Facultad de Ciencias

GENERACIÓN AUTOMÁTICA DE MAZMORRAS

(Procedural generation of dungeons)

Trabajo de fin de grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Juan Ramón Mendicute Arminio

Director: Domingo Gómez Pérez

Junio - 2018

Agradecimientos

Quiero dedicar este trabajo a mis padres y mi pareja, los cuales me han brindado siempre todo el apoyo y han sido mi principal razón para llegar hasta el final.

También a Francisco Calatayud, por ayudarme a plantear el diseño original del juego y motivarme con las ideas que nos surgían desarrollándolo juntos.

Y por último, a Domingo Gómez que me ha guiado en la dirección adecuada desde el principio, proponiendo las técnicas que podían valerme y proporcionándome toda la información que necesitaba.

A todos vosotros, gracias.

Juan R. Mendicute Arminio

GENERACIÓN AUTOMÁTICA DE MAZMORRAS

Resumen

palabras clave: Automática, Gramáticas Generativas, Mazmorras, Juegos

El proceso de creación de contenidos virtuales, tanto para videojuegos como simulaciones, requiere una enorme cantidad de recursos, tanto en personas como tiempo.
Por esa razón la industria de los videojuegos confía en los sistemas procedimentales para automatizar ésta tarea, esto es, generar contenidos usando algoritmos basados en reglas pre-establecidas por el desarrollador y modificando los parámetros hasta obtener el resultado que se desee en cada caso.
El objetivo de este proyecto es el de analizar los diferentes tipos de algoritmos existentes para generar mapas y misiones, con especial énfasis en las gramáticas generativas. Aplicaremos este método para crear un sistema que permita la generación de un entorno 2D para un videojuego de mazmorras, la cuál tendrá un considerable nivel de aleatoriedad en la forma del entorno y en los elementos dentro del mismo (puertas, llaves, monstruos, trampas, etc.).

(Procedural generation of dungeons)

Abstract

keywords: Procedural, Generative Grammars, Dungeons, Games

The process of creating virtual content, both for videogames and simulations, requires an enormous amount of resources, both in people and time.
That's why the videogame industry relies on procedural systems to automate this task, that is, to generate content using algorithms based on rules pre-established by the developer and modifying the parameters until the desired result is obtained in each case.
The objective of this project is to analyze the different types of existing algorithms to generate maps and missions, with special emphasis on generative grammar. We apply this method to create a system that allows the generation of a 2D environment for a dungeon videogame, which will have a considerable level of randomness in the shape of the environment and in the elements within it (doors, keys, monsters, traps, etc.).

Índice

1 Introducción

1.1. La generación de contenidos virtuales

La creación de entornos virtuales (texturas, formas, objetos...) es uno de los aspectos más importantes a la hora de recrear un juego o simulación virtual que le muestre cierto nivel de detalle y variedad al usuario.

Con el continuo aumento del contenido de los juegos generación tras generación y la necesidad de las compañías de ofrecer nuevos productos a usuarios que esperan mundos cada vez más amplios y variados, se hacen necesarios métodos de agilizar el desarrollo de éstos. Además, la aplicación de actualizaciones generando contenidos "ad hoc" requiere un desorbitado gasto en desarrolladores (artistas, programadores, diseñadores...) y horas de trabajo. Por ambas razones se requieren herramientas que, entre otros, permitan a los estudios pequeños competir con los desarrollos de las grandes compañías.

En los últimos años se han desarrollado nuevas formas de crear estos contenidos de forma algorítmica, es decir, de forma automatizada mediante algoritmos con reglas preestablecidas, que pueden ser de gran ayuda tanto por su eficiencia como por su capacidad de incentivar la creatividad del desarrollador de contenidos usando lo generado por el algoritmo como modelo o base. Este reciente desarrollo e implantación de los sistemas de generación automática es debido entre otros al aumento de potencia en los equipos informáticos, ya que muchas veces requieren de una gran cantidad de memoria y capacidad de procesamiento para llevar a cabo sus funciones. Esto no quiere decir que sean sistemas totalmente novedosos, ya que como se verá más adelante, llevan existiendo entre nosotros desde hace décadas.

La principal ventaja de la creación de contenidos automáticamente es, lógicamente, el ahorro en tiempos de desarrollo una vez se ha creado e implementado el sistema, dado que a partir de un mismo algoritmo, y a base de variar las entradas a este, podemos generar una cantidad prácticamente ilimitada de contenidos distintos, y también el ahorro en almacenamiento que obtenemos, pues se eliminan mapeados y figuras pre-configuradas que evitamos almacenar en los equipos.

Estos sistemas permiten, por ejemplo:

- Juegos con una re-jugabilidad enorme, variando cada partida, o incluso juegos cuyo contenido se desarrolle en tiempo real conforme se avanza en él. Son capaces incluso de la creación de entornos totalmente inabarcables, con el tamaño de «universos».
- Juegos que aumenten su dificultad según la habilidad del jugador o que aporten nuevas experiencias a este con una gran cantidad de situaciones distintas, aleatorias o al gusto del jugador de acuerdo a su forma de jugar.

Esta clase de algoritmos de generación automática de contenidos (procedural content generation, PCG), se ha explotado mayoritariamente en la generación de juegos de estrategia, RPGs y, como en nuestro caso, mazmorras, ya que permiten la interrelación entre distintos elementos: salas, trampas, retos, monstruos, etc.

El problema se encuentra, como decíamos, en el procesamiento de todo ese sistema, exigiendo equipos (procesadores, memoria, GPUs...) cada vez más potentes y eficientes, que permitan ejecutarlos con el mínimo consumo de recursos y a la mayor velocidad posible, de manera que no afecte a la jugabilidad.

Otro problema se encuentra en que los desarrolladores al usar sistemas de PCG tienen cierta dificultad para obtener el resultado que desean, ya que en muchos casos no comprenden o saben manejar el sistema de generación, pues tampoco entienden la mecánica detrás del mismo.

Dichos algoritmos pueden funcionar por ejemplo mediante autómatas celulares, gramáticas, métodos basados en agentes, etc. Para más información ver Adams y Dormans [2012].
En la mayoría, el control de los resultados se realiza mediante funciones (llamadas fitness) que evalúan una serie de parámetros. Un ejemplo sencillo seria calcular la longitud del camino al jefe mediante el algoritmo A*, pero salvo que el algoritmo PCG esté altamente restringido, lo que conlleva perder capacidad de variación y una mayor dificultad para controlar todos los parámetros, los resultados obtenidos pueden no ser los esperados.

Debido a esto, la mejor forma, o la más extendida para obtener buenos resultados, es realizando un trabajo a medio camino entre el creado por el algoritmo PCG y el propio desarrollador, ya sea:

- A mano a partir del propio resultado generado, sirviendo como punto de partida o inspiración al desarrollador como indica Togelius et al. [2011].
- Siendo el desarrollador un intermediario en métodos de generación evolutivos, seleccionando los valores que más se aproximan al deseado.
- Permitiendo al desarrollador introducir sus variaciones en el propio algoritmo, de manera que pueda variar los resultados obtenidos para acercarse a su objetivo.

1.2. La generación automática de contenidos en los videojuegos

Los juegos con contenidos generados automáticos llevan alrededor nuestro desde hace muchos años. Algunos de los primeros y más importantes fueron DND de Gary Whisenhunt y Ray Wood [1974] y Dungeon de Don Daglow [1975], los dos basados en el universo de Dungeons And Dragons, Beneath Apple Manor de Don Worth [1978] y Rogue de Michael Toy, Glenn Wichman y Ken Arnold [1980]. Todos ellos consistían en la exploración de una mazmorra.

Una mazmorra, aunque comúnmente se refiere a celdas subterráneas y oscuras en donde se encierra a los presos, en el mundo de los juegos (de mesa o virtuales) también comprende otros tipos de escenarios, como pueden ser cuevas o fortificaciones, en los cuales un aventurero o grupo de aventureros debe adentrarse con el fin de explorarla y alcanzar un objetivo: rescatar una princesa (o príncipe), derrotar un monstruo... o simplemente sobrevivir.

Por ello esta clase de escenario es perfecto para los algoritmos PCG, y de ahí que su uso haya sido muy utilizado en éstos, pues combina perfectamente las necesidades de escenarios conectados entre sí, aparición de trampas, monstruos, elementos de ayuda como pociones, armas o hechizos, y un largo etc.

Ilustración 1: Pantalla del juego Dungeon (1975)

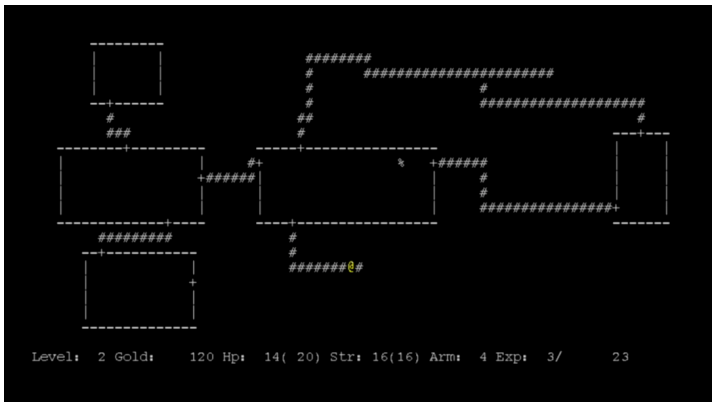

Ilustración 2: Pantalla del juego Rogue (1980)

El código fuente de estos juegos y sus estrategias de generación son fácilmente accesibles hoy día en la web (como los códigos publicados por Jon Lane). El planteamiento típico de estos algoritmos podría denominarse como de fuerza bruta, estando creados estrictamente para la generación de estructuras de juegos de mazmorras.

Una estrategia común en ellos, es la de generar un mapeado de celdas, «tiles», que es llenado con celdas representando rocas (paredes) y después «taladrar» los túneles y habitaciones partiendo de la entrada. Se pueden generar múltiples caminos taladrando en distintas direcciones desde varias localizaciones previas. Después, la mazmorra es poblada con criaturas, trampas y tesoros.
Otras estrategias se basan en crear grandes salas en el mapeado, determinar que salas se conectan mediante túneles, y poblarlas posteriormente.

A estos juegos comentados anteriormente, les siguieron otros muchos, como Moria de Robert Alan Koeneke y Jimmey Wayne Todd [1983], el cuál tuvo poca repercusión en ventas, pero fue y sigue siendo objeto de interés académico.

Otros posteriores, como The Elder Scrolls II: Daggerfall de Bethesda Softworks [1996], contenía según la desarrolladora, Bethesda, un mapeado 3D con arboles, casas, cuevas... de $229{,}848km^2$, o dicho de otro modo, un terreno más grande que Gran Bretaña generado de manera automatizada. Al mismo tiempo juegos como Diablo de Blizzard North [1996], utilizaron la generación automática no para generar la estructura de las mazmorras, sino como complemento para la generación de los elementos que contenían estas.

Más allá de la generación de los propios escenarios, estos métodos se han utilizado en los videojuegos para otros aspectos, como modificar el comportamiento de NPCs (non-player character) según el comportamiento del jugador (Left 4 Dead, Valve Software [2008]), permitiendo a la IA de los enemigos adaptarse a nuestra habilidad y dificultando a la vez el encontrar elementos de ayuda.
También es interesante el juego Borderlands, Gearbox Software [2009], del cual existen actualmente dos secuelas y cuyo PCG permite que existan armas y equipación de todo tipo, como metralletas de fuego, pistolas de ácido, o granadas de hielo que explotan en más granadas de hielo, a la vez que cada una tiene una estética y forma completamente diferente del resto de armas y objetos.

Por último, existen dos juegos que han captado la atención del público en la última década precisamente por el uso que hacen de los algoritmos procedimentales, Minecraft, Mojang [2011] y No Man's Sky, Hello Games [2016].

El primero, un mundo abierto creado enteramente de «cubos» de distintos materiales, tuvo una gran repercusión a nivel mundial. Tanto la generación de entornos, los cuales se generaban en cada nueva partida y podían tener una extensión descomunal, como la variedad de todo tipo de creaciones fueron una revolución. Como muestra, se podían encontrar «calculadoras» a tamaño gigante creadas con los elementos dentro del juego. Esto le abrió las puertas por ejemplo a escuelas de primaria como método de enseñanza.

No Man's Sky debe su auge debido casi exclusivamente a su algoritmo de generación procedimental. El juego, basado en la exploración espacial y catalogación de planetas, animales y plantas, se nutría de un sistema PCG para conseguir crear un completo universo, con dieciocho quintillones de planetas, con sus estrellas, ambientes y especies distintas en todos ellos.
La expectación que genero fue inmensa, sin embargo, debido a los fallos que mostraba el juego en sus inicios, mostrando animales con formas absurdas y entornos poco logrados en otros casos (además de promesas incumplidas en cuanto a jugabilidad), sepultaron su éxito al poco tiempo.

Ilustración 3: Captura de No Man's Sky (2016)

1.3. Objetivo del proyecto

El objetivo de este proyecto consiste en el desarrollo de un sistema automatizado que genere distintos niveles aleatorios para un videojuego de mazmorras y en analizar su efectividad y versatilidad. El sistema creará una mazmorra en dos dimensiones (2D) distinta en cada caso, que constará de puntos clave, cerraduras, objetos, trampas, etc.

La estructura base del proyecto consiste en crear una «matriz» de tamaño variable, sobre la que se superpondrán las distintas partes de escenario pre-configuradas, creando «celdas» interconectadas entre sí de manera correcta.

