Rubén Martínez Vilar.

Grado en Ingeniería Multimedia. 2014-2015.

Universidad de Alicante.

Videojuego basado en generación procedimental de mazmorras

ÍNDICE DE CONTENIDOS

[CAPÍTULO 1. EL PROYECTO 4](#_Toc421548684)

[Motivación 4](#_Toc421548685)

[Objetivos 5](#_Toc421548686)

[Estructura del documento 6](#_Toc421548687)

[CAPÍTULO 2. PROCEDIMENTAL. 8](#_Toc421548688)

[La generación procedimental 8](#_Toc421548689)

[CAPÍTULO 3. GENERACIÓN DE CONTENIDO EN VIDEOJUEGOS 22](#_Toc421548690)

[Los limitados comienzos 22](#_Toc421548691)

[Una nueva generación 26](#_Toc421548692)

[La era independiente 29](#_Toc421548693)

[CAPÍTULO 4. ALGORITMOS 40](#_Toc421548694)

[Clasificación 40](#_Toc421548695)

[Introduciendo algunos conceptos 42](#_Toc421548696)

[El autómata celular 50](#_Toc421548697)

[Generación de mazmorras. Posicionamiento aleatorio. 61](#_Toc421548698)

[Generación de mazmorras. Particionado BSP. 63](#_Toc421548699)

[Generación de contenido en mazmorras 73](#_Toc421548700)

[CAPÍTULO 5. DESARROLLO DE UN VIDEOJUEGO 78](#_Toc421548701)

[Generación de mazmorra basada en árbol BSP 78](#_Toc421548702)

[Agregando contenido a la mazmorra 82](#_Toc421548703)

[CAPÍTULO 6. METODOLOGÍA DE TRABAJO 84](#_Toc421548704)

[Planificación 84](#_Toc421548705)

[Herramientas 84](#_Toc421548706)

[CAPÍTULO 7. VALORACIÓN FINAL Y CONCLUSIONES 88](#_Toc421548707)

[BIBLIOGRAFÍA 89](#_Toc421548708)

# CAPÍTULO 1. EL PROYECTO

## Motivación

Desde el lanzamiento de Minecraft los videojuegos basados en la generación de niveles o mundos de manera semi-aleatoria ha explotado en popularidad entre las creaciones de corte más independiente, donde los recursos son bastante limitados y hay que recurrir a técnicas que puedan aportar la mayor cantidad de contenido sin que esto robe más tiempo del proyecto de lo que debería ser necesario. Con esto me refiero a pequeños estudios donde no se pueden permitir tener a personas dedicadas únicamente a un aspecto del diseño como pueden ser los escenarios. En este aspecto cabe destacar a Minecraft debido a que desde su lanzamiento han proliferado una gran cantidad de clones y variaciones desde todos los rincones del ambiente académico, pudiendo encontrar en internet cientos de repositorios con alguna pequeña implementación de generación de un mundo utilizando algún algoritmo de los que vamos a estudiar y cualquier lenguaje de programación imaginable.

Ahora incluso estamos presenciando lo que parece ser una nueva generación en este sector con el prometedor No Man’s Sky, que pretende recrear un enorme universo lleno de vida y actividades para el jugador. En este caso ya no estamos hablando solamente de utilizar algoritmos y parámetros para generar el terreno, estructuras y las estadísticas de los objetos, sino que incluso las especies animales, plantas y demás vida que podamos encontrar en este universo.

Mi interés por este tipo de técnicas comenzó hace unos años cuando ejecuté por primera vez el videojuego Minecraft y, aunque reticente al principio, debido principalmente a su aspecto simple y un poco amateur, en cuanto me puse a explorar me quedé fascinado al ver como debajo de una jugabilidad y aspecto simple se escondían mundos enormes que se generaban delante de mis ojos, donde se podían encontrar rincones escondidos con formaciones extrañas y nada habituales dentro de ese mismo mundo. Disfrutaba solo con el hecho de comenzar una nueva partida, introduciendo un nombre de semilla a mi gusto y viendo el resultado, o adentrándome en algún foro o hilo de Reddit para buscar semillas curiosas que otros usuarios habían descubierto, aportando coordenadas donde podíamos encontrar formaciones de minerales extraños y poco frecuentes.

A pesar de mi interés por esos mundos aleatorios, no ha sido hasta ahora que he encontrado el momento perfecto para estudiar estas técnicas de generación, ahora que mis conocimientos de programación me dan la confianza suficiente para hacerlo y tengo la oportunidad perfecta con este trabajo de fin de grado.

En este trabajo propongo un **estudio de técnicas de generación procedimental** como las utilizadas en Minecraft, Spelunky o Faster than light, así como la realización de un videojuego de tipo **dungeon crawler** o **rogue-like** que genere el contenido, desde la estructura general de habitaciones hasta los objetos, de esta manera.

