Universidad de Huelva

ESCUELTA TÉCNICA SUPERIOR DE INGENIERÍA

DE LA UNIVERSIDAD DE HUELVA

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Generación de Mazmorras. Análisis e investigación del problema. Acercamiento apoyado en búsqueda.

Autor:

Alejandro Seguí Díaz

Tutores:

Gonzalo A. Aranda Corral

Daniel Márquez Quintanilla

Huelva, 30 de junio de 2015.

Curso académico 2014/15.

Índice general

Índice de figuras

Introducción

En la actualidad, los videojuegos han conseguido posicionarse en un mercado líder indiscutible a nivel internacional. En el ámbito del entretenimiento, la industria cinematográfica está dando paso a los videojuegos, que avanzan a pasos de gigante. Muchas grandes personalidades del cine se dirigen a los creadores de videojuegos para continuar su carrera. Un ejemplo claro de una persona adelantada para su época en este sentido, es George Lucas ([1]), autor de la famosa saga Star Wars. George Lucas se introdujo en el mercado de los videojuegos en el 1986 con el título Labyrinth, dando paso posteriormente a otros títulos muy sonados y de culto como Monkey Island o Grimm Fandango.

Un videojuego, al igual que una película, puede contar una historia. Aún así, existen diferentes géneros de videojuegos, que van desde simulación de juegos de mesa, donde evidentemente no existe historia, o está intrínseca en el origen del mismo juego, hasta las denominadas aventuras gráficas, donde el enfoque está en la parte argumental. Es por ello que la industria del cine deja paso a los videojuegos de manera tan rápida.

Mapas y escenarios.

Muchos géneros de videojuegos plantean su argumento en un escenario donde se da rienda suelta a la capacidad perceptiva del jugador, dando lugar en mayor o menor medida a que complete la historia con su imaginación. En este escenario, el jugador desarrollará las acciones que se le ofrezcan según el tipo de juego. Así, podrá completar la historia, permitiendo además en muchos casos que las acciones influyan en el desarrollo argumental del videojuego. Tipos de juegos que requieren mapas son juegos de acción, estrategia o rol entre otros.

El tipo de escenario que podemos encontrarnos en un juego es de lo más variopinto. A la hora de plantear la elaboración de un juego, se tiene en cuenta la elección del tipo de escenario. Muchas cuestiones sobre las que se basan estas elecciones radican sobre la complejidad de optimización del mismo:

- Si el escenario es demasiado grande, no nos interesa renderizar el escenario completo, sino solamente la parte visible.
- Lo mismo pasa con las físicas. No nos interesa comprobar colisiones con elementos que es obvio que no van a colisionar.
- El *nivel de detalle* (Level of detail por el inglés [2]) es otro aspecto a tener en cuenta. Si una geometría está demasiado lejana del jugador, no necesitamos renderizarla exactamente como es. Este aspecto es relevante a los juegos 3D principalmente.
- La división o no del escenario por zonas. Esto puede evitarnos la obligatoria presencia en memoria de un mapa completo, pudiéndolo dividir en partes para ahorrar en el preciado recurso que es la memoria. Es un aspecto a tener en cuenta sobre todo cuando se elaboran juegos para videoconsolas, las cuales suelen contar con una memoria muy limitada.

Así, se han elegido dos criterios principales como forma de clasificar los tipos de escenario:

- Organización del escenario. Este criterio se refiere a la existencia de cortes en el desarrollo del juego para la carga de las distintas partes del mundo en el que se desarrolla el juego. Algunos tipos son:
  - Escenarios totalmente continuos continuos representando un mundo totalmente abierto. Ejemplos son GTAV o Minecraft.
  - Dividido por zonas donde no existen cortes en la misma zona. Un ejemplo es Borderlands.
  - Dividido por zonas donde pueden existir cortes en la misma para interiores. Un ejemplo es Skyrim.
- Composición del escenario. Nos referimos a si la geometría empleada ayuda de forma implícita al manejo a nivel computacional del escenario. Algunos tipos son:

- Escenarios en 2D discretizados en *tiles*. Básicamente, la geometría está representada por una matriz de dos dimensiones, donde cada casilla de la matriz está relacionada con una textura, siendo todas las texturas del mismo tamaño para la misma matriz. Este tipo de escenarios son muy fácilmente optimizables. Ejemplos de este tipo es, por ejemplo, Final Fantasy VI.
- Una extensión a 3D de los escenarios de tiles son los escenarios compuestos por *voxels* [3]. Estos han sido recientemente popularizados por el famoso título Minecraft, donde el escenario está gestionado por un *motor de voxels*.
- Escenarios con geometría deliberada. Es decir, la geometría (ya sea 2D o 3D), no presenta ninguna propensión a ser optimizada. Este tipo de escenarios, suelen optimizarse con el uso de *quadtrees* [4] (para 2D) u *octrees* [5] (para 3D)
- Escenarios hechos con los denominados *brushes* [6]. Esta técnica se utiliza para 3D, y consiste en el uso de geometrías convexas para componer un escenario. Ejemplos famosos de este tipo de escenarios es *Quake* [7].

Una vez expuesta los criterios para clasificar los tipos de mapas, remarcar que hay casos en los que los desarrolladores convienen formas nuevas de organizar el mapa debido a necesidades particulares, por lo que los ejemplos expuestos en cada criterio no completan de ninguna manera las múltiples categorías para cada uno.

Para el desarrollo de este proyecto y atendiendo a las especificaciones presentadas en el capítulo 2, abarcaremos *mapas de tiles* para el género de juego *roguelike* [8]. Este género se caracteriza precisamente por mapas generados de forma procedimental. Cada nivel es entendido como una planta, donde el jugador tiene que llegar desde donde aparece hasta un tile considerado como final para poder continuar hasta la siguiente planta. Así, el jugador va avanzando niveles hasta que llega al último y completa el juego al finalizar el mismo.

Motivación y Objetivos.

Como se ha mencionado previamente, la industria de los videojuegos avanza a pasos de gigante. Es por ello que las compañías invierten cada

vez más en videojuegos, siendo a veces el coste en marketing superior al de desarrollo [9]. Existe un amplio espectro de puestos encontrados en este campo [10] [11], entre los cuales existen

- *Game programmer*. Personal dedicado a la programación de las mecánicas del juego, eventos y otros relacionados con el propio juego
- *Core programmer*. Personal dedicado a la elaboración del motor que será la base sobre la que funcionará el juego. Normalmente se subdivide en subroles como *graphics programmer* o *physics programmer*. En muchos casos, se parte de un motor ya elaborado sobre el que se realizan las modificaciones necesarias para el juego en cuestión.
- *Game designer*. Dedicados a la elaboración de las mecánicas del juego, así como el plot del mismo.
- *Level designer*. Personal dedicado exclusivamente a la elaboración de escenarios empleando programas externos o el mismo motor que se esté utilizando, si éste posee dichas capacidades.
- *AI programmer*. Dedicado a la programación de los distintos sistemas de inteligencia artificial existentes en el juego.
- *Muchos más...*

En este listado, solo hemos tenido en cuenta algunos de los puestos del aspecto de desarrollo, pero quedaría añadir muchos más como diseñadores gráficos y apartado de marketing.

Destacar que el *level designer*, pese a que no se encarga de la construcción de las herramientas para crear el escenario, debe conocer las bases del tipo de escenario que se empleará en el juego, de forma que adapte sus técnicas de diseño al tipo de escenario.

Recientemente, y debido a las cada vez más crecientes facilidades para crear un videojuego, han surgido los llamados *equipos indies* [**?**], caracterizados principalmente por tener un bajo presupuesto y personal. Normalmente suelen empezar con un presupuesto nulo, pero en algunos casos llegan a triunfar de manera inesperada incluso por los mismos desarrolladores. Ejemplos son *Hotline Miami*, *Minecraft* o *Risk of Rain*.

Los equipos con un presupuesto considerable (coloquialmente denominados AAA), no tienen problemas a la hora de contratar *diseñadores de*

niveles, ya que disponen de una gran cantidad de dinero para depositar en los distintos roles necesarios para un juego. Aún así, en combinación con un generador de mapas, un *diseñador de niveles* puede desarrollar ideas que den un resultado muy bueno. Ejemplos de esto son *Diablo* o *Torchlight*, donde los mapas son generados automáticamente partiendo de patrones elaborados a mano por *diseñadores de niveles*.

Así, el la motivación de este proyecto reside en dos ideas principales.

- Ahorro de coste y tiempo para equipos indies, que no pueden permitirse el lujo de gastar en personal exclusivo para el diseño de niveles.
- Adición de variedad a los escenarios de juegos, permitiendo incluso a un equipo de desarrolladores AAA

Solución Propuesta.

La solución que se propone es un sistema capaz de generar escenarios de manera automática con una mínima interacción (o incluso nula si se desea) de los diseñadores. Esto además, conlleva dinamismo en los escenarios que se podrán jugar, de forma que de partida a partida, la distribución del escenario será completamente distinta.

La elaboración de un generador de escenarios, une las competencias de dos roles en el desarrollo de videojuegos:

- *Diseñador de niveles.* Se necesitan conocimientos sobre la composición de los escenarios del juego, además de las propiedades concretas en cuanto a los criterios mencionados anteriormente.
- *Programador de IA*. La creación del sistema que genere los escenarios es un trabajo que compete a este rol, ya que estamos tratando de resolver un problema donde las posibilidades de resolución son casi infinitas.

Finalmente, se elaborará el sistema reduciendo el problema a una búsqueda, donde el *espacio de estados* son todas las posibles combinaciones de habitaciones conectadas.

Estructura de la memoria.

Esta memoria se estructura en varios capítulos, con la siguiente distribución de los temas trabajados:

- Capítulo 1. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.
- Capítulo 2. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.
- Capítulo 3. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.
- Capítulo 4. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.
- Capítulo 5. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.
- Capítulo 6. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.
- Capítulo 7. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.
- Capítulo **??**. Nunc viverra volutpat bibendum. Nunc augue orci, tempus nec interdum sit amet, ornare id elit. Integer congue risus vitae

ipsum pharetra rutrum. Curabitur mollis sagittis pretium. Etiam a lacus sed mauris rhoncus ullamcorper non sit amet felis.

- Bibliografía.

Capítulo 1

Estado del arte.

En este capítulo, analizaremos el ámbito de la generación procedimental de contenido para videojuegos. Veremos las distintas subdisciplinas presentes, así como los problemas que resuelven. Se hará hincapié en la generación de escenarios, ya que es el campo que compete al proyecto.

1.1. Generación de contenido procedimental.

Más conocida por su nombre en inglés (Procedural Content Generation), se refiere a la disciplina de generar contenido partiendo de algoritmos, en lugar de hacerlo manualmente.

Tuvo sus inicios en la subcultura informática llamada *Demoscene* [12]. Este movimiento tuvo sus inicios a finales de los años 70 y principios de los 80, y continúa a día de hoy. Consiste en crear contenido visual y sonoro de forma programada, ya sea en parte o en su totalidad, y tenía como uno de sus objetivos escudriñar al máximo las limitadas capacidades de los ordenadores de la época.

Los *sceners* de este campo, conseguían generar contenidos que eran impensables para la época, como por ejemplo escenas en 3D cuando a OpenGL aún le quedaban un par de décadas para aparecer. Por su habilidad, la mayoria de los demosceners terminaban trabajando para empresas de videojuegos de la época, cuando aún no había tantas facilidades a la hora de crearlos.

Existen diversas subdisciplinas en las que se aplica el concepto de generación procedimental de contenido:

- *Escenarios*.
- Texturas [13]. Un ejemplo de ello es la generación de texturas que imitan el mármol o la madera. En algunos casos se emplean autómatas celulares.
- Geometría. Generación de follaje, o árboles empleando L-Systems. [13]
- Mecánicas. El juego Left4Dead y su secuela, emplean un sistema procedimental para gestionar los momentos de tensión y la dificultad de las situaciones según las acciones que han ido tomando los jugadores.

Otra clasificación de los métodos procedimentales es según el momento de ejecución del procedimiento. Si el procedimiento es ejecutado antes del lanzamiento del juego, o después con la particularidad de que se realiza en servidores ajenos al jugador, lo denominaremos *offline*. Sin embargo, si el procesamiento se realiza en el sistema en que se está ejecutando el juego, lo llamaremos *online*.

1.2. Generación de mapas.

La generación de escenarios se considera un concepto muy amplio, ya que según el contexto del juego, puede variar bastante. Por ejemplo, en un juego ambientado en el espacio exterior, el escenario puede entenderse como una galaxia completa, como es el caso de *Elite: Dangerous*.

Otro tipo de modelo a usar en la generación de escenarios es el ruido Perlin. Éste puede ser muy útil en conjunto con el concepto de mapas de alturas para generar terrenos al aire libre montañosos [14]. También se ha empleado el *algoritmo de Voronoi* para la generación de terrenos [15].

Algún ejemplo más histórico de generación de escenarios pueden ser los múltiples algoritmos de generación de laberintos que se conocen [16].

Debido a que los métodos anteriores son demasiado genéricos, una opción muy popular es idear y construir un generador específico para el juego en cuestión que se esté desarrollando. Un ejemplo de ello es *TinyKeep*, cuyo autor describe a grandes rasgos en [17] las fases y el desarrollo del algoritmo que ha creado. Esta opción es la que se desarrollará en este proyecto, procurando mantener un nivel de genericidad y capacidad de personalización del sistema.

Capítulo 2

Especificaciones.

En este capítulo definiremos las especificaciones y directrices tanto del tipo de juego como del sistema de generación. Dichas directrices que delimitarán el sistema a crear, han sido establecidas por la empresa indie de videojuegos *TheGameKitchen*, y el sistema será empleado en un futuro título de la misma.

2.1. Género de juego.

En esta sección abarcaremos el tipo de juego y las directrices establecidas en cuanto al género de juego y otras especificaciones relacionadas con el mundo en el que se desarrollará el mismo.

2.1.1. Roguelike

El género de juego en el que nos basaremos será *roguelike* [8], cuyas características principales son:

- *Generación de mazmorras*. Cada vez que el jugador inicia una partida, la experiencia será ligeramente distinta.
- *Importancia considerable a la exploración*. El hecho de que las mazmorras no sean siempre iguales, incita al jugador a tener que invertir

tiempo en explorar para poder encontrar la salida.

- *Desarrollo del juego por plantas*. El objetivo del jugador suele ser llegar a una habitación considerada como final, donde puede elegir entre tomar las escaleras para pasar a una siguiente planta, o investigar un poco más en la presente.

- *Dificultad progresiva*. Cada planta, tendrá una dificultad ligeramente mayor a la de la anterior hasta llegar a la última.

- *Muerte permanente*. Una vez que el jugador muere, no hay manera de cargar la partida. La unica opción es comenzar de nuevo.

Históricamente, el género *roguelike* se identificaba además por otro tipo de características, como la mecánica por turnos o el énfasis en jugabilidad y desinterés en los gráficos, pero con el tiempo, el género se ha ido abriendo paso a una definición más genérica. Debido a ésto, hoy en día existen desde *shooters* considerados *roguelike*, como por ejemplo *Tower of Guns* o *Paranautical Activity*, hasta *plataformas*, como *Risk of Rain* o *Spelunky*.

2.1.2. Especificaciones del mundo

Así, las únicas especificaciones que se han dado en cuanto a las reglas de juego son las siguientes:

- Entorno en dos dimensiones

- El jugador comenzará en una habitación y tendrá como objetivo llegar a una habitación final

- Se empleará una vista cenital (figura 2.2).

Estas especificaciones son bastante genéricas, por lo que el sistema elaborado podrá servirnos para otro tipo de juegos. Más adelante veremos que se ha procurado dar un enfoque de alta flexibilidad en este aspecto al sistema.

Figura 2.1: Vista cenital en el juego Action Hollywood por TCH, 1995

2.2. Requisitos del escenario.

La representación de los escenarios elegida será un mapa de tiles. Como se comentó anteriormente, los mapas de tiles están representados por una matriz, donde cada casilla corresponde a un gráfico de un tamaño normalizado para todos los tiles (un gráfico para las paredes, otro para el suelo, etc). Suelen componerse por capas para añadir decorado, pero para nuestra finalidad, hemos obviado esta característica, ya que se hará hincapié principalmente en la distribución de las habitaciones. No obstante, sería muy sencillo añadir capas como mejora futura.

La optimización en cuanto a clipping (evitar renderizar zonas no visibles) en un mapa de tiles está muy investigada. Debido a que el propio mapa es una rejilla, se puede recorrer y renderizar exclusivamente la zona que va a verse. En la figura 2.2 se representa en rojo el viewport del jugador. Los tiles amarillos están en el borde del viewport, y los verdes están completamente dentro. Así, solo hace falta recorrer y dibujar los tiles verdes y amarillos, ahorrándonos el dibujado de los azules.

Otra característica importante de los mapas de tiles es la sencillez y flexibilidad que da a la hora de componer escenarios; podemos reutilizar tiles y añadir variaciones de los mismos para, en el mapa final, dar sensación de variedad sin tener que elaborar muchos gráficos.

Por todo ello, es un tipo de escenario muy popular desde las antiguas

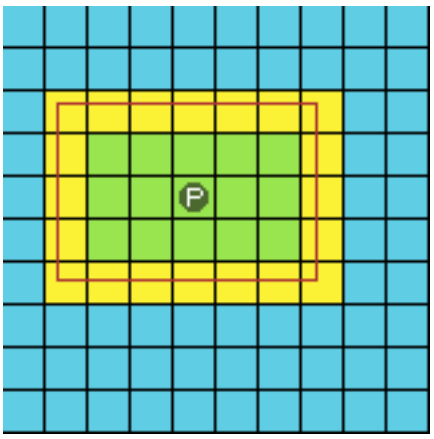

Figura 2.2: Clipping en un mapa de tiles

consolas como *NES* o *SNES* hasta hoy.

El escenario estará compuesto por habitaciones, cada una representada como una matriz. Será nuestra tarea plantear un sistema que elabore distribuciones de las habitaciones según los requisitos que se expondrán a continuación.

2.3. Directrices para la generación.

En esta sección explicaremos las directrices impuestas para la generación de los escenarios.

2.3.1. Lista inicial de habitaciones

El sistema que construiremos deberá cumplir una serie de características que analizaremos en detalle en esta sección. Como se ha mencionado previamente, se ha tomado la libertad de añadir características que no estaban en los requisitos previos de *TheGameKitchen*.

El sistema debe generar un mapa de tiles partiendo de una *lista inicial de habitaciones* previamente construida. Para construir las habitaciones, se ha elaborado un editor via web usando el canvas que proporciona HTML y JavaScript (se puede ver en la figura 2.3). Así, el objetivo será colocar las

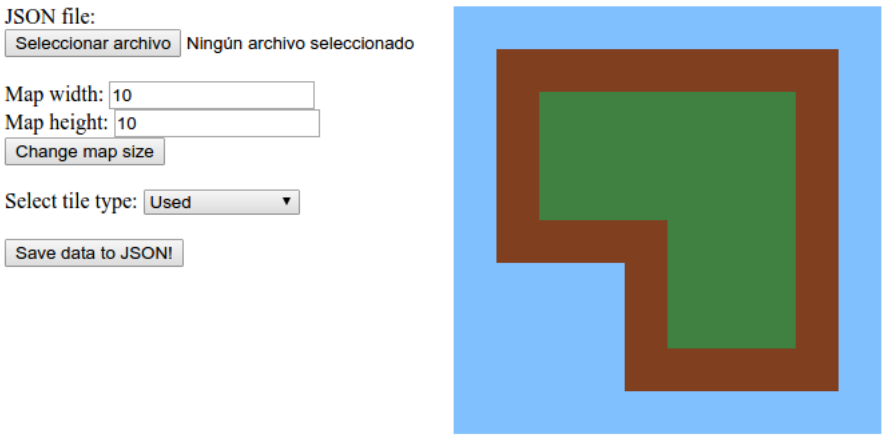

Figura 2.3: Editor de habitaciones

habitaciones presentes en esta lista inicial.

Las habitaciones *pueden repetirse*, es decir, si hemos creado dos modelos de habitación A y B, podemos tener 3 instancias del modelo A y 2 instancias del modelo B en la lista inicial. Hablaremos de este detalle en el próximo capítulo, que nos servirá para conseguir una mejora de optimización tanto de memoria como en procesamiento.

2.3.2. Requisitos de distribución

A continuación, explicaremos los requisitos que ha de cumplir la distribución las habitaciones.

Se ha de *maximizar el camino* desde la habitación inicial hasta la habitación final. De esta forma conseguimos en parte promover que el jugador tenga que investigar.

Obviamente, se ha de conseguir una variabilidad en los niveles añadiendo un componente aleatorio, de forma que la experiencia de cada jugador (o incluso del mismo en partidas distintas) no sea idéntica.

Es indiferente que par de habitaciones son la inicial y la final, podemos elegir cualquier par.

Partiremos de un mapa de tiles vacío, cuyo tamaño supondremos suficiente como para albergar todas las habitaciones para cualquier distribu-

ción de las mismas. Para ello, es posible crear un mapa con tamaño autoajustable según sea necesario, pero se ha optado por elegir un tamaño lo suficientemente grande para hacer las pruebas.

Se ha de fomentar la existencia de caminos alternativos, no necesariamente caminos alternativos a la solución, sino más bien callejones sin salida, de forma que incite aún más a la exploración.

2.3.3. Requisitos técnicos

La generación será *online*, implicando que el sistema ha de actuar en un tiempo considerable para evitar largas esperas. Para esto, se ha construido un sistema de caché del que hablaremos más adelante.

Debe funcionar en sistemas móviles. Esto implica un especial cuidado en la optimización y en las librerías utilizadas. Para abarcar este punto correctamente, se ha utilizado Java como lenguaje de programación, ya que es el más popular para sistemas móviles. Además, el sistema de caché que se mencionó antes, nos servirá también en este aspecto.

Capítulo 3

Representación.

En este capítulo entraremos en detalle de la representación de los distintos elementos que componen el generador. Repasaremos el concepto de *mapa de tiles* de nuevo. Analizaremos la representación lógica y física necesaria con el objetivo de establecer un modelo claro para las habitaciones y el mapa que usaremos posteriormente para la construcción del sistema de generación.

En nuestro contexto, disponemos de dos tipos de entidades:

- *Habitaciones* que compondrán el mapa.
- *Mapa* o escenario donde se desarrolla el juego, compuesto por habitaciones. Posee algunos elementos ausentes en las habitaciones.
- *Puertas* que servirán de conexión entre habitaciones.

Es importante tener claro el aspecto de la representación de las entidades que se utilizarán en el sistema, ya que el mismo se construirá en base a lo que se detalle en la representación.

3.1. Topología.

Como se ha repetido varias veces anteriormente, usaremos un *mapa de tiles* para la representación del escenario. Esto implica que los modelos tan-

to de las habitaciones como del propio mapa, han de mantener una matriz para la representación física. En la figura 3.1 podemos ver la representación en memoria como una matriz de un mapa de tiles. En la figura 3.2 se observa la representación gráfica. Se puede comprobar la relación directa entre ambas representaciones.

0	1	1	1	0
0	1	0	1	1
1	1	0	0	1
1	0	0	0	1
1	1	1	1	1

Figura 3.1: Representación en memoria de un mapa de tiles

En principio, los tipos de tile que se han considerado son los siguientes:

- *Tile pared*. Representa una porción de geometría sólida infranqueable. A la hora de colocar habitaciones, habrá que añadir un mecanismo para evitar el solapamiento entre tiles sólidos de entidades distintas.
- *Tile interior*. Representa una porción de geometría traspasable. A igual que el tile sólido, habrá que evitar solapamiento con tiles pertenecientes a otras entidades (habitaciones o el mapa).
- *Tile exterior*. Representa una porción de geometría traspasable *externa* a la habitación. Este tipo de tiles sí que puede solaparse con tiles pertenecientes a otras entidades.
- *Tile puerta*. Representa una conexión entre dos habitaciones. No podrá solaparse, pero si traspasarse para permitir el cruce entre habitaciones.

En la figura 3.3 se ve un resumen de las propiedades inherentes a cada tipo de tile.

Con todo ésto, ya tenemos clara la representación física de los distintos elementos implicados en el sistema de generación. Ahora, entraremos en detalle con elementos específicos para el mapa y la habitación, que principalmente nos serán necesarios para algunas optimizaciones y llevar a cabo la lógica del generador.

Más adelante veremos un tipo de tile especial (tile tipo *puerta*) en el que nos apoyaremos para la conexión entre habitaciones.

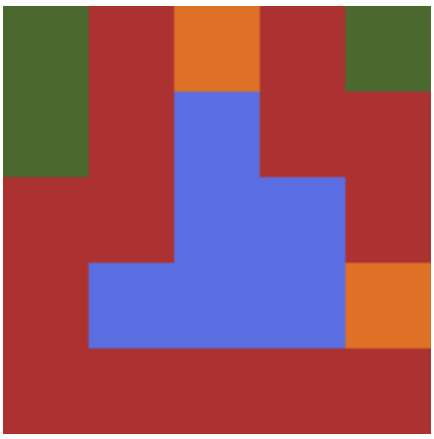

Figura 3.2: Representación gráfica de un mapa de tiles

	Sólido	Solapable
Pared	✓	X
Interno	X	X
Externo	X	✓
Puerta	X	X

Figura 3.3: Propiedades de los tiles

3.2. Habitaciones.

Analizaremos los elementos relevantes para la representación de las habitaciones. Atendiendo a la especificación de que las habitaciones *se pueden repetir*, se ha dividido la representación en dos componentes:

- *Prefab*. Representación que engloba los elementos comunes para un modelo de habitación. Sirve, entre otras cosas, para almacenar solamente un mapa de tiles por modelo de habitación.
- *Instancia*. Representación específica para una concretización o instancia de un modelo de habitación o prefab. Contendrá información sobre la posición de una habitación en el mapa y las puertas.

Ésto nos permite un ahorro de memoria, ya que guardamos información común a muchas habitaciones en un solo modelo, y procesamiento, que veremos en detalle en el capítulo 4 (INSERTAENLACE).

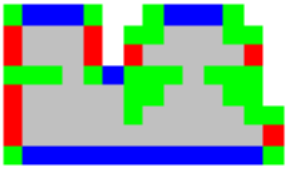

Figura 3.4: Representación gráfica de puertas potenciales verticales y horizontales

3.2.1. Puertas potenciales.

Antes de abarcar los dos componentes empleados para representar las habitaciones, vamos a introducir el concepto de *puerta potencial*, que será necesario para explicar una de las motivaciones del *prefab*, además de tener que mantener solo un mapa de tiles por modelo.

Con el concepto de puertas potenciales nos referimos a las posibles conexiones de un modelo de habitación. Cualquier tile que cumpla la restricción de ser puerta potencial, puede convertirse en puerta para establecer una conexión entre habitaciones en el transcurso de la generación en el sistema.

Así, consideraremos que un tile $t(x, y)$ será una puerta potencial de la habitación R teniendo en cuenta las dos orientaciones posibles.

- *Horizontal* si se cumple:
  - Existe un tile *pared* a la izquierda y derecha del tile.
  - Existe un tile *interior* arriba y un tile *exterior* abajo, o viceversa.
- *Vertical* si se cumple:
  - Existe un tile *pared* arriba y abajo del tile.
  - Existe un tile *interior* a la izquierda y un tile *exterior* a la derecha, o viceversa.

En la figura 3.5 se puede ver la definición formal del concepto de *puerta potencial*, y en la figura 3.4, una representación gráfica, teniendo en cuenta ambos casos.

$$t \in R, \begin{cases} outer(x+1,y), & inner(x-1,y), & wall(x,y\pm 1) & vertical \\ outer(x-1,y), & inner(x+1,y), & wall(x,y\pm 1) & vertical \\ outer(x,y+1), & inner(x,y-1), & wall(x\pm 1,y) & horizontal \\ outer(x,y-1), & inner(x,y+1), & wall(x\pm 1,y) & horizontal \end{cases}$$

Figura 3.5: Definición formal de una puerta potencial y sus casos

Según la complejidad de las habitaciones, con este concepto conseguiremos ahorrarnos casos en los que será imposible conectar habitaciones. Obviamente, habrá casos en los que aún siendo un tile una puerta potencial, es imposible conectarla con otra habitación por motivos de solapamiento entre las mismas. Aún así, podemos ahorrarnos parte de las comprobaciones.

3.2.2. Prefabs.

Como se ha mencionado anteriormente, los prefabs representarán el modelo de un tipo de habitación, común a todas las instancias o concretizaciones del mismo. Las motivaciones de este componente son de eficiencia, tanto de memoria como de procesamiento.

En cuanto a memoria, solo tendremos que guardar un mapa de tiles, ya que las instancias correspondientes a un prefab, guardarán la misma representación física. Además, solo será necesario guardar la lista de puertas potenciales en el prefab, ya que también será común a todas las instancias.

El cómputo de las puertas potenciales también conlleva un coste, y gracias a este componente, se realizará una sola vez en la carga.

3.2.3. Instancias.

La instancia representa una concretización de un prefab. Una vez tenemos los modelos de habitación necesarios, se crearán las instancias pertinentes a partir de cada modelo. En estas instancias, incluiremos la información necesaria para situar las habitaciones en el mapa, así como para

poder manejar las conexiones entre puertas de las distintas instancias de habitaciones y poder realizar estimaciones sobre el mapa.

Para conectar dos instancias de habitación, necesitaremos una puerta por cada instancia, de forma que ambas puertas sean contiguas para que el jugador pueda cruzar de una habitación a otra. En la figura 3.6 se puede observar la definición formal de ésto.

$$p1(x1, y1) \in R1, p2(x2, y2) \in R2, \begin{cases} p1 = p2 + (0, 1) & \text{conexión horizontal} \\ p1 = p2 - (0, 1) & \text{conexión horizontal} \\ p1 = p2 + (1, 0) & \text{conexión vertical} \\ p1 = p2 - (1, 0) & \text{conexión vertical} \end{cases}$$

Figura 3.6: Conexión entre dos instancias de habitación

Así, guardaremos una lista de las puertas presentes en la instancia.

Otro dato importante a mantener en a instancia es una referencia al prefab, de forma que podamos realizar los cómputos necesarios que corresponden al modelo de habitación.

Por último, necesitaremos conocer la posición de la instancia en el mapa. Esta posición no se usará mientras la instancia no esté colocada en el mapa. Asimismo, se establecerá el valor de este dato, cuando coloquemos la instancia en el mapa.

Cabe destacar que a efectos lógicos, dos instancias de un mismo prefab son consideradas como habitaciones *distintas*. La utilización del prefab es solo una medida de conveniencia para evitar la duplicación de información.

3.3. Puertas.

Una vez elegido un par de tiles de puertas potenciales de dos instancias de habitación distintas que conectaremos, tenemos que añadir cierta información a las puertas, que será necesaria para el funcionamiento del sistema.

- *Instancia de Habitación propietaria* de la puerta
- *Puerta conectada* miembro de la instancia de habitación conectada.
- La *habitación conectada* se puede extraer a partir de la *puerta conectada*.
- *Posición local* relativa a la instancia de habitación a la que pertenece la puerta.
- *Tipo* de puerta. Se refiere a la orientación: horizontal o vertical.

Con toda esta información, podemos tratar las puertas como elementos independientes, ya que se tiene acceso a todos los componentes necesarios para cualquier cómputo o acción que desee realizarse sobre la conexión que representa esta puerta.

3.4. Mapa.

Por último, el modelo que usaremos para el mapa contendrá información física y lógica de las habitaciones y su disposición en el escenario.

El modelo empleado para el mapa, empleará un mapa de tiles donde se colocarán las habitaciones de forma física. El mapa de tiles nos servirá para comprobar solapamiento a la hora de colocar habitaciones para conectarlas. Además, una vez completo el mapa, tendremos el resultado final de la generación en este mapa de tiles.

El mapa de tiles empleado en el mapa, según las especificaciones, se supone lo suficientemente grande como para albergar cualquier distribución de habitaciones. Se ha elegido hacer uso de un mapa de tiles lo suficientemente grande para cumplir este requisito, pero podría haberse elaborado un mapa autoajustable, de forma que, en caso de ser necesario, se expandiría el tamaño.

Resultará práctico mantener las conexiones entre las instancias de habitaciones presentes en el mapa. Para ello, se ha hecho uso de una matriz superior, tal que en cada posición $(R1, R2)$, tendremos una estimación de la distancia entre las habitaciones $R1$ y $R2$. Consideraremos un valor muy alto como la ausencia de conexión. Esta matriz superior nos será de mucha utilidad a la hora de computar propiedades del mapa de forma rápida.

También será útil mantener una lista con el conjunto de todas las puertas potenciales de las habitaciones en el mapa. Se hablará en detalle de ésto en el próximo capítulo.

Capítulo 4

Estrategia constructiva.

Hasta ahora hemos visto la representación de las distintas entidades que entran en juego. En este capítulo introduciremos parte de la lógica que seguirá el sistema. Destacar que, aunque el enfoque final ha sido reducir el problema a una modificación de búsqueda, el sistema está pensado de forma que sea flexible a la hora de utilizar otro tipo de estrategias.

4.1. Algoritmo de generación

A la metodología que hemos utilizado la hemos denominado *estrategia constructiva*, cuyo origen radica en la forma de generar el mapa. Como se puede ver en el listado 4.1, el sistema construirá el mapa por pasos. de forma que en cada paso se hará una elección de una habitación y se colocará en el mapa. Así, el sistema dará por concluida la generación cuando no queden habitaciones en la lista inicial.

Listado 4.1: Algoritmo constructivo para generar mapas

```
1  Mapa GenerarMapa( List<Habitacion> habitaciones, InterfazSeleccion mapSolver
       ) {
2    Mapa mapa = MapaVacio();
3    while( !habitaciones.isEmpty() ) {
4      List<Movimiento> movimientos = GenerarMovimientos( mapa, habitaciones );
5      Movimiento elegido = mapSolver.ElegirMovimiento( );
6      mapa.InsertarHabitacion( elegido.habitacion, elegido.posicion );
7      habitaciones.remove( elegido.habitacion );
8      GuardarMovimiento( elegido );
9    }
10   EstablecerRecorridoPrincipal( mapa );
```

```
11    return mapa;
12  }
```

En el listado 4.1 se introducen dos conceptos nuevos: *movimientos* e *interfaz de selección de movimiento*, y ambos son clave para el entendimiento del funcionamiento del sistema.

A grandes rasgos, se elige un par $(habitacion, posicion)$ y se inserta en el mapa (línea 6). Posteriormente, se elimina la habitación de la lista de habitaciones, ya que se acaba de colocar (línea 7). Por último, guardamos el movimiento (línea 8). Más adelante veremos la utilidad de ésto.

Antes de devolver el mapa, establecemos el recorrido principal (línea 10), es decir, elegimos la habitacion inicial y final. Para ello, se ha empleado el algoritmo de *Floyd-Warshall*, ayudándonos de la *matriz superior de conexiones entre habitaciones* comentada en el capítuloX (INSERTARENLACE). El objetivo algoritmo *Floyd-Warshall* (APENDICEINSERTARENLACE) es, dado un grafo, establecer los caminos mínimos entre todos los pares de nodos posibles. Así, para establecer el recorrido principal, elegiremos el par de nodos cuya distancia mínima sea mayor.

4.2. Movimientos

Un movimiento $M_i(R_i, P_i)$ está constituido por:

- R_i: *instancia de habitación* a colocar en el mapa
- P_i: *posición* del mapa donde colocaremos dicha habitación

Así, en cada paso de colocación de una habitación, se generarán todos los movimientos posibles a realizar (listado 4.1, línea 4). La lista de movimientos posibles dependerá del estado del sistema, como veremos a continuación.

4.2.1. Cómputo de posibles movimientos.

Consideraremos que el *estado del sistema* depende de, y se ve afectado por:

- Instancias de habitación restantes en la lista inicial
- Modelo de instancias de habitaciones colocadas en el mapa
- Posición de las instancias de habitaciones colocadas

Como se observa en la figura 4.1, dado un estado del sistema $S(map, RR)$, la lista de posibles movimientos se computará a partir de las *posibles conexiones* entre las *puertas potenciales de las habitaciones colocadas en el mapa* y las *puertas potenciales de las habitaciones restantes*. El concepto de *puerta potencial* se discutió en el CAPITLOX (insertar enlace).

$$S(map, RR), RR = habitacionesRestantes$$

$$movimientos = \bigcup_{r \in RR} posiblesConexiones(map, r)$$

Figura 4.1: Cómputo de lista de movimientos dado un estado del sistema

El cómputo de las posibles conexiones entre el mapa y una de las habitaciones restantes, se define en la figura 4.2. Destacar que el desplazamiento para testear la posible conexión entre una habitación de las restantes y una del mapa, solamente se puede realizar sobre la habitación restante, ya que las del mapa ya han sido colocadas y están fijas.

El cómputo de todos los posibles movimientos queda como la unión de los conjuntos que dan como resultado calcular las posibles conexiones entre todas las habitaciones restantes y las presentes en el mapa.

Así, el cómputo de todas las posibles conexiones entre una habitación en la lista de habitaciones restantes y una habitación del mapa consiste en calcular el conjunto de todas las posibles colocaciones de la habitación de la lista de restantes contigua a la habitación del mapa.

4.3. Interfaz de selección de movimiento

Como se ha comentado anteriormente, la *interfaz de selección de movimiento* es otro elemento importante del sistema, y en ella radica parte de la

$$posiblesConexiones(map, r) =$$

$$\bigcup_{p \in rooms(map)} \{cx(map, r, p, u) : \neg col(map, r, u), u \in map\}$$

$$u = \begin{cases} u = v + (1, 0) \\ u = v - (1, 0) \\ u = v + (0, 1) \\ u = v - (0, 1) \end{cases}$$

$col(map, room, t)$: colisión al colocar $room$ en map en el tile t

$cx(map, r, p, u)$: acción de conectar r con p colocando r en tile u de map

Figura 4.2: Cómputo de las posibles conexiones entre una habitación de las restantes y el mapa

flexibilidad del sistema. En el listado 4.2 se muestra el esqueleto de dicha interfaz, cuyo único método de interés es el relacionado con la elección del movimiento.

Listado 4.2: Interfaz de generación

```
1 interface InterfazSeleccion {
2   Movimiento ElegirMovimiento( List<Movimiento> movimientos );
3 }
```

Una vez generados todos los movimientos posibles, se delega la decisión de elegir un movimiento a la *interfaz de generación*. En esta interfaz radica uno de los puntos de flexibilidad del sistema. Mediante componentes que implementen esta interfaz, podemos idear otras formas de elegir un movimiento a partir de la lista de movimientos posibles.

En este proyecto se han creado dos interfaces de selección de movimiento:

- *Interfaz aleatoria*. Elige un movimiento de forma aleatoria de entre todos los posibles.
- *Interfaz basada en búsqueda*. Asigna una puntuación a cada movimiento y elige la mejor.

La interfaz aleatoria es simple y no tiene interés computacional. En el listado 4.3 se muestra la implementación de dicha interfaz por completitud, pero se ha realizado solo para testeo.

Listado 4.3: Implementación aleatoria de la interfaz de selección de movimiento

```
1 class RandomMovementSelector implements IMovementSelector {
2   Movimiento ElegirMovimiento( List<Movimiento> movimientos ) {
3     int indiceMovimiento = random( 0, movimientos.size() - 1);
4     Movimiento elegido = movimientos.get( indiceMovimiento );
5     return elegido;
6   }
7 }
```

Capítulo 5

Interfaz basada en búsqueda.

En este capítulo se abarcará en cierta profundidad la *interfaz de selección basada en búsqueda*. Se han omitido algunos detalles como la forma de crear y manejar las puertas, ya que se considera trivial y fuera del objetivo de la práctica. Además, algunos concepto como el concepto de *movimiento* no se traduce literalmente en el código, pero la idea esencial se conserva.

5.1. Estrategia basada en búsqueda

Como se observa en el listado 5.1, la forma de seleccionar el movimiento incluye el cálculo de una puntuación numérica que llamaremos *fitness*. Para obtener el *fitness*, aplicaremos una función que denominaremos como *función guía*, y devolverá un valor numérico indicando la calidad del movimiento.

Listado 5.1: Interfaz de selección de movimiento basada en búsqueda

```
1  class BestSearchMovementSelector implements IMovementSelector {
2    Movimiento ElegirMovimiento( List<Movimiento> movimientos ) {
3      List<Float> fitnesses;
4      for( Movimiento m : movimientos ) {
5        fitnesses[m.index] = FuncionGuia( m );
6      }
7      Movimiento elegido = ObtenerMejor( movimientos, fitnesses );
8      return elegido;
9    }
10 }
```

Aplicando la *función guía* a todos los movimientos, se computará el *fitness* para cada uno. Por último, elegiremos el movimiento cuyo *fitness* sea mejor. Convendremos que la calidad del *fitness* será directamente proporcional al mismo, es decir, cuanto mayor *fitness*, mejor movimiento.

5.2. Interfaz de cómputo de fitness

Para el cómputo del fitness, se ha empleado una interfaz que definimos en el listado 5.2, añadiendo flexibilidad a este componente. De esta forma, podemos también idear formas nuevas de computar el fitness y llevarlas a la práctica fácilmente.

Listado 5.2: Interfaz de cómputo de fitness

```
1 interface IFitnessSolver {
2   float FuncionGuia( Movimiento m );
3 }
```

En el listado 5.4 se ha sustituido la llamada a la función guía por la interfaz de cálculo de fitness. De esta forma, añadimos un componente de flexibilidad en esta parte, permitiendo la implementación de diferentes formas de calcular el fitness.

Listado 5.3: Interfaz de selección de movimiento basada en búsqueda

```
1  class BestSearchMovementSelector implements IMovementSelector {
2    IFitnessSolver fitnessSolver;
3    Movimiento ElegirMovimiento( List<Movimiento> movimientos ) {
4      List<Float> fitnesses;
5      for( Movimiento m : movimientos ) {
6        fitnesses[m.index] = fitnessSolver.FuncionGuia( m );
7      }
8      Movimiento elegido = ObtenerMejor( movimientos, fitnesses );
9      return elegido;
10   }
11 }
```

En las líneas 2 y 6 se observa el cambio sustancial que representa la interfaz de cálculo de fitness.

Se han elaborado dos implementaciones de la interfaz de cómputo de fitness. Una de ellas es sencilla, y define el fitness una puntuación numérica de *una sola propiedad del mapa*, en concreto, la distancia del camino principal (lo analizaremos más adelante (ENLACE A TAMAÑO CAMINO

PPAL). Lo importante a destacar de esta implementación es que el fitness se define como un solo valor, mientras que en la segunda implementación veremos como computamos el fitness a partir de múltiples propiedades del mapa.

5.2.1. Cómputo de fitness múltiple

El propósito de esta implementación de la interfaz de cómputo de fitness es poder utilizar diferentes propiedades del mapa que influyan en la selección del movimiento a través del fitness. En el listado **??**, se ha empleado un número constante de 3 propiedades del mapa por utilidad, pero podría elaborarse para generalizar para aceptar un número cualquiera de propiedades.

Listado 5.4: Interfaz de selección de movimiento basada en búsqueda

```
1  class MultiFitnessSolver implements IFitnessSolver {
2    IFitnessCombinator fitnessCombinator;
3    float FuncionGuia( Movimiento m ) {
4      float fitnesses = new float[3];
5      fitnesses[0] = ComputarPropiedad0( m );
6      fitnesses[1] = ComputarPropiedad1( m );
7      fitnesses[2] = ComputarPropiedad2( m );
8      fitnessCombinator.Combinar( fitnesses );
9    }
10 }
```

Se observa en el listado **??** que se emplea otra interfaz para dar flexibilidad a la hora de combinar las múltiples propiedades. Se han elaborado dos tipos de combinadores de fitness:

- *Combinador parametrizado*, que computa el fitness final como una suma ponderada de todas las propiedades.

$$\sum_{f \in F} f * k(f)$$

  $F =$ conjunto de propiedades, $k(f) =$ ponderación de la propiedad f

- *Combinador adaptativo parametrizado*. Al igual que el anterior, se computa como una suma ponderada de todas las propiedades. La diferencia es que el parámetro de ponderación es variable, de forma que la mejor propiedad del último movimiento elegido baja y el resto aumenta en dos factores que denominamos *attack* y *decay* respectivamente.

Con el combinador adaptativo parametrizado conseguimos una personalización más ajustada de la influencia de cada propiedad en el mapa. En el capítulo de experimentación (ENLACE A CAPI EXP) veremos como afecta a la generación.

5.3. Propiedades del mapa

En esta sección analizaremos las propiedades del mapa empleadas en el cómputo del fitness. Son totalmente independientes de la implementación de la interfaz de cómputo de fitness, ya que éstas solo establecen si se tiene en cuenta una o más propiedades.

5.3.1. Tamaño del camino principal

Para calcular el tamaño del camino principal dado un estado intermedio del sistema, emplearemos el estado del mapa sin tener en cuenta las habitaciones restantes. Como se comentó en (ENLACE A REPRESENTACION/uppermatrix), se mantiene una matriz de conexiones entre habitaciones con una estimación de la distancia entre las habitaciones conectadas. Actualizaremos la matriz cada vez que se coloque una habitación en el mapa.

Como se comentó en (ENLACE A FW ANTES), mediante esta matriz, podemos aplicar el algoritmo de *Floyd-Warshall* (apendice FW) para calcular la distancia mínima entre todos los pares de nodos. Elegiremos el par de nodos cuya distancia mínima sea máxima como habitaciones inicial y final, pudiendo computar así, una estimación del tamaño del camino principal.

5.3.2. Caminos no principales y bifurcaciones

En esta sección se analizan dos propiedades distintas en conjunto por estar relacionadas en la forma del cómputo de las mismas.

Si una habitación es un callejón sin salida, tendrá un solo enlace a la habitación que lleva hasta ella. Si se corresponde a un camino sin bifurca-

ción, tendrá dos enlaces: uno hacia la habitación de la que viene el jugador, y otro a la que se tiene que dirigir por necesidad, al no haber otro enlace. Diremos en este caso que constituye un camino *sin* bifurcación. Atendiendo a todo esto, se puede considerar que si una habitación tiene más de dos enlaces, existen $n - 1$ caminos a los que se puede dirigir un jugador, sin contar la habitación de la que proviene, y por ello, estimamos que esta habitación contribuye a que existan bifurcaciones en el mapa.

Para establecer una estimación de *cuanto* tiene de caminos no-principales, se ha hecho un conteo de las habitaciones que no constituyen el camino principal. Debido a que en principio todas las bifurcaciones del mapa que no sean camino principal influirían a la estimación de caminos no-principales, se omite el conteo para la propiedad de *caminos no principales* cuando dicha habitación influye en la propiedad de *bifurcaciones*

Para la estimación de ambas propiedades, se ha empleado el algoritmo de *FloodFill* que se puede ver en el apéndice (ENLACE A APENDICE FLOODFILL). De esta forma, recorremos todas las habitaciones sin repetir ninguna y realizaremos el conteo de habitaciones que influyen en cada propiedad.

5.4. Fitness caché

Experimentalmente, se ha comprobado que el *bottleneck* de nuestra aplicación reside en el cómputo del fitness en su totalidad. Si recordamos las especificaciones (ENLACE A QUE TIENE QUE SER PARA MOVIL), el sistema tiene que funcionar en sistemas móviles. Para resolver este requisito, se ha creado una interfaz que nos ayudará a modular el tiempo de ejecución, sacrificando calidad en cuanto a las propiedades calculadas, pero dando menores tiempos de ejecución.

Listado 5.5: Interfaz de selección de movimiento basada en búsqueda con mejora de caché

```
1 class BestSearchMovementSelector implements IMovementSelector {
2   IFitnessSolver fitnessSolver;
3   IFitnessCache fitnessCache;
4   Movimiento ElegirMovimiento( List<Movimiento> movimientos ) {
5     List<Float> fitnesses;
6     for( Movimiento m : movimientos ) {
7       Fitness f = fitnessCache.Get(m);
8       if( f != null ) {
```

```
9               fitnesses[m.index] = fitnessSolver.FuncionGuia( m );
10              fitnessCache.Cachear( m, fitnesses[m.index] );
11          } else {
12              fitnesses[m.index] = f;
13          }
14      }
15  }
16 }
```

En el listado 5.5 vemos las modificaciones realizadas para permitir el empleo de la caché. Ahora, antes de realizar el cómputo del fitness, comprobamos mediante la interfaz si está el cálculo de dicho movimiento está cacheado. En caso positivo, se usa el valor cacheado, y en caso negativo, se calcula el fitness y se guarda en la caché.

Como se ha mencionado antes, esto conlleva a una peor fiabilidad del valor asociado al movimiento, ya que no se tiene en cuenta actualizaciones del mapa, pero podemos crear cachés nuevas que cada cierto número de pasos haga un recómputo de todos los movimientos en caché. Veremos más posibilidades en el capítulo de trabajo futuro (ENLACE A MEJORAS DE FITNESS CACHE).

Así, se han realizado dos implementaciones de la caché:

- *Dummy*. No cachea nada. El sistema se comporta como si no hubiera caché.
- *Always*. Cachea siempre y devuelve el valor cacheado.

Como vemos, las dos opciones son totalmente opuestas, pero en el capítulo de trabajo futuro (ENLACE A MEJORAS DE FITNESS CACHÉ) estableceremos las pautas para crear cachés modulables con una implementación rápida.

Capítulo 6

Experimentación.

6.1. Interfaz de experimentación.

6.2. Flexibilidad y posibilidades.

6.3. Eficiencia

Capítulo 7

Trabajo futuro.

7.1. Soporte para capas.

Es arte, no interfiere en el sistema, pero viene bien. Dependiente del soporte.

7.2. Mapa de tamaño autoajustable

7.3. Otros fitness

7.4. Portar a móvil

Debido a los tiempos de ejecución y a la fitness caché, seguro que se puede.

7.5. Otros fitness caché

7.6. Backtracking con guardado de movimientos

Guardar no mejores, pero si muy buenos.

Bibliografía

[1] LucasArts Wikipedia page

[2] Techopedia LOD page

[3] Voxel wikipedia

[4] Quadtree wikipedia

[5] Octree wikipedia

[6] Brush wikipedia

[7] Quake wikipedia

[8] RogueLike wikipedia

[9] List of most expensive video games to develop.

[10] Gamedev Job Roles at CreativeSkillSet

[11] Video Game Development Wikipedia page

[12] DemoScene wikipedia

[13] http://www.amazon.com/Texturing-Modeling-Third-Edition-Procedural/dp/1558608486

[14] http://libnoise.sourceforge.net/tutorials/tutorial3.html

[15] http://www-cs-students.stanford.edu/ amitp/game-programming/polygon-map-generation/

[16] http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap

[17] Explicación del algoritmo de generación de mazmorras empleado en TinyKeep.