ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

ORCS AND TACTICS

Realizado por

Ángel Martínez Olivares

Dirigido por

José Ramón Portillo Fernández

Departamento de Matemática Aplicada 1

Sevilla, Diciembre 2012

Índice de contenido

1. Introducción

Los videojuegos, desde hace ya unos años, cubren las necesidades de entretenimiento de casi cualquier familia o individuo. La vida que se vive hoy día requiere de unas vías para desconectar un poco del estrés o sencillamente pasar un buen rato. Gracias a eso los videojuegos han tomado una gran relevancia en el tiempo libre de las personas.

Orcs and Tactics es un videojuego de estrategia táctica para el sistema operativo Android. Es una idea que surge de la afición por los videojuegos, basado en las historias y conceptos al estilo de El Señor de los Anillos, Dungeons and Dragons, Warhammer, etc. Todo salpicado con un toque personal y humorístico.

El sistema creado abarcará todos los aspectos de la ingeniería del software, desde el concepto del juego y los diseños del mismo hasta su implementación final.

Es un título para cualquier edad y se puede jugar de varias formas ya que, a parte de las reglas que pueda contener, da mucho lugar a la imaginación que quiera usar el jugador a la hora de plantear sus movimientos y acciones.

1.1. Idea principal

El juego de estaretegia táctico Orcs and Tactics se centrará en batallas que se desarrollarán de manera consecutiva, pudiendo el jugador repetir cualquier combate que ya haya jugado.

El tipo de juego será un RPG (Role-Playing Game) estratégico o táctico. Un RPG es un tipo de juego que se basa en que el jugador se mete en el papel de uno o más personajes, controlando a los mismos y a sus acciones y desencadenando hechos que afectan a la historia del juego. Otra característica son los niveles de jugador, que subirán mediante combates, y que harán más poderosos a los personajes.

Para explicar el desarrollo del juego podemos imaginar un tablero por el cual los personajes se moverán. Para saber qué personaje actúa en cada momento se instaurará un sistema estático de turnos, la posición en esa lista dependerá del personaje. Un personaje ocupa una sola casilla, se podrá mover un límite de casillas por turno y podrá interactuar con sus casillas adyacentes, todo dependiendo del tipo de personaje que sea.

El tablero tendrá obstáculos: partes sobre las cuales un personaje no podrá pasar y que obstaculizarán su camino en caso de que quiera pasar por ellas. Los escenarios serán distintos en tamaño y en aspecto, al igual que los obstáculos que podrán llegar a ocupar más de una casilla.

El enemigo tendrá las mismas restricciones que los personajes jugadores, pero tendrán acciones distintas y serán de tipos distintos. Habrán partidas de jugador contra jugador y jugador contra máquina.

El detonante para desarrollar un juego de este tipo fue sin duda una afición personal por los juegos de este estilo, la poca existencia de este género de juego en android y la posibilidad de contribuir a la comunidad del software libre.

El motor gráfico fue seleccionado gracias a que ya tenía una forma de reproducir los mapas hechos a partir de casillas, además tiene código abierto y su desarrolladorha estado aportando constantemente mantenimiento y extensiones nuevas. La última actualización remodeló el código del proyecto y comprimir sus extensiones en un archivo.

2. Definición de Objetivos

El mundo de los videojuegos es dinámico y cambiante, y las necesidades de los usuarios son específicas mientras que sus gustos pueden ser extensos o particulares. El objetivo es desarrollar un videojuego que se ajuste a las necesidades de usuarios específicos, aportando una aplicación entretenida, ágil, ligera y con un apartado visual decente.

Siendo este un proyecto de fin de carrera, el apartado visual será lo de menos, mientras que se le dará especial importancia al apartado lógico y de desarrollo. Aún así una base visual es necesaria para hacerse una idea de qué queremos y cómo, a parte los mapas sobre los que tendrán lugar las partidas necesitan ser diseñados con sus respectivos obstáculos ya que si no no podríamos aplicar los algoritmos de movimiento.

Los objetivos mínimos que se cumplen son:

- Responder a interactuación entre el usuario y la pantalla. Al ser un dispositivo táctil, todo lo que el usuario quiera hacer lo hará a través de toques en la pantalla del dispositivo. La respuesta será inmediata y exacta, y se reducirá al máximo el número de toques en pantalla que se tengan que dar para desarrollar las acciones durante el juego, para aportar agilidad a un sistema que depende de turnos y puede llegar a ser estático para el jugador.

- Un proyecto reutilizable, robusto y escalable. Lo importante del desarrollo es aportar ideas nuevas e innovar, pero no se debe olvidar que la estructura y el código deben ser reutilizables, escalables y de fácil mantenimiento. Se debe buscar un equilibrio entre los componentes del mismo para fomentar estos atributos.

- Aplicar el algoritmo de la A estrella (A star) para calcular el camino mínimo de un punto a otro. La necesidad de esto se debe a que al haber obstáculos en el mapa, para que un personaje vaya automáticamente de un punto a otro habiendo un obstáculo este debe elegir la ruta más corta hasta ese punto para rentabilizar el límite de movimiento con el que cuenta.

- Desarrollar una inteligencia artificial para partidas de máquina contra jugador. Como es un juego que va turno a turno, las acciones de la máquina dependerán de los personajes cercanos que tenga, cómo están de cerca, su nivel.

- Desarrollar una inteligencia artificial para partidas de máquina contra jugador. Como es un juego que va turno a turno, las acciones de la máquina dependerán de los personajes cercanos que tenga, cómo están de cerca, su nivel de vida, si puede llegar a ejecutar alguna acción ofensiva contra ellos, entre otras.

- Variedad de personajes. El jugador deberá ser capaz de controlar a un grupo de personajes, todos con aptitudes únicas y distintas, con fuerzas y debilidades, para que el componente táctico del juego fuera aún más fuerte. Si esto no fuera así el juego podría dejar de ser entretenido.

Al igual que hay objetivos mínimos, la aplicación cubrirá algunos de estos aspectos:

- Cuidar el nivel de gráfico y de diseño. Debería haber al menos un diseño por cada tipo de personaje, y deberían haber animaciones para el movimiento y las acciones de cada personaje. A parte el diseño de los personajes deberá ser cuidado para asegurar que el jugador nunca se confunda entre uno y otro.

- Posibilitar el lanzamiento de dados en pantalla mediante el acelerómetro. Teniendo en cuenta el sistema de juego, podría añadirse la posibilidad de que el jugador lanzara un dado en pantalla para el cáculo de los números aleatorios que tendrá internamente el juego. Como es una prestación añadida que puede ralentizar la experiencia en el juego se tiene que hacer posible su deshabilitación a gusto del ususario.

- Incluir crecimiento de personajes. Como cualquier RPG, los personajes deberían ganar algún tipo de puntuación determinada por cada partida, haciendo que al llegar a cierta puntuación las aptitudes de los personajes se modificaran. Es el sistema de subir niveles de personaje, los puntos necesarios se podrían conseguir por acción o se repartirían después de cada partida.

- Aptitudes en grupo. Si queremos que el juego fuera completo, acciones entre dos o más personajes deberían ser implementadas, esto daría al jugador una experiencia más gratificante a nivel de tácticas, ya que deberá decidir qué personajes poner en qué posición con más cuidado, pero con una recompensa en forma de nuevas acciones únicas.

3. Análisis de antecedentes y aportación realizada

Los videojuegos son un software creado para el entretenimiento. Su aparición se remonta a los primeros ordenadores como juegos de ajedrez para las supercomputadoras tipo ENIAC, durante los años 70 aparecen las primeras consolas: plataformas que reproducen su propio software contando con cierta diversidad de juegos.

El mundo de los videojuegos ha tenido una evolución exponencial en los últimos años: el paso a consolas que permitían reproducir gráficos poligonales hizo que la realidad virtual fuera cada vez más real, hasta las consolas de última generación (Xbox 360, PS3, Wii, Nintendo 3DS, PSVita, etc.) que tienen unas especificaciones de hardware que pocos ordenadores de su mismo año de salida pudieran tener. La potencia de éstas consolas suele ir acompañada de un impecable plan de marketing. Todos estos factores han hecho que aparezca un boom consumista que afecta a consolas, ordenadores y los propios videojuegos.

Poco a poco, los videojuegos se han convertido en la afición de más y más gente. Incluso hay una forma de llamar a esas personas que se tiran tantas horas delante de sus consolas u ordenadores: son llamados gamers. La pasión de estas personas es jugar a videojuegos, ya sea por su historia, por la adrenalina que generan o la satisfacción de jugar a los mismos.

El consumo de estos productos se ha disparado en los últimos años, gracias a eso cada vez más desarrolladores y compañías se interesan por este mundo tan trepidante y cambiante.

Hoy en día en España el 90% de los productos de entretenimiento lo cubren los videojuegos: desde los que se reproducen en las consolas hasta tabletas gráficas y los teléfonos móviles. Siendo los últimos los que mayor interés han captado entre los desarrolladores de software del mundo entero, en especial para los dispositivos con sistema operativo Android.

El interés en el desarrollo de videojuegos y aplicaciones para smartphones y tablets con Android se debe a que desarrollar para el mismo es gratuito. Subir tus propias aplicaciones al precio que quieras a la plataforma Google Play sólo te cuesta unos 24 dólares y esa licencia dura de por vida; con Apple tienes que pagar para ser desarrollador y para subir tus aplicaciones a AppStore cuesta unos 100 dólares al mes.

¿Por qué ese precio tan distinto entre Google y Apple? La explicación es larga pero lo más importante es que Apple protege todos sus productos con una licencia que evita las copias, la piratería e incluso la reproducción de parte de las ideas de los desarrolladores de los juegos. Así, usar un personaje en otro juego que se parezca bastante al de otro que ya estuviera antes en la AppStore puede hacer que tu juego sea retirado e incluso en casos extremos el autor puede ser sometido a denuncia.

Con esta información en mente se empezó el proyecto Orcs and Tactics, un RPG táctico basado en grandes predecesores como Final Fantasy Tactics, Tactics Ogre y Fire Emblem entre otros. La idea principal del proyecto es aportar un código a la comunidad de Software Libre que sea escalable, de fácil modificación y adaptación a otro juego del estilo.

El motor gráfico era el principal problema del desarrollo del proyecto: un motor gráfico en sí supone un proyecto tan grande como un videojuego. Después de buscar y comparar con el resto de motores gráficos de Android se acabó eligiendo AndEngine. Entre otras cosas porque es de código abierto y libre para modificaciones. Además cuenta con una extensión para la carga de mapas por casillas, algo fundamental para el desarrollo de un juego de RPG táctico.

4. Análisis de requisitos

4.1. Introducción

El objetivo de esta sección es analizar y estudiar todos los requisitos del sistema que vamos a construir.

Como este proyecto no cuenta con especificaciones externas ni referentes pasados habrá que definir los elementos y las reglas que constituyen Orcs and Tactics.

Los objetivos de esta sección son:

- Describir los elementos del sistema usando un lenguaje menos formal para poder entender la aplicación de una forma más natural.

- Definidos ya los elementos y el entorno se pasará a un modelado del mismo mediante UML 2.0.

- Modelado de casos de uso para poder ver todas las funcionalidades que incorporará la aplicación.

- Problemas encontrados y errores cometidos. También llamado documento post mortem, servirá para aprender de los errores encontrados para no volver a repetirlos, y prevenir en un futuro cualquier problema parecido a los vistos anteriormente.

4.2. Estudio de los elementos que conforman el sistema

En esta sección se verán todas las características de los personajes, sus límites y restricciones y los elementos que conforman un escenario.

Al ser un juego táctico los mapas juegan un papel muy importante, según qué tipo de mapa sea hará que un combate sea más o menos difícil llegando a suponer un reto para la habilidad del jugador.

Todos los elementos serán citados, analizados y descritos en este capítulo.

4.2.1. Reglas de juego

El tipo juego está basado en un tablero, es decir, los personajes son como fichas que se mueven a través del mapa y al igual que en el resto de juegos de tablero tenemos ciertas restricciones que serán comunes para todos los personajes.

Los personajes tienen unos valores llamados características, estos son: ataque, defensa, movimiento, puntos de golpe, daño y velocidad. Estas características son la columna vertebral del juego, todas las acciones posibles dependen de ellas.

- Victoria y derrota

La condición para ganar un combate es siempre la misma: que no quede ningún personaje enemigo con vida. Si el enemigo consigue abatir a todos los personajes la partida resultará en una derrota.

La forma de abatir personajes es intercambiando ataques con ellos hasta que no le queden puntos de golpe, cada vez que un ataque impacte con éxito se reducirán los puntos de golpe del objetivo, estos serán representados como barras de vida encima de cada personaje para saber cuánto le queda para ser derrotado.

Si un personaje es abatido no desaparece para siempre, en el siguiente combate estará disponible al máximo de su capacidad de vida, al igual que el resto. La muerte permanente no existe en Orcs and Tactics.

Al ganar un combate, el jugador podrá elegir repetirlo si así lo desea, sin afectar al resto de combates o al desarrollo de los mismos. Cada combate es independiente del resto, pero hay que superar combates para poder desbloquear los siguientes y que se puedan jugar.

- El sistema por turnos

El juego está basado en combates por turnos, es decir, cuando es el turno de un personaje ningún otro podrá actuar hasta que este termine sus acciones.

Lo rápido que un personaje actua depende de su característica de velocidad, esta característica determinará su posición en la listade turnos, dicha lista podrá ser visualizada en cualquier momento por el jugador para saber de qué forma se desarrollará el combate.

Hay una regla especial que hace que un personaje adelante su posición en la lista de turnos, eso depende de la acción que haga en su turno:

a) El personaje se mueve y ataca o al revés. Su turno se queda en la misma posición en la lista por lo que su turno siguiente llegará cuando el resto de personajes haga su turno.

b) El personaje se mueve y no ataca. Su turno se adelantará una posición en la lista, por lo que su siguiente turno llegará antes.

c) El personaje ataca. Su turno se quedará igual que antes, al igual que el caso a).

d) El personaje no hace nada. Su turno se adelantará dos posiciones en la lista de turnos.

Así, introducimos el concepto de acciones, una acción entonces tiene varias posibilidades:

- Acción ofensiva: atacar o usar una habilidad activa.

- Acción pasiva: usar una habilidad pasiva.

- Movimiento.

- Esperar: termina el turno aunque le queden acciones al personaje.

Un personaje suele tener un máximo de dos acciones por turno, y nunca puede repetir una que ya haya hecho antes como por ejemplo atacar dos veces en el mismo turno pero hay casos especiales en los que el personaje tiene más de una oportunidad para intentar un ataque, esto depende de si activamos dicha habilidad antes o no.

- El sistema de probabilidades

En este juego todas las acciones excepto el movimiento están condicionadas a la probabilidad que se representa como un porcentaje, el valor de ese porcentaje que se debe superar depende de las características enfrentadas correspondientes de los personajes.

Cada vez que se proceda a hacer una acción, se calculará un valor aleatorio entre uno y cien, si el valor resultante supera la prueba de probabilidad definida la acción se ejecutará correctamente y si no la acción fallará.

Las posibilidades son:

a) Ambos valores son iguales, la prueba a superar será el valor 50% inclusive.

b) El valor de la característica del personaje que ejecuta la acción es mayor que el del personaje objetivo, la prueba a superar será el 25% inclusive.

c) El valor de la característica del personaje que ejecuta la acción es menor que el del personaje objetivo, la prueba a superar será el 75% inclusive.

Si un atacante consigue superar mediante el cálculo de la probabilidad la prueba que se exige, podrá hacer daño a su objetivo, dicho daño se explicará más adelante.

En caso de que el resultado del cálculo sea 90% o más se considera un impacto crítico, lo que duplicará el daño que se provoca a su objetivo.

- Movimiento

Lo más importante que se debe saber del movimiento es que los personajes se mueven en un área. El área está limitada por un número genérico para cada personaje y, normalmente, nunca cambia lo que quiere decir que deberemos tenerlo en cuenta cuando llegue nuestro turno ya que un movimiento arriesgado puede poner en peligro la vida de un personaje.

Para poder hacernos una idea de cómo afecta el límite de movimiento de un personaje al área en la que puede moverse es mejor ver la figura1, que describe el tamaño del área según el valor del movimiento del personaje. Se debe tener en cuenta que no se está contando con los posibles obstáculos que puedan haber en el mapa, por lo que el área de movimiento descrita abajo es el máximo que puede haber para cada valor de movimiento.

