Escuela
Politécnica
Superior

Creación de video-juego Rogue-Like

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Vicente Román Ibáñez

Tutor/es:
Carlos Villagrá Arnedo

Fidel Aznar Gregori

Universitat d'Alacant
Universidad de Alicante

Junio 2014

Creación de Videojuego Rogue-Like

UNIVERSIDAD DE ALICANTE

CURSO 2013-2014

DCCIA

Tutores

Carlos Villagrá Arnedo

Fidel Aznar Gregori

Alumno

Vicente Román Ibáñez

Índice general

Capítulo 1

Introducción

1.1. Aleatoriedad en videojuegos

La aleatoriedad ha formado parte de los videojuegos desde hace muchos años, concretamente desde 1978, cuando apareció Space Invaders, incluyendo por primera vez el factor de aleatoriedad como parte de su jugabilidad, haciendo que la cadencia de disparo de los enemigos e incluso alguno de sus movimientos fueran aleatorios y haciendo que al jugador le fuera más difícil prever el siguiente movimiento de la máquina, lo que añadía un extra a la jugabilidad del juego y permitía prolongar el tiempo que un jugador podía seguir interesado en él.

Hasta la aparición de dicho juego, la aleatoriedad no formaba parte de los videojuegos y todos los juegos usaban otro tipo de técnicas para ofrecer cierta jugabilidad al usuario, como en el caso del Pong, en el que la máquina podía calcular el movimiento necesario para golpear la bola, o en juegos como el ajedrez, nim, tic-tac-toe o damas, donde se exploraba el árbol de soluciones hasta un cierto nivel para ofrecer un movimiento al jugador.

Si bien es cierto que, en juegos como los mencionados no necesitan la aleatoriedad para ofrecer al usuario la jugabilidad esperada, en otro tipo de géneros de videojuegos como en RPG, puzzle, ..., puede ser un factor útil para extender la vida útil del juego como en el caso de los RPG o incluso un factor completamente necesario como en algunos

juegos puzzle, solo hay que imaginarse una máquina de Tetris donde siempre aparezcan las piezas en el mismo orden y pensar en cuanto tiempo tardaríamos en dejar de jugar a ese juego en particular.

Dentro del género RPG existen videojuegos que hacen uso de la aleatoriedad hasta cierto punto, por ejemplo usándola sólo para generar objetos, otros no la usan para nada, haciendo que el juego pierda prácticamente todo el sentido de ser jugado más de una vez, y otros que hacen uso de la aleatoriedad para un gran número de sus características (generación de mapas, items e incluso historia), permitiendo así que su rejugabilidad sea interesante.

En este proyecto se ha procedido a implementar un juego de tipo rogue-like desde cero, donde la rejugabilidad sea el factor principal y la aleatoriedad el modo de conseguirlo.

1.2. Objetivos del proyecto

Se desarrollará un videojuego para PC de tipo roguelike, y se centrará principalmente en la generación de contenido aleatorio del mismo en diferentes partes del proyecto, aunque no es competencia de este proyecto el que el juego esté completo en su totalidad, si no que sea jugable y pueda mostrar los elementos que lo componen como la interfaz de usuario y principalmente la generación de contenido aleatorio del juego, para lo que se usarán ciertos elementos como sonidos, música o fondos de pantallas que no son propiedad del autor de este trabajo de final de carrera y que, en caso de usarse una parte o la totalidad del presente proyecto en futuros proyectos comerciales, deberán ser removidos para evitar problemas legales.

Para lograr el objetivo, diseñará e implementará un sistema de interfaz GUI que permitirá al usuario interactuar con el videojuego y viceversa mediante sus elementos y navegar entre diferentes pantallas y menús del juego, y que será reutilizable en futuros proyectos en los que sea necesaria una interfaz de usuario en entornos gráficos.

El universo virtual que conforma los diferentes mapas del juego, será generado de manera aleatoria al comienzo de la partida y esos mapas serán diferentes cada partida

nueva que se quiera jugar. Habrá un personaje principal que controlará el usuario mediante el ratón y que haciendo uso de diferentes habilidades disparadas con el teclado, podrá eliminar los enemigos que aparecerán de manera aleatoria en el mapa creado también de manera aleatoria, avanzando en su aventura mapa tras mapa hasta llegar al mapa del enemigo final, y ganando experiencia/habilidades sobre la marcha, creando una sensación de avance progresivo durante el transcurso de la partida.

Para poder representar ese universo virtual, se realizará un estudio de las diferentes proyecciones posibles disponibles para su visualización y se seleccionará una de ellas que será la que mejor se ajuste a las necesidades del proyecto.

Se realizará un estudio previo también de los diferentes algoritmos presentes en la actualidad para la generación de mapas aleatorios de los diferentes tipos soportados por el proyecto y se analizarán y seleccionarán los que se ajusten mejor a los requerimientos.

Otros elementos como la generación de objetos de manera aleatoria, algoritmos de búsqueda de rutas o pathfind, generación de enemigos aleatorios, inteligencia artificial básica etc. serán módulos que se tendrán en cuenta en la elaboración del videojuego.

1.3. Metodología

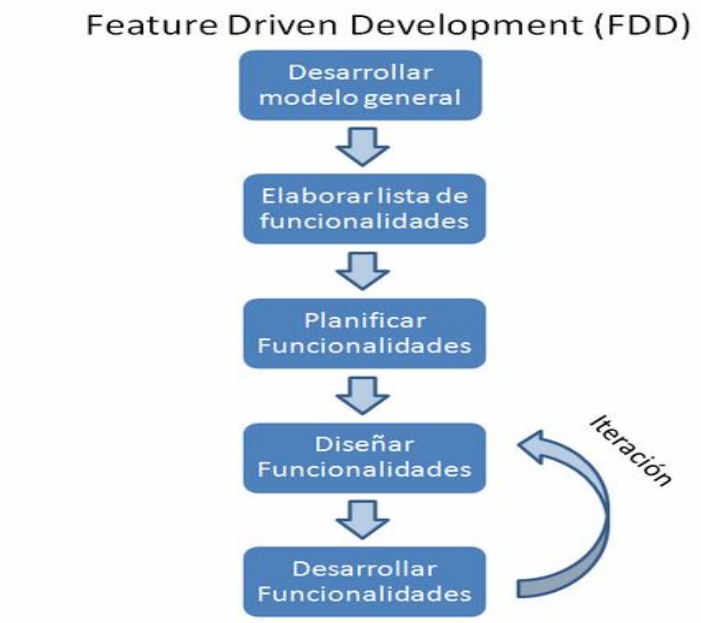
El desarrollo de un videojuego engloba ciertas limitaciones en la planificación como definición de requisitos débiles o poco específicos, largos periodos de construcción del proyecto que dificultan la planificación temporal, la necesidad de mucho aprendizaje y la incertidumbre de si ciertas partes van a ser viables o no al inicio. Por todo esto se ha decidido usar como metodología de trabajo una metodología ágil ya que se ajusta con la mayoría de restricciones comentadas.

Dentro de las metodologías ágiles hay varias opciones disponibles como Scrum, xP, ASD, FDD y muchas otras, pero se ha seleccionado FDD (Feature-Driven Development), debido a que el juego dispone de muchos módulos separados y se pueden ir añadiendo al sistema como features en cada iteración.

Hay que tener en cuenta que aunque se haya usado una metodología ágil, estas suelen estar preparadas para trabajo en equipo, y dado que este trabajo se ha desarrollado de manera individual no se han seguido todas las características de la metodología al pie de la letra.

Le metodología es iterativa e incremental y se compone de 5 fases, 3 de ellas iniciales en las que se crea una visión general del proyecto y se obtiene una lista de funcionalidades ordenadas por prioridad y las dos últimas fases en las que se itera por cada una de las funcionalidades de la lista y en las que se diseña y se implementa esa funcionalidad.

Las reuniones con el cliente (en este caso se entiende como 'cliente' a los tutores del TFG) se han realizado en periodos variables entre 1 y 3 semanas en los que se mostraban los resultados finales de la funcionalidad programada para la iteración y se recogía el feedback para modificar si fuera necesario y posteriormente se iniciaba la siguiente iteración hasta que el producto estuviera terminado.



Fases de la metodología FDD.

La aplicación de la metodología puede observarse con más detalle en el capítulo 10, donde se analizan las horas dedicadas al proyecto, así como su distribución por módulos que han servido como base para la organización a base de features de la metodología.

Capítulo 2

Estado del arte

2.1. Juegos Roguelike

Los videojuegos roguelike son un subgénero de los juegos de rol RPG (Role Playing Games) basados en el juego *Rogue* que apareció en 1980 para sistemas Unix. Rogue estaba basado en las reglas del juego de mesa D&D (Dungeons and Dragons) y la idea principal es que el jugador explore varios niveles de mazmorras, peleando contra enemigos durante el camino para ganar experiencia y equipamiento de manera progresiva.

Dentro de las características que definen con más detalle a este subgénero, se encuentran las siguientes:

· Niveles de mazmorra generados aleatoriamente, aunque pueden incluir zonas estáticas, como habitaciones o niveles de enemigos finales que puedan ser invariantes.

· Las propiedades mágicas de un objeto son variables, una espada corta puede ser distinta a otra espada corta en la misma partida así como en partidas distintas.

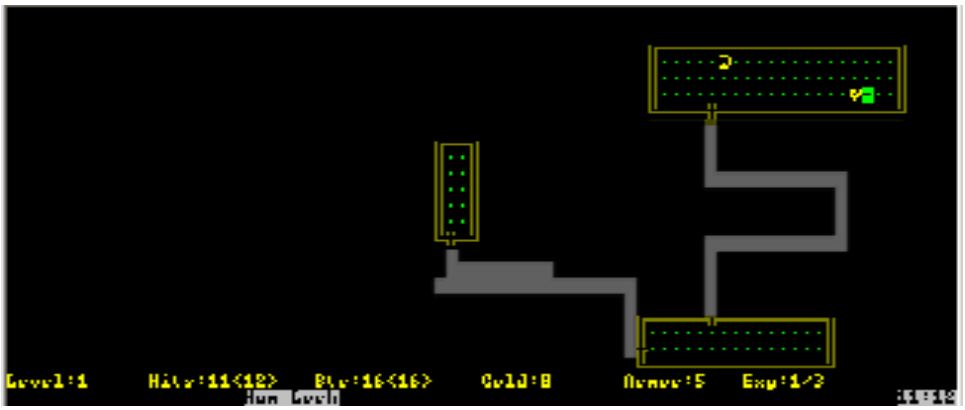
· El sistema de combate es basado en turnos, generalmente basado en pasos, cuando un jugador efectúa un movimiento, se recalculan los movimientos del resto de objetos.

· La mayoría suelen ser de un solo jugador.

· Permadeath, cuando un personaje muere, no puede resucitar y deberá comenzar una nueva partida.

Existen diferentes modificaciones de estas reglas que derivan en nuevos subtipos del género, como en el juego *Diablo* de 1996, donde la muerte permanente solo está implementada en el modo 'Hardcore', mientras que se puede jugar en el modo 'Softcore' donde la muerte conlleva una penalización de experiencia y oro, pero no la pérdida del personaje caído. Otra de las diferencias de este subgénero llamado A-RPG (Action-RPG) es que la dinámica del combate no está basada en turnos como los roguelike, si no que los movimientos son en tiempo real, y los enemigos pueden acercarse al jugador sin que éste tenga que realizar un movimiento para que los movimientos de los enemigos sean recalculados.

En los ordenadores antiguos se usaba una representación del juego mediante una matriz de caracteres ASCII, donde unos símbolos representaban muros, otros el interior de las habitaciones, al personaje, enemigos, cofres, etc. incluyendo otros factores de identificación como el color del carácter para, por ejemplo, representar la dificultad del enemigo o la calidad de un objeto. Un ejemplo de este tipo de visualización se puede ver en el juego antes mencionado *Rogue*:



Captura del videojuego *Rogue* (1980)

Con el aumento constante de la capacidad en los ordenadores de sobremesa, se han implementado a lo largo de la historia diferentes visualizaciones para el género,

empezando por reemplazar los caracteres ASCII por tiles 2D como ocurrió en *Ultima III* aparecido en 1983 para Apple II

Captura de *Ultima III* (1983)

En 1990 apareció para IBM PC otro juego del género, *Ultima VI*, que mostraba una proyección oblicua 2.5D conseguida haciendo que todos los objetos estuvieran girados 45º respecto al suelo. Además aumentó la paleta de colores a 256 (de los 4 a 16 disponibles anteriormente) e incluyó el uso del ratón como dispositivo de entrada, mejorando la calidad en la interacción del usuario con el juego.

Captura de *Ultima VI* (1990)

Otra de las visualizaciones posibles la encontramos en juegos de los 90 como *Darklands* (1992) y *Diablo* (1996), que hacen uso de la perspectiva isométrica 2.5D, donde los 3 ejes X,Y,Z crean ángulos iguales de 120 grados, efecto que se consigue en los juegos rotando 30 o 45 grados el tile y haciendo que el ancho sea el doble que el alto.

Darklands (1992) y *Diablo* (1996) respectivamente.

Uno de los saltos evolutivos más notables en cuanto a representación en el género ha sido el paso a 3D, que añade un nuevo grado de libertad al movimiento de la cámara, permitiendo una mejor inmersión en la aventura. Uno de los primeros juegos basados en D&D en hacer uso del 3D fue *Neverwinter Nights* en el año 2002.

Captura de *Neverwinter Nights* (2002)

2.2 Partes

Una vez definido lo que es un juego de tipo roguelike y mostrados algunos ejemplos de estos juegos a lo largo de su historia, es hora de adentrarse un poco más en ellos y listar ciertos elementos que suelen aparecer en juegos de este estilo.

2.2.1 Inventario

Una de las partes más utilizadas dentro de un roguelike es probablemente el inventario, que es a donde van todos y cada uno de los objetos que recogemos, bien sea por recompensa de derrotar a un enemigo o bien por haber abierto un cofre. Hay diferentes representaciones de inventarios, algunos muestran una matriz bidimensional que representa los 'slots' o espacios disponibles para colocar los objetos, teniendo en cuenta que ciertos objetos pueden tener un tamaño distinto a otros a la hora de insertarlos en el inventario, mientras que en otros inventarios se guardan de cualquier manera, como si de una bolsa se tratase.

El inventario puede tener dos limitaciones, una de espacio físico por haber sido ocupados todos los slots disponibles y otra por peso, si la suma de los pesos de todos los objetos supera un umbral, calculado normalmente en base a un atributo base como la constitución o fuerza del personaje.

Inventarios en *Ultima VIII* y *Diablo I* respectivamente.

2.2.2 Habilidades / Skills

El personaje del jugador no solo dispone de armas para combatir el mal, si no que usualmente cuenta con diferentes habilidades que le proporcionan una ventaja táctica en determinadas situaciones y que ayuda en la sensación de progresión del personaje antes descrita. Dichas habilidades normalmente pueden obtenerse de manera natural al ir subiendo niveles, dándole a elegir al jugador entre varias opciones, mientras que otras pueden obtenerse como recompensa de misiones o en libros u otros objetos al eliminar enemigos.

Dichas habilidades suelen tener restricciones de nivel y pre-requisitos para evitar que pueda el jugador obtener una ventaja excesiva y prematura que destruiría la fluidez de progresión de la partida. La cantidad de hechizos distintos que pueden ser lanzados durante el combate también suele estar limitada, para que el usuario deba tomar decisiones y priorizar unos sobre otros según la situación en la que se encuentre.

Ejemplo de árbol de habilidades en *Diablo 2*

2.2.3 Atributos / Stats

Parte importante en la misión de mantener una progresión fluida en el transcurso de la partida la tienen los atributos o stats del propio personaje, que determinan la potencia y

características en la que otros elementos de los que el personaje hace uso como habilidades o armas afecta al entorno que le rodea en el mundo virtual.

Dichos atributos normalmente se separan en atributos primarios y secundarios, estando los segundos ligados a los primeros mediante fórmulas matemáticas específicas que determinan su relación, como por ejemplo podría ser decir que el maná de un personaje es igual a 8 veces la inteligencia del mismo. Mientras que unos atributos afectan directamente a la capacidad del personaje de realizar o recibir más o menos cantidad de daño, también pueden tener otro tipo de efectos como que un personaje más fuerte sea capaz de llevar consigo más cantidad de objetos en el inventario.

Atributos comunes suelen ser fuerza que suele determinar el daño de armas cuerpo a cuerpo y la vida, inteligencia que suele estar ligado al daño de los hechizos y al maná disponible, y destreza que está ligado al daño de armas a distancia y a la armadura.

En otros juegos, normalmente basados en D&D, existen otros atributos que refinan a los anteriores, como constitución que se encargaría de la vida, sabiduría que es una variante de inteligencia y suele afectar a lanzadores de hechizos no magos, y carisma que puede influir en el comportamiento de los NPC o en habilidades de influencia.

Estos atributos pueden ser posteriormente modificados por objetos mágicos, habilidades, hechizos o efectos temporales, etc. y suele permitírsele al jugador que se incrementen conforme su personaje sube de nivel en una determinada cantidad.

2.2.4 Minimapa

Un elemento que no suele faltar en estos juegos es el minimapa, que no es más que una visualización reducida del mapa, como su nombre indica, que permite al jugador situarse en el mismo y poder avanzar en la aventura sin perderse.

Suelen verse tres tipos de minimapa, el que se sobrepone al renderizado actual del juego con semitransparencia o sin ella, el que aparece en una esquina de la pantalla con tamaño reducido o en algunos casos la combinación de ambas opciones.

Minimapa reducido y superpuesto respectivamente.

2.2.5 Diario de misiones

Contiene una lista de las misiones que el jugador tiene actualmente pendientes de realizar y, en algunos casos, las que ya ha terminado anteriormente a modo de histórico. También contiene información relevante para poder completarlas e información decorativa relativa al guión del juego.

Diario de misiones de *World of Warcraft*

2.2.6 NPC

Los NPC o personaje no jugador (Non-Player Character), son entes del mundo virtual controlado por el ordenador que pueden desempeñar diferentes roles dentro de la aventura, desde herreros, curanderos, armeros, acompañantes de aventura que ofrecerán en ocasiones servicios a los jugadores mientras que en otras ocasiones serán los que proveerán al jugador con misiones que realizar y que recompensarán al jugador una vez que la misión sea terminada.

2.3. Aleatoriedad

Se ha comentado previamente que la aleatoriedad es un factor que añade rejugabilidad a los juegos de rol, pero hay dos factores muy importantes a tener en cuenta sobre ella.

Lo primero a tener en cuenta es que la aleatoriedad debe tener un control, en la mayoría de casos no interesa tirar un dado y terminar, si no que es interesante aplicar cierto control sobre la aleatoriedad mediante límites y usando diferentes probabilidades.

Por ejemplo, si hablamos de un sistema de generación de objetos con propiedades mágicas aleatorio, un objeto de rareza única no podrá tener la misma probabilidad de ser obtenido por un jugador que uno de rareza común, para lo que debe aplicarse la diferencia de probabilidades al menos entre rarezas de objetos.

En cuanto a los límites, imaginemos en el mismo caso que a pesar de que la rareza ya se obtiene con probabilidades distintas, aparece al inicio de la partida un objeto de rareza común en el suelo con unos atributos de daño altos y objetos de rareza única con poco daño a niveles altos de la partida, en el segundo caso puede ser irritante mientras que en el primero el jugador no tendrá rival hasta avanzada la partida, perdiéndose la progresión requerida en los juegos roguelike.

Lo segundo a tener en cuenta es que la aleatoriedad puede aplicarse a diferentes elementos del juego roguelike y que, en cada una de ellas, se debe manejar de distinta manera. Algunas de las partes donde puede ser aplicada son las siguientes:

· **Objetos**: Cuando se abre un cofre de recompensa o se mata a un enemigo, se generan ciertos objetos de recompensa de manera aleatoria pero controlada, como se comentó anteriormente, para evitar que la jugabilidad y la progresión se vea afectada negativamente. Atributos como daño, propiedades, tipo de arma, estado del arma, rareza, nivel del objeto, etc. son algunos de los parámetros a seleccionar mediante el uso de la aleatoriedad de manera controlada, teniendo en cuenta los límites y probabilidades.

· **Misiones**: Normalmente la aleatoriedad aquí se suele usar en misiones llamadas secundarias, que no son necesarias para seguir el hilo argumental de la historia del juego, pero que permiten añadirle libertad al jugador y evitan que la historia sea siempre la misma. La aleatoriedad en estos casos puede decidir las misiones componiéndolas a partir de ciertas órdenes genéricas como "Matar a X enemigo" o "Ir a recuperar Y objeto", controlando que no siempre se pida lo mismo o que no se pidan cosas imposibles para el nivel del usuario en ese punto.

· **Eventos**: Son una interpretación concreta de misiones, ocurren con una baja probabilidad en el mundo de la aventura, suelen ser más complejos que las misiones normales y su recompensa mejor y única, siendo imposible encontrar esa recompensa en misiones normales.

· **Mapas**: La generación de mapas puede ser aleatoria, ya sean mazmorras, cuevas o exteriores, y aunque tengan zonas fijas como zonas de enemigo final. La aleatoriedad aquí debe ser controlada, por que no debe ocurrir que el mapa sea solo una habitación por ejemplo, o demasiadas en los primeros niveles. También se debe controlar que exista conectividad entre el punto de entrada al mapa y el de salida o de siguiente nivel para que la aventura pueda fluir de inicio a fin entre generaciones.

· **Enemigos**: Los diferentes enemigos pueden ser generados aleatoriamente de entre los enemigos posibles del juego dentro de unos límites, al igual que ocurría con los objetos, no debe aparecer un enemigo de nivel alto en una mazmorra de nivel bajo, así como tampoco deben aparecer muchos enemigos únicos y pocos enemigos comunes.
La posición, rareza, nombre, atributos, nivel, habilidades, personalidad de IA, son algunos de los parámetros que pueden ser seleccionados usando aleatoriedad.

Capítulo 3

Requisitos

3.1. Requisitos funcionales

Los requisitos funcionales nos definen las funciones del sistema de software o de sus componentes, donde cada función es un conjunto de entradas, comportamientos y salidas que define una funcionalidad específica que el software debe cumplir.
A continuación se listan los requisitos funcionales de este proyecto:

- Debe existir un menú principal desde donde el usuario podrá acceder a las opciones del juego, a sus personajes activos, a un historial de héroes caídos en combate, salir del juego o continuar la aventura con el último personaje jugado justo en el punto donde lo dejó.

- El usuario debe poder modificar opciones de audio del juego como volumen de música, volumen de juego o silenciar la música y el volumen de juego por separado.

- El usuario debe poder modificar opciones de gráficos como la resolución de pantalla a la que se renderizará o si el juego va a mostrarse en pantalla completa o no.

- Si el usuario modifica la resolución de pantalla, se deberá esperar unos segundos para darle tiempo al monitor a cambiar la resolución y en caso de no aceptar, la resolución volverá al estado anterior, evitando bloqueos en resoluciones no soportadas.

- El usuario debe poder modificar opciones específicas del juego como mostrar el contador de fotogramas, un reloj, mostrar números de vida/mana en las esferas del hud o mostrar el daño realizado al enemigo.

- El usuario debe poder modificar las teclas con las que se accederá a las diferentes partes de la interfaz dentro del juego tales como inventario, hoja de personaje, habilidades, libro de misiones, mapa o menú de opciones del juego.

- Los datos de configuración deben tener persistencia, el usuario puede decidir si guarda las modificaciones o las desecha y éstas deben ser cargadas automáticamente al iniciar el juego o al ser modificadas.

- El jugador podrá salir del juego, previo mensaje de aceptación, tanto desde el menú principal como desde el menú de juego.

- Dentro de la partida, el jugador podrá acceder a su inventario, donde podrá almacenar los objetos que encuentre durante la partida, así como deshacerse de ellos o equiparlos.

- Dentro de la partida, el jugador podrá acceder a su hoja de personaje, donde podrá visualizar sus atributos.

- Dentro de la partida, el jugador podrá acceder a las habilidades disponibles, donde podrá seleccionar cuales va a usar durante la partida y al subir de nivel, cuales desbloquear.

- Dentro de la partida, el jugador debe poder ver tanto su vida como su mana y experiencia, y poder acceder a las diferentes secciones ingame (inventario, hoja de personajes, etc) con botones en una barra en el HUD.

- Durante la partida, el jugador debe disponer de un mapa con el que poder orientarse y en el que se marcará la posición del jugador.

- Durante la partida, el jugador debe poder mostrar un menú, desde el que podrá o bien ir al menú principal, al menú de opciones, volver a la partida o salir del juego.

- Al crear una nueva partida, el jugador debe poder asignar un nombre y seleccionar una clase de personaje (aunque en esta 'demo' se limitará a una clase de personaje)

- El usuario debe disponer de una ventana donde seleccionar entre los múltiples héroes vivos para continuar su aventura, donde aparecerá información sobre el mismo como el nombre, la clase o el nivel.

- Los objetos del juego deben generarse aleatoriamente, aunque deben tener un control que no permita generar objetos de nivel alto en niveles bajos y viceversa.

- Los objetos además, podrán ser de diferentes tipos, disponer de diferentes propiedades mágicas y ser de distinta rareza, la cual determinará la cantidad y calidad de esas propiedades.

- Se usará el oro como moneda en el juego, que se obtendrá como recompensa al matar enemigos o al completar misiones, y se podrá usar en los comerciantes NPC del juego.

- Al crear una nueva partida, se generarán los mapas de manera aleatoria, aunque algunos serán estáticos para asegurar batallas contra enemigos finales o como la ciudad, y esos mapas se usarán durante toda la partida de ese héroe.

- Si durante la partida el jugador muere, ese héroe queda automática e irreversiblemente inservible, aunque aparecerá en la lista de héroes caídos.