Las partes que se interconectarán entre sí, tendrán las siguientes formas posibles:

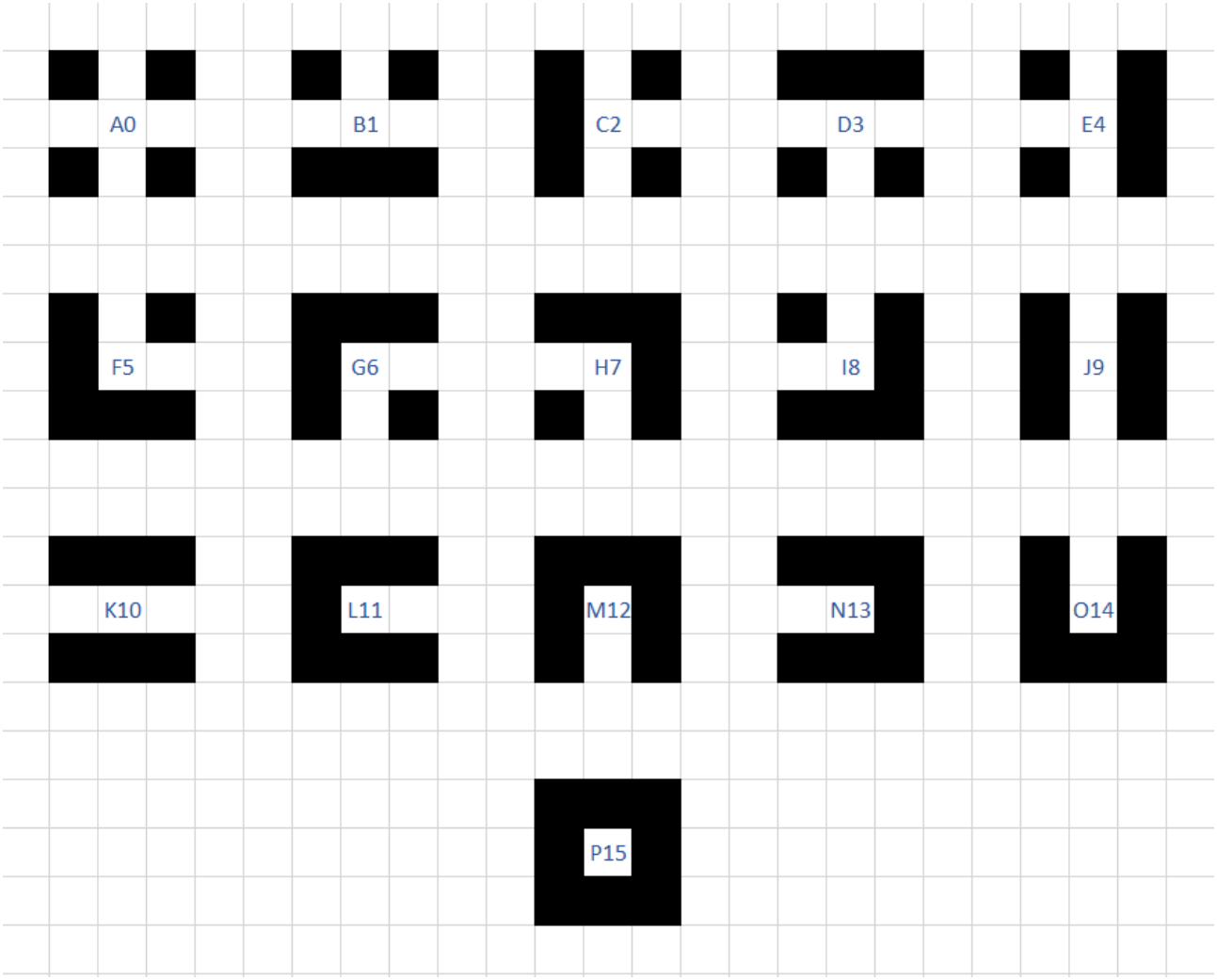

Ilustración 4: Tipos de segmentos que pueden conformar la mazmorra

La numeración mostrada en la Ilustración 4 será la empleada en el código del programa, de manera que la celda cero sea la celda con mayor numero de caminos/salidas, cuatro, y la celda quince no dispondrá de salidas.

El personaje del jugador, comenzará en la posición central de la matriz, y deberá alcanzar el final de la mazmorra, donde encontrara un jefe final y un tesoro como recompensa. Por el camino encontrara «bloqueos» que impidan su paso, y deberá encontrar las llaves u objetos que le permitan avanzar. Además podrá encontrar objetos como pociones que le ayuden a luchar con el resto de posibles enemigos y a cruzar las trampas que encuentre. La mazmorra debe ser completamente jugable y ningún objeto (como llaves o pistas) se creará sin sentido, teniendo cada uno un cometido.

Esto significa que, entre otros:

- Las cerraduras o puertas que se creen como «bloqueos» deben cumplir con su propósito, y no será posible acceder a un área que esté cerrada mediante otras vías.
- Las llaves o mecanismos que abran dichos bloqueos deben encontrarse en la parte transitable del entorno (y no tras el bloqueo), de manera que puedan ser utilizadas con su correspondiente cerradura/puerta.
- Las trampas o monstruos deben situarse de manera que puedan cumplir con el objetivo de estorbar/matar al jugador y no estar, por ejemplo, en una esquina que no interfiera con el transcurso del nivel, salvo que el área en que se encuentra sea optativa.
- Los monstruos importantes, los jefes, deben ser enfrentados obligatoriamente, de manera que los caminos que llevan al final del nivel pasen a través de ellos.

1.4. Entorno de desarrollo

A la hora de desarrollar el proyecto el primer punto consistía en elegir el motor gráfico con el que se llevaría a cabo.

Este motor debido a presupuestos debía ser gratuito y disponer de la suficiente versatilidad como para configurar todo el sistema que teníamos en mente. Por ello, tras analizar los distintos motores disponibles (Unity, CryEngine, Unreal...), se optó por la utilización del motor Godot Engine.

Se trata de un motor 2D y 3D que no solo es open source, sino que es completamente gratuito tanto para uso personal como comercial.

Toda su arquitectura y sistema de desarrollo se basa en lo que denominan «escenas». Las escenas pueden ser un escenario propiamente dicho, un objeto, un monstruo... Estas tienen dentro de sí objetos denominados nodos, los cuales pueden ser imágenes, código, colisiones, sonidos, etc. En otras palabras, dentro de una escena se encuentran todos sus componentes.

Hasta aquí nada extraño. La parte importante es que estas escenas una vez creadas pueden instanciarse y utilizarse dentro de otras escenas al igual que cualquier otro «nodo», por lo que podría decirse que tenemos un motor «orientado a objetos» u «orientado a escenas».
Dado que nuestro proyecto se basa en instanciar distintos fragmentos de escenario y objetos para configurar un escenario mayor, la mazmorra, este tipo de motor se veía, y es, perfecto para esta labor.

Por otro lado, el lenguaje de programación de Godot Engine, GDScript, es un lenguaje de programación de alto nivel y dinámicamente tipado.
Utiliza una sintaxis similar a Python (los bloques de código se basan en «indentación» y muchas palabras clave son similares). Su objetivo es estar optimizado y fuertemente integrado a Godot Engine, permitiendo gran flexibilidad para la creación e integración de contenido.
Este lenguaje permite una gran versatilidad y comodidad a la hora de crear nuestro código, y siendo similar a Python, partimos del conocimiento que ya tenemos de este para agilizar el desarrollo.

2 Las gramáticas formales

Una gramática formal es un conjunto de reglas de formación que definen las cadenas de caracteres que conforman un determinado lenguaje, sea formal o natural.

Tal como describen Luis Migel Pardo Vasallo y Domingo Gómez Pérez [2010], para describir un lenguaje L, se podrían tomar dos caminos:

- Dar con una forma de generar todas las palabras del lenguaje.
- Dar con un algoritmo para demostrar que una palabra forma parte del lenguaje.

Lógicamente, la primera opción no es viable de ningún modo para un lenguaje mínimamente complejo. Por ejemplo, podríamos pensar en todas las palabras que se podrían formar en un lenguaje que solo conste de los simbolos «a» y «b»:

$$L_{ab} = a, abba, aaabbaaabbaaab, ...$$

Junto a esto, hay que tener en cuenta que reconocer si una «palabra» pertenece a un lenguaje no implica saber generar elementos del mismo. Con ello se entiende que la única opción viable para describir un lenguaje es definir un algoritmo que contenga las reglas del mismo, una gramática formal.

Una gramática formal es una cuaterna G= (V,Σ,Q_0,P), donde:

- V es un conjunto finito llamado alfabeto de símbolos no terminales o variables.
- Σ es otro conjunto finito, que verifica V$\cap\Sigma = \emptyset$ (conjunto vacío) y es llamado alfabeto de símbolos terminales.
- $Q_0 \in$ V es una variable especial, denominada símbolo inicial.
- P $\subseteq$ (V$\cup\Sigma$)*$\times$(V$\cup\Sigma$)* es un conjunto finito llamado conjunto de producciones.

Para poder definir la dinámica asociada a la gramática es necesario asociarle un sistema de transición. Este sistema asociado ($S_G \rightarrow$G) dispone de las propiedades siguientes:

El espacio de configuraciones viene dado por:

$$S_G := (\text{V}\cup\Sigma)^*.$$

Dadas dos configuraciones $S_1, S_2 \in S_G$, decimos que $S_1 \rightarrow_G S_2$ si se verifica la siguiente propiedad:

$$\exists x, y, \alpha, \beta \in S_G = (V\cup\Sigma)^*, \text{ tales que}$$

$$S_1 := x \cdot \alpha \cdot y, \qquad S_2 := x \cdot \beta \cdot y, \qquad (\alpha, \beta) \in \text{P}.$$

Dada una gramática G= (V,Σ,Q_0,P) y su espacio de configuraciones S_G se define el lenguaje generado por una gramática como el conjunto de configuraciones S_1 tales que:

$$Q_0 \rightarrow_G S_1 \qquad \text{y} \qquad S_1 \in \Sigma^*$$

Tomemos como ejemplo la gramática G= (V,Σ,Q_0,P), donde

$$\text{V:=}Q_0 \qquad \Sigma \text{ :=a,b} \qquad \text{P:=}(Q_0,\text{a}Q_0),(Q_0,\lambda)$$

El sistema de transición tiene por configuraciones S:=Q_0,a,b* .
Un ejemplo de una computación sería:

$$aaQ_0bb \rightarrow aaaQ_0bb \rightarrow aaaaQ_0bb \rightarrow aaaa\lambda bb = aaaabb.$$

Nótese que las dos primeras veces se ha usado la regla de re-escritura (Q_0,aQ_0) y la última vez se ha usado (Q_0, λ).
Esta misma gramática permitiría estudiar también el lenguaje L={a,aa,aaa,...}.
Para generar una palabra de este, tan solo seria necesario computar la variable Q_0 tantas veces como deseemos:

$$Q_0 \rightarrow aQ_0 \rightarrow ... \rightarrow a...aQ_0 \rightarrow a...a$$

Por comodidad, se suele usar la notación $A \rightarrow B$ en lugar de $(A, B) \in P$ para indicar una producción. En el caso de tener más de una producción que comience en el mismo símbolo, se suele usar $A \rightarrow B|C$, en lugar de escribir $A \rightarrow B, A \rightarrow C$.

Por ejemplo, considerando la gramática: G = (V , Σ, Q_0, P), donde

$$\text{V :=}\{Q_0\}, \qquad \Sigma\text{:=} \{a, b\}, \qquad P := \{Q_0 \rightarrow aAb, aA \rightarrow aaAb|\lambda\}.$$

Un ejemplo de una computación sería:

$$Q_0 \rightarrow aAb \rightarrow aaAb \rightarrow aab.$$

Hay que recalcar que un mismo lenguaje puede ser especificado por distintas gramáticas indistintamente. Por ejemplo, para el caso anterior se podría haber utilizado la gramática:

$$\text{V} := \{Q_0\}, \qquad \Sigma\text{:=}\{a, b\}, \qquad P := \{Q_0 \rightarrow b|aA, A \rightarrow aA|b\}.$$

Notación BNF y EBNF

La notación BNF (Backus Normal Form) es un metalenguaje utilizado comúnmente para especificar lenguajes de programación. En él, se introducen los siguientes cambios:

- Las variables X ∈ V se representan mediante X.
- Los símbolos terminales (Σ) no son modificados.
- El símbolo asociado a las producciones → es reemplazado por =.

De este modo, la gramática que genera el lenguaje:

$$\text{L} = \{\lambda, \text{ a, aa, aaa, ...}\}$$

se puede describir de la siguiente manera:

$$\text{V} = \{ \text{ Q } \} \qquad \Sigma = \{\text{a}\}$$

donde las producciones seran:

$$Q = aQ$$
$$Q = \lambda$$

EBNF añade funcionalidad a la notación BNF, permitiendo repeticiones o diferentes opciones.

Estas son las principales cambios con respecto a la notación BNF:

- Las variables X ∈ V no son modificadas.
- Los símbolos terminales (Σ) se representan entre comillas simples.
- El símbolo asociado a las producciones → es remplazado por : .
- Se introducen nuevos símbolos para representar repeticiones: * (ninguna, una o mas repeticiones) y + (una repetición al menos).
- ? indica que la expresión puede ocurrir o no.

Por ejemplo, el lenguaje anterior podría definirse tal que:

$$V = \{ Q \} \qquad \Sigma = \{'a'\} \qquad P=\{(Q : a^*)\}$$

Tipos de gramáticas formales

Dentro de las gramáticas formales existen cuatro tipos, con mayor o menor restricción a los lenguajes que pueden generar.

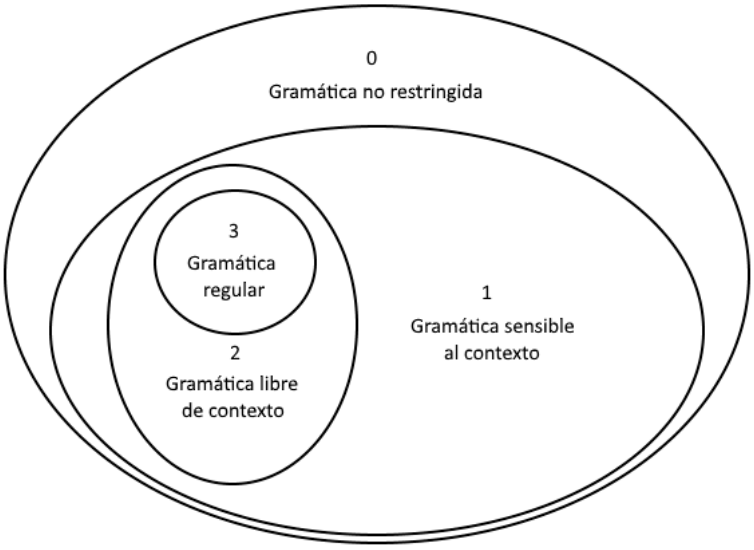

Ilustración 5: Esquema de tipos de lenguajes formales

Tipo 0 o gramáticas no restringidas:

Este tipo incluye todas las gramáticas formales cuyos lenguajes pueden ser reconocidos por una máquina de Turing.

Tipo 1 o gramáticas sensibles al contexto:

Las gramáticas sensibles al contexto son aquellas cuyas producciones toman la forma:

$$C_0 \rightarrow C_1$$

Donde C_0 y C_1 son cadenas de símbolos terminales y/o variables (excluyendo la cadena vacía λ), y donde la cadena C_0 debe ser de menor o igual longitud a la cadena C_1.

Se denominan sensibles al contexto porque tanto C_0 como C_1 determinan la forma que debe tener una cadena para poder generar las producciones.

Nota: Estas son las gramáticas que se utilizarán a lo largo de la memoria y que han sido utilizadas en el proyecto en sí.

Tipo 2 o gramáticas libres de contexto:

Las gramáticas libres de contexto son aquellas cuyas producciones son de la forma:

$$\mathrm{A} \rightarrow C_0$$

Siendo A una variable (o símbolo no terminal) y C_0 una cadena de símbolos terminales y/o variables. Como indica su nombre, las producciones de una gramática libre de contexto pueden generarse independientemente del contexto en que se apliquen, ya que la variable se puede tomar por si sola.

Tipo 3 o gramáticas regulares:

Una gramática regular es aquella cuyas producciones toman la forma:

$$\mathrm{A} \rightarrow C_0$$

Donde A es una variable, y C_0 es una cadena que contiene:

- Un símbolo terminal (distinto de λ) seguido de una variable o una variable seguida de un símbolo terminal (distinto de λ).
- Un símbolo terminal, incluido λ.

3 Métodos de generación automáticos

A continuación se explicarán los distintos métodos que se han analizado a la hora de elegir como desarrollar el sistema.

3.1. Sistemas basados en restricciones

Estos sistemas se basan en la aplicación de restricciones a todos y cada uno de los objetos prediseñados para el sistema de generación.

A partir del diseño de las reglas básicas, como el objetivo (matar al monstruo final), punto de inicio, tamaño máximo del mapa, etc., se aplican reglas a las distintas partes que conforman el «mapeado» (celdas en nuestro caso) para que cada una solo sea utilizable de la manera correcta (posición, forma, «vecinos» admitidos...).
Esto conlleva un gran número de condiciones a la hora de desarrollar el código, ya que se deben contemplar todas y cada una de las posibilidades previamente, e indicar paso por paso como debe procederse en esos casos en el propio código del programa.

