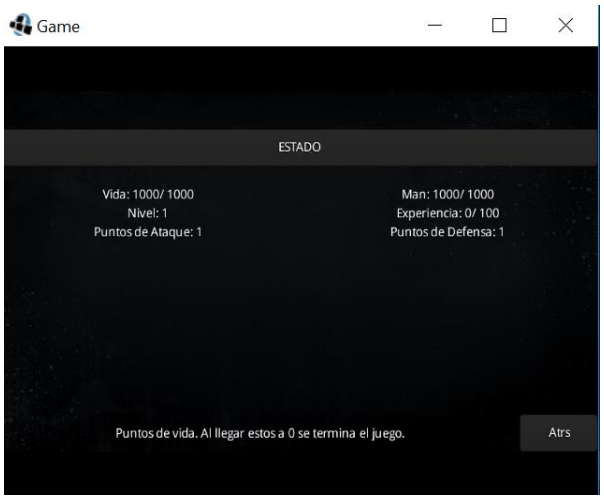
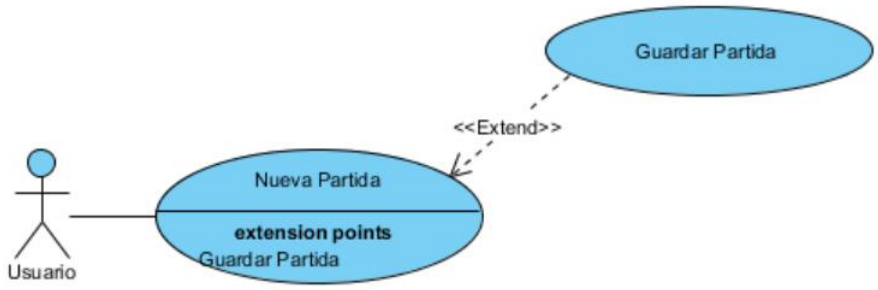


Ilustración 1

CASO DE USO EXTENDIDO GUARDAR PARTIDA



Nombre: Guardar Partida

Descripción: El usuario guardará la partida hasta el punto actual, de forma que pueda resumirla en ese mismo punto en otro momento.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario está jugando la partida
   [Si pulsa sobre la "I"]
   2a. Se abre su menú principal in-game, con todas las opciones. (*Ilustración 1*)
   [Si pulsa sobre "Guardar Partida"]
   3aa. Se mostrará el mensaje de confirmación: "La partida ha sido guardada correctamente"

Poscondiciones: Se almacena el mapa actual del jugador, su posición en él, sus atributos, equipo y habilidades. Se almacena también el punto de la historia en el que se encuentra el jugador.

Interfaz gráfica:

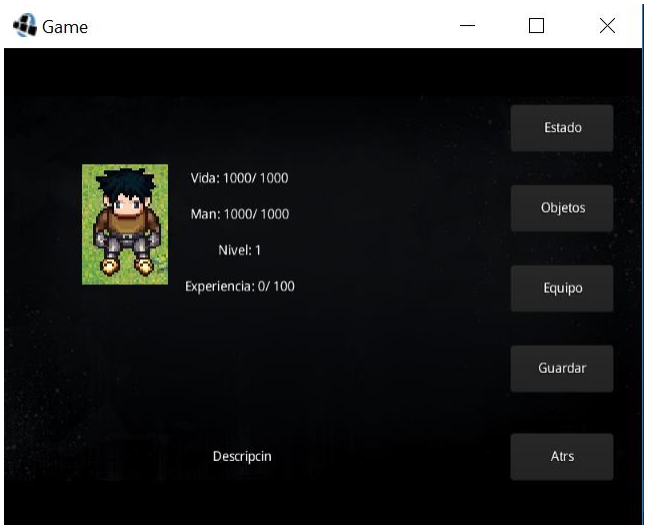
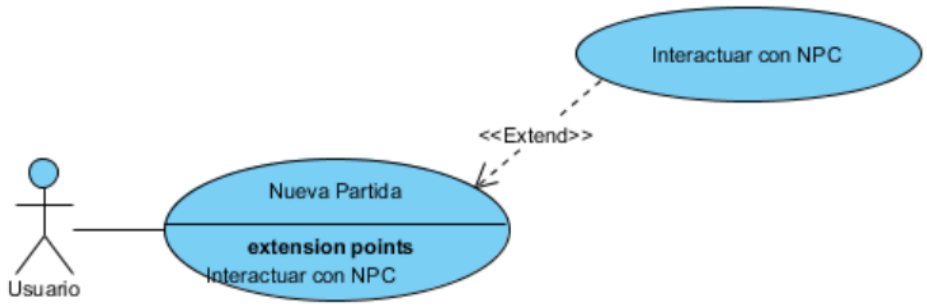


Ilustración 1

CASO DE USO EXTENDIDO INTERACTUAR CON NPC



Nombre: Interactuar con NPC

Descripción: El usuario interactúa con un NPC, ya sea para dialogar, avanzar en la historia o realizar compras/ventas.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario está jugando la partida y se acerca a un NPC
   [Si colisiona con el NPC]
   2a. Se le mostrará una nueva ventana con el diálogo correspondiente al NPC (Ilustración 1)
   [Si pulsa sobre Aceptar]
   3a. Se cierra la ventana y puede seguir jugando.

Poscondiciones: Ninguna

Interfaz gráfica:

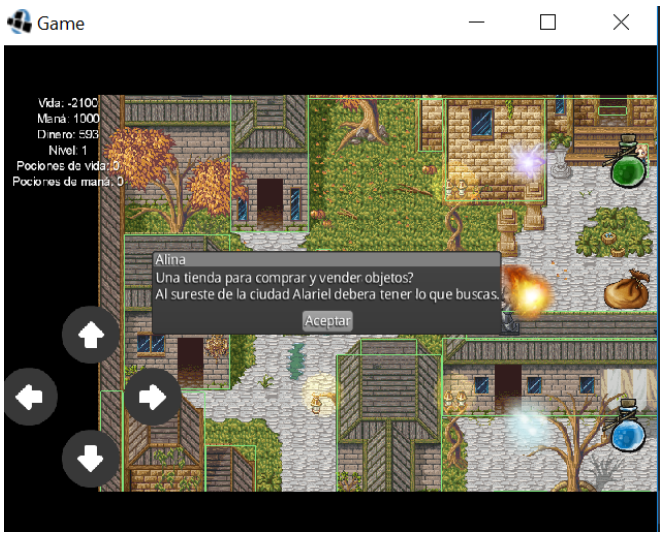
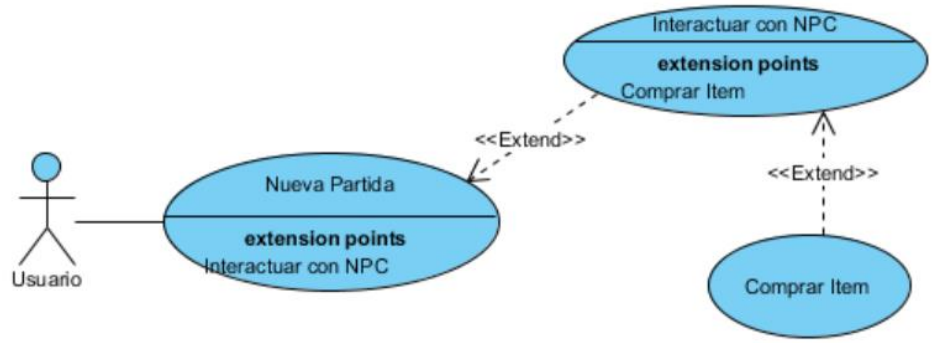


Ilustración 1

CASO DE USO EXTENDIDO COMPRAR ITEM



Nombre: Comprar Item

Descripción: El usuario compra un ítem, ya sea equipo u objeto, de un NPC que lo permita.

Actores: Usuario

Precondiciones: El NPC puede comprar y vender ítems.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario se encuentra en la ventana de diálogo del NPC, dónde se le muestran las opciones: " Comprar/ Vender" *(Ilustración 1)*

[Si pulsa sobre "Comprar"]

2a. Se le muestra un nuevo menú con los objetos que puede comprar, así como su precio. *(Ilustración 2)*

[Si pulsa sobre un objeto]

[Si tiene dinero suficiente]

3aaa. Comprará el objeto, de forma que se restará el precio de este a su dinero actual, y se añadirá el objeto a su inventario correspondiente.

[Si no tiene dinero suficiente]

3aab. Sonará un sonido de error

Poscondiciones: Ninguna

Interfaz gráfica:

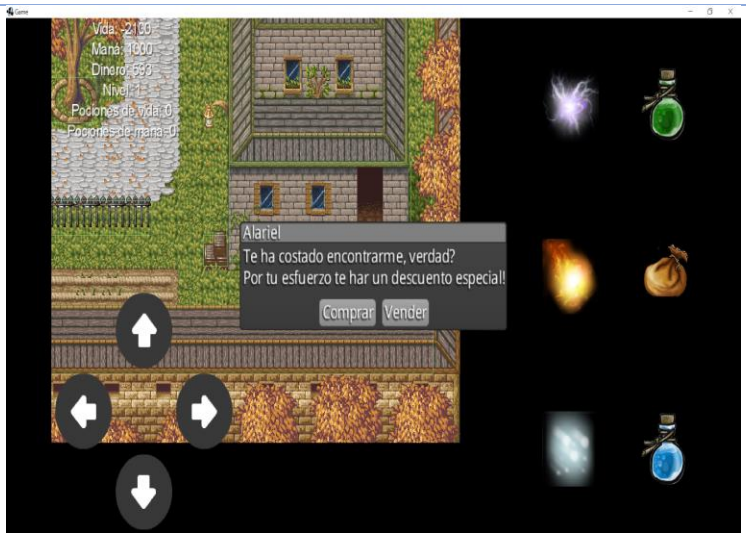


Ilustración 1

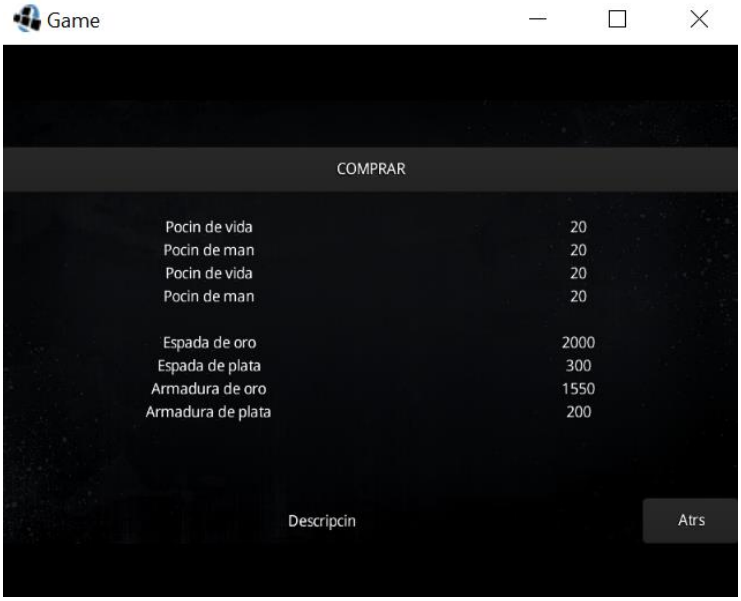
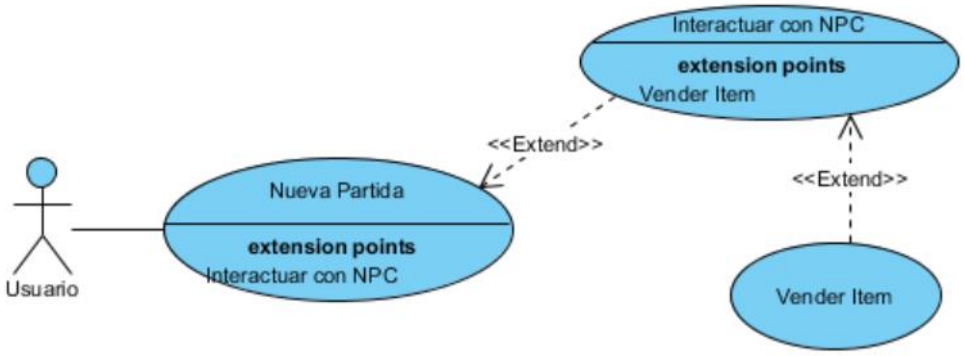


Ilustración 2

CASO DE USO EXTENDIDO VENDER ITEM



Nombre: Vender Item

Descripción: El usuario vende un ítem, ya sea equipo u objeto, a un NPC que lo permita.

Actores: Usuario

Precondiciones: El NPC puede comprar y vender ítems.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El usuario se encuentra en la ventana de diálogo del NPC, dónde se le muestran las opciones: " Comprar/ Vender"

   [Si pulsa sobre "Vender"]

   2a. Se le muestra un nuevo menú con los objetos que puede vender, así como su precio de venta. (Ilustración 1)

   [Si pulsa sobre un objeto]

   3aa. Se vende el objeto, sumando el dinero de la venta a su dinero Actual y restando una unidad al total de objetos del tipo que se ha vendido. Si solamente hubiese uno, se elimina la fila.

Poscondiciones: Ninguna

Interfaz gráfica:

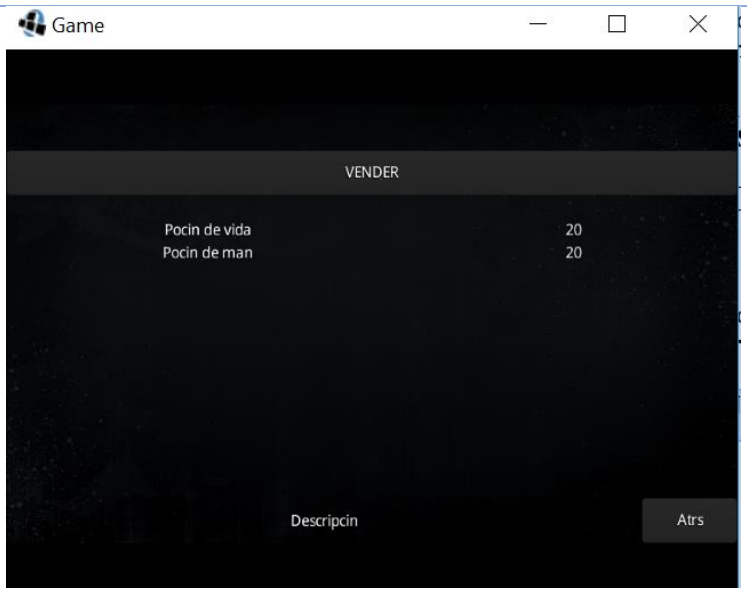
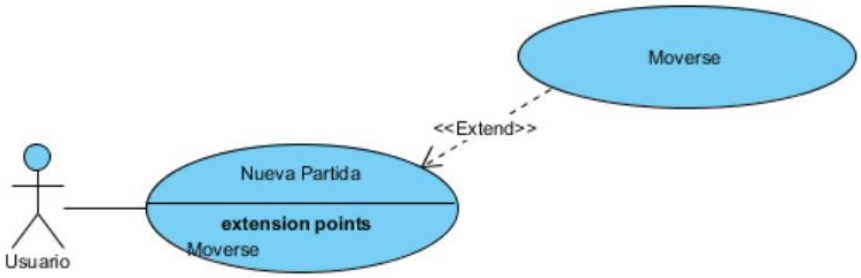


Ilustración 1

CASO DE USO EXTENDIDO MOVERSE



Nombre: Moverse

Descripción: El usuario puede moverse por el mapa en 4 direcciones: derecha, izquierda, arriba y abajo.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El jugador se encuentra en un mapa

[Si el mapa es de tipo top-down]

[Si el jugador pulsa "Flecha arriba", "Flecha Izquiera", "Flecha Atrás" o "Flecha derecha" o los botones táctiles de movimiento]

2aa. El jugador se moverá en la dirección correspondiente a la tecla que ha pulsado, siendo esta "arriba", "izquierda", "abajo" o "derecha" respectivamente. (Ilustración 1)

[Si mantiene la tecla o botón pulsados]

3aaa. Se seguirá moviendo en la misma dirección

[Si el mapa es de tipo side-scroll]

[Si el jugador pulsa "Flecha derecha" o "Flecha Izquierda" o los botones táctiles de movimiento] (Ilustración 2)

2ba. El jugador se moverá en la dirección correspondiente a la tecla que ha Pulsada, siendo esta "derecha" o "izquierda" respectivamente.

[Si mantiene la tecla o botón pulsados]

3baa. Se seguirá moviendo en la misma dirección

Poscondiciones: Ninguna

Interfaz gráfica:

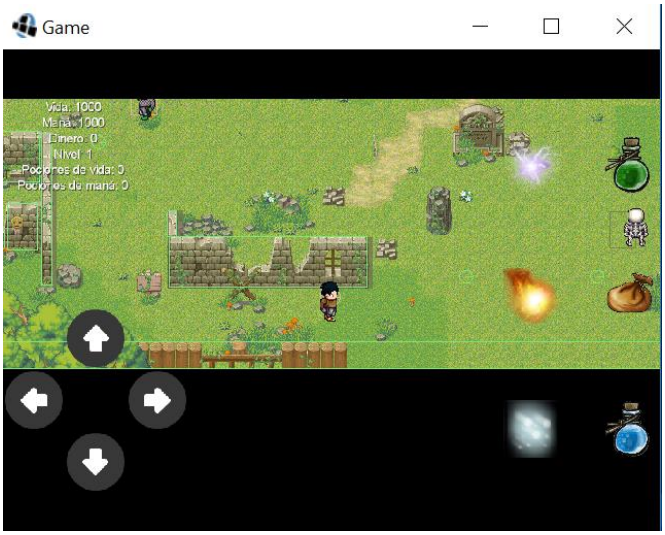


Ilustración 1

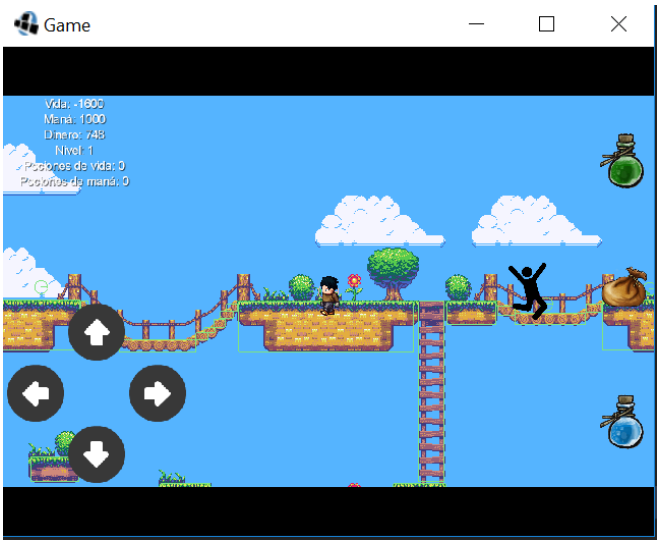
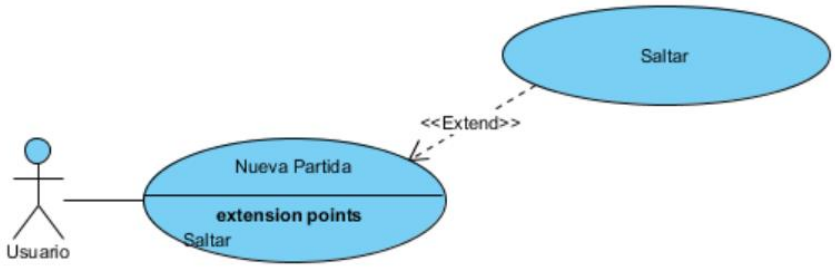


Ilustración 2

CASO DE USO EXTENDIDO SALTAR



Nombre: Saltar

Descripción: El jugador salta, elevándose y volviendo a caer.