## Objetivos

* **Estudio** de la definición de **generación procedimental** de contenido dentro del sector de los videojuegos como herramienta que contribuye a la jugabilidad de este.
* Estudio de distintos videojuegos que utilizan técnicas de generación procedimental de contenido.
* Estudio de distintos algoritmos de generación de mazmorras y del contenido de estas. Implementación visual de algunos de estos algoritmos que sean más orientados a la generación de mazmorras, pudiendo modificar algunos parámetros desde una interfaz gráfica.
* Diseño y desarrollo de un videojuego sencillo de estilo rogue-like, haciendo uso justificado de alguno de los algoritmos estudiados para la generación del contenido.

## Estructura del documento

En este documento vamos a explorar las distintas definiciones de generación procedural aplicada a videojuegos, veremos el funcionamiento de algunos algoritmos para la generación de mazmorras y el desarrollo del videojuego que acompaña al proyecto.

Este documento consta de 6 capítulos con el siguiente contenido:

* **Capítulo 1**: Motivación y objetivos del proyecto. Introducción al documento.
* **Capítulo 2**: Definición de generación procedimental. Taxonomía.
* **Capítulo 3**: Algoritmos de generación de contenido en videojuegos. Estudio de videojuegos.
* **Capítulo 4**: Clasificación y estudio de algoritmos.
* **Capítulo 5**: Detalles sobre el desarrollo del videojuego. Algoritmos escogidos e implementación específica. Estructura de clases y objetos. Inteligencia artificial.
* **Capítulo 6**: Planificación y herramientas utilizadas en el desarrollo del proyecto.
* **Capítulo 7**: Valoración final y conclusiones.

### Adjuntos. Documento de diseño del videojuego (GDD)

Para evitar la sobrecarga de información secundaria sobre el videojuego, se adjunta por separado el documento de diseño de este. El documento de diseño se crea al comienzo del proyecto para establecer los distintos aspectos del videojuego, comenzando con la idea básica y desgranando cada sección de este, como la jugabilidad, el diseño de niveles, los objetos y enemigos. El documento entonces se puede ir actualizando, principalmente en las primeras etapas del desarrollo cuando aún se están probando las ideas descritas al inicio.

Es importante dejar claro que en el documento de diseño no se llega a incluir ningún detalle sobre el mismo desarrollo del videojuego y que esto se deja para esta misma memoria, donde encontramos los detalles de implementación de los aspectos descritos en el documento de diseño.

# CAPÍTULO 2. PROCEDIMENTAL.

## La generación procedimental

Hablando en términos generales, la **generación procedimental**, o **generación por procedimientos**, es eso mismo, generar **contenido** por medio **algoritmos** en vez de manera manual. Suele estar relacionado con aplicación de computación gráfica y diseño de niveles en videojuegos.

Pero vamos a usar una definición más precisa para este proyecto y describir la generación procedimental de contenido como **la creación de contenido para videojuegos mediante algoritmos y con una limitada e indirecta intervención del usuario**.

### ¿Procedimental o procedural?

La razón por la que planteo esta pregunta es porque si buscamos en la Real Academia Española (RAE) la palabra **procedimental** obtenemos la siguiente definición:

*“adj. Perteneciente o relativa al procedimiento (método de ejecutar algunas cosas)*

Pero si buscamos **procedural** no encontramos nada, y es que esta palabra es constantemente utilizada incorrectamente como anglicismo de la palabra, probablemente porque podría pasar por una palabra española y porque es más rápida y sencillo de pronunciar. En entornos no académicos este anglicismo es la palabra más utilizada por encima de la que encontramos en la RAE.

En cualquier caso no quiero ser demasiado meticuloso con el uso de estas palabras y quiero dejar claro que en este documento voy a intentar utilizar la versión correcta “**procedimental**”, pero que tanto procedimental como procedural son dos palabras que se refieren a lo mismo y que ambas son utilizadas con la misma frecuencia al hablar de este tema.

### Aleatorio vs procedimental

La definición que hemos visto indica que el contenido se genera por procedimientos, pero en ningún momento menciona que se deben aplicar ciertos requisitos. Decir que algo es procedimental frente a aleatorio no es la forma correcta de plantearlo, ya que no son exclusivos uno del otro. El uso de algoritmos para la generación de contenido implica el uso de valores aleatorios, pero siempre dentro de ciertos parámetros, con lo que los resultados son, en cierta medida, predecibles. Decir que algo es simplemente aleatorio no implica más que la semilla inicial, y los resultados pueden ser totalmente impredecibles, pero el uso de aleatoriedad en procedimientos para la generación de contenido es esencial.

### ¿Y a qué nos referimos con contenido?

Pues prácticamente a casi todo lo que podemos encontrar en el juego, desde **objetos** físicos a las mismas **estadísticas** o **propiedades** de estos, así como la **