Como punto a favor tiene su facilidad de desarrollo (que no rapidez), ya que básicamente consiste en codificar que partes son utilizables y como se relacionan temporalmente unas con otras, que objetos deben crearse previamente a otros, etc.
Como punto negativo tiene obviamente que cualquier variación importante que queramos hacer en el entorno acarrea el análisis de todas estas condiciones para cerciorar que no afecte a otras, y que debido a esta necesidad de tener «atadas» todas las posibilidades, también se delimita el número de variaciones que pueda alcanzar el entorno.

Ilustración 6: Ejemplo de mapa generado mediante restricciones E. Yeguas and R. Muñoz-Salinas and R. Medina Carnicer

3.2. Autómatas celulares

Los autómatas celulares consisten en una estructura, una malla de «células», en un numero limitado de «dimensiones» en las que se encuentran sus correspondientes vecinos. Las células contienen información, que se denomina estado de la célula o simplemente estado. Cada célula tiene referencias a sus vecinos próximos y un estado inicial.
Para nuestro objetivo por ejemplo, estos estados pueden ser «caminable» y «no caminable». Con cada iteración estos estados varían (o no) siguiendo un algoritmo que se ve afectado por el estado actual de la célula y el estado de sus vecinos. Tras sucesivas iteraciones, se crean formas en la malla, que darían lugar a «mazmorras».

Este tipo de sistema es muy útil para el desarrollo de niveles con aspecto de cuevas y también sirve para otras muchas tareas, como la simulación de propagación de bacterias tal y como demuestran Krawczyk et al. [2003].

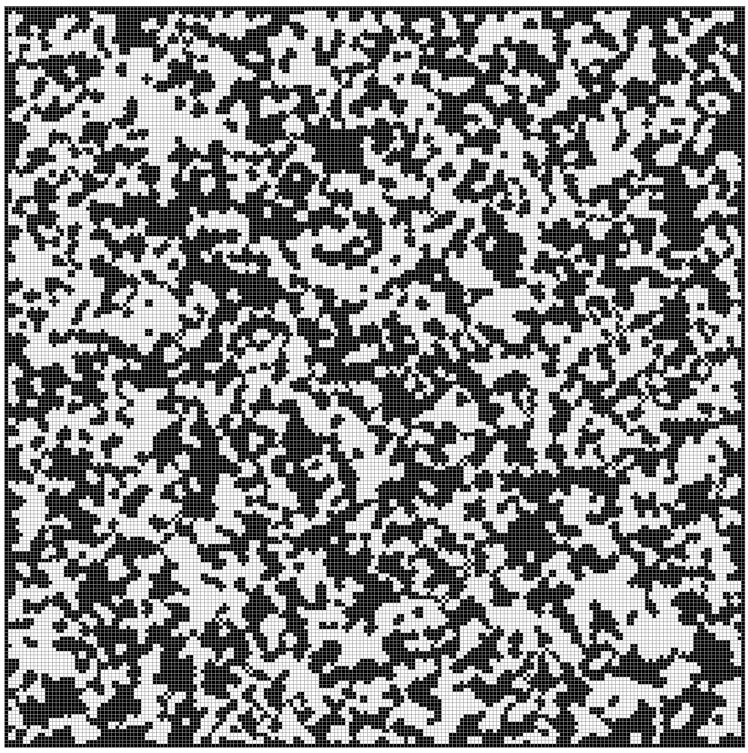

Ilustración 7: Ejemplo de mapa generado mediante autómatas celulares Jeremy Kun

Las ventajas de este método son múltiples: puede generar una variedad prácticamente infinita de escenarios, las reglas que aplican al cambio de estados pueden ser relativamente simples, y el aspecto «desorganizado» que se genera como resultado final es más realista.

Un aspecto negativo que es necesario remarcar es la dificultad para controlar los «caminos» que se generan, y que por tanto no se puede garantizar que el escenario siga la secuencia correcta que necesite el desarrollador para generar sus «misiones».

3.3. Algoritmos genéticos

Un algoritmo genético es un algoritmo evolutivo que emplea una función fitness y una representación genética para encontrar la solución al problema planteado. Esta representación genética codifica las soluciones en «cromosomas», y la función fitness se encarga de analizar la calidad de estas soluciones, que en nuestro caso, representarían mapas.

Para ello, a base de múltiples iteraciones, el algoritmo combina las mejores soluciones alcanzadas en cada caso entre sí, creando nuevas soluciones «hijas». En este continuo proceso de fusiones se pueden añadir variables que cambien aleatoriamente partes de los «cromosomas», creando así nuevas evoluciones artificiales. Con este procedimiento se espera que después de un cierto tiempo se alcance la solución optima o una solución suficientemente buena.

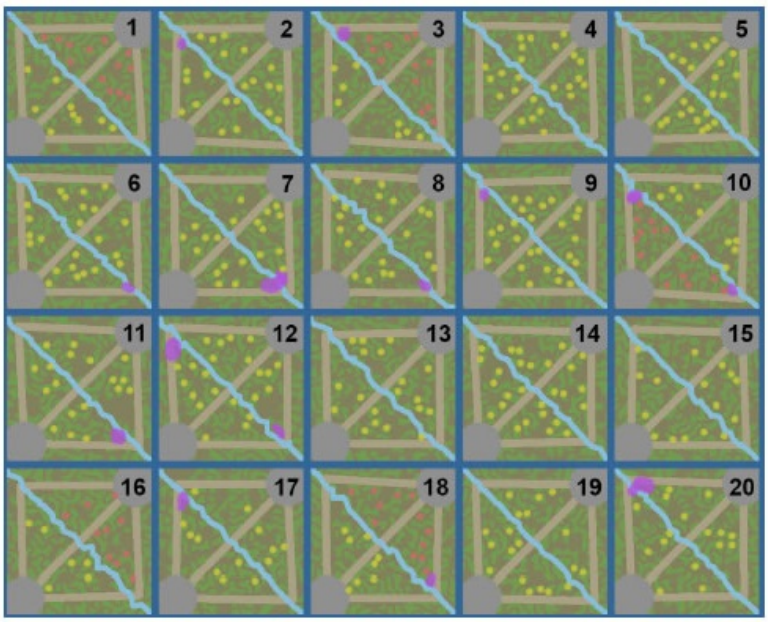

Ilustración 8: Ejemplo de mapa generado mediante algoritmos genéticos Cannizzo y Ramírez [2015]

Para generar mapas, se podría haber diseñado un algoritmo genético que tras crear una variedad de mapas completamente aleatorios y registrando los caminos entre ellos, realizase una mezcla entre los mismos, aplicando a su vez mutaciones (añadir o quitar caminos), hasta alcanzar un mapa cuya función fitness (mínimo de celdas conectadas y enemigos, por ejemplo), satisficiese las necesidades del desarrollo.

Debemos mencionar que aunque este tipo de algoritmo puede crear una variedad considerablemente enorme de mapas hasta encontrar una solución que pueda encajar con la historia que desee el desarrollador, puede conllevar una enorme cantidad de tiempo de ejecución, ya que necesita de la creación de múltiples mapas (aunque solo a nivel teórico), y aplicar un proceso evolutivo que técnicamente puede llegar a ser ilimitado.

3.4. Gramáticas generativas

Las gramáticas generativas parten de una gramática dada para, a partir de ésta, crear «palabras» que pueden representar diversos elementos. En nuestro caso, escenarios y misiones.
Como se ha explicado, su lógica parte de la misma que la propia lengua, que a través de unas reglas es capaz de crear todo un lenguaje, el cual suele pasar a tener un numero ilimitado de posibilidades.

Se parte de una base, un símbolo inicial o núcleo, que puede generar varias configuraciones hasta conformar una palabra final. Estas configuraciones se puede componer tanto de símbolos finales como de símbolos que pueden transformarse en otros, variables. Generalmente se distinguen unos de otros usando mayúsculas (variables) y minúsculas (finales). La palabra final sera aquella que se componga únicamente de símbolos finales.
En nuestro caso, estos símbolos finales representarían trampas, pociones, monstruos, y todos los elementos que deseemos incluir en la mazmorra.

Las reglas de la gramática serían las que, a partir de dicho núcleo, permitirían ir transformando todas las variables que aparezcan en símbolos finales. En este proceso se pueden añadir más símbolos a la palabra según lo indique la transformación que se realice. Es decir, no necesariamente son transformaciones 1:1.

Por ejemplo, si partimos de la palabra inicial S, disponemos de las transformaciones:

$$S \rightarrow bA|aB$$
$$A \rightarrow bAA|aS|a$$
$$B \rightarrow aBB|bS|b$$

con las que podríamos obtener:

S ->bA ->ba o S->aB ->aaBB ->aabSb ->aabbAb ->aabbab

Para nuestro objetivo, una vez disponemos de la gramática tan solo es necesario crear una nueva palabra para que, siguiendo las condiciones que el desarrollador decida, esta se transforme en distintas formas de mapa y objetos en el mismo, creando un mapeado completamente aleatorio.

El mayor problema que conllevan es el siguiente. Si por ejemplo tenemos un enemigo que es necesario eliminar para avanzar y los símbolos {n,l,i}, los cuales el sistema traduce como una celda de tres salidas, dos salidas en angulo y dos salidas enfrentadas respectivamente, se podría generar la palabra «nillin», y que genere una estructura como la siguiente:

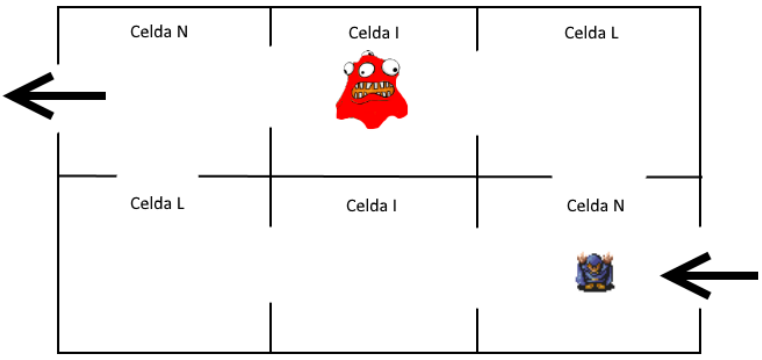

Ilustración 9: Ejemplo de error de generación

El enemigo podría ser completamente obviado, perdiendo con ello la «gracia» de esa zona.

La dificultad por tanto se encuentra en diseñar una gramática que además de correcta a nivel teórico, debe ser funcional, ya que dependiendo de los símbolos que existan, como se generen y como se configure la representación de cada uno de ellos, se podrían generar formas de mapa indeseadas. Como ventaja tiene que una vez creada y tras múltiples pruebas, las formaciones que se crean son considerablemente distintas unas de otras, es difícil que se creen errores en el mapeado, y si se desea, es fácilmente alterable para crear nuevas estructuras (aunque necesitará de las correspondientes pruebas para asegurar que los cambios no causan fallos).

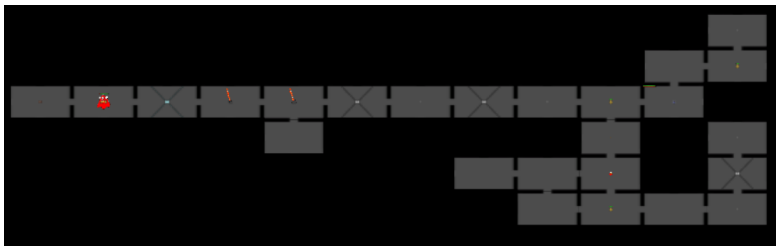

Ilustración 10: Ejemplo de mapa generado mediante gramáticas generativas

4 Desarrollo del algoritmo

Como primera aproximación al desarrollo del algoritmo se tomó el enfoque basado en restricciones.

Partiendo de la idea de crear una pequeña matriz de nueve por nueve celdas (posteriormente ampliado a sesenta y una por sesenta y una), la celda central tomaría la forma cero (ver 4), con cuatro salidas, una en cada pared de la celda. Desde esa celda se añadirían aleatoriamente celdas, con una forma de las quince posibles, a todos los espacios conectados.

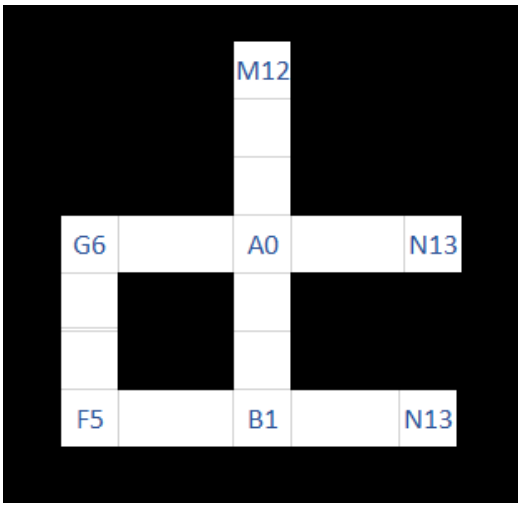

Ilustración 11: Ejemplo de mapa excesivamente pequeño

Como es lógico, este planteamiento inicial sufría de una importante falla, y es que no se garantizaba un tamaño mayor a cinco espacios en toda la mazmorra, un tamaño excesivamente pequeño y que carecia de jugabilidad.
Como solución a este problema y para que el recorrido del personaje jugable tuviese un tamaño mínimamente aceptable, se asignó una celda final que contendría al jefe del nivel y que se posicionaba aleatoriamente en una de las celdas en los extremos de la matriz.
Desde este punto en el extremo, se calculaba mediante el algoritmo A* un camino hasta la celda central de inicio del nivel, de manera que todas las celdas que se colocasen en ese camino tuviesen una conexión abierta entre ellas. Dado que el «peso» que se daba a las celdas en este algoritmo A* era aleatorio, permitía que el camino variase y no fuese un camino recto necesariamente.
A partir de este camino seguro hasta el jefe se colocarían en todas las vías abiertas celdas de manera aleatoria, y tras alcanzar cierto número de celdas añadidas, se cerraban los pasos que quedasen abiertos (de haberlos) añadiendo celdas con el mínimo de vías posible que encajase con sus celdas vecinas.

Para este control de qué celdas pueden ser o no pintadas para encajar con sus vecinas, se creo una función específica, *aquiQuePinto()* (ver apéndice A), a la que se le pasaban como parámetros:

- opciones: Un Array[15] de booleanos, el cual servía como representación de todas las posibles celdas «pintables». Desde la celda cero de cuatro salidas, a la celda quince, cerrada (ver 4).
- estoyCerrando: Un booleano cuya función es meramente indicar si se están cerrando los caminos o se están añadiendo celdas de manera normal.
- PosX y PosY: Dos valores que indicaban la posición en la que se está «pintando» la celda dentro de la matriz.

