UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de un videojuego de aventuras en C# sobre Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Fuentes Agustí, Alberto

Tutor: Mollá Vayá, Ramón Pascual

2013-2014

Resumen

El objetivo de este proyecto es crear un videojuego del tipo RPG de aventuras para dispositivos Android. Todo el desarrollo se ha realizado desde cero y se han implementado un nivel, tres personajes jugables, tres enemigos diferentes y varias zonas especiales. Se empleó el motor de juego Unity3D y el código fue escrito en C#.

Abstract

The project's goal is to create a RPG adventure video game for Android. All the development has been made from scratch, and a level, three playable characters, three different enemies and some special zones have been implemented. We used Unity as game engine and the code was written in C#.

Palabras clave: videojuego, Unity, C#, RPG, Android.

Keywords: videogame, Unity, C#, RPG, Android.

Tabla de contenidos

1. Introducción

1.1. Motivación

Uno de los productos más demandados últimamente en todo el mundo, desafiando las dificultades económicas actuales, son los videojuegos, cuyas ventas siguen creciendo año tras año [7]. En una tesitura similar a la de los videojuegos podemos encontrar a los dispositivos móviles: tablets y *smartphones* [8]. Desde hace unos años, este tándem videojuegos-dispositivos móviles ha ido fortaleciéndose mutuamente a un ritmo acelerado gracias a la gran cantidad de desarrolladores que aportan sus juegos y a que mucha gente quiere ocupar pequeños ratos libres en actividades lúdicas que pueda empezar y terminar en breves espacios de tiempo.

A todo esto ha contribuido la cultura que se está creando en Internet y las nuevas herramientas que tratan de agilizar el proceso de creación del software, ahorrando así meses de trabajo y reduciendo el tiempo de aprendizaje de las anteriores herramientas. Unity es una de estas nuevas herramientas.

La motivación personal que ha propiciad la realización de este proyecto ha sido, principalmente, el interés en trabajar en un futuro en esta área laboral. Con él se pretende realizar un acercamiento a los diferentes aspectos de la creación de un videojuego, desde el diseño hasta la implementación.

Por otra parte, se ha observado que en la Play Store [1] existe una gran competencia en juegos *arcade* y sociales, como los creados por *King* o *Gameloft*, pero se observa una carencia de juegos similares a los que se pueden encontrar en videoconsolas portátiles, como la *PlayStation Portable* o la *Nintendo DS*. Las anteriores y la afición por este tipo de juegos han sido las principales razones que han propiciado la elección de este trabajo.

En cuanto a la temática, un ambiente post-apocalíptico nos permite introducir una gran variedad de situaciones diferentes. Como jugadores de videojuegos y espectadores de películas de esta temática, se ha observado la gran cantidad de oportunidades que ofrece esta situación. Oportunidades que pueden desatar diferentes sentimientos. Se pueden encontrar desde situaciones cómicas hasta algunas realmente trágicas.

1.2. Objetivos

La finalidad de este proyecto es la de desarrollar un videojuego de aventuras-*RPG* para dispositivos móviles con sistema operativo Android, aprendiendo en

[1] Servicio de Google que recoge la mayoría de las aplicaciones para Android

el proceso los diferentes aspectos a tener en cuenta en un proyecto de tales características y profundizar en las capacidades de Unity.

Para lograr tal objetivo, previamente se ha redactado un documento de diseño para guiar el proyecto. Aquí se especifica el diseño de la interfaz, con el objetivo de conseguir que sea sencilla, funcional y consistente. También se describen los personajes y sus funciones, así como su aspecto, y los elementos que se pueden encontrar en el mundo.[2]

También se desea aprender de una forma práctica el proceso de creación de un videojuego y de la programación de éste, así como conseguir un producto que se pueda ampliar para conseguir un producto más completo y seguir aprendiendo.

[2] Este documento puede encontrarse en los anexos

2. Selección de herramientas

2.1. Motor de juego

En la actualidad se pueden encontrar una gran variedad de motores de juego, cada uno con sus características distintivas, sus ventajas y sus inconvenientes. Entre todos ellos se han tenido en cuenta los que se centraban en el desarrollo de videojuegos para dispositivos móviles.

Corona SDK es una herramienta centrada en el desarrollo de juegos en 2D para móviles y tablets creada por *Corona Labs Inc.* Posee un gran catálogo de librerías para ayudar en la programación de diversos aspectos, y permite su uso de forma gratuita, sacrificando el uso de ciertos aspectos reservados a los usuarios de pago. Utiliza el lenguaje de script Lua y es *cross-platform* (el mismo código sirve para varias plataformas). *Major Magnet, Blast Monkeys, The Lost City* y *Freeze!* son algunos de los videojuegos que lo han usado.

Marmalade C++ SDK, de *Marmalade Technologies Ltd.*, es un motor *cross-platform* que permite programar videojuegos para una gran variedad de dispositivos usando C++, acelerando dicho proceso gracias a sus librerías. Algunos videojuegos programados con Marmalade SDK son *Godus, Draw Something, Plants vs Zombies* y *Vector.*

Cocos2d-x es un motor de juego de código abierto desarrollado por el *MIT.* Facilita el desarrollo de videojuegos para cualquier plataforma con C++, Lua o Javascript. También posee potentes características que lo hacen una buena elección frente a otros. *Badland, Castle Clash, Star Wars: Tiny Death Star* y *Family Guy: The Quest for Stuff*, entre otros, lo han empleado.

Unity3D (Unity) es propiedad de *Unity Technologies*, y actualmente es uno de los motores de juego más importantes y extendidos. Permite utilizar Javascript[3], Boo y C#, y dispone de un editor que agiliza el desarrollo del videojuego. También posee una amplísima comunidad, que ayuda a resolver dudas y crea *plugins* para cubrir aspectos puntuales del desarrollo, algunos de los cuales son muy potentes y su uso está muy extendido. Entre todos los videojuegos que se han desarrollado usándolo podemos destacar algunos: *Monument Valley*, *The Forest, Oddworld: New 'n' Tasty, Rust* y *Broforce.*

Finalmente se decidió emplear Unity, pese a las similitudes entre todos, debido a su fuerte comunidad que proporciona apoyo ante cualquier duda, crea tutoriales para ayudar a realizar ciertas tareas y *plugins*[4] que cubren o

[3] Realmente es un lenguaje muy similar a Javascript, Unityscript.
[4] Aunque se decidió no utilizar ninguno, en el futuro se podría utilizar alguno para mejorar algún aspecto del videojuego.

simplifican algunas funcionalidades deseables y cuya implementación podría ser complicada o enrevesada.

2.2. Lenguaje de programación

Una vez elegido Unity como motor de juego se debía escoger un lenguaje de programación. Este motor permite emplear Javascript, Boo y C#, incluso a la vez. Debido a recomendaciones de los foros de Unity que indicaban que utilizar ambos en un mismo proyecto propiciaba que éste fuese propenso a errores, ya que habría que tener en cuenta el orden de ejecución y de compilación de cada script, se descartó rápidamente esta opción. Boo, un lenguaje OO[5] inspirado en Python, también fue descartado al principio debido a la falta de apoyo por parte de la comunidad.

Teniendo en cuenta que C# presentaba una sintaxis mucho más similar a la de Java, un lenguaje OO con el que se ha trabajado en gran parte de la carrera, se decidió utilizar éste. Además proporciona un mayor control, aunque esto mismo exija una mayor responsabilidad por parte del programador. Por último, y para reforzar la decisión, se consultó algunas fuentes digitales (principalmente foros) y libros, y la mayoría recomendaban C# [9].

2.3. Plataforma objetivo

Debido al auge de los dispositivos móviles es una gran oportunidad sacar un videojuego centrándose en ellos. Android e iOS son los SO[6] mayoritarios, sin ningún rival que suponga una amenaza real ahora mismo.

iOS presenta una gran base de usuarios, y la monetización de sus aplicaciones es mayor a la de Android. Pese a ello, presenta algunas dificultades generales como son el requisito indispensable de poseer una licencia de pago para desarrollar aplicaciones para ella y las rígidas reglas de diseño que exigen. Además de ello, se requiere un ordenador con OS X para compilar la aplicación.

Android fue la elección debido a que, además de lo anteriormente dicho, sólo se disponía de dispositivos con Android, y eran necesarios para realizar las pruebas y asegurarnos que realmente funcionaba todo correctamente.

Pese a haberse decidido a utilizar Android, dado que Unity permite el desarrollo *cross-platform*, el código desarrollado podrá ser empleado, si no en su totalidad en gran medida, en la creación de la versión para iOS o, incluso, para Windows Phone, Linux, OS X y consolas.

[5] Orientado a objetos
[6] Sistema operativo

3. Introducción a Unity3D

Unity se fundamenta en unos objetos llamados *GameObjects*. La mayoría de los elementos del juego, desde los *scripts* hasta elementos gráficos suelen depender de ellos.

Empezando por el principio, se introduce el concepto *Scene*. Una *Scene* es un entorno donde se desarrolla una parte del juego, incluyendo menús, ya que todo debe estar dentro de una. A menos que se especifique lo contrario, todos los elementos contenidos en un *Scene* se eliminan cuando se cambia a otra *Scene*. En la Ilustración 1 se puede observar una *Scene* con unos elementos (personajes, terreno, cámara, iluminación, ...) que formarán parte del juego. Una cuestión importante a tener en cuenta es que el editor sólo es una ayuda, y no por verse bien en la ventana de *Scene* se va a ver bien en el juego, ni al contrario, pues pueden existir *scripts* que cambien completamente el comportamiento de los elementos de la *Scene*, incluyendo a la cámara y componentes de renderizado.

Ilustración 1. Scene en el editor de Unity

Dentro de cada *Scene* se encuentran los *GameObjects*, que representarán a los diferentes elementos, desde personajes hasta la cámara, y que son la clase básica de Unity. Cualquier componente que queramos que tenga el juego debe estar contenida en un *GameObject*[7], siendo un componente de éste. Un símil en Java sería la clase *Object*. Algunos de los componentes más importantes son el *Renderer*, que se encarga de que el objeto sea visible, dándole una forma y un color o textura, el *Rigidbody* y el *Collider*, que gestionan las colisiones con otros elementos y las características de la física simulada por Unity, el *Camera*, que es, como indica su nombre, el encargado de renderizar la *Scene* de acuerdo a su configuración. Los componentes pueden modificarse mediante código, en un script que modifique los atributos de éstos, lo que permite modificar su

[7] A excepción de algunos *scripts*, pero son casos puntuales.

comportamiento en tiempo de ejecución, pero si esto no es necesario siempre se puede emplear el inspector de *GameObjects*, que permite modificarlos de una forma sencilla y sin necesidad de crear un *script* o de escribir código (véase Ilustración 2).

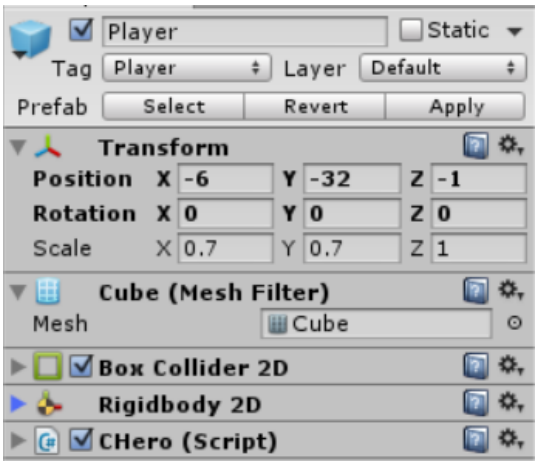


Ilustración 2. GameObject y sus componentes en el inspector

Otro elemento principal e indispensable de Unity es el componente *Transform*. Todos los *GameObjects* poseen un único componente de este tipo y es necesario; no se puede eliminar. Es una parte esencial del *GameObject* que indica la posición, la escala y la rotación de éste. Cualquier *Transform* puede tener un padre en la jerarquía. Esto implicará que la posición, rotación y escala se volverán relativas a su padre. En el caso de no tenerlo, los anteriores atributos serán relativos a la *Scene*.

Respecto al aspecto de interfaces, Unity carece de un sistema gráfico para diseñarlas, por lo que la mayoría de sus componentes, a excepción del texto, se debe programar mediante código. Esto presenta dos dificultades. La principal es la de que no se puede ver el diseño programado hasta que se ejecuta la aplicación. La segunda es más bien una peculiaridad de este motor. Cada vez que se ejecuta *OnGUI()*, el método encargado de dibujar los elementos de la interfaz, se destruyen todos los anteriores componentes y se redibujan por completo de nuevo.

Para hablar del ciclo de ejecución de los *scripts*, primero se debe explicar la clase *MonoBehaviour*. Esta clase es un pilar principal del motor. Cualquier clase que quiera acceder a ciertas funciones básicas del motor debe heredar de ésta. Esto permite acceder a métodos como el *Start()*, que se ejecuta la primera vez que el *script* se activa en la *Scene*, o el *Update()*, que se ejecuta continuamente mientras el *script* esté activo, una vez cada *frame*[8] . También contiene métodos *listeners*[9],

[8] Actualización de la pantalla
[9] Métodos que se ejecutan cuando un evento determinado es lanzado

que resultan muy útiles para controlar colisiones, entradas de usuario, eventos de renderizado, eventos del sistema, etc.