- El jugador deberá poder acceder a una lista con los héroes caídos en combate, en la que se mostrarán estadísticas de juego como fecha/hora de creación y muerte, tiempo jugado, nivel logrado, enemigos eliminados, etc.

- El juego debe reproducir música de fondo y efectos sonoros.

- Deberán generarse enemigos de manera aleatoria junto con los mapas y estos deben generar objetos y dar experiencia al héroe que los elimina.

- El comportamiento de los enemigos debe ser controlado por el ordenador.

3.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que no afectan a la funcionalidad del programa pero que son exigidos por las especificaciones. En el caso de este proyecto, los requisitos no funcionales son los siguientes:

· El juego será uniplataforma; debe ser ejecutable y funcional en Windows 7 y versiones superiores.

· El idioma del juego será el castellano en su totalidad, no se contempla multi-idioma en este proyecto.

· El juego debe poder ser ejecutado de manera fluida en al menos un procesador I3 a 2.4 Ghz con 4 GB de RAM y tarjeta gráfica integrada ATI HD 5450.

· El juego debe poder ejecutarse en distintas resoluciones gráficas, como mínimo en 1024x768 para establecer un mínimo de calidad.

· El juego debe poder renderizar el mismo tiempo de juego independientemente de la velocidad del ordenador que lo ejecuta, para evitar que un ordenador el doble de potente haga que el jugador vaya el doble de rápido por el mapa o viceversa.

La selección de requisitos hardware y software se ha realizado en base al equipo portátil que se ha usado para realizar las presentaciones del proyecto, mientras que las de idioma se basan en que, a pesar de que el multi-idioma es una característica interesante, sale de los objetivos del presente proyecto de final de grado y por lo tanto se deja como tarea futura en caso de proseguir con el proyecto tras la entrega.

3.3. Herramientas y tecnologías

En esta sección se muestran las herramientas y tecnologías que se han empleado para la creación del proyecto, así como la justificación de las elecciones.

Lenguaje de programación: C#

Si bien es cierto que la mayoría de juegos en el mercado están programados en C++, se ha de tener en cuenta que este proyecto no es llevado por una gran compañía de videojuegos que se puede permitir el tiempo, el personal y el dinero necesarios para la creación del videojuego usando C/C++ y que, las ventajas de velocidad y rendimiento que se obtendrían no compensan la inversión sobre todo temporal, ya que este proyecto de final de grado debe entregarse en tiempo finito.

Otro lenguaje que se ha tenido en cuenta es Java, que aportaría portabilidad además de sencillez a la hora de programar, pero la portabilidad no es requerida por lo que no es un factor de peso en la decisión.

Finalmente se ha seleccionado C# como lenguaje a usar en el proyecto, debido a la rapidez con la que permite la creación de código y que va a permitir que me pueda centrar más en los objetivos del proyecto que en batallas con el código, teniendo en cuenta además que la eficiencia va a ser menor que una compilación en C++, y a pesar de que no ofrece ejecutables multiplataforma, ya que esto último no es un requerimiento de los requisitos del proyecto.

Editor de código: Visual Studio 2010

La elección de Visual Studio es natural, debido al lenguaje de programación y librería multimedia seleccionados y además, se han tenido en cuenta las facilidades que este editor aporta para facilitar la tarea de creación de código, como resaltado de código, depuración avanzada, posibilidad de control de versiones, pruebas unitarias, diseñador de clases, autocompletado, etc. así como una extensa, actualizada y fácil de usar ayuda a través de MSDN.

Librería multimedia: XNA 4.0

Este conjunto de bibliotecas de clases para el manejo de elementos multimedia como fuentes, imágenes, sonido, música, modelos 3D, etc. proporciona una capa de abstracción que permite usar estos componentes de manera más sencilla y no nos obliga a crear un motor gráfico para poder desarrollar el proyecto.

A pesar de que Microsoft no va a sacar más versiones por encima de la 4.0, esta versión contiene todo lo necesario para la realización del videojuego y se integra perfectamente con el entorno Visual Studio 2010 y el lenguaje C#.
Otra de las desventajas de XNA es que es específico de Windows, XBOX y Windows Phone, pero como se vio en los requisitos no funcionales, es suficiente con que tenga soporte para Windows.

Otra alternativa similar, que solventaría tanto el problema de futuras versiones o el de la compatibilidad con otras plataformas con Linux o Mac, podría ser portar el código de XNA a Mono, sin apenas ningún cambio.

Editor 3D: Blender 2.68

Para el modelado de los elementos 3D del juego se ha seleccionado Blender, principalmente por que otras herramientas de similares como Maya o 3DStudio son de pago a un precio desorbitado siendo estudiante y teniendo en cuenta que el proyecto que se va a realizar no es comercial y es más que suficiente para obtener el resultado esperado. Otro factor tenido en cuenta ha sido que en la asignatura de sistemas gráficos interactivos nos fue explicada esta herramienta por lo que aun habiendo sido gratuitas el resto, posiblemente habría sido seleccionada Blender de igual manera.

Editor 2D: Gimp 2

En el caso de manejo de imágenes 2D tales como fondos de pantalla, cursores, iconos, objetos de interfaz, etc, ha sido necesario el uso de una herramienta de diseño gráfico y, de igual manera que ocurría con Blender en el caso del modelado 3D, esta herramienta gratuita proporciona lo necesario para cumplir tal objetivo.

Capítulo 4

Módulo generación de objetos

En esta sección se va a hablar sobre la generación aleatoria de los objetos del juego y de sus modificadores y de como se va a realizar de tal forma que no destruya la sensación de avance o mejora necesaria en los juegos del género. Se mostrará primero lo que se busca que la librería encargada de ésta tarea contenga para posteriormente detallar la implementación realizada de la misma.

Como este módulo es independiente de otros, se va a generar una librería .dll con todas las clases necesarias para la generación aleatoria, de tal manera que las modificaciones no interfieran con otros módulos y permitiendo la reutilización de la misma para posibles futuros proyectos basados en este tipo de juego.

4.1. Algoritmo

Dentro de los objetos hay varias maneras de categorizarlos, una de ellas es mediante la rareza, vemos que hay diferentes tipos de objeto según su rareza:

· **Comunes** (Blancos): Objetos sin modificadores, solo nombre, tipo, nivel..

· **Infrecuentes** (Azules): Objetos con 2-3 modificadores, nombre basado en modificadores (afijos, sufijos) obtenidos de una DB de modificadores por nivel

· **Raros** (Amarillos): Objetos con 3-5 modificadores, nombre aleatorio por tipo de objeto

· **Únicos** (Dorados): Previamente generados en una DB con nombre, nivel y modificadores.

· **Consumibles:** Pociones, cantidad, tipo y calidad generadas según nivel y random.

· **Oro:** Se generará una cantidad aleatoria de oro si sale seleccionado este tipo.

La base de datos de modificadores define el tipo de modificador, nombre, nivel mínimo necesario para que sea seleccionado por el generador, rango de valor del modificador (Ej. +3-10 de vida),etc. Al generarse el objeto se seleccionará un valor aleatorio de este rango para asignárselo al objeto.

Los items podrán ser generados aleatoriamente cuando se le requiera a la .dll (cuando muere un enemigo, cuando abre un cofre, en la tienda..) o de manera estática (objetos de misión..)

Dado que se puede limitar tanto la generación del item como la de sus modificadores a un determinado nivel, se garantiza que el objeto se ajuste a las características de la zona en la que se ha generado, evitando desequilibrios en el desarrollo de la partida. Estos desequilibrios podrían ser provocados si en un mapa de nivel 3 al jugador le apareciera aleatoriamente un objeto de nivel 20, lo que reduciría la dificultad a niveles absurdos y destruiría la jugabilidad como ya se comentó previamente.

Para disponer de más control sobre la generación, se podrán ajustar los porcentajes de probabilidad de que un objeto sea de un tipo u otro de los descritos en la lista anterior.

Finalmente el objeto generado podrá ser serializado/deserializado para garantizar su persistencia.

A la hora de generar los objetos de manera aleatoria, hay que seguir un determinado algoritmo formado por pasos que defina de manera completa la generación del objeto.

El algoritmo de creación aleatoria seguirá estos pasos:

1.- Aleatoriamente determinar el número de objetos a generar.

2.- Determinar aleatoriamente el tipo de drop a generar (oro, objeto, consumible).

2.1.- Si es oro, generar una cantidad aleatoria basada en nivel de zona y modificador +% oro en los enemigos del jugador.

2.2.- Si es consumible, determinar aleatoriamente tipo, cantidad y calidad, por nivel

2.3.- Si es objeto, determinar rareza (normal, infrecuente, raro, único) basándose en un número aleatorio y en las tablas que determinan la probabilidad base de cada rareza adaptadas al porcentaje de probabilidad actual acumulado por el jugador, así como seleccionar aleatoriamente un subtipo de objeto (arma, escudo, anillo, casco..).

2.3.1: Si es común, devolver el objeto tal cual estaba en 2.3

2.3.2: Si es infrecuente, generar aleatoriamente 2 o 3 modificadores acordes al objeto y nivel, y devolver el objeto generado.

2.3.3: Si es raro, generar 3-5 modificadores y un nombre aleatorio según tipo de objeto y nivel, y devolver el objeto generado

2.3.4: Si es único, buscar aleatoriamente un objeto en la base de datos de únicos, que cumpla las restricciones de nivel y tipo seleccionadas y devolver el objeto generado.

2.4 Si alguno de los pasos fallara (Ej. No existe un único para ese tipo de objeto y nivel), se desestimaría el objeto y se continuarían generando el resto de items del paso 1 en el paso 2.

En este apartado no es necesario buscar ningún otro algoritmo, puesto que éste generará el resultado requerido durante el transcurso de toda la partida y es extremadamente rápido debido a la simplicidad de la idea, y a pesar de esto generará una

gran variedad de objetos que se adaptarán a la jugabilidad de las áreas donde sean generados.

Programas auxiliares:

· Para facilitar la tarea de ajustes futuros en los objetos se podría crear un programa de edición de items/modificadores que permitiera ajustar los valores de nivel y de intervalo de los modificadores.

· Se podría generar otro programa para comprobar que la generación aleatoria de objetos se ajusta a los valores predefinidos, generando un nº de objetos y realizando gráficas con los resultados donde poder comparar.

4.2. Implementación

El primer paso para poder implementar esta librería de generación de objetos aleatoria ha sido crear un diagrama UML que represente la jerarquía que describe los diferentes tipos de objetos posibles a generar. Por un lado se han separado dos grandes tipos de objetos, los que son equipables y los que no, mientras que los equipables a su vez pueden ser de tipo arma y equipable en la mano izquierda del héroe o armadura que puede ser equipable tanto en la cabeza, pecho, piernas, pies o mano derecha. Los objetos equipables además cuentan con una lista de modificadores que permiten al objeto tener propiedades mágicas y modificar los atributos del jugador al ser equipados.

En cuanto a los objetos no equipables, no se les permite que tengan modificadores ni que sean equipables y pueden ser de dos subtipos que son oro y poción. En el caso del oro su única propiedad específica es la cantidad del mismo mientras que en las pociones lo es el tipo de poción que puede ser de vida o de maná.

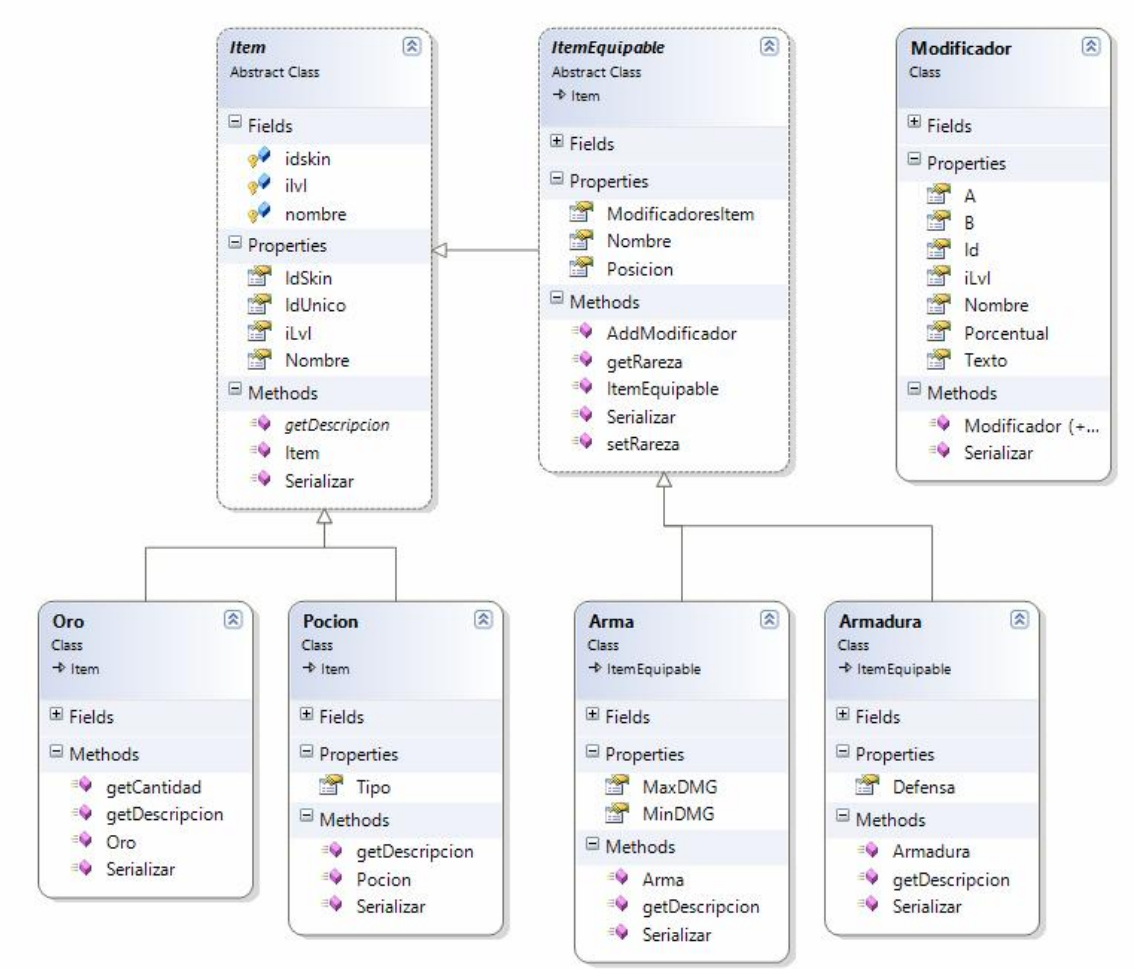
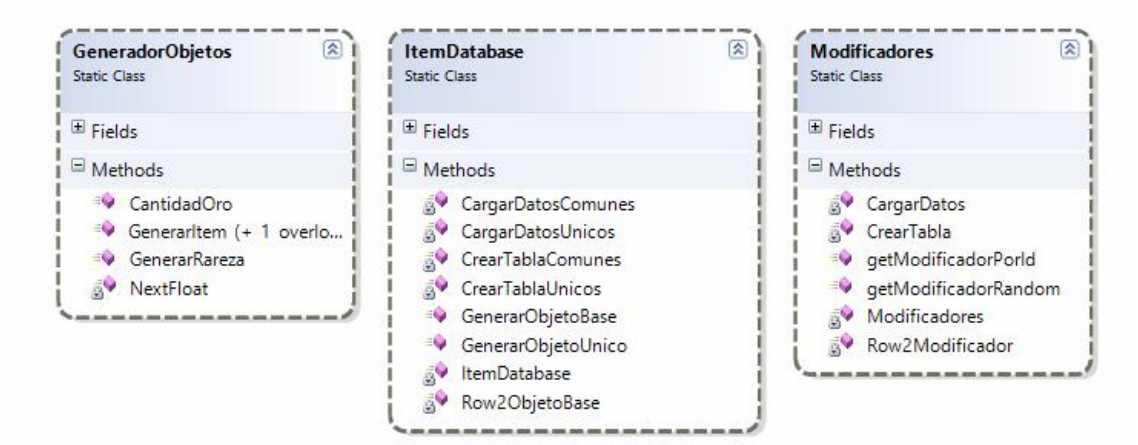


Diagrama de clases de tipos de objetos



Factorías de generación de objetos y de sus modificadores

Para poder generar objetos con el algoritmo expuesto es necesario mantener cierta información y poder filtrarla según ciertos criterios, por lo que se ha decidido hacer uso de los componentes ADO.NET de conexión con bases de datos por que permiten la carga en memoria y el posterior filtrado de datos incluso sin un origen de datos. En este caso se abren ficheros de texto con un StreamReader, se procesan los datos y se añaden a una tabla en memoria de tipo DataTable previamente declarada. Una vez se dispone de los datos en esta tabla en memoria, se puede hacer uso del método Select(filtro) del DataTable que permite obtener un subconjunto de filas que cumplen el determinado filtro, como por ejemplo, objetos de nivel menor o igual a 2.

En este proyecto se mantienen en la carpeta 'Datos/' tres listas, la primera llamada objetosbase.txt que contiene el nombre, el nivel, un valor min y max, el identificador del skin usado al dibujarlo y la posición donde se equipa de los objetos base o blancos sin modificadores. La segunda lista se llama 'objetosunicos.txt' y contiene el nivel y la posición donde se equipa y además una cadena de caracteres que incluye el serializado del objeto de rareza única y que permitirá posteriormente crearlo desde la factoría. La tercera y última lista con nombre 'sufijos.txt' contiene el identificador, nombre, nivel, texto formateado, un valor min y max de un modificador, que servirá para cuando se deban crear objetos con rareza distinta a la de objeto base que incluyan modificadores de atributos.

```
// Devuelve un objeto base de los posibles para una determinada posicion y nivel maximo dados
public static ItemEquipable GenerarObjetoBase(Posicion pos, int nivel)
{
    // Hace la consulta por nivel y pos, selecciona uno random de los devueltos
    ItemEquipable item = null;
    DataRow[] res = dtcomunes.Select("ilvl <= " + nivel + "AND pos=" + (int)pos);
    if (res.Length > 0)
    {
        int nrand = rand.Next(0, res.Length);
        DataRow r = res[nrand];
        item = Row2ObjetoBase(r);
    }
    return item;
}
```

Código para generar objeto base aleatorio

Se puede observar en el código de arriba que se vuelve muy sencillo obtener un objeto base aleatorio para una determinada posición y con un nivel menor o igual a uno dado.

Por encima de todo esto, la clase que se encarga de la generación aleatoria de objetos de todas las maneras posibles y de manera transparente es la estática GeneradorObjetos que dispone de un método con dos sobrecargas llamado GenerarItem que se encarga de esto.

El primer método recibe como parámetro un string con el serializado de un objeto, el cual descompone ya que está compuesto por cadenas de texto separadas por el símbolo '|' y va interpretando los tokens, llamando al constructor de la clase de objeto correspondiente con los parámetros incluidos en la cadena serializada y devolviendo una instancia en un objeto Item que es la base de la herencia de cualquier objeto, por lo que independientemente del tipo de objeto creado podrá ser devuelto por la función.

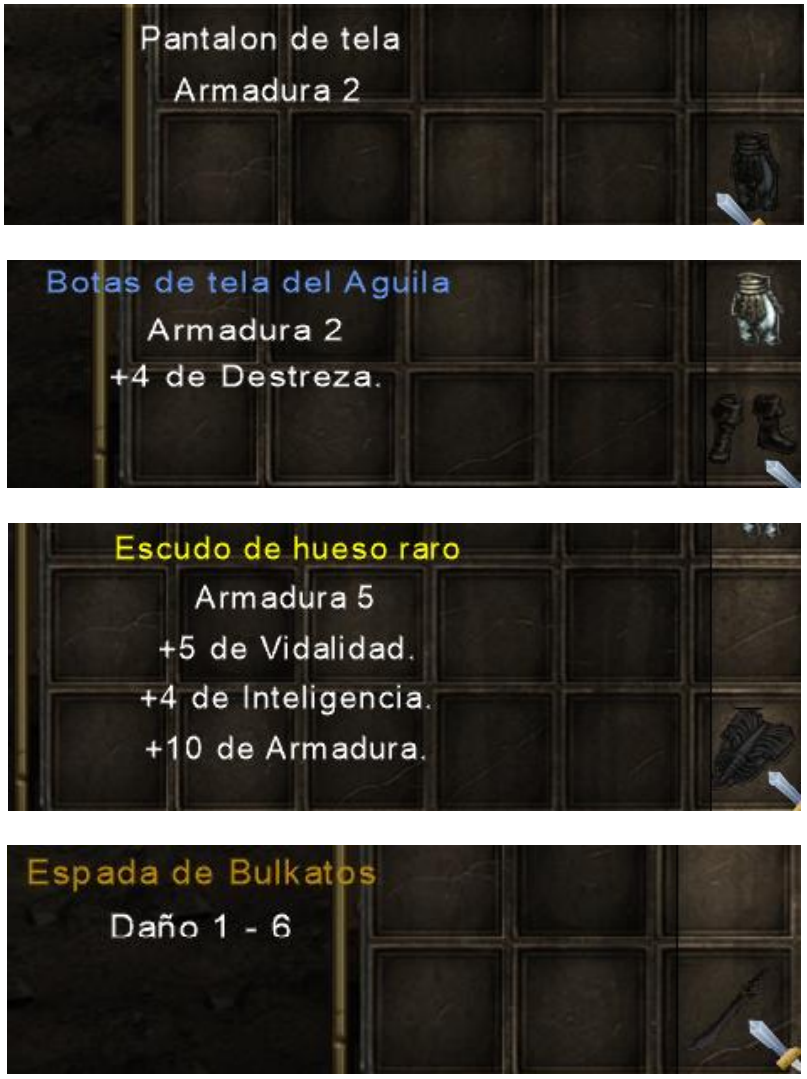
El segundo método recibe como parámetro el nivel máximo del objeto a generar y la probabilidad. Primero se determina aleatoriamente el tipo de objeto de entre los posibles, con una probabilidad equitativa, lo que puede derivar en tres posibilidades, que sea oro, consumible u objeto. Si es oro se calcula una cantidad de oro aleatoria limitada por el nivel del personaje, si es consumible tiene un 50% de probabilidades de ser una poción curativa y otro 50% de ser de maná y si es objeto se genera una posición aleatoria de entre las posibles posiciones donde un objeto puede ser equipado y se genera un objeto base con la factoría ItemDatabase.GenerarObjetoBase para ese nivel o inferior y para esa posición. Si no se encontrara un objeto con esos requisitos, se devolvería oro en su lugar, y si se encuentra se genera una rareza de entre las posibles (70% de ser común, 20% de ser mágico o azul, 7% de ser amarillo o raro y 3% de ser único), y se modifica la rareza del objeto acorde al resultado obtenido con el método setRareza del item.

El método setRareza modifica convenientemente las propiedades del objeto, eliminando modificadores si se quiere obtener un objeto común o base, solicitando a la factoría de modificadores una cantidad aleatoria de entre 1 a 3 modificadores de atributos, de 3 a 6 si es un objeto raro y solicitando un objeto de la lista de únicos en el último caso.

Se ha incluido un método virtual Serializar() en la clase base Item que será sobrescrito en las clases que lo hereden para que concatene los datos específicos de cada clase separados por '|' de tal manera que posteriormente se pueda usar en la factoría para crear el objeto a partir de esta cadena de texto generada, lo que facilita la posterior persistencia del

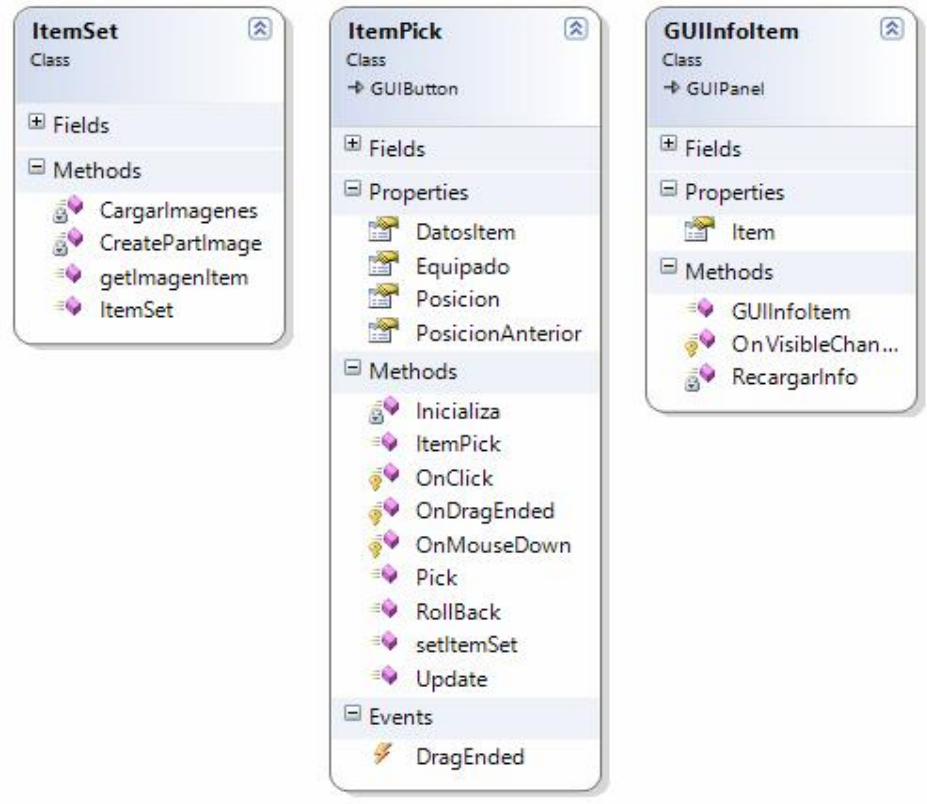
objeto en el inventario del personaje, ya que si se volviera a generar aleatoriamente daría como resultado un objeto distinto, por lo que es de vital importancia que este serializado se lleve a cabo de manera correcta si se pretende recuperar el objeto posteriormente.