Actores: Usuario

Precondiciones: El jugador está en un mapa de tipo side-scroll.

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El jugador se encuentra en un mapa
   [Si pulsa sobre "Espacio" en PC o el botón táctil correspondiente en dispositivo móvil]
   2a. El jugador saltará (Ilustración 1)

Poscondiciones: Ninguna

Interfaz gráfica:

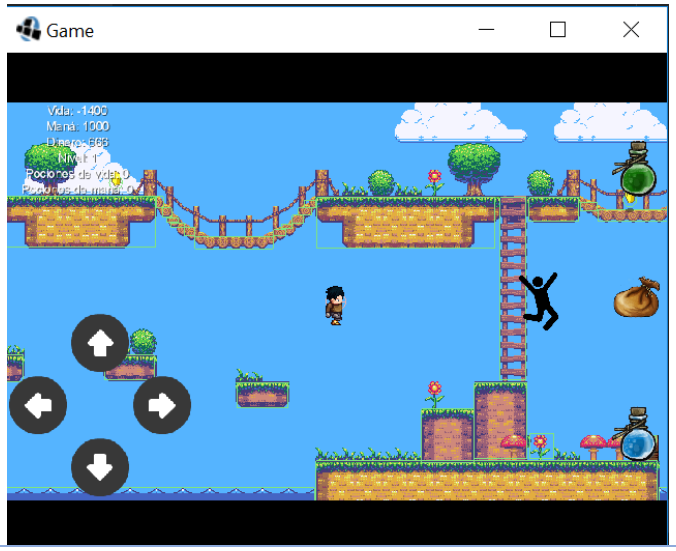
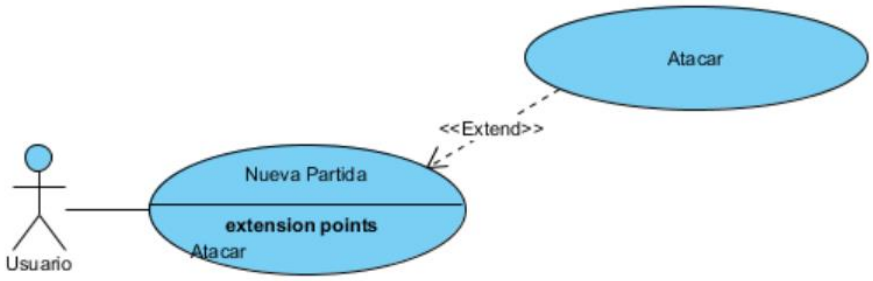


Ilustración 1

CASO DE USO EXTENDIDO ATACAR



Nombre: Atacar

Descripción: El jugador atacará, o bien a un enemigo o al aire, mediante el uso de sus auto-ataques o alguna de las habilidades que haya aprendido.

Actores: Usuario

Precondiciones: Las habilidades tienen que estar desbloqueadas para ser usadas

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El jugador se encuentra en un mapa

   [Si pulsa sobre "1" o el botón táctil correspondiente en dispositivo móvil]

   2b. El jugador lanzará una bola de fuego en la dirección en la que mira y perderá maná alto. *(Ilustración 1)*

   [Si pulsa sobre "3" o el botón táctil correspondiente en dispositivo móvil]

   2c. El jugador lanzará un rayo en la dirección en la que mira y perderá maná leve. *(Ilustración 2)*

   [Si pulsa sobre "2" o el botón táctil correspondiente en dispositivo móvil]

   2d. El jugador lanzará una bola de hielo en la dirección en la que mira y perderá maná leve. *(Ilustración 3)*

[Si el ataque alcanza a dar en un enemigo]

[Si es una bola de fuego]

3ab. El enemigo recibirá daño alto

[Si es un rayo]

3ac. El enemigo recibirá daño moderado

[Si es una bola de hielo]

3ad. El enemigo recibirá daño leve y su velocidad de movimiento se verá reducida.

[Si el enemigo pierde toda su vida]

3a. El enemigo morirá, y por lo tanto, desaparecerá del mapa.

Poscondiciones: Ninguna

Interfaz gráfica:

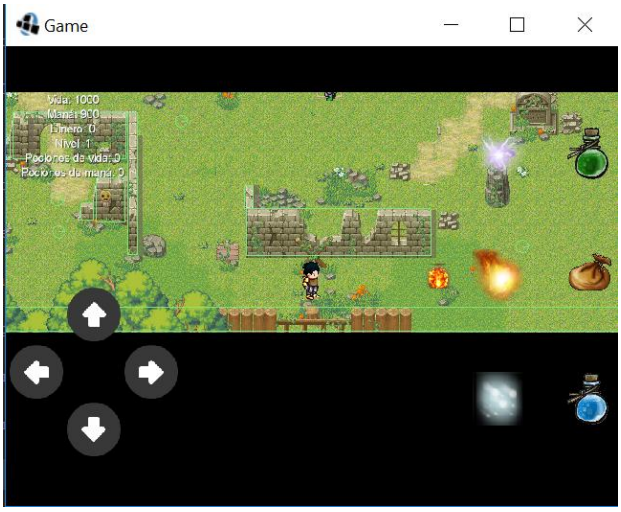


Ilustración 1

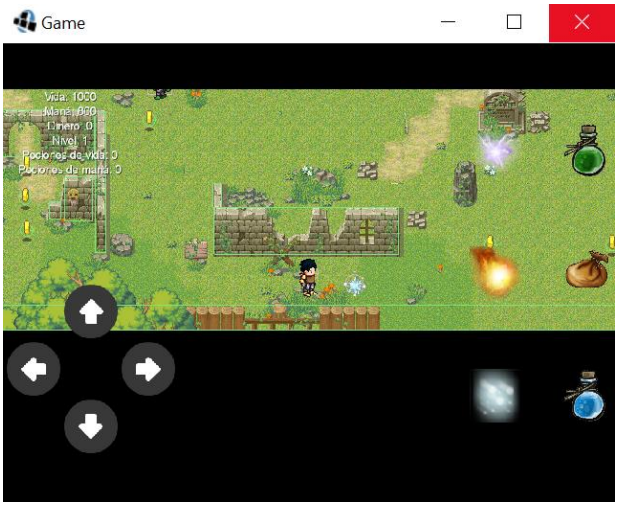


Ilustración 2

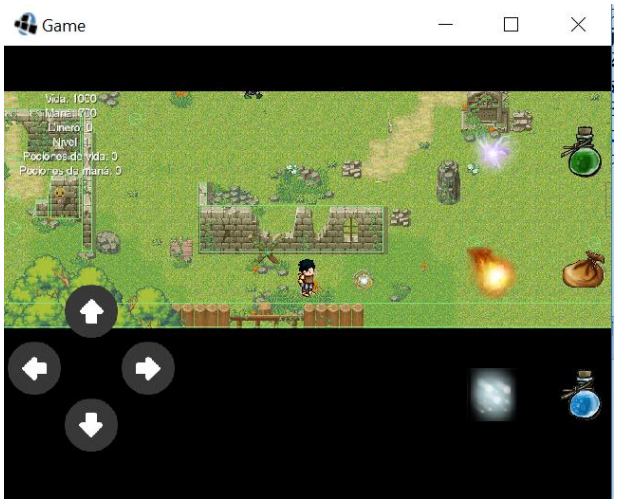
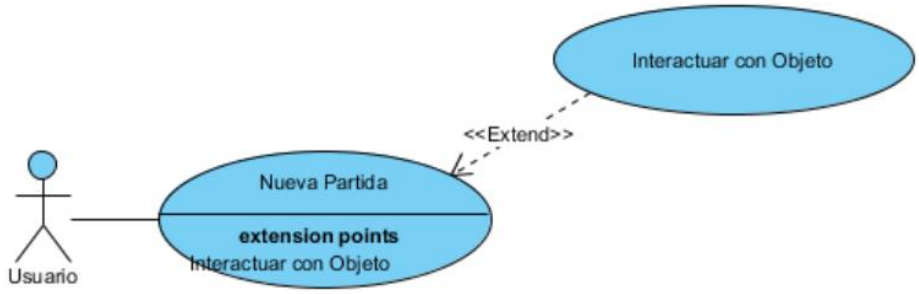


Ilustración 3

CASO DE USO EXTENDIDO INTERACTUAR CON OBJETO



Nombre: Interactuar con Objeto

Descripción: El usuario interactúa con un objeto que se encuentra en el mapa en el que se encuentre.

Actores: Usuario

Precondiciones: Ninguna

Requisitos no funcionales: Ninguno

Flujo de eventos:

1. El jugador está jugando la partida y se acerca a un objeto

[Si el objeto está en el suelo]

[Si el jugador entra en contacto directo con el objeto]

2aa. El jugador obtendrá el objeto, desapareciendo este del suelo y sumándose a su inventario.

3aa. Se mostrará una pequeña ventana con un mensaje de: "*Se ha obtenido Objeto*".

[Si el objeto está dentro de un cofre]

[Si el jugador está en contacto con el cofre]

2baa. Se abrirá el cofre y se obtendrá el objeto que hay en su interior.