Una vez introducida la clase *MonoBehaviour,* podemos pasar a descubrir el flujo de ejecución de los *scripts.* Debido a que Unity Technologies no proporciona una documentación clara en este aspecto, algunos usuarios han experimentado hasta encontrar el ciclo de vida de los *scripts* (véase Ilustración 3). *Awake(), Start()* y *Update()* son los métodos más empleados, pues sirven para inicializar los objetos y actualizarlos a lo largo de la ejecución del juego. *Awake()* se emplea para inicializar los propios atributos, independientes de los demás objetos. *Start()*, cuyo funcionamiento es muy similar al *Awake()* pero que se ejecuta posteriormente, se suele utilizar para inicializar variables que dependen de otros métodos y que ya habrán sido preparadas en el *Awake().* En cuanto a *Update()*, se ejecuta después de las anteriores, y a diferencia de éstas se ejecutará continuamente. Se emplea para actualizar estados y atributos, modelando el comportamiento de los objetos a lo largo de la ejecución.

Otra característica muy útil que nos proporciona *MonoBehaviour* son las corrutinas. Se trata de métodos especiales que permiten retrasar la ejecución un tiempo determinado y que se ejecutan en paralelo al resto del código, resultando útiles, por ejemplo, para programar algunas animaciones. Estas corrutinas tienen la peculiaridad, respecto al resto de métodos, de que no dejan de ejecutarse aunque el *script* que las contenía se desactive. No se ejecutan como métodos normales, sino mediante el método *StartCoroutine()*, y se detienen cuando terminan de ejecutar su código o mediante una llamada a *StopCoroutine().*

A parte de las anteriores herramientas, que se pueden considerar más técnicas y enfocadas al comportamiento del juego, Unity posee otras características muy interesantes. Todo esto lo convierte hoy en día en un motor muy empleado por una gran variedad de empresas y desarrolladores independientes. Como se puede observar en la información proporcionada por Unity [13], que se basa en muchos casos en informes y estadísticas externas como se puede observar en las fuentes citadas, este motor presenta una gran penetración en el mercado y casi (un 47%) de los desarrolladores de juegos han empleado Unity alguna vez mientras que un 29% lo emplea como su motor principal. Además, su base de desarrolladores registrados asciende a más de 3,3 millones, lo que resulta posible gracias a su política de licencias, que permite a cualquier desarrollador utilizar Unity para crear su juego gratuitamente, sacrificando algunas características secundarias como la posibilidad de usar un servidor que gestione el proyecto para un desarrollo concurrente o algunas opciones gráficas avanzadas. La licencia gratuita también limita el volumen de ventas e ingresos siendo que si alguno de estos se supera, se debe comprar la versión completa (Unity3D Pro).

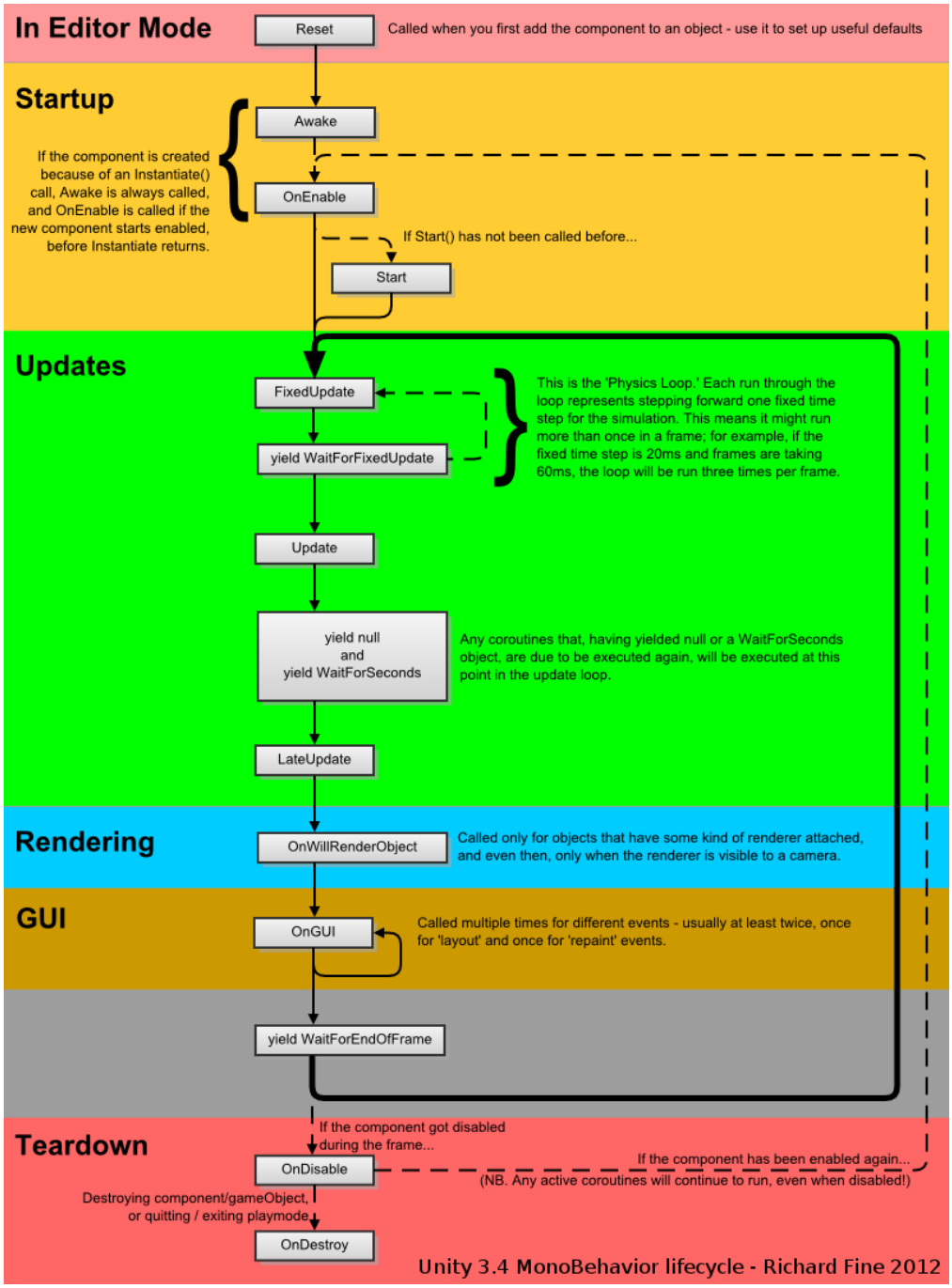


Ilustración 3. Ciclo de vida de MonoBehaviour, por Richard Fine

4. Análisis

Al comenzar el proyecto se realizó un proceso para prepararlo, estableciendo las necesidades y la planificación que se seguiría. Esta es la primera fase del proyecto que precederá al diseño y a la implementación.

Primero se realizó una planificación aproximada del tiempo del proyecto, representado mediante un diagrama de Gantt (véase Figura 1), que después permitiría realizar una comparación con el tiempo que finalmente dedicamos a cada apartado (véase Figura 2).

A la fase de diseño, formada por el diseño de la estructura de clases y al de las estructuras de datos, se decidió darle un tiempo considerable, 12 días, aunque finalmente se empleó un tiempo menor, 10 días. Esto fue debido a que no fue necesario diseñar varias clases para los enemigos, ya que lo único que cambian entre ellos son los atributos que permiten modificar la fuerza y frecuencia de su ataque, la defensa, la vida y la experiencia. También se comprobó que el cálculo inicial del tiempo, que se estimó en 12 días debido a la dificultad que pensábamos que nos podía presentar este proceso, fue menor porque las pautas aprendidas en la carrera agilizaron el proceso.

La implementación fue la que consumió la mayor parte del tiempo. Ya desde el principio se pensaba que así sería, y aun así el cálculo resultó ser demasiado optimista. Se le otorgaron 34 días, pero finalmente ocupó 52, pues surgieron más problemas de diversa índole que no se esperaban, algunos de ellos relacionados con el uso de Unity, como el control de la interfaz y algunos aspectos de la persistencia de los datos.

También se tuvo en cuenta una fase extra dedicada a solucionar errores y fallos menores, pues al programarlo en un ordenador, el resultado era susceptible a no funcionar de la misma manera en un dispositivo móvil. A esta fase se le asignaron 6 días, pero finalmente, y debido a la demora en la implementación, se alargó hasta los 12.

Después se observaron videojuegos similares para visualizar la magnitud del proyecto completo. Tras esto, se decidió que se recortarían algunos aspectos y se añadirían otros para dar cabida al proyecto en el tiempo disponible y centrarse más en los aspectos informáticos del proyecto, dejando en segundo lugar a los más visuales y artísticos. Entre los aspectos recortados se encuentran muchos aspectos gráficos, como efectos visuales y sonoros, pues quedaban fuera de lo que nos interesaba mostrar en este proyecto. Principalmente los efectos eliminados (o pospuestos para futuras actualizaciones) son la inclusión de efectos sonoros en los ataques recibidos y realizados en combate, un efecto visual y sonoro cuando el enemigo o el héroe realiza un ataque o lo recibe, otro

para cuando el héroe sube de nivel o es derrotado. También se pospuso la creación de un jefe final debido a que, para darle consistencia, implicaba añadir otros elementos como una misión que llevará al usuario hasta él y algún efecto que mostrara que es diferente a los demás. En cuanto a las características añadidas, la más notoria fue la adaptación de estos videojuegos, típicamente jugados con controladores analógicos, a un dispositivo con entrada táctil.

También se analizaron las necesidades que se debían cubrir en el juego. Para ello se realizó un diagrama de casos de uso (véase Diagrama 1) [10]. El usuario debe poder crear a su personaje con el que jugará, podrá aumentar y disminuir el volumen de la música internamente dentro del juego, podrá leer las instrucciones para saber cómo manejar a su personaje así como información relativa al juego. También podrá guardar y cargar partida además de borrar a un personaje. Y por supuesto, podrá navegar por los diferentes menús del juego con bastante libertad.

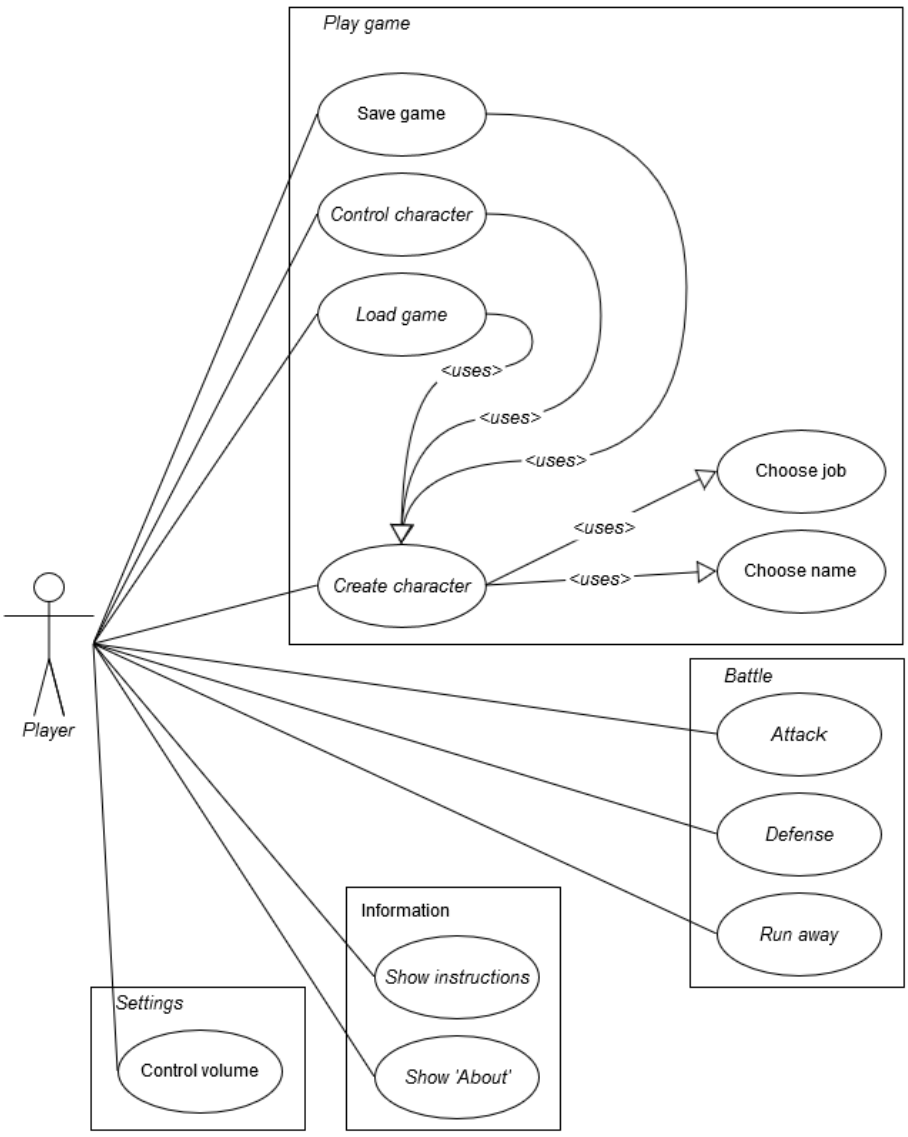


Diagrama 1. Casos de uso.