Con estos datos, y a partir de la matriz (que por comodidad en realidad consistía en un array unidismensional) que contiene todas las celdas llamada *mundo*, se especificó la función tal que:

- Si la posición X+1 (la derecha de la celda que estoy pintando) es menor a sesenta y uno (la anchura límite de la matriz [0..60]):
  - Si contiene una celda tipo {0,1,3,4,7,8,10,13} (celdas que tienen una salida por la izquierda, ver 4):
    - Cancelo los «tipos» en el array *opciones* que no tienen salida por su derecha, ya que no encajarían en caso de pintarse.
  - Si no, compruebo si la celda a la derecha está vacía. En caso de que no, significa que dicho espacio tiene una celda sin salida por su izquierda, por lo que cancelo las opciones que tienen una salida por su derecha {0,1,2,3,5,6,10,11}.
- Si la posición X+1 es igual a sesenta y uno (mayor al límite), se cancelan todas las opciones que disponen de camino a la derecha, ya que no puede haber caminos más allá.

De igual manera, se hacían las comprobaciones correspondientes a las celdas superior, inferior e izquierda de la celda actual, comprobando:

$$(\text{X-1} > 0),\ (\text{Y-1} > 0) \text{ y } (\text{Y+1} < 61)$$

y anulando las opciones disponibles según correspondiese.

Una vez tengo las opciones que encajarían en el mapeado, si estoy cerrando los caminos que aún estén abiertos, se toma la última celda del array *opciones* que esté marcada como válida (con valor *True*), ya que es la celda que encaja con sus vecinos con menor número de salidas (no añade caminos). Si no estoy cerrando caminos, entonces se escoge aleatoriamente una de las celdas que esté marcada como válida.

Dado que se decidió que la «matriz» se representara como un array unidimensional, para acceder a cualquier punto de la matriz se debía realizar un simple cálculo como:

$$\text{mundo}[(\text{PosY} * 61) + \text{PosX}]$$

Con la creación de la mazmorra mediante este método se conseguía una variedad de escenarios/-caminos bastante aceptable. Sin embargo llegados a este punto se planteaba un problema: Dado que el objetivo del juego consiste en alcanzar el jefe final a base de encontrar objetos, derrotar enemigos, etc., aún existía la posibilidad de alcanzar dicho jefe por un camino directo.

Como trabas a este avance directo se necesitaban poner puertas u obstáculos que impidiesen alcanzar la celda final de forma sencilla, por lo que se planteó que, ya que el jefe se encuentra en un extremo de la matriz y el personaje inicial en el centro, se podrían añadir a todas las celdas que rodean la celda central puertas u obstáculos que solo sean traspasables obteniendo un objeto dentro del área que abarcan, es decir:

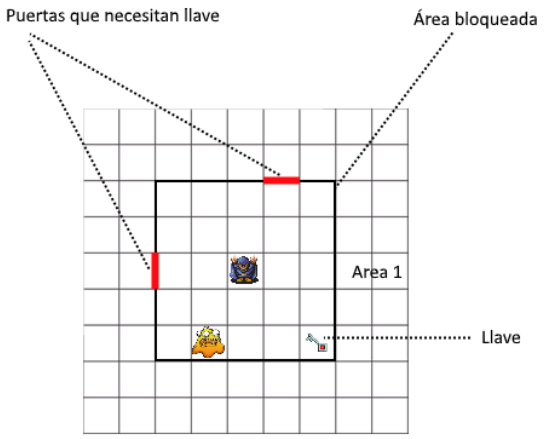

Ilustración 12: Primer área que bloquearía el avance

De igual forma, en la siguiente zona, se encontraría el objeto que permitiese avanzar hasta el jefe.

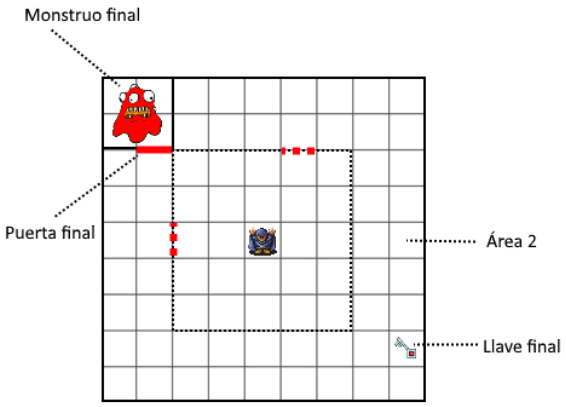

Ilustración 13: Segundo área que bloquearía el avance

Ésta manera de limitar la velocidad de avance del jugador, aunque efectiva, parecía un poco simple y causaba que los niveles tuviesen una secuencia similar:

- Encuentra la primera llave, que sabrás que se encuentra en alguna de las veinticinco celdas adyacentes (o las que correspondan según el ancho del mapa).
- Avanza de zona.
- Sabrás que la siguiente llave se encuentra en el nuevo área desbloqueada.

Viendo esta linealidad en el avance del juego, se replanteó el enfoque y diseño de los niveles por completo, y se decidió añadir a esta mecánica por restricciones la creación mediante gramáticas generativas.

5 Introduciendo gramáticas generativas

Siguiendo este nuevo enfoque, decidimos partir de las reglas mostradas en Dormans [2010]:

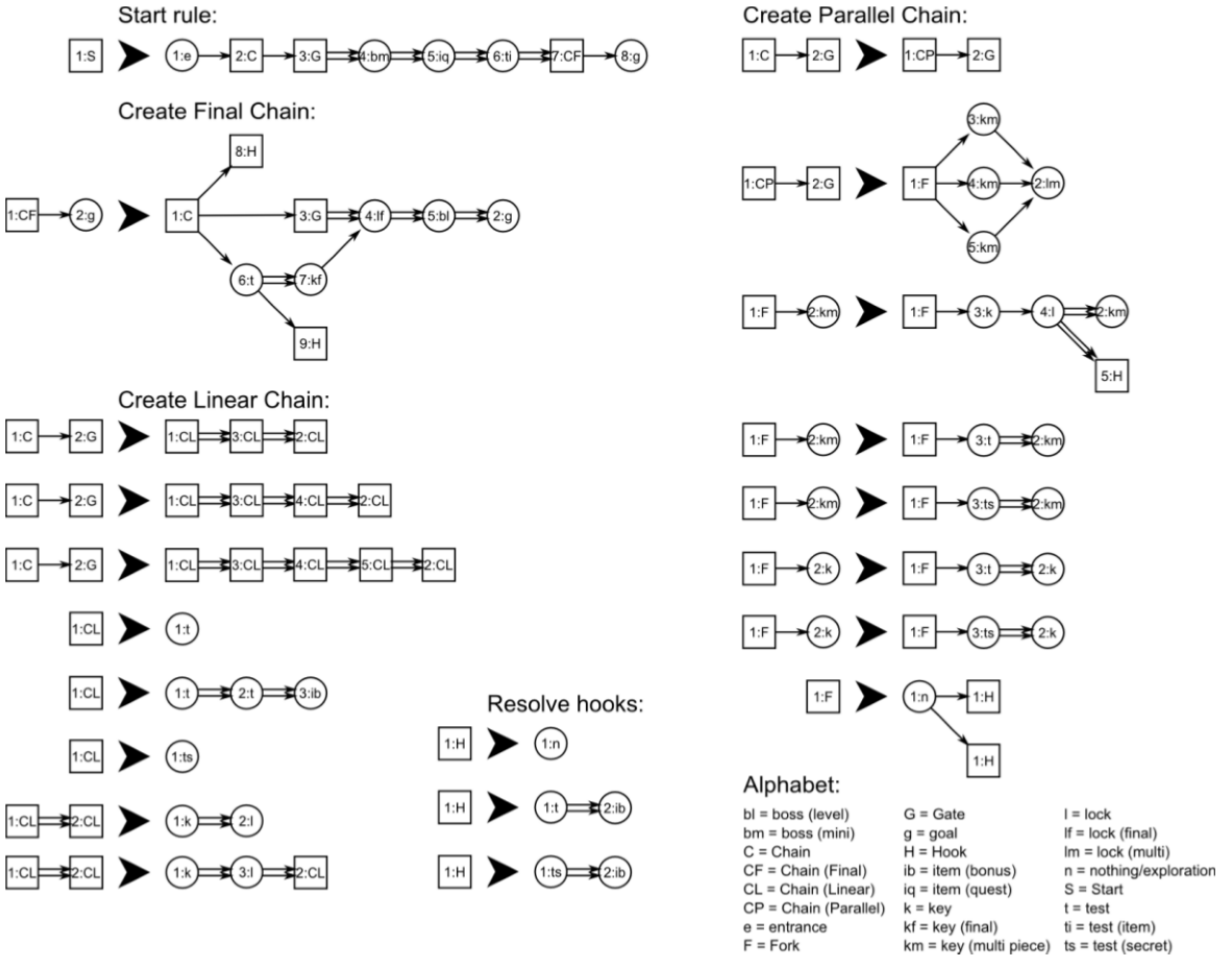

Ilustración 14: Gramática obtenida de Dormans [2010], página 4

Aunque básica, ésta gramática permite construir una gran variedad de mazmorras sin limitar las formas, la cantidad de objetos, ni prever el orden que seguirían las celdas.

Dado que la forma que la mazmorra puede tomar con esta gramática es aleatoria y por tanto de tamaño indefinido, para evitar problemas a la hora de representar la mazmorra se optó como comentamos anteriormente por ampliar la matriz hasta un tamaño de sesenta y un celdas.
El «mapeado» podría igualmente tomar una forma que superase los limites de la matriz, pero es una posibilidad remota, y en todos los casos ejecutados, más de cien, no ha tenido lugar.

Para aplicar la gramática y poder realizar modificaciones sobre la misma en cualquier momento, se crearon documentos de texto planos, uno por cada regla descrita, con una estructura similar a la utilizada en grafos DOT, teniendo por ejemplo para la regla inicial S:

```
digraph S {
    1 [shape= "circle" , label= "e" ];
    2 [shape= "square" , label= "C" ];
    3 [shape= "square" , label= "G" ];
    4 [shape= "circle" , label= "bm" ];
    5 [shape= "circle" , label= "iq" ];
    6 [shape= "circle" , label= "ti" ];
    7 [shape= "square" , label= "CF" ];
    8 [shape= "circle" , label= "g" ];
    -
    1 -> 2 ;
    2 -> 3 ;
    3 => 4 ;
    4 => 5 ;
    5 => 6 ;
    6 => 7 ;
    7 -> 8 ;
}
```

Como se puede ver a partir de la letra S se crean ocho nodos (siete nuevos y uno que modifica al mismo nodo S), algunos finales (marcados con «circle») y otros variables (marcados con «square»). A cada nodo se le asigna un nombre y posteriormente se indican los enlaces entre los distintos nodos. Aquellos nodos cuyos enlaces se marcan con una flecha simple(->), indican que deben tener conexión, pero no necesariamente adyacentes uno al otro, en cambio las flechas dobles (=>), implican que esos nodos deben estar adyacentes de manera obligatoria una vez representados en el mapeado.

Una vez definido el formato en que se pasan las reglas de la gramática, es necesario crear un «lexer» en el código del juego que permita leer las mismas.
Para esta tarea, es necesario saber entre otras cosas, cuantas reglas de traducción existen de cada nodo, y qué nodos son variables. Por ello se creó junto a estos documentos un «index» que mostrará que tipos de reglas existen, por ejemplo:

```
CF–g
C–G
CL
CL–CL
CP–G
F
F–k
F–km
H
S
EOF #este último sirve únicamente como indicador de fin de documento
```

A su vez se añadió otro documento, «traducibles», que numera los tipos de nodo que son variables:

CF
C
CL
CP
F
H
S
EOF

De este modo, el «lexer» puede recorrer la palabra que se vaya formando, y en caso de localizar un nodo variable, como el nodo CL, puede conocer que existen reglas para traducirlo por si solo o junto a otro nodo CL.

El código que realiza esta función, *leeTiposDeFichero()* (ver apéndice A), primero lee cuantos tipos de reglas existen en el index linea a linea, y las añade a un diccionario. Una vez tengo este dato, por cada *key* añadida al diccionario se comprueba si existe la regla uno, la dos, etc.
Por ejemplo, para la regla CL, se disponía de la CL1, CL2 y CL3.
Se cuentan cuantas reglas para una misma traducción existen comprobando si la regla *Regla+cuenta* (siendo *cuenta* la interacción correspondiente) existe en la carpeta que contiene los ficheros de reglas. Si es el caso, se suma uno al número de reglas que existen para esa traducción, si no, se pasa a la siguiente *key*.
Tras ello mediante la función *leeFicherosTraducibles()* (ver apéndice A), se lee el fichero «traducibles» linea a linea para ver que nodos tienen traducción, y se añaden estos nodos a un array.

Con todos estos datos, solo falta crear el grafo desde el cuál se construirá el mapa.
Para ello se utilizan dos clases (ver apéndice A): Grafo y Nodo.

La clase Nodo dispone de las siguientes variables:

- var padres = [] : guarda los nodos padre de este nodo.
- var hijos = [] : guarda los nodos hijo de este nodo.
- var nombre = 0 : nombre del nodo, único para cada nodo, no se repite.
- var final = true : indicador de si el nodo es o no final.
- var tipo = ' ' : tipo de nodo, pueden existir varios nodos con mismo tipo.
- var hijoDoble = false : indica si es un hijo directo, obligatoriamente adyacente a su padre.
- var padreDoble = null : indica su nodo padre directo (obligatoriamente adyacente al hijo), de haberlo.
- var pintado = false : indica si el nodo ya se ha añadido al grafo.

La clase Grafo por su parte, dispone de las siguientes variables:

- var Nodos = [] : Guarda todos los nodos que componen el grafo.
- var NodoOrigen = null : Guarda el nodo inicial desde el que parte el grafo.
- var index = 0 : Guarda el tamaño (número de nodos) del grafo.
- var todoFinales = false : Indica si todo el grafo se compone únicamente de nodos finales.

Ademas, esta clase dispone entre otras de la función *toString()*, que permite guardar el grafo en un fichero con el mismo formato de un fichero DOT.

El siguiente paso consiste en crear un método que permita sustituir los nodos aplicando alguna de las reglas. Este método se hizo de manera que fuese general para que se pudiesen crear nuevas traducciones y que se aplicasen sin necesidad de modificar el código.
El primer paso que realiza es comprobar si el nodo que queremos traducir se puede traducir en conjunto con uno de sus hijos.
Si es el caso se marca el tipo de traducción (por ejemplo, CL-CL), y si no, se elije una traducción de un solo nodo. Si cualquiera de los dos tipos de traducciones dispusiera de más de una regla, se elije aleatoriamente cuál de ellas aplicar y comienza el proceso de traducción.

Este proceso primero crea un array en el que se almacenarán todos los nodos mientras se realiza la tarea. En él se introduce desde el principio el nodo inicial (NI) que se está traduciendo, y si es el caso, también al hijo correspondiente.
Después comienza el proceso de lectura de la regla. La primera linea de una regla se aplica al nodo inicial, y la siguiente a su hijo (si es una traducción de dos nodos simultáneamente). De este modo se marcan estos nodos (que ya existen en el grafo) como finales si corresponde, y con el nuevo tipo (kf, iq, etc.), también, según corresponda.
El resto de nodos se crean de cero y se les aplica la condición especificada por la regla (nodo final/variable, tipo...).

Una vez creados los nodos, se pasa a la segunda fase de la traducción, que consiste en añadir a cada nodo las conexiones con sus hijos y padres.
Este proceso es un poco complicado, ya que requiere eliminar las conexiones que tuviese el padre con su hijo, pues este último pasa a estar al final de las conexiones de la traducción y todos los nodos nuevos se añaden entre ambos. De igual modo hay que hacer lo propio con las conexiones al padre en el hijo.
Como tercer y último paso se añaden al grafo general todos los nodos de uno en uno.

Este proceso, aplicado a todos los nodos variables que conforman la palabra, genera al final un grafo cuyo fichero DOT tendrá un aspecto similar al siguiente:

```
digraph A{
        1 [shape = "circle", label = "e"];
        2 [shape = "circle", label = "k"];
        3 [shape = "circle", label = "t"];
        4 [shape = "circle", label = "bm"];
        5 [shape = "circle", label = "iq"];
        6 [shape = "circle", label = "ti"];
        7 [shape = "circle", label = "k"];
        8 [shape = "circle", label = "g"];
        9 [shape = "circle", label = "k"];
        10 [shape = "circle", label = "ts"];
        11 [shape = "circle", label = "ts"];
        12 [shape = "circle", label = "lf"];
        13 [shape = "circle", label = "bl"];
        14 [shape = "circle", label = "t"];
        15 [shape = "circle", label = "kf"];
        16 [shape = "circle", label = "n"];
        17 [shape = "circle", label = "t"];
        18 [shape = "circle", label = "l"];
        19 [shape = "circle", label = "l"];
        20 [shape = "circle", label = "k"];
        21 [shape = "circle", label = "t"];
        22 [shape = "circle", label = "l"];
        23 [shape = "circle", label = "ib"];
        24 [shape = "circle", label = "l"];
        1 -> 2;
        2 -> 18 [penwidth=3];
        3 -> 4 [penwidth=3];
        4 -> 5 [penwidth=3];
        5 -> 6 [penwidth=3];
        6 -> 7 [penwidth=3];
        7 -> 16;
        7 -> 14;
        7 -> 19 [penwidth=3];
        9 -> 22 [penwidth=3];
        10 -> 3 [penwidth=3];
        11 -> 12 [penwidth=3];
        12 -> 13 [penwidth=3];
        13 -> 8 [penwidth=3];
        14 -> 15 [penwidth=3];
        14 -> 17;
        15 -> 12;
        17 -> 23 [penwidth=3];
        18 -> 9 [penwidth=3];
        19 -> 20 [penwidth=3];
        20 -> 24 [penwidth=3];
        21 -> 11 [penwidth=3];
        22 -> 10 [penwidth=3];
        24 -> 21 [penwidth=3];
```

Cuyo orden es =
1 2 18 9 22 10 3 4 5 6 7 19 20 24 21 11 12 13 8 14 15 17 23 16

Lo que genera la palabra: e k l k l ts t bm iq ti k l k l t ts lf bl g t kf t ib n ; y da como resultado el siguiente grafo:

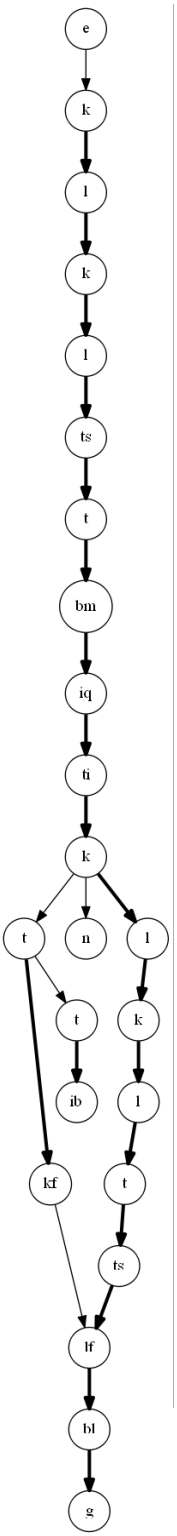

Ilustración 15: Grafo generado

6 Generación del mapa

Una vez el grafo es generado, el siguiente paso consiste en aplicarlo al juego.

Todo el código que se encarga de esta tarea consta de las siguientes variables:

- **var mundo = [3721]:** Define el array unidimensional que representa la forma completa del mundo dando un valor (equivalente a una «forma» de celda) a cada posición del mismo.
- **var pertenencia = [3721]:** Este array guarda la información de quién es el padre del nodo equivalente en el array mundo.
- **var tipoCelda = [3721]:** Este array guarda el tipo de celda (tesoro, jefe, etc.) de su equivalente en el array mundo.
- **var opciones = [15]:** Este array registra que opciones están disponibles para pintar en la celda que se esté creando en el momento.
- **var caminosX = [] y var caminosY = []:** Registran las posiciones X e Y de los caminos abiertos en el momento.
- **var grafo:** Guarda el grafo generado.
- **var multillaves = 0:** Ya que existen las llaves compuestas por varios fragmentos, indica cuantos fragmentos se han obtenido.
- **var endurecer = 0.1:** Valor por el que se empeora la dificultad del juego (más monstruos y/o trampas).

Disponiendo de estas variables, se crearon los siguientes métodos (ver apéndice A) que serían los encargados de mostrar en pantalla el mapeado del juego:

LimpiaOpciones() y BloqueaOpciones()

LimpiaOpciones(), simplemente reinicia el array de opciones, poniendo todas ellas a *True*.

BloqueaOpciones() por otro lado, se encarga de cerrar o abrir opciones dependiendo del tipo de celda que se le indique. Esta función existe para que el mapeado tenga una forma correcta, como por ejemplo los candados o bloqueos (l, lm o lf), que deben tener forma de pasillo para facilitar que el único camino viable al otro lado pase por ella, o el tipo n, el cual existe únicamente como una bifurcación de caminos, por lo que debe ser una celda de tres salidas:

ListaLosCaminos()

ListaLosCaminos() tiene la función de, una vez decidido que tipo de celda se desea pintar sobre el mapa de juego, registrar en los arrays *CaminosX* y *CaminosY* las nuevas vías disponibles a partir de esa celda hacia nuevas celdas vacías y a su vez, registrar quien es el padre desde el que se ha dado acceso a dichas celdas vacías.
Al registrar los nuevos caminos, se ha de tener en cuenta la forma de la celda para mirar las celdas vecinas y comprobar si ya «tiene un vecino» o si por el contrario al otro lado de dicho camino existe una celda vacía, en cuyo caso se anota el camino. Al anotar esta celda vacía se insertará en una posición aleatoria en los arrays de caminos abiertos, para que al disponer de ellos no se siga ningún patrón.

OrdenaNodos() y anadeHijo()

La función ordenaNodos() se encarga de cerciorarse que los nodos son pintados en el mejor orden posible.
Esto es así debido a la necesidad de poner adyacentes a algunos padres e hijos para evitar fallos en la jugabilidad.
Por ejemplo, los nodos iq (item quest) son objetos propios de la misión (¿encontrar una carta con pistas?). Para que no puedas alcanzarlos sin más, es necesario que en la celda previa se encuentre un obstáculo, como un nodo bm (boss mini), al cual debes enfrentarte y derrotar.
Por ello, estas relaciones se marcan en el grafo con flechas dobles, y en el código guardamos los valores *hijoDoble* y *PadreDoble* en los nodos de manera que quede registrada esta necesidad.

Cuando la función detecta estas relaciones, altera el orden de pintado para que dichas celdas se pinten consecutivamente de la siguiente manera:

- Se crea un nuevo array.
- A partir del nodo original, el de inicio, se comprueban los hijos de este. Si cualquiera de ellos es *hijoDoble*, no ha sido añadido ya al nuevo array y el nodo origen es su *padreDoble* (un nodo puede tener dos «padres» y ser *hijoDoble* solo de uno...), lo añado al array.
- Para añadirlo, llamo a la función *anadeHijo()*, la cual es una función recursiva:
  - Añado el nodo y sigo mirando los hijos de este. Si cualquiera de ellos es *hijoDoble*, realizo la misma tarea: se añade llamando a *anadeHijo()* y se miran los hijos de ese nodo.
  - Cuando se llega a un nodo sin más «hijosDobles», se añaden los otros hijos de ese último nodo (los cuales pueden contener «hijosDobles» también), y se regresa al nodo anterior, hasta que no quede un solo hijo por añadir.

SetNewArray()

SetNewArray() es la base del algoritmo de generación.
Primeramente se re-ordenan los nodos mediante el método *ordenaNodos()*. Después partimos de la celda inicial en el centro de la matriz (que recordemos es un array unidimensional realmente) a la que se le asigna el nodo inicial con una celda con la forma cero (ver ilustración 4) y se le da pertenencia «-1», no tiene padre. Además se añaden los caminos abiertos a través de *listaLosCaminos()*, y se marca ese nodo inicial como «pintado».

El siguiente paso es, por cada nodo:

- Compruebo si se ha recreado ya. Si no esta recreado continuo.
- Compruebo si es tipo «n». Esta comprobación es necesaria ya que el tipo «n» como se ha comentado, es un caso que crea bifurcaciones de caminos y su función consiste en pintar simultáneamente tres celdas de tres caminos:
  - Si no es *hijoDoble*, por cada una de las 3 celdas que debo pintar tomo por orden los caminos disponibles.
  - Se bloquean las opciones de celda que no se correspondan con una celda de 3 puertas con *bloqueaOpciones()*.
  - Si la función *aquiQuePinto()* da un resultado válido, elimino ese camino de la lista de caminos abiertos y añado al array *mundo* el valor devuelto, así como el tipo de celda en el array *tipoCelda*. A su vez se listan los nuevos caminos resultantes de esta nueva celda añadida.
  - Si *aquiQuePinto()* no da un resultado válido (valor menos uno), se comprueban las posibilidades con el resto de caminos abiertos, uno por uno.
  - Si por el contrario, es *hijoDoble*, se realiza la misma tarea, pero a su vez se comprueba que la celda tomada de los caminos libres tenga marcada su pertenencia al padre de este nodo en el array *pertenencia*, de manera que se añada adyacente a su *padreDoble*.
  - Tras «pintar» las 3 celdas, se marca el nodo como recreado.
- Si no es tipo «n», compruebo si es *hijoDoble*. De no serlo se realiza el bloqueo que corresponda a su tipo y se le intenta añadir uno por uno a los caminos abiertos existentes. Si hay uno válido, se registra la celda en *mundo*, *tipoCelda* y se listan los nuevos caminos.
  En caso de ser *hijoDoble*, se realiza el mismo caso pero asegurando que el camino abierto, además de válido, pertenece al padre del nodo.

spawnNewObject()

La funcion *spawnNewObject()* como se deduce, se encarga de pintar sobre el mapa aquellos objetos distintos de las celdas, esto es, monstruos, llaves, cofres, trampas, etc.
Para esta tarea toma sencillamente el *tipo* de la celda que se está rellenando, el cual define el contenido. Por ejemplo, el tipo «kf» indica que esa celda contiene la llave final, la que da acceso al jefe del nivel. Con esto, se crea la instancia del objeto y lo posiciona a partir de las coordenadas de la propia celda.

Ready()

La función ready() se ejecuta al iniciar el juego, y es la que se encarga de ejecutar todo el sistema. Primero toma el grafo generado y reinicia todas las variables del sistema.
Se ejecuta *setNewArray()* hasta que retorne un resultado válido. En cuanto tiene el resultado correcto, comienza el proceso de «pintado».
Este proceso consisten en recorrer todo el array *mundo* en orden. Si la celda es nula o con valor menos uno, se rellena con la celda quince, sin puertas, lo que lo hace terreno no transitable.
Si tiene un valor distinto, se toma dicho valor y se instancia la celda correspondiente de entre las quince restantes. Seguidamente se instancian los objetos que contiene la celda llamando a *spawnNewObject()*.

7 Resultados y observaciones

Tras la configuración de todo el sistema, se pasó al proceso de creación de los distintos contenidos que conformarían la mazmorra y a darles una estética básica.
Así mismo se configuraron los controles y acciones de todos ellos, incluyendo control de personaje, movimiento de enemigos, ataques, movimiento de trampas, uso de pociones y efectos en la barra de vida, etc.

7.1. Pruebas

Con estos elementos creados se pasó a la fase de pruebas, en las que se pudo comprobar la eficiencia del algoritmo.
El proceso de generación de la gramática, tanto con las lecturas de las reglas como de conformación del grafo conllevan un tiempo despreciable, de entre setenta y cien mili-segundos.
El proceso de configuración de la matriz, conlleva un tiempo ligeramente inferior, de entre veinte y veinticinco mili-segundos. En determinados casos, debido a la imposibilidad de encajar una celda en el mapeado, por la forma que requiere o por no encajar junto a su padre, se fuerza el reinicio de la gramática y la configuración, en cuyo caso a este tiempo se le añaden los entre setenta y cien mili-segundos de estos procesos, pudiendo sumarse repetidas veces si se diera el caso, aunque no ha tenido lugar a lo largo de todas las pruebas realizadas.

Paso \ Nº de Prueba	P1	P2	P3	P4	P6	P5	P7	P8	P9	P10	
Momento de inicio (milisegundos)	2994	2646	3250	2207	44826	4847	2430	2326	2515	2361	
Lectura de reglas	3002	2653	3257	2216	44834	4853	2438	2334	2523	2368	
Lectura de variables	3005	2657	3259	2220	44837	4857	2441	2338	2525	2373	
Generación del grafo	3013	2663	3265	2226	44847	4865	2449	2345	2531	2379	
Inicio de configuración del mundo	3069	2713	3322	2285	44924	4944	2504	2400	2586	2440	
ERROR: Celda imposible de instanciar	-	-	-	-	ERROR: caso km	ERROR: caso ib	-	-	-	-	
Re-inicio de gramática	-	-	-	-	44952	4977	-	-	-	-	
Re-lectura de reglas	-	-	-	-	44958	4984	-	-	-	-	
Re-lectura de variables	-	-	-	-	44961	4986	-	-	-	-	
Nuevo grafo generado	-	-	-	-	44970	4992	-	-	-	-	
Re-inicio configuración del mundo	-	-	-	-	45052	5046	-	-	-	-	
Mundo configurado	3091	2736	3347	2310	45074	5065	2521	2423	2607	2464	
TOTAL	97	90	97	103	248	218	91	97	92	103	MEDIA = 123.6 milisegundos

Ilustración 16: Tiempos de generación de grafos y configuración de mapeados

El proceso que conlleva más tiempo es el de la propia recreación del mundo. Esto se debe en su mayor parte a que instanciar y colocar un objeto/celda requiere de una cantidad de recursos que aumenta junto con la propia cantidad de objetos y la calidad de sus texturas.
Aún con ello, el tiempo de recreación del escenario supone un tiempo medio de cuatro con tres segundos, lo cuál es perfectamente asumible para los tiempos de carga que suelen encontrarse en los juegos, generalmente superiores.

Paso \ Nº de Prueba	P1	P2	P3	P4	P6	P5	P7	P8	P9	P10	
Inicio de la recreación del mundo	3094	2740	3350	2315	45077	5068	2525	2427	2610	2468	
Mundo recreado	7328	7111	7709	6681	49507	9510	6873	6722	6936	7015	
TOTAL	4234	4371	4359	4366	4430	4442	4348	4295	4326	4547	MEDIA = 4371.8 milisegundos

Ilustración 17: Tiempos de recreación de mapeados

Dado que se ha configurado el sistema para reiniciar la gramática si detecta un mapeado erróneo o no representable, el proceso obtiene en más de un ochenta por ciento de los casos un resultado correcto.

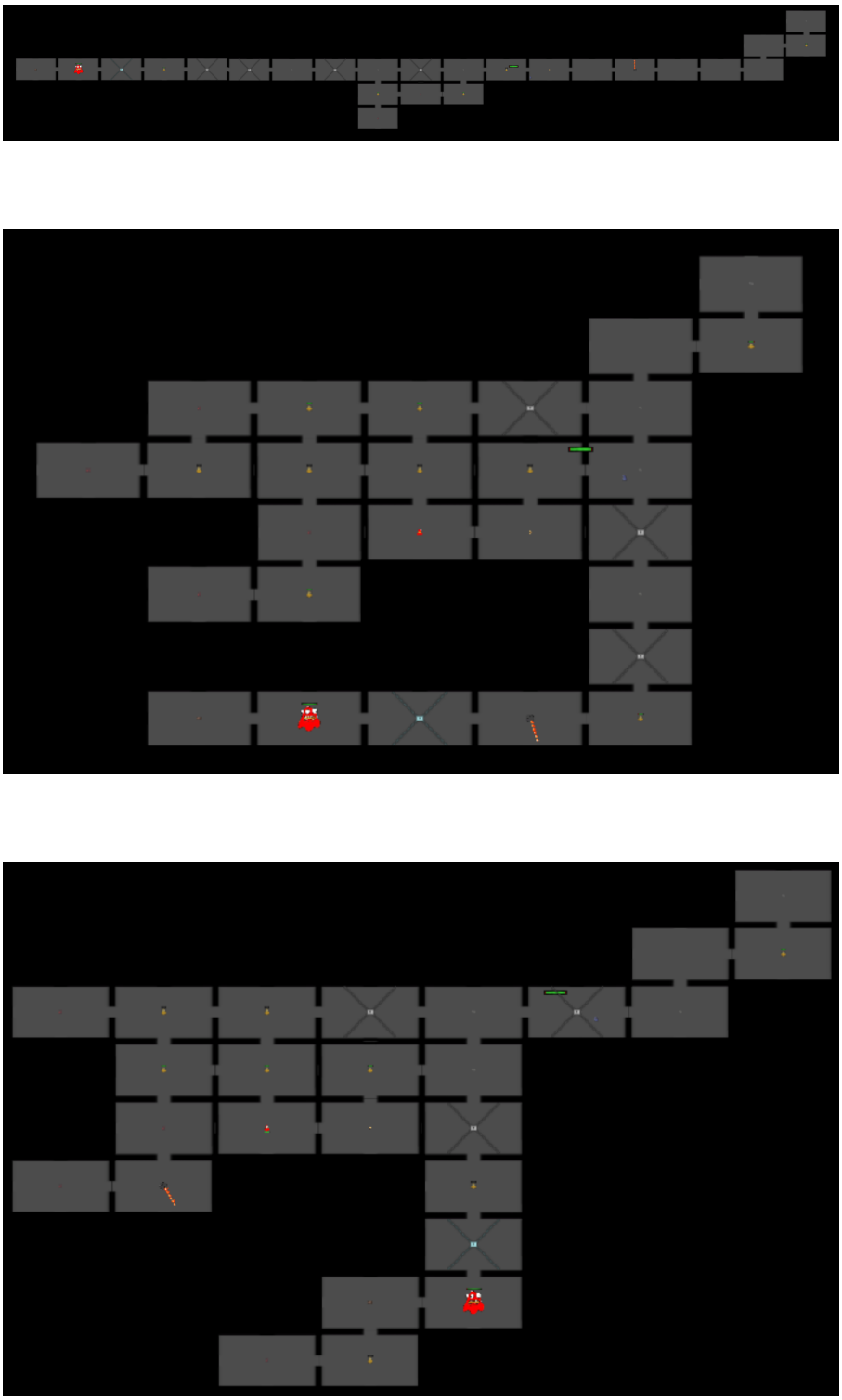

Ilustración 18: Ejemplos de generaciones

Sin embargo que se configure un mapeado correcto no significa un mapeado que se pueda considerar de «calidad».

Esto es así debido a que las reglas de la gramática pueden generar escenarios como pasillos en los que encuentras sucesivamente llaves y bloqueos, siendo un resultado monótono y sin emoción.
También podemos encontrar trampas o enemigos que son fácilmente sorteables por caminos adyacentes, lo cual le quita dificultad al mapeado.

Aún así, estos «problemas» no dejan de ser causados por las restricciones y condiciones que aplica el sistema a cada nodo, y no por la gramática o el sistema de recreación en si mismo, con lo que pueden ser solventables a base de mejorar las mismas.

7.2. Conclusiones

Tras la realización de múltiples pruebas, con pequeñas variaciones de las reglas introducidas inicialmente, se ha podido comprobar que el uso de gramáticas generativas para el desarrollo de entornos virtuales es perfectamente factible y válido.
Los niveles creados, aún considerando la simplicidad de las reglas, contienen la suficiente variedad de formas como para ser calificados de aleatorios.

Pese a que se han detectado leves fallos en la creación de algunos escenarios, estos mismos no son debidos a errores en la propia gramática y la aventura generada por la misma (funcional desde el punto de vista técnico), sino a la aplicación de ésta, que requiere de un análisis más detallado de las condiciones para que cada nodo pueda recrearse correctamente.
Esto es debido al uso de una matriz compuesta de «cajas» que comuniquen unas secciones con otras, que de por sí, ocasiona una serie de restricciones que dificultan reflejar lo definido por la gramática, y por tanto puede ocasionar errores al recrear el escenario.
Sin embargo, estos errores son completamente subsanables siempre que se dedique el tiempo necesario a la detección de los mismos y a sus causas, lo cuál, con un equipo y tiempo dedicado a ello, es completamente abordable.

Pese a ello, es importante recalcar que evolucionar la gramática es un proceso complejo que conlleva largos tiempos de ensayo y error, por lo que se debe considerar este aspecto a la hora de utilizar esta clase de sistemas frente a las alternativas (tanto las procedimentales como el desarrollo directo), valorando qué compensa más dependiendo del trabajo que se tenga entre manos.

8 Trabajos futuros

El presente trabajo no solo es concebido como un Trabajo Fin de Grado, sino que se ha desarrollado con el objetivo de continuarlo y alcanzar un estado perfectamente funcional que permita su lanzamiento de cara al público, tanto para estudiosos de la PCG a nivel técnico como para amantes de los juegos en general.

De este modo, a corto-medio plazo se realizarán una serie de mejoras necesarias, entre las cuales las más importantes, aunque no las únicas, están:

- Desarrollo exhaustivo de la gramática generativa, de manera que permita una mayor variedad de situaciones, objetos, pruebas y en gerenal, versatilidad a las misiones.
- Eliminación de estructuras semejantes en el código, unificando arrays (mundo, tipoCelda, pertenencia) creando nuevas clases y simplificando el acceso a los datos.
- Mejora del control en la recreación del escenario, añadiendo soluciones a los problemas de inserción, como puede ser añadiendo celdas «extensión» que permitan instanciar dichas celdas en áreas menos masificadas.
- Inclusión de nuevas estructuras, como salas compuestas por múltiples celdas, que permitan un mapeado más irregular.
- Generación de texturas y animaciones de los objetos/seres que pueblan el juego de mayor calidad, y con cierto nivel de variabilidad.

A medio y largo plazo, aun siendo campos distintos, el desarrollo de este sistema me ha incitado a ir un paso más allá que el del desarrollo de sistemas meramente procedimentales e interesarme por el desarrollo de la inteligencia artificial, particularmente en el campo de los videojuegos (aunque no cerrado a este campo únicamente), debido entre otros al inmenso interés tanto de la comunidad científica como de la industria en este área, y a ser según mi parecer, un campo con un potencial enorme y con mucho que descubrir por delante.

A Recuento de saltos

Método aquiQuePinto():

```
func aquiQuePinto(estoyCerrando, PosX, PosY, opciones):
        var retornar = −1

#Tenemos 5 casos, los normales y X=61, Y=61, X=0 e Y=0, los extremos.

#Desde la celda actual miro a la derecha, si la celda de la derecha esta abierta por la izquierda,
#cancelo las opciones sin puerta a la derecha.

        if(PosX+1)<61:
                if (mundo[(PosY * 61) + PosX + 1] == 0
                || mundo[(PosY * 61) + PosX + 1] == 1
                || mundo[(PosY * 61) + PosX + 1] == 3
                || mundo[(PosY * 61) + PosX + 1] == 4
                || mundo[(PosY * 61) + PosX + 1] == 7
                || mundo[(PosY * 61) + PosX + 1] == 8
                || mundo[(PosY * 61) + PosX + 1] == 10
                || mundo[(PosY * 61) + PosX + 1] == 13):
                        opciones[4] = false
                        opciones[7] = false
                        opciones[8] = false
                        opciones[9] = false
                        opciones[12] = false
                        opciones[13] = false
                        opciones[14] = false

#Si la celda de la derecha NO esta vacía, es que está cerrada por la izquierda y cancelo las
#opciones con puerta a la derecha.
                elif(mundo[(PosY * 61) + PosX + 1] != null):
                        opciones[0] = false
                        opciones[1] = false
                        opciones[2] = false
                        opciones[3] = false
                        opciones[5] = false
                        opciones[6] = false
                        opciones[10] = false
                        opciones[11] = false
        else:
                opciones[0] = false
                opciones[1] = false
                opciones[2] = false
                opciones[3] = false
                opciones[5] = false
                opciones[6] = false
                opciones[10] = false
                opciones[11] = false
```

```
#Miro por arriba.
#Si la celda de arriba esta abierta por abajo, cancelo opciones sin puerta hacia arriba.
        if(PosY−1)>0:
                if (mundo[((PosY−1) * 61) + PosX] == 0
                || mundo[((PosY − 1) * 61) + PosX] == 2
                || mundo[((PosY − 1) * 61) + PosX] == 3
                || mundo[((PosY − 1) * 61) + PosX] == 4
                || mundo[((PosY − 1) * 61) + PosX] == 6
                || mundo[((PosY − 1) * 61) + PosX] == 7
                || mundo[((PosY − 1) * 61) + PosX] == 9
                || mundo[((PosY − 1) * 61) + PosX] == 12):
                        opciones[3] = false
                        opciones[6] = false
                        opciones[7] = false
                        opciones[10] = false
                        opciones[11] = false
                        opciones[12] = false
                        opciones[13] = false

#Si la celda de arriba NO esta vacía, está cerrada por abajo y cancelo las opciones con puerta
#hacia arriba.
                elif (mundo[((PosY − 1) * 61) + PosX] != null):
                        opciones[0] = false
                        opciones[1] = false
                        opciones[2] = false
                        opciones[4] = false
                        opciones[5] = false
                        opciones[8] = false
                        opciones[9] = false
                        opciones[14] = false
        else:
                opciones[0] = false
                opciones[1] = false
                opciones[2] = false
                opciones[4] = false
                opciones[5] = false
                opciones[8] = false
                opciones[9] = false
                opciones[14] = false

#Miro por la izquierda.
#Si la celda de la izquierda esta abierta por la derecha, cancelo opciones sin puerta a la izquierda.
        if(PosX−1)>0:
                if (mundo[(PosY * 61) + PosX−1] == 0
                || mundo[(PosY * 61) + PosX − 1] == 1
                || mundo[(PosY * 61) + PosX − 1] == 2
                || mundo[(PosY * 61) + PosX − 1] == 3
                || mundo[(PosY * 61) + PosX − 1] == 5
                || mundo[(PosY * 61) + PosX − 1] == 6
                || mundo[(PosY * 61) + PosX − 1] == 10
                || mundo[(PosY * 61) + PosX − 1] == 11):
                        opciones[2] = false
                        opciones[5] = false
                        opciones[6] = false
                        opciones[9] = false
                        opciones[11] = false
                        opciones[12] = false
                        opciones[14] = false
```

```
#Si la celda de la izquierda NO esta vacía, está cerrada por la derecha y cancelo las opciones
#con puerta a la izquierda.
                elif (mundo[(PosY * 61) + PosX − 1]!=null):
                        opciones[0] = false
                        opciones[1] = false
                        opciones[3] = false
                        opciones[4] = false
                        opciones[7] = false
                        opciones[8] = false
                        opciones[10] = false
                        opciones[13] = false
        else:
                opciones[0] = false
                opciones[1] = false
                opciones[3] = false
                opciones[4] = false
                opciones[7] = false
                opciones[8] = false
                opciones[10] = false
                opciones[13] = false

#Miro por abajo.
#Si la celda de abajo esta abierta por arriba, cancelo opciones sin puerta hacia abajo.
        if(PosY+1)<61:
                if (mundo[((PosY+1) * 61) + PosX] == 0
                || mundo[((PosY + 1) * 61) + PosX] == 1
                || mundo[((PosY + 1) * 61) + PosX] == 2
                || mundo[((PosY + 1) * 61) + PosX] == 4
                || mundo[((PosY + 1) * 61) + PosX] == 5
                || mundo[((PosY + 1) * 61) + PosX] == 8
                || mundo[((PosY + 1) * 61) + PosX] == 9
                || mundo[((PosY + 1) * 61) + PosX] == 14):
                        opciones[1] = false
                        opciones[5] = false
                        opciones[8] = false
                        opciones[10] = false
                        opciones[11] = false
                        opciones[13] = false
                        opciones[14] = false

#Si la celda de abajo NO esta vacía, está cerrada por arriba y cancelo las opciones con
#puerta hacia abajo.
                elif (mundo[((PosY + 1) * 61) + PosX] != null):
                        opciones[0] = false
                        opciones[2] = false
                        opciones[3] = false
                        opciones[4] = false
                        opciones[6] = false
                        opciones[7] = false
                        opciones[9] = false
                        opciones[12] = false
        else:
                opciones[0] = false
                opciones[2] = false
                opciones[3] = false
                opciones[4] = false
                opciones[6] = false
                opciones[7] = false
                opciones[9] = false
```

```
                opciones[12] = false

#Si estamos cerrando los caminos existentes se toma la última celda válida, la más restrictiva.
        if (estoyCerrando):
                var i = 14
                while (i>=0 && retornar == -1):
                        if (opciones[i] != false):
                                retornar = i
                        i = i - 1

#En otro caso se elije aleatoriamente una de las 15 posiciones del array y si está marcada
#como válida, se toma.
        else:
                var EncontradoViable = false
                var arrayDeOpciones = [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]
                randomize()
                retornar = floor(rand_range(0, arrayDeOpciones.size()))
                while (EncontradoViable == false && !arrayDeOpciones.empty()):
                        if (opciones[arrayDeOpciones[retornar]] != false):
                                EncontradoViable = true
                                retornar = arrayDeOpciones[retornar]
                        else:
                                randomize()
                                arrayDeOpciones.remove(retornar)
                                retornar = floor(rand_range(0, arrayDeOpciones.size()))
                if arrayDeOpciones.empty():
                        retornar = -1
        return retornar
```

Método leeTiposDeFichero():

```
static func leeTiposDeFichero():
#Leere que tipos de nodo (S,H,CL...) existen y su número a partir del index.
                var file = File.new()
                var diccionarioNodos = {}
                if file.open("res://Reglas/index.txt", File.READ) != 0:
                    print("ERROR: No existe INDEX.")
                    return

                #Tomo la siguiente frase como un array delimitado por los espacios
                var next = file.get_line()
                while(str(next)!='EOF'):
                        diccionarioNodos[str(next)]=0
                        next = file.get_line()
                file.close()

                var file2Check = File.new()
                var tipos = diccionarioNodos.keys()
                var cuenta = 1
                for i in tipos:
                        var siguiente = false
                        while (siguiente == false):
                                if !file2Check.file_exists("res://Reglas/"+str(i)+str(cuenta)+".dot"):
                                        cuenta = 1
                                        siguiente = true
                                else:
                                        diccionarioNodos[i] = cuenta
                                        cuenta = cuenta + 1
                return diccionarioNodos
```

Método leeTiposTraducibles():

```
static func leeTiposTraducibles():
#Leere que símbolos son las variables a partir del fichero traducibles.
                var file = File.new()
                var ArrayTraducciones = []
                if file.open("res://Reglas/traducibles.txt", File.READ) != 0:
                    print("ERROR: No existe TRADUCIBLES.")
                    return
                var next = file.get_line()
                while(str(next)!='EOF'):
                        ArrayTraducciones.append(str(next))
                        next = file.get_line()
                file.close()
                return ArrayTraducciones
```

Clase Nodo:

```
class Nodo:
        var padres = []
        var hijos = []
        var nombre = 0
        var final = true
        var tipo = ''
        var hijoDoble = false
        var padreDoble = null
        var pintado = false

        func _init(final, descrip, nombre):
                self.tipo = descrip
                self.final = final
                self.nombre = nombre

        func addHijo(nodoHijo):
                self.hijos.append(nodoHijo)

        func addPadre(nodoPadre):
                self.padres.append(nodoPadre)

        func deletePadre(nodoPadre):
                var pos = self.padres.find(nodoPadre,0)
                if pos != -1:
                        self.padres.remove(pos)

        func deleteHijo(nodoHijo):
                var pos = self.hijos.find(nodoHijo,0)
                if pos != -1:
                        self.hijos.remove(pos)
```

Clase Grafo:

```
class Grafo:
        var Nodos = []
        var NodoOrigen = null
        var index = 0
        var todoFinales = false


        func _init(inicial):
                self.index = 1
                self.NodoOrigen = inicial
                self.Nodos.append(inicial)

        func addIndex():
                self.index = self.index + 1
                return self.index

        func addNodo(nodoNuevo):
```

```
        self.Nodos.append(nodoNuevo)

    func toString():
        print('digraph␣A{')
        var retorno = 'digraph␣A{'
        var finalOno = 'square'
        for i in range(0, self.Nodos.size()):
            if self.Nodos[i].final == true:
                finalOno = 'circle'
            else:
                finalOno = 'square'
            print('\t'+str(self.Nodos[i].nombre)+'␣[shape␣=␣"'+finalOno +'",label␣=␣"'+self.Nodos[i].tipo+'"];')
            retorno = retorno + '\r\n'+str(self.Nodos[i].nombre)+'␣[shape␣=␣"'+finalOno+'",␣label␣=␣"'+self.Nodos[i].tipo+'"];'
        for i in range(0, self.Nodos.size()):
            for j in range(0, self.Nodos[i].hijos.size()):
                if self.Nodos[i].hijos[j].padreDoble == null:
                    print('\t'+str(self.Nodos[i].nombre)+'␣->␣'+str(self.Nodos[i].hijos[j].nombre)+';')
                    retorno = retorno + '\r\n'+str(self.Nodos[i].nombre)+'␣->'+str(self.Nodos[i].hijos[j].nombre)+';'
                else:
                    if self.Nodos[i].nombre == self.Nodos[i].hijos[j].padreDoble :
                        print('\t'+str(self.Nodos[i].nombre)+'␣->␣'+str(self.Nodos[i].hijos[j].nombre)+'[penwidth=3];')
                        retorno = retorno + '\r\n'+str(self.Nodos[i].nombre)+'␣->␣'+str(self.Nodos[i].hijos[j].nombre)+'␣[
                            penwidth=3];'
                    else:
                    #El hijo puede tener un padre doble, pero no con este padre
                        print('\t'+str(self.Nodos[i].nombre)+'␣->'+str(self.Nodos[i].hijos[j].nombre)+';')
                        retorno = retorno + '\r\n'+str(self.Nodos[i].nombre)+'␣->␣'+str(self.Nodos[i].hijos[j].nombre)+';'
        print('}')
        retorno = retorno + '\r\n}'
        return retorno
```

Método sustituir():

```
func sustituir(NI, grafo):
    var tipo = str(NI.tipo)
    var ArrayHijos = []
    for i in conteoDeReglas.keys():
        #Miro los posibles hijos que puede tener el nodo
        if (tipo+"−") in i:
            var longitudHijo = i.length ()
            var longitudPadre = tipo.length ()
            var tipoDeHijo = i.right(longitudPadre+1)
            #Compruebo si el nodo padre tiene un hijo que permita una doble sustitución
            for j in NI.hijos:
                if j.tipo == tipoDeHijo:
                    ArrayHijos.append(j)

    var hijo = null
    if !ArrayHijos.empty():
        var numOpciones = ArrayHijos.size()
        randomize()
        var eligeTraduccion = floor(rand_range(0, numOpciones))
        hijo = ArrayHijos[eligeTraduccion]
        tipo = str(tipo + "−" + hijo.tipo)
    var eligeReglaNumero = str(floor(rand_range(1, conteoDeReglas[tipo]+1)))

    var file = File.new()
    if file.open("res://Reglas/"+tipo+eligeReglaNumero+".dot", File.READ) != 0:
        print("ERROR:␣No␣existe␣esa␣regla.")
        return
    #Me deshago de la primera linea
    var next = file.get_line()
    var nodos = []
    nodos.append(NI)
    var numNodos = 1

    #Si tiene un hijo en la transformacion
    if hijo!=null:
        nodos.append(hijo)
        numNodos = 2

    var segundaParte = false;
```

```
#Tomo la siguiente frase como un array delimitado por los espacios
next = file.get_csv_line(' ')
#Hasta el final del fichero DOT
while (next[0] != '}'):
	#Mientras no vea un guión sigo
	if (next[0] != '-' && segundaParte == false):
		#Se guarda para las conexiones
		var numeroNodo = next[0]
		var final = null
		if (next[2] == 'circle'):
			final = true
		else:
			final = false
		#Tomo los valores entre " "
		var titulo = next[5]
		if (int(numeroNodo) == 1):
			nodos[0].final = final
			nodos[0].tipo = titulo
			NI.final = final
			NI.tipo = titulo
		#Es una doble transformacion. Este sera el nodo final
		elif (int(numeroNodo) == 2 && hijo!=null):
			nodos[1].final = final
			nodos[1].tipo = titulo
			hijo.final = final
			hijo.tipo = titulo
		else:
			var nuevoNodo = Nodo.new(final,titulo,grafo.addIndex())
			nodos.append(nuevoNodo)
			numNodos = numNodos +1
	elif (segundaParte == true):
		var union = next[1]
		if union == '->':
			nodos[int(next[0])-1].addHijo(nodos[int(next[2])-1])
			nodos[int(next[2])-1].addPadre(nodos[int(next[0])-1])
		elif union == '=>':
			#Indico que ese nodo tiene una unión de hijoDoble (doble flecha)
			nodos[int(next[2])-1].hijoDoble = true
			#Sino cualquier unión a un hijoDoble es directa
			nodos[int(next[2])-1].padreDoble = nodos[int(next[0])-1].nombre
			nodos[int(next[0])-1].addHijo(nodos[int(next[2])-1])
			nodos[int(next[2])-1].addPadre(nodos[int(next[0])-1])
	elif (next[0] == '-'):
		segundaParte = true
	next = file.get_csv_line(' ')

#Los nodos están creados, asi que los meto en el grafo
var inicio = 1
if (hijo!=null):
	NI.deleteHijo(hijo)
	hijo.deletePadre(NI)
	inicio = 2
for i in range(inicio,numNodos):
	grafo.addNodo(nodos[i])
file.close()
```

Métodos limpiaOpciones() y bloqueaOpciones():

```
func limpiaOpciones():
	opciones.clear()
	for i in range(15):
		opciones.append(true)

func bloqueaOpciones( var tipo):
	#solo puedo pintar celdas en forma de 'T'
	if (tipo == 'n') || (tipo == 't') || (tipo == 'ts') || (tipo == 'ib') :
		opciones.clear()
		for i in range(15):
			opciones.append(false)
		opciones[1] = true
		opciones[2] = true
		opciones[3] = true
		opciones[4] = true

	#solo puedo pintar celdas en forma de '-'
	elif (tipo == 'l') || (tipo == 'lm') || (tipo == 'lf') :
		opciones.clear()
		for i in range(15):
			opciones.append(false)
		opciones[9] = true
		opciones[10] = true

	else:
		opciones.clear()
		for i in range(15):
			opciones.append(true)
```

Método listaLosCaminos():

```
func listaLosCaminos (posActualX, posActualY, celdaAnhadida, padre):
	randomize()
	#Aleatoriamente eligo una posición donde insertare los posibles nuevos caminos, de manera que
	#al tomarlos en orden no se siga un patrón
	var insertarEn = floor(rand_range(0, caminosX.size()))

	#Miro si hay hueco/camino debajo, en caso que sí, añado opción de camino
	if ((celdaAnhadida == 0
	|| celdaAnhadida == 2
	|| celdaAnhadida == 3
	|| celdaAnhadida == 4
	|| celdaAnhadida == 6
	|| celdaAnhadida == 7
	|| celdaAnhadida == 9
	|| celdaAnhadida == 12) && (posActualY + 1 < 61) && (mundo[((posActualY+1) * 61) + posActualX] == null)):
		caminosX.insert(insertarEn, posActualX)
		caminosY.insert(insertarEn, posActualY + 1)
		pertenencia[(posActualY+1) * 61 + posActualX] = padre

	#Miro a la derecha
	if ((celdaAnhadida == 0
	|| celdaAnhadida == 1
	|| celdaAnhadida == 2
	|| celdaAnhadida == 3
	|| celdaAnhadida == 5
	|| celdaAnhadida == 6
	|| celdaAnhadida == 10
	|| celdaAnhadida == 11) && (posActualX + 1 < 61) && (mundo[(posActualY * 61) + posActualX + 1] == null)):
		caminosX.insert(insertarEn, posActualX + 1)
		caminosY.insert(insertarEn, posActualY)
		pertenencia[posActualY * 61 + (posActualX+1)] = padre

	#Miro arriba
	if ((celdaAnhadida == 0
	|| celdaAnhadida == 1
	|| celdaAnhadida == 2
	|| celdaAnhadida == 4
	|| celdaAnhadida == 5
	|| celdaAnhadida == 8
	|| celdaAnhadida == 9
	|| celdaAnhadida == 14) && (posActualY - 1 >= 0) && (mundo[((posActualY-1) * 61) + posActualX] == null)):
		caminosX.insert(insertarEn, posActualX)
		caminosY.insert(insertarEn, posActualY - 1)
		pertenencia[(posActualY-1) * 61 + posActualX] = padre

	#Miro a la izquierda
	if ((celdaAnhadida == 0
```

```
    || celdaAnhadida == 1
    || celdaAnhadida == 3
    || celdaAnhadida == 4
    || celdaAnhadida == 7
    || celdaAnhadida == 8
    || celdaAnhadida == 10
    || celdaAnhadida == 13) && (posActualX − 1 >=0) && (mundo[(posActualY * 61) + posActualX − 1] == null)):
        caminosX.insert(insertarEn, posActualX − 1)
        caminosY.insert(insertarEn, posActualY)
        pertenencia[posActualY * 61 + (posActualX−1)] = padre
```

Método ordenaNodos():

```
func ordenaNodos(var nodos):
    #pongo los nodos en el orden preferible para su inclusión al mapa
    var nuevoArray = []
    var origen = nodos[0]
    nuevoArray.append(origen)
    var devolucion = −1 #comprobante de si un nodo está añadido

    for i in origen.hijos:
        devolucion = nuevoArray.find(i)
        #si el nodo no esta añadido, y es hijo doble de ESTE padre...
        if i.hijoDoble && devolucion == −1 && origen.nombre == i.padreDoble:
            nuevoArray = anadeHijo(i, nuevoArray, origen, true)
    for i in origen.hijos:
        devolucion = nuevoArray.find(i)
        #si el nodo no esta añadido y no es hijo doble
        if !i.hijoDoble && devolucion == −1:
            nuevoArray = anadeHijo(i, nuevoArray, origen, false)

    return nuevoArray
```

Método anadeHijo():

```
func anadeHijo(var nodo, var nuevoArray, var padre, var tipo):#los hijos se añadirán de forma recursiva
    var pos = nuevoArray.find(padre)

    if !(nodo in nuevoArray):
        if tipo:
            #si es hijoDoble, lo añado directamente al lado del padre
            if pos == nuevoArray.size():
                nuevoArray.append(nodo)
            else:
                nuevoArray.insert(pos+1,nodo)

        else:
            #buscaré en el array el siguiente nodo no hijoDoble
            var next = pos+1
            while next < nuevoArray.size() && nuevoArray[next].hijoDoble:
                next = next + 1
            if next == nuevoArray.size():
                nuevoArray.append(nodo)
            else:
                nuevoArray.insert(next,nodo)

    var devolucion = −1
    for i in nodo.hijos:
        devolucion = nuevoArray.find(i)
        #lo añado como hijoDoble si es hijo de este nodo y no de otro
        if i.hijoDoble && i.padreDoble == nodo.nombre && devolucion == −1:
            nuevoArray = anadeHijo(i, nuevoArray, nodo, true)
    for i in nodo.hijos:
```

```
            devolucion = nuevoArray.find(i)
            if !i.hijoDoble && devolucion == −1:
                nuevoArray = anadeHijo(i, nuevoArray, nodo, false)

    return nuevoArray
```

Método setNewArray():

```
func SetNewArray():
    var escogido = −1 #Pieza de mapa elegida
    limpiaOpciones()

    var PosX = 31
    var PosY = 31

    var Nodos = grafo.Nodos
    #Los ordeno de manera optima para pintarse
    Nodos = ordenaNodos(Nodos)

    #En el centro del mapeado una celda abierta por los cuatro lados
    mundo[PosY * 61 + PosX] = 0
    pertenencia[PosY * 61 + PosX] = −1
    tipoCelda [PosY * 61 + PosX] = 'e'
    listaLosCaminos(PosX,PosY,0,−1)
    Nodos[0].pintado = true

    for i in Nodos:
        limpiaCaminosUsados()

        if i.pintado == false:
            if i.tipo == 'n':
                var pintaTres = 0
                if i.hijoDoble == false:
                    #Pinto 3 celdas
                    for num in range(0,3):
                        var j = caminosX.size()−1
                        escogido = −1
                        while escogido == −1 && j>=0:
                            PosX = caminosX[j]
                            PosY = caminosY[j]
                            bloqueaOpciones('n')
                            escogido = aquiQuePinto(false, PosX, PosY, opciones)
                            if escogido != −1:
                                #Si es valido para pintarse, elimino el camino libre
                                caminosX.remove(j)
                                caminosY.remove(j)
                                pintaTres = pintaTres + 1
                                mundo[PosY * 61 + PosX] = escogido
                                tipoCelda[PosY * 61 + PosX] = i.tipo
                                listaLosCaminos(PosX,PosY,escogido,i.nombre)
                            j = j−1

                else:
                    for num in range(0,3):
                        var t = 0 #index para recorrer 'pertenencia'
                        escogido = −1
                        while t<pertenencia.size() && escogido == −1:
                            #Es un espacio vacío junto al 'padre'
                            if (pertenencia[t] == i.padreDoble && mundo[t] == null):
                                PosY = t/61 #Obtengo su posicion x,y
                                PosX = t %61
                                #Index para recorrer los caminos abiertos
                                var p = 0
                                while p < caminosX.size() && escogido == −1:
                                    if (caminosX[p] == PosX && caminosY[p] == PosY):
                                        bloqueaOpciones('n')
                                        escogido = aquiQuePinto(false, PosX, PosY, opciones)
                                        if escogido != −1:
                                            #Voy a pintar este 'camino', por lo que se elimina
                                            caminosX.remove(p)
                                            caminosY.remove(p)
                                            pintaTres = pintaTres + 1
                                            mundo[t] = escogido
                                            tipoCelda[t] = i.tipo
                                            listaLosCaminos(PosX,PosY,escogido,i.nombre)
                                    p = p + 1
                            t = t + 1

                if pintaTres <3:
                    print ('ERROR:␣El␣nodo␣' + str(i.nombre) + '(caso␣n)␣no␣se␣ha␣podido␣pintar')
                    return null
                else:
                    i.pintado == true

            else:
                if i.hijoDoble == false:
                    var j = caminosX.size()−1
                    escogido = −1
                    while escogido == −1 && j>=0:
                        PosX = caminosX[j]
                        PosY = caminosY[j]
                        bloqueaOpciones(i.tipo)
                        escogido = aquiQuePinto(false, PosX, PosY, opciones)
                        if escogido != −1:
                            #Si es valido para pintarse, elimino el camino libre
                            caminosX.remove(j)
                            caminosY.remove(j)
```

```
                                mundo[PosY * 61 + PosX] = escogido
                                tipoCelda[PosY * 61 + PosX] = i.tipo
                                listaLosCaminos(PosX,PosY,escogido,i.nombre)
                                i.pintado = true
                        j = j-1

                else:
                        var t = 0 #Index para el array 'pertenencia'
                        escogido = -1
                        while t<pertenencia.size() && escogido == -1:
                                #Es un espacio vacio junto al 'padre'
                                if (pertenencia[t] == i.padreDoble && mundo[t] == null):
                                        PosY = t/61 #Obtengo su posicion x,y
                                        PosX = t%61
                                        #Index para el array de caminos abiertos
                                        var p = 0
                                        while p < caminosX.size() && escogido == -1:
                                                if (caminosX[p] == PosX && caminosY[p] == PosY):
                                                        bloqueaOpciones(i.tipo)
                                                        escogido = aquiQuePinto(false, PosX, PosY, opciones)
                                                        if escogido != -1:
                                                                #Voy a pintar este 'camino' por lo que se elimina
                                                                caminosX.remove(p)
                                                                caminosY.remove(p)
                                                                mundo[t] = escogido
                                                                tipoCelda[t] = i.tipo
                                                                listaLosCaminos(PosX,PosY,escogido,i.nombre)
                                                                i.pintado = true
                                                        p = p + 1
                                        t = t + 1

                if i.pintado != true:
                        print ('ERROR: El nodo ' + str(i.nombre) + '(caso '+i.tipo+', padre '+str(i.padreDoble)+') no se ha podido
                                pintar')
                        return null

        #Habiendo creado todas las celdas, cierro los caminos que queden abiertos
        while !caminosX.empty():
                var j = caminosX.size()-1
                PosX = caminosX[j]
                PosY = caminosY[j]
                caminosX.pop_back()
                caminosY.pop_back()
                if mundo[(PosY * 61 + PosX)] ==null:
                        limpiaOpciones()
                        escogido = aquiQuePinto(true, PosX, PosY, opciones)
                        if escogido != -1:
                                print('limpiando' + str(escogido))
                                mundo[(PosY * 61 + PosX)] = escogido
                        listaLosCaminos(PosX, PosY, escogido, -1)
                else:
                        print(str(PosX)+","+str(PosY)+" : Esa posicion ya estaba pintada" + str(mundo[(PosY * 61 + PosX)]) + ".")

        print('Retorno el mundo')
        return mundo
```

Método spawnNewObject():

```
func spawnNewObject(objeto, posicion, inamovible):
        var x
        var y
        var object
        var object_instance
        var instanciado = false

        var area = 7
        if !inamovible:
                randomize()
                area = floor(rand_range(0, 7))
        var vector = getOtherPos(area, posicion)

        if objeto == 0:#bl - BossLevel
                        object = load("res://Scenes/bl.tscn")
                        object_instance = object.instance()
                        object_instance.set_pos(vector)
                        object_instance.set_name("BossLevel")
        elif objeto == 1:#bm - BossMini
                        object = load("res://Scenes/bm.tscn")
                        object_instance = object.instance()
                        object_instance.set_pos(vector)
                        object_instance.set_name("BossMini")
        elif objeto == 2:#g - objetivo final, cofre
```

```
                object = load("res://Scenes/g.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("Cofre")
        elif objeto == 3:#ib - item bonus
                object = load("res://Scenes/ib.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("ItemBonus")
        elif objeto == 4:#iq - item quest
                object = load("res://Scenes/iq.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("ItemQuest")
        elif objeto == 5:#k - llave
                object = load("res://Scenes/k.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("Llave")
        elif objeto == 6:#kf - llave final
                object = load("res://Scenes/kf.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("LlaveFinal")
        elif objeto == 7:#km - llave multi
                if multillaves == 0:
                        object = load("res://Scenes/km1.tscn")
                        multillaves = 1
                elif multillaves == 1:
                        object = load("res://Scenes/km2.tscn")
                        multillaves = 2
                elif multillaves == 2:
                        object = load("res://Scenes/km3.tscn")
                        multillaves = 3
                else:
                        print('ERROR, multillaves mal distribuidas o exceso de las mismas')
                        object = load("res://Scenes/km1.tscn")
                        multillaves = 1
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("KeyPart")
        elif objeto == 8:#l - candado
                object = load("res://Scenes/l.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("Lock")
        elif objeto == 9:#lf - candado final
                object = load("res://Scenes/lf.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("LockFinal")
        elif objeto == 10:#lm - candado multiple
                object = load("res://Scenes/lm.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("LockMulti")
        elif objeto == 11:#t - prueba bolas de fuego
                object = load("res://Scenes/monstruo.tscn")
                object_instance = object.instance()
```

```
                object_instance.set_pos(vector)
                object_instance.set_name("trampa")
        elif objeto == 12:#ti - prueba + item
                object = load("res://Scenes/t.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("monstruo")
        elif objeto == 13:#ts - prueba + secreto
                object = load("res://Scenes/t.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("pruebasecreto")
        if objeto <14 && objeto >-1 && instanciado == false:
                add_child(object_instance)

        #Ahora añado algún objeto o monstruo aleatorio
        var area2 = area
        while(area2==area):
                randomize()
                area2 = floor(rand_range(0, 7))

        var vector = getOtherPos(area2, posicion)
        randomize()
        var ayudaOTropiezo = floor(rand_range(0, 10))
        if ayudaOTropiezo < 9*endurecer:
                object = load("res://Scenes/monstruo.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("monstruo")
                add_child(object_instance)
        else:
                object = load("res://Scenes/ib.tscn")
                object_instance = object.instance()
                object_instance.set_pos(vector)
                object_instance.set_name("ItemBonus")
                add_child(object_instance)
```

Método getOtherPos():

```
func getOtherPos(area, posicion):
        var x
        var y

        if area == 1:
                x = posicion.x-384
                y = posicion.y-216
        elif area == 2:
                x = posicion.x
                y = posicion.y-216
        elif area == 3:
                x = posicion.x+384
                y = posicion.y-216
        elif area == 4:
                x = posicion.x-384
                y = posicion.y+216
        elif area == 5:
                x = posicion.x
                y = posicion.y+216
        elif area == 6:
                x = posicion.x+384
```

```
            y = posicion.y+216
        else:
            x = posicion.x
            y = posicion.y

        return Vector2(x,y)
```

Método ready():

```
func _ready():
    var milisegundos = OS.get_ticks_msec()
    var timeDict = OS.get_time();
    var hour = timeDict.hour;
    var minute = timeDict.minute;
    var seconds = timeDict.second;
    print(str(hour) + ':' + str(minute) + ':' + str(seconds)+ ':' + str(milisegundos) + ' :: Inicio de la configuración del mundo')

    grafo = get_node("NodoGramatica").grafo
    mundo.clear()
    pertenencia.clear()
    tipoCelda.clear()
    for i in range(3721):
        mundo.append(null)
        pertenencia.append(null)
        tipoCelda.append(null)

    var nuevoMundo = SetNewArray()

    while nuevoMundo == null || (mundo[31 * 61 + 31] == null) || (mundo[30 * 61 + 31] == null && mundo[32 * 61 + 31] == null && mundo
        [31 * 61 + 30] == null && mundo[31 * 61 + 32] == null ):
        print('Re-ejecuto EscenaCodigo por fallo de uniones')
        get_node("NodoGramatica").creaOtroGrafo()
        grafo = get_node("NodoGramatica").grafo
        limpiaOpciones()
        mundo.clear()
        pertenencia.clear()
        tipoCelda.clear()
        for i in range(3721):
            mundo.append(null)
            pertenencia.append(null)
            tipoCelda.append(null)

        milisegundos = OS.get_ticks_msec()
        timeDict = OS.get_time();
        hour = timeDict.hour;
        minute = timeDict.minute;
        seconds = timeDict.second;
        print(str(hour) + ':' + str(minute) + ':' + str(seconds)+ ':' + str(milisegundos) + ' :: Inicio de la configuración del mundo')

        nuevoMundo = SetNewArray()

    milisegundos = OS.get_ticks_msec()
    timeDict = OS.get_time();
    hour = timeDict.hour;
    minute = timeDict.minute;
    seconds = timeDict.second;
    print(str(hour) + ':' + str(minute) + ':' + str(seconds)+ ':' + str(milisegundos) + ' :: Mundo configurado')

    var PosX = -1
    var PosY = -1

    milisegundos = OS.get_ticks_msec()
    timeDict = OS.get_time();
    hour = timeDict.hour;
    minute = timeDict.minute;
    seconds = timeDict.second;
    print(str(hour) + ':' + str(minute) + ':' + str(seconds)+ ':' + str(milisegundos) + ' :: Inicio de la recreación del mundo')

    var decima = 3721/10
    var sumale = decima
    var porcentaje = 0
    print('Loading: '+str(porcentaje)+' %')
    for i in range(3721):

        if(i!=0):
            if(i>decima):
                porcentaje +=10
                decima += sumale
                print('Loading: '+str(porcentaje)+' %')

        if (mundo[i]==null):
            var scene = load("res://Secciones/15.tscn")
            PosY = i / 61
            PosX = i % 61
            var scene_instance = scene.instance()
            scene_instance.set_pos(Vector2((PosX-31)*1920,(PosY-31)*1080))
            scene_instance.set_name("scene15")
            add_child(scene_instance)

        else:
            var scene = load("res://Secciones/"+str(mundo[i])+".tscn")
            PosY = i / 61
            PosX = i % 61
            var scene_instance = scene.instance()
            var positionPadre = Vector2((PosX-31)*1920,(PosY-31)*1080)
            scene_instance.set_pos(positionPadre)
```

```
                scene_instance.set_name("scene"+str(mundo[i])+"")
                add_child(scene_instance)

                var inamovible = false
                var elijoObjeto = -1
                if !tipoCelda[i] == 'e':
                        if tipoCelda[i] == 'bl':
                                elijoObjeto = 0
                                inamovible = true
                        elif tipoCelda[i] == 'bm':
                                elijoObjeto = 1
                                inamovible = true
                        elif tipoCelda[i] == 'g':
                                elijoObjeto = 2
                                inamovible = true
                        elif tipoCelda[i] == 'ib':
                                elijoObjeto = 3
                        elif tipoCelda[i] == 'iq':
                                elijoObjeto = 4
                        elif tipoCelda[i] == 'k':
                                elijoObjeto = 5
                        elif tipoCelda[i] == 'kf':
                                elijoObjeto = 6
                        elif tipoCelda[i] == 'km':
                                elijoObjeto = 7
                        elif tipoCelda[i] == 'l':
                                elijoObjeto = 8
                                inamovible = true
                        elif tipoCelda[i] == 'lf':
                                elijoObjeto = 9
                                inamovible = true
                        elif tipoCelda[i] == 'lm':
                                elijoObjeto = 10
                                inamovible = true
                        elif tipoCelda[i] == 't':
                                elijoObjeto = 11
                        elif tipoCelda[i] == 'ti':
                                elijoObjeto = 12
                        elif tipoCelda[i] == 'ts':
                                elijoObjeto = 13

                        spawnNewObject(elijoObjeto,positionPadre,inamovible)

        milisegundos = OS.get_ticks_msec()
        timeDict = OS.get_time();
        hour = timeDict.hour;
        minute = timeDict.minute;
        seconds = timeDict.second;
        print(str(hour) + ':' + str(minute) + ':' + str(seconds)+ ':' + str(milisegundos) + ' :: Mundo recreado')
```

Referencias

E. Adams y J. Dormans. *Game mechanics: advanced game design*. New Riders, 2012.

Bethesda Softworks. The elder scrolls ii: Daggerfall, 1996.

Blizzard North. Diablo, 1996.

A. Cannizzo y E. Ramírez. Towards Procedural Map and Character Generation for the MOBA Game Genre. *IngenierÃa y Ciencia*, 11:95 – 119, 07 2015.
http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S1794-91652015000200005&nrm=iso

Don Daglow. Dungeon, 1975.

Don Worth. Beneath apple manor, 1978.

J. Dormans. Adventures in level design: Generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 1:1–1:8, New York, NY, USA, 2010. ACM.
http://doi.acm.org/10.1145/1814256.1814257

E. Yeguas and R. Muñoz-Salinas and R. Medina Carnicer. Example-based procedural modelling by geometric constraint solving.
https://youtu.be/3ydDlAuPr3g Accessed: 2017-07-11.

Gary Whisenhunt y Ray Wood. Dnd, 1974.

Gearbox Software. Borderlands, 2009.

Godot Engine. Godot docs.
https://godot.readthedocs.io/ Accessed: 2017-05-02.

Hello Games. No man's sky, 2016.
https://www.nomanssky.com/

Jeremy Kun. The cellular automaton method for cave generation.
https://jeremykun.com/2012/07/29/the-cellular-automaton-method-for-cave-generation/ Accessed: 2017-07-11.

Jon Lane. Original rogue source.
http://www.rots.net/rogue/source/source.htm Accessed: 2017-08-14.

K. Krawczyk, W. Dzwinel y D. A. Yuen. Nonlinear development of bacterial colony modeled with cellular automata and agent objects. *International Journal of Modern Physics C (IJMPC)*, 14(10): 1385–1404, 2003.
https://EconPapers.repec.org/RePEc:wsi:ijmpcx:v:14:y:2003:i:10:n:s0129183103006199

Luis Migel Pardo Vasallo y Domingo Gómez Pérez. Ocw unican - teoría de autómatas y lenguajes formales: Las gramáticas formales, 2010.
https://ocw.unican.es/course/view.php?id=183

Michael Toy, Glenn Wichman y Ken Arnold. Rogue, 1980.

Mojang. Minecraft, 2011.

Robert Alan Koeneke y Jimmey Wayne Todd. Moria, 1983.

J. Togelius, G. N. Yannakakis, K. O. Stanley, y C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, Sept. 2011.
doi.ieeecomputersociety.org/10.1109/TCIAIG.2011.2148116

Valve Software. Left 4 dead, 2008.