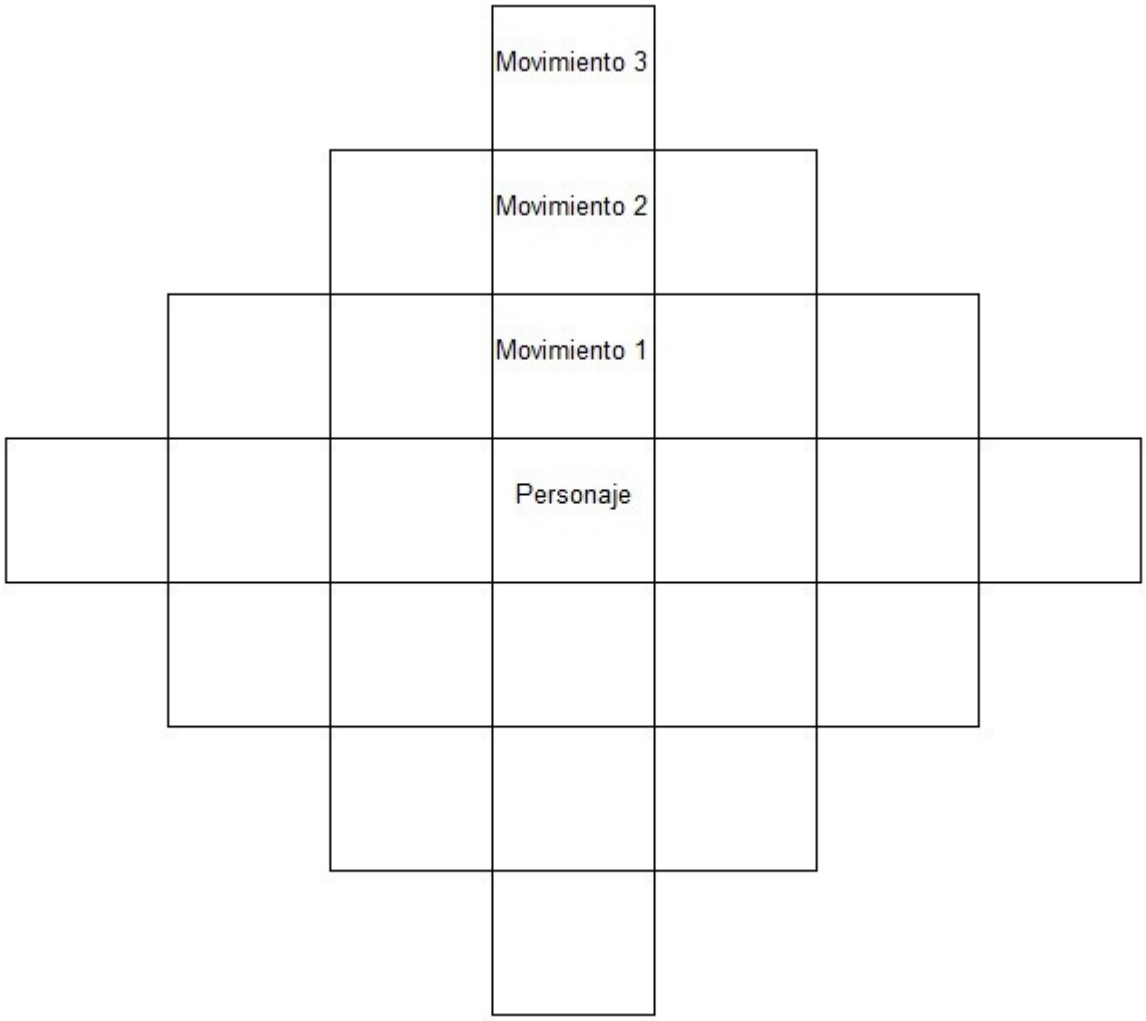


Figura 1

En la imagen vista anteriormente vemos representada la rejilla de movimiento correspondiente al valor 3, el personaje puede moverse hasta cualquiera de las casillas dentro del área de movimiento en caso de que no esté ocupada por algún obstáculo.

Hay personajes exentos de la penalización de movimiento, también los hay que ocupan más de una casilla, los veremos más adelante, pero el movimiento se aplica de la misma forma.

- El ataque y la defensa

Lo más básico para poder vencer en una batalla es que todos los enemigos sean vencidos y que sobreviva al menos un personaje. Para acabar con los enemigos debemos atacarles, pero no los ataques nunca son cien por cien seguros, ya que el éxito de los mismos está condicionado a la probabilidad que se delimita por unos valores de ataque y defensa que procederemos a explicar.

Cuando un personaje ataca a otro, se comparan los valores de defensa y de ataque de cada uno, como se describe en la sección de probabilidades, se comparan ambos valores, y la prueba de probabilidad a superar depende de los mismos.

Cada personaje tiene un valor de ataque y defensa predeterminado, a más altos sean estos valores hay más posibilidad de impactar un golpe y de evitar ser impactado.

- El daño y los puntos de golpe

Al igual que cada personaje tiene ataque y defensa para poder impactar golpes y esquivar los mismos, tiene un daño predeterminado y unos puntos de golpe predeterminados.

Lo que diferencia estas características del ataque y la defensa es que no tienen probabilidades: son parte de la acción de atacar. Es decir, para provocar daño a un personaje primero hay que atacarle y, si el ataque tiene éxito, el atacante provocará daño dependiendo de su característica de daño.

El daño tiene un funcionamiento especial: suele oscilar entre dos números, un máximo y un mínimo. Es decir, el daño provocado nunca es fijo, la diferencia entre el daño máximo y mínimo y el valor de los mismos depende del personaje al igual que los puntos de golpe.

Si los puntos de golpe llegan a cero, es decir, que la barra de vida que representa los mismos se vacía, el personaje quedará fuera de combate para el resto de la batalla.

- El rango de ataque

Para poder intentar un ataque contra otro personaje, debemos tener en cuenta la distancia a la que se encuentra de nuestro personaje. El rango de ese ataque también depende de cada personaje y el rango del mismo se representa igual que el movmiento, el objetivo debe estar dentro de alguna de las casillas que representan el área de ataque.

Hay personajes que sólo pueden atacar a los que estén en casillas adyacentes a los mismos mientras que hay otros que pueden atacar a distancia incluso desde detrás de alguna casilla que no permita movimiento como una piedra o un tronco.

Para seleccionar un objetivo que esté dentro de nuestra área de ataque sólo tenemos que tenerlo en dicha área, que se verá representada a petición del jugador si toca la pantalla en la casilla en la que se encuentra el personaje. Si toca una casilla donde esté un enemigo y se encuentre en el rango de ataque del personaje, el jugador podrá intentar un ataque contra él y, si tiene éxito, provocará daño al objetivo.

Abajo tenemos dos ejemplos: uno de un personaje de corto alcance (Figura 2) y otro de largo alcance (Figura 3).

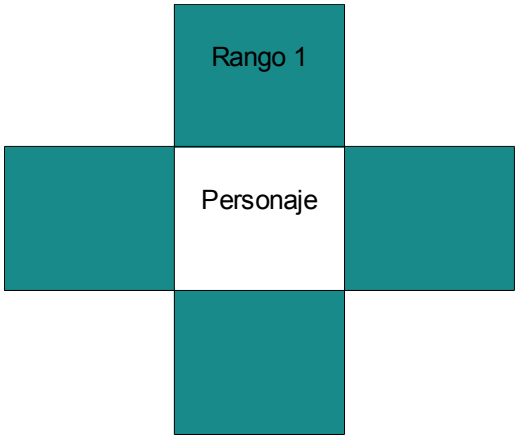


Figura 2

Este es el ejemplo de corto alcance, sólo puede atacar a las casillas que comparten uno de los lados con ella, el siguiente ejemplo es el de un personaje de largo alcance:

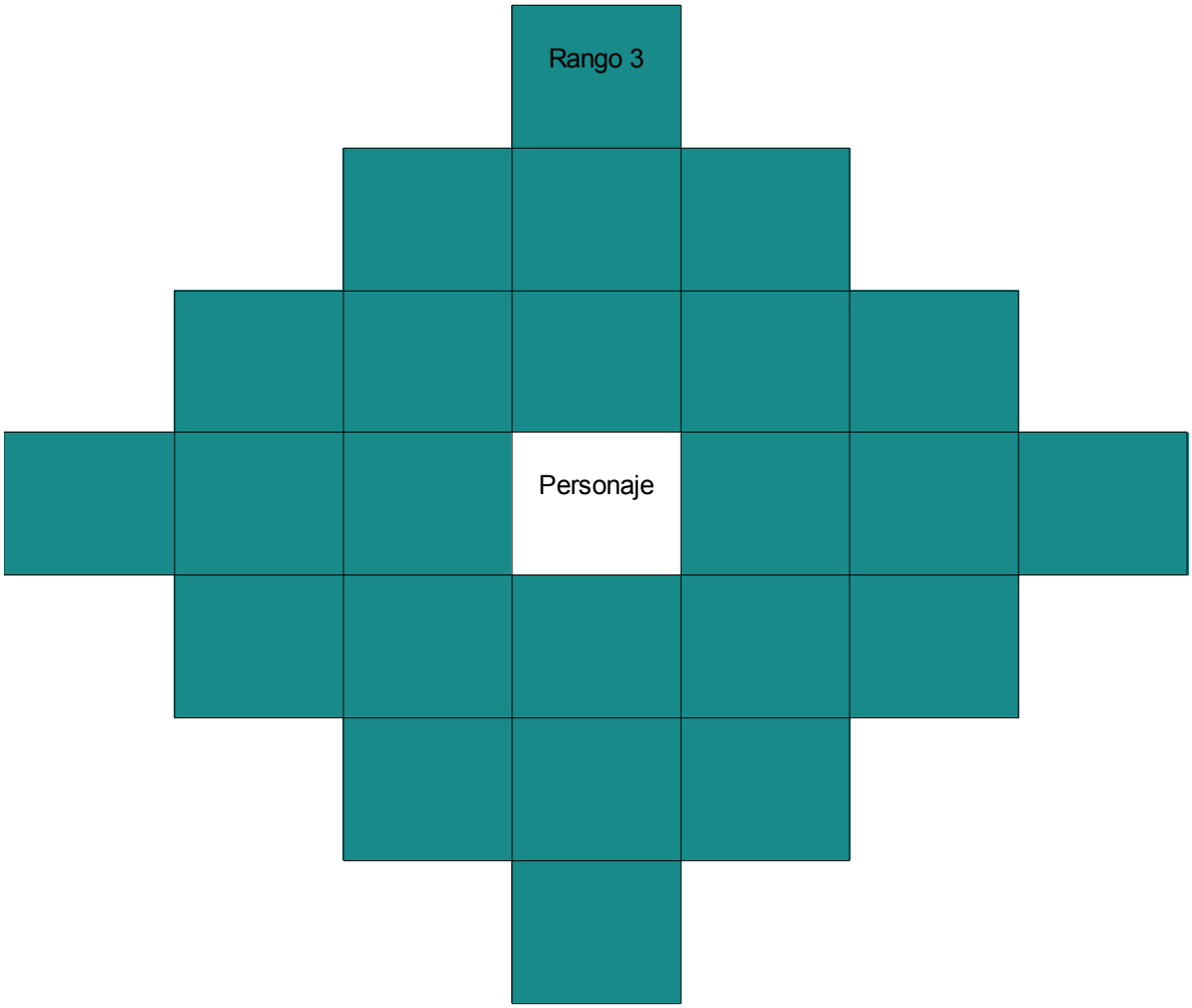


Figura 3

La ventaja sobre el personaje de corto alcance es clara, no sólo puede atacar a casillas adyacentes, sino a todas las demás que estén dentro de su área, y si tiene un obstáculo entre su objetivo y el personaje lo ignora si elige atacar al objetivo.

- Habilidades

Cada personaje puede tener una serie de habilidades, que son acciones distintas al ataque y al movimiento, las podemos agrupar en dos grupos:

- Pasivas: Pueden consumir una acción del personaje o no, normalmente incrementan momentáneamente alguna o algunas características del personaje, o hacen acciones especiales como atacar dos veces o esconderse.

- Activas: Usadas para hacer daño a su objetivo, pueden consumir todas las acciones del personaje o sólo la acción de ataque o movimiento.

Hay casos especiales en los que las habilidades tienen una característica no vista hasta ahora: el área. Esto se ve en el caso del mago, tiene una habilidad que tiene un rango para lanzarla y otro rango más que describe el área donde afecta la habilidad. A la hora de describirla lo veremos con más detalle.

- El mapa y el terreno

Un elemento fundamental del juego es el mapa, este es representado como un tablero en el que cada casilla puede estar:

a) Vacía y sin elemento bloqueante, es decir, cualquier personaje puede pasar por ella o quedarse quieto en ella.

b) Vacía y con elemento bloqueante, un personaje no puede quedarse quieto en dicha casilla, ni puede pasar por ella, para sortear estas casillas el personaje debe rodearlas.

c) Ocupada por un personaje, al igual que el caso anterior, un personaje no puede ocupar la misma casilla que otro que ya está en ella, y para pasar debe sortearlo al igual que con los elementos bloqueantes.

Los elementos bloqueantes, su número y su posición, dependerán del mapa que estemos jugando, y se representarán como objetos del terreno: piedras, árboles, etc. Para que se distingan de las casillas que sí pueden ser atravesadas por el personaje.

Cada casilla representa un espacio sobre el que el personaje puede estar o no, por tanto, el número de casillas que un personaje puede moverse depende de su característica de movimiento, es decir, si un personaje tiene un número X de movimiento podrá moverse en un área de X casillas.

Si un personaje quiere sortear una casilla bloqueada deberá rodearla, por tanto si tiene un bloqueo en su camino la cantidad de movimiento que podrá hacer se verá reducido según el tamaño del bloqueo.

4.2.2. Descripción de personajes

Aquí se hará una descripción de los personajes que hay en el juego incluyendo su descripción ,sus características y sus habilidades.

- Personajes controlados por el jugador

Estos son los personajes con los que el jugador contará desde el principio hasta el final del desarrollo del juego. Tiene una gama versátil de los mismos, dejando la estrategia para ganar los combates totalmente abierta a la libertad del jugador.

Los personajes tienen una descripción, para poder hacernos una idea del aspecto o raza mitológica que tienen. Las caracterísitcas son fundamentales para la comprensión de las aptitudes de los personajes en el combate, ya que son los números usados en todas las acciones que pueden hacer.

Las habilidades son acciones especiales únicas de cada personaje, que le permite conseguir ventaja ante situaciones particulares, estas habilidades suelen sustituir (aunque se diga lo contrario) al intento de ataque del personaje y/o al movimiento. Las habilidades son cruciales en fases avanzadas del juego, donde la desventaja numérica será la principal dificultad.

- Gwail el Guerrero

Descripción:

Un Enano de las montañas del hierro, es el típico personaje que sirve tanto para atacar como para defender, no es el que más daño provoca pero los problemas que le generará a los enemigos para abatirle serán suficientes para que los personajes diseñados para hacer un daño mayor pero que sean más débiles defensivamente tengan tiempo de acabar con los enemigos.

Características:

- Ataque: 15
- Defensa: 15
- Daño: 3-13
- Movimiento: 3
- Velocidad: 20
- Rango de ataque: 1
- Puntos de golpe: 60

Habilidades especiales:

- Carga: si tiene un enemigo en rango de alcance de movimiento más lejano de una casilla y no hay un obstáculo entre él y el enemigo, el enano corre hacia su objetivo y se pone justo delante de él para intentar un ataque con dos puntos más a su característica, es decir, a la hora de comparar entre su ataque y la defensa del enemigo cuenta como si el guerrero tuviera un 17 en ataque. Esta habilidad sustituye una acción de movimiento y ataque por lo que se debe usar antes de hacer nada con el personaje.

- Formación defensiva: el enano concentra todos sus esfuerzos en no ser golpeado hasta su siguiente turno, su defensa sube a 17 pero no puede atacar hasta el siguiente turno. Esta habilidad sustituye a una acción de ataque.

- Elwyn el Arquero

Descripción:

Un experto explorador de los bosques de los altos elfos de Nevindia, equipado con un gran arco que le permite atacar a distancia y con una puntería casi insuperable es un personaje que prefiere siempre mantener tierra de por medio entre él y sus enemigos. Debido a su bajo nivel de puntos de golpe y con una defensa que, ante enemigos hábiles, es prácticamente inútil es un personaje con el que hay que tener cierto cuidado para evitar una muerte fácil aunque su ventaja con el arco es indiscutible.

Características:

- Ataque: 18
- Defensa: 13
- Daño: 2-9

- Movimiento: 5
- Velocidad: 25
- Rango de ataque: 5
- Puntos de golpe: 40

Habilidades:

- Tiro doble: Elwyn dispara dos flechas rápidamente, descuidando su ataque en ambos disparos pero añadiendo un ataque más, es decir, en lugar de hacer una prueba para un sólo ataque con su característica a 18 hace dos intentos de ataque con la característica a 15. Esta habildiad sustituye a una acción de ataque.

- Concentración: una vez por combate, el elfo puede elgir hacer un ataque sin haberse movido antes, cambiando sus características de ataque y daño a 21 y 8-15 respectivamente.

- Rolf el mago

Un sabio e inteligente mago de la escuela arcana de la ciudadela Stutgart. Este personaje no tiene fortalezas en cuerpo a cuerpo, por lo que evita en todo momento lanzando proyectiles con su vara, también está dotado de dos habilidades muy poderosas que pueden dañar a varios grupos de enemigos a la vez o dañar a uno repetidas veces.

Características:

- Ataque: 10
- Defensa: 11
- Daño: 1-6

- Movimiento: 3
- Velocidad: 15
- Rango de ataque: 3
- Puntos de golpe: 20

Habilidades:

- Bola de fuego: el mago lanza una bola de fuego, que hace daño en un área a todo personaje que se encuentre en ella aunque sean personajes controlados por el jugador. El rango de ataque para lanzar la bola es de 5 mientras que el área en la que impacta mide 2. Este hechizo no se podrá lanzar hasta que pasen dos turnos del personaje, y una vez lanzado se tendrá que volver a esperar este tiempo para lanzarlo. El daño del hechizo es de 8-48.