Para controlar gran parte de juego se pensó en una máquina de estados que controlara diversos eventos como la aparición de los enemigos o el flujo del combate. Esta máquina debía cubrir las necesidades del juego, permitiendo gestionar los sucesos y las acciones que tendrían lugar al ejecutar el juego, al desplazarse por el mapa, al entrar en un combate, a salir de éste, etc. La máquina resultante de dicho análisis se puede encontrar en el Diagrama 2. Por defecto, el estado inicial es "null". Una vez se inicializa el héroe, su estado pasa a "standing", que se trata del estado que indica que el jugador no está en una batalla y su personaje está quieto. La única forma de cambiar esto es que el personaje se desplace, con lo que su estado cambiaría a "exploring", que expresa que el jugador está moviéndose. Ese estado se mantendrá mientras no separe, con lo que volvería al "standing" o mientras no aparezca un enemigo, momento en el cual el estado pasaría a ser "waiting", que es el estado encargado de, en una batalla, impedir que el jugador realice ninguna acción y espere su turno de atacar. Desde este estado pueden suceder dos cosas: que se termine el tiempo de espera y pasemos al estado "attack" que indica que ya se puede atacar y desbloquea las acciones pertinentes o que el enemigo nos derrote y pasemos al estado "defeat". En el primer caso, podrán darse tres transiciones: que el jugador decida atacar o defenderse, con lo que volveríamos al estado "waiting", que el jugador decida huir, con lo que pasaríamos al estado "standing", que el jugador sea derrotado e iría al estado "defeat" y por último, que el jugador ataque y derrote al enemigo, con lo que pasaríamos al estado "victory". El estado "victory" se encarga de procesar la victoria, otorgar recompensas y demás acciones a tomar cuando el jugador gana un combate. Una vez haya terminado de procesarla, el estado pasa a ser "standing" de nuevo. Si el jugador es derrotado y nos encontramos en el estado "defeat", la partida se carga automáticamente desde el último punto guardado.

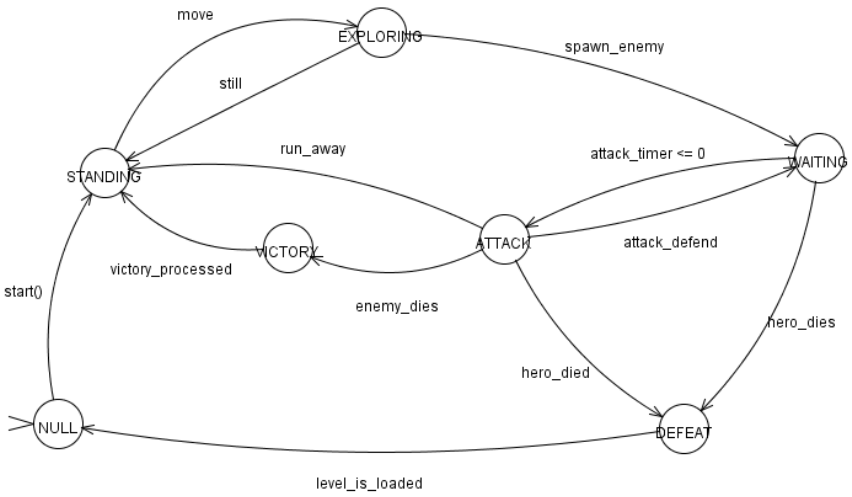


Diagrama 2. Máquina de estados del héroe.

En cuanto al control de los enemigos, se realiza mediante otra máquina de estados subordinada a la primera, pues sólo se pondrá a funcionar cada vez que la otra la prepare para ello cuando deseemos, que en este caso será cuando aparezca un enemigo. Cuando esto suceda, el enemigo será puesto en el estado "waiting", que al igual que en el héroe es un estado que esperará hasta que pueda realizar alguna opción, cosa que en los enemigos se limita a atacar. Cuando esta cuenta venza, se pasará al estado "attacking", donde, una vez que ataque, volverá al estado "waiting". Por supuesto, si el enemigo es derrotado en cualquiera de los dos estados anteriormente mencionados, es decir, si su salud llega a 0, pasará al estado "diying", que lo pondrá en espera de ser invocado de nuevo, cosa que lo devolverá al estado "waiting", además de dejarlo listo para el combate.

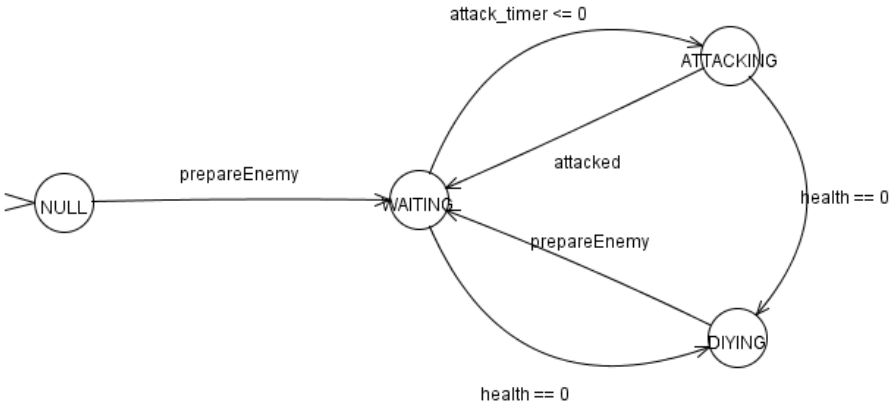


Diagrama 3. Máquina de estados de los enemigos.

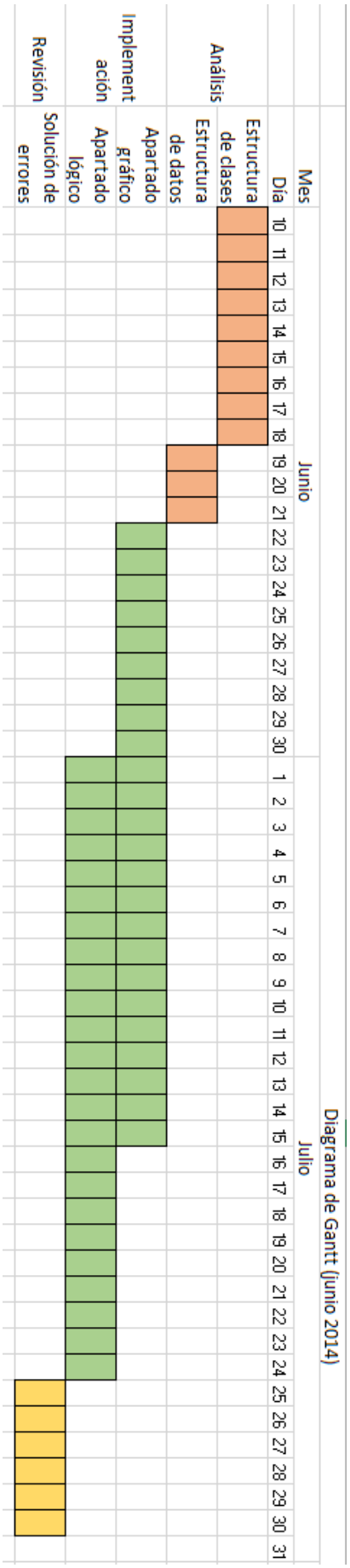


Figura 1. Diagrama de Gantt (junio 2014)

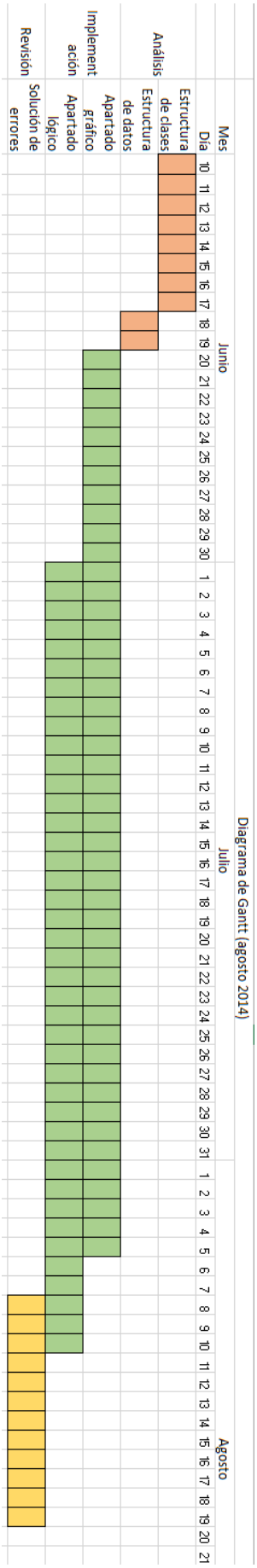


Figura 2. Diagrama de Gantt (agosto 2014)

5. Diseño

La creación de un videojuego requiere un diseño a varios niveles. Se necesita saber cómo va a ser el videojuego, el concepto, que se describirá en un documento de diseño del videojuego (*GDD, Game Design Document*)[10] y cómo se va a programar y qué clases van a ser necesarias. Para explicar este proceso, se ha dividido el aspecto de la programación en dos partes, ya que se ha considerado que el diseño de la interfaz es una parte suficientemente extensa y diferenciada de la lógica interna del resto de clases como para formar otro punto.

5.1. Aspecto conceptual

El videojuego se puede clasificar como un *RPG* de aventuras. El jugador se crea un personaje, escogiendo entre tres clases diferentes para lograr diferentes estilos de juego para ofrecer variedad. Estas clases son las siguientes:

- *Engineer*: es más resistente que los demás, por lo que resiste más ataques antes de morir pero tarda más en eliminar con los enemigos.
- *Magician:* su ataque es superior al de los demás, pero es menos resistente.
- *Technomancer:* su resistencia y ataque están equilibrados, situándose en el medio entre las otras dos clases.

A cada tipo de héroe, para conseguir estos efectos, cuando es creado se le asignan unos atributos diferentes. Estos atributos se actualizarán cuando se sube de nivel, mejorándose.

El jugador puede desplazar a su héroe a lo largo y ancho del mapa. Cada cierto tiempo aleatorio, un enemigo aparecerá. Este enemigo será elegido por el sistema aleatoriamente entre tres tipos diferentes, donde cada enemigo puede ser configurado independientemente de los otros para otorgar variedad al jugador. Además, los enemigos se hacen más fuertes de acuerdo al nivel del héroe. Si el héroe consigue derrotar a su rival, ganará experiencia que le servirá para aumentar de nivel. Si por el contrario pierde, se cargará la partida desde el último punto guardado. Siempre, si prevé su derrota, el jugador puede optar por huir de la batalla, quedándose con la vida que posee en el momento de huir pero no obteniendo ninguna recompensa. Para recuperar la salud perdida el jugador puede subir de nivel o acudir a ciertos lugares repartidos por el mapa que poseen un aspecto de un *ankh*[11].

[10] Se puede encontrar en el Anexo
[11] Jeroglífico egipcio en forma de cruz cuya extremidad superior es similar a un óvalo. Significa 'vida'.

En cuanto a la secuencia de pantallas, se ha tratado de conseguir un diseño consistente y flexible, de forma que se puedan añadir nuevos pasos a la secuencia. Un diagrama de flujo puede observarse en el Diagrama 4, mostrando las diferentes transiciones posibles entre las pantallas. Se puede observar cómo no es posible crear un personaje y ponerse a jugar si únicamente se ha seleccionado el *job* y no se le ha puesto nombre, o cómo la única forma de volver al menú principal desde el juego es abrir el menú *in-game* estando en *Game* (explorando el mundo, sin estar en combate) y pulsar en *Exit.*

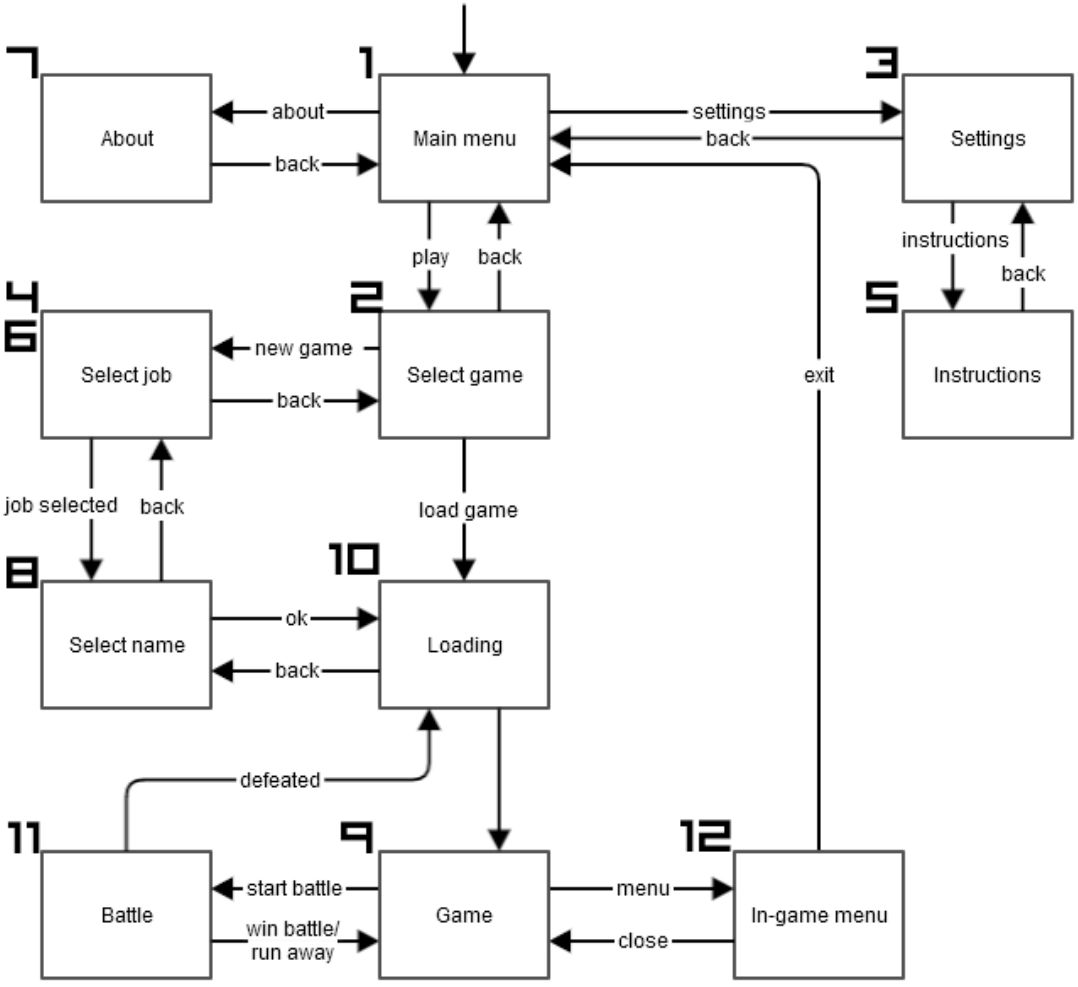


Diagrama 4. Diagrama de flujo.