De una manera similar se ha incluido la propiedad virtual Nombre, que se sobreescribe posteriormente en ItemEquipable para construir el nombre del objeto según su rareza, por ejemplo los objetos base reciben el nombre desde la factoría mientras que los mágicos o azules lo componen con la concatenación de los modificadores que lo componen.



Objetos generados aleatoriamente con diferentes rarezas.

Una vez generados, los objetos de tipo Oro se añadirán directamente al inventario del jugador, mientras que los de poción harán lo mismo en la barra de estado inferior del HUD. Para los objetos equipables sin embargo, el proceso es algo más complejo y se obtiene mediante el uso de 3 clases, ItemSet se encarga de mantener una relación de imágenes para todos los objetos posibles del juego mientras que ItemPick representa el objeto visual que se muestra en pantalla, encapsulando además los datos propios de ese objeto aleatoriamente generado en la propiedad DatosItem y controlando el arrastre del objeto desde el inventario a la zona equipable y viceversa. Por otro lado, para mostrar la información de los atributos y modificadores de un objeto se usa la clase GUIInfoItem, a la que se le asigna un objeto al pasar el ratón por encima del mismo, y muestra toda la información del mismo en un recuadro semitransparente, realzando la rareza del objeto mediante el uso de los códigos de color del algoritmo de generación de objetos aleatorios.



Clases para la visualización de los objetos equipables generados.

Capítulo 5

Módulo IA

En cualquier videojuego es común encontrarse con contrincantes controlados por el ordenador, sustituyendo a oponentes humanos en juegos 1 vs 1 como ajedrez o damas, o comportándose como un tipo de entidad específica del juego como ocurre con los NPC (Non-Player Character) o los propios enemigos en juegos RPG por ejemplo. Para que la máquina pueda controlar los comportamientos de estas entidades se debe diseñar e implementar una IA (Inteligencia Artificial) que se haga cargo a partir de unas entradas como pueden ser la distancia al héroe o un temporizador, procesar y decidir qué salidas modificar y de qué manera, por ejemplo, si el enemigo está cerca del héroe debe atacarlo.

En juegos basados en Rogue, como el actual proyecto, se suele trabajar con multitud de enemigos y es imperativo disponer de una IA que controle los comportamientos de los diferentes enemigos para añadir jugabilidad al videojuego y evitar que los enemigos estén parados a la espera de que el jugador los elimine sin ningún tipo de oposición, lo que le quitaría cualquier aliciente al usuario para seguir jugando.

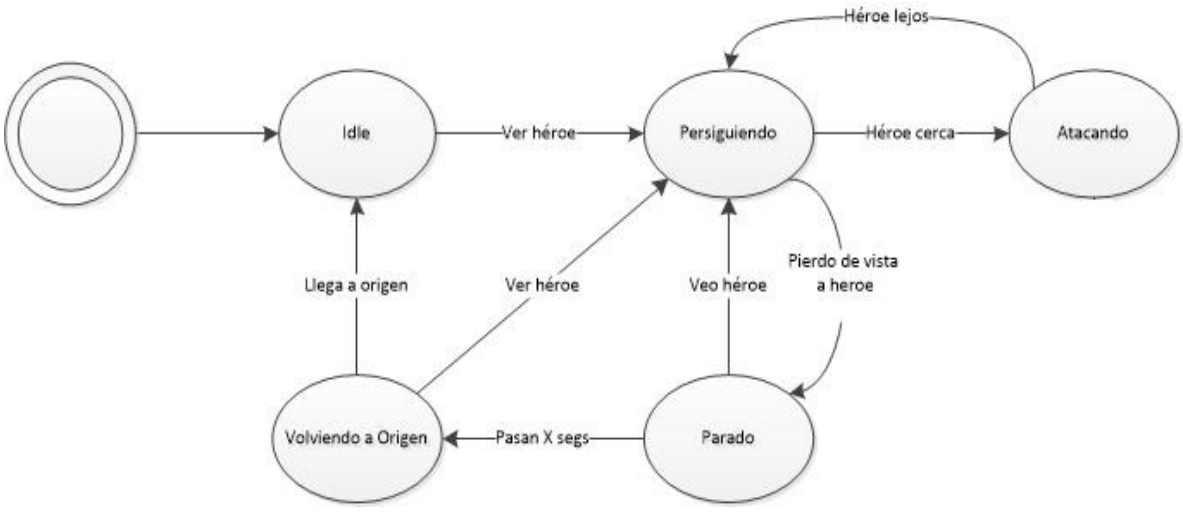
Como en otros módulos del proyecto, se va a generar una librería .dll donde se añadirán las clases genéricas de los diferentes apartados relacionados con la IA para que puedan ser reutilizados en futuros proyectos, aunque habrán otras clases específicas del proyecto que heredarán de las clases de la .dll que se incluirán en el proyecto pero fuera de la librería, como se verá en las siguientes secciones del capítulo.

5.1. Máquina de estados

Cuando se ha tenido que seleccionar un algoritmo de IA que se encargara de controlar los comportamientos de los enemigos, teniendo en cuenta que se deseaba que fuera sencillo pero eficaz y sobre todo que pudiera funcionar en tiempo real, se ha decidido escoger la máquina de estados sobre sus otros candidatos.

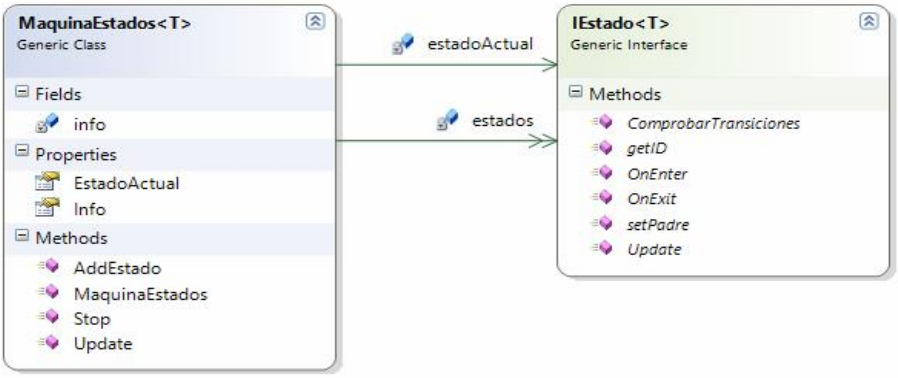
El funcionamiento de una máquina de estados es muy simple, está basada en diferentes estados que define las diferentes etapas en las que puede estar la entidad a controlar por la IA en un determinado momento, y por transiciones entre estados, que determinan qué evento debe ocurrir para que estando en un determinado estado se pase a otro definido por la dirección de la fecha que representa la transición entre estados.

En el caso de este proyecto, se ha usado la misma máquina de estados para todos los tipos de enemigos posibles, aunque es normal diseñar varias máquinas de estados distintas para disponer de diferentes comportamientos distintos de los enemigos, como por ejemplo una IA defensiva, otra agresiva, una para cuerpo a cuerpo u otra para ataques a distancia. El diseño de la máquina para ataques cuerpo a cuerpo se muestra a continuación:



Máquina de estados para enemigos.

Se puede observar en la máquina de estados que no tiene estado de aceptación, una vez que inicia la IA no parará hasta que muera o acabe con el héroe. Aunque el estado muerto no aparezca en el diagrama, si el enemigo muere, independientemente del estado en el que se encuentre, se quedaría en un estado de absorción 'muerto' del que no saldría, evitando cualquier transición para un enemigo abatido.



Máquina de estados genérica.

En la imagen anterior se puede ver las dos clases que componen el apartado de máquina de estados genérica dentro de la librería que implementa un patrón GOF State. Por un lado la interfaz IEstado<T> define todos los métodos que un estado necesitará declarar para formar parte de la máquina de estados y además con el tipo genérico T se puede especificar un objeto contenedor de información con el que se permitirá al estado tener contacto con el entorno. Los métodos de IEstado OnEnter y OnExit serán ejecutados por la máquina de estados cuando el entrado sea activado o cambiado respectivamente, update y comprobarTransiciones se ejecutarán en cada ciclo de Update() del juego pero el primero permite al estado realizar tareas de actualización mientras que el segundo se encarga de determinar si el estado debe cambiar a otro al cumplirse alguna de las transiciones del diagrama. Por otro lado setPadre permite guardar la máquina de estados a la que pertenece el estado y getID se puede sobrescribir en la clase heredada para especificar el identificador del estado que es está creando a partir de esta interfaz.

La clase MaquinaEstados<T> mantiene los diferentes estados que la componen y que deben implementar la interfaz IEstado y se encarga de llamar a los eventos del estado en cada ciclo y de intercambiar el estado actual de la máquina cuando comprobarTransiciones lo pide, y disparando los eventos OnEnter y OnExit cuando se produce el cambio.

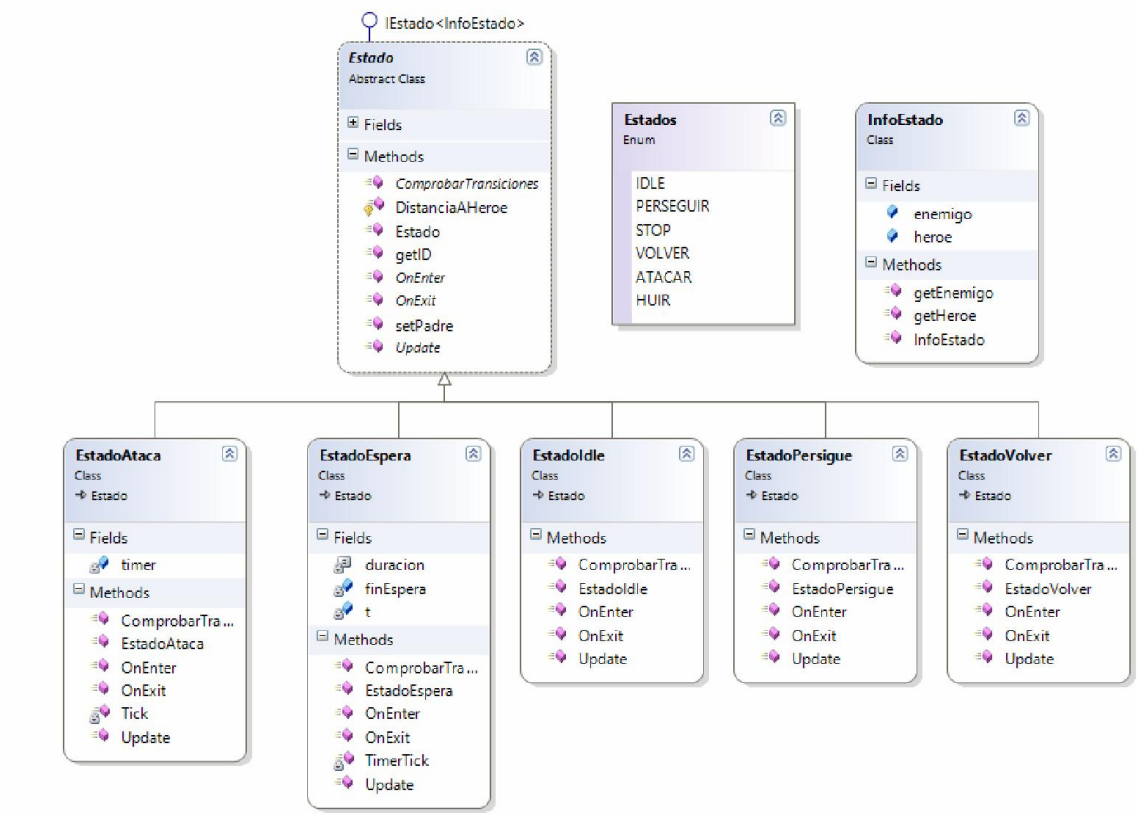


Diagrama de clases de la máquina de estados para la IA del proyecto.

Una vez definidas las clases contenidas en la librería se pasa a explicar el contenido de las clases específicas del juego que definen el diagrama de estados mostrado al inicio del capítulo para lo que se ha creado una enumeración con los estados posibles y una clase abstracta que hereda directamente de IEstado<T>, incluyendo funciones auxiliares útiles para el resto de estados como DistanciaAHeroe y asignando a T el tipo de InfoEstado, que será la clase que se usará para permitir a los estados interactuar con lo que ocurre fuera de la máquina de estados, concretamente se le proporciona en este objeto información referente a un Héroe y a un Enemigo, por lo que la IA de una instancia en concreto de enemigo dispondrá información del héroe al que debe atacar y del propio enemigo.

De la clase abstracta Estado nacen los 5 estados que componen la máquina de estados, sobrecargando los métodos necesarios como ComprobarTransiciones donde cada estado hará las comprobaciones de las transiciones que salen de ese estado a otros o los métodos OnEnter y OnExit para realizar tareas de inicialización o finalización al dispararse el evento, como por ejemplo en el estado EstadoAtaca que en el OnEnter comienza a golpear el héroe o en el EstadoEspera que inicia un contador de tiempo.

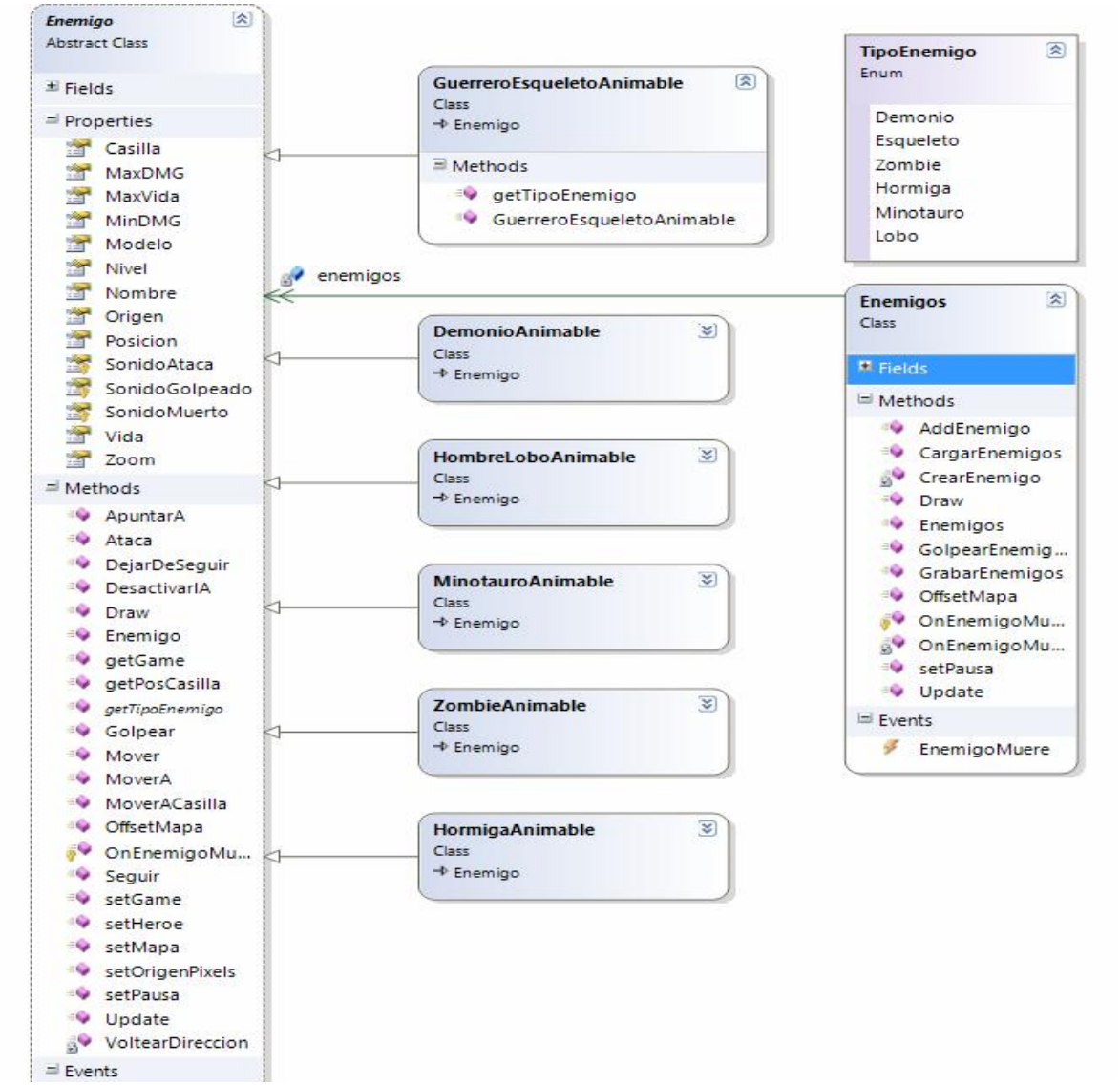


Diagrama de clases de los enemigos.

En este punto ya está definida la parte genérica de la máquina de estados y la máquina de estados específica, y el siguiente paso es combinarla con el propio enemigo para conseguir su autonomía. Se ha creado la clase Enemigo que se encarga de contener los datos del enemigo como la vida o el daño, un objeto de animación y una instancia de máquina de estados, de tal manera que encapsule y controle tanto el comportamiento, los movimientos como el dibujado del modelo. De esta clase abstracta heredan cada uno de los tipos de enemigos distintos del juego, ya que todos se van a comportar de la misma manera, y la única diferencia es el modelo y los sonidos específicos de cada tipo de enemigo que son asociados en el constructor pertinente. La clase Enemigos contiene y controla todos los enemigos que aparecen en un determinado mapa y su persistencia.

5.2. Pathfind

Otra de las necesidades genéricas incluidas en esta librería es la búsqueda de rutas o algoritmo de pathfind, debido a que hay estados en la máquina de estados definida en el apartado anterior que necesitan de ellos, ya sea al perseguir al héroe evitando obstáculos si fuera necesario o bien volver al punto de origen si el enemigo está mucho tiempo sin ver al héroe. Se ha optado por una implementación estándar del algoritmo A* de manera iterativa, separando el algoritmo en dos clases, la primera llamada AStar contiene el algoritmo iterativo con las funciones auxiliares para comprobar si un tile es muro, si una posición es válida, para expandir los nodos a los que se puede acceder desde el actual o para reconstruir el camino final una vez se ha terminado el algoritmo. La otra clase representa a cada uno de los nodos del algoritmo A* y contiene los valores de F, G y H que se utilizan por el algoritmo para ordenar los nodos con una cola de prioridad, la dirección a la que se debe mover, la posición actual de la casilla que representa el nodo, así como la casilla origen y la destino para que se pueda comprobar con el método EsMeta si el nodo es final. Al finalizar el algoritmo se obtiene una lista ordenada de NodoAStar donde cada nodo representa una posición del camino a recorrer para llegar del origen al destino.

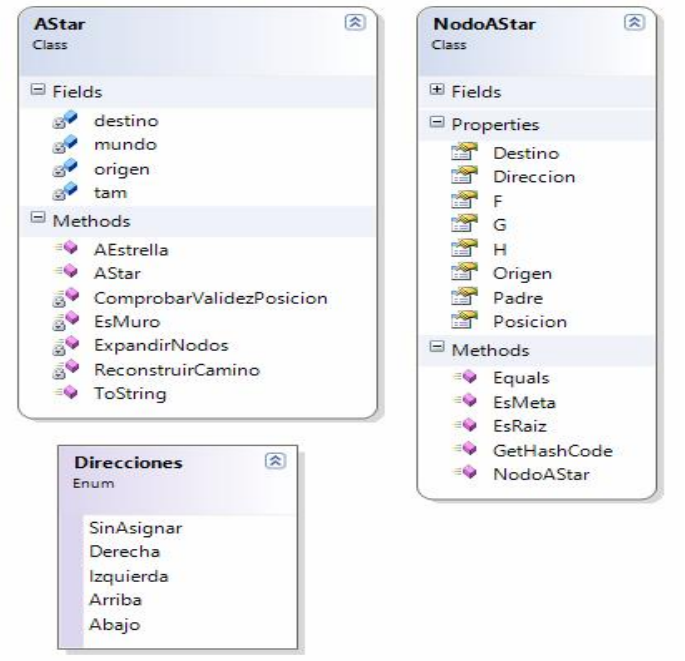


*Diagrama de clases de A**

Para obtener la ruta del origen al destino es necesario crear un objeto AStar y llamar al método AEstrella con el punto de origen y destino, lo que devuelve una lista de nodos como ya se ha comentado. Se puede observar en el código siguiente, en el que además se mira la dirección a la que hay que moverse en la primera posición de la ruta, que es lo que se usará en la clase Enemigo ya que al ir recalculando la ruta solo te interesa realmente el primer paso:

```
AStar pathfind = new AStar(mapa);
List<NodoAStar> ruta = pathfind.AEstrella(origen, destino);
Direcciones dir = ruta[0].Direccion;
```

Código para la obtención de rutas.

El algoritmo A* usa 3 valores para poder establecer un orden en la lista con los nodos pendientes de procesar y que se calculan para cada uno de estos nodos, por un lado se tiene g(x) que es el coste de llegar desde el nodo inicial al nodo actual, h(x) que es el coste estimado para llegar desde el nodo actual al nodo final o meta y f(x) = g(x) + h(x). El coste se calcula mediante la función de distancia de Manhattan para 2 dimensiones.

$$|x_1 - x_2| + |y_1 - y_2| \,.$$

Distancia de Manhattan para 2D.

El funcionamiento del algoritmo comienza por añadir el nodo inicial a la lista de nodos a procesar, y en bucle sacar de la lista de nodos el que tenga menor coste f(x) y meterlo en la lista de nodos visitados, de ese nodo se calculan los adyacentes no visitados que no sean muro y por cada uno de esos nodos expandidos se calculan sus valores f(x), g(x) y h(x) y se le asigna como nodo padre el actual, luego se añaden a la lista de nodos para calcular, repitiendo este proceso hasta que aparece el nodo final en la lista de nodos procesados, tras lo que se recorre recursivamente el nodo final hasta que su padre sea nulo, construyendo en el proceso la deseada lista con los nodos de la ruta entre el origen y el destino.

Capítulo 6

Módulo GUI

Una de las necesidades más comunes en las aplicaciones y en especial en los videojuegos es la de disponer de una interfaz que le permita al jugador interactuar con el juego en cualquiera de los dos sentidos, ya sea para añadir o modificar información en la partida o en el sentido contrario, para recibir notificaciones o visualizar información interna de la partida.

En aplicaciones de escritorio es común disponer de librerías que nos faciliten ciertos componentes genéricos de interfaz como botones, textbox, listbox, combobox, label, etc. y que podemos usar directamente arrastrándolos desde una toolbox, como ocurre con GTK, QT o en el caso de .NET, Windows Forms.

En el caso particular de este texto, al usar XNA no disponemos de esas librerías, debido al funcionamiento distinto que tiene un videojuego o una aplicación gráfica en general comparada con una aplicación de escritorio o web, por lo que será necesario o bien encontrar una librería existente o crear una desde cero.

En este capitulo se verá cual ha sido la decisión tomada, y como se ha diseñado e implementado la interfaz específica del videojuego basada en la solución seleccionada.

6.1. Caso general

Para poder tener cierto control sobre la interfaz y poder modificar y personalizar como sea necesario la misma, se ha decidido crear una librería genérica con controles básicos de interfaz, con la que además, al ser genérica, se podrá usar en futuros proyectos basados en XNA.

Al analizar los requisitos de la aplicación y los mockups de los menús, se llega a la conclusión de que se necesitan los siguientes tipos de componentes:

GUIButton:	Reacciona al evento click.
GUICheckBox	Permite marcar/desmarcar.
GUITextBox	Entrada de texto por teclado.
GUILabel	Texto con alineación y otros parámetros de formato.
GUICombobox:	Permite selección de una lista de elementos.
GUIPanel	Objeto contenedor de otros objetos de interfaz
GUIDialog	Panel con texto, botones para aceptar/cancelar y retorno.
GUIButtonKeyboard	Botón específico para configuración de teclas en opciones.
GUIListBox	Lista de elementos con scroll vertical
GUIScrollBar	Barra de scroll vertical/horizontal
GUIProgressBar	Barra de progreso horizontal
GUICursor	Muestra una imagen en la posición del cursor del ratón.

Controles incluidos en la librería de componentes gráficos.

Todos estos componentes tienen ciertas características comunes, como por ejemplo la propiedad de visibilidad o activación, el evento de click, etc. por lo que todos ellos heredan de la clase abstracta GUIControl y extienden su funcionalidad. Además algunos componentes como GUIPanel desempeñan la función de contenedor mientras que otros extienden la funcionalidad de algunos objetos, como en el caso del CheckBox que hereda de Button y modifica su comportamiento para generar otro control distinto.