3baa. Se mostrará una pequeña ventana con un mensaje de: "*Se ha obtenido objeto*". *(Ilustración 1)*

Poscondiciones: Ninguna

Interfaz gráfica:

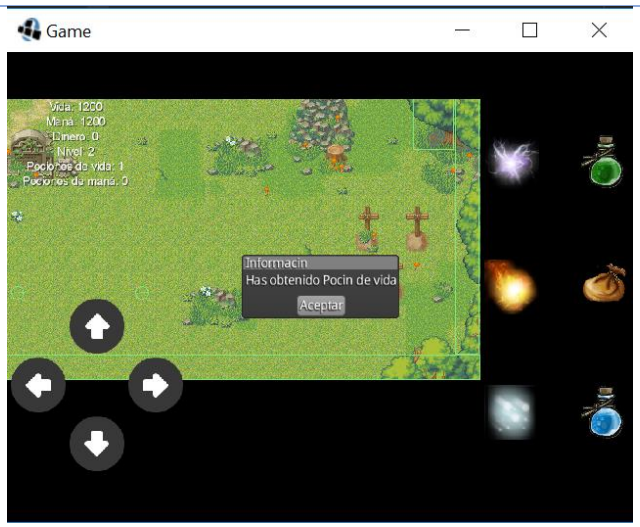


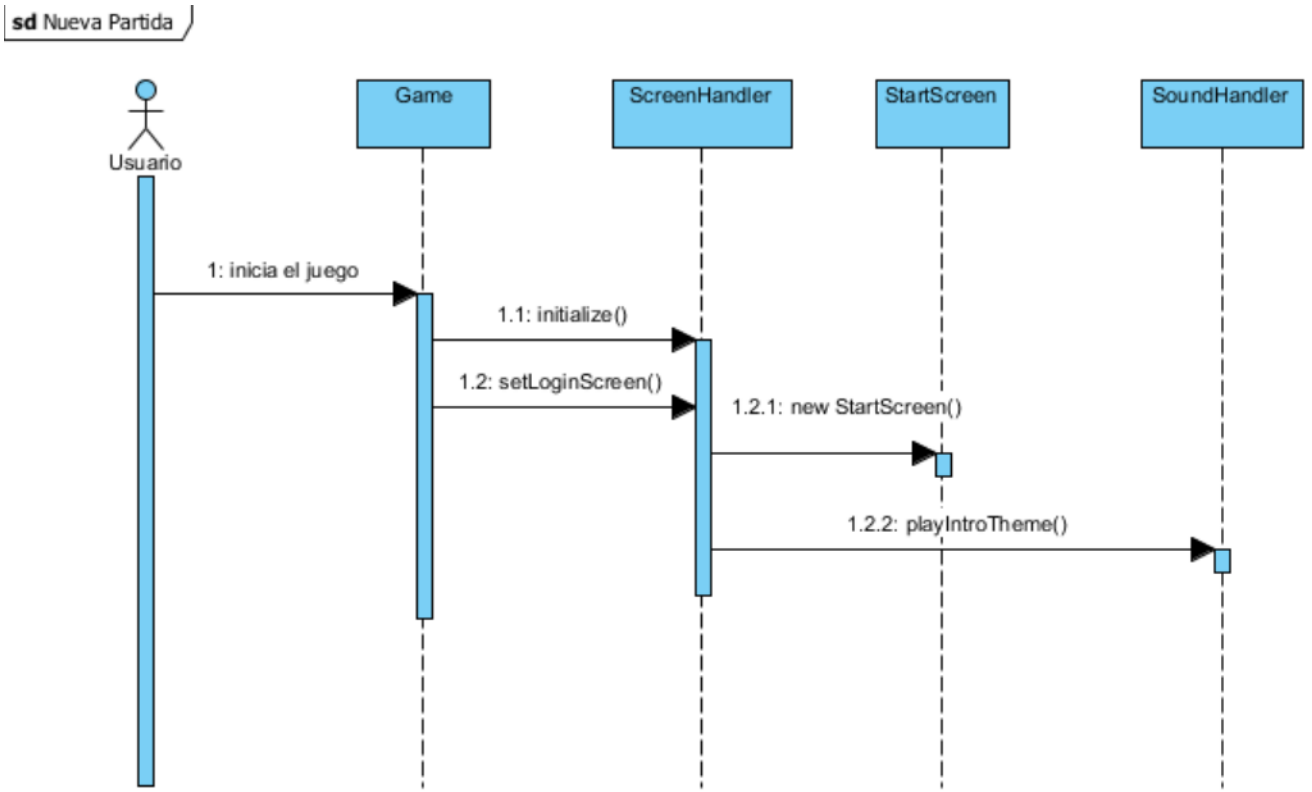
Ilustración 1

JERARQUÍA DE ACTORES

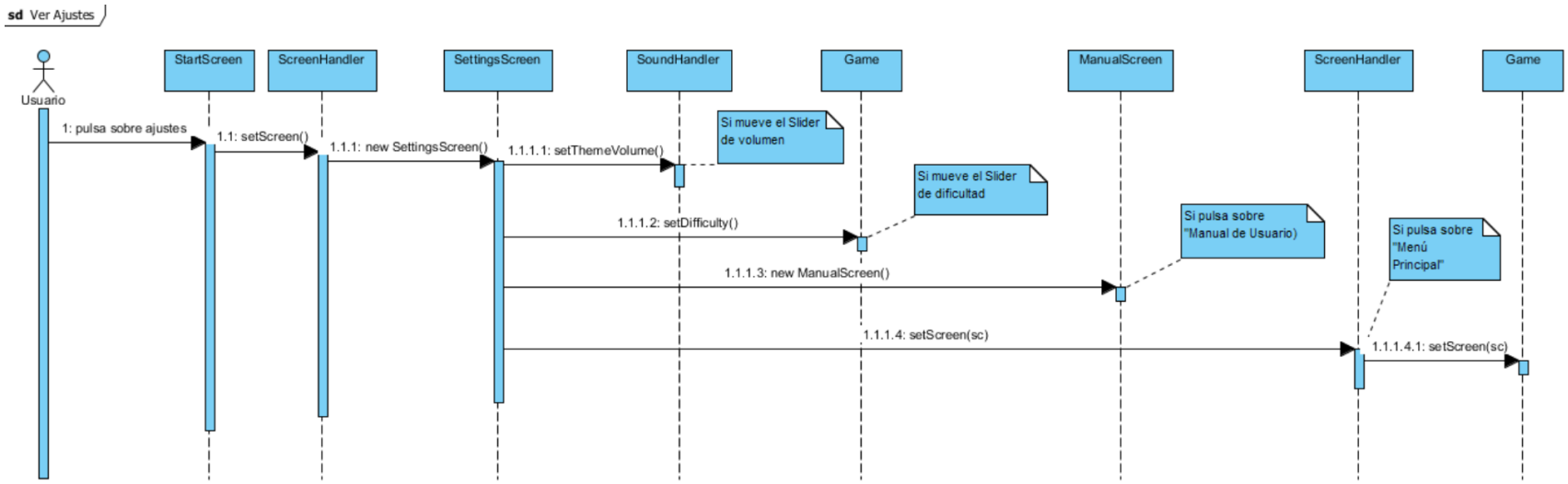
ANEXO II

DIAGRAMAS DE SECUENCIA

NUEVA PARTIDA



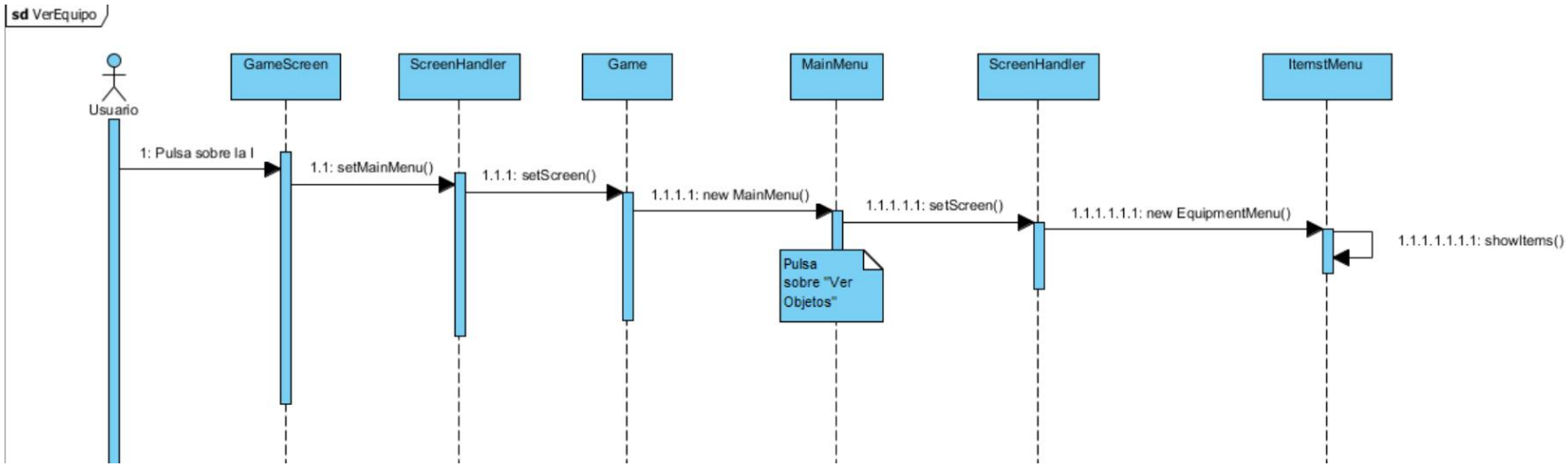
VER AJUSTES



En este diagrama no solo se incluye el Caso de Uso Extendido “Ver Ajustes, sino también sus tres sub-casos “Seleccionar Audio”, “Seleccionar Dificultad” y “Ver Manual/Ayuda”. Se ha hecho de esta forma ya que, no solamente son los casos muy similares en cuanto a las clases que se usan, sino que son también relativamente cortos y se ha estimado que de esta forma se daría una visión más global.