Una muestra de cada pantalla del diagrama de flujo puede encontrarse en la Ilustración 4. En estas podemos destacar el botón que encontramos en las capturas 2, 3, 4, 7 y 8 en la esquina superior derecha, que sirve para volver a la anterior pantalla y el botón “>” que podemos ver en la captura 2 que tiene dos funciones diferentes dependiendo si en ese ranura hay ya una partida, en cuyo caso se carga, o si no la hay, en cuyo caso se pasa al proceso de creación de un nuevo personaje, como se indica en el texto informativo.

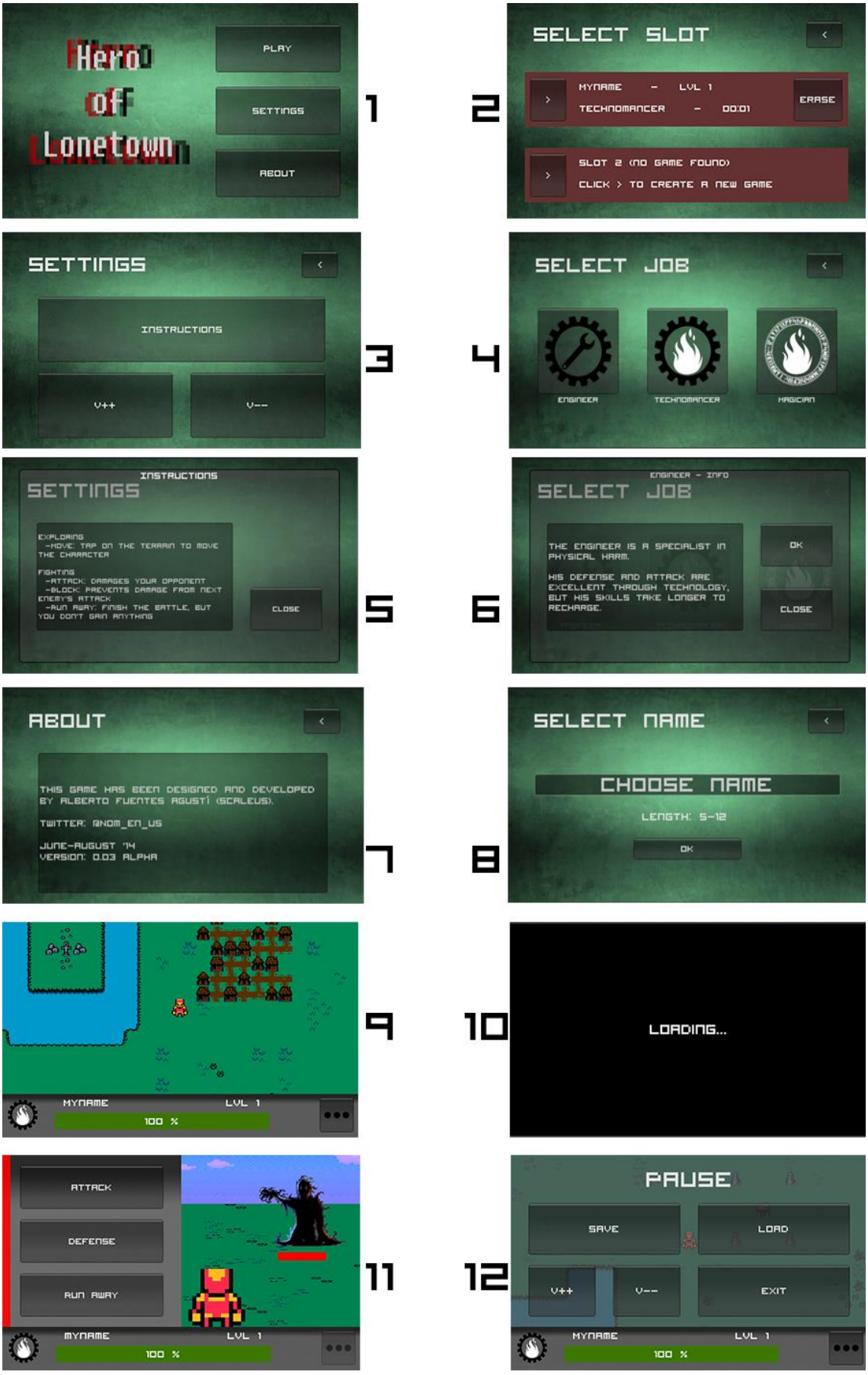


Ilustración 4. Capturas de las diferentes pantallas.

5.2. Aspecto estructural

Las clases empleadas en el proyecto se pueden dividir en tres grupos: los *managers*, clases básicas y clases de interfaz de usuario (GUI, *graphic user interface* en inglés).

Los *managers* son clases encargadas de gestionar a grupos de objetos que comparten una clase básica como por ejemplo los enemigos, facilitando el acceso a todos a través de éste y permitiendo la aplicación directa de efectos a todos a la vez. Esto se consigue, normalmente, añadiendo todos los elementos de un mismo tipo es una estructura, de forma que cuando se desea aplicar a todos los elementos una función o modificarlo de alguna forma, se itera a través de esta aplicándolo uno a uno, sin necesidad de buscarlos por todo el código. También controlan los aspectos transversales del juego, como puede ser la música, el acceso a los datos persistentes, la entrada del usuario, etc. Para centralizar su acceso y pudiendo realizar cambios sin tener que modificar gran parte del código.

Las clases básicas son aquellas que contienen el comportamiento de los objetos o sus atributos. Son muy variadas, desde las clases que contienen la inteligencia artificial de cada personaje independiente hasta los que controlan las animaciones por código, cambiando los *sprites*.

En cuanto a los *scripts* de UI, son aquellos que se encargan de dibujar cada elemento de interfaz en la pantalla, gestionando su entrada[12] y cambiando para adaptarse a ella.

5.3. Aspecto gráfico

Este es, cara al público, un aspecto realmente importante. Es el aspecto que la gente ve y que puede modificar totalmente la percepción que tiene el jugador del juego.

La interfaz es parte importante de un juego. Si es sencilla y consistente facilita la interacción del jugador con la aplicación [12]. El diseño de la interfaz se ha realizado tratando de seguir las reglas de oro dictadas por Ben Shneiderman [1]:

1. *Consistencia en el diseño.* La interfaz mantiene una estructura lo más similar posible entre ella, para evitar confundir al usuario cambiando los botones de lugar si su función es similar. Un ejemplo de esto se encuentra en los menús de creación de personaje con el botón de *volver atrás* colocado en la misma posición. Los botones de volumen también los encontramos situados en el mismo orden.

[12] La entrada a la UI no la controla el CInputManager.

2. *Posibilidad de usar tajos.* Debido a que el juego es muy sencillo en este aspecto, no se ha visto necesario crear ningún atajo.
3. *Ofrecer feedback*[13]. Todos los botones cambian de aspecto al ser pulsado y al volverse inactivos. También, cuando los enemigos atacan o son atacados, se ofrece un estímulo visual para indicar que algo ha sucedido, así como cuando ganas. La música también ayuda a situarse en el juego.
4. *Acciones secuenciales.* Un ejemplo de esto es la secuencia de creación del personaje. El usuario va pasando por los diferentes procesos de configuración de su personaje, y se ha tratado de que siempre quede claro el paso actual y las acciones a realizar.
5. *Gestión de errores sencilla.* Se ha tratado, sobretodo, evitar errores de usuario. Esta es una decisión que limita el control que el usuario tiene de la aplicación, pero puede facilitar el uso de ésta a los usuarios poco experimentados. Un ejemplo es el de que no se permite crear un personaje de una longitud determinada ocultado el botón que permite avanzar en el proceso de creación.
6. *Reversión de las acciones.* En esta aplicación este era un aspecto que no presentaba mucha importancia ni utilidad. El único ejemplo es el que podemos encontrar a la hora de crear un personaje: si alguna decisión de personalización no le gusta al usuario, siempre puede volver atrás y cambiarla antes de que lo cree.
7. *Dar el control al usuario.* Como se ha dicho en el punto de la gestión de errores, en ciertos aspectos se ha dado preferencia a la seguridad ante el control por parte del usuario, pero eso no siempre es así. Algunos ejemplos son que el jugador siempre pueda huir de la batalla, ya que se ha decidido que no pueda abrir el menú dentro de ella. Que el usuario pueda volver atrás en el proceso de creación del personaje podría ser otro ejemplo.
8. *Reducir la carga en la memoria del usuario.* Debido a que los menús son sencillos, este es un aspecto que se ha presentado sin prestarle demasiada atención. Pese a eso, sí que hay un ejemplo en el que se siguió este punto voluntariamente. En la secuencia de creación del personaje se indica en todo momento en qué parte de proceso está el usuario.

[13] R.A.E. "Feed-back": (voz i.) m. Retroalimentación, conjunto de reacciones o respuestas que manifiesta un receptor respecto a la actuación del emisor, lo que es tenido en cuenta por este para cambiar o modificar su mensaje:

Otra parte importante del diseño gráfico de un videojuego son todos los gráficos que poseerán los personajes, el mapa y los elementos contenidos en éste. Debido a que el proyecto no se centra completamente en este aspecto, la aplicación resultante ha pasado por alto algunos elementos deseables y se ha centrado en la interfaz. Esto mismo causó que se usaran gráficos ya creados de varias fuentes. Se buscó que las texturas fuesen de baja resolución (*pixeladas*) para otorgar un efecto *retro*.

Queda por comentar que algunos recursos empleados han sido realizados por varios artistas distintos. Tanto los *sprites* de los personajes, el *tileset*[14] empleado para crear el mapa y la música han sido obtenidos de diversas fuentes, las cuales se encuentran especificadas en el punto segundo del Anexo. Aun así, algunos elementos como los iconos de cada *job* y los demás dibujos se han realizado dentro del ámbito de este trabajo.

6. Implementación

La implementación ha sido uno de los procesos más costosos, pero a la vez uno de los que más cosas nos han enseñado. Hemos reforzado nuestro conocimiento de C# y de Unity, y hemos podido seguir la creación de una aplicación desde la nada hasta un videojuego jugable.

Esta fase ha sido la más costosa y duradera debido a las diversos cambios en el diseño que surgieron a lo largo del desarrollo. Estos cambios se debieron principalmente a la experiencia en el campo de la creación de videojuegos adquirida en un entorno laboral durante el entorno estival, que tuvo lugar a la vez que este proyecto avanzaba.

El resultado de la implementación han sido 24 clases que podemos clasificar en cinco grupos: *managers*, personajes, interfaz gráfica, estructuras de datos y utilidades.

6.1. Managers

Estas clases formarán la columna vertebral del proyecto. Son las encargadas de la gestión y el control de otras clases y de aspectos generales como el audio o la persistencia de datos. Una ventaja importante es que permiten escalar el proyecto fácilmente, cosa realmente importante en un videojuego ya que siempre es deseable poder ampliarlo y añadirle nuevas características.

[14] Se trata de un mapa de imágenes que se emplea para crear escenarios o imágenes más grandes. Suelen representar objetos o espacios tipo, de forma que si se juntan de forma lógica dibujen un mapa o una zona deseada. Es muy empelado en juegos con gráficos de 8 bits.

CSoundManager maneja la música del juego. Puede pausar, reanudar, parar, aumentar el volumen de la música y reproducirla. En la actualidad hay tres músicas diferentes, la del menú, la de batalla y la del juego cuando no estás en un combate. En la Figura 3 podemos observar el código que hace que se reproduzca el sonido especificado por el argumento.

```
public void PlayMusic(SOUND clip) {

    if (gameObject.transform.GetComponent<AudioSource> () != null) {
        if(gameObject.transform.GetComponent<AudioSource> ().clip == _acSounds[(int)clip])
            return;
        DestroyImmediate(gameObject.transform.GetComponent<AudioSource> ());
    }

    AudioSource aSource = (AudioSource)gameObject.AddComponent<AudioSource>();
    aSource.clip = _acSounds[(int)clip];
    aSource.volume = PlayerPrefs.GetFloat ("volume");
    aSource.tag = "sound";
    aSource.loop = true;
    aSource.Play ();
}
```

Figura 3. Fragmento de código de CSoundManager.