- Proyectil mágico: el mago lanza varios proyectiles que impactan en su objetivo haciendo daño de manera consecutiva, este hechizo se puede lanzar al principio del combate, pero el personaje tiene que esperar al menos un turno para volver a lanzarlo. El daño del hechizo es de 6-24.

- Gwen la ladrona

Una mediana del valle verde, viajante, aventurera y especialista en ataques por la espalda. La principal aptitud de este personaje es la capacidad de ocultarse si tiene cerca algún obstáculo como piedras o árboles y coger desprevenido a cualquier enemigo que se le acerque. También cuenta con una buena defensa para evitar los ataques en caso de que se quede al descubierto, aunque con pocos puntos de golpe y ataque y daño mediocres en combate normal, es un personaje que debe moverse con cautela para evitar grandes daños.

Características:

- Ataque: 13
- Defensa: 15
- Daño: 1-6
- Movimiento: 5
- Velocidad: 30
- Rango de ataque: 1
- Puntos de golpe: 30

Habilidades:

- Esconderse: la habilidad principal del personaje, si está pegada a una piedra, árbol o cualquier obstáculo puede esconderse, desapareciendo de la vista del enemigo y evitando por tanto cualquier daño que se le provoque. Esta habilidad se puede usar en lugar de un ataque permitiendo al personaje moverse y esconderse en el mismo . El personaje puede usarla al principio del combate pero no podrá volver a usarla hasta que pase un turno y no podrá volver a esconderse en el mismo sitio.

- Ataque furtivo: si está escondida y tiene un enemigo al alcance puede intentar hacer un ataque especial con una mayor característica de ataque y daño. Después de ejecutar esta acción se podrá mover como cuando hace un ataque normal. El ataque de esta habilidad es 18 y el daño es 5-30.

- Ludvig el clérigo

Un bonachón practicante de las artes sagradas del monasterio de monjes de Tiberia, es el único personaje que puede hacer que otros recuperen puntos de golpe, tiene buenas aptitudes defensivas aunque un poco peores que las del guerrero pero lo compensa con magia sagrada con la que no sólo cura sino que también protege a otros personajes de los daños.

Características:

- Ataque: 13
- Defensa: 14
- Daño: 2-9
- Movimiento: 3
- Velocidad: 15

- Rango de ataque: 1

- Puntos de golpe: 50

Habilidades:

- Curar heridas: una vez cada dos turnos el personaje puede curar a otro que esté en un rango de 1, un personaje que está inconsciente no puede ser curado de nuevo. Esta acción sustituye una acción de ataque. La cantidad de vida curada es de 7-21.

- Escudo de los dioses: una vez cada dos turnos el personaje puede generar una bonificación a la característica defensiva de cualquier personaje que esté en un rango de 3, sumando 2-4 puntos de defensa durante tres turnos del clérigo. El efecto de esta bonificación es inmediato al igual que su desaparición. Esta acción sustituye una acción de ataque.

- Enemigos

Los enemigos, al igual que los personajes, tienen descripciones y características propias e incluso algunas habilidades dependiendo del enemigo.

La variedad de enemigos es amplia, algunos tienen condiciones especiales como ignorar los obstáculos del terreno u ocupar más espacio del que normalmente ocuparían, haciendo más complicado a los personajes poder moverse en el combate u obstaculizar su paso para que otras unidades puedan atacar a distancia.

La inteligencia artificial será descrita en otro capítulo distinto, aquí nos centraremos en describirlos de la misma forma que a los personajes controlados por el jugador.

- Goblin

Pequeño, mezquino y prescindible. Es usado por sus líderes para distraer a sus enemigos mientras unidades más fuertes los usan de escudo para recibir menos daño de los ataques enemigos. Suelen ir en grupos de al menos dos o tres, aunque parezcan débiles en un grupo suficientemente grande pueden hacer que el combate sea bastante complicado para el jugador ya que restarían movilidad en muchas casillas.

Características:

- Ataque: 10
- Defensa: 10
- Daño: 1-6
- Movimiento: 4
- Velocidad: 25

- Rango de ataque: 1

- Puntos de golpe: 10

Habilidades: Ninguna

-Goblin arquero

Al igual que el anteriore, débil y pequeño, pero tiene un poco más de ataque y está equipado con un arco por lo que congenia perfectamente con los que están equipados con armas de corto alcance. Es la fuerza principal de largo alcance del enemigo.

Características:

- Ataque: 11

- Defensa: 10

- Daño: 1-6

- Movimiento: 4

- Velocidad: 25
- Rango de ataque: 3
- Puntos de golpe: 8

Habilidades: Ninguna

- Orco

Grande, intimidatorio, verde y muy fuerte. Es la columna vertebral de corto alcance del ejército enemigo, equipado con hachas a dos manos, inflinge graves daños a sus enemigos, cuenta con una habilidad para incrementar su ataque y su daño a precio de defensa (que de por sí no tienen mucha) y es un enemigo a tener en cuenta si no se acaba con él pronto.

Características:

- Ataque: 13

- Defensa: 12
- Daño: 5-17
- Movimiento: 3
- Velocidad: 20
- Rango de ataque: 1
- Puntos de golpe: 40

Habilidades:

- Furia primigénea: una vez por combate el orco se enfurece durante tres turnos, incrementando su ataque y su daño en dos puntos, alcanzando un ataque de 15 y un daño de 7-19 pero baja su defensa hasta 10. Esta acción sustituye una acción de movimiento pero también la puede usar en lugar de una de ataque, dependiendo de si el personaje se ha movido o no.

- Troll

Personaje de leyendas, devorador de aventureros y combatientes por igual. Aunque no se encuentra hasta fases avanzadas del juego, es un enemigo temible: ocupa un espacio de dos casillas por dos casillas, dificultando la posibilidad de rodearle para acceder a otros enemigos o huir de él. Es muy fuerte y tiene muchos puntos de golpe, pero descuida mucho la defensa. Su manera de atacar es especial ya que puede cubrir más espacio que otros personajes que estén equipados con armas de corto alcance.

Características:

- Ataque: 18
- Defensa: 11
- Daño: 10-16

- Movimiento: 4

- Velocidad: 15

- Rango de ataque: 2

- Puntos de golpe: 80

Habilidades:

- Engullir: su afición por devorar todo lo que le rodea hace que los goblins que tenga cerca sean muy suculentos. Dos veces por combate el troll puede devorar un goblin y restablecer 10 puntos de golpe. Esto consume su acción de ataque para ese turno.

- Regeneración: de manera pasiva, el Troll se cura automáticamente 6 puntos de golpe al principio de cada turno del mismo.

4.3. Modelo de análisis

El modelo de análisis es una parte fundamental del desarrollo de cualquier software lo suficientemente complejo como para que necesite un modelo de análisis.

El objetivo principal de este modelo es comprender cómo se relacionan todos los elementos descritos en el anterior apartado para poder visualizar la envergadura que puede alcanzar el proyecto.

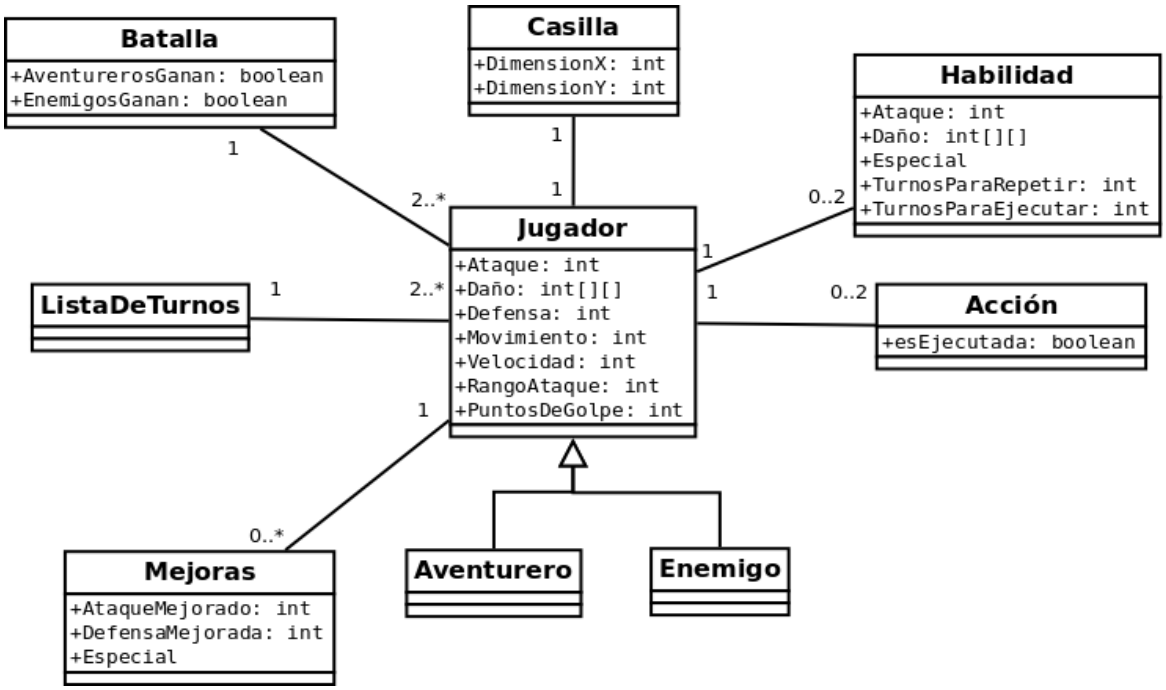
Después de tener un modelo de análisis, extraer la información relevante para el siguiente modelo es mucho más sencillo que intentar hacer un modelo de clases directamente sin analizar previamente el problema. Por eso el modelo de análisis no debe tener ningún error o incoherencia ya que se arrastraría hasta el modelo principal y corregirlo sería bastante costoso en tiempo.

Para poder comprender correctamente el modelo global de análisis iremos analizando parte por parte del modelo total para que sea más sencillo de comprender y de localizar posibles errores que pueda tener el modelo.

4.3.1. Entorno del jugador

En esta sección veremos la información que el sistema debe reflejar respecto a todos los elementos ligados directamente al jugador.

El siguiente diagrama recoge dicha información:


Batalla
+AventurerosGanan: boolean
+EnemigosGanan: boolean
Casilla
+DimensionX: int
+DimensionY: int
Habilidad
+Ataque: int
+Daño: int[][]
+Especial
+TurnosParaRepetir: int
+TurnosParaEjecutar: int
Jugador
+Ataque: int
+Daño: int[][]
+Defensa: int
+Movimiento: int
+Velocidad: int
+RangoAtaque: int
+PuntosDeGolpe: int
ListaDeTurnos
Acción
+esEjecutada: boolean
Mejoras
+AtaqueMejorado: int
+DefensaMejorada: int
+Especial
Aventurero
Enemigo
1
2..*
0..2
0..*

El diagrama muestra las siguientes relaciones e identidades:

- *Jugador*: hace referencia al personaje jugable o al enemigo en cuestión con todos los elementos descritos en el anterior capítulo que hacen que se distinga uno de otro.

- *Características del jugador*: refleja todas las características descritas en el capítulo anterior, estas características tienen nombres comunes entre personajes pero sus valores son exclusivos de cada uno.

- *Lista de turnos*: la lista que visualiza todos los turnos de cada personaje que se encuentre en el cambo de batalla.

- *Acción*: como su nombre indica, describirá más tarde todas las acciones que puede hacer un jugador, la multiplicidad entre el jugador y la acción crea la restricción que no permite a un jugador hacer más de dos acciones en un mismo turno.

- *Lista de mejoras:* recoge toda la información referente a las habilidades que afecten a las características del jugador y en qué manera afectarán.

- *Aventurero*: recoge la información de todos los personajes que el jugador puede manejar, para así diferenciarlos de los enemigos.

- *Enemigo*: al igual que su homónimo jugable recoge la información de los enemigos, aunque después veremos que tienen un tratamiento distinto al de los personajes jugables.

- *Habilidad*: cada personaje puede llegar a tener hasta dos habilidades, que dependiendo del caso sustituirán acciones del personaje o no.

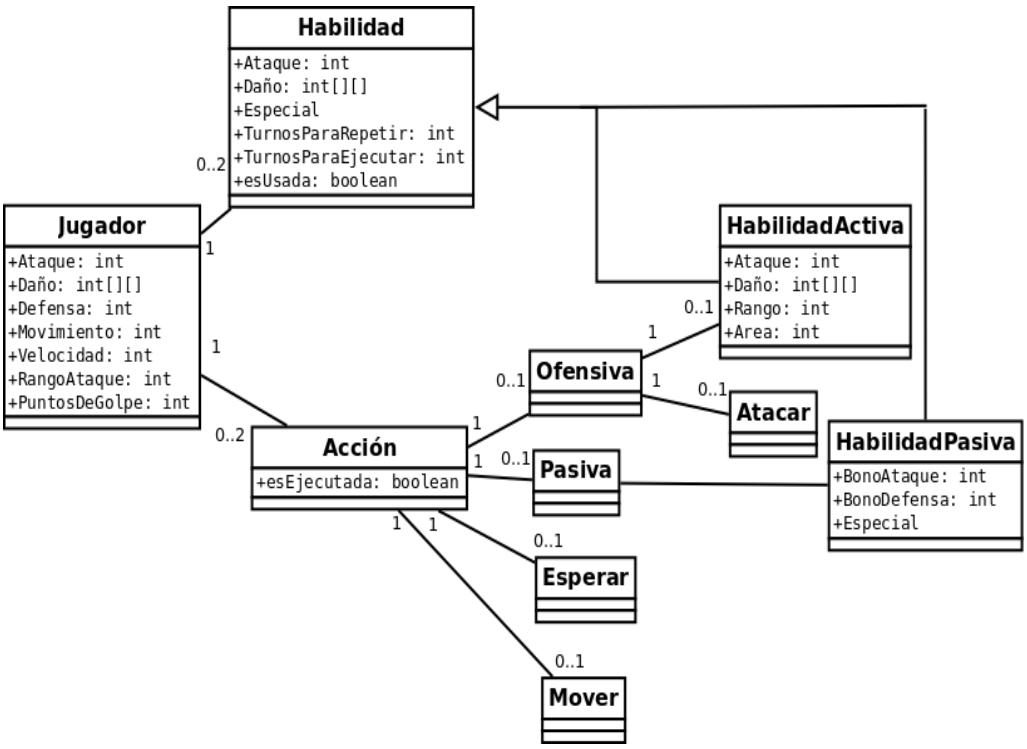
- *Casilla*: un personaje está encima de una casilla, este valor debe ser conocido para el sistema para calcular los rangos desde los que el jugador puede hacer sus acciones ofensivas o para recibir ataques de otros jugadores.

- *Batalla*: aquí es donde el sistema observará qué niveles de vida tienen los jugadores controlados por el jugador y los enemigos, para saber de quién será la victoria.

4.3.2. Entorno de las acciones

Aquí veremos todos los tipos de acciones que el sistema deberá tener en cuenta para saber qué hacer con las órdenes que reciba del jugador.

Para entender la multiplicidad entre el personaje y las acciones debemos saber que un personaje siempre puede optar por hacer o no una de las acciones abajo descritas, si puede hacerlas, y que si elige esperar no hará más acciones en su turno. El diagrama de más abajo representa las acciones que el jugador puede hacer:


Habilidad
+Ataque: int
+Daño: int[][]
+Especial
+TurnosParaRepetir: int
+TurnosParaEjecutar: int
+esUsada: boolean
0..2
Jugador
+Ataque: int
+Daño: int[][]
+Defensa: int
+Movimiento: int
+Velocidad: int
+RangoAtaque: int
+PuntosDeGolpe: int
1
1
0..2
HabilidadActiva
+Ataque: int
+Daño: int[][]
+Rango: int
+Area: int
0..1
1
0..1
Ofensiva
1
0..1
Atacar
1
1
Acción
+esEjecutada: boolean
1
0..1
Pasiva
HabilidadPasiva
+BonoAtaque: int
+BonoDefensa: int
+Especial
1
1
0..1
Esperar
0..1
Mover

El diagrama muestra las siguientes relaciones e identidades:

- *Jugador*: para saber qué jugador está haciendo las acciones debe haber una relación entre este y sus acciones.

- *Acción*: al igual que en el apartado anterior, define el máximo de acciones que puede hacer un jugador y está relacionada con las acciones específicas del jugador.

- *Habilidad*: recoge la información de las habilidades del jugador, si una habilidad ha sido usada o no.

- *EsUsada*: característica que usa el sistema ya que es necesario saber si una acción ha sido usada o no, para que el jugador no pueda repetir dos mismas acciones en un turno.

- *Ofensiva*: una acción orientada a dañar a otro jugador, tiene dos tipos ya que podremos hacer daño atacando o usando habilidades.

- *Dados*: Toda acción ofensiva requiere de un lanzamiento de dados, el sistema necesita recoger esa información ya que es indispensable para el cálculo del acierto de las acciones del jugador.

- *Atacar*: es la acción en la que un personaje elige atacar a otro que esté dentro de su rango, el rango, el ataque y el daño dependen del jugador que ataque y la defensa y los puntos de golpe dependen del jugador atacado.

- *HabilidadActiva*: recoge la información de las habilidades activas que el jugador puede hacer. Las habilidades activas tienen un ataque, daño y rango propias ya que a veces las características del jugador son modificadas por las habilidades. También tiene un atributo área pero sólo se usa en el caso de una de las habilidades del mago.

- *HabilidadPasiva*: recoge la información de las habilidades pasivas del personaje. La mayoría de las veces dan bonos a ataque, defensa y/o daño, pero también pueden tener efectos especiales como atacar dos veces en el mismo turno o esconderse y quedar fuera de la vista de los enemigos.

- *Mover*: como su nombre indica, la acción de movimiento necesita utilizar la característica movimiento del jugador para calcular el área de movimiento que tiene el propio jugador. El cálculo de este área se ha descrito anteriormente.

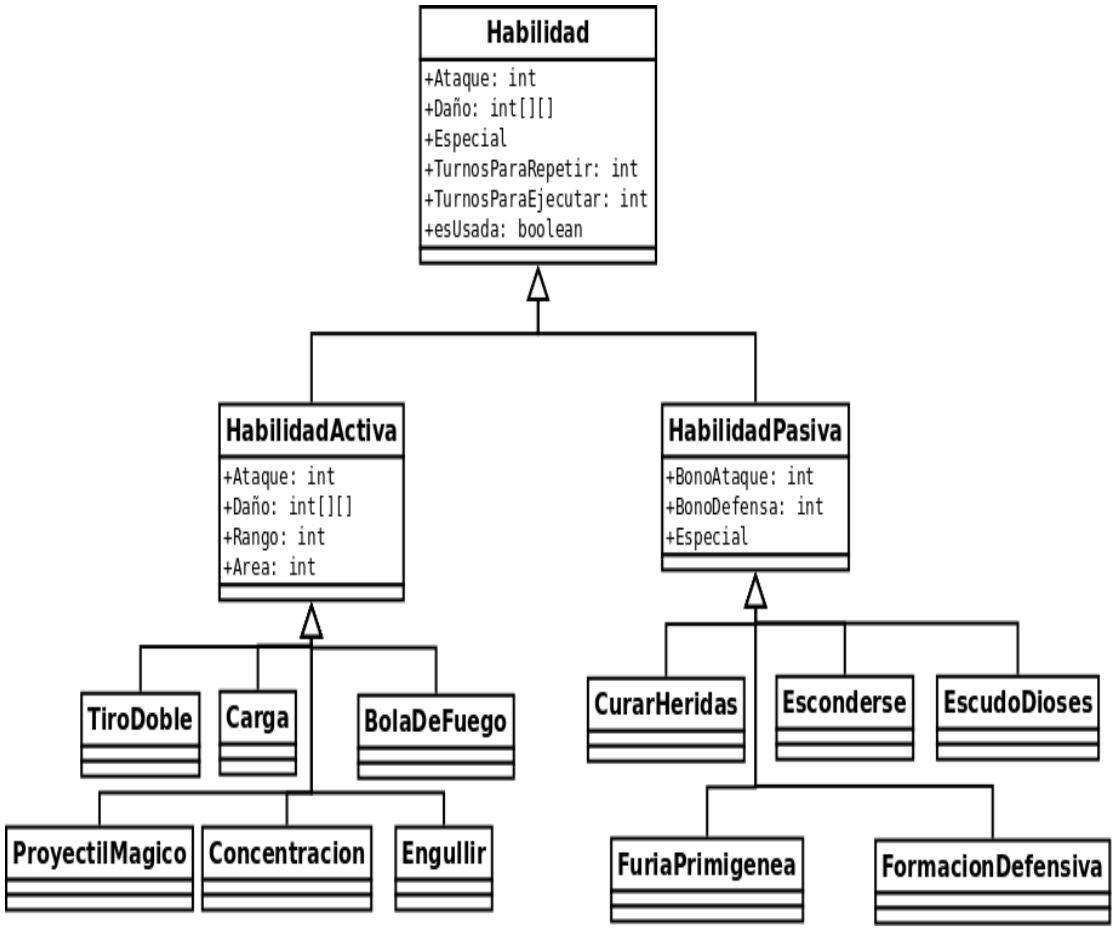
- *Esperar*: si el jugador elige hacer esta acción ya no podrá hacer ninguna otra hasta su siguiente turno. El sistema deberá conocer los valores del atributo EsUsada de cada acción para saber si aplicar el beneficio de no hacer ninguna acción. También deberá conocer la posición del jugador en la lista para adelantarle su turno si fuera necesario.

•

4.3.3. Entorno de las habilidades

Aquí explicaremos la relación entre las habilidades y sus tipos, ya que están agrupadas en dos tipos, como hemos visto anteriormente.

En los diagramas de más abajo tenemos los datos referentes a cada tipo de habilidades:



Las entidades mostradas en el diagrama son:

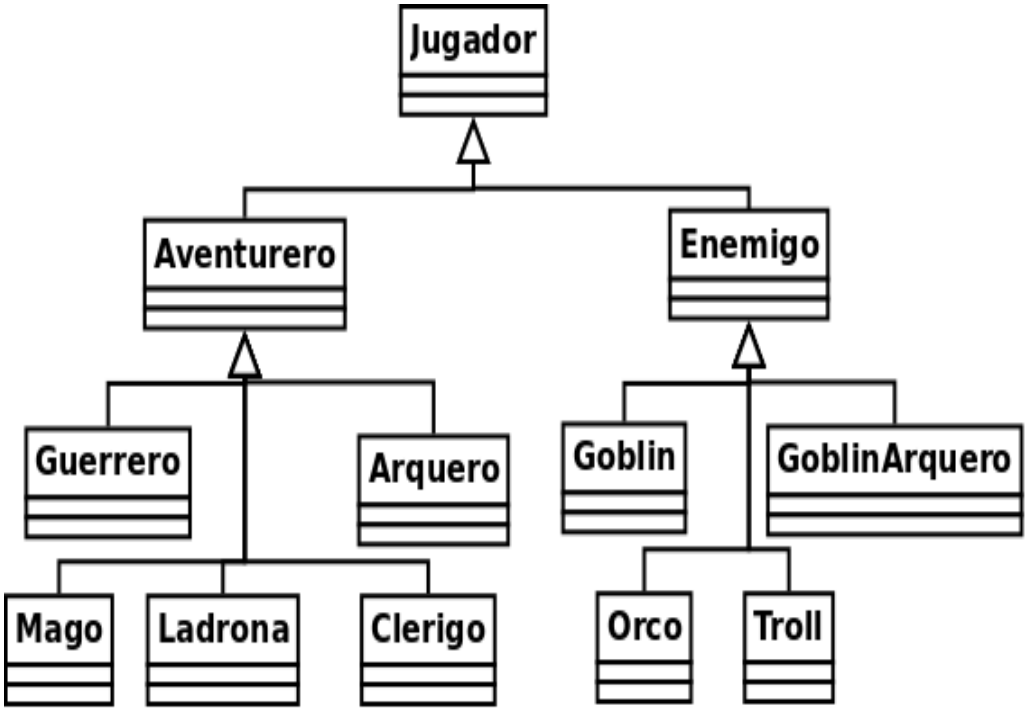
- Habilidades activas: las entidades hijas de Activa tendrán almacenadas las características particulares de cada habilidad, ya que el sistema necesita conocer dichos valores para poder aplicarlos a la acción concreta. Cada habilidad ya ha sido descrita anteriormente por lo que no serán descritas de nuevo.

- Habilidades pasivas: cada habilidad pasiva tendrá sus valores propios, aunque se darán casos en los que algunos de éstos estén vacios ya que no todas las habilidades pasivas tienen las mismas bonificaciones a las mismas características. El campo Especial le servirá al sistema para aplicar los beneficios dos habilidades que afectan al juego de manera distinta: Esconderse de la ladrona y Disparo doble del arquero, ya que tienen condiciones especiales.

4.3.4. Entorno de los jugadores

Aquí abordaremos todos los tipos de jugadores posibles, como cada jugador tiene unas características propias, tendremos una entidad por cada uno que guardará dichos valores para que el sistema pueda utilizarlos para sus cálculos.

El diagrama que representa el grupo de enemigos y el grupo de personajes jugables es el siguiente:



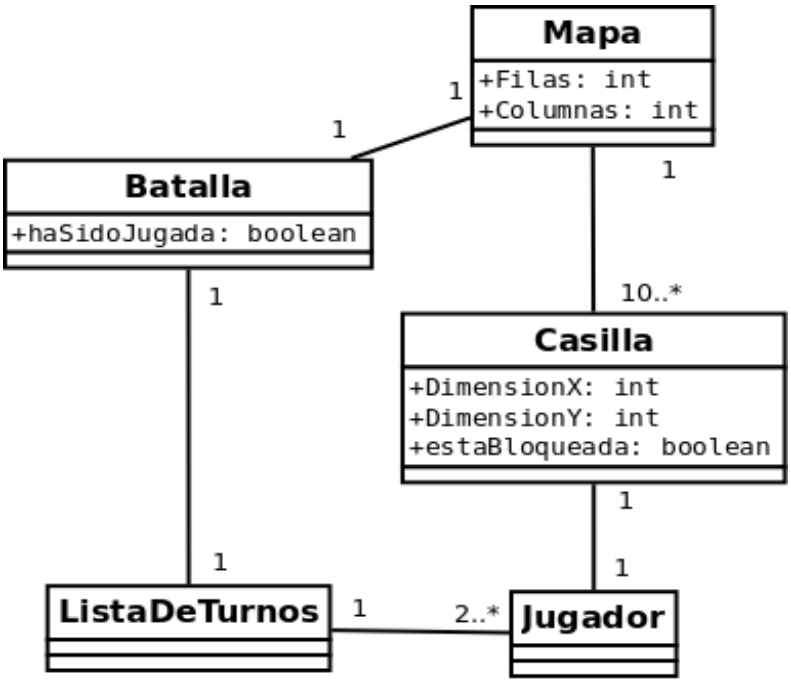
El diagrama tiene los siguientes elementos:

- Una entidad por cada personaje, como sus características y habilidades ya han sido descritas no se volverán a repetir.

4.3.5. Entorno de la batalla

Aquí se verán representados todos los elementos que conforman una batalla, para aclarar aún más el sistema de las mismas.

El diagrama que representa el entorno de la batalla es el siguiente:



Los elementos que forman el diagrama son:

- *Batalla*: el sistema deberá saber si el jugador ha jugado antes o no la batalla, ya que hay varias y se juegan de forma consecutiva. Hay que tener en cuenta que cada batalla tiene su mapa único asignado, y un número de batalla para saber el orden en que el jugador deberá jugarlo.

- *Características de batalla*: *haSidoJugada*, que nos sirve para saber si el jugador ha superado antes esta batalla.

- *Mapa*: cada batalla tiene lugar en un mapa, que tiene unas dimensiones en filas y columnas de casillas. Cada mapa está formado por un conjunto de casillas.

- *Casilla*: el elemento básico que forma los mapas, aquí almacenamos la dimensión de cada casilla.

- *Características de casilla*: hay una que es necesaria que sea conocida por el sistema llamada estaBloqueada, que sirve para saber si una casilla puede ser pisada por un jugador o no.

- *Lista de turnos*: explicada anteriormente, sirve para que el sistema sepa a qué personaje le toca jugar en cada momento en la batalla.

- *Jugador*: el papel que juega un jugador es formar parte de la lista de turnos y ocupar su espacio en una casilla.

4.4. AndEngine

Antes de continuar con el modelo de diseño, es muy importante hacer una sección para el motor gráfico ya que es la columna vertebral de la aplicación. La mayoría del tiempo de desarrollo se ha invertido en el estudio del motor gráfico, ya que es muy extenso y tiene muchas extensiones de las cuales se han elegido las necesarias.

4.4.1. Conceptos básicos de AndEngine

Antes de abordar las distintas extensiones y funciones que se han usado de AndEngine hay que cubrir los conceptos básicos del motor gráfico.

Los conceptos básicos que tiene AndEngine son:

- Engine

El *Engine* (motor) hace que el juego se ejecute en pequeños pasos discretos de tiempo. El Engine consigue sincronizar una actualización y un dibujo constante de la *Scene* (escena) que contiene todo el contenido que el juego esté manejando en ese momento.

EngineOptions se encargan de definir en qué orientación poner la pantalla, cargar la cámara y elegir si la visualización ocupará toda la pantalla o no del dispositivo.

- IresolutionPolicy

Una implementación de *IresolutionPolicy* es parte de *EngineOptions.* Le dice a AndEngine cómo operar con los diferentes tamaños de pantalla de los diferentes dispositivos.

Una de las funciones es *RatioResolucionPolicy*, que maximiza la vista de la superficie a las limitaciones de la pantalla, mientras mantiene un ratio específico. Eso quiere decir que los objetos no se distorsionarán mientras se tenga el tamaño máximo posible.

- Scene

La clase Scene es el contenedor raíz de todos los objetos que serán dibujados en la pantalla. La *Scene* tiene un número específico de Layers (capas), que pueden contener un número de *Entities* (entidades). Hay subclases, como la *CameraScene*/*HUD*/*MenuScene* que se dibujan ellas solas en la misma posición de la Scene sin importar dónde esté posicionada la cámara.

- Camera

Una *Camera* (cámara) definde el rectángulo de la escena que es mostrado en la pantalla, ya que toda la escena nunca es visible todo el tiempo. Normalmente hay una cámara por escena, excepto por las *SplitScreenEngines*. Hay subclases que permiten hacer zoom y suavizar cambios de posición de la cámara.

- Entity

Una *Entity* (entidad) es un objeto que puede ser dibujado, como los *Sprites* (gráficos), *Rectangles* (rectángulos), *Text* (texto) o *Lines* (líneas). Una entidad tiene posición /rotación /escala /color /etc...

- BitmapTextureAtlas

El *BitmapTextureAtlas* se puede intentar ver como un lienzo, donde se irán poniendo los TextureRegion que se vayan creando. Al ser la base de todas las imagenes debe tener un tamaño considerable que sea potencia de dos.

- TextureRegion

Una *TextureRegion* (región de textura) define un rectángulo en la textura. Una *TextureRegion* es usada por los Sprites para que el sistema sepa qué parte de la textura grande está mostrando el *Sprite*.

-Sprite

Un *Sprite* (gráfico) es un objeto de dos dimensiones que se coloca en la pantalla. La diferencia entre un Sprite y una imagen cualquiera es que el *Sprite* puede interactuar con el resto de elementos que carguemos.

- PhysicsConnector

Un *PhysicsConnector* (conector de física) se encarga de actualizar el AndEngine-Shapes (las formas: rectángulos, gráficos, etc..) cuando sus representaciones físicas llamadas "cuerpos" cambian.

Usando un Physics (y un *PhysicsConnector*) con un AndEngine-Shape se puede desactivar las Physics calculadas por AndEngine llamando *setUpdatePhysics(false)* con el *Shape*.

Los cambios hechos al AndEngine-Shape no se reflejan en los Physics – tienes que llamar a los métodos mediante el objeto *Body* (cuerpo) que has usado para crear el *PhysicsConnector*.

- SimpleBaseGameActivity

Una parte fundamental del juego ha sido el uso de esta clase, ya que la clase principal del juego extiende de ésta e implementa sus métodos de inicialización de motor gráfico y de escena.

Esta clase extiende a *BaseGameActivity*, que contiene el Engine y se encarga de crear un *SurfaceView* en el que se mostrarán los contenidos del Engine. Siempre hay exactamente un Engine por un *BaseGameActivity*. Se puede ir de un *BaseGameActivity* a otro usando mecanismos comunes de Android.

4.4.2. Carga de imágenes en AndEngine

Una parte importante de AndEngine es la carga de imágenes. En un principio podemos cargar imágenes en cada momento que la necesitemos, pero, al no estar estas almacenadas en memoria la aplicación puede llegar a ir extremadamente lenta.

La manera correcta de colocar imágenes mediante el motor gráfico es con el uso de *Sprites*. Para colocar un *Sprite* en pantalla necesitamos crear previamente un *BitmapTextureAtlas*, donde colocaremos todos los *TextureRegion*. Una vez creados los *TextureRegion* podremos crear los Sprites.

El código para crear un *BitmapTextureAtlas* es el siguiente:

```
private BitmapTextureAtlas mBitMapTextureAtlas;

...

BitmapTextureAtlasTextureRegionFactory.setAssetBasePath("gfx/
");

mBitMapTextureAtlas = new
BitmapTextureAtlas(this.getTextureManager(), 256, 256);
```

Lo primero que se tiene que hacer es asignar una ruta para que el motor gráfico busque las imágenes que se le pasan al crear el *TextureRegion*, después de generar el *BitmapTextureAtlas* se debe cargar en el motor gráfico. Para crear un *TextureRegion* se usa el siguiente código:

```
private ITextureRegion mDwarfTextureRegion;

...


mDwarfTextureRegion = BitmapTextureAtlasTextureRegionFactory.
createFromAsset(mBitMapTextureAtlas, this, "dwarfFront.png",
0, 0);
```

Como se ve en el código, al crear el *TextureRegion* ya estamos cargando la imagen deseada, los dos últimos número son los que posicionan la imagen dentro del atlas.

Y por último, para crear un *Sprite* usamos el siguiente código:

```
private Sprite mDwarfSprite;

...

mDwarfSprite = new Sprite(0, 0, mDwarfTextureRegion,
this.getVertexBufferObjectManager());
```

El *Sprite* se debe cargar en el *Scene* en el que vamos a trabajar, sino aunque esté creado estará almacenado en memoria, pero no se mostrará en pantalla.

A la hora de hacer desaparecer y reaparecer un Sprite no se quita y pone de la *Scene*. Si se usa el método *setVisible(false)* entonces el *Sprite* quedará cargado en memoria pero no podrá ser visto por el usuario, si se usa *setVisible(true)* volverá a aparecer.

4.4.3. El mando en pantalla

Una vez se ha cargado un *Sprite* en pantalla, el desarrollo requiere que el usuario pueda interactuar con el *Sprite*. Para permitir que el usuario mueva el *Sprite* se usa un mando digital en pantalla.

Para poner el mando en pantalla se deben usar los siguientes import:

```
import
org.andengine.engine.camera.hud.controls.BaseOnScreenControl;
import
org.andengine.engine.camera.hud.controls.BaseOnScreenControl
          .IOnScreenControlListener;
import
org.andengine.engine.camera.hud.controls.DigitalOnScreenContr
ol;
```

Una vez hechos los import, se creará un *TextureRegion* para la base del mando y para el "pomo" que se irá moviendo cada vez que pulsemos en una de las cuatro direcciones:

```
private BitmapTextureAtlas mOnScreenControlTexture;
private ITextureRegion mOnScreenControlBaseTextureRegion;
private ITextureRegion mOnScreenControlKnobTextureRegion;
    ....

this.mOnScreenControlTexture = new
BitmapTextureAtlas(this.getTextureManager(), 256, 128,
TextureOptions.BILINEAR);

this.mOnScreenControlBaseTextureRegion =
BitmapTextureAtlasTextureRegionFactory.createFromAsset
(this.mOnScreenControlTexture, this,
"onscreen_control_base.png", 0, 0);

this.mOnScreenControlKnobTextureRegion =
BitmapTextureAtlasTextureRegionFactory.createFromAsset
(this.mOnScreenControlTexture, this,
"onscreen_control_knob.png", 128, 0);
```

Una vez creados, se cargarán en el motor gráfico. Después se creará el mando con su constructor predefinido:

```
this.mDigitalOnScreenControl = new DigitalOnScreenControl(0,
CAMERA_HEIGHT -
this.mOnScreenControlBaseTextureRegion.getHeight(),
this.mBoundChaseCamera,
this.mOnScreenControlBaseTextureRegion,
this.mOnScreenControlKnobTextureRegion,
0.1f,this.getVertexBufferObjectManager(),
mOnScreenControlListener);
```

Los parámetros que se le pasan al método son:

1) La posición x del mando.

2) La posición y del mando.

3) La cámara que seguirá el mando. Es necesario que su valor sea distinto de null ya que sino el mando desaparecerá de la vista del usuario cada vez que se aleje del lugar donde esté el mando.

4) La *TextureRegion* de la base del mando.

5) La *TextureRegion* del pomo del mando.

6) Este valor define el tiempo que debe pasar entre cada vez que se pulsa el mando. Debe ser un valor no demasiado alto ya que sino el usuario tardará mucho en volver a interactuar con el mando.

7) Este valor es el *vertexBufferObjectManager* del motor con el que estemos trabajando.

8) El último atributo es el más importante, es una implementación de la clase *IonScreenControlListener*. Una vez implementado su método *onControlChange* le podremos dar una función al mando.

4.4.4. Hacer que un Sprite se mueva usando Physics

Una vez creado el mando para definir en qué direcciones se moverá el Sprite se procederá a crear un *PhysicsHandler*. El *PhysicsHandler* o "manejador" de física, es la clase encargada de atribuirle a un *Sprite* el comportamiento de un cuerpo físico real. Por tanto es capaz de definirle velocidad, movimiento, aceleración, etc.

Para crear un *PhysicsHandler* antes deberá haberse creado un *Sprite*, ya que el constructor requiere de un *Entity* para crearse. A parte tendremos que tener su import adecuado:

```
import org.andengine.engine.handler.physics.PhysicsHandler;

...

protected static PhysicsHandler mPhysicsHandler;

...
mPhysicsHandler = new PhysicsHandler(actualPlayerSprite);
actualPlayerSprite.registerUpdateHandler(mPhysicsHandler);
```

Se supondrá que el *actualPlayerSprite* es un objeto de tipo Sprite ya creado y se habrá probado que se muestre en pantalla.

Una vez se le pasa al método de la clase *Sprite registerUpdateHandler* el *PhysicsHandler* creado anteriormente el Sprite adquiere atributos de un cuerpo "real".

Aunque ya se haya hecho todo esto aún no se ha conseguido que el *Sprite* se mueva por la pantalla. Sólo se ha conseguido que el *Sprite* se pueda usar como un cuerpo físico.

Si se vuelve atrás en el código que se escribió sobre el mando, hay un atributo que se le pasó al constructor llamado *DigitalOnScreenControl*:

```
this.mDigitalOnScreenControl = new DigitalOnScreenControl(0,
CAMERA_HEIGHT -
this.mOnScreenControlBaseTextureRegion.getHeight(),
this.mBoundChaseCamera,
this.mOnScreenControlBaseTextureRegion,
this.mOnScreenControlKnobTextureRegion,
0.1f,this.getVertexBufferObjectManager(),
mOnScreenControlListener);
```

El atributo al que se quiere hacer hincapié es el llamado *mOnScreenControlListener*. Dicho atributo es una implementación de la interfaz IonScreenControlListener, que tiene un método llamado *onControlChange* que se implementará de la siguiente manera:

```
mOnScreenControlListener = new IOnScreenControlListener() {
            @Override
            public  void  onControlChange(BaseOnScreenControl
arg0,
                        float arg1, float arg2) {


      mPhysicsHandler.setVelocity(arg1*100,arg2*100);

                  }
            };
```

Lo importante de este código es el método setVelocity, que es el método que hace que el Sprite se mueva por pantalla. Los atributos que se le pasan al método son las coordenadas de los ejes x e y del mando.

Para conocer el funcionamiento del mando se debe saber que existe un vector con las coordenadas x e y, los valores del vector dependen de qué dirección se pulse:

- Arriba(0,1)
- Abajo(0,-1)
- Derecha(1,0)
- Izquierda(-1,0)

4.4.5. Mover un Sprite con un Path o recorrido

La forma definitiva que se ha escogido para mover a los personajes en Orcs and Tactics es mediante un recorrido o Path.

La clase *Path* se encuentra en el paquete *PathModifier*. Esto se debe a que, para que un *Sprite* haga un recorrido mediante un *Path*, se requiere registrar un *PathModifier* como *entityModifier*.

El constructor de un *Path* requiere definir la longitud del recorrido previamente. Después de definir la longitud si se quiere que el recorrido lleve a alguna parte se debe usar el método *to(float x, float y)* de la clase *Path*.

Aquí hay un ejemplo de cómo hacerlo:

```
final Path path = new Path(5).to(10, 10).to(10, CAMERA_HEIGHT - 74).to(CAMERA_WIDTH - 58, CAMERA_HEIGHT - 74).to(CAMERA_WIDTH - 58, 10).to(10, 10);
```

En este caso es un recorrido con 5 cambios de ruta. En esos puntos el recorrido cambia de sentido, o debe ser así, si un recorrido es en línea recta no se debería usar más de dos veces el método to. Esto se debe a que la primera vez que se llama a este método se asigna el punto de inicio del recorrido y la segunda vez y posteriores se asignan los puntos a los que va a llegar. Los recorridos con Path en Orcs and Tactics son todos en horizontal o en vertical, nunca en diagonal.

Una vez creado un *Path* se debe registrar un modificador para el *Sprite*. El modificador que se debe usar es *PathModifier*, concretamente se usa el siguiente constructor:

```
public PathModifier(final float pDuration, final Path pPath, final IPathModifierListener pPathModifierListener)
```

Los parámetros son:

1) La duración del recorrido. Debe ser menor a un segundo para que el usuario no experimente una jugabilidad lenta.

2) El recorrido que se ha creado anteriormente.

3) Un Listener de tipo *PathModifier*. Su uso se describirá más abajo.

Una vez definido todo eso, se procede a usar el método registerentitymodifier definido para la entidad Sprite. A ese método se le pasa como entrada el *PathModifier* creado anteriormente.

- La interfaz IPathModifierListener

El uso de la interfaz *IpathModifierListener* es el siguiente:

Orcs and Tactics es un juego en el que los personajes deben desplazarse casilla a casilla. Para esto cada vez que el usuario pulsa el joystick hacia una dirección la dirección nueva es igual a las coordenadas del mando multiplicadas por el tamaño del lado de cada casilla ya que son cuadradas (32 píxeles por 32). Hay evitar que, mientras el personaje se desplace por la pantalla, el jugador cambie el recorrido a mitad de ejecución. Para eso se define un valor tipo boolean que se pone a false al inicio del recorrido, y mientras sea false, se bloquea el joystick para el usuario.

El valor *boolean* usado se debe poner a *true* al final del recorrido. Esto se hace con una implementación de la interfaz *IpathModifierListener*, específicamente, de su método:

```
public void onPathFinished(final PathModifier pPathModifier, final IEntity pEntity)
```

Sólo se necesita poner el boolean a true dentro de este método, y así se volverán a desbloquear los mandos para el usuario.

El uso de esta interfaz es fundamental para que el juego se desarrolle como es debido.

4.4.6. El programa Tiled

Antes de proceder con la explicación de la extensión usada para cargar mapas de casillas en la escena hay que definir qué programa se ha usado para generar dichos mapas.

La aplicación se llama *Tiled* y su uso se basa en usar un archivo de imagen que tiene las casillas dibujadas. Al crear un mapa nuevo se define el tamaño de la casilla, por lo que el programa procede a dividir la imagen que se le pasa en tantas casillas como se permita.

Una vez dibujado el mapa, se le podrá definir una propiedad a una de las casillas plantilla que usamos. Si se define por ejemplo la propiedad "wall" con valor "true" se podrá recoger después dicho valor para hacer lo oportuno con la casilla. Al definir una propiedad en una casilla de la plantilla todas las casillas de ese tipo que estén pintadas en el mapa adquirirán esa propiedad.

4.4.7. La extensión TMXTiledMap

Una parte fundamental de la creación de un RPG táctico es el mapa. El mapa debe estar formado por casillas a ser posibles con la misma altura que anchura para que sean cuadradas. La perspectiva en la que se trabajará será la superior, aunque la extensión también admite mapas generados con perspectiva isométrica.

Los import que se usarán de la extensión TMX serán:

```
import org.andengine.extension.tmx.TMXLayer;
import org.andengine.extension.tmx.TMXLoader;
import org.andengine.extension.tmx.TMXTiledMap;
import
org.andengine.extension.tmx.util.exception.TMXLoadException;
```

Para que funcione correctamente la carga de un mapa tmx se deberá tener todos esos import. Para conocer un poco más el uso de la extensión se explicarán los conceptos más relevantes:

1) *TMXLoader* contiene los métodos de carga del mapa para luego generar un *TMXTiledMap*.

2) *TMXTiledMap* es lo que generamos con el *TMXLoader*, es el mapa completo que se le pasa como ruta al *TMXLoader* por lo que contiene todas las capas. Aunque se cargue un *TMXTiledMap* la escena no mostrará nada ya que no se ha seleccionado la capa a mostrar.

3) El *TMXLayer* es la capa del mapa que se cargará. Un mapa puede tener varias capas, y si se quiere que la misma escena cuente con capas distintas se deberá cargar la adecuada en cada momento deseado. Una vez seleccionada la capa se deberá cargar en la escena para ser visualizada.

Un ejemplo del código de carga de mapa es el siguiente:

```
try{

      final TMXLoader tmxLoader = new
TMXLoader(this.getAssets(),
      this.mEngine.getTextureManager(),
      TextureOptions.BILINEAR_PREMULTIPLYALPHA,
                  this.getVertexBufferObjectManager());
      this.mTMXTiledMap =
tmxLoader.loadFromAsset("rutaDelMapa.tmx");
      }catch(final TMXLoadException e){
            Debug.e(e);
      }

      mTMXLayer = mTMXTiledMap.getTMXLayers().get(0);
      mScene.attachChild(mTMXLayer);
```

Se suponen ya creados los atributos *mTMXTiledMap*, *mTMXLayer* y *mScene*.

Para que podamos entender la carga del mapa mejor se explicará el proceso:

1) Se crea una variable de tipo *TMXLoader* con la que se cargarán el AssetManager de nuestro motor gráfico, junto con el *VertexBufferObjectManager* y unas opciones de textura. En este caso las opciones de textura son BILINEAR_PREMULTYPLYALPHA, lo que permite que el mapa tenga un Alpha Multipass que mejora el renderizado del mapa.

2) Al haber creado las opciones que tendrá el mapa, se procede a cargarlo desde su ruta con el método loadFromAsset.

3) Es importante capturar cualquier excepción posible para solucionar lo antes posible futuros conflictos en la carga.

Ya se tiene el mapa creado, lo que queda es seleccionar la capa deseada de todas las capas disponibles. El método *getTMXLayers* devuelve una lista con todas las capas del mapa, en este caso se selecciona la primera.

TMXLayer es una entidad, así que podrá ser cargada como cualquier otra con el método *attachChild*.

Si se desea reconocer las características de las casillas de nuestro mapa el código que cambia es al crear el *tmxLoader*, pero también se deberán importar las siguientes clases:

```
import
org.andengine.extension.tmx.TMXLoader.ITMXTilePropertiesListe
ner;
import org.andengine.extension.tmx.TMXProperties;
import org.andengine.extension.tmx.TMXTile;
import org.andengine.extension.tmx.TMXTileProperty;


...
```

El código para conseguir sacar las casillas con cierta propiedad es:

```
try {
      final       TMXLoader       tmxLoader       =       new
TMXLoader(this.getAssets(),
      this.mEngine.getTextureManager(),
```

```
      TextureOptions.BILINEAR_PREMULTIPLYALPHA,
      this.getVertexBufferObjectManager(),                new
ITMXTilePropertiesListener() {
      @Override
      public     void     onTMXTileWithPropertiesCreated(final
TMXTiledMap pTMXTiledMap,     final TMXLayer pTMXLayer, final
TMXTile pTMXTile, final       TMXProperties<TMXTileProperty>
pTMXTileProperties) {



      if(pTMXTileProperties.containsTMXProperty("wall",
"true")) {
            //Aquí irá el código a implementar

                  }
            }
      });
```

Como se puede ver, la variación respecto al código con el que sólo cargamos un mapa es el método de carga. La variación es crear una implementación de la interfaz *ITMXTilePropertiesListener* y después implementar su método *onTMXTileWithPropertiesCreated*.

Para saber si una casilla tiene una propiedad se usa el método *containsTMXProperty*. Después de saber si una casilla tiene propiedades se procederá a implementar el código que se requiera para operar sobre dichas casillas o almacenarlas para un futuro uso.

4.4.8. El algoritmo de la A estrella en AndEngine

El método principal del algoritmo de la A estrella en AndEngine necesita para su correcto funcionamiento los siguientes recuros:

```
            public Path findPath(final IPathFinderMap<T> pPathFinderMap, final int pXMin, final int pYMin, final int pXMax, final int pYMax, final T pEntity, final int pFromX, final int pFromY, final int pToX, final int pToY, final boolean pAllowDiagonal, final IAStarHeuristic<T> pAStarHeuristic, final ICostFunction<T> pCostFunction, final float pMaxCost, final IPathFinderListener<T> pPathFinderListener)
```

- Una implementación de IpathFinderMap<T> y de su método *isBlocked*. Al implementarlo se debe definir de qué manera se sabrá si una casilla está bloqueada o no. La mejor forma de implementarlo es con una lista de casillas bloqueadas del mapa que se identificarán mediante su propiedad ya que las casillas libres no tienen ninguna propiedad añadida.

- Los cuatro siguientes enteros son los límites de la capa del mapa. Es fundamental que estos límites estén bien fijados o el algoritmo se saldrá de la capa.

- Una entidad, en este caso, la capa del mapa que se esté usando.

- Los cuatro siguientes valores de tipo entero son el inicio y el final del camino.

- El siguiente valor es un *boolean* que define si se va a permitir desplazamiento en diagonal o no.

- Una implementación de una heurística. AndEngine viene con las heurísticas más comunes implementadas.

- La función de coste. Dicha función define el coste para moverse a cualquiera de las casillas adyacentes. Es una parte importante del algoritmo ya que el recorrido se elige según los costes.

- El coste máximo del recorrido. Si sobrepasa este coste el algoritmo se parará, es un número que está hecho para evitar recorridos demasiado largos.

- Una implementación de *IPathFinderListener* que tiene un método:

```
  public void onVisited(final T pEntity, final int pX, final
int pY);
```

La implementación del método permitirá definir un código que se ejecutará cuando se visite una casilla, pero es opcional.

Una vez hecho todo lo anterior, el nuevo recorrido será el devuelto por el método de la clase *AstarPathFinder*. El *Path* resultante no es un *Path* como el que se le registra a un *Sprite* con *PathModifier*. Este *Path* es específico del paquete algorithm de AndEngine.

La forma que tiene AndEngine de usar este *Path* es introduciendo en el mismo dos listas con los valores X e Y de forma consecutiva que la entidad debe tener para hacer el recorrido en cuestión.

Para hacer que un *Sprite* se mueva con el *Path* resultante basta con volcar todos los datos del *Path* del algoritmo dentro de un *Path*. Esto se hace con un bucle for:

```
final int astarPathLength = astarPath.getLength();
Path tPath = new Path(astarPathLength);
float tileSpeed = astarPathLength / 32;

for(int i = 0; i < astarPathLength; i++){
      tPath.to(astarPath.getX(i),astarPath.getY(i));
}
```

De las variables creadas antes del bucle cabe destacar un recorrido básico, que aunque no tenga una llamada al método to, tiene un tamaño de recorrido asignado.

La variable *tileSpeed* sirve para asignar una velocidad que depende de la longitud del recorrido, aunque es opcional.

Una vez terminado el bucle for se usa lo siguiente:

```
mDwarfSprite.registerEntityModifier(new
PathModifier(tileSpeed, tPath));
```

Donde *mDwarfSprite* es el Sprite que vamos a mover en cuestión.

4.4.9. Botones táctiles

Se explicará la manera de implementar un botón táctil en pantalla, que tenga una imagen asociada y unas funciones al pulsarlo. Además, se explicará la forma en la que el botón siga a la cámara para que no se quede fuera de la parte que el usuario esté viendo en todo momento.

Para el código que se ha desarrollado en el proyecto, hacen falta dos import:

```
org.andengine.entity.sprite.ButtonSprite

org.andengine.engine.camera.hud.HUD
```

- El primero es para el tipo ButtonSprite, que sirve para que un Sprite pueda tener las funciones de un botón táctil.

- El segundo es la clase utilizada para que el botón táctil siga a la cámara.

Para explicar el código se supondrá creados ya un BitmapTextureAtlas y un TextureRegion ya explicados para el botón.

El código que crea el ButtonSprite es:

```
mRedButtonSprite = new ButtonSprite(CAMERA_WIDTH –
mSpriteRedButtonTextureRegion.getWidth(), CAMERA_HEIGHT –
mSpriteRedButtonTextureRegion.getHeight(),
mSpriteRedButtonTextureRegion,
getVertexBufferObjectManager(), mOnClickListener);
```

1) Los dos primeros valores sitúan el botón en la esquina abajo derecha de la pantalla.

2) Segundo valor es el *TextureRegion* del *Sprite*.

3) El *vertexbufferObjectManager* del Engine.

4) Una implementación de la interfaz interna de ButtonSprite llamada *OnClickListener.*

La interfaz *OnClickListener* tiene un método:

```
public void onClick(final ButtonSprite pButtonSprite, final
float pTouchAreaLocalX, final float pTouchAreaLocalY);
```

En el caso descrito anteriormente, su implementación es *mOnClickListener*.

En el código de Orcs and Tactics los botones táctiles abarcan la mayoría del código en ejecución del juego. Esto se debe a que es la única manera a parte del joystick que el usuario tiene de interactuar con el juego. En todos los botones táctiles se tienen en cuenta varias variables booleanas que controlan el estado de los comandos que el jugador introduce.

El siguiente paso es añadir el botón táctil a la cámara:

```
HUD = new HUD();
hud.attachChild(mRedButtonSprite);
hud.registerTouchArea(mRedButtonSprite);
this.mBoundChaseCamera.setHUD(hud);
```

La explicación de las líneas de código es:

1) Se crea un objeto de tipo HUD para operar con él.

2) Se usa el método attachChild para que el botón se quede fijado en el movimiento del hud.

3) Se registra el área táctil. Si no se hace esto aunque se haya implementado el método onClick el botón no responderá a la pulsación.

4) Suponiendo ya creada una cámara *mBoundChaseCamera* que es de tipo persecución (sigue a un elemento) sólo queda añadir el hud a la cámara.

De lo que hay que asegurarse antes de acoplar el botón al hud es que no está acoplado al scene, porque puede dar error ya que un *Sprite* sólo puede tener un padre.

4.4.10. Animaciones

Una vez creados los Sprites y se quiera pasar a la fase de animación debemos tener en cuenta que se implementan de una forma distinta a los Sprites normales.

Para explicar las animaciones en AndEngine debemos imaginar las "posturas" en las que se puede encontrar un Sprite. Es decir, en el caso de un RPG táctico de vista superior como Orcs And Tactics, el personaje andará en las cuatro direcciones sin diagonales.

Hay que tener un diseño para cada dirección en la que se mueve el personaje. Además hay que tener otros diseños para emular el movimiento del personaje, una imagen con un pie levantado y otra con el otro pie levantado. Cuando el personaje se mueva el orden será: pie levantado, dos pies quietos, otro pie levantado. Esto nos permitirá dar la impresión de que el personaje mueve los pies mientras anda.

Al final lo que hay que generar es una imagen que tenga todas estas imagenes contenida. El propio motor gráfico mediante el código que se usa las dividirá. Como son cuatro direcciones y tres posturas en total deberemos tener doce imágenes dentro de una misma.

Para crear el textureRegion de un AnimatedSprite el código cambia comparado con lo visto hasta ahora:

```
this.mSnapdragonTextureRegion = BitmapTextureAtlasTextureRegionFactory.createTiledFromAsset(this.mBitmapTextureAtlas, this, "snapdragon_tiled.png", 4, 3);
```

Como se puede ver, el cambio es el método create de *BitmapTextureAtlasTextureRegionFactory*, que ahora pasa a ser *createTiledFromAsset*. Esto se debe a que la imagen después se dividirá en trozos dependiendo de los dos últimos valores que le pasamos al método. Estos valores definen el número de filas y columnas en la que se dividirá la imagen. El resto de valores del método son los mismos que los descritos en la carga de imágenes en un *BitmapAtlas*.

El constructor para un *AnimatedSprite* es casi idéntico al de un *Sprite*:

```
final AnimatedSprite snapdragon = new
AnimatedSprite(300, 200,
this.mSnapdragonTextureRegion,
this.getVertexBufferObjectManager());
```

Sólo cambia el tipo de Sprite y el nombre del Constructor.

Para animar la imagen en pantalla se usa la función animate:

```
snapdragon.animate(100);
```

Este código anima el *Sprite* de tal forma que se muestran todas las imagenes contenidas dentro de la imagen que se le pasara como entrada durante ese tiempo cada una.

Pero en Orcs and Tactics eso no es lo deseado. Lo que se necesita es que según la dirección a la que mire el personaje. Esto se consigue poniendo el siguiente código dentro del registro del *entityModifier*:

```
mDwarfAnimatedSprite.registerEntityModifier

(new        PathModifier(tileSpeed,        tPath,        new
PathModifier.IPathModifierListener() {

      public      void      onPathWaypointStarted(PathModifier
pPathModifier,
            IEntity pEntity, int pWaypointIndex) {
```

```
                switch (pWaypointIndex) {
                      case 0:

      mDwarfAnimatedSprite.animate(new long[] { 200,
                            200, 200 }, 0, 2, true);
                            break;

                      case 1:

      mDwarfAnimatedSprite.animate(new long[] { 200,
                            200, 200 }, 6, 8, true);
                            break;

                      case 2:

      mDwarfAnimatedSprite.animate(new long[] { 200,
                            200, 200 }, 9, 11, true);
                            break;

                      case 3:

      mDwarfAnimatedSprite.animate(new long[] { 200,
                            200, 200 }, 3, 5, true);
                            break;

                            }
                }

                public                                    void
onPathWaypointFinished(PathModifier pPathModifier,
                            IEntity       pEntity,       int
pWaypointIndex) {
                }

                public    void    onPathStarted(PathModifier
pPathModifier, IEntity                          pEntity) {
                }

                public   void   onPathFinished(PathModifier
pPathModifier, IEntity                          pEntity) {
                      mDwarfAnimatedSprite.stopAnimation();
                }
           }));
```

Así, según la dirección a la que se mire, se animará con una serie de imagenes u otra durante 200 milisegundos cada una.

Al final del recorrido se debe parar la animación para que no siga dando la impresión de que se mueve el personaje.

4.4.11. Textos dinámicos y TimerHandler

Una de las principales características de los RPG tácticos es que se muestran los datos de todo lo que hace el jugador por pantalla. Estos datos pueden ser el daño provocado a otro jugador o jugadores, las características del jugador, la vida restante que tiene, etc.

Se han usado los textos para representar:

1)El daño que se ha hecho con alguna acción a otro jugador o si esa acción ha fallado.

2)Una ventana con todos los datos de un jugador cuando llega su turno y los de otros jugadores cuando se pasa por encima el recuadro rojo para apuntar a un objetivo.

3) El movimiento restante que tiene un personaje mientras se mueve por la pantalla.

Para ambos usos se ha creado un texto que se modifica cada vez que obtenemos nuevos datos.

En un principio la opción de usar textos dinámicos fue descartada. Pero en Julio de 2012 el autor de AndEngine, Nicholas Grammlich, actualizó la clase Text para que se pudiera cambiar un texto sin tener que crear un nuevo. Además, se rentabilizó el coste de las operaciones de escritura en objetos de tipo texto, para que ya no sobrecargaran tanto el Engie. El efecto final ha sido que no se reducen los FPS (Frames Per Second) y no se dan "saltos" durante la experiencia de usuario.

La diferencia entre los dos primeros casos es que en el segundo caso el texto desaparece o cambia si se mueve el recuadro rojo por la pantalla. En el primer caso el texto aparece durante un tiempo determinado y luego desaparece. En el tercer caso el texto desaparece en cuanto otro de los textos aparece, pero reaparece cuando el personaje se mueve.

- Crear un texto y cambiar su valor

Se va a explicar la forma en la que se carga una fuente y se crea un texto. Lo primero es cargar una fuente:

```
this.mFont = FontFactory.create(this.getFontManager(),
this.getTextureManager(), 256, 256, TextureOptions.BILINEAR,
Typeface.create(Typeface.DEFAULT, Typeface.BOLD), 48);
this.mFont.load();
```

Este método es original de openGL. OpenGL (open graphics library) es una especificación estándar que define una API multilenguaje y mutliplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. Una de sus funciones es la de crear una fuente. En este caso AndEngine está creando una fuente como si fuera una textura de 256 por 256 pixeles, en negrita y de tamaño de fuente 48.

Después para generar un texto sólo hay que utilizar el código siguiente:

```
final Text elapsedText = new Text(100, 160, this.mFont, "Seconds elapsed:", "Seconds elapsed: XXXXX".length(), this.getVertexBufferObjectManager());
```

El texto del ejemplo estará colocado en la posición x e y de los dos primeros atribuos del método. El tercer atributo es la fuente que se ha creado antes. El cuarto es el texto a poner y el siguiente es el tamaño máximo que tendrá el texto. El último es el *vertexBufferObjectManager* del Engine que está en uso.

Una vez creado un *Text* con una cadena de texto para ser representado en pantalla para cambiarlo sólo hay que usar el método *setText*:

```
elapsedText.setText("Seconds elapsed: " + secondsElapased );
```

El cambio de texto se puede usar igual que *System.out.println* y a la cadena se le puede añadir cualquier valor que se tenga en uso. Hay que tener en cuenta que dicho valor debe tener un método *toString* definido o ser compatible con *String*: *int, double, char, float, etc.*

Ya se ha cubierto la fase de creación y edición de texto. Un texto de ser añadido como hijo de una Scene o una *HUD* para ser representado en pantalla con el método *attachChild*.

- TimerHandler

Antes de poder seguir explicando cómo se han usado los textos en Orcs and Tactics es necesario definir cómo se ha usado el *TimerHandler*.

Para el primer caso es necesario que el daño hecho entre jugadores se representara sólo durante unos segundos. Para cubrir esa necesidad se ha utilizado un *TimerHandler*. En este caso se ha usado un handler para que, al terminar un determinado tiempo desaparezca el texto.

A la hora de hacer que desaparezca el texto se usa la función *setVisible*. Esto se hace así para no tener que generar un texto nuevo cada vez que se muestre un texto distinto, así se aprovecha la memoria de la que se dispone y se evitan problemas con el recolector de basura.

El constructor usado para el *TimerHandler* es el siguiente:

```
public TimerHandler(final float pTimerSeconds, final boolean pAutoReset, final ITimerCallback pTimerCallback)
```

Donde:

1) Los segundos que deben pasar hasta que se active el handler.

2) Si se debe autoresetear solo o no. En este caso se desactiva el autoreset. Esto se debe a que si no se hace así, cada vez que el contador llegue al valor que se le ha pasado como entrada se volverá a poner a cero. Si esto se permite, entonces estará actualizando el valor del texto de forma constante durante el tiempo, provocando fallos o bugs a la hora de ver el texto.

3) Una implementación de *ItimerCallback*.

La interfaz *ItimerCallback* tiene un método que se debe implementar para que el texto desaparezca:

```
public void onTimePassed(final TimerHandler pTimerHandler);
```

Al implementar el método sólo hay que poner la visibilidad del texto a false. Con esto cada vez que el contador llegue al tiempo que se ha definido el texto desaparecerá.

Hay que recordar que como cualquier otro handler hay que registrarlo con *registerUpdateHandler*, en este caso, en el texto deseado.

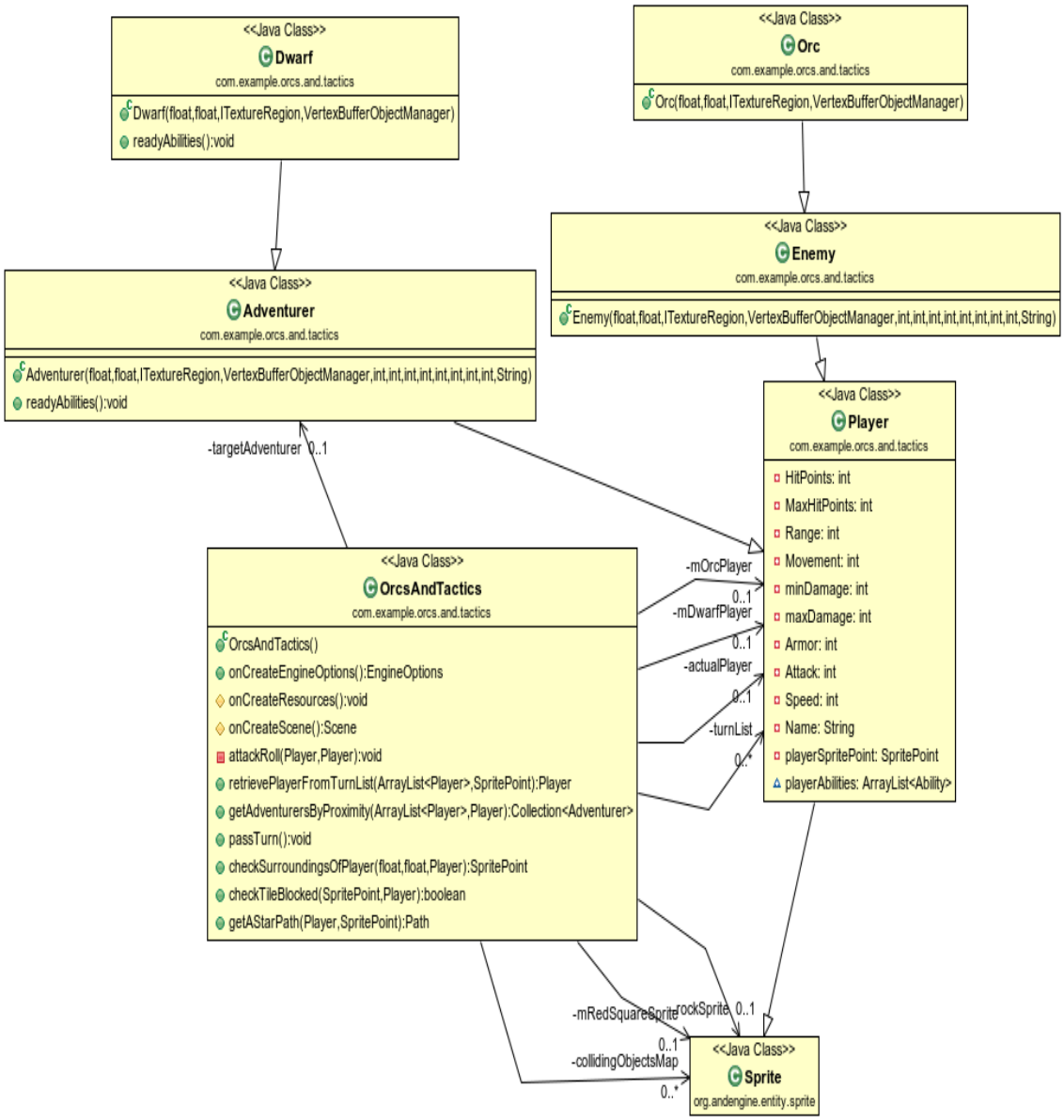
4.5. Modelo de diseño

Una vez terminado el modelo de análisis, se procede a realizar el modelo de diseño. Aquí se representarán todas las clases que tendrá el programa.

Las clases serán descritas una a una, al igual que los métodos y las partes del código relevantes que lo conforman, para poder entender al completo lo que se ha hecho durante el desarrollo de la aplicación.

4.5.1. Diagrama de clases

El diagrama de clases de Orcs and Tactics ha sido creado a partir de las ideas descritas en el diagrama de análisis:



Se procederá a explicar las clases del diagrama:

- *MainActivity*: la clase principal del juego, extiende la clase de AndEngine *SimpleBaseGameActivity* y carga todos los elementos que se representan en el juego.

- *Player*: almacena todos los datos del jugador, a parte de los descritos en el modelo de análisis almacena listas con sus habilidades y tiene algunos métodos que se explicarán posteriormente.

- *Enemy*: una clase hija de *Player*, sirve para distinguir a los enemigos de los personajes jugables (*Adventurer*).

- *Adventurer*: una clase hija de *Player* con el mismo funcionamiento que la clase *Enemy*.

- *Orc*: una clase hija de *Enemy*, nos rive para poder inicializar todos los valores del jugador Orco definido en el modelo de análisis.

- *Dwarf*: una clase hija de *Adventurer*, nos sirve para poder inicializar todos los valores del personaje jugable Gwail el Guerrero definido en el modelo de análisis.

4.5.2. Colisiones

Como en cualquier videojuego, es necesario que el comportamiento del mismo sea "real". Lo que quiere decir es que un personaje no debería atravesar a otro ni salirse del escenario en el que está.

Al hecho de que un personaje no pueda atravesar a otro o no pueda pasar sobre casillas bloqueadas se le llama colisión.

Para controlar las colisiones en Orcs and Tactics se han usado comprobaciones a cada paso que da un jugador. Sólo ha habido que mirar si la siguiente casilla a la que el jugador se quiere mover está ocupada por un bloqueo o por otro jugador.

Todo se ha conseguido iterando a cada paso que da un jugador por una lista con el resto de jugadores. Esto se ha introducido como el siguiente método que devuelve un jugador si está en la lista y sino un jugador null:

```
public Player retrievePlayerFromTurnList(ArrayList<Player>
tL, SpritePoint sp){
	int i = 0;
	boolean found = false;
	Player tP = null;
	Player targetPlayer = null;
	while(i<tL.size() && !found){
		tP = tL.get(i);
		if(tP.getX() == sp.getX() && tP.getY() ==
sp.getY()){
			targetPlayer = tP;
			found = true;
		}
		i++;
	}
	return targetPlayer;
}
```

El motivo por el que se ha introducido un jugador null es para controlar si una casilla está no ocupada, y que se debe devolver un objeto de tipo *Player*, sea null o no.

Este método se usa tanto para saber si una casilla está bloqueada por un jugador como para atacar a otro jugador con el botón rojo.

Además, se tiene en cuenta que la casilla esté bloqueada también por un bloque de piedra. Dichos bloques de piedra están almacenados en un *HashMap*, estructura que permite el acceso de orden 1 (constante) a cualquiera de sus posiciones. Dicha implementación de *Map* contiene una lista de parejas clave-valor que contienen la posición en el mapa de un objeto y el objeto en sí. A la hora de saber si una casilla está o no bloqueada sólo hay que mirar primero si está bloqueada por un jugador y si no es el caso mirar si está bloqueada por un elemento que esté contenido en el mapa en cuestión.

El único problema del uso del *HashMap* como alternativa a una lista para almacenar los objetos bloqueantes es el siguiente: a la hora de comprobar si existe una clave asociada a un valor la clave que se le pasa al método *get(Object key)* del mapa debe ser igual e idéntica a la clave que buscamos. Para que esto funcione se ha implementado la clase *SpritePoint*.

- La clase SpritePoint

La clase *SpritePoint* se utiliza para crear puntos y almacenarlos en el mapa de objetos bloqueantes. Se descartó el uso de un array de *float* ya que a la hora de comparar si una clave era igual a otra la clase *float* genera un código hash único para cada *float*. Esto desencadena que, aunque se le pase como valor un array de *float* con los mismos valores que uno que ya esté contenido en el mapa, la clase *HashMap* devuelve *null*. Esto se debe a que a la hora de buscar dicho valor lo hace mediante su código hash, y al ser este único de cada objeto, considera que no existe otro idéntico.

Para solucionar dicho problema se ha hecho un Override (sobrecarga) de los métodos de la clase *Object hashCode* y *equals*. Como la clase *HashMap* usa ambos métodos para buscar y comparar claves hay que controlar que compare como se requiere dos objetos. El código de ambos métodos es:

```
@Override
public boolean equals(Object o){
	SpritePoint sp = (SpritePoint)o;
	return sp.getX()==posX && sp.getY()==posY;
}
@Override
public int hashCode(){
	return (int)(17*posX+19*posY);
}
```

Como se puede ver, cada *SpritePoint* tiene un *hashCode* que depende de los valores almacenados en su interior. Se multiplica por dos números primos para hacer único cada código para dos direcciones de posición. El método equals compara directamente las posiciones X e Y de un *SpritePoint* por lo que si dos puntos tienen las mismas coodenadas son iguales.

Una vez hecho esto se puede usar el método *get* de *HashMap* sin problemas.

4.5.3. Lista de turnos y método passTurn

Para resolver el problema de los turnos de los personajes se ha creado una lista que contiene a todos los jugadores. Para pasar el turno se itera sobre dicha lista y se asigna la variable global *actualPlayer* a esa nueva posición. Además la cámara persecutoria sigue a ese nuevo jugador.

Todo esto lo controla el método *passTurn*:

```
public void passTurn(){
            buttonsBlocked = true;
            if(indexOfTurnList == turnList.size()-1){
                  indexOfTurnList = 0;
            }else{
                  indexOfTurnList ++;
            }
            actualPlayer = turnList.get(indexOfTurnList);
            mBoundChaseCamera.setChaseEntity(actualPlayer);
            tempMovement = actualPlayer.getMovement();
            buttonsBlocked = false;
      }
```

Dentro de este método se colocará la inteligencia artificial. La variable *buttonsblocked* sirve para bloquear los botones de pantalla.

4.5.4. Inteligencia Artificial

Una vez cubierta la parte en la que el usuario interactúa con el juego hay que cubrir la parte en la que la máquina toma lugar. Como en los juegos de su estilo, en Orcs and Tactics también hay que definir una Inteligencia Artificial o IA.

La IA es una parte fundamental de cualquier videojuego, ya que es la única manera de que el usuario pueda jugar sólo al mismo. De lo que se va a encargar esta parte es de tomar decisiones según la situación en la que se encuentra un enemigo en cierto momento. En nuestro caso, dichas decisiones dependerán de a qué distancia estan los jugadores de un enemigo.

Uno de los comportamientos que puede tener un enemigo es atacar al jugador más cercano, y este será el criterio de acciones que tendrá. En el momento que un usuario termina su acción, es decir, que ataca o usa una habilidad, le tocará el turno al siguiente *Adventurer* o *Enemy*. Si es el caso de *Enemy* (a partir de ahora enemigo) entonces se ejecutará la IA, y buscará qué jugador controlado por el usuario hay más cercano (*Adventurer* o aventurero) y le atacará.

Encontrar al jugador más cercano no supone un gran problema. Pero definir qué recorrido hará el enemigo sí lo es. El movimiento de los aventureros los define el usuario casilla a casilla, pero el enemigo no puede hacer eso ya que su movimiento debe ser "automático".

La manera de resolver el problema del recorrido a seguir por el enemigo es usando la implementación que AndEngine tiene del algoritmo de la A estrella. Dicho algoritmo devolverá el recorrido más corto hasta el punto más cercano al jugador al que se quiere llegar.

Pero antes de poder usar el algoritmo hay que resolver una duda muy importante: la casilla a la que el enemigo debe moverse para atacar a un aventurero. Para resolver esta duda hay que tener en cuenta:

a) El rango de alcance del enemigo, si tiene rango 1 sólo podrá atacar a las unidades adyacentes a su casilla pero si tiene más podrá atacar en ángulo y a casillas lejanas.

b) El movimiento del enemigo, el número de casillas a recorrer por el enemigo no debe ser mayor que el movimiento del enemigo, o no podrá llegar.

Antes de empezar con el algoritmo hay que tener en cuenta el caso mejor: que el enemigo no tenga que moverse porque tenga a un jugador a su alcance. En este caso no entrará en el algoritmo de cálculo de movimiento y atacará al jugador más cercano. Esto se consigue mediante el código:

```
if((targetAdventurer.getX()==(actualPlayer.getX()+32))
&&(targetAdventurer.getY()==actualPlayer.getY())

                        ||
(targetAdventurer.getX()==(actualPlayer.getX()-32)) &&
      (targetAdventurer.getY()==actualPlayer.getY())

                        ||
(targetAdventurer.getY()==(actualPlayer.getY()-32))    &&
      (targetAdventurer.getX()==actualPlayer.getX())

                        ||
(targetAdventurer.getY()==(actualPlayer.getY()+32))    &&
      (targetAdventurer.getX()==actualPlayer.getX())){
                              adyacentToEnemy = true;
                        }
```

Este código va dentro del bucle *while* que itera sobre los jugadores en torno al enemigo.

Lo primero que se hace es comprobar qué jugador está más cerca del enemigo.Esto se consigue mediante un método que devuelve una colección con todos los jugadores en orden de proximidad. La distancia se ha calculado mediante el método convencional:

```
public Collection<Adventurer> getAdventurersByProximity(
                  ArrayList<Player> turnList, Player
                        actualPlayer) {
            Player tempPlayer = null;
            TreeMap<Float,Adventurer>
                  adventurerSortedMapByProximity = new
                  TreeMap<Float, Adventurer>();
            Float tempKey;
            for(int i = 0; i<turnList.size(); i++){
                  tempPlayer = turnList.get(i);
                  if(tempPlayer instanceof Adventurer){
                        Float xdist = actualPlayer.getX()-
                              tempPlayer.getX();
                        Float ydist = actualPlayer.getY()-
                              tempPlayer.getY();
                        tempKey = (float)
                              Math.sqrt(Math.pow(xdist,
                                    2)+Math.pow(ydist, 2));

                        adventurerSortedMapByProximity.put
                        (tempKey,(Adventurer)tempPlayer);
                  }
            }
            return adventurerSortedMapByProximity.values();
      }
```

Esto sirve para saber a qué casilla deberá moverse. Para cubrir todas las posibilidades hay que restar las coordenadas X e Y del objetivo a las del enemigo, así tenemos los casos: arriba, derecha arriba, izquierda arriba, abajo, derecha abajo, izquierda abajo, derecha e izquierda.

El código principal que controla las opciones y decide qué hacer en cada una es:

```
while(i<listOfAdventurersSortedByProximity.size()&&!
adyacentToEnemy){

                        targetAdventurer =

      listOfAdventurersSortedByProximity.get(i);

      if((targetAdventurer.getX()==(actualPlayer.getX()+32))

      &&(targetAdventurer.getY()==actualPlayer.getY())
                                  ||
(targetAdventurer.getX()==(actualPlayer.getX()
                        -32)) &&
(targetAdventurer.getY()==actualPlayer.getY())
                                  ||
(targetAdventurer.getY()==(actualPlayer.getY()
                        -32)) &&
(targetAdventurer.getX()==actualPlayer.getX())
                                  ||
(targetAdventurer.getY()==(actualPlayer.getY()
      +32)) &&
(targetAdventurer.getX()==actualPlayer.getX()))
      {
                                             adyacentToEnemy =
                                             true;
                        }else{
                              xdistance =
                              targetAdventurer.getX() -
                              actualPlayer.getX();
                              ydistance =
```

```
                targetAdventurer.getY() -
                      actualPlayer.getY();
                targetPlayerMoveTileForEnemy =

          checkSurroundingsOfPlayer(xdistance,
                            ydistance,
                            targetAdventurer);
                i++;
          }
    }
    if(adyacentToEnemy){
          attackRoll(actualPlayer,
                targetAdventurer);

    }else if(targetPlayerMoveTileForEnemy!
          =null){
          Path path =

          getAStarPath(actualPlayer,

          targetPlayerMoveTileForEnemy);

           actualPlayer.registerEntityModifier
    (new PathModifier(path.getLength()/64,
    path, new IPathModifierListener() {

                public void
          onPathWaypointStarted(PathModifier
                      pPathModifier,
                      IEntity pEntity, int
                      pWaypointIndex) {
                }
                public void
          onPathWaypointFinished(PathModifier
                      pPathModifier,
                      IEntity pEntity, int
                      pWaypointIndex) {
                }
                public void
          onPathStarted(PathModifier
                      pPathModifier, IEntity
                      pEntity) {
                }
                public void
          onPathFinished(PathModifier
                      pPathModifier, IEntity
                      pEntity) {
                      attackRoll(actualPlayer,
```

```
                                        targetAdventurer);
                            }
                        }));
                }else
                    if(targetPlayerMoveTileForEnemy==null){
                      passTurn();
                }
        }
```

Dicho código se sitúa en el método *passTurn*() y sólo se ejecuta si el nuevo jugador es de tipo enemigo mediante el código:

```
if(actualPlayer instanceof Enemy)
```

Una vez cubiertos los casos hay que decidir qué hacer en cada uno. En cada caso hay que seleccionar la o las casillas más cercanas para el enemigo. Por ejemplo, si el objetivo está abajo a la izquierda el enemigo deberá intentar moverse a la casilla justo a la derecha o justo arriba del objetivo. En caso de tener rango de alcance mayor que 1 basta con moverse lo justo para poder atacar a su objetivo con su rango máximo.

En el caso de que las casillas más favorables estén ocupadas el enemigo procederá a intentar atacar al siguiente jugador más cercano. Si ese jugador ya está fuera del rango de movimiento del enemigo, dejará de buscar objetivos y se pasará el turno al siguiente personaje. Si el siguiente personaje fuera enemigo se repite de nuevo el algoritmo.

En el caso en que el enemigo se desplace hasta la casilla en la que pueda atacar a otro jugador el enemigo ataca al jugador. Después de eso se pasa el turno al siguiente personaje.

Una vez obtenida la casilla a la que el enemigo debe moverse se procede a calcular el Path del algoritmo de la A estrella. Una vez calculado se vuelca sobre un *Path* de *Pathmodifier* y se aplica al enemigo para moverse. El código es:

```
public Path getAStarPath(Player actualPlayer, SpritePoint
        targetPlayerMoveTileForEnemy){
           IAStarHeuristic<TMXLayer> heuristic = new

                 ManhattanHeuristic<TMXLayer>();
           IPathFinder<TMXLayer> finder = new
           AStarPathFinder<TMXLayer>();
           final org.andengine.util.algorithm.path.Path
           astarPath = finder.findPath(new
                 IPathFinderMap<TMXLayer>() {
                 public boolean isBlocked(int pX, int pY,
                       TMXLayer pEntity) {
                       boolean blocked = false;
                       for(Player p: turnList){
                             if(p.getX()==pX &&
                                   p.getY()==pY){
                                   blocked = true;
                                   break;
                             }
                       }
                       if(collidingObjectsMap.get(new
                             SpritePoint(pX,pY))!=null){
                             blocked = true;

                       }
                       return blocked;
                 }
           }, 0, 0, (int)layerWidth, (int)layerHeight,
                 mTMXLayer,(int)actualPlayer.getX(),
                 (int)actualPlayer.getY(),
                 (int)targetPlayerMoveTileForEnemy.getX(),

                 (int)targetPlayerMoveTileForEnemy.getY(),
                 false, heuristic,new
                 ICostFunction<TMXLayer>() {

                       public float
                       getCost(IPathFinderMap<TMXLayer>
```

```
                        pPathFinderMap,
                        int pFromX, int pFromY, int
                        pToX, int pToY, TMXLayer
                              pEntity) {
                              return 0;
                        }
                  });

      final int astarPathLength =
            astarPath.getLength();
      Path tPath = new Path(astarPathLength);
      for(int i = 0; i < astarPathLength; i++){

tPath.to(astarPath.getX(i),astarPath.getY(i));
      }
      return tPath;
}
```

Este código es el que usa el algoritmo de la A estrella y calcula un recorrido en base a los valores que recibe. Los métodos *getCost* y *isBlocked* se han explicado con anterioridad. Una vez terminado se procede a atacar al objetivo.

4.5.5. El sistema de habilidades

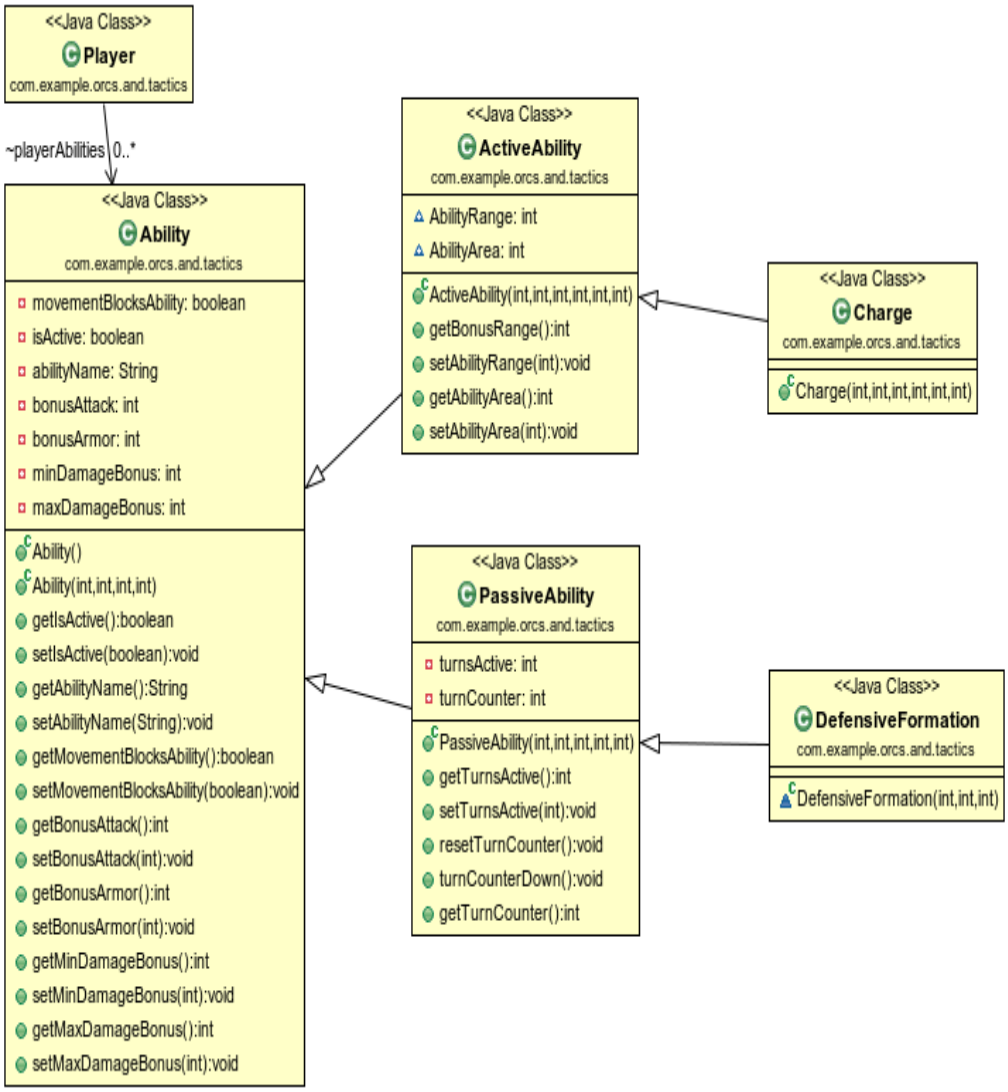
El sistema de habilidades cubre todas las necesidades a la hora de crear y controlar las habilidades de los personajes. Las necesidades son:

a) Habilitar una habilidad y usarla debendiendo de su tipo: activa o pasiva.

b) Controlar en qué momento (turno) termina la habilidad que esté habilitada.

c)Aplicar los atributos de una habilidad dependiendo del caso.

Lo primero es mostrar el diagrama de clases que controla esto:


<<Java Class>>
Player
com.example.orcs.and.tactics
~playerAbilities 0..*
<<Java Class>>
Ability
com.example.orcs.and.tactics
movementBlocksAbility: boolean
isActive: boolean
abilityName: String
bonusAttack: int
bonusArmor: int
minDamageBonus: int
maxDamageBonus: int
Ability()
Ability(int,int,int,int)
getIsActive():boolean
setIsActive(boolean):void
getAbilityName():String
setAbilityName(String):void
getMovementBlocksAbility():boolean
setMovementBlocksAbility(boolean):void
getBonusAttack():int
setBonusAttack(int):void
getBonusArmor():int
setBonusArmor(int):void
getMinDamageBonus():int
setMinDamageBonus(int):void
getMaxDamageBonus():int
setMaxDamageBonus(int):void
<<Java Class>>
ActiveAbility
com.example.orcs.and.tactics
AbilityRange: int
AbilityArea: int
ActiveAbility(int,int,int,int,int,int)
getBonusRange():int
setAbilityRange(int):void
getAbilityArea():int
setAbilityArea(int):void
<<Java Class>>
Charge
com.example.orcs.and.tactics
Charge(int,int,int,int,int,int)
<<Java Class>>
PassiveAbility
com.example.orcs.and.tactics
turnsActive: int
turnCounter: int
PassiveAbility(int,int,int,int,int)
getTurnsActive():int
setTurnsActive(int):void
resetTurnCounter():void
turnCounterDown():void
getTurnCounter():int
<<Java Class>>
DefensiveFormation
com.example.orcs.and.tactics
DefensiveFormation(int,int,int)

Como se puede ver en el diagrama, hay una división entre habilidades pasivas y activas, al igual que en el diagrama de análisis.

No existe el caso en el que una habilidad pasiva y activa bonifique al mismo atributo de un personaje, ya que la condición de activar una habilidad pasiva es pasar el turno. En algunos casos hay que controlar que un personaje se haya movido o no, dependiendo de la habilidad que sea. Esto se controlará mediante una variable booleana que se le pasará como condición a la activación de dicha habilidad.

Para aplicar las habilidades se sigue el procedimiento:

a) Se crea un botón táctil amarillo que selecciona la habilidad pasiva del personaje en cuestión si se pulsa una vez y la activa si se pulsa por segunda. Si falta alguna de las dos sólo elige la que hay y si el personaje no tiene ninguna el botón no hace nada.

b) En el momento que se elige la habilidad el jugador debe pulsar el botón rojo para que se active dicha habilidad. En caso de que la habilidad sea pasiva pasará turno y activará la habilidad. En caso de que sea activa pasará al modo de selección de objetivo como si estuviera atacando y usará la habilidad. Previamente hay que reconocer las condiciones de la habilidad para poder activarla, esto dependerá del caso.

c) Si la activada es pasiva y dura un turno basta con desactivarla al inicio del turno del jugador. Si dura más de uno se guardará en una variable que será única del jugador y se usará de contador para desactivar la habildiad cuando sea necesario.

En el momento que se activa una habilidad los métodos *get* de la clase *Player* aplican los cambios. Como una habilidad tiene una variable que almacena si está o no activada cuando su valor cambia a true se le suma o resta dependiendo del caso un número al atributo del jugador. Estos números dependen de la habilidad que se active. Además de comprobar que una habilidad esté o no activa los métodos get también comprueban que el contador interno de la habilidad sea mayor o igual que cero, es decir, que la habilidad dure activa más turnos que el actual.

Así se consigue no tener que cambiar o crear un nuevo método de sistema de dados, usando siempre el mismo sistema para que los jugadores intercambien ataques entre ellos.

5. Análisis temporal y de costes de desarrollo

5.1. Análisis temporal

Fecha	Horas	Tipo	Tarea
Septiembre 2011	20	Diseño	Diseño de personajes
Octubre 2011	15	Análisis	Búsqueda y comparación de motores gráficos
Enero 2012	10	Desarrollo	Preparación de entorno de desarrollo
Febrero 2012	5	Análisis	Búsqueda y comparación de algoritmos de recorrido
8 y 9 Marzo 2012	10	Desarrollo	Hackaton de AndEngine
Abril 2012	20	Análisis	Creación de sistema de juego
Mayo 2012	5	Análisis	Revisión de sistema de juego
Junio 2012	20	Desarrollo	Aprendizaje de uso de mapas TMX en AndEngine
15 Julio 2012	10	Desarrollo	Descarga de nueva rama de AndEngine, extensiones y ejemplos
17 Julio 2012	4	Documentación	Introducción, definición de objetivos
18 Julio 2012	4	Diseño	Diseño de mapas propios
19 Julio 2012	5	Desarrollo	Prueba de carga de

			mapas TMX propios
20 Julio 2012	6	Documentación	Estudio de alternativas
27 Julio 2012	6	Desarrollo	Estudio de formas de carga de una imagen
28 Julio 2012	4	Desarrollo	Prueba de carga de imagen en mapa
29 Julio 2012	6	Desarrollo	Estudio del algoritmo A estrella proporcionado por AndEngine
31 Julio 2012	4	Análisis	Análisis de requisitos
31 Julio 2012	4	Documentación	Análisis de requisitos
1 Agosto 2012	4	Análisis	Modelado de análisis
2 Agosto 2012	3	Documentación	Modelado de análisis
3 Agosto 2012	4	Desarrollo	Estudio del motor gráfico
4 Agosto 2012	4	Documentación	Descripción de los elementos que conforman el juego
5 Agosto 2012	6	Desarrollo	Estudio de funcionamiento del mando en pantalla
8 Agosto 2012	8	Desarrollo	Movimiento del personaje mediante el mando
9 Agosto 2012	6	Desarrollo	Estudio del movimiento del personaje mediante el algoritmo A estrella
10 Agosto 2012	3	Documentación	Estudio de mercado
11 Agosto 2012	3	Desarrollo	Estudio de texturas y mapas de bits
12 Agosto 2012	6	Desarrollo	Prueba de movimiento en bucle de personaje
13 Agosto 2012	5	Desarrollo	Creación de clases
15 Agosto 2012	5	Documentación	Paso de modelado de análisis a imagen con el editor de diagramas
16 Agosto 2012	4	Documentación	Sistema de habilidades, movimiento y rango de ataque
17 Agosto 2012	4	Desarrollo	Prueba de movimiento

			mediante recorrido sencillo
18 Agosto 2012	6	Desarrollo	Prueba de movimiento mediante recorrido de la A estrella
20 Agosto 2012	3	Desarrollo	Carga de personajes a partir de sus clases
22 Agosto 2012	4	Desarrollo	Uso de un sprite para que los personajes interactuen entre ellos
24 Agosto 2012	1	Documentación	Análisis y diseño
25 Agosto 2012	1	Documentación	Diseño e implementación
26 Agosto 2012	3	Documentación	Análisis temporal
28 Agosto 2012	4	Documentación	Análisis de costes
30 Agosto 2012	2	Documentación	Estudio de alternativas
1 Septiembre 2012	2	Documentación	Otras alternativas
2 Septiembre 2012	2	Documentación	Maquetación
septiembre 2012	10	Documentación	Maquetación
septiembre 2012	5	Desarrollo	Paso de turnos y lista de turnos.
octubre 2012	10	Documentación	Maquetación
octubre 2012	10	Desarrollo	Inteligencia Artificial
octubre 2012	5	Desarrollo	Implementación de método de probabilidad
noviembre 2012	10	Desarrollo	Sistema de habilidades
noviembre 2012	5	Documentación	Inteligencia Artificial
noviembre 2012	5	Documentación	Sistema de habilidades
noviembre 2012	2	Diseño	Aprendizaje de uso básico de Inkscape
noviembre 2012	2	Diseño	Diseño de personajes para mostrar en pantalla
noviembre 2012	5	Documentación	Maquetación
Total de horas:	320		

5.2. Análisis de costes

Tarea	Tiempo(h)	Coste unitario (€/h)	Coste total (€)
Analista	53	15	795
Desarrollador	146	8	1168
Diseñador	28	15	420
Documentalista	93	6	558
Total			2941

5.3. Estudio de mercado

Llevo más de un año observando el flujo de juegos para móviles, las ideas más originales, las que más beneficios aportan y la manera de conseguirlos. Mi objetivo nunca fue un videojuego comercial a gusto de cualquier persona, sino un juego dirigido a esas personas que les siguen gustando los juegos de 16 bits y que disfrutan jugándolos.

Aún así, los juegos tipo RPG (role-playing game o juego de rol) tienen un lugar pequeño entre la cantidad de juegos del mercado de smartphone, esto es algo bueno y malo: un juego que se haga de ese tipo puede hacerse famoso antes porque sea uno de los pocos de ese tipo o no tener éxito ninguno, lo cual hace que el éxito dependa de la demanda que haya en el momento. Aunque existen bastantes juegos RPG para smartphone, su precio suele ser elevado para un formato digital (entre 10 y 15 euros) incluso siendo de compañías independientes.

Si pensáramos en hacer negocio mediante la distribución de un producto software para smartphone tendríamos en cuenta las siguientes posibilidades:

1.-Vender el juego por un precio que nos parezca no abusivo. Si es demasiado barato no se consiguen beneficios y si es demasiado caro los usuarios no lo comprarán o lo comprarán demasiado poco. Ésta opción es la menos recomendada ya que últimamente la piratería ha aumentado en Google Play, publicando versiones gratuitas de juegos de pago, y es recomendable tener en cuenta otras opciones.

2.- Tener una versión gratuita y otra de pago. La versión gratuita tendrá menos prestaciones que la de pago: ya sea no poder guardar tu partida o puntuación, tener menos niveles, menos personajes donde elegir, etc. Para desbloquear las nuevas prestaciones tendremos que pagar por la versión completa.

3.- La publicidad, se puede presentar de tres formas distintas:

3a.- Integrada en un juego gratuito que esté en versión completa. La publicidad se mostrará siempre, pero el juego será gratuito, los beneficios saldrán exclusivamente de la publicidad.

3b.- Pagando para quitar la publicidad. Otra forma de obtener beneficios es poner dos versiones del producto: una gratuita con publicidad y otra de pago sin publicidad.

3c.- Combinar el pago para quitar la publicidad con una versión de prueba. La opción más completa sería tener dos versiones del producto: una gratuita con publicidad y menos prestaciones, y otra sin publicidad con el juego completo. Esto haría que la versión de pago fuera más apetecible para el usuario, ya que vería la recompensa de pagar por la versión completa.

Esto son sólo algunas de las opciones que podemos tener en cuenta, realmente el éxito o fracaso de un juego depende en gran medida de la viralidad que se consiga mediante Intenet y las redes sociales, que cada vez más representan un medio de distribución de publicidad para las empresas que no pueden permitirse grandes anuncios o que no tengan un legado que se remonte a tantos años atrás.

6. Comparativa con otras alternativas

6.1. Software Libre y Licencia Libre

El software libre es una denominación de software que defiende la libertad de los usuarios sobre su producto, permitiéndoles no sólo usarlo sino modificarlo y redistribuirlo libremente. Esto no quiere decir que el software sea de dominio público, ya que el software tiene licencia de autor.

La licencia libre, cada día más usada, es una licencia que no cuesta dinero comprar. Tiene ciertas restricciones que se imponen en el producto, yendo desde la más libre de todas: se puede copiar, distrubir, modificar, vender sin permiso y sin mención al autor; hasta la más restrictiva: no se puede copiar ni modificar pero sí se puede distribuir. Es posible incluso llegar a acudir a vías legales si no se siguen las restricciones dictadas por la licencia.

Sin duda una gran ventaja que tiene Android es que una creciente comunidad de software libre de desarrolladores que siguen los ideales de compartir el código de lo que crean aportando riqueza al mundo del desarrollo de software.

Esta idea está mal vista en muchas compañías, sobre todo las grandes corporativas, que defienden la exclusividad de sus tecnologías y sus productos, siendo el mercado de los videojuegos para las consolas muy cerrado al igual que el acceso al desarrollo para las mismas. Este no es el caso de otras compañías, llamadas Indie (Independent o independientes) que desarrollan videojuegos normalmente porque son unos apasionados de los mismos, o porque quisieron formar una empresa por sí solos sin depender de las grandes compañías. Es común que las compañías Indie apoyen el software libre y lo usen, alimentando la comunidad con ideas normalmente originales y únicas.

6.2. IOs contra Android

Otra alternativa posible habría sido usar el sistema operativo para Smartphone de Apple: IOs. Aunque es un sistema operativo consolidado y bastante potente tiene dos grandes desventajas para los desarrolladores de aplicaciones: el lenguaje que usa es más específico que Java y que hasta hace poco la comunidad de desarrolladores era muy cerrada.

Todo esto se debía a que Apple tenía una política de privacidad tan fuerte que no se podía ni hablar del SDK en foros o en páginas personales como blogs o redes sociales. Esto ha desencadenado en que hay muy poca documentación, ejemplos de código o frameworks de otros usuarios disponibles en la red.

Por esa razón y porque hay que pagar unos 99 dólares anuales para tener acceso al SDK de IOs y poder subir aplicaciones a la appStore quedó desestimada como opción para desarrollar esta aplicación para este sistema.

7. Pruebas

Al ser Orcs and Tactics un videojuego, las pruebas forman una parte fundamental del desarrollo del mismo.

A cada pequeño paso que se ha dado en el desarrollo se ha ido probando exhaustivamente la aplicación. Esto ha sido posible gracias a que se puede cargar la aplicación en el teléfono desde Eclipse.

Además se ha podido ir cambiando el entorno de la aplicación e idear nuevas formas de lidiar con los problemas. Esto se debe a que un videojuego para Smartphone debe ser dinámico y lo más divertido posible. Parte de esa diversión se debe a que el movimiento de los elementos sea un poco más rápido, que el aprendizaje de uso sea el más sencillo posible o que la interfaz de usuario sea simple.

8. Conclusiones y desarrollos futuros

Como se puede suponer echando un vistazo a la aplicación, para que llegue a ser un producto listo para el mercado le falta más trabajo. Ahora mismo está en fase de demostración pero la idea principal siempre ha sido continuar con su desarrollo.

Parte del desarrollo futuro es completar el diseño de todos los personajes, de las lógicas de Inteligencia Artificial, de las habilidades y permitir la subida de niveles de los personajes. Todo esto va a ser posible gracias a que el sistema se ha ideado para ser lo más reutilizable posible por lo que sólo faltaría crear el contenido descrito arriba y crear menús para la selección de niveles por el usuario.

Otra gran parte del desarrollo futuro es la posibilidad de jugar en línea entre dos jugadores. AndEngine ya tiene una extensión que permite conectar dos dispositivos mediante Bluetooth lo cual ahorra bastante trabajo.

9. Bibliografía

Autor	Título	Referencia	Fecha de consulta
RealMayo	Getting Started With AndEngine	http://www.andengine.org/forums/tutorials/getting-started-with-andengine-t4858.html	Junio 2012
Ahmet Öztürk	Free Software	http://cisn.metu.edu.tr/2002-6/free.php	Julio 2012
Patrick Lester	A * Pathfinding para principiantes	http://www.policyalmanac.org/games/articulo1.htm	Agosto 2012
Ed Burnette	Introducción a Android	Programación Android, editorial Anaya	