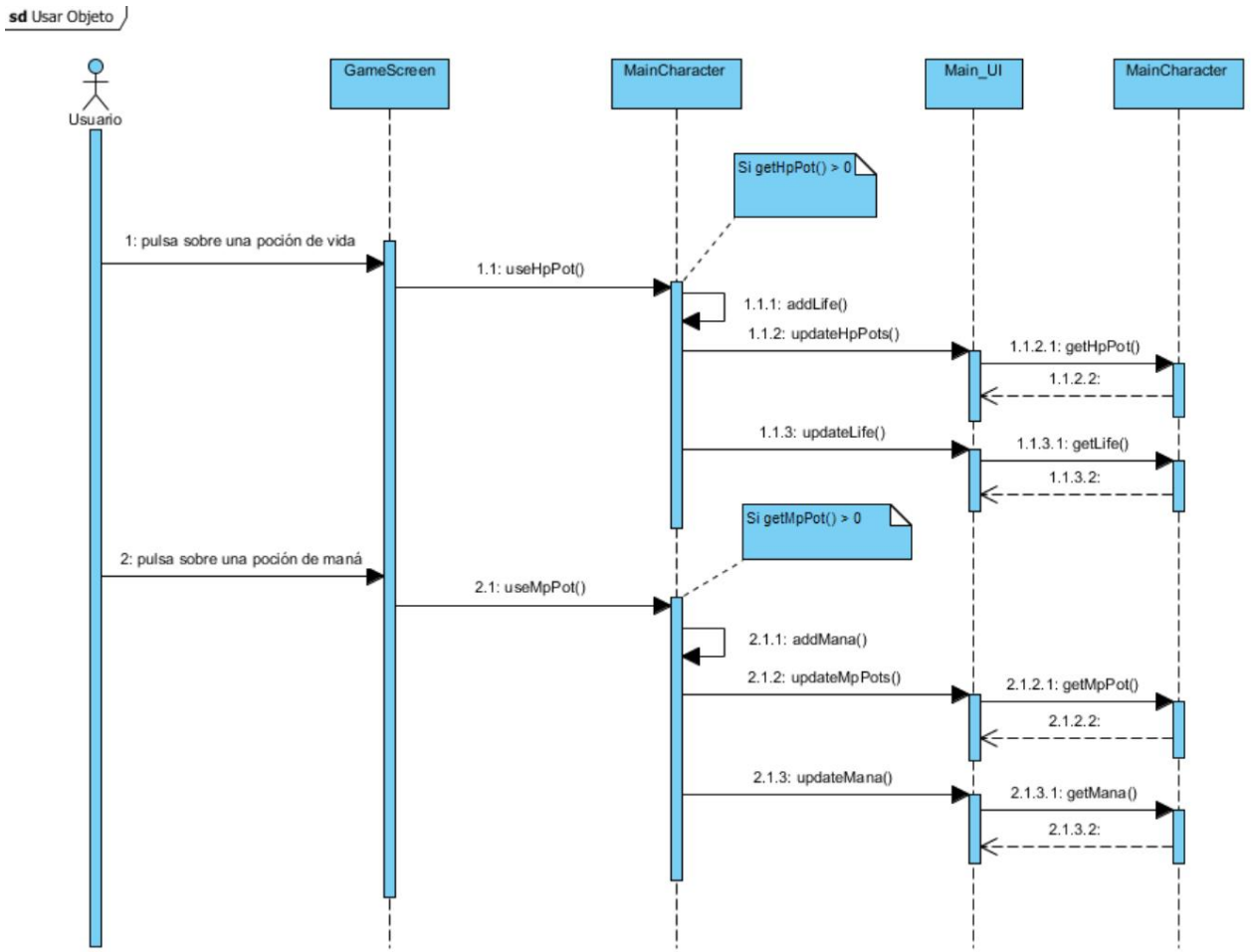
Aunque no se refleja en el diagrama, si nos encontrásemos en la pantalla de Manual/Ayuda, el pulsar sobre el botón “Menú de ajustes” dispararía exactamente la misma secuencia que el caso 1.1.4.1, siendo la única diferencia la pantalla parámetro (Una nueva SettingsScreen en este caso).

VER OBJETOS

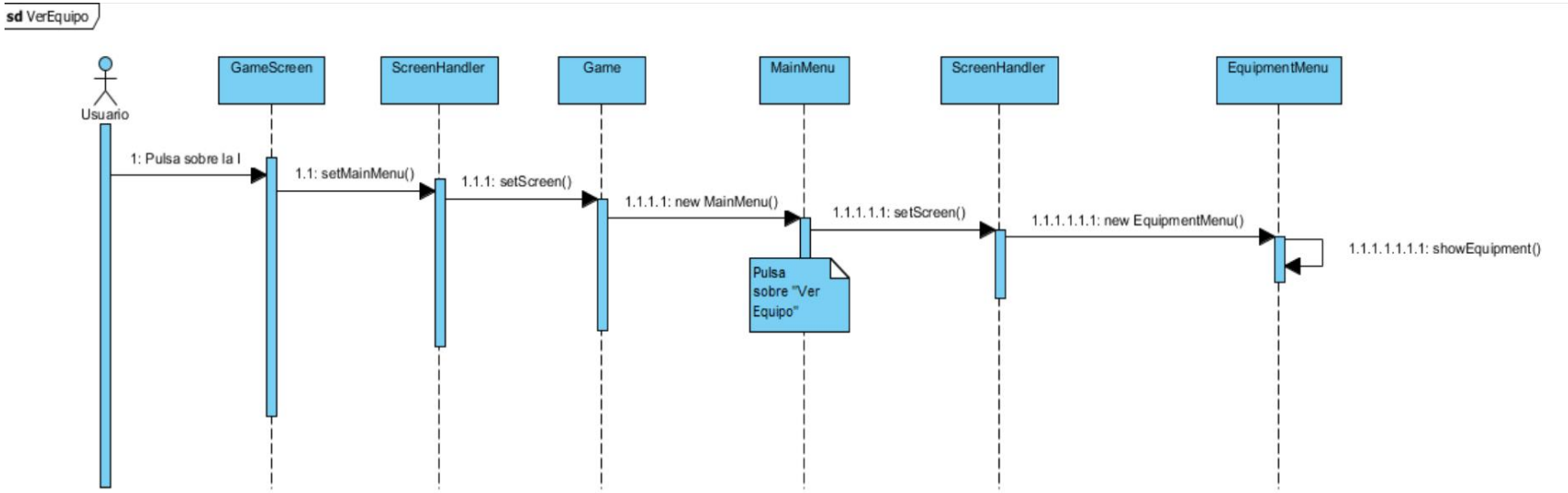


Como se ve en el diagrama de clases, ItemsMenu recibe en su constructora una referencia al jugador principal (MainCharacter). De esta forma, showItems() tiene acceso a los inventarios del jugador, y puede por lo tanto mostrarlos en pantalla.

USAR OBJETO

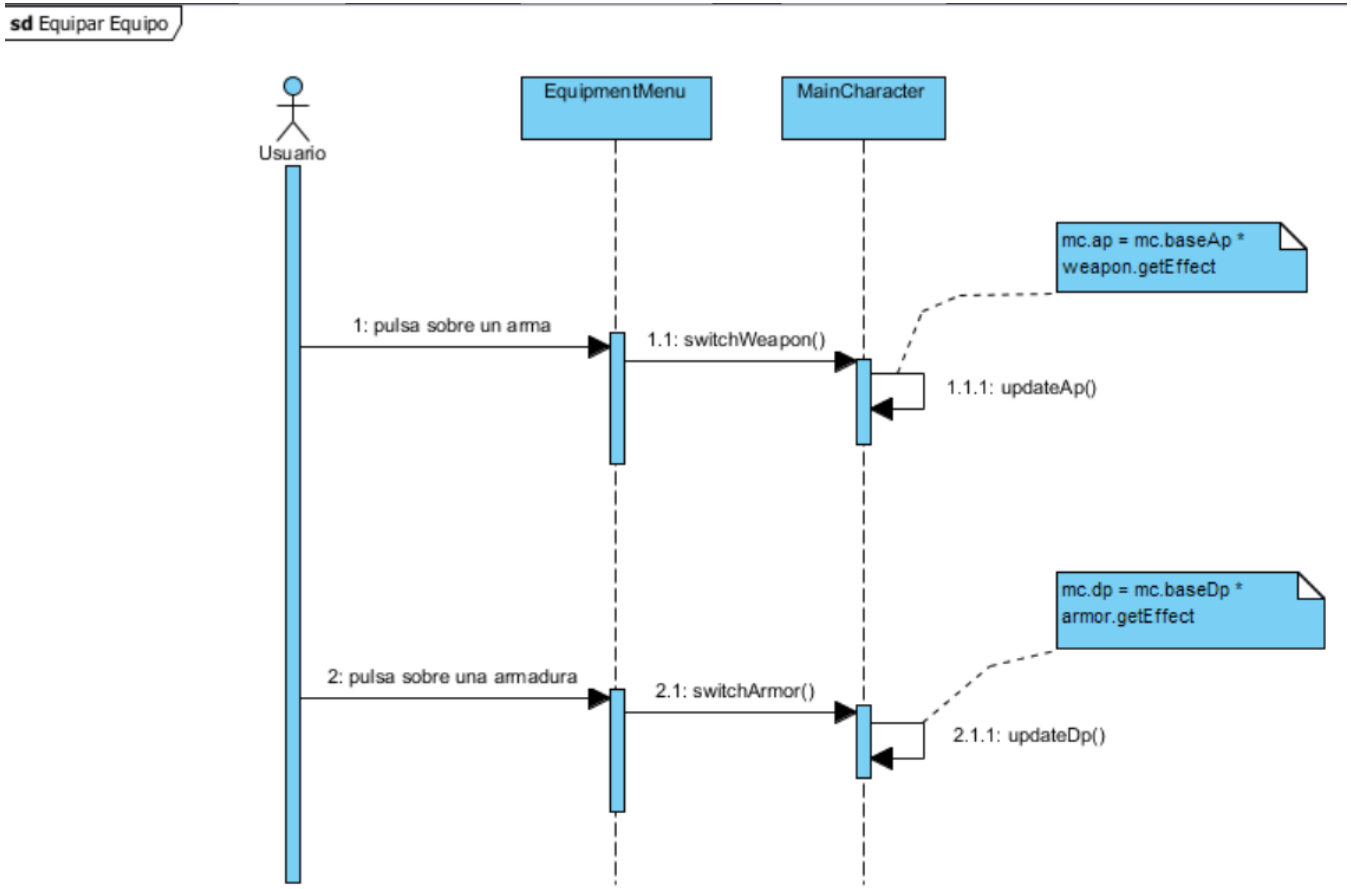

sd Usar Objeto
Usuario
GameScreen
MainCharacter
Main_UI
MainCharacter
Si getHpPot() > 0
1: pulsa sobre una poción de vida
1.1: useHpPot()
1.1.1: addLife()
1.1.2: updateHpPots()
1.1.2.1: getHpPot()
1.1.2.2:
1.1.3: updateLife()
1.1.3.1: getLife()
1.1.3.2:
Si getMpPot() > 0
2: pulsa sobre una poción de maná
2.1: useMpPot()
2.1.1: addMana()
2.1.2: updateMpPots()
2.1.2.1: getMpPot()
2.1.2.2:
2.1.3: updateMana()
2.1.3.1: getMana()
2.1.3.2:

VER EQUIPO

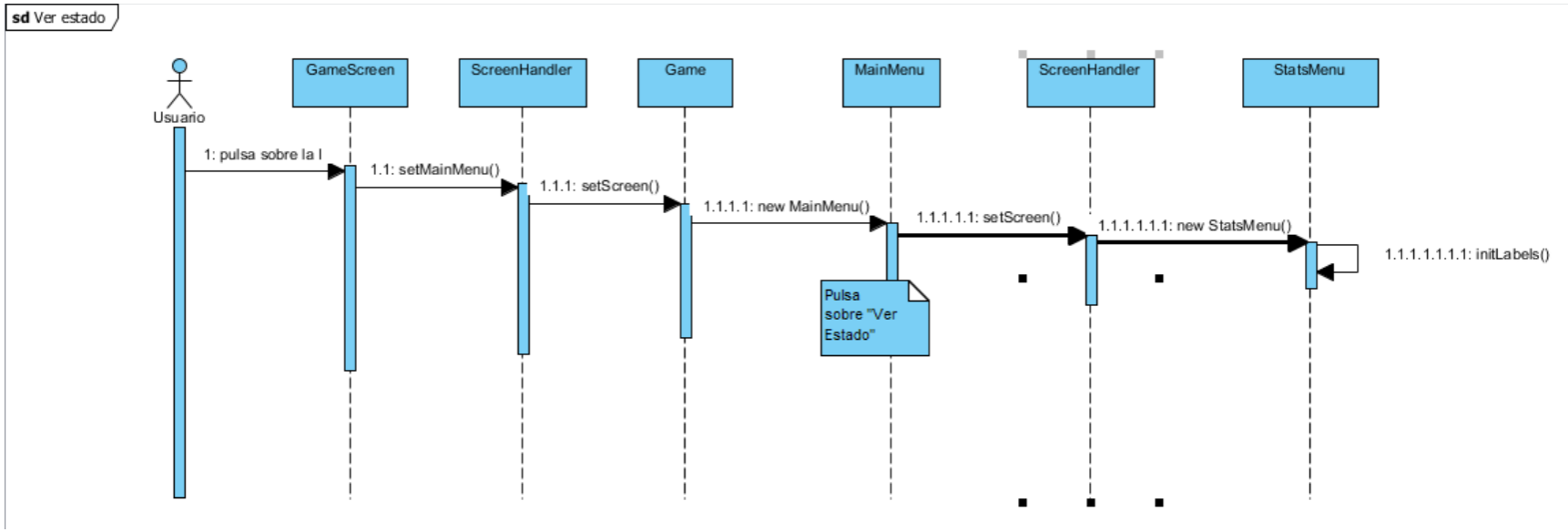


Como se ve en el diagrama de clases, EquipmentMenu recibe en su constructora una referencia al jugador principal (MainC haracter). De esta forma, showEquipment() tiene acceso a los inventarios del jugador, y puede por lo tanto mostrarlos en pantalla.

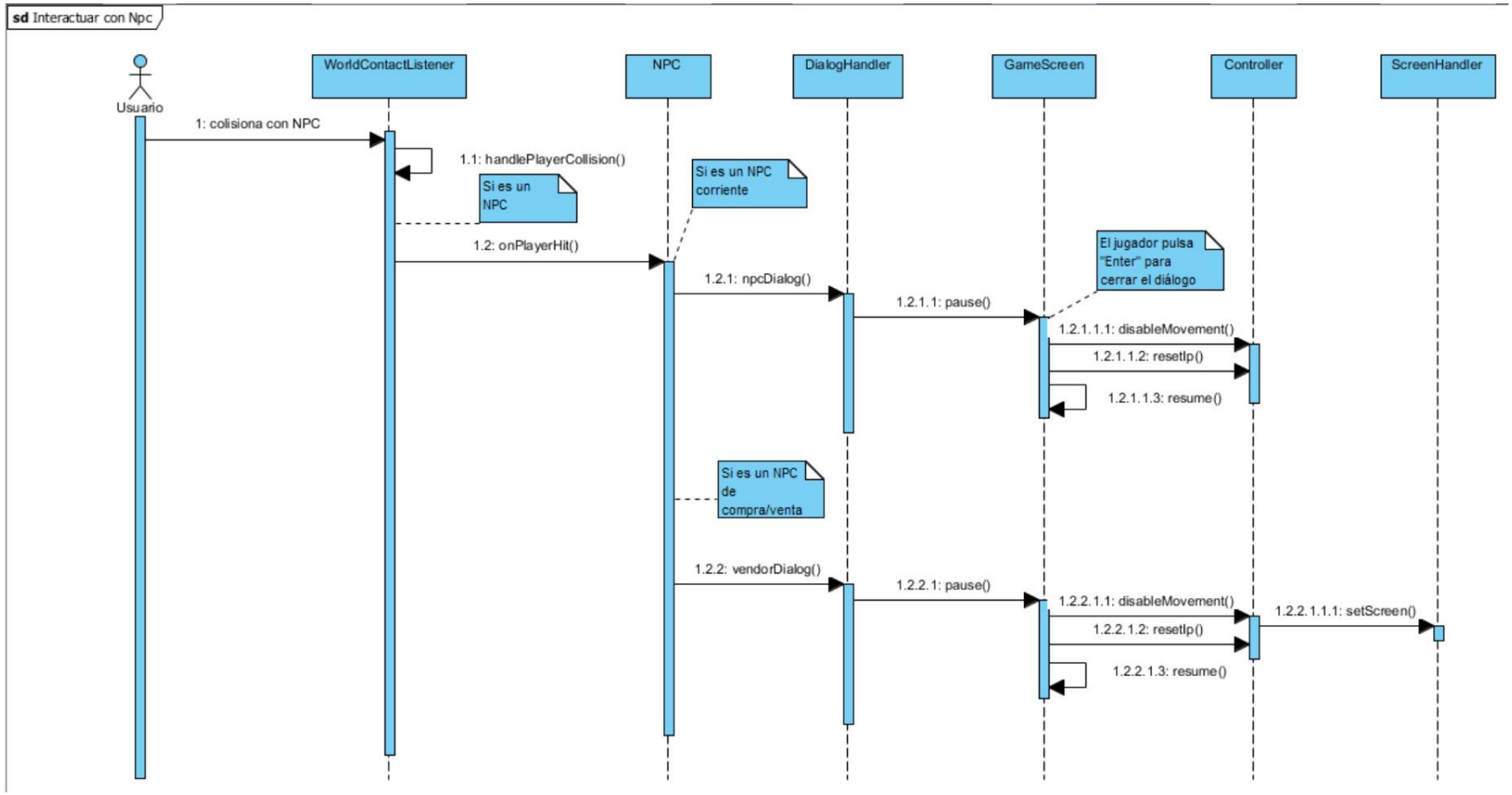
EQUIPAR EQUIPO



VER ESTADO

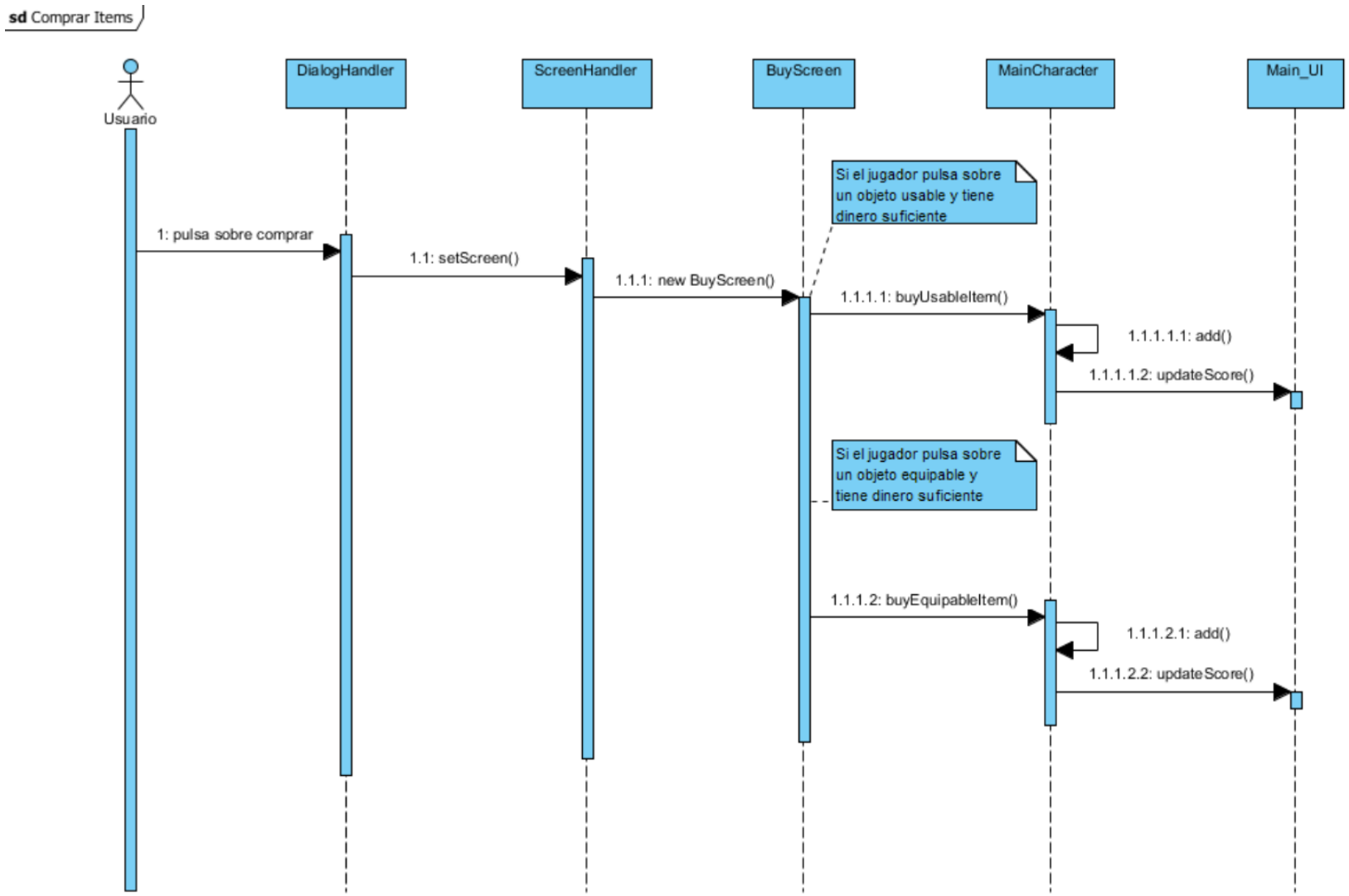

sd Ver estado
Usuario
GameScreen
ScreenHandler
Game
MainMenu
ScreenHandler
StatsMenu
1: pulsa sobre la I
1.1: setMainMenu()
1.1.1: setScreen()
1.1.1.1: new MainMenu()
1.1.1.1.1: setScreen()
1.1.1.1.1.1: new StatsMenu()
1.1.1.1.1.1.1: initLabels()
Pulsa sobre "Ver Estado"

INTERACTUAR CON NPC

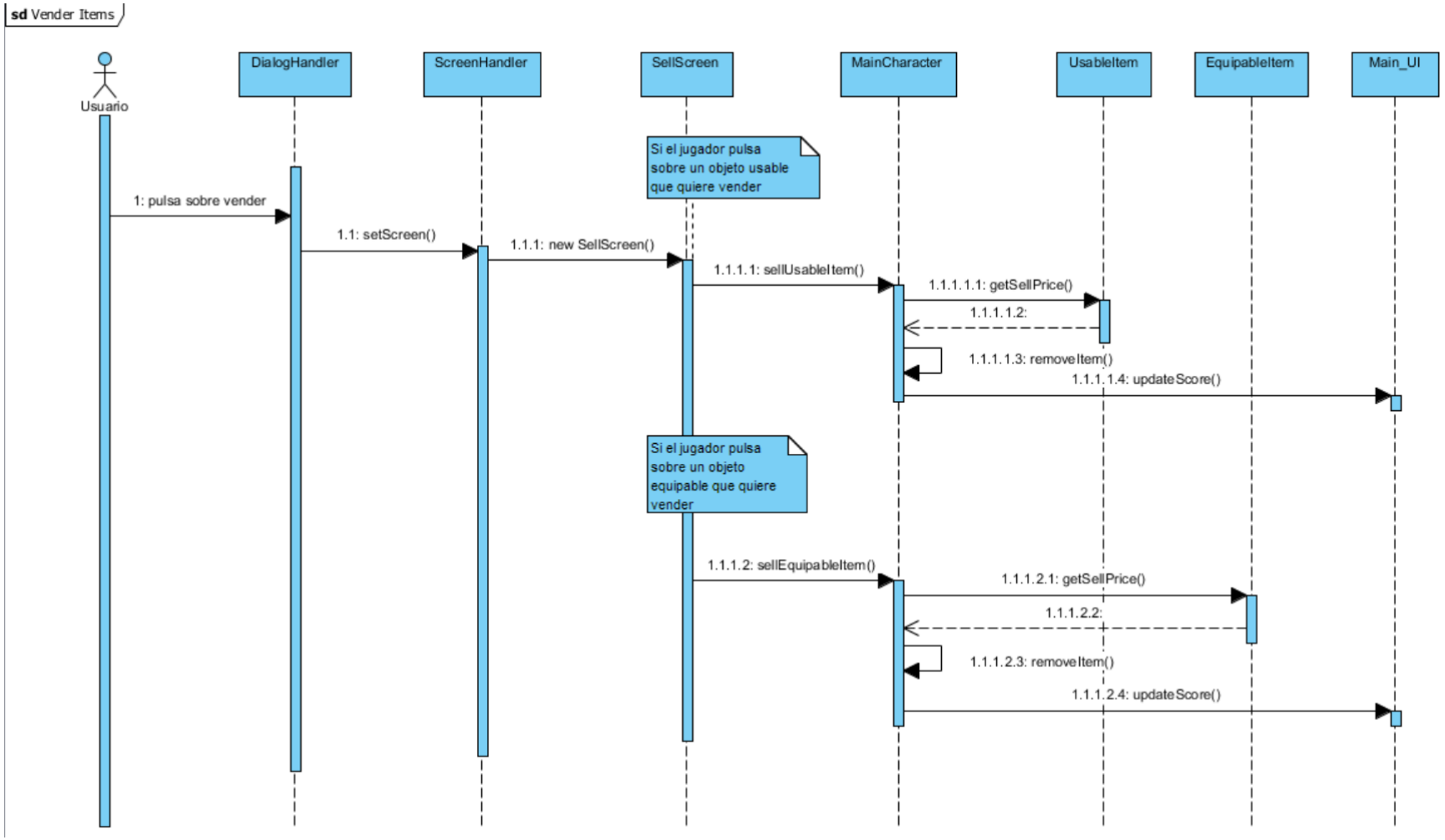


Uno de los parámetros de 1.1.2.1.1.1 setScreen() puede ser tanto una new BuyScreen() como una new SellScreen(), según la opción sobre la que pulse el jugador. Este caso se verá más a fondo en el siguiente diagrama de secuencia.

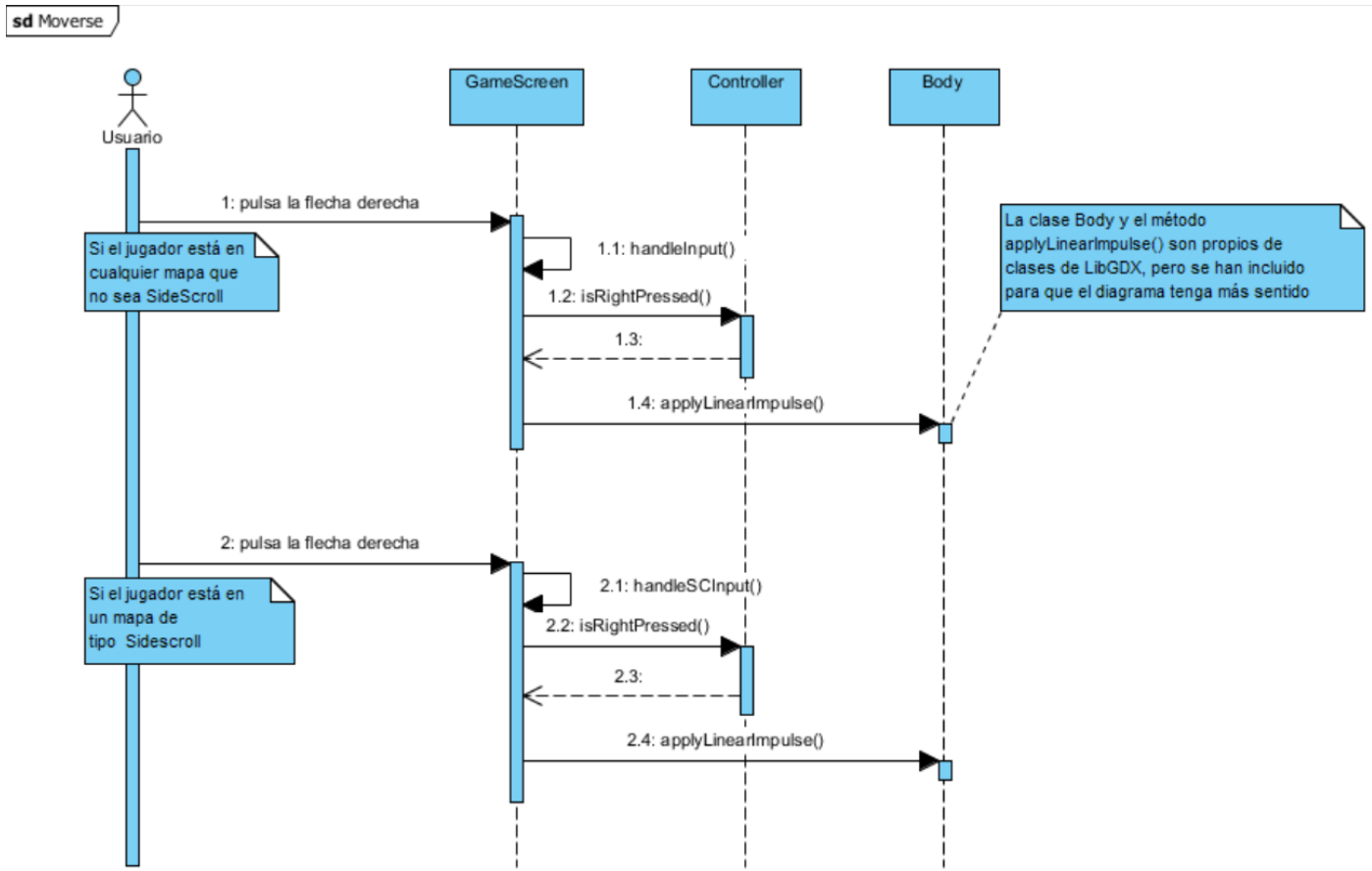
COMPRAR ITEMS



VENDER ITEMS

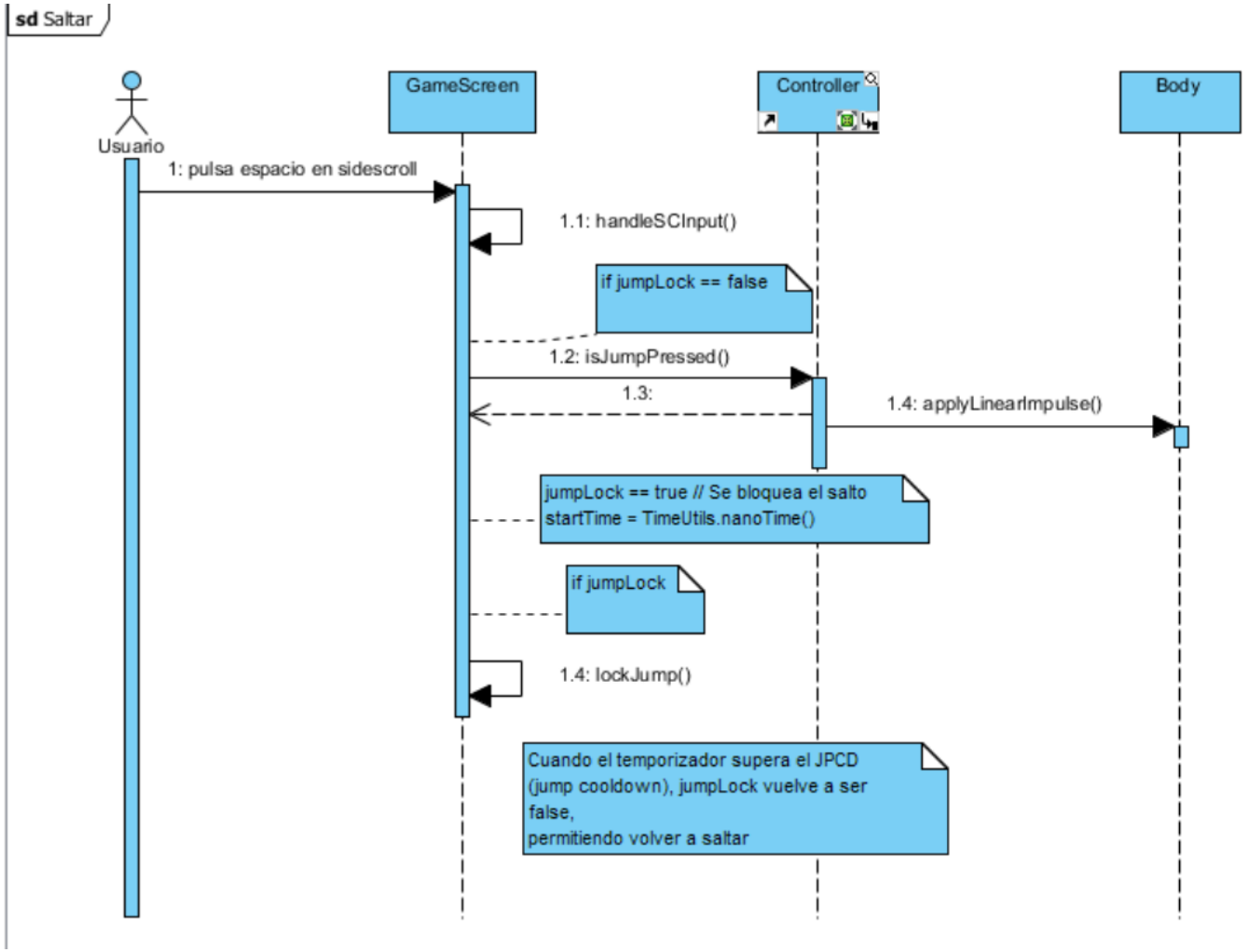


MOVERSE

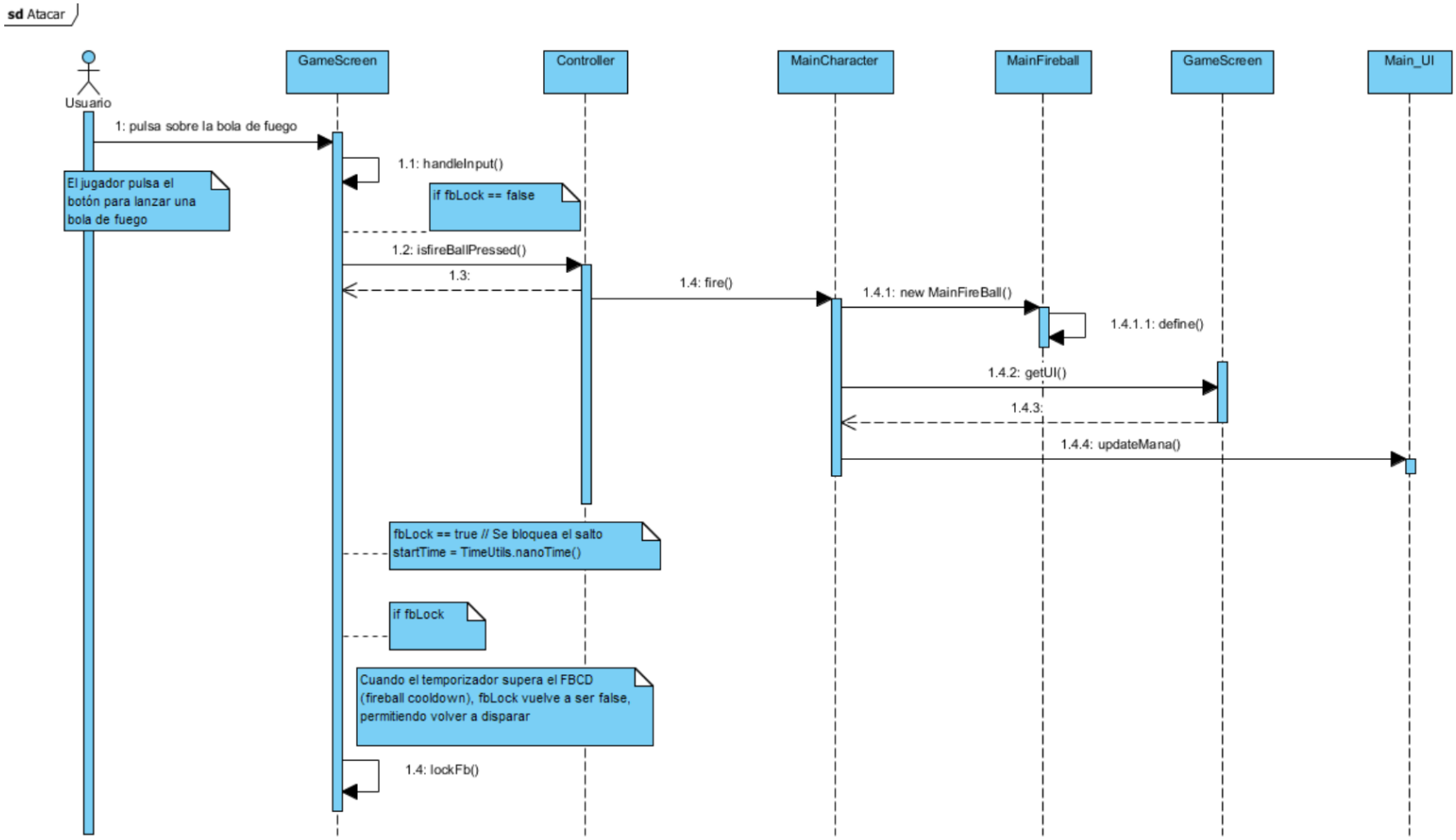


Por razones de claridad, se ha mostrado únicamente en el diagrama el caso de que el jugador se mueve hacia la derecha, haciendo la única comprobación isRightPressed(). En el modo normal, el jugador se puede mover también hacia adelante, atrás e izquierda. Para mostrar estos casos en el diagrama, únicamente habría que sustituir este método por isLeftPressed(), isUpPressed() o isDownPressed(), quedando el resto exactamente igual. Para el modo SideScroll, únicamente isRightPressed() y isLeftPressed() son los métodos comprobados, ya que solo se puede mover en dos direcciones.

SALTAR


sd Saltar
Usuario
GameScreen
Controller
Body
1: pulsa espacio en sidescroll
1.1: handleSCInput()
if jumpLock == false
1.2: isJumpPressed()
1.3:
1.4: applyLinearImpulse()
jumpLock == true // Se bloquea el salto
startTime = TimeUtils.nanoTime()
if jumpLock
1.4: lockJump()
Cuando el temporizador supera el JPCD
(jump cooldown), jumpLock vuelve a ser
false,
permitiendo volver a saltar

ATACAR



Al igual que ocurría en el diagrama de “Moverse”, solo se ha representado uno de los tres ataques posibles (bola de fuego, rayo y hielo). Esto se debe a que la casuística de las bolas de rayo y hielo son exactamente iguales a las de la bola de fuego, con la única diferencia de sustituir “Fireball” por “LightningBall” o “Iceball” ahí donde lo ponga.

INTERACTUAR CON OBJETO