Este fragmento gestiona el audio de forma que cuando esta función es invocada, se encarga de comprobar si ya hay alguna música sonando. Si es así, comprueba si es la misma que se quiere hacer sonar, en cuyo caso se deja como está para que siga sonando, pero si la que suena es distinta, se destruye el componente actual para crear el nuevo componente de audio con la nueva música.

CDataManager se encarga de gestionar los aspectos persistentes del juego: cargar, guardar, comprobar ficheros de guardado y la lectura del fichero XML que configura a los enemigos. La persistencia de los datos de la partida se consigue encapsulando en un archivo binario la información del jugador mediante las clases *BinaryFormatter* y *FileStream*, de las librerías de C# (véase Figura 4). La información a almacenar se estructura en una clase que se comenta más adelante, en el punto 6.2.1 de este trabajo.

```
public static void Save(string fileName, PlayerData data)
{
    BinaryFormatter bf = new BinaryFormatter();
    FileStream file = File.Create(Application.persistentDataPath + "/" + fileName);

    PlayerData toSave = new PlayerData();
    toSave = data;

    bf.Serialize(file, toSave);
    file.Close();
}

public static PlayerData Load(string fileName)
{
    if (Exists(fileName))
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = File.Open(Application.persistentDataPath + "/" + fileName, FileMode.Open);
        PlayerData data = (PlayerData)bf.Deserialize(file);
        file.Close();

        return data;
    }

    return null;
}
```

Figura 4. Fragmento de código de CDataManager.

En este código se puede observar cómo se realiza el proceso de guardado y cargado de una partida. Este código está inspirado en la ayuda que ofrece Unity en su documentación. El objeto *BinaryFormatter* permite la conversión de una estructura de datos a un formato que pueda ser almacenado, mientras que *FileStream* se encarga de gestionar el fichero donde se escribirán o leerán estos datos ya convertidos.

El *CInputManager* gestiona la entrada del usuario. De acuerdo a la sensibilidad especificada, actualiza un vector posición que indicará el punto de contacto del dedo con la pantalla, donde el personaje deberá dirigirse (véase Figura 5).

La sensibilidad se emplea para evitar que el personaje se mueva a posiciones no deseadas producto de a falta de precisión que puede surgir al utilizar los dedos en la pantalla táctil.

```
void Update () {

    if(Input.GetMouseButton(0)){
        _vTouchPos = Input.mousePosition;

        if(_vTouchPos.y > (CHud.hudHeight * Screen.height) && checkSensibility()){

            _vAux.z = -1f;
            _vAux.x = _vTouchPos.x - midX;
            _vAux.y = _vTouchPos.y - midY;
            _vAux.Normalize();

            _vDir = _vAux;

        }else{
            _vDir = Vector3.zero;       //  Si no se esta pulsando en la zona valida
        }
    }else{
        _vDir = Vector3.zero;           //  Si no se esta pulsando
    }
}


private bool checkSensibility(){
    if ((_vTouchPos.x < midX - (sensibility * Screen.width)) || (_vTouchPos.x > midX + (sensibility * Screen.width)))
        return true;
    if((_vTouchPos.y < midY - (sensibility * Screen.height)) || (_vTouchPos.y > midY + (sensibility * Screen.height)))
        return true;
    return false;
}
```

Figura 5. Fragmento de código de CInputManager.

Como se puede observar, se recoge la posición de contacto con la pantalla y se comprueba si está fuera de la posición de sensibilidad, pues de no ser así el vector resultante se iguala al vector cero[15] para indicar que no hay movimiento. También se comprueba que se haya pulsado por encima de *CHud.hudHeightK*, una variable estática que indica la altura de la barra de interfaz inferior. En caso de pulsar en una zona válida, la componente Z del vector se iguala a -1 para mantener siempre el personaje a la misma profundidad y las otras componentes se obtienen de la entrada de la pantalla, tras lo cual se normaliza y se le asigna al vector principal, que será el que las demás clases leerán. La razón de realizar todo el proceso con un vector auxiliar es evitar situaciones no deseadas en la lectura del vector, pues si se leyese mientras se está actualizando podría dar resultados erróneos.

El CLevelManager es el principal encargado de manejar el flujo de los acontecimientos una vez se está en la escena de juego. Emplea los estados del héroe para controlar las acciones que se deben realizar, es decir, utiliza una máquina de estados con los estados del héroe para gestionar la aparición de los enemigos/batallas, la salida del combate, ya sea por victoria o derrota, y el cambio al siguiente estado.

Mientras el jugador está quieto, la máquina de estados espera. Cuando el jugador se mueve, pone en marcha una cuenta atrás (el tiempo es obtenido aleatoriamente entre un rango) que únicamente cuenta el tiempo que el jugador está moviéndose. Cuando este contador llega a cero, calcula el siguiente tiempo

[15] Se trata de un vector cuyas componentes son todas iguales a 0

en el que aparecerá el enemigo y prepara el actual combate. Tal preparación consiste en activar la cámara que renderiza el espacio del combate y desactivar la principal, habilitar el menú de combate, elegir un enemigo aleatorio y activarlo y poner al héroe en estado de espera en el combate, dando así comienzo a éste.

CGameManager tiene tres funciones en el juego. Contiene información básica que se encuentra centralizada para poder cambiarla, de ser necesario, de un solo punto. También maneja la información en el cambio de escenas, ya que, cuando esto sucede, muchos objetos se borran, pero debido a que esta clase persiste en toda la ejecución es una clase válida para gestionar el cambio. Por último y por el mismo motivo anteriormente mencionado, controla las acciones a ejecutar en cada cambio de escena, como puede ser la aparición del personaje cuando empieza el juego, la vuelta al menú, los cambios de música, etc.

CEnemyManager es muy sencilla debido a que en estos momentos no se necesitaba gestionar más de un enemigo a la vez, pero se ha creado para poder implementar un combate contra varios enemigos a la vez.

```
public enum ENEMY_TYPE
{
    TYPE_DEMIGOD, TYPE_MUTANT, TYPE_SHADOW,
    TYPE_MAX
};

public enum ENEMY_STATE
{
    ST_NULL, ST_WAITING, ST_ATTACKING, ST_DIYING, ST_RELEASE
};
```

Figura 6. Fragmento de código de CEnemyManager.

En esta versión únicamente contiene los estados y los tipos de los enemigos (véase Figura 6). Al ser enemigos relativamente sencillos, constan de pocos estados. Un estado inicial, otro que indica que está esperando, otro para cuando ataca, otro más para cuando muere y un último estado para cuando ya ha muerto.

6.2. Personajes

6.2.1. Héroe

La clase *CHero* es la clase principal del personaje principal (héroe), y controla el comportamiento de éste, lo que es capaz de hacer en cada momento y en qué estado se encuentra. También gestiona sus atributos, en especial la vida y el nivel.

Se basa en el modelo de máquina de estados (véase Diagrama 2), controlando las acciones que se realizan entre cambios de estado y al mantenerse en un estado. Este modelo permite manejar limpiamente qué se hace en cada situación y facilita incluir nuevas acciones. Los estados controlan lo que el héroe puede o no puede hacer, como se puede ver en la Figura 7 y 8.

```
void _updateWaiting(){
    fTimeBetweenAttacks += (Time.deltaTime * _pdHero.recharge);

    if(fTimeBetweenAttacks < ATTACK_TIME)       return;

    setState (HERO_STATES.STATE_ATTACK);
}
```

Figura 7. Fragmento de código de CHero

```
void Update () {
    _pdHero.totalTime = (int)Time.time;

    if(_bPaused)    return;             //  Si esta pausado, no hacer nada


    switch(state){
        case HERO_STATES.STATE_NULL:
            break;
        case HERO_STATES.STATE_STANDING:
        case HERO_STATES.STATE_EXPLORING:
            _updateMovement ();
            break;
        case HERO_STATES.STATE_ATTACK:
            break;
        case HERO_STATES.STATE_WAITING:
            _updateWaiting();
            break;
        case HERO_STATES.STATE_RESTING:
            break;
        case HERO_STATES.STATE_BUSY:
            break;
    }

}

void _updateMovement(){
    _vDir = _hInputManager.getDir ();

    if (_vDir == Vector3.zero) {                //  Si he entrado aqui y no anda, esta quieto
        if(state != HERO_STATES.STATE_STANDING)
            setState(HERO_STATES.STATE_STANDING);
        return;
    }

    if (state != HERO_STATES.STATE_EXPLORING) { //  Si estaba quieto pero ahora su vDir no es 0, anda
        setState(HERO_STATES.STATE_EXPLORING);
    }

    transform.position += _vDir * speed * Time.deltaTime;
    _pdHero.positionX = transform.position.x;
    _pdHero.positionY = transform.position.y;
}
```

Figura 8. Fragmento de código de CHero

Otra clase relacionada con el héroe es *PlayerData*. Se trata de una clase *serializable* [16] empleada para almacenar los atributos del héroe y alguna información más, como el tiempo de juego. Además de ser utilizada para el acceso dentro del juego, como un *struct* típico, también se emplea para almacenar la información de forma persistente.

Figura 9. Diferentes sprites del héroe que van alternándose

Las animaciones del héroe las controla *RunAnimations*. Este *script* cambia el *sprite* [17] de acuerdo a ciertos parámetros: la *frecuencia de actualización*, que indica la velocidad de cambio entre un conjunto de *sprites* (véase Figura 9). Otro parámetro es el estado del héroe, que indica si debe estar alternando sus *sprites* (está moviéndose) o no (está quieto). Esta clase emplea el vector devuelto por *CInputManager* para saber qué grupo de *sprites* debe estar alterando si están activos o no sus miembros.

6.2.2. Enemigos

CEnemyBasic es la clase que modela el comportamiento de los enemigos que encontramos en el juego. Emplea una máquina de estados para controlar qué debe hacer en cada momento, que se encuentra en la clase *CEnemyManager* (véase Figura 6), ya que la emplearán futuros enemigos con diferentes *scripts* de comportamiento, como los jefes. Esta clase también gestiona las animaciones que se reproducen al recibir daño, así como también prepara al enemigo al comienzo de la batalla, lo escala para igualar su fuerza a la del héroe y gestiona los ataques y el daño recibido.

Como se puede ver en las Figuras 10 y 11, el comportamiento básico del enemigo es el de esperar hasta que pueda atacar y, entonces, atacar. Tras atacar, se vuelve a poner en espera. Este comportamiento será cíclico hasta que la batalla termine. Esta estructura nos permite añadir fácilmente un nuevo estado o modificar los actuales para cambiar el comportamiento del enemigo,

[16] Al marcar una clase como *serializable*, el compilador almacena todas sus variables en un espacio continuo de memoria, permitiendo que se pueda volcar en un fichero directamente.
[17] Aunque en su origen se trataba de un tipo de mapa de bits especial para reducir la carga del procesador, en la actualidad se emplea para referirse a un dibujo pixelado.

```
void Update () {
    if(_bPaused)    return;             //  Si esta pausado, no hacer nada


    switch(state){
        case CEnemyManager.ENEMY_STATE.ST_NULL:
            break;
        case CEnemyManager.ENEMY_STATE.ST_WAITING:
            _uploadWaiting();
            break;
        case CEnemyManager.ENEMY_STATE.ST_ATTACKING:
            break;
        case CEnemyManager.ENEMY_STATE.ST_DIYING:
            break;
        case CEnemyManager.ENEMY_STATE.ST_RELEASE:
            break;
    }

}

private void _uploadWaiting(){
    _fTimeBetweenAttacks += (Time.deltaTime * enemyStats.recharge);

    if(_fTimeBetweenAttacks < ATTACK_TIME)      return;

    setState(CEnemyManager.ENEMY_STATE.ST_ATTACKING);
}
```

Figura 10. Fragmento de código de CEnemyBasic.

```
public void setState(CEnemyManager.ENEMY_STATE st){
    state = st;

    switch(st){
        case CEnemyManager.ENEMY_STATE.ST_NULL:
            break;
        case CEnemyManager.ENEMY_STATE.ST_WAITING:
            _wait();
            break;
        case CEnemyManager.ENEMY_STATE.ST_ATTACKING:
            _attack();
            break;
        case CEnemyManager.ENEMY_STATE.ST_DIYING:
            break;
        case CEnemyManager.ENEMY_STATE.ST_RELEASE:
            break;
    }
}

private void _wait(){

    _fTimeBetweenAttacks = 0;
}

private void _attack(){

    _hHero.addLife (-enemyStats.damage);

    setState (CEnemyManager.ENEMY_STATE.ST_WAITING);
}
```

Figura 11. Fragmento de código de CEnemyBasic

6.3. Interfaz gráfica

La implementación de la interfaz ha sido realizada enteramente mediante código. Cada escena tiene su propio *script* encargado de dibujarla, y dentro del juego podemos encontrar varios, para los diferentes elementos.

En las escenas del menú principal, los menús de creación de personajes, el menú de opciones y el de información[18] cuentan con unos *scripts* muy similares, cambiando el número y la posición de los elementos y añadiendo o eliminando alguno. Estos *scripts* siempre están activos, pues el menú siempre debe estar activo. Un ejemplo se puede observar en la Figura 12.

```
void OnGUI () {
    GUI.skin.font = CGameManager.generalFont;
    GUI.skin.button.fontSize = (int)((float)fontSize * (Screen.height / CGameManager.fontResize));


    if (Input.GetKey (KeyCode.Escape)) {
        Application.Quit();
    }

    if (GUI.Button (new Rect (Screen.width * 0.6f, Screen.height * 0.10f, Screen.width * 0.35f, Screen.height * 0.22f), "Play")) {
        Application.LoadLevel("PlayScene");
    }
    if (GUI.Button (new Rect (Screen.width * 0.6f, Screen.height * 0.39f, Screen.width * 0.35f, Screen.height * 0.22f), "Settings")) {
        Application.LoadLevel("SettingsScene");
    }
    if (GUI.Button (new Rect (Screen.width * 0.6f, Screen.height * 0.68f, Screen.width * 0.35f, Screen.height * 0.22f), "About")) {
        Application.LoadLevel("AboutScene");
    }
}
```

Figura 12. Fragmento de código de GUIMainMenu.

Por otra parte podemos encontrar la interfaz una vez empezada la partida. Para controlar la barra inferior que encontramos al entrar en la partida (véase Figura 13) empleamos la clase *CHud*. Aquí se controla la barra de vitalidad del personaje, el nombre, el nivel y el botón que abre el menú *in-game*[19] (véase Figura 14).

Figura 13. Barra de información.

[18] Llamado 'About' en el juego
[19] Referente a lo que ocurre dentro del juego o la partida

```
void OnGUI () {
    if(_hHero != null){
        _fHealthIndex = (float)_hHero.GetHero().healthRemaining / (float)_hHero.GetHero().health;

        refreshLabels ();
        refreshSafeZoneLabel();

        //  Control de la vida
        _goLifeBar.transform.localPosition = CUtils.changeX (_goLifeBar.transform.localPosition, 0.2f + _fHealthIndex * 0.3f );
        _goLifeBar.transform.localScale = new Vector3 (0.6f * _fHealthIndex, _goLifeBar.transform.localScale.y, _goLifeBar.transform.localScale.z);
    }else{
        _hHero = GameObject.Find ("Player").GetComponent<CHero> ();
        _textName.text = _hHero.GetHero().name;
        return;
    }


    GUI.DrawTexture (new Rect (0.02f * Screen.height, 0.815f * Screen.height, 0.16f * Screen.height, 0.16f * Screen.height), jobIcon [_hHero.GetHero ().job - 1]);

    //  Boton de opciones
    GUI.enabled = (!_bMapActive && (_hHero.state == CHero.HERO_STATES.STATE_STANDING) && (_hHero.state != CHero.HERO_STATES.STATE_EXPLORING));

    if(GUI.Button (new Rect (Screen.width - (Screen.height * 0.18f), 0.815f * Screen.height, 0.16f * Screen.height, 0.16f * Screen.height), menuIcon)){
        OpenMenu();
    }
}
```

Figura 14. Fragmento de código de CHud.

Dentro de la partida podemos encontrar otra clase encargada de la interfaz. Esta vez se trata de la clase que maneja el menú de pausa (véase Figura 15). Este menú permite guardar y cargar la partida, así como aumentar y disminuir el volumen de la música y volver al menú principal. Para ello, el botón del menú de la barra de información, al ser pulsado, activa el *script*, de forma que activa el proceso de dibujado de la interfaz del menú. Al volver a pulsar este botón, se desactiva, deteniendo el redibujado y haciendo desaparecer el menú.

Figura 15. Menú de pausa in-game.

La batalla también muestra un nuevo menú, el *menú de batalla* (véase Figura 16). Este menú consta de tres botones. El primero, empezando por arriba, sirve para realizar un ataque, el segundo permite al jugador defenderse del siguiente ataque, y el último termina el combate, así el jugador puede dejar ese combate y continuar explorando el mundo. La barra de recarga, que se encuentra a la izquierda de la pantalla en las batallas, indica si puede o no realizar alguna acción el jugador. Para reforzar esta información, los botones se desconectan

cuando el jugador no puede actuar. También se desactiva el botón de menú, para impedir situaciones no deseadas.

Por último, la vida del enemigo también forma parte de la interfaz (véase Figura 16). En este caso concreto, este elemento de la interfaz lo controla cada enemigo.

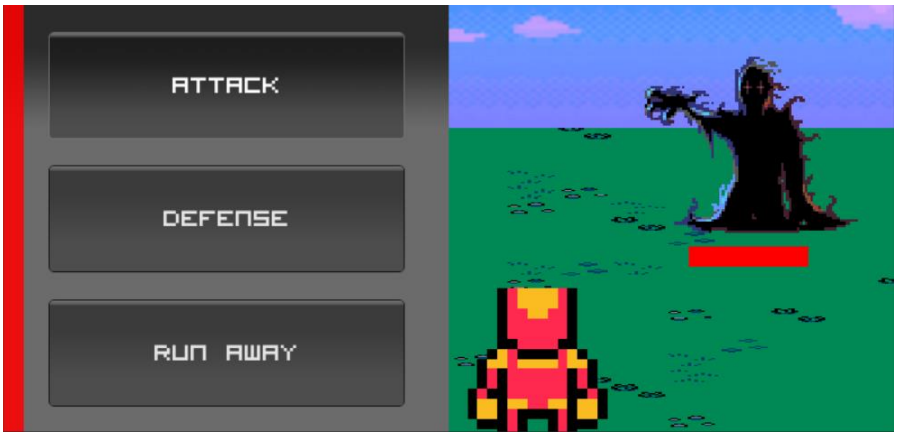


Figura 16. Batalla.

6.4. Estructuras de datos

La información debe estar organizada en estructuras de forma que faciliten los accesos y su modificación. En este caso encontramos principalmente tres estructuras.

La primera estructura es el fichero XML encargado de la configuración de los enemigos. Especifica la vitalidad, el daño de ataque, su defensa, la frecuencia de ataque y la experiencia que otorga cada uno (véase Figura 17). Este fichero se puede editar sin tocar nada del código, facilitando así su edición por parte de personas no acostumbradas a programar. Se analiza este fichero (comúnmente llamado *parsear*) mediante la clase *CDataManager* y se traspasa su contenido a un *struct*[20], evitando tener que acceder continuamente a un fichero externo, evitando así costes innecesarios.

[20] En C#, es un tipo empleado para encapsular un grupo de variables relacionadas.

```
<enemies>
    <enemy id="enemyDemigod">
        <life>20</life>
        <damage>7</damage>
        <recharge>0.5</recharge>
        <defense>0.9</defense>
        <experience>40</experience>
    </enemy>
    <enemy id="enemyMutant">
        <life>20</life>
        <damage>7</damage>
        <recharge>0.5</recharge>
        <defense>0.9</defense>
        <experience>20</experience>
    </enemy>
    <enemy id="enemyShadow">
        <life>20</life>
        <damage>7</damage>
        <recharge>0.5</recharge>
        <defense>0.9</defense>
        <experience>25</experience>
    </enemy>
</enemies>
```

Figura 17. Fichero de configuración de los enemigos.

Como se puede observar y se ha introducido anteriormente, el fichero XML posee los campos necesarios para configurar a los enemigos. Para esto, los enemigos están rodeados por la etiqueta *<enemies>*, dentro de la cual se encuentran varias entradas con la misma etiqueta *<enemy>* pero con diferente *id*. Este *id* es el que identifica a cada enemigo, y al cuál corresponderán los atributos que se encuentran dentro de esta etiqueta. Se les ha puesto nombres directos para facilitar su edición.

En este caso sólo es necesario realizar dos aclaraciones. La primera es que el atributo *recharge* indica cuántos segundos transcurren por cada segundo real, por lo que si un enemigo tiene un *recharge* de 0.5, quiere decir que cada segundo real transcurren 0.5 segundos en su contador. Esto se utiliza porque todos los personajes atacan cada 1 segundo personal que se calcula multiplicando el tiempo real transcurrido por este atributo, por lo que el personaje del ejemplo atacará cada 2 segundos reales (si 1 segundo real son 0.5 segundos personales, para lograr 1 segundo personal deben transcurrir 2 segundos reales). La segunda aclaración es que la defensa sirve para indicar qué daño sufrirá el agredido de forma que el atributo es un multiplicador del daño por lo que cuanto menor sea menos daño recibirá.

6.5. Utilidades

Para facilitar el funcionamiento de los demás *scripts*, se decidió implementar unas clases que servirían para resolver problemas muy puntuales de forma similar a las API de cualquier lenguaje.

La primera clase creada de esta categoría es *CUtils*, que incluye diferentes funciones como la de cambiar un único elemento de un vector sin tener que pasar por escribir un código de varias líneas, haciéndolo sólo aquí. Se observó que esto era una tarea muy recurrente y que era susceptible de ser implementada en una clase que sirviese como contenedor de herramientas.

Por otra parte, se ha programado una pequeña clase con funciones que permiten testear ciertos aspectos del juego rápidamente. Actualmente permite aumentar y disminuir la vida del héroe para probar ciertos elementos relacionados con ésta. Por otra parte, permite implementar nuevas funciones de testeo fácilmente.

6.6. Mejoras de optimización

Existe la creencia de que nos dispositivos de hoy en día son tan potentes que la optimización no es necesaria. Si bien es cierto en parte, y dependiendo de la aplicación, que ya no es tan necesario como antes este proceso, realizarlo puede mejorar la fluidez y evitar que el dispositivo se caliente innecesariamente. En este proyecto se han aplicado algunas características que proporciona Unity para tratar de mejorar el rendimiento.

Usualmente se debe escoger entre optimizar el uso del procesador cargando más la memoria o liberar memoria a costa del procesador. En este caso se ha decidido cargar la memoria debido a que la carga no es grande. Se ha logrado gracias a la característica de Unity que nos permite desactivar objetos, lo que permite que sigan existiendo en memoria, con lo que no deberán ser creados y evitando la consecuente carga del procesador, pero no interactúan en la partida. El ejemplo más notorio de esto se encuentra en el combate: todos los elementos del combate (excepto la interfaz, que va por código y no puede estar precargada), desde la cámara hasta los enemigos pasando por el fondo, están ya cargados en memoria desde el inicio de la partida, pero desactivado. Cuando se les necesita, se activan los elementos solicitados y se ponen en funcionamiento.

Por supuesto, cuando la cámara que renderiza el combate se activa, la cámara principal se desactiva para evitar que renderice en vano.

7. Pruebas realizadas

Para comprobar el correcto funcionamiento de la aplicación se han empleado cuatro dispositivos con diferentes características: dos ordenadores y dos móviles.

7.1. Ordenadores

Las pruebas en los ordenadores se realizaron para comprobar la capacidad *cross-platform* que anunciaba Unity, además de que resultaba más rápido realizar las pruebas en el mismo sistema en el que se estaba programando que tener que crear el *apk*[21] y pasarlo a un dispositivo externo.

Como ya se ha comentado, dispusimos de dos computadores diferentes:

Ordenador 1:

SO:	*MS Windows 8.1 64 bits*
Procesador:	*AMD A10-5750M a 2,50 GHz*
Gráfica:	*AMD Radeon HD 8970M*
RAM:	*8,00 GB*

Ordenador 2:

SO:	*MS Windows 7 64 bits*
Procesador:	*Intel Core i5-2400 a 3,10 GHz*
Gráfica:	*AMD Radeon HD 7750*
RAM:	*8,00 GB*

En el *Ordenador 1* se probó tanto el ejecutable generado para Windows (archivos *.exe*) como mediante el propio editor de Unity. En el *Ordenador 2* se ejecutó únicamente el ejecutable generado.

Los resultados fueron muy satisfactorios. Se trataron de evaluar todos los aspectos: los combates, la carga y el guardado, el audio, la entrada de usuario y la interfaz. Todos los aspectos funcionaron correctamente, pero debido a que la interfaz está pensada para funcionar con las resoluciones de los dispositivos móviles, algunas resoluciones, como las que se pueden conseguir mediante el modo ventana, pueden ocasionar pequeños fallos visuales (véase Figura 18). Además, ya que, para ahorrar cómputo los cálculos de reescalado se realizan sólo al inicio de cada *scene*, al cambiar el tamaño de la ventana en el ordenador la interfaz no se corrige. Este es un fallo sin importancia debido a que la aplicación se ejecutará en dispositivos móviles donde este comportamiento no puede tener lugar. Aun así, si se quisiese corregir, se podrían mover los cálculos de la interfaz desde el *Awake()/Start()* hasta el *OnGUI()*.

[21] Paquete de instalación de componentes para un sistema Android

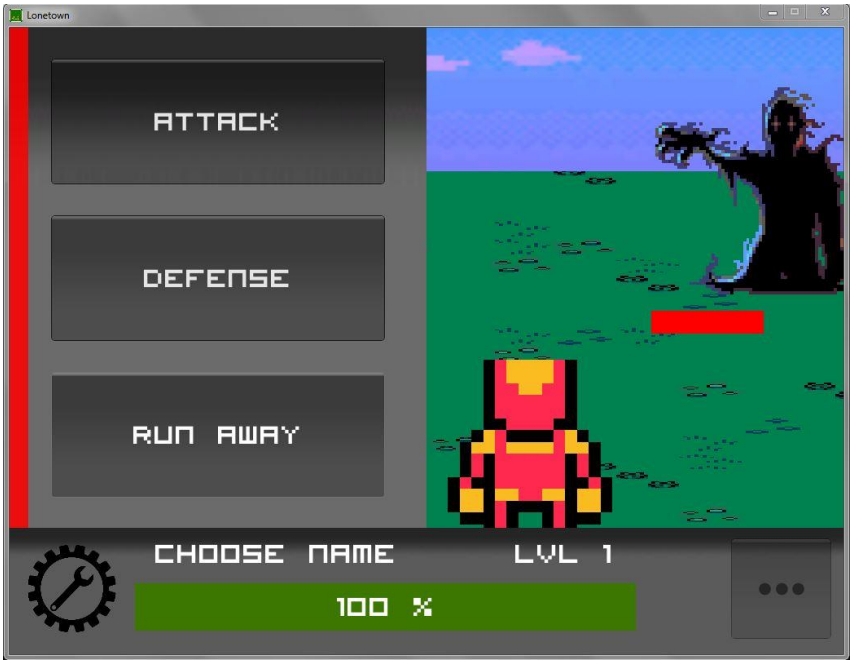


Figura 18. Captura del juego con resolución 1024x768 (4:3).

7.2. Dispositivos móviles

Los dispositivos móviles disponibles para las pruebas fueron:

Samsung Galaxy Mini 2 (GT-S6500D):

SO:	*Android 2.3.6 Gingerbread*
Procesador:	*Qualcomm MSM7227A Snapdragon a 800 MHz*
GPU:	*Adreno 200*
RAM:	*512 MB*

Samsung Galaxy SIII mini (GT-I8190):

SO:	*Android 4.4.4 KitKat*
Procesador:	*Qualcomm Snapdragon 400 a 1,20 GHz*
GPU:	*Adreno 305*
RAM:	*1,00 GB*

Primero se probó en el *Galaxy SIII mini*. En este móvil no pudimos encontrar ningún error ni fallo visual. El juego se ejecuta fluidamente, la música se reproduce correctamente y el sistema de persistencia funciona perfectamente. Tras comprobar que no habían problemas de rendimiento en un móvil de gama media como el anterior se procedió a testear el videojuego en el *Galaxy mini 2*,

un dispositivo de gama baja. En este móvil, al igual que con el anterior, se realizaron diferentes pruebas y todas resultaron satisfactorias.

Habiendo probado el videojuego en estos cuatro dispositivos y a falta de otros con los que testearlos, se concluyó que el juego no presenta problemas de rendimiento. Para asegurarnos de que no presenta problemas con alguna GPU concreta[22] debería hacerse un testeo con cada uno de los diferentes dispositivos, o al menos con la mayoría.

8. Conclusiones

Unity ha supuesto un elemento clave en el desarrollo de este proyecto. La simplicidad, frente a otros motores de juego, es bastante notable. En anteriores proyectos, en los que no se había empleado Unity, el coste de desarrollo fue mucho mayor y el resultado final fue mucho peor. Ha permitido desarrollar una aplicación que funciona perfectamente en dispositivos tan dispares como un ordenador y un *smartphone* Android. Pese a estas ventajas, el proyecto no ha estado exento de problemas pues algunos aspectos no han funcionado como se esperaba y han requerido ajustes posteriores.

8.1. Relación con la carrera

Haber cursado ciertas asignaturas de la carrera ha resultado realmente útil a la hora de afrontar el proyecto, especialmente las centradas en la programación, el diseño y estructuración del código, el diseño de interfaces y las que han tratado el funcionamiento de los videojuegos. A la hora de crear la interfaz, *Interfaces persona-computador* ha resultado especialmente provechosa, pues he podido conocer reglas y consejos a la hora del diseño de las interfaces. *Ingeniería del Software* me ha ayudado a crear una estructura sólida y abierta a cambios, con el código ordenado en las clases adecuadas y con las relaciones adecuadas. Las asignaturas *Introducción a la programación de videojuegos* y *Arquitectura y entornos de desarrollo para videoconsolas* me han acercado al mundo de la creación de videojuegos, mostrándome las diferentes posibilidades y soluciones ante diversos problemas, así como algunas de las dificultades que puedo encontrar y conceptos que debo evitar. Tras estas asignaturas más específicas, también han sido de vital importancia todas aquellas que me han enseñado a programar y a hacerlo de forma eficiente.

[22] Por experiencias personales en otros proyectos, pueden presentarse problemas con GPUs concretas, sin importar la gama de ésta

8.2. Dificultades encontradas

Pese a todo lo anterior, y debido a la falta de experiencia en este tipo de proyectos, han surgido dificultades notables. La principal ha sido la estructuración del código. A lo largo del desarrollo se ha tenido que cambiar el diseño de clases varias veces debido a diferentes problemas que han ido surgiendo y a la experiencia adquirida en la empresa, en la que se ha observado cómo se debía crear un videojuego. Eso podría haberse evitado mejorando la planificación previa, pero sobretodo con más experiencia en este tipo de proyectos. Otro gran problema ha sido el de programar la interfaz. Como se ha dicho en el apartado *Introducción a Unity3D*, este motor presenta una debilidad frente a otros motores en este aspecto. Usualmente se suele cubrir con el uso de *plugins*, pero se decidió no emplearlos para entender mejor el funcionamiento de Unity. La decisión de escoger Android también ha acarreado dificultades: está presente en una gran variedad de dispositivos con una gran variedad de resoluciones diferentes y *aspect ratio*[23].

8.3. Resultados

Aun así, el resultado ha sido bastante satisfactorio (en el tercer punto del Anexo se pueden encontrar algunas capturas del resultado final). Personalmente, he aprendido otro lenguaje, C#, que me ha resultado bastante cómodo y potente. También he profundizado en cómo debe funcionar un videojuego internamente, y este conocimiento me ayudará a atajar problemas en futuros proyectos. He observado la importancia de una buena planificación inicial, creando una estructura consistente pero flexible a la vez, abierta a pequeños cambios de diseño.

Por último me gustaría recomendar el uso de Unity a cualquiera que desee crear un videojuego sin disponer de los recursos de una gran empresa, pues permite crear desde juego sencillo hasta verdaderas maravillas, aunque no alcance la potencia de otros motores como *Unreal Engine* o *CryEngine*.

9. Trabajo futuro

El juego se diseñó para ser mucho más grande, pero debido a la falta de tiempo se han tenido que recortar algunos aspectos. La estructura está montada y preparada para dichas implementaciones, que son la inclusión de misiones,

[23] Relación de aspecto, proporción entre anchura y altura de una imagen o pantalla.

mayor número de enemigos, enemigos finales, NPC [24] aliados, objetos que mejoren las características del personaje y el rediseño del modo de combate.

También se quiere mejorar el aspecto de la interfaz, modificando las texturas, y modificar el aspecto del personaje principal.

Si el juego llega a ser completado, se pensará en la opción de ser comercializado y en añadir características susceptibles de ser monetizadas, como la posibilidad de elegir el aspecto del personaje o nuevos niveles.

Por otra parte, la experiencia adquirida en este proyecto me ha permitido plantearme nuevas metas. También me ha ofrecido oportunidades laborales, que aunque al principio retrasaron el proyecto, más tarde permitieron acelerarlo y mejorarlo mucho más de lo que hubiese sido posible sin ellas.

Por lo tanto, y para finalizar, en el futuro debo seguir aprendiendo y mejorando todo lo que nos sea posible para conseguir cada vez mejores productos y mayor conocimiento.

[24] Non-player character, "personaje no jugador", se refiere a aquellos personajes que no maneja ningún jugador externo al propio juego.

Bibliografía

[1] B. Shneiderman, Designing the user interface, Pearson Education India, 2003.

[2] Unity Technologies, «Unity documentation,» [En línea]. Available: http://docs.unity3d.com/ScriptReference/. [Último acceso: 22 agosto 2014].

[3] Microsoft, «Microsoft Developer Network,» [En línea]. Available: http://msdn.microsoft.com/en-us/library/618ayhy6.aspx. [Último acceso: 22 agosto 2014].

[4] R. Engelbert, COCOS2d-x - Beginner's guide, Birmingham: Packt Publishing Ltd., 2013.

[5] Corona Labs, «Corona Docs,» 31 julio 2014. [En línea]. Available: http://docs.coronalabs.com/. [Último acceso: 22 agosto 2014].

[6] Marmalade Technologies Ltd, «Marmalade Documentation,» [En línea]. Available: http://docs.madewithmarmalade.com/display/MD/Marmalade+Documentation. [Último acceso: 22 agosto 2014].

[7] R. van der Meulen y J. Rivera, «Gartner,» 29 octubre 2013. [En línea]. Available: http://www.gartner.com/newsroom/id/2614915. [Último acceso: 23 agosto 2014].

[8] H. Blodget, «Business Insider,» 11 diciembre 2013. [En línea]. Available: http://www.businessinsider.com/chart-number-of-smartphones-tablets-and-pcs-2013-12. [Último acceso: 23 agosto 2014].

[9] T. Norton, Learning C# by Developing Games with Unity 3D Beginner's Guide, Packt Publishing Ltd, 2013.

[10] Carnegie Mellon University, 2004. [En línea]. Available: https://www.andrew.cmu.edu/course/90-754/umlucdfaq.html. [Último acceso: 28 agosto 2014].

[11] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley Professional, 2004.

[12] K. . D. Saunders y J. Novak, Game Development Essentials: Game Interface Design, Cengage Learning, 20012.

[13] Unity Technologies, «Unity3D Public relations,» [En línea]. Available: https://unity3d.com/es/public-relations. [Último acceso: 28 agosto 2014].

[14] . J. Schell, The Art of Game Design: A book of lenses, CRC Press, 2008.

[15] T. Fullerton, Game Design Workshop: A Playcentric Approach to Creating Innovative Games, CRC Press, 2008.

[16] J. W. Murray, C# Game Programming Cookbook for Unity 3D, A K Peters/CRC Press, 2014.

Anexos

Anexo 1

Documento de diseño

Hero of Lonetown

por Alberto Fuentes Agustí

Índice

1. Visión general del proyecto

1.1. Resumen ejecutivo

1.1.1. Concepto general

Se basará en aumentar el nivel del personaje y completar las misiones que se proponen, completando así episodios y salvando a otros personajes.

1.1.2. El gancho

Conseguir un personaje más fuerte y rescatar a más personas será el aliciente para jugar, averiguando poco a poco la historia que hay detrás.

1.1.3. Sinopsis

Tras una terrible guerra civil con magia y armas biológicas, la que había sido la potencia más grande del planeta desapareció junto con parte de los países vecinos, dejando únicamente tierras yermas, habitadas por espantosas y feroces criaturas resultado de los experimentos y las sustancias empleadas en las batallas. Tras varios meses de vagar por las ruinas, uno de los pocos supervivientes humanos, encuentra un pequeño pueblo donde solicitan desesperadamente ayuda. Las bestias atacaban continuamente, raptando y destrozando a quienes encontraban a su paso. El recién llegado acepta a cambio de un lugar donde vivir y recursos. A partir de entonces empezará el resurgimiento del que posiblemente sea el último asentamiento humano del país, Lonetown.

1.1.4. Género

Podría clasificarse como un juego de aventuras y rol por turnos.

1.1.5. Estilo visual

El juego tendrá un estilo 2D, con gráficos sencillos.

1.1.6. Motor y editor

Se empleará Unity configurado para videojuegos 2D y MonoDevelop como IDE.

1.2. Núcleo del gameplay

El jugador podrá:

- Crear y mejorar armas, armaduras y otros objetos con la ayuda de los habitantes del pueblo, suministrándoles los materiales necesarios.

- Buscar bestias para combatirlas, ya sea para completar misiones o para subir de nivel con la experiencia obtenida.

- Modificar los objetos equipados y emplear la experiencia obtenida para mejorar al personaje.

- Navegar por las diferentes zonas que rodean a Lonetown.

1.3. Características del juego

1.3.1. Ambientación

La historia se desarrolla en un tiempo indefinido, dentro país destruido por la guerra. La tecnología parece estar, en algunos aspectos, más avanzada que la actual, pero no se dan detalles exactos. También se puede notar que antes de la guerra había cierta tensión entre la magia y la tecnología. En el pueblo habrá quienes no querrán usar nunca la magia y quienes rechazarán totalmente la tecnología. Todo esto se mezclará con un tono desenfadado e informal.

1.4. Alcance del proyecto

1.4.1. Ubicaciones del juego

La acción se desarrolla en Lonetown, un pequeño pueblo fundado por unos supervivientes de la Gran Autoguerra y en las zonas que lo rodean.

1.4.2. Descripción de misiones y niveles

Las misiones se dividirán en dos tipos: obligatorias (*mainquests*) y secundarias (*sidequests*). Dentro de los dos tipos podremos encontrar las misiones

clasificadas como: de rescate, de matanza de bestias, de búsqueda de objeto o de exploración de zona.

1.4.3. Descripción de NPCs

Podremos encontrar tanto enemigos como aliados como neutrales. Los enemigos los encontraremos en combate, y son variados. Los aliados/neutrales son habitantes del pueblo, que ayudarán al jugador a crear objetos, lo sanarán, le pedirán ayuda (misiones). También será posible hablar con ellos, aunque sólo dirán una frase (para ambientar la historia). Por supuesto también hablarán los personajes al ser rescatados. El personaje neutral será un androide que vive encerrado en un almacén de productos y actuará como comerciante.

1.4.4. Descripción de armas

Las armas se conseguirán creándolas mediante la recolección de piezas. Éstas serán llevadas al NPC correspondiente que se la creará. También pueden ser mejoradas o recicladas para obtener algún componente.

1.4.5. Descripción de clases

El jugador podrá escoger entre 3 clases distintas:

- *Magician*: sus ataques especiales se basan en el control y generación de la electricidad. Es una clase poco resistente pero con un ataque potente.

- *Soldier*: sus ataques especiales se centran en el uso de la tecnología. Muy resistente, pero su ataque es reducido al de las demás clases.

- *Technomancer*: mezcla hechizos mágicos con el uso de la tecnología. Posee un ataque y resistencia equilibrados.

1.4.6. Sistema de comercio

Para evitar situaciones sin sentido, se implementa un sistema de comercio mediante el cual el jugador puede vender objetos a un NPC. El comercio no tendría sentido si tu vida depende de que le des o no un objeto a alguien en lugar de cobrárselo, por lo que se incluye el androide comerciante. Aunque su vida dependa de dar o no un objeto al personaje, lo cobrará pues está programado para hacerlo. Es imposible destruirlo porque está encerrado en su almacén fortificado y sólo él sabe cómo abrirlo.

1.5. Público objetivo

El juego va destinado sobre todo a jugadores casuales, pero también a jugadores regulares. Busca ser jugable en breves espacios de tiempo.

1.6. Plataformas

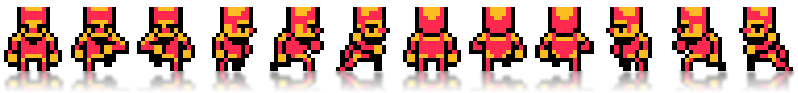
Se lanzará para los móviles Android, pero no se descarta lanzarlo en más plataformas, como iOS o Windows Phone.

2. Diseño conceptual del juego

2.1. Diseño de los personajes

2.1.1. Héroe

El héroe es el personaje principal, controlado por el jugador. Será una de las tres clases descritas arriba, e irá aumentando su nivel y sus atributos (ataque, defensa y cadencia de ataque) de acuerdo a éste.



Será el encargado de completar las misiones, derrotar a los monstruos y rescatar a los ciudadanos secuestrados.

2.1.2. Enemigos

Los enemigos son muy variados, en aspecto y en atributos. Son los rivales del héroe, y aparecen aleatoriamente por el mapa (siempre fuera de la zona segura, la ciudad). Su objetivo es eliminar al héroe. Entre ellos también contamos a los minijefes y jefes, que son enemigos especiales más fuertes que los enemigos normales.

Los atributos de cada enemigo son configurables por el diseñador de niveles mediante un fichero XML, especificando su vitalidad, ataque, defensa, velocidad de acción y experiencia que otorga al ser derrotado.

2.1.3. Aliados

Son los habitantes de Lonetown y gente raptada que el héroe debe rescatar. Hablarán con el héroe, ambientando la historia y dando pistas al jugador. También pedirán la ayuda del héroe, dándole misiones a cambio de una recompensa. Las misiones serán configuradas mediante un fichero XML, así como los diálogos de cada personaje.

2.1.4. Neutrales

El robot de la tienda es el único personaje neutral. Su única función es la de comerciar con el jugador, comprando y vendiendo objetos. Algunos de los objetos requerirán piezas concretas para ser completados y funcionales, evitando así que el jugador los use hasta que el diseñador lo desee.

2.2. Diseño de los objetos

2.2.1. Armas

Las armas son objetos que aumentan el daño que inflige el héroe a los enemigos. Sólo se puede llevar una a la vez, y se cambia en el baúl que posee el héroe en el pueblo. Se pueden crear mediante las piezas que sueltan los enemigos al ser derrotados o comprarse en la tienda del robot que se halla en el pueblo.

2.2.2. Armaduras

Reducen el daño que recibe el héroe. Al igual que las armas, sólo se puede llevar una a la vez, se cambia en el baúl del pueblo y se pueden crear mediante piezas o comprar en el pueblo.

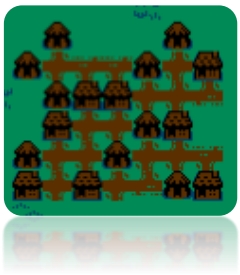
2.2.3. Objetos varios

Otorgan atributos y efectos variados, desde algunos útiles como aumentar la velocidad hasta algunos gráficos como cambiar la skin de los enemigos o la del mapa. Únicamente se obtienen creándolos por piezas (en un futuro podrían ser conseguidos mediante pagos en la aplicación).

2.3. Diseño del mapa

2.3.1. Lonetown

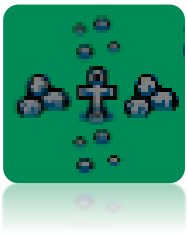
Es la ciudad principal del juego. En ella encontraremos a la mayoría de los personajes neutrales y aliados. En esta zona no podrán aparecer enemigos, por lo que estaremos seguros, mostrando un cartel indicándolo.



Al pisar su superficie, surgirá un botón en la pantalla preguntando al jugador si desea hablar con los habitantes. Si lo pulsa, aparecerán más botones, uno por cada personaje y otro para salir, permitiendo continuar explorando el mundo. También permite almacenar objetos en un baúl.

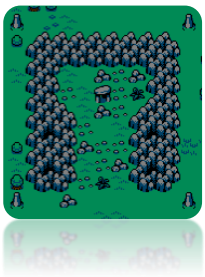
2.3.2. Símbolos de sanación

Podremos encontrarlos por todo el mapa. Su función es la de restaurar completamente la salud del héroe al pasar sobre ellas.



2.3.3. Zonas de jefes y minijefes

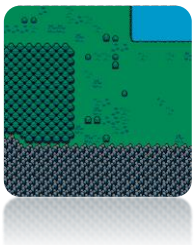
En ellas nos enfrentaremos a un minijefe. Tras enfrentarnos contra todos los minijefes, podremos enfrentarnos con cierta probabilidad de éxito contra el jefe final del mapa actual.



Mientras no se derroten todos los minijefes, el jefe fina del nivel será accesible pero resultará imposible de derrotar, pues derrotará al héroe al primer golpe, y éste no causará ningún daño al enemigo.

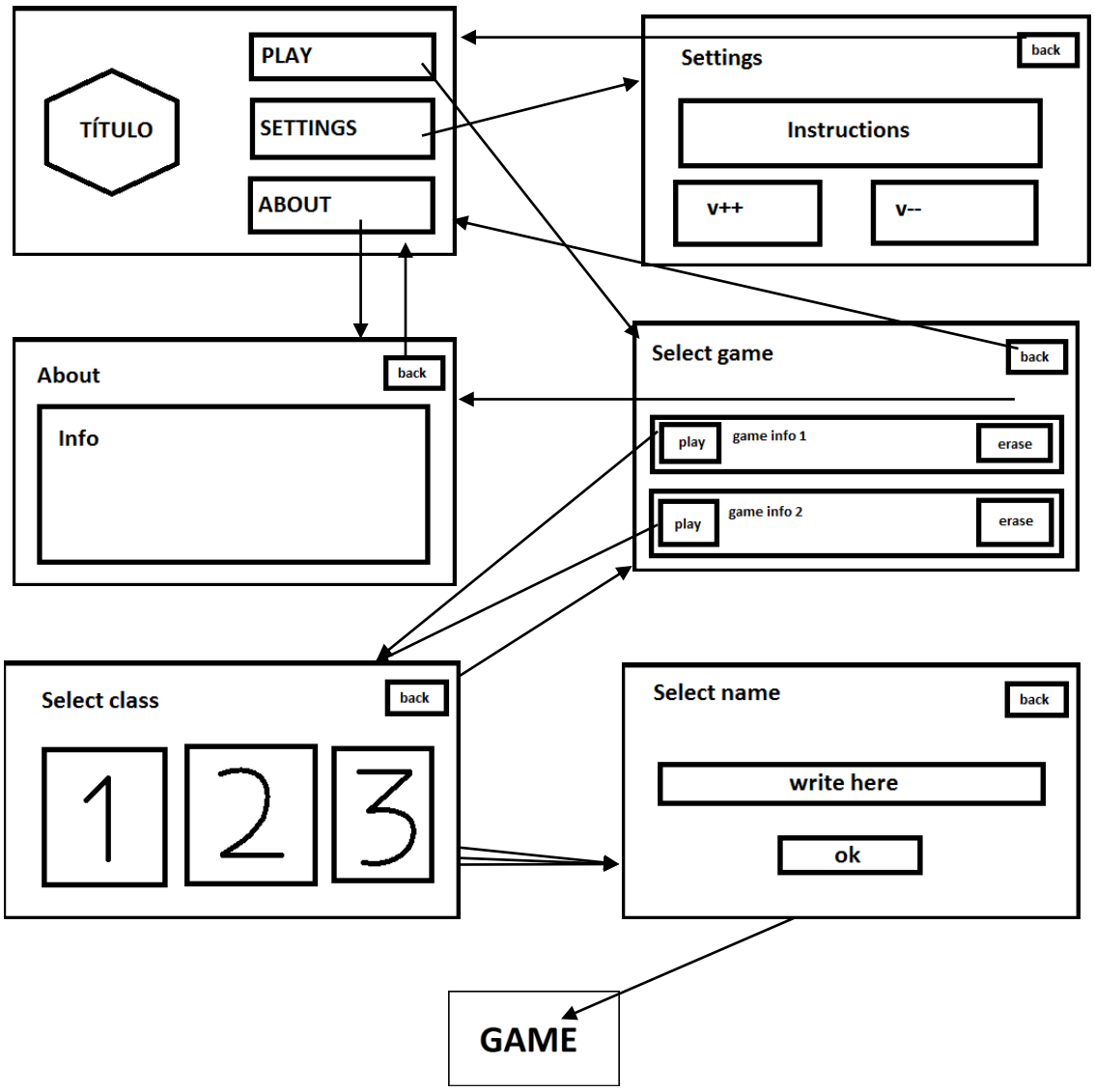
2.3.4. Áreas de bloqueo

Evitan que el héroe se desplace a áreas no deseadas. Son, principalmente, agua, árboles y montañas.

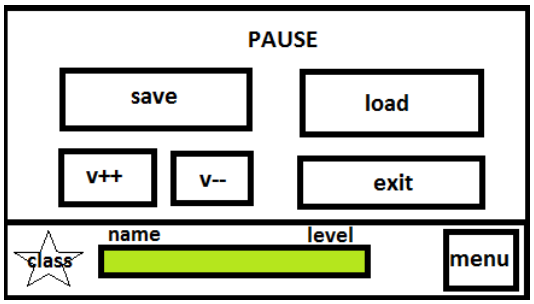


2.4. Diseño de la interfaz

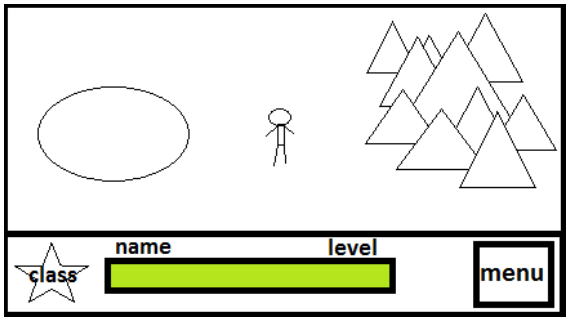
2.4.1. Menús (fuera del juego)



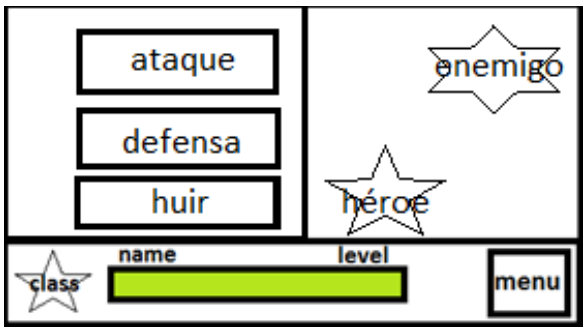
2.4.2. Menús (dentro del juego)



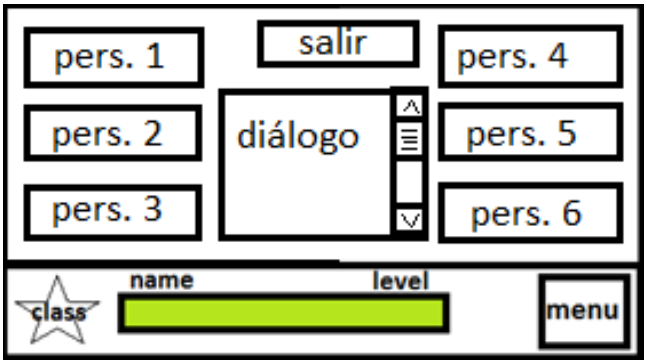
2.4.3. Juego



2.4.4. Batalla



2.4.5. Diálogos



3. Recursos

3.1. Audios

- Menú: http://www.looperman.com/loops/detail/75568
- Combate: http://www.looperman.com/loops/detail/74107
- Explorando: http://www.looperman.com/loops/detail/73606

3.2. Imágenes

- Enemigos: http://opengameart.org/content/bosses-and-monsters-spritesheets-ars-notoria
- Héroe y *tileset* del mapa: http://opengameart.org/content/blowhard-2-blow-harder

3.3. Fuentes de texto

- Fuente *in-game*: http://www.fontsupply.com/fonts/V/Visitor2.html
- Fuente actual (y del resto de la memoria): http://www.linuxlibertine.org/