La relación de herencia entre clases se puede ver mejor en este diagrama UML:

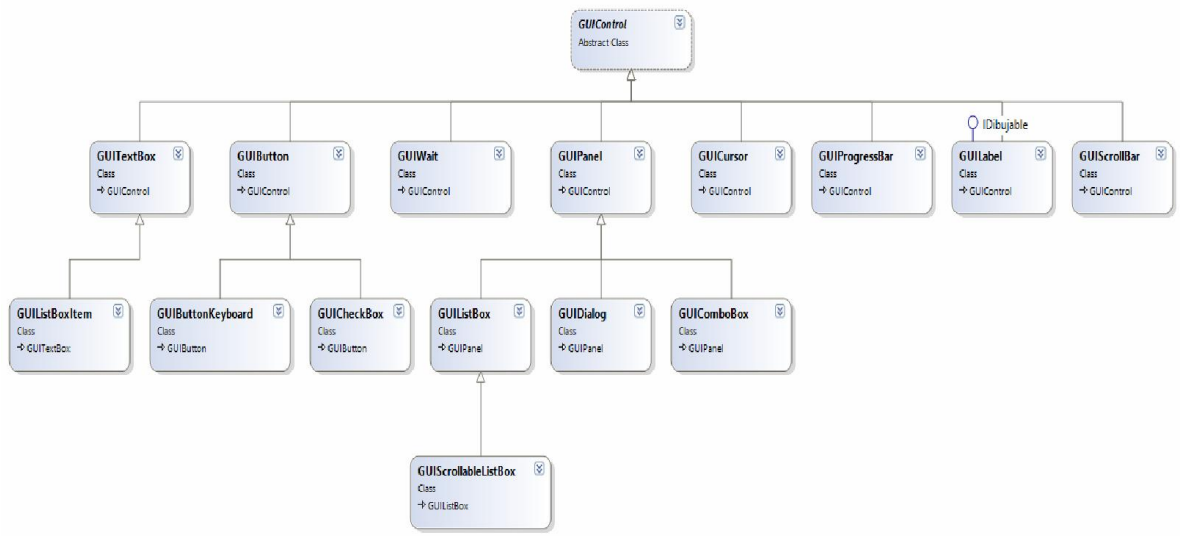
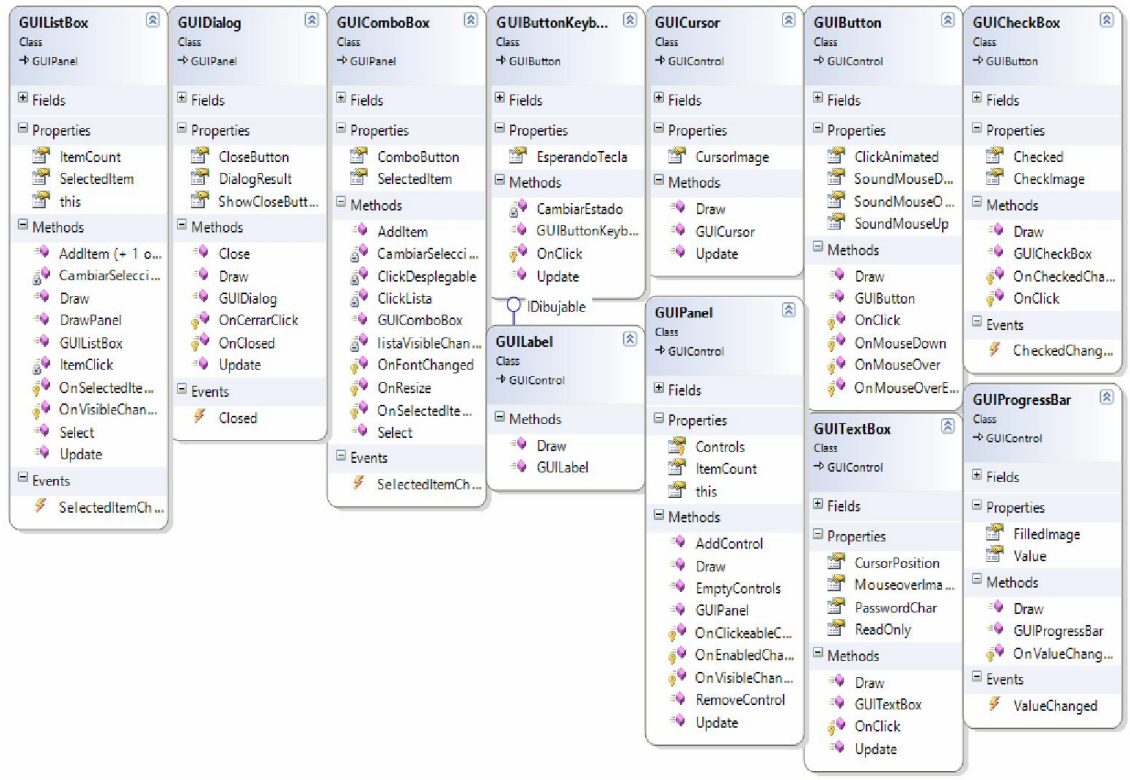


Diagrama de clases GUI

A la hora de diseñar las clases de objetos se han tenido en cuenta ciertos requerimientos específicos de la interfaz como son el nivel de transparencia, imágenes de fondo, tipo de fuente, estilos de borde, color/imagen para control deshabilitado y para cuando pasa el ratón por encima, sonidos asociados al click y al paso de ratón. También se ha intentado que el control responda a eventos típicos en una interfaz como son el click, el paso de ratón por encima del control, redimensionado, pérdida de foco, cambios en propiedades, etc.

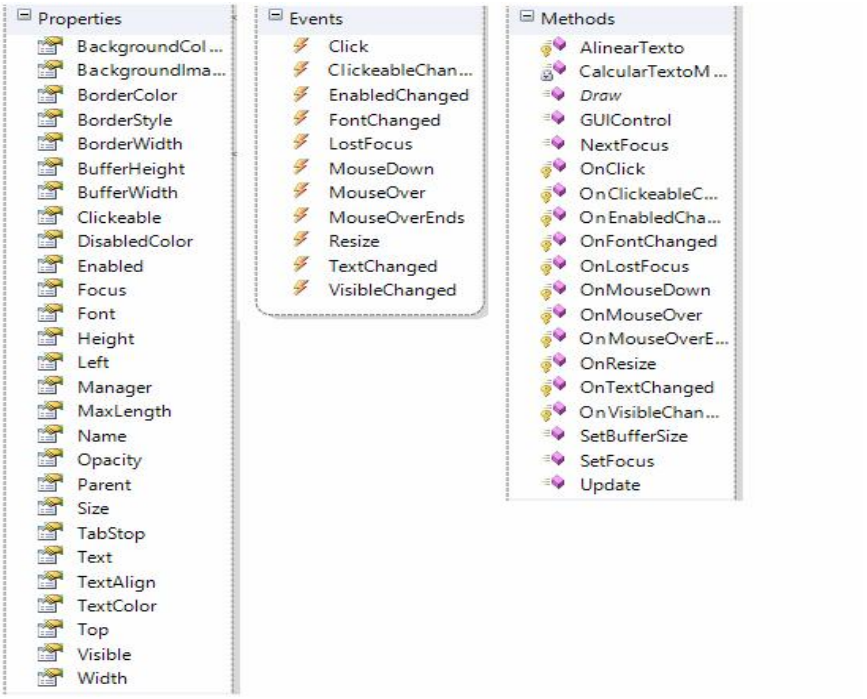
Se dispone de algunos campos estáticos en GUIControl que serán comunes a todas las instancias de objetos que hereden de esta clase abstracta, que ofrecen funcionalidades deseadas en una interfaz como mantener el único control posible que dispone del foco, o el control que está siendo presionado en un momento dado, pasando por datos de interés general para todos los controles como el tamaño del buffer de la pantalla evitando así tener que pasar a cada instancia estos datos que deberán ser inicializados antes de que los controles sean usados, preferentemente al inicio de la ejecución del videojuego.

Se muestra el contenido de las clases en la siguiente imagen:



Contenido de las clases GUI

En esta otra imagen se puede ver el contenido de la clase base abstracta GUIControl:



Clase abstracta GUIControl

6.2. Caso específico

Una vez se dispone de la librería con los componentes básicos, es necesario disponer de un mapa de navegación que nos muestre la relación que tiene cada pantalla o menú con otros y poder saber así las transiciones que se deben programar y las restricciones que se deben aplicar. Este diagrama también nos ayudará a saber cuantas pantallas va a tener nuestro programa y junto a los mockups de las mismas y a los requisitos relacionados, nos va a permitir desarrollar todo el sistema de interfaz y hacerlo funcional.

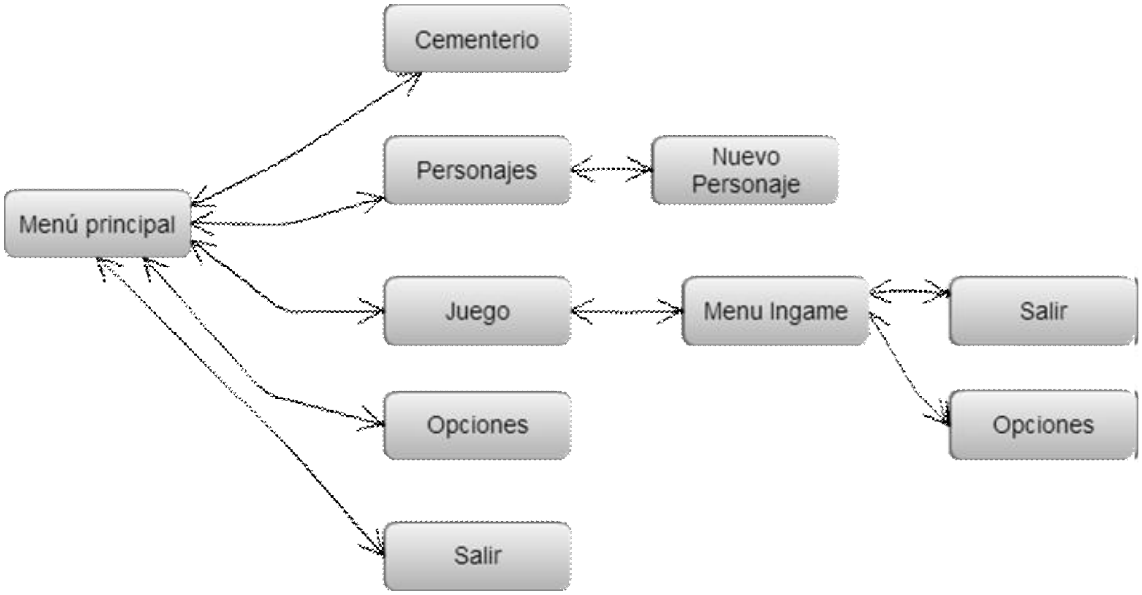
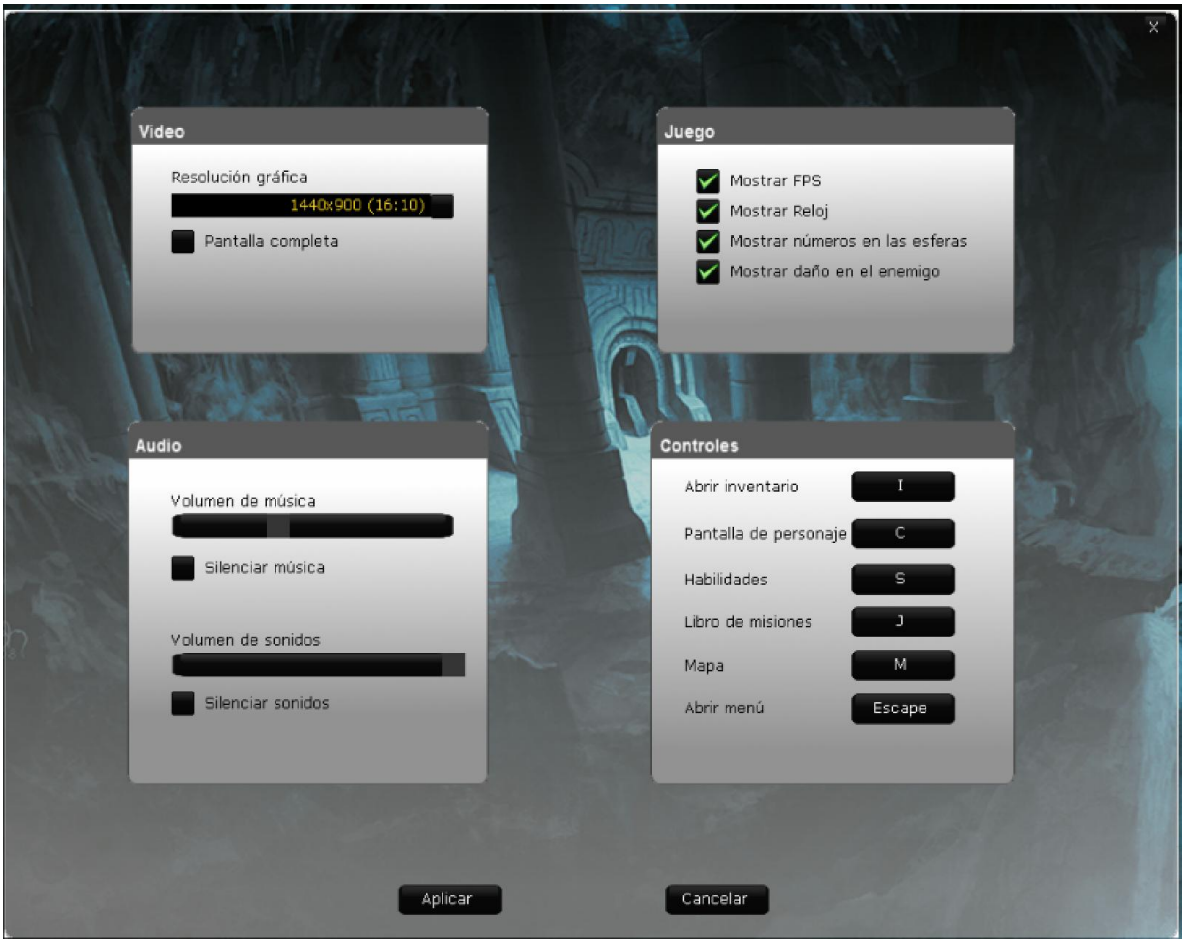


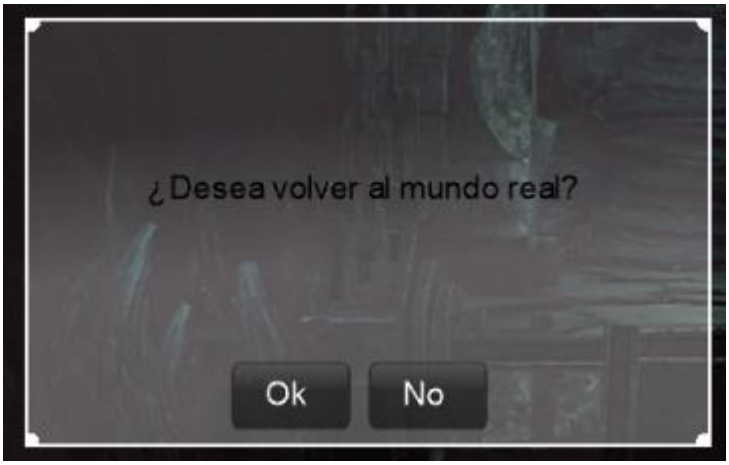
Diagrama de navegabilidad de la interfaz gráfica.

Se adjuntan en las siguientes páginas algunas capturas de los diferentes elementos de la interfaz gráfica que aparecen en el diagrama de navegabilidad una vez que ya han sido implementados.

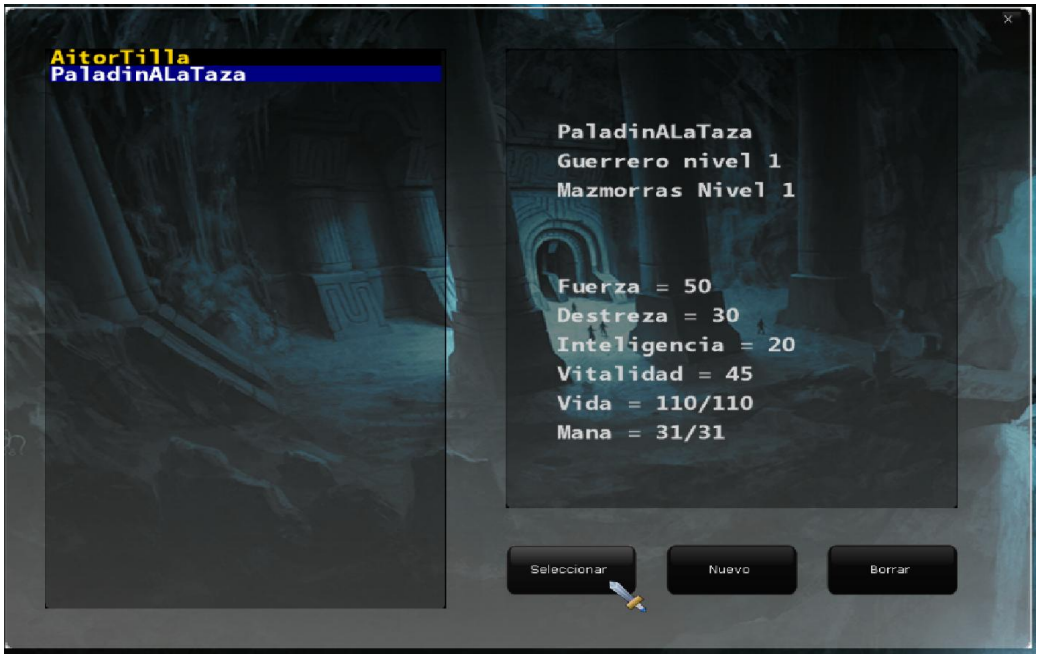
Menú principal.



Menú de Opciones.

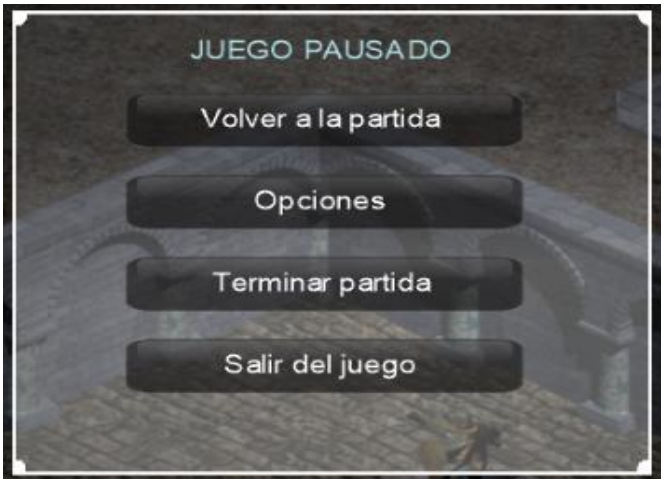


Confirmación de salida del juego.

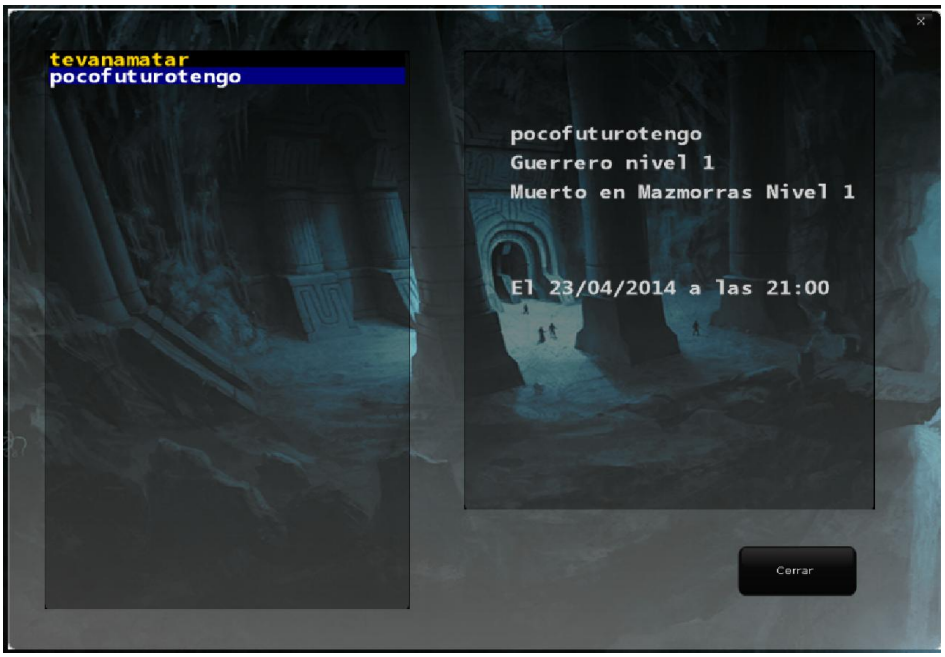


Menú de selección de personajes.

Creación de nuevo personaje.



Menú ingame.



Menú de héroes caídos en combate.

Para que exista navegabilidad entre los diferentes elementos de interfaz será necesario separar esta navegabilidad en dos tipos, el primero de ellos consta de las pantallas enteras de las que solo puede existir una simultánea y que heredan de XNAScreen, por lo que la transición entre una y otra se puede hacer simplemente llamando al método heredado base.changeScreen() con la ventana nueva a la que efectuar la transición para darle el control a esta y liberar recursos de la actual que realiza la llamada. De este tipo solo se van a usar dos, el FormPrincipal e InGame, la primera contendrá al menú principal más el fondo de pantalla y en la segunda se efectuará la lógica del juego.

El segundo tipo de navegabilidad lo vamos a tener mostrando y ocultando menús, teniendo una clase heredada de GUIPanel por cada menú necesario y una clase ControladorPaneles que se encargue de mantener y dibujar todos los menús que se le añadan. Cuando se llama a un menú, se puede suscribir al evento OnClose para que cuando ese menú haya terminado y se cierre, le devuelva el control al método llamador y se pueda controlar la información retornada por el menú abierto, como en el ejemplo de selección de personajes, que cuando se ha abierto el menú de selección y se ha elegido uno, al volver el menú que lo ha llamado debe conocer cual ha sido la selección para poder seguir trabajando. En este tipo de transiciones estarán el resto, pues la ventana FormPrincipal va a tener un ControladorPaneles al que se le añade inmediatamente un MenuPrincipal de tipo GUIPanel, como este menú tiene el control, puede añadir otros menús cuando se le soliciten haciendo click en sus botones como son el MenuOpciones, MenuCementerio, MenuPersonajes, el dialogo de confirmación de salida de tipo MessageBox o realizar la transición de formulario del actual al InGame y comenzar así realmente la partida, siempre y cuando se haya seleccionado un personaje.

6.3. HUD Ingame

Una vez ha iniciado la partida, esto es, se ha dado el control a la pantalla InGame, se necesita crear un HUD basado en los controles de la interfaz genéricos, que se dibujará siempre por encima del renderizado del juego y siguiendo el orden:

Juego → HUD → Menú → Ratón.

En el HUD, todas las partes se pueden mostrar u ocultar simultáneamente mediante atajos de teclado configurados en la sección de controles de las opciones o bien desde botones en la barra rápida de estado. La visibilidad de alguna de estas partes puede estar además condicionada con otras opciones distintas a las de controles, como ocurre con la opción que permite mostrar u ocultar los FPS o el reloj.

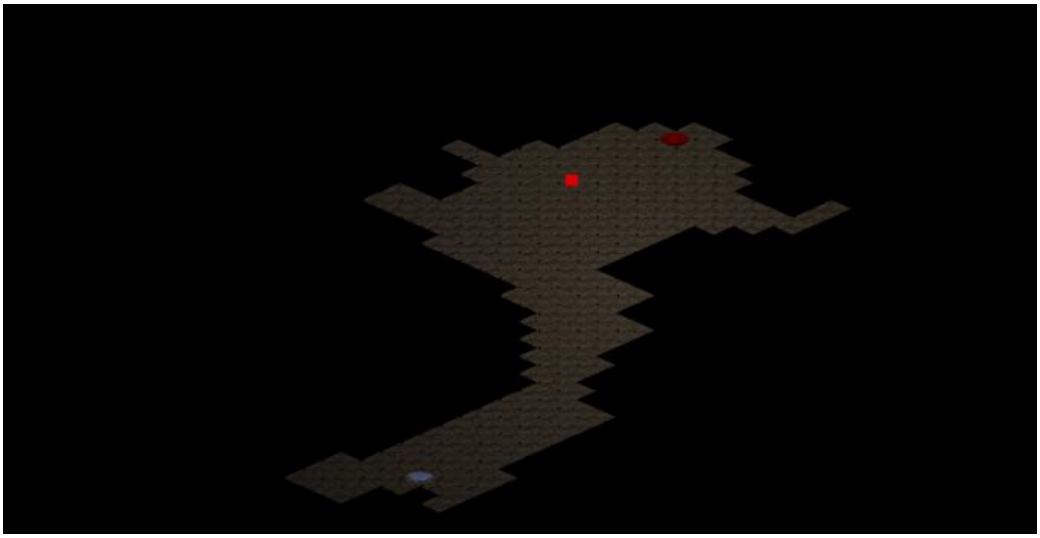
Nombre de mazmorra y hora actual

Barra rápida con vida y maná del jugador

La barra rápida de estado muestra en todo momento la vida y el maná actuales del jugador sin necesidad de abrir otro menú como el de atributos para acceder a ese dato y además permite la visualización de la cantidad actual de pociones de vida y maná que el jugador podrá usar utilizando las teclas 1 y 2 para incrementar en un 60% la vida o maná del jugador dependiendo del tipo de poción usada.

Menú de atributos del jugador.

En el menú de atributos del jugador (que se abre por defecto con la tecla C) se muestran los valores actuales de los atributos del héroe como fuerza, destreza, inteligencia o vitalidad, y los efectos que tienen en el personaje como el daño, la mitigación del daño entrante, el maná o la vida del personaje. También se incluye el nombre y la clase del jugador, así como el nivel actual y la experiencia del nivel actual junto con la experiencia necesaria para obtener un nivel extra. En el caso de que el personaje suba de nivel, se añaden 5 puntos de atributos a 'Puntos disponibles' y se le permite al usuario que los distribuya entre los 4 atributos principales del héroe (Fuerza, Destreza, Inteligencia, Vitalidad), poniendo la experiencia del nivel actual a 0 y recalculando la experiencia necesaria para el siguiente nivel.



Mapa descubierto

Si se pulsa la tecla por defecto M, se muestra el mapa actualmente descubierto por el héroe, lo que ayudará al jugador a no dar vueltas en círculos por el mapa ya que puede comprobar si ya ha pasado por una zona antes o no. El héroe se representa por un punto rojo y los puntos de entrada y salida del mapa se representan por círculos azul y rojo respectivamente.

Inventario del jugador

Si se pulsa la tecla por defecto I, se muestra el inventario del jugador, formado por 6 slots en la parte superior que permiten equipar o desequipar un objeto equipable de tipo casco, armadura, pantalón, botas, escudo y arma, asegurando que un objeto de un tipo solo puede ser equipado en su correspondiente casilla, lo que evita que el jugador pueda equipar una armadura en el hueco destinado a las botas.

En la parte inferior se dispone de una matriz de 21 slots formada por 3 filas de 7 columnas en las que se van acumulando los objetos aleatorios que se encuentran durante la aventura y que pueden o bien ser descartados arrastrándolos fuera de la ventana del inventario o bien equipados si son arrastrados al slot correcto de la parte superior y también se permite mover el objeto de una casilla inferior a otra inferior vacía o a otra ocupada por otro objeto, en cuyo caso se intercambiarán las posiciones de ambos en la matriz inferior. Equipar o desequipar objetos modifica directamente los atributos del jugador según los modificadores que tenga el objeto, atributos que pueden ser visualizados colocando el ratón por encima del objeto, mostrándose un pop-up con la información del mismo, incluyendo el nombre, una lista con los modificadores de atributos y la rareza del mismo representada por el color del nombre.

También se incluye la cantidad de oro del héroe justo debajo de la matriz de objetos.

Capítulo 7

Módulo generación de mapas

En esta sección se va a profundizar en la generación aleatoria de los mapas del juego roguelike, realizando previamente una recopilación y análisis de algunos de los algoritmos usados para esa tarea en la actualidad y seleccionando de ellos los que cumplan con los requerimientos del proyecto, así como modificando los algoritmos seleccionado para adaptarse a la visualización seleccionada.

En cuanto a tipos de mapas distintos, deben existir tres tipos base:

- Mapas predefinidos/estáticos

- Mapas de cueva aleatorios

- Mapas de mazmorra aleatorios

En los mapas aleatorios (cuevas y mazmorras) se debe poder obligar al generador aleatorio a que existan determinadas habitaciones o zonas estáticas predefinidas para niveles tipo enemigo final de zona o misión, lo que asegurará que, pese a la aleatoriedad de la generación, el mapa será jugable.

7.1. Mapas predefinidos / estáticos

Los mapas predefinidos o estáticos podrán obtenerse de ficheros de texto y serán usados en principio para zonas como la ciudad inicial del héroe, que siempre tendrá la misma forma, y se reserva la posibilidad de usarlo en otros lugares durante la aventura como por ejemplo las zonas con enemigo final anteriormente comentadas.

7.2. Mapas de cueva aleatorios

Estos mapas tienen un aspecto menos lineal y cuadrado que los de tipo mazmorra, debido principalmente a la ausencia de habitaciones y serán usados para simular lugares no creados por el ser humano como por ejemplo cuevas.

Una característica necesaria por el algoritmo seleccionado para su generación es que exista conectividad entre el punto de entrada y el de salida de la cueva, para asegurar que el mapa sea jugable, siempre y cuando sea un mapa de subniveles. Si el mapa es de solo exploración y no vamos a poder viajar a más subniveles, podremos obviar esta comprobación.

Ejemplo de mapa tipo cueva

A continuación se listan diferentes algoritmos para su creación, con detalles, pros y contras, empezando por los procedurales (PCG):

Drunkard Walk

Es un algoritmo muy sencillo que genera unos resultados aceptables en un tiempo reducido y que nos asegura que el mapa va a tener conectividad y no van a existir islas no conectadas en nuestro mapa generado.

Los pasos del algoritmo son los siguientes:

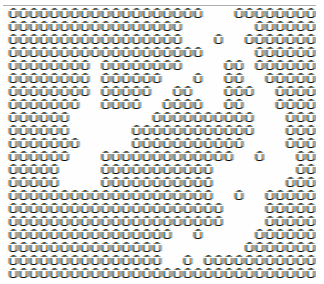
1.- Rellena toda la matriz bidimensional de mapa con casillas muro/ocupadas.

2.- Selecciona un punto aleatorio, preferiblemente de la zona central del mapa y márcalo como un tile vacío, donde se podrá caminar.

3.- Elige una direccion N,S,E,W aleatoria

4.- Moverse en esa direccion, y marcar el tile vacío, salvo que ya lo fuera

5.- Repetir los pasos 3-4 hasta que se tengan destapados el nº de tiles deseados.



Resultado del algoritmo *Drunkard Walk*

Uno de los problemas que tiene este algoritmo es que el único control que se tiene sobre la generación en un principio es el número de tiles que quieres destapar, aunque con unas modificaciones se podrían incorporar otros parámetros a la creación del mapa, como por ejemplo forzar la inclusión de habitaciones o zonas predefinidas.

Autómatas celulares (Cellular automata)

Este tipo de algoritmos se basan en la evolución del mapa desde su origen aleatorio más primigenio hasta su evolución final.

Hay diferentes maneras de abordar este tipo de algoritmos, pero voy a explicar la que usa la "regla 4-5":

1.- Crear 2 arrays bidimensionales vacíos del tamaño X,Y del mapa a generar

2.- Rellena aleatoriamente el primer array con muro/vacío, usando un porcentaje de probabilidad, por ejemplo 40/60.

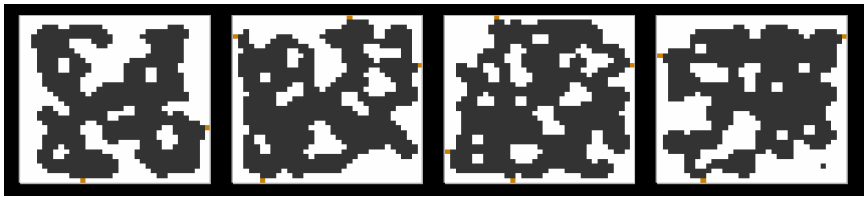
3.- Rellena los bordes del array con muro

4.- Por cada punto del mapa, cuenta los 9 vecinos (contando diagonales) que sean muro y seguir estas reglas:

4.1 Si el punto es muro y al menos 4 de sus vecinos también lo son, o el punto no es muro pero al menos 5 de sus vecinos si lo son, marcar como muro en el segundo array.

4.2 En cualquier otro caso, marcarlo como suelo/vacío en el segundo array.

5.- Copiar el segundo array en el primero y volver al punto 4 entre 4 y 7 veces.



Resultado final del algoritmo *Autómata celular*

El problema de estos algoritmos es que son algo más lentos que otros como el drunkard walk, aunque el tiempo dependerá en gran medida del número de pasadas que se le vayan a hacer al algoritmo y al tamaño del mapa, pero en este caso el número no es muy alto y podría ser aceptable temporalmente hablando.

Otro de los problemas que tiene este algoritmo es que no asegura conectividad, por lo que pueden aparecer islas no conectadas en el mapa o no existir camino entre la entrada y la salida de la cueva.
Este último problema se puede solventar realizando otra pasada con algoritmos de Pathfind como A* para comprobar que el resultado es conexo e ir destapando un camino entre las dos zonas inconexas.

Delving

Este algoritmo está a mitad de camino entre uno procedural y un autómata celular, como los anteriormente vistos en este apartado, lo cual permite obtener las ventajas de uno y de otro.

El algoritmo tiene una serie de parámetros que nos permitirán modificar su resultado:

· ngb_min (entre 1 y 3): Mínimo número de vecinos de tipo suelo que un muro debe tener para convertirse también en suelo.

· ngb_max(entre ngb_min y 8): Máximo nº de vecinos de tipo suelo que un muro debe tener para convertirse también en suelo.

· cchance: Porcentaje de probabilidad de que se permite una nueva conexión.

· max: Máximo número de casillas a destapar con el algoritmo.

Se necesita una lista de casillas en la que se permiten repeticiones y que tiene dos operaciones básicas, colocar una nueva casilla en la parte superior y extraer una de la lista. La extracción será aleatoria si hay menos de 125 casillas en la lista o en cualquier otro caso de las 25 * raizcúbica(num_casillas) primeras (dado que raiz3(125)=5*25=125).

Pasos del algoritmo:
1.- Rellenar todo el mapa inicial con muro, menos unas semillas de suelo.

2.- Colocar las casillas muro de los vecinos de suelo en la lista

3.- Sacar una casilla de la lista

4.- Si la casilla sacada tiene un numero de vecinos muro entre ngb_min y ngb_max y o bien al hacerla suelo no crea nuevas conexiones o se permite una conexión basandose en cchance, entonces se convierte la casilla en suelo y sus vecinos muro se añaden en orden aleatorio a la lista de casillas.

5.- Repetir desde el paso 3 hasta llegar al máximo número de casillas a convertir en suelo o no queden casillas en la lista que procesar.

Existen algunas variantes como por ejemplo en el paso 3 seleccionar siempre de todo el conjunto aun siendo más de 125, lo que dará como resultado mapas más compactos, o otra variante sería guardar los vecinos en el sentido de las agujas del reloj en vez de aleatoriamente, lo que generaría mapas con forma espiral.

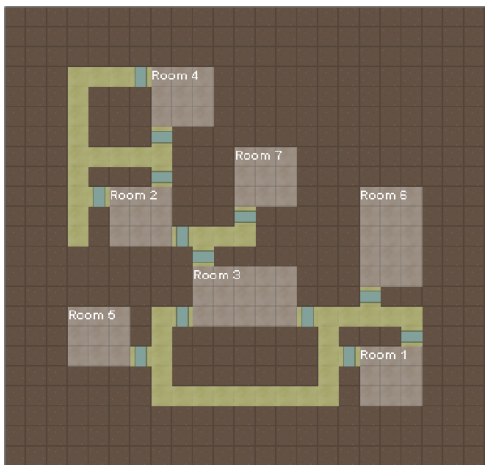
Un ejemplo del resultado generado por el algoritmo se puede ver en las siguientes imágenes, se han inicializado de distinta manera los valores para ngb_min, ngb_max y cchance.

Resultado del algoritmo *delving* con diferentes parámetros.

7.3. Mapas de mazmorra aleatorios

Este otro tipo de mapas presenta una estructura mucho más lineal que los de tipo cueva y contiene una cantidad finita de habitaciones bien definidas interconectadas mediante pasillos.

En este tipo de mapas también se debe asegurar la conectividad entre la entrada y la salida si es un mapa de subniveles. Si el mapa es de solo exploración y no vamos a poder viajar a más subniveles, podremos obviar esta comprobación.



Ejemplo de mapa de tipo mazmorra o dungeon.

Se listarán a continuación diferentes algoritmos usados para la generación de este tipo de mapas específicos:

Colocación aleatoria de habitaciones

Es uno de los algoritmos más sencillos para la generación de este tipo de mapas, se basa en colocar las habitaciones de tamaño aleatorio en posiciones aleatorias con un número aleatorio de puertas y luego usar un algoritmo de pathfind como A* para crear los pasillos interconectando las puertas de las habitaciones con él. Los pasos son:

1.- Determinar el nº de habitaciones a crear

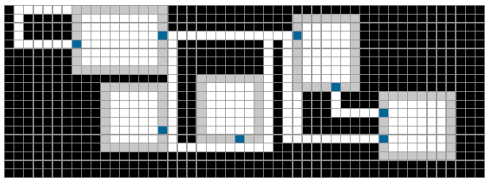
2.- Crear una habitación de tamaño aleatorio y posición aleatoria, colocar un nº aleatorio de puertas en los bordes de la sala (limitable por parámetro)

3.- Comprobar si la habitación está dentro de los límites del mapa y no solapa con otras habitaciones previamente generadas. Borrar la sala si no se cumple.

4.- Repetir paso 2 hasta que se creen todas las habitaciones

5.- Usar A para unir las puertas, los pesos de A* se usan para hacer que los pasillos sean más propensos a ir rectos y unirse a otros pasillos*

6.- Si queda alguna puerta sin conectar, o bien se omite la puerta o se crea un pasillo aleatorio.



Resultado del algoritmo

Este algoritmo permite la inserción de habitaciones predefinidas, añadiéndolas antes del paso 1, y se pueden parametrizar algunas opciones como el nº máximo de habitaciones, de puertas por habitación o el tamaño de las habitaciones.

BSP Tree

Usando árboles binarios para el particionado del espacio o BSP, nos aseguramos que la división del espacio no solapará en ningún momento, por lo que podemos aprovecharlo para añadir habitaciones sin que en ningún momento tengamos que descartar ninguna por que ya exista otra en esa posición.

Este algoritmo está formado por los siguientes pasos:

1.- Empezar con una mazmorra rectangular

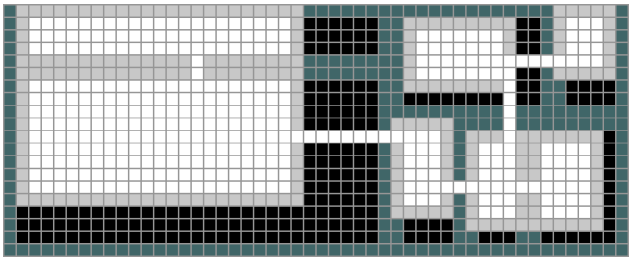
2.- Elegir aleatoriamente una dirección (horizontal o vertical) y una posición (x para vertical, y para horizontal)

3.- Divide la mazmorra en dos sub-mazmorras usando los datos del paso 2

4.- Selecciona aleatoriamente una sub-mazmorra y repite el paso 2, N veces

5.- Después de N iteraciones, colocar habitaciones de tamaño aleatorio en cada una de las particiones

6.- Conectar las habitaciones recorriendo las subdivisiones



Resultado del algoritmo BSP.

Este algoritmo también permite la inserción de habitaciones predefinidas, siempre y cuando se realice en una subdivisión en la que quepa (por ejemplo en la primera).

No se tiene mucho más control sobre la generación que el nº de divisiones.

Construcción procedural

En este caso, se hace "crecer" al mapa añadiéndole habitaciones desde el centro, lo que genera unos resultados visuales muy buenos con un coste computacional pequeño. Los pasos generales del algoritmo son los siguientes:

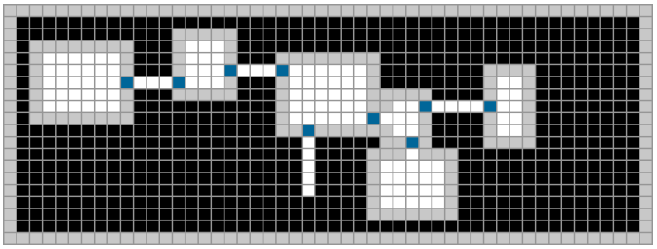
1.- Rellena todo el mapa con muro

2.- Coloca una habitación de tamaño aleatorio en el centro

3.- Selecciona una pared aleatoria de cualquier habitación

4.- Crear una nueva habitación y si cabe, ponerla en la dirección seleccionada

5.- Repetir al paso 3 hasta que se hayan generado las habitaciones solicitadas.

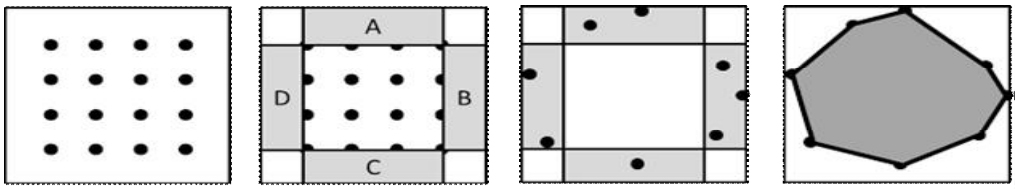


Resultado del algoritmo procedural

También se permite añadir habitaciones predefinidas, simplemente añadiéndolas a la lista de habitaciones a seleccionar por el paso 3, o bien siendo la habitación central.
Se pueden modificar el nº de habitaciones, el nº máximo de puertas y el tamaño máximo de las habitaciones.

Mejoras

Una posible mejora en este tipo de mapas podría ser la inclusión de habitaciones con formas irregulares aleatorias, lo que puede darnos en algunas situaciones mapas mixtos de cueva-mazmorra:



La idea aquí es separar los bordes en 4 rectángulos, seleccionar aleatoriamente entre 1 a N puntos de cada rectángulo y unir los puntos usando el algoritmo de Bresenham, que es un algoritmo para dibujar rectas usando píxeles.

Una vez creada la habitación deberemos proporcionarle al menos una entrada, esto lo podemos hacer o bien a la hora de la creación, seleccionando en una cara los dos puntos más cercanos al borde del rectángulo o bien a posteriori, cambiando tiles de los bordes de muro a vacío para ajustarse a ese borde.

7.4. Algoritmos seleccionados y ajustes

Finalmente, de los algoritmos comentados previamente se han seleccionado dos para su uso en el proyecto, el algoritmo drunkard walk para mapas de tipo cueva irregular debido a su sencillez y a la calidad visual del resultado y el de construcción procedural para los mapas de tipo mazmorra, aunque en este caso todos los algoritmos se comportaban de una manera muy similar, el resultado visual de éste en concreto ha sido preferido.

Tras la selección de los algoritmos ha sido necesario modificar los dos algoritmos para que pudieran ser correctamente visualizados en perspectiva isométrica. En el caso del algoritmo drunkard walk, solo se tenía en cuenta dos casos posibles para cada una de las casillas del mapa, o bien era muro infranqueable o bien suelo usable, lo que se convierte enseguida en un problema en cuanto intentamos dibujar el mapa por pantalla, debido a que los bordes quedarían todos en ángulos de 90º sin suavizar y no se le podría dar altura en isométrico por que requeriría de más tiles para las curvas y las diferentes rectas.

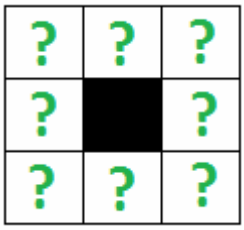
La solución para este problema reside en usar un algoritmo de auto-tiling para asignarle a cada casilla del mapa el tile correcto a pesar de que se genere el mapa realmente solo destapando casillas convirtiéndolas de 0 a 1, lo que nos va a ahorrar comprobaciones que harían impracticable el algoritmo, ya que controlar todas las posibles opciones de los 8 vecinos sería controlar 2^8 = 256 posibilidades. Para evitar esto vamos a tener primero que conocer que tiles se van a usar en la representación isométrica, haciendo una lista de 12 casos, que es mucho más manejable que los 256 casos que tendríamos si no usáramos este algoritmo:

0	Casilla vacía
1	Suelo
2	Esquina superior izquierda
3	Esquina superior derecha
4	Esquina inferior izquierda
5	Esquina inferior derecha
6	Pared norte
7	Pared este
8	Pared sur

9	Pared oeste
11	Esquina interior inferior izquierda
12	Esquina interior superior derecha

Tabla de tiles disponibles

El funcionamiento del algoritmo es el siguiente, cada vez que una casilla se destapa y pasa a ser de tipo suelo, se marcan todos sus vecinos adyacentes que no sean suelo a desconocido con el número -1.



Marcado de vecinos inicial

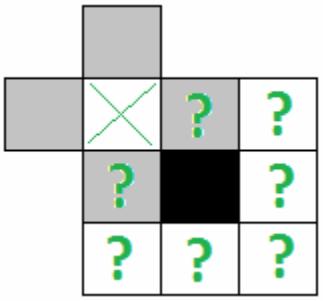
Para cada una de esas casillas marcadas, se realiza el siguiente procedimiento, se comprueba si desde esa casilla en su Norte, Este Sur y Oeste (sin mirar diagonales y en sentido horario empezando por el Norte) hay algo que no sea vacío y de ser así se apunta aparte un 1 o un 0 en caso contrario, formando así un número binario. Con este número binario se mira en la siguiente tabla de correspondencia binario-tile y se coloca el tile correspondiente en la casilla que se estaba evaluando.

Binario	Decimal	Tile
1110	14	6
1100	12	3
1101	13	7
1001	9	5
1011	11	8
0011	3	4
0111	7	9
0110	6	2
1111	15	11 ó 12 ó 1

Tabla de correspondencia binario-tile

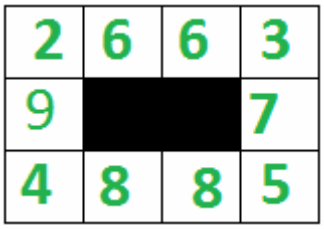
La tabla ha sido generada a base de probar los diferentes casos para los tiles que se usarán en el juego y anotando los resultados obtenidos. El caso 1111 es un caso especial en el que se controlará con condicionales si la casilla será suelo, esquina interior inferior izquierda (si su diagonal opuesta es 0) o esquina interior superior derecha (mismo caso que el anterior).

Para ayudar a la comprensión del algoritmo se muestra un ejemplo, suponiendo que destapamos una casilla y ya se han marcado los tiles adyacentes como interrogante, seleccionamos la casilla de la esquina superior izquierda y formamos el número con sus cuatro casillas vecinas marcadas en gris en la imagen:



Comprobación de vecinos de vecino

Empezamos por el Norte, vemos que la casilla está vacía y apuntamos aparte un 0, en la siguiente casilla en sentido de las agujas del reloj tenemos al Este, marcado con interrogante por lo que no está vacía y añadimos un 1 a lo que ya teníamos por lo que tenemos un 10 de momento. Siguiendo el sentido, miramos el Sur y ocurre lo mismo por lo que ahora tenemos 110 y finalmente llegamos al Oeste en el que no hay nada marcado y colocamos un 0 teniendo como resultado binario final el 0110, que si lo buscamos en la tabla de correspondencia binario-tile vemos que en decimal es un 6 y que le corresponde el tile 2, que es el de esquina superior izquierda. Ahora el algoritmo deberá realizar el mismo paso para cada uno de los 7 vecinos restantes a la casilla destapada.



Resultado de auto-tiling en la segunda iteración.

Para convertir de binario a decimal sobre la marcha en el algoritmo anterior de auto-tiling, se utiliza la siguiente fórmula en la que se va acumulando en decimal la suma de 2^i siempre que en esa posición se anote un 1.

$$decimal = \sum_{i=0}^{n-1} 2^i \quad \forall\, bin_i = 1$$

Binario a decimal

En el caso del algoritmo de construcción procedural de mazmorras, la única modificación necesaria ha sido el tener que identificar qué esquina es cada una, por que para la generación del mapa solo se necesita conocer si una esquina lo es o no y así poder comprobar si esa habitación que se intenta colocar cabe o no en el mapa actualmente generado, mientras que a la hora de dibujar en perspectiva isométrica el mapa resultante será necesario conocer cual es cada una de las esquinas para poder seleccionar el tile correcto, lo que se puede realizar de manera sencilla con 4 condicionales.

Los algoritmos de tipo autómata celular han sido desechados por que como ya se comentó, no aseguran conectividad entre el punto de entrada y de salida del mapa por lo que no son útiles para los dos tipos de mapas que se van a generar en el proyecto, aunque se ha visto que podrían ser usados en otros proyectos para dibujar mapas de exteriores como bosques por la forma de los mapas resultantes.

7.5. Visualización

Una vez decididos los algoritmos a usar y solventadas las modificaciones que van a ser necesarias para su correcto funcionamiento en el proyecto, se pasa a diseñar de qué manera se va a implementar. En este caso, como se tienen diferentes algoritmos y todos ellos son intercambiables, se puede usar el patrón GOF Strategy para poder definir una plantilla común a todos los algoritmos que permita intercambiar entre uno y otro sin mayor problema. Para ello se ha definido una interfaz IAlgoritmoGeneracion con la declaración de la parte común entre algoritmos de generación aleatoria y una clase específica para cada algoritmo. También es necesario añadir una clase que represente el mapa y que controle el acceso al array bidimensional interno donde se guardarán los tiles para cada posición, y

para el caso de mapas de tipo mazmorra se usarán objetos de tipo habitación que son un caso particular de sub-mapas con menor tamaño y por lo que se heredará de Mapa. Todos estas clases se engloban en una biblioteca de clases, proyecto aparte que se añade a la solución y que permite su reutilización en otros programas de los algoritmos de generación que contiene.

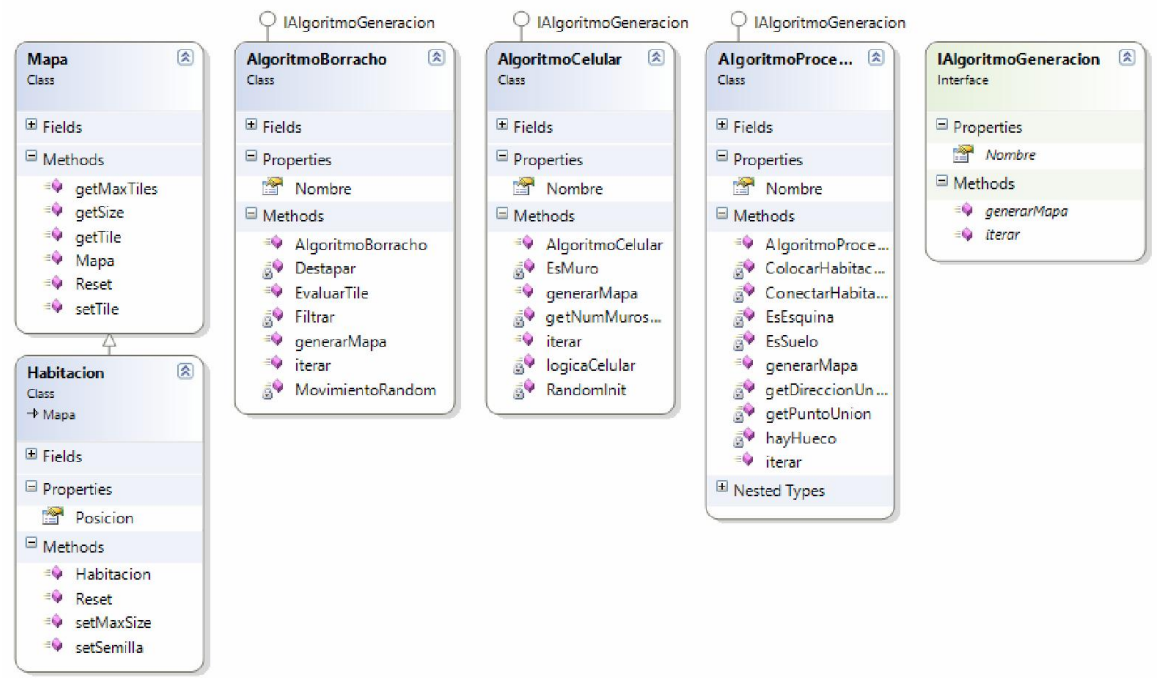


Diagrama de clases de la biblioteca de algoritmos.

En la parte común, todos los algoritmos tienen un método generarMapa que modifica el mapa pasado por parámetro con el algoritmo concreto, una propiedad Nombre que devolverá el nombre del algoritmo que se está usando y un método iterar que ejecutará solamente un paso del algoritmo y que ayudará en las pruebas y depuración de los algoritmos. Un ejemplo de uso en el que se aprecia la potencia de usar Strategy para este problema es el siguiente, donde se puede observar como es posible declarar una sola variable de algoritmo y asignarle dos algoritmos completamente distintos.

```
Mapa m = new Mapa(80,80,255) // Crea mapa
IAlgoritmo alg;
alg = new AlgoritmoBorracho(150);
alg.generarMapa(m);
alg = new AlgoritmoProcedural(100, 4, 4, 9, 11);
alg.generarMapa(m);
```

En este punto ya se pueden generar mapas aleatorios con los diferentes algoritmos, por lo que el siguiente paso es visualizarlos, para ello se ha creado una interfaz con los métodos que se consideran necesarios para una visualización mínima, para posteriormente crear la clase Visor2D que se encargará de representar el mapa generado en 2D pudiendo especificar la posición absoluta y el desplazamiento del mapa, añadir texturas de tiles en el orden a usar (en el de la tabla de tiles disponibles vista anteriormente en esta sección), asignar el objeto Mapa a representar, aplicar zoom y controlando dos de los métodos imprescindibles a la hora de dibujar entidades en XNA como son Update y Draw.

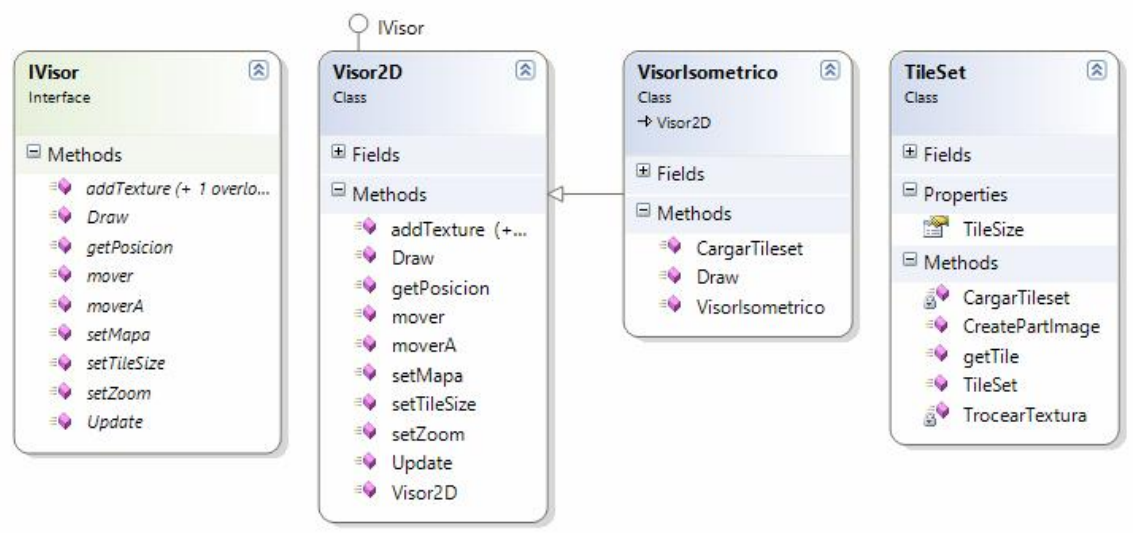
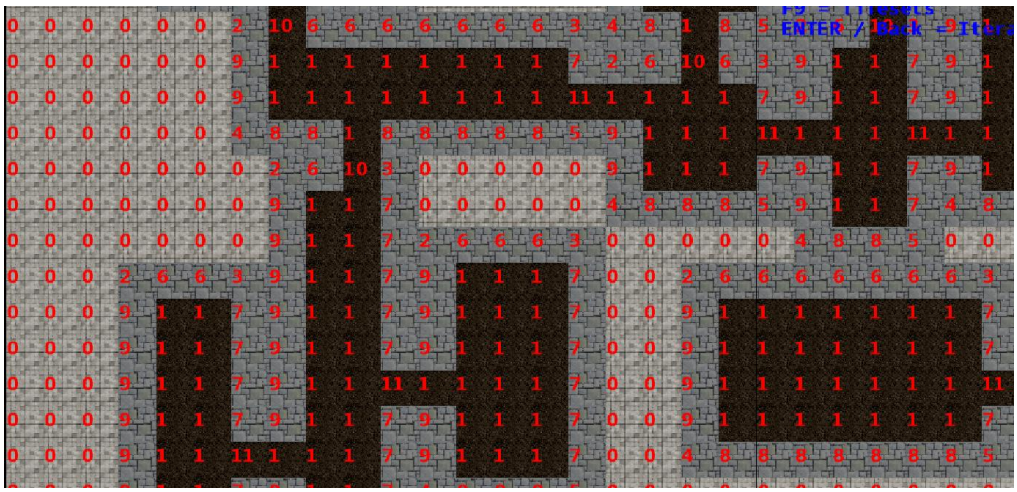
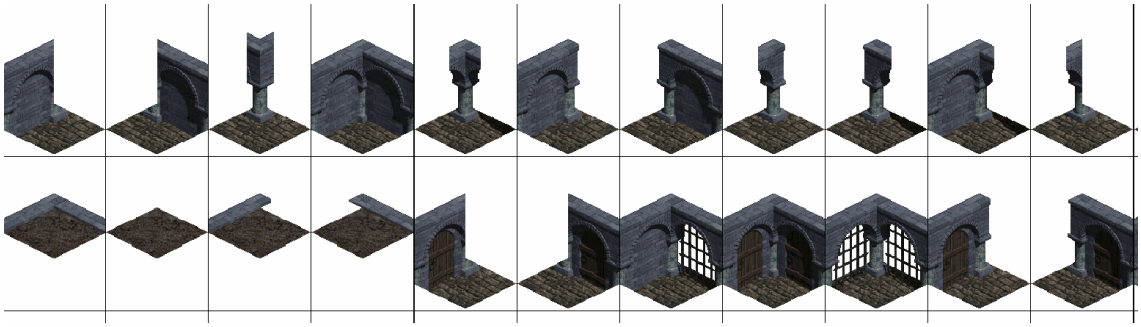


Diagrama de clases de los visores de algoritmos

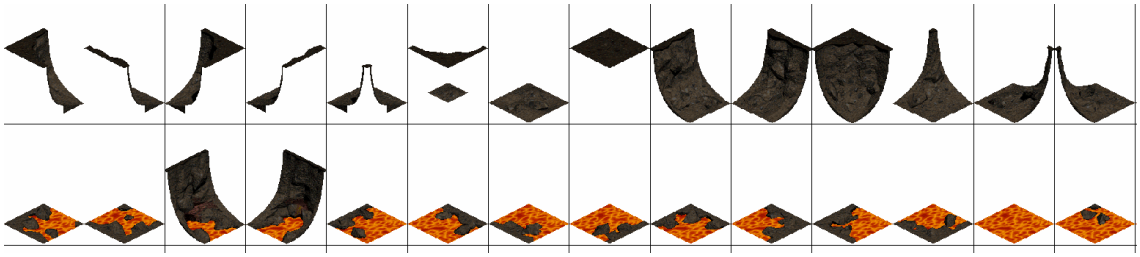


Visualización 2D de mazmorra con número de tiles.

La clase TileSet ha sido introducida para mantener en memoria todos los tiles referentes a una zona representada por una imagen que contiene todos los tiles, como puede ser cueva o mazmorra, permitiendo el fácil y rápido acceso a cada uno de los componentes y principalmente asegurando una consistencia en la numeración de los tiles, puesto que por ejemplo el tile del suelo no tiene por que estar en la misma posición en la imagen de tiles del tileset de cueva que en el de mazmorra, pero a la hora de acceder al tile es vital que siempre sea el tile número 1. Como todos los tiles tienen un tamaño de 128x192 ,aunque intuitivamente al ser isométricos debieran ser 128x64 por tener que ser el doble de anchos que de altos, al representar estos tiles alturas con los muros, con 192 de altura permite al juego representar 3 capas de altura (suelo, muro y techo), lo primero que hace la clase tileset es cargar un fichero .cfg que contiene toda la especificación del tileset como tamaño en casillas y ruta de la imagen que contiene todos los tiles, así como las posiciones x,y de cada uno de los tiles, en orden, para que la clase pueda cortar de la imagen del tileset todos los tiles especificados.



Parte del Tileset isométrico de mazmorra



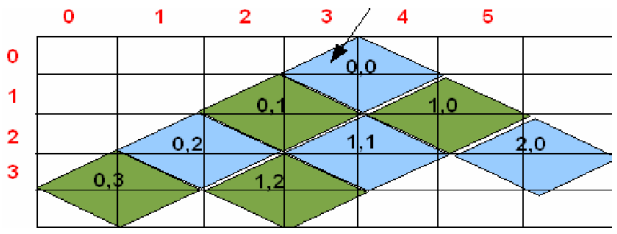
Parte del Tileset isométrico de cueva

Con el tileset isométrico ya cargado en memoria, se le asigna a la clase VisorIsométrico para que usando la matriz bidimensional contenida en un objeto Mapa ya modificado por un algoritmo de generación, pueda mostrar el mapa isométrico del mismo, no sin antes tener en cuenta algunos problemas que surgen al representar en isométrico y que no teníamos anteriormente.

El primer problema viene al posicionar cada tile, ya que primero hay que convertir las coordenadas de entrada de pixeles a isométrico para poder dibujar el tile en su posición correcta, para lo que utilizaremos las siguientes operaciones.

```
float xCo = (y + x) * (tilesize.X / 2) * 0.98f * zoom + offsetx;
float yCo = (x - y) * (tilesize.Y / 2) * 0.98f * zoom + offsety;
```

xCo e yCo pasarán a ser las coordenadas cartesianas x,y que se usarán para dibujar el tile que estaba en la posición del array del mapa (x,y), 0.98 es un factor de ajuste para suavizar bordes, zoom especifica un factor de ampliación o reducción de escala con valor por defecto a 1.0 y finalmente offsetx y offsety representan el desplazamiento del mapa con respecto al origen, lo que permitirá mover el mapa posteriormente.



Posible orden x,y en isométrico

El segundo problema viene al dibujar los tiles sin un orden específico, esto no era un problema en la representación 2D cartesiana, pero en la isométrica al estar la perspectiva girada, sin un orden se da el caso de que un tile de tipo suelo se pinte encima de un muro, dando origen a fallos en la visualización como se muestra en la siguiente imagen:

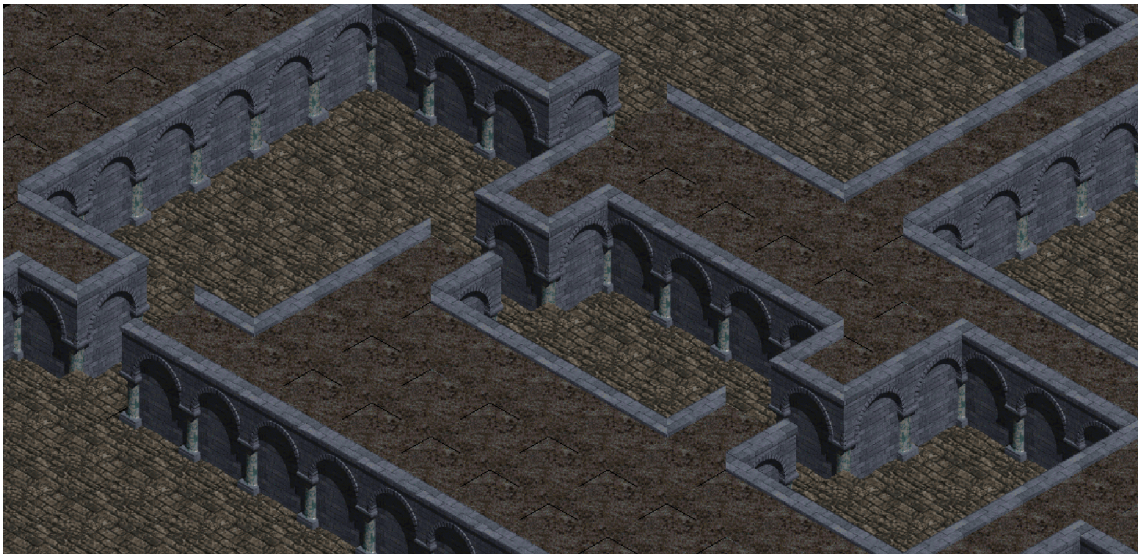
Fallos al renderizar sin orden en isométrico.

Para solventar este problema, hay que asignarle a cada tile un valor de profundidad Z, ya que a pesar de estar trabajando en 2D, la perspectiva isométrica es en realidad 2.5D o falso 3D, lo que significa que es una emulación de 3D usando 2D, por lo que en realidad los objetos en isométrico tienen una determinada profundidad, que será usada posteriormente para ordenar de manera descendente los tiles por su profundidad (que estará normalizada a valores entre 0.0 y 1.0, siendo 1.0 lo que se pintará antes que los valores inferiores, hasta llegar al 0.0 que será lo último que se dibuje.

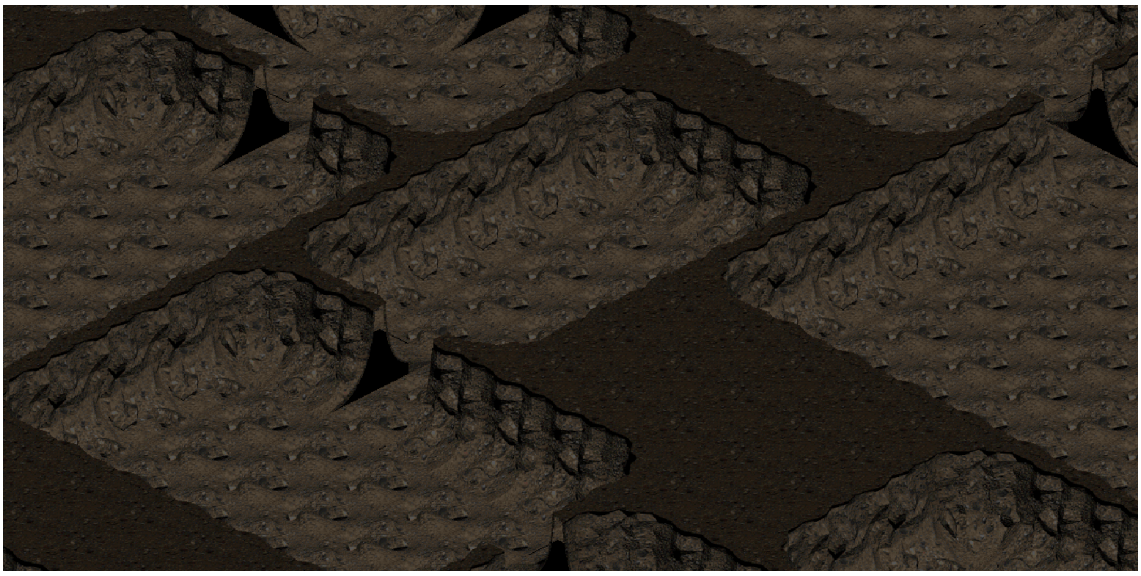
El orden correcto en coordenadas isométricas será de izquierda a derecha y de arriba a abajo, por lo que hay que calcular con un par de fórmulas estos valores que serán dependientes de la posición del tile en la matriz bidimensional del mapa. Para dibujar por capas, se le ha dado a todo tile de suelo el valor Z = 1.0, para asegurarnos de que el suelo sea lo primero en ser pintado, para posteriormente asignar un valor calculado de profundidad a los tiles de muro (en este paso también se le debe asignar un valor a los objetos y personajes que pertenezcan al mapa, de una manera simular, como se verá en siguientes apartados, voy a centrarme en el aspecto del mapa solamente aquí). Posteriormente el spriteBatch se encargará de dibujarlos usando este orden, como veremos en el siguiente capítulo.

```
maxdepth = ((map.getSize().X + 1) * ((map.getSize().Y + 1) * 128))/20;
depthOffset = 0.7f + ((-x + (y * 128)) / maxdepth);
```

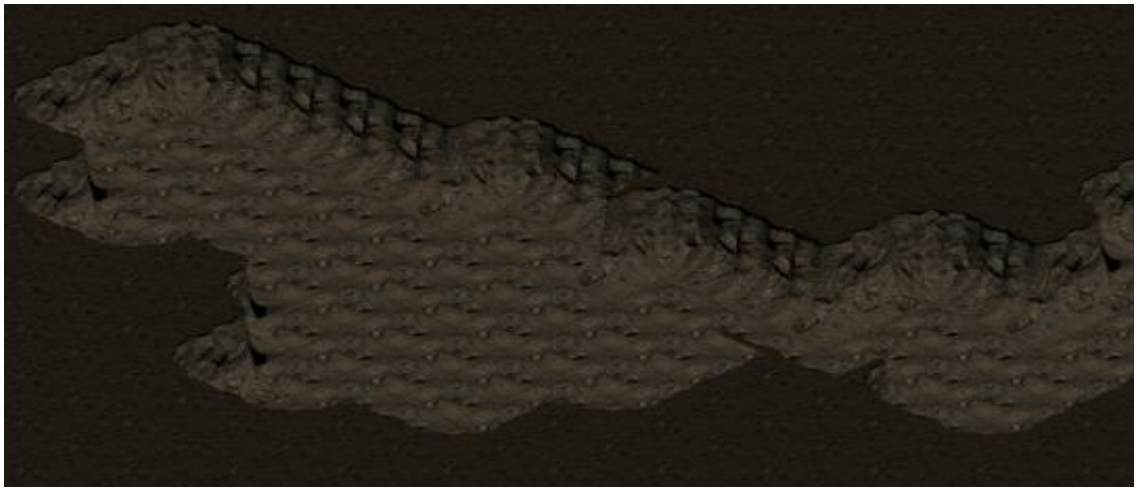
Se muestran como ejemplo de uso del visor isométrico, 3 imágenes que contemplan los tipos de mapas aleatorios distintos representados en perspectiva isométrica que se usarán en el juego.



Visualización de mapa de tipo mazmorra

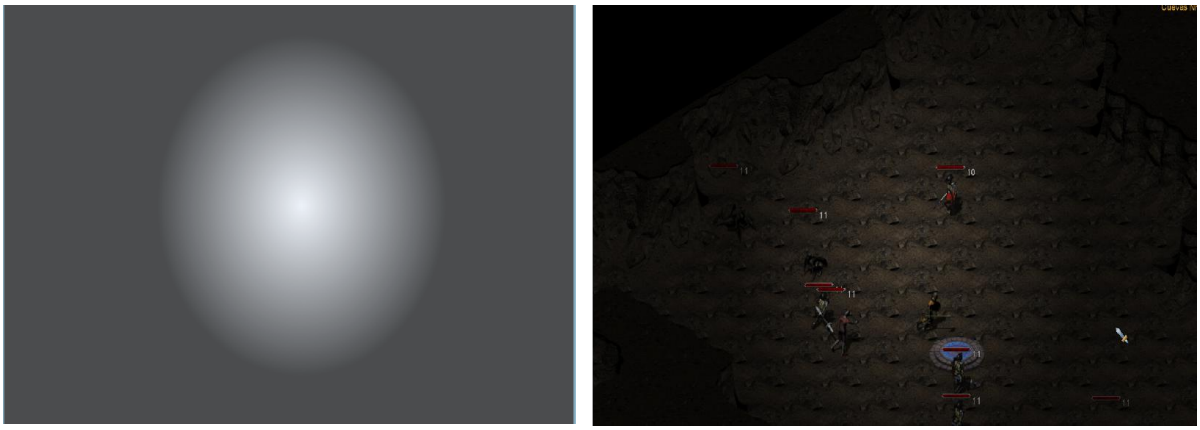


Visualización de mapa de tipo mazmorra con tile de cueva



Visualización de mapa de tipo cueva irregular

Una vez se tiene el mapa visualizado, es deseable buscar algún efecto que mejore la credibilidad y la calidad del render. En este proyecto se ha implementado una manera muy sencilla y eficiente de aplicar un efecto de iluminación al visor isométrico, proporcionando ese toque de oscuridad que favorece a juegos de este estilo. El efecto consiste en crear con un programa de diseño gráfico una imagen del tamaño del buffer de pantalla sin redimensionar que en este caso será 1600x1050 y aplicarle un efecto concéntrico de gradiente de más transparente en el centro del círculo y más oscuro cuanto más lejos del centro se encuentre. Con esta imagen solo resta que una vez se ha dibujado todo el mapa con el visor se dibuje encima la imagen con el efecto de semitransparencia de manera que como al moverse el jugador en realidad permanece en el centro de la pantalla y lo que desplaza es al resto de entidades, el centro del efecto seguirá siempre al héroe.



Efecto de iluminación y su efecto al aplicarse.

Capítulo 8

Módulo animaciones

Aunque no es estrictamente necesario que un juego de este estilo disponga de animaciones ya que como se vio en otros capítulos anteriores, el juego original Rogue estaba basado en una representación con caracteres ASCII, si que es un plus disponer de un módulo que otorgue al juego de animaciones, ya sea del personaje y enemigos o de objetos del entorno. En este capítulo se detalla todo lo que ha sido necesario desarrollar para tener funcionando el módulo de animaciones en el proyecto, siempre con la idea de que sea modular y reutilizable en otros proyectos futuros.

En su expresión más simple, una animación no es más que una sucesión de imágenes cada determinado tiempo que produce el efecto de movimiento o transformación en el objeto, por lo que necesitamos por un lado esa sucesión de imágenes dibujada y por otro lado hará falta el código que controle dicha sucesión en un determinado orden y con unos intervalos de tiempo entre imágenes correcto.

Para el dibujado, siendo el proyecto en 2.5D, (a efectos prácticos para la animación, 2D), se puede optar por diferentes opciones, desde dibujarlo a mano (poco recomendable por la falta de suavidad y precisión en las transiciones entre imágenes), dibujar y animar con vectoriales tipo Flash (lo que no es recomendable si hay rotaciones del objeto como pasa aquí por ser isométrico) o modelar la entidad en un programa de diseño 3D como Blender y posteriormente animarlo y renderizar los frames de las animaciones en una imagen en modo rejilla, lo que asegura que las animaciones van a ser uniformes en cada uno de los giros y que no será necesario redibujar las animaciones cada vez que se quiera

animar en un ángulo distinto. Como el diseño gráfico no es en lo que está centrado este proyecto, se ha decidido buscar las animaciones libres realizadas por otros autores con el método comentado de modelar y animar en 3D y renderizar en 2D en una gran imagen para cada modelo que contiene en las columnas las diferentes animaciones de ese modelo o entidad acotadas por diferentes intervalos, y en las filas se repiten las mismas animaciones pero en cada fila se representa un ángulo distinto, lo que se usará en este proyecto para controlar que en el mundo isométrico renderizado existen 8 direcciones (4 puntos cardinales + 4 diagonales)

Imagen con animación en diferentes ángulos de giro.

8.1. Implementación

La idea es abstraer el manejo de la animación lo máximo posible para facilitar no solo la reutilización del código en futuros proyectos, si no para facilitar el control de las animaciones por el proyecto actual y dejar que el módulo haga su trabajo para que el resto del programa pueda centrarse en sus funciones. Para ello se ha creado una clase Animación que se encarga de controlar una única animación pero con todos sus posibles giros que, en este caso serán 8, porque se tienen en cuenta los 4 puntos cardinales y las 4 diagonales. Para poder construir este objeto se le debe proporcionar la imagen que contiene todas las animaciones, el número de filas y columnas (que usará junto al tamaño de la imagen para calcular el tamaño de cada frame), y un número de frame inicial y final que representa el intervalo de columnas en las que se encuentra la animación dentro de la imagen. En cada

llamada a Update, el objeto animación incrementará el frame actual hasta que llegue al final del intervalo de la animación y vuelva al frame 0 para continuar indefinidamente, y calculará el rectángulo donde se encuentra en la imagen el frame actual para luego dibujarlo por pantalla con el método DrawFrame.

```
public enum Direccion8 { O, NO, N, NE, E, SE, S, SO }
public enum Acciones {Idle=0, Andar=1, Atacar=2, Morir=3}
```

Definición de las 8 direcciones y acciones posibles.

En este punto ya se pueden controlar animaciones simples, por lo que objetos y NPC ya podrían ser representados solamente con esa clase, aunque se ha decidido añadir una nueva capa de abstracción con la clase abstracta ObjetoAnimable, al que se le asigna una única animación con setAnimación(inicio, fin, fila) y controla la reproducción, la posición del objeto en pantalla, el tamaño y el delay entre frames de la animación.

A partir de esta clase, que como es abstracta no permita la instanciación de objetos, será necesario heredar los casos específicos de objetos o NPC animados (los NPC cuentan porque solo disponen de una sola animación, la de estar en idle, y un solo ángulo de giro), como por ejemplo el caso del NPC herrero, en cuyo constructor se inicializa la animación concreta asociada a esa entidad y que permite separar futuros comportamientos diferentes entre entidades, además de no obligar a memorizar la fila y columna en la que comienza cada animación a la hora de crear el NPC.

```
this.Inicializar(tex, 9, 25);
this.setAnimacion(0, 15, 5);
this.Play();
```

Constructor del caso específico del NPC Herrero

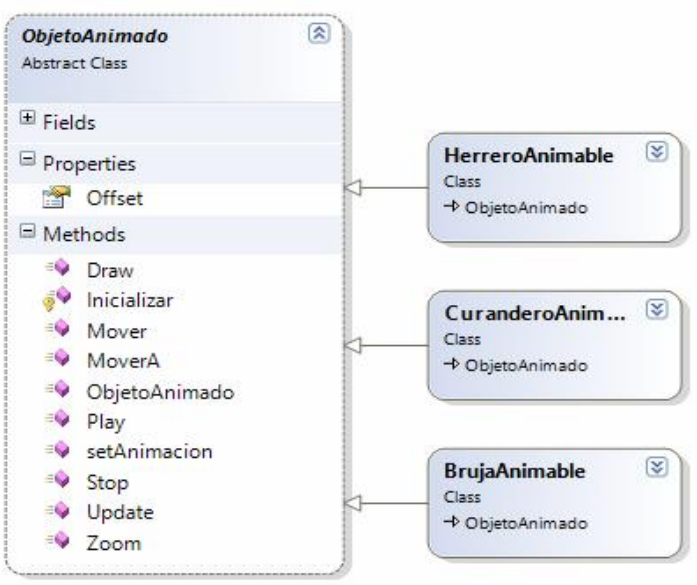


Diagrama de clases de Animaciones, objetos y NPC

Para el caso de animaciones más complejas de controlar como la de los enemigos o el propio héroe, será necesario añadir otra capa de abstracción de manera similar a la que se ha realizado con el caso de los objetos animables, en este caso con la clase abstracta PersonajeAnimable, que controla lo mismo que en el caso de ObjetoAnimable pero además controla un conjunto de varias animaciones posibles, asociadas a posibles acciones (parado o idle, andando, atacando y muriendo) y para cada una de ellas, 8 posibles direcciones para lo que en este caso se maneja un vector de objetos Animación del que se selecciona la acción a dibujar en el método Draw. Se ha añadido por comodidad código para que al mover el objeto animado, se modifique la dirección automáticamente teniendo en cuenta la posición actual y a la que se quiere desplazar dicho objeto. En este caso también se deben crear clases específicas para cada caso de personaje animable, como es el caso del HeroeAnimable que encapsula las animaciones del personaje guerrero principal del juego y en cuyo constructor se añaden todas las posibles animaciones:

```
this.Inicializar(tex, 8, 32);
this.AddAnimacion(Acciones.Idle, 0, 3);
this.AddAnimacion(Acciones.Atacar, 12, 15);
this.AddAnimacion(Acciones.Andar, 4, 11);
this.AddAnimacion(Acciones.Morir, 16, 22);
```

Posteriormente se pueden añadir capas de abstracción superiores, como se ha realizado en el caso del Héroe, para unificar la animación, los atributos y datos que describen al jugador dentro de la aventura RPG, control de teclado/ratón, y su comportamiento en un mismo lugar del código.

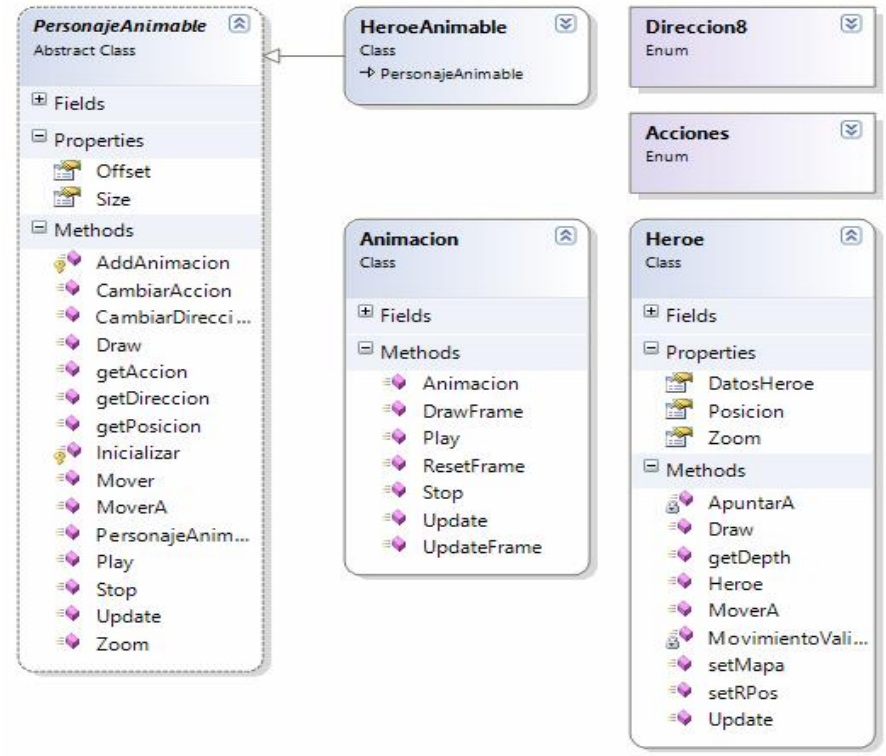


Diagrama de clases de Animaciones, personajes.

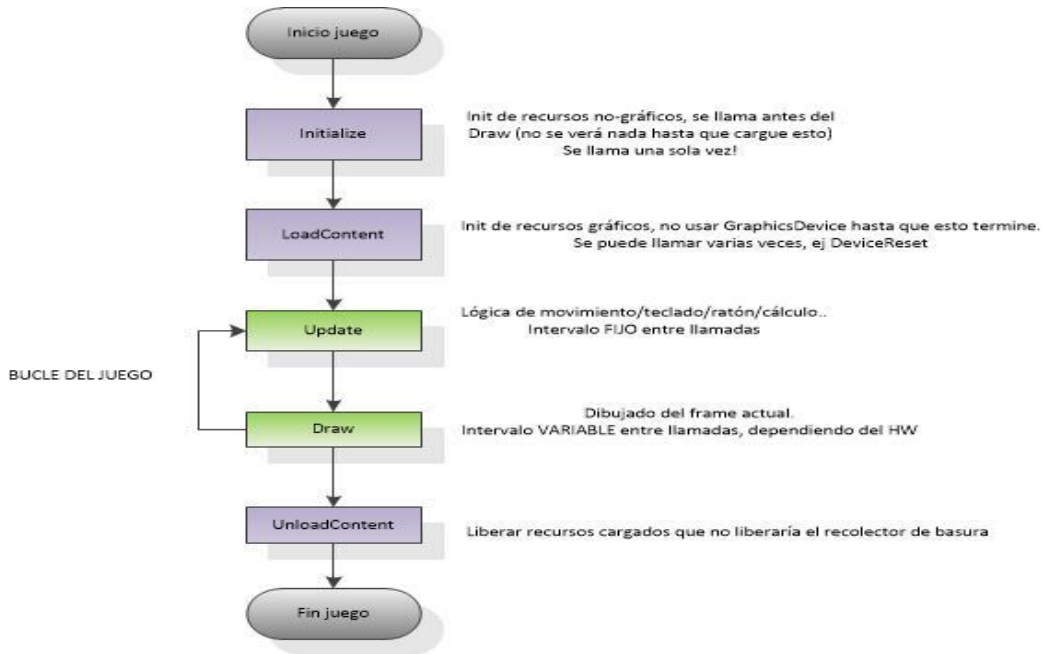
Varios enemigos animados usados en el juego.

Capítulo 9

Módulo juego

En el presente capítulo se va a hablar de todo lo que involucra al juego propiamente dicho, sin entrar en detalles de los módulos previamente expuestos, principalmente se trata de la unión de estos módulos, y del ciclo de trabajo del juego desde el arranque del ejecutable hasta su cierre.

Antes de empezar, es necesario conocer el ciclo de vida que XNA usa, para poder entender que hace cada fase y cuando lo hace, lo que se puede ver en la siguiente imagen que muestra los métodos de la clase Game inicial, cuándo se disparan y para qué se usan.



Ciclo de vida de juegos XNA

9.1. Problemas comunes

Uno de los problemas que hay que afrontar al crear un videojuego y que aparece además en la sección de requisitos no funcionales es que ese juego puede ejecutarse en una gran diversidad de máquinas con distinto hardware/software, por lo que hay que, de alguna manera, asegurar que aunque en una máquina pueda ir más o menos rápido que en otra, haya una consistencia en el paso del tiempo renderizado en ambas, es decir, que aunque en el ordenador A se ejecute el juego a 60 FPS, al cabo de 10 segundos el juego debe estar en el mismo segundo que en otro ordenador que vaya a 30 FPS o 10 FPS, aunque lógicamente el juego se verá mucho más fluido en el de 60 FPS que en el resto. Viendo el diagrama del ciclo de vida, se puede ver que simplemente asegurándonos de que todos los cálculos de lógica del programa y movimientos se realice en el método Update no habrá problema, por que este se llama con un intervalo fijo que no varía entre diferentes máquinas, dejando el método Draw única y exclusivamente para labores de pintado, ya que este será llamado tantas veces como el hardware lo permita para renderizar el juego.

Otro problema que aparece también al tener en cuenta diferentes configuraciones de hardware, es la resolución de pantalla en la que se ejecutará el juego, ya que esta puede variar tanto en tamaño cono en relación de aspecto (16:10, 16:9, 4:3, etc) y el juego debe ser capaz de ser dibujado correctamente en todas ellas, con el menor impacto posible en el código fuente. Hay varias soluciones a tal problema, algunas como renderizar más o menos objetos del juego o redimensionar y poner el origen de coordenadas (0,0) en el centro de la imagen en vez de en la esquina superior izquierda añaden complejidad al código y puede derivar en que los jugadores con determinadas resoluciones puedan tener una ventaja al ver más área de juego que otros jugadores con monitores menos aventajados. Finalmente, la solución que se ha usado en este proyecto es sencilla y eficaz, todos los objetos del juego que quieran dibujarse, en vez de hacerlo directamente en la pantalla lo harán en un buffer de tamaño fijo 1680x1050, y cuando todas las entidades estén dibujadas en ese buffer, antes de pasar al siguiente frame, se vuelca el buffer sobre la pantalla estirando o encogiendo el buffer para adaptarlo al tamaño de la resolución del usuario final. De esta manera, como el buffer es de tamaño fijo, solo hay que programar el juego teniendo en cuenta una resolución y el juego se ajustará automáticamente a la resolución requerida por el usuario sin mayor complicación.

Para conseguir esto de manera transparente, se ha creado la clase abstracta llamada XNAScreen que hereda de DrawableGameComponent de XNA por lo que es capaz de recibir los mensajes Initialize, LoadContent, Update y Draw del Game principal que mueve todo el ciclo de vida del juego.
En Initialize creará el buffer que se usará para dibujar el contenido del juego, en LoadContent se crea un SpriteBatch que permitirá pasar ese buffer a pantalla, en Update se actualizará la posición del ratón y en Draw primero se permitirá que el Draw de la clase que herede pueda dibujar su parte en el buffer y posteriormente hara el intercambio de buffer a pantalla.

Como solo debe existir a la vez una pantalla XNAScreen dibujando de manera simultánea, se ha añadido el método ChangeScreen al que se le pasa un objeto de una clase que herede de XNAScreen y se intercambia el control de la nueva por la actual.

Para comodidad a la hora de modificar la resolución de pantalla desde cualquier parte del juego, se ha añadido la clase estática ManagerGrafica con el método CambiarResolución.

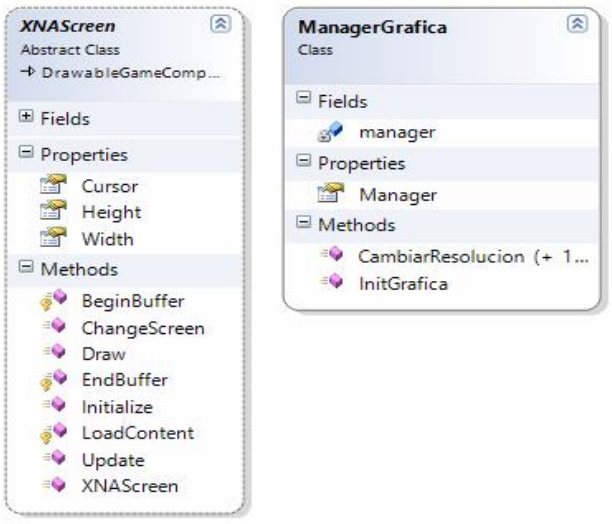


Diagrama de clases de control de pantalla

9.2. Atributos

En el capítulo 2 se habló de la necesidad de incluir determinados atributos que describen y determinan el comportamiento del héroe durante la aventura y en este apartado se van a explicar los diferentes atributos que se han seleccionado para tal efecto y las relaciones matemáticas que los unen. El juego se compone de unos atributos primarios que modifican otros atributos secundarios basados en nivel y stats, siguiendo esta relación:

· Fuerza: Determina el daño realizado, junto con el arma y el nivel del personaje

```
Daño = (fuerza * nivel / 100) + daño_arma
```

· Destreza: Determina el porcentaje de mitigación de daño del personaje, junto con la armadura. `%Mitigación = (Destreza * Nivel / 100) + (Armadura / 50)`

· Inteligencia: Determina la cantidad máxima de maná del personaje como:

```
Mana = 1.5 * Inteligencia + Nivel
```

· Vitalidad: Determina la cantidad máxima de vida del personaje como:

```
Vida = Vitalidad * 2 + 2 * Nivel + 18
```

· Nivel: Indicador del progreso del héroe, se incrementa cuando se supera un límite de experiencia obtenida y se usa para cálculos como la experiencia obtenida, el drop de items o el daño realizado. Cada nivel requiere más experiencia que el anterior, con la fórmula:

```
SiguienteNivel = 772 + (1 + (nivel * 8 / 100)) * 772;
```

· Experiencia: Matar un enemigo la genera, y varía según la diferencia de nivel entre el enemigo y el héroe, hasta un máximo de 10 niveles, donde ya no genera experiencia.

```
Experiencia = 123 * (1.0 + 0.1 * (nivelenemigo - nivelheroe))
```

· Nombre: Texto que distingue al héroe de otros, debe ser único.

· Tipo: Siempre guerrero en esta demo del juego.

9.3. Persistencia

La persistencia de datos es un factor de vital importancia en videojuegos, especialmente de este tipo, ya que de no existir, poder llegar al final de uno de estos juegos sería una odisea que requeriría de mucha cafeína y tiempo libre.

En este videojuego, existen varios aspectos que requieren de persistencia tales como las opciones de juego, el nombre de la última partida guardada, los atributos de los diferentes héroes, así como sus mapas, la posición sobre ellos y los tiles que el héroe ha visto, información de posición y estado de los enemigos, etc. En esta sección se van a comentar las clases auxiliares, los formatos y estructura de ficheros que se han decidido usar para cada una de las tareas que han requerido persistencia en el videojuego.

En el menú principal y desde el menú ingame se pueden acceder a las opciones de juego, que como se vio en el apartado de interfaz, permite modificar determinados aspectos del juego como volumen y mute de música y audio, resolución gráfica y pantalla completa, controles y configuración varia referente al HUD ingame. De no existir persistencia en este aspecto, habría que configurar el juego cada vez que se ejecutara, por lo que se ha decidido proporcionarle persistencia.

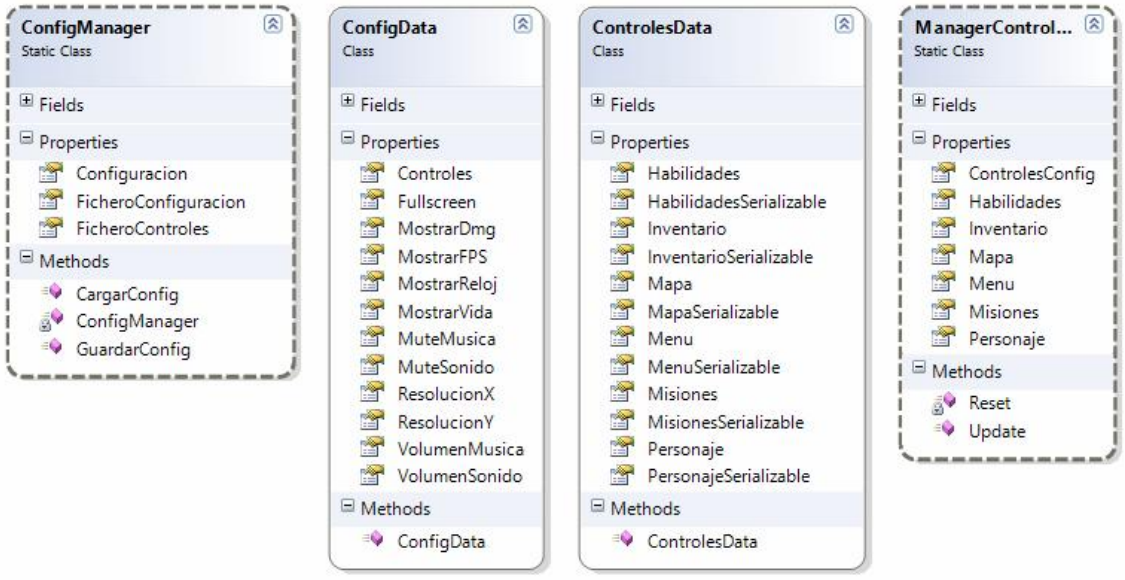


Diagrama de clases de configuración.

La persistencia en la configuración del juego comienza en la clase estática ConfigManager, donde al especificarle los nombres de los ficheros de configuración y de controles mediante propiedades, se podrán cargar y guardar los cambios realizados en la propiedad Configuración, que es un objeto de tipo ConfigData, contenedor de datos con una propiedad por cada configuración a guardar y que además contiene otro objeto en la propiedad Controles de tipo ControlesData que mantiene los datos referentes a la asignación de teclas para determinadas acciones ingame como abrir un el inventario o los atributos. Las operaciones de carga y guardado de los ficheros de configuración se puede realizar de manera sencilla serializando o deserializando en formato XML, gracias a que se han separado en clases distintas la parte que contiene los datos de la que los maneja como se puede observar en el siguiente código:

```
XmlSerializer writer = new XmlSerializer(typeof(ConfigData));

StreamWriter file = new StreamWriter(FicheroConfiguracion);
writer.Serialize(file, config);
file.Close();

writer = new XmlSerializer(typeof(ControlesData));
file = new StreamWriter(FicheroControles);
writer.Serialize(file, config.Controles);
file.Close();
```

Código de guardado de configuración mediante serialización XML.

Por comodidad se ha añadido la clase ManagerControles, que abstrae el control de pulsaciones sin repetición de la asignación de teclas configurada en ControlesConfig, lo que evita tener que añadir esos condicionales y añadir el control anti-repeticiones dentro de la lógica del juego.

Como ConfigManager y ManagerControles son clases estáticas, se podrán acceder desde cualquier punto del juego para o bien leer / modificar la configuración o comprobar de manera sencilla si alguna de las teclas configuradas ha sido pulsada.

Todos los datos de todas las partidas se guardarán en el directorio 'partidas/', donde además se ha incluido en la raíz el fichero 'ultima.txt' que contiene una única línea de texto indicando el nombre del héroe que se ha jugado por última vez, para que en caso de existir dicho fichero el botón de Continuar en el menú principal esté activado y permita continuar esa partida desde el punto donde se dejó, o bien deshabilitar el botón en caso contrario.

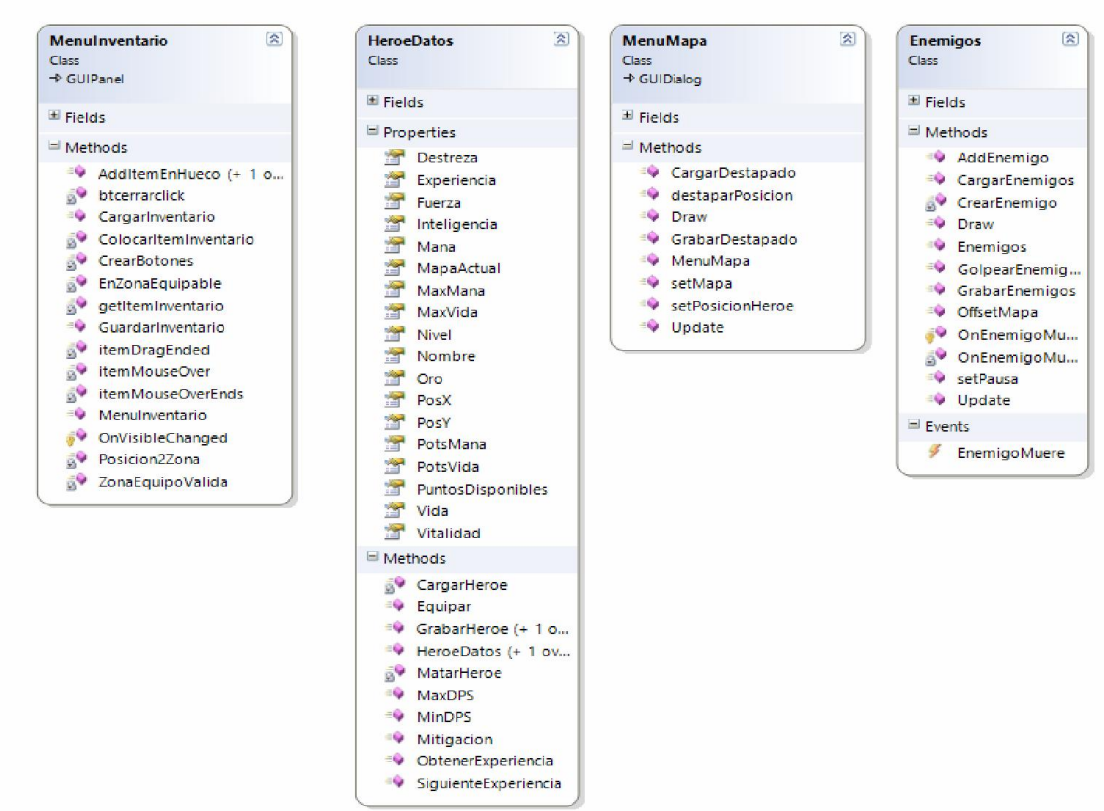
Este fichero es modificado al seleccionar un héroe y será eliminado al borrarse el héroe, ya sea manualmente o por muerte del mismo.

Otro de los ficheros añadidos a la raíz del directorio 'partidas/' es el fichero 'cementerio.txt', al que se añaden líneas con el formato 'nombre|nivel|mapa|fecha|hora|', una por cada héroe caído en combate, justo antes de borrar toda la carpeta de guardado del héroe caído, debido a que se aplica la característica inherente de los juegos rogue-like de muerte permanente.

Todos los datos referentes a una partida de un héroe concreto se guardarán en una carpeta creada con el nombre del personaje dentro del directorio 'partidas/' justo cuando se acepta la selección de nombre en el menú crear nuevo personaje.

En esta carpeta se encuentra el fichero 'stats.txt' que incluye varias líneas, cada una representando cada uno de los atributos del héroe como fuerza, nivel o nombre entre otros, así como el mapa actual en el que se encuentra y su posición en el mismo. Otro fichero relacionado con el héroe es 'inventario.txt' en el que se incluyen 28 líneas con cadenas de texto conteniendo cada una un objeto serializado, las 7 primeras corresponden a los objetos equipados mientras que las 21 restantes a los elementos en la matriz del inventario, ya que está formada por 7 columnas y 3 filas. El manejo del fichero de stats lo realiza la clase HeroeDatos, manteniendo además en memoria esos valores, mientras que para el inventario se usa la clase MenuInventario, ambas clases conteniendo funciones para la carga y guardado de los datos.

Para cada mapa generado, se crea un fichero con el nombre 'mapaX.txt', donde X es un entero que determina el orden del mapa. En su interior hay varias lineas con información sobre el mapa como el nombre, el nivel, las posiciones X,Y de los teleports de entrada y salida del mapa, el tipo de mapa (cueva, mina o mazmorra), el tamaño del ancho y alto, y el dato más importante, la semilla aleatoria con la que se generó, que al llamar a los algoritmos de generación de mapas aleatoria con la semilla y el tamaño podrá reconstruir el mapa dejándolo idéntico al original.



Alguna de las clases encargadas de persistencia de datos.

Para cada uno de esos mapas se guardan además las casillas que el héroe ha ido destapando con su visita, que se usará en el menú del mapa para tener una visión del camino realizado por el héroe y las zonas que ha ido descubriendo. Para ello se ha creado un fichero paralelo a cada uno de los mapas con el formato 'destapadoX.txt' que es simplemente un volcado de la matriz que contiene el mapa por filas y columnas, donde cada casilla es representada por el valor numérico de su tile. La clase encargada de cargar, mantener, visualizar y guardar los datos de los mapas destapados es MenuMapa.

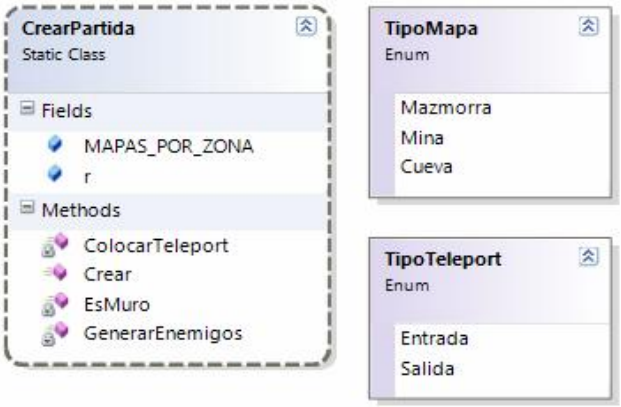
Finalmente, para cada mapa es necesario mantener una lista con todos los enemigos que contiene junto con su estado y atributos. Al crear el mapa se generan también una cantidad de ficheros de texto con el formato 'enemigosX.txt' donde X iguala al número de mapas generados, en el que se guarda una línea por cada enemigo, donde cada línea contiene valores separados por el símbolo '|' como el nivel, vida, daño, tipo de enemigo, coordenadas de origen, posición actual y estado actual de su máquina de estados.

9.4. Objetivos

El objetivo de esta demo es conseguir que el héroe consiga atravesar uno por uno todos los mapas que han sido generados aleatoriamente (en la demo se han creado 9, siendo 3 de cada tipo, aunque la cantidad se puede variar con solo modificar una variable) hasta llegar al final del último de los mapas. Para ello deberá eliminar a todos los enemigos que se interpongan en su camino, incrementando su poder a medida que gana experiencia con las muertes de los enemigos y objetos que podrán ser equipados para modificar sus atributos, lo que será necesario ya que los enemigos aumentan en vida y daño conforme se va progresando en los niveles.

Tanto la creación de los mapas, como de las estadísticas iniciales del héroe y los enemigos se realizan justo en el momento en el que el héroe es creado, y al inicio de cada partida se cargan los datos previamente generados desde el disco duro con las clases de persistencia comentadas en el apartado anterior, datos que serán modificado durante la experiencia de juego del usuario para posteriormente ser grabados otra vez en disco.

Todo este proceso procedural se ha encapsulado en la clase estática CrearPartida, cuyo método Crear se encarga de todo el proceso mencionado de manera transparente con solo pasarle por parámetro el nombre del jugador a crear y ayudándose de los métodos auxiliares ColocarTeleport, EsMuro y GenerarEnemigos.



Clase para la creación de la partida

9.5. Sonido

El juego cuenta con efectos sonoros y música, que pueden ser reproducidos desde cualquier parte del juego, ya sea una vez o en bucle. Para facilitar la tarea se ha creado una clase auxiliar y estática AudioHelper que se encarga de gestionar todo lo relacionado con el sonido en el juego, desde cambiar el volumen o silenciar tanto la música como el sonido por separado con valores de volumen del 0 al 100, hasta reproducir un sonido aleatorio de una colección de sonidos haciendo uso solamente de una instrucción con PlaySonidoRandom, lo cual es útil para efectos de sonido como los de un héroe atacando o andando, ya que en estos casos y para evitar que el sonido sea repetitivo se crean listas con varios sonidos diferentes del mismo tipo y a la hora de reproducir se selecciona uno de manera aleatoria.

Clase auxiliar para el control del sonido.

9.5. Cámara

El movimiento de la cámara en este tipo de juegos es algo peculiar, el héroe tiene una posición X,Y dentro del mapa en el que se encuentra y esos valores se van modificando a la vez que el héroe se mueve, pero a la hora de visualizarlo en realidad el héroe siempre aparece inmóvil en la posición central de la pantalla. Para conseguir este efecto se deben guardar las posiciones reales de todos los elementos del juego que se muestran sobre el mapa, esto es los enemigos, el héroe y el propio mapa y por otro lado se debe guardar para

cada uno de ellos el valor X,Y del offset o desplazamiento que se sumará o restará a la hora de dibujarlo en pantalla. De esta manera se mantienen tanto las coordenadas reales para poder realizar cálculos de distancia o colisión por ejemplo y las coordenadas en las que se deben dibujar en pantalla. Todo va guiado por el héroe, por lo que si el héroe se mueve en una dirección se deben desplazar todos los enemigos y el mapa en sentido contrario y con la misma intensidad.

El control del héroe se realiza mediante el ratón, atacando en la posición del ratón con el click izquierdo y movimiento en la dirección a la que apunta el ratón con el click derecho. Para el primer caso basta con aplicar la siguiente fórmula y obtener el ángulo que forma la línea que separa los puntos de la posición del héroe y el ratón y discretizarla a 8 direcciones posibles a fin de determinar la animación correcta simplemente dividiendo el resultado entre 45, ya que 360/45=8.

$$angulo = atan2(y2 - y1, x2 - x1) * \frac{180}{\Pi}$$

Fórmula para obtener el ángulo entre dos puntos.

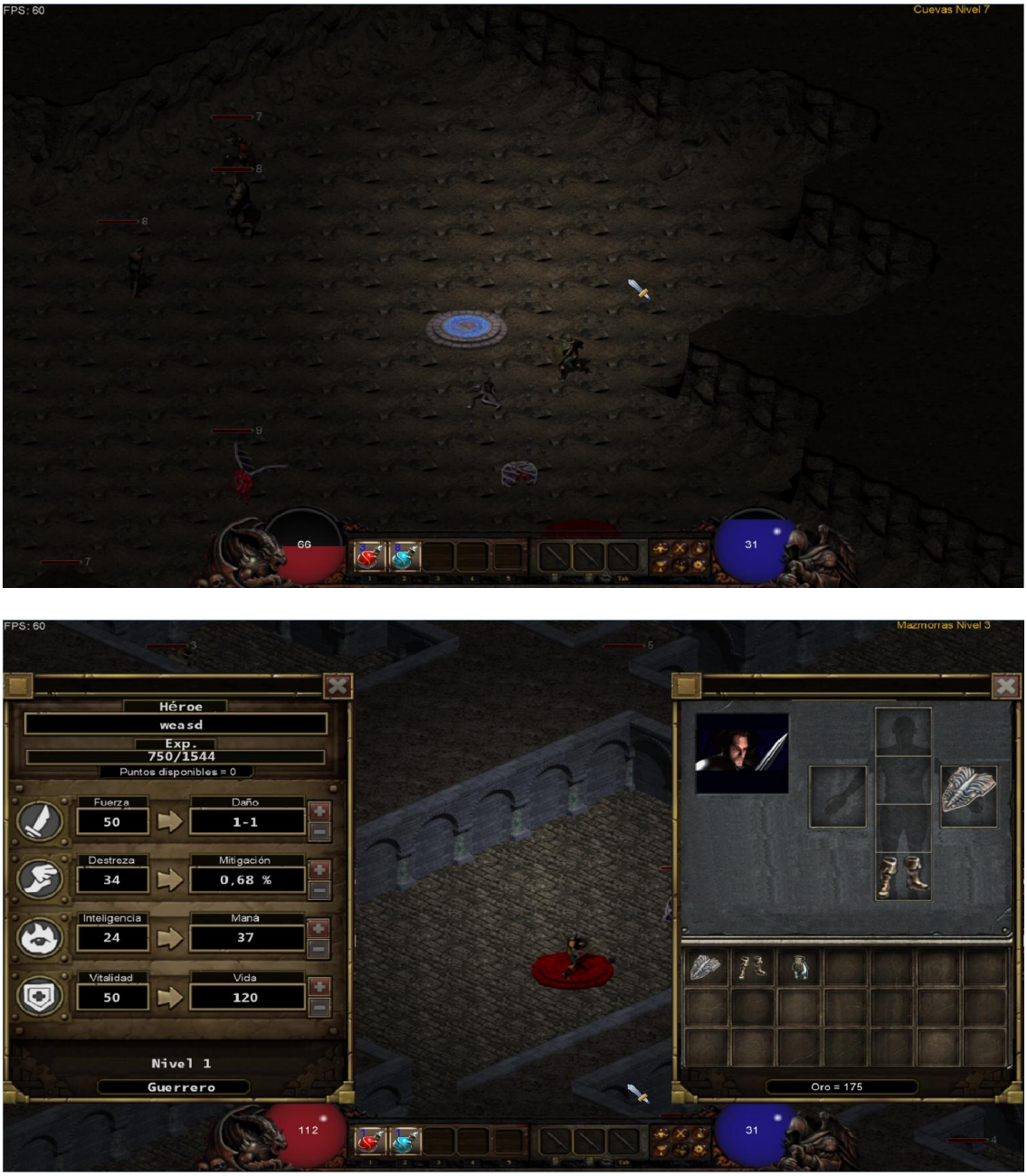
Para el caso del movimiento hay que usar esa misma función para conocer la dirección de movimiento, pero esta vez antes de discretizar su valor, se debe calcular el offset X,Y del avance en esa dirección teniendo en cuenta que en isométrico el valor de la X es el doble que el de la Y, para lo que se aplican las siguientes funciones:

$$offsetx = 2 * cos(angulo) * velocidad * zoom$$
$$offsety = sin(angulo) * velocidad * zoom$$

Fórmula para obtener el desplazamiento en isométrico hacia un punto.

9.6. Resultado final

Tras unificar todos los módulos presentados en este documento, y programar la lógica del videojuego, se puede ver el resultado final en las siguientes capturas de pantalla, en las que se muestra la ventana de juego con y sin los menús del HUD abiertos. El resultado podrá apreciarse mejor en la demo a realizar en la presentación del trabajo ante el tribunal.



Capturas del resultado final del proyecto.

Capítulo 10

Métricas

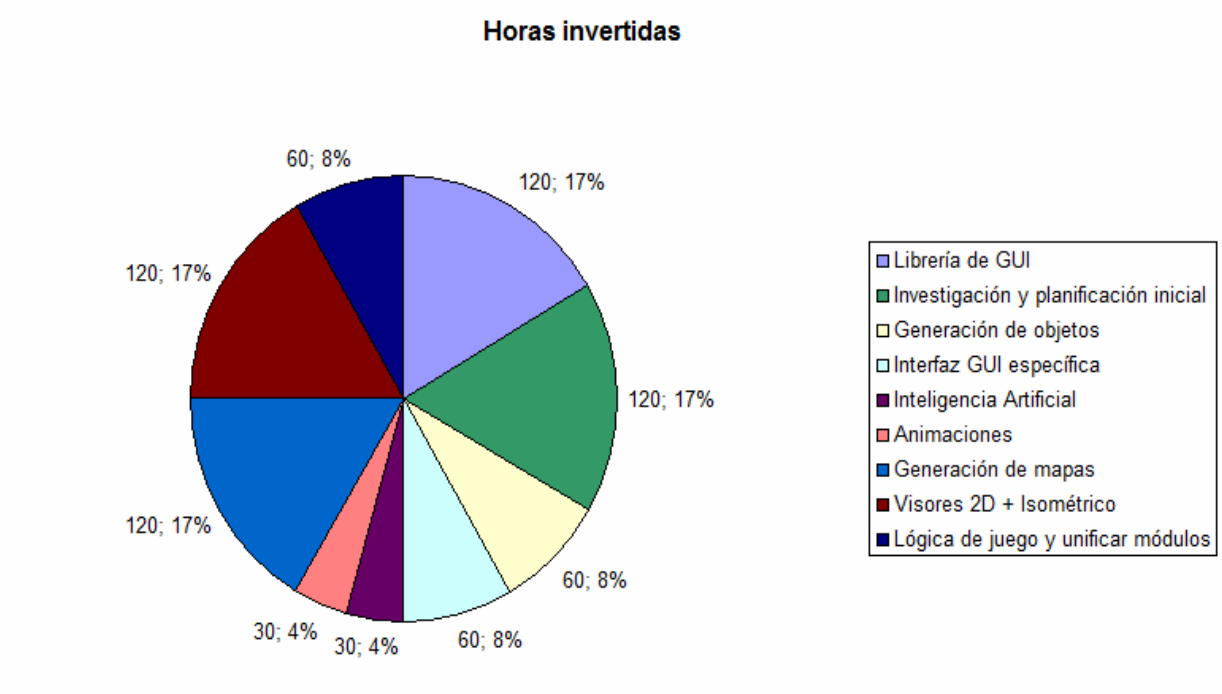
10.1. Coste temporal

Dado que en este proyecto no ha habido ningún tipo de coste económico por la ausencia de salarios y el uso de herramientas gratuitas o de pago con licencia de estudiante, esta sección se va a centrar en el coste temporal aproximado que ha sido necesario para completar el trabajo de final de grado. La idea de realizar un videojuego como proyecto final ya rondaba por la cabeza mucho antes de los periodos de solicitud de TFG, por lo que en Julio de 2013 inicié el proyecto del módulo GUI genérico, puesto que eso iba a servir para cualquier juego que decidiera realizar en un futuro. Para Septiembre realicé la solicitud y tras ser aceptada comencé a recopilar información sobre el estado del arte en el género de videojuego y sobre los algoritmos que iba a poder necesitar, así como iniciando el diseño en papel de algunos módulos y creando parte de la interfaz específica del juego, fase que duró aproximadamente hasta Enero de 2014, donde se iniciaron las iteraciones por módulos, uno por uno hasta completarlos todos, generando la documentación sobre la marcha hasta el día de hoy en Junio de 2014.

Se muestran a continuación la tabla y la gráfica con los datos aproximados de horas consumidas en cada una de los módulos del juego así como en otras fases necesarias como la obtención de información y estado del arte mencionadas.

modulo	horas
Librería de GUI	120
Investigación y planificación inicial	120
Generación de objetos	60
Interfaz GUI específica	60
Inteligencia Artificial	30
Animaciones	30
Generación de mapas	120
Visores 2D + Isométrico	120
Lógica de juego y unificar módulos	60
TOTAL	720

Horas consumidas en cada sección del proyecto.



Horas invertidas y porcentaje de horas por cada módulo.

Se incluye además un diagrama de Gantt que muestra la estructura de la planificación y en el que se aprecian las iteraciones basadas en features o módulos que caracterizan a la metodología FDD usada en el proyecto realizadas de manera secuencial. Cabe destacar que aunque la primera fase de FDD debería ser la de planificación inicial, la librería GUI genérica fue un proyecto aparte y al ser genérico y usable por otros proyectos podía ser iniciado antes de la planificación, la cual tuvo que esperar a tener asignado un TFG.

Nombre de tarea	Duración
Librería GUI genérica	40 días
Investigación y planificación inicial	40 días
Interfaz GUI específica	20 días
Generación de mapas	40 días
Visores 2D + Isométrico	45 días
Generación de objetos	20 días
Animaciones	10 días
Inteligencia artificial	10 días
Lógica de juego y unificar módulos	20 días

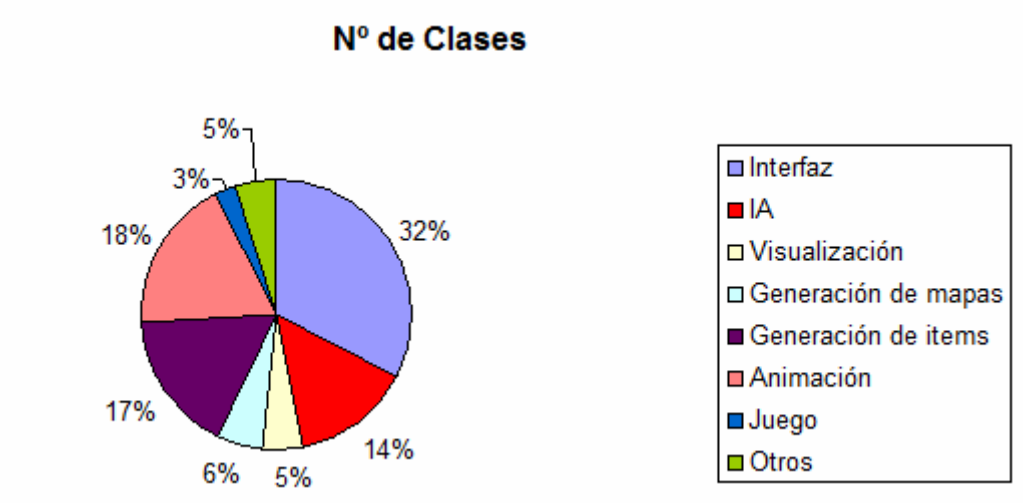
Diagrama de Gantt con la planificación.

10.2. Otras métricas

Otro tipo de métricas que pueden mostrar cierta información sobre la complejidad de cada módulo o sobre el esfuerzo invertido en ellos pueden ser el número de líneas o el número de clases que se han debido crear para completar sus funcionalidades. Se muestra el gráfico con la distribución de las 107 clases que han sido requeridas para completar el proyecto en los diferentes módulos.

Módulo	Clases
Interfaz	35
IA	15
Visualización	5
Generación de mapas	6
Generación de items	18
Animación	20
Juego	3
Otros	5
TOTAL	107

Nº de clases creadas por módulo.



Distribución de clases por módulo.

Se ha efectuado un análisis del código para extraer otras métricas como el número de líneas de código, complejidad ciclomática, índice de mantenibilidad o la profundidad de herencia. Cabe destacar que el total de líneas de código aun sin contar líneas en blanco ni comentarios asciende a un total de 5422 líneas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Lines of Code
AlgoritmosGeneracion (Debug)	77	162	2	288
AlgoritmosIA (Debug)	83	134	1	267
Generadoritems (Debug)	82	141	7	477
GUI (Debug)	83	501	4	968
TFG (Debug)	75	933	5	3,442

Otras métricas extraídas del código fuente con Visual Studio 2010 Ultimate.

Capítulo 11

Conclusiones y futuros

11.1. Conclusiones

Con la información obtenida en la fase de investigación y siguiendo la planificación establecida, se ha llegado a obtener un resultado final jugable, como estaba estipulado en los objetivos. En cuanto a la aleatoriedad, se han conseguido terminar varias partes que hacen uso de ella, como son la generación de mapas, de enemigos o de objetos.

Se ha podido terminar la librería genérica de controles de interfaz en su totalidad, incluso en su estado final contiene más controles de los que finalmente han sido necesarios para este proyecto en particular, aunque se deja abierta la posibilidad de añadir cualquier control más que fuera necesario en un futuro. De igual manera, se han implementado todos los menús del juego con esta interfaz, desde el menú principal al de opciones, pasando por el de creación de personajes, cementerio o confirmación de salida.

En cuanto a la generación aleatoria de mapas, se ha realizado un extenso esfuerzo en este apartado, y finalmente se ha conseguido adaptar dos algoritmos para la visualización en isométrico, con los que se han creado tres tipos diferentes de mapas aleatorios (mazmorra, mina y cueva). El algoritmo de generación celular no se ha utilizado finalmente porque es usado principalmente para generar mapas de exteriores, de los cuales el juego actual no incluye ninguno, aunque la clase está implementada en el proyecto.

La lógica del juego ha sido terminada, permitiendo al jugador iniciar en un punto del mapa principal e ir avanzando mapa por mapa hacia el final, eliminando enemigos a su paso y mejorando sus atributos y equipamiento para conseguirlo. Se ha dejado fuera finalmente a las habilidades del personaje por que no aportaba nada nuevo al proyecto y se ha preferido usar ese tiempo para otros módulos más importantes.

La generación aleatoria de enemigos es funcional, al igual que la inteligencia artificial asociada a estos enemigos, aunque se podrían haber añadido enemigos de tipo rango y no solo de cuerpo a cuerpo, pero se ha considerado suficiente para la demo disponer de un solo tipo de enemigos, aunque con diferentes skins, proporcionando autonomía a los enemigos controlados por la IA.

La progresión en el avance de la aventura se ha conseguido en gran medida a las fórmulas implementadas en los atributos que vinculan el nivel del personaje a estos atributos, junto con los modificadores de atributos de los objetos generados aleatoriamente por la librería diseñada a tal efecto.

Se ha implementado el control del personaje mediante el uso del ratón, tanto el movimiento como el ataque, mientras que con el teclado se controla el resto de la interfaz como la barra de estado o el inventario por ejemplo.

Otro de los apartados que se ha completado en su totalidad, es la persistencia de datos, en todos los componentes de la partida que lo requieren tales como la configuración, mapas, enemigos, héroe, etc. lo que proporciona la capacidad de volver a jugar en el estado en que se dejó la partida anteriormente.

Finalmente, la visualización usada ha sido la isométrica, aunque se ha implementado la 2D y se han realizado pruebas sobre la 3D, por lo que se ha cumplido el objetivo de mejorar la visualización del rogue original, dejándola a medio camino entre las 2D y las 3D.

11.2. Reflexiones

Estoy satisfecho con el resultado ya que se ha conseguido terminar con un producto jugable de una complejidad elevada y con muchos módulos diferentes que alguno de ellos se podría considerar un proyecto aparte como el de la interfaz o el de la generación de mapas de manera aleatoria. Otro de los hitos conseguidos ha sido el estructurar el proyecto por módulos, de tal manera que gran cantidad del esfuerzo realizado pueda ser reutilizado en un futuro sin gran esfuerzo por otros proyectos, incluso aunque no sean del mismo género.

Por otro lado, el poder completar un proyecto de tal envergadura, con un ritmo frenético solventando problema tras problema, me ha motivado a seguir desarrollando videojuegos en un futuro gracias a la experiencia obtenida y sobre todo a que gracias a este TFG ya no veo la programación de videojuegos como una utopía inalcanzable.

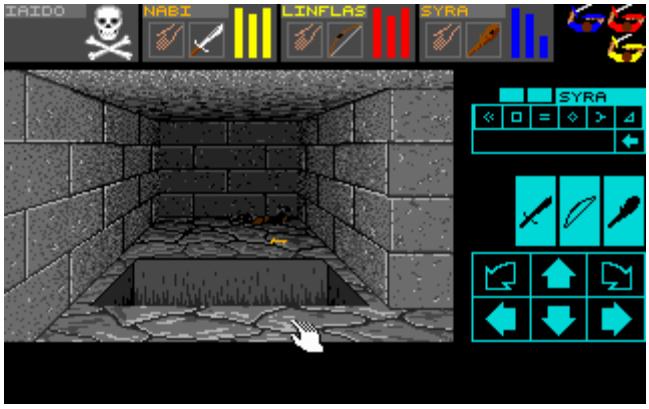
El mismo hecho de que el realizar un videojuego de cierta complejidad es una tarea ardua y que requiere de gran cantidad de tiempo, explica por qué las grandes compañías de videojuegos disponen de grandes grupos de trabajo y no dejan la tarea a una sola persona.

Se ha podido comprobar que a pesar de que la programación del videojuego ha llevado gran cantidad de tiempo, se invierte mucho más tiempo en la parte de análisis y diseño lo que evita en gran medida que haya que refactorizar grandes porciones de código o incluso que se deban reescribir o descartar. La búsqueda de información previa como algoritmos ya existentes para evitar la reinvención de la rueda también suma coste temporal pero al final reduce el tiempo total de desarrollo del proyecto.

En resumen y para concluir, este trabajo de final de grado ha sido toda una experiencia positiva de la que espero sacar partido en un futuro.

11.3. Futuros

Dentro de objetivos futuros hay diferentes opciones viables a las que atenerse, primeramente y haciendo uso de la modularidad de ciertas partes del proyecto como la interfaz GUI, los algoritmos de generación aleatoria tanto de mapas como de objetos o la visualización, se podrían reutilizar dichos módulos para la realización de otros proyectos de videojuegos y dependiendo de la similitud al proyecto roguelike presentado en el presente documento se podrían reutilizar más o menos módulos. Por ejemplo la generación de mazmorras y la interfaz se podría reutilizar para realizar otros juegos de géneros similares como Dungeon Master aunque habría que diseñar otro módulo de visualización que permitiera la nueva perspectiva.



Juego Dungeon Master (1989)

Otro de las posibles opciones es terminar el juego actual, ya que al estar enfocado a un proyecto de final de carrera y no a un proyecto comercial real, se ha intentado profundizar más en los diferentes aspectos técnicos que en la finalización completa del mismo, es decir, era mucho más importante que apareciesen todas las secciones que forman el juego detalladas antes que enfocar el esfuerzo en terminar de manera completa algunos módulos y dejar a otros sin nombrar.

Para poder terminar este proyecto habría que incluir elementos multimedia propios como sonidos, música o gráficos, sería necesario completar el resto de clases ya que para la demostración solo se ha incluido el guerrero y se han excluido el arquero y el mago, habría que añadir habilidades en las clases, incluir una IA decente en los enemigos, incluir una ciudad inicial con mercaderes y misiones secundarias, optimizar y depurar, etc.

Esta opción parece viable gracias a plataformas como Steam o Desura que soportan e incentivan el desarrollo de videojuegos indie.

Finalmente, la última opción que se baraja es la más costosa, que sería adaptar los conocimientos adquiridos durante la realización del presente proyecto a otras plataformas móviles como iOS o Android, y el coste puede venir por dos flancos, el primero si se desea reutilizar el código sin apenas modificaciones vendría de la necesidad de contratar servicios como el de Xamarin que a partir del código C# del proyecto genera ejecutables para las diferentes plataformas móviles pero a un precio de 300$ por año y plataforma. Si se desea evitar este pago, el coste vendría por la necesidad de reescribir el código del TFG o de sus módulos a los lenguajes usados por las distintas plataformas, ya sea Java, ObjectiveC o C++, y la necesidad de mantener un proyecto por separado para cada plataforma. Con C++ existe una librería llamada Cocos2d-x que contiene clases enfocadas a la creación de videojuegos en 2D multiplataforma sin la necesidad de disponer de un proyecto distinto por cada plataforma lo que evita el problema mencionado.

Sea cual fuere la decisión final en cuanto al futuro del presente proyecto, hay que recalcar que no sería posible de no haber realizado el presente trabajo de final de grado, ya no por los módulos reutilizables programados, si no por la experiencia obtenida durante todo el proceso de realización del videojuego, donde se han aprendido infinidad de algoritmos y sobre todo se han superado los constantes, diferentes y en algunos casos difíciles problemas que aparecen en la creación de un proyecto de esta envergadura.

Bibliografía

[1] Brian Schwab. *AI Game Engine Programming*. Thomson Learning, 2004.

[2] Steven John Metsker. *Design Patterns In C#.* Addison-Wesley, 2004.

[3] Stuart Russell & Peter Norvig. *Artificial Intelligence*. Prentice Hall, 1995.

[4] Roger S.Pressman. *Software Engineering 7ed.* McGraw Hill, 2010.

[5] Danny Kodicek. *Mathematics and physics for programmers*. Cengage, 2011.

[6] Tom Miller. *XNA Game Studio 4.0 Programming.* Addison-Wesley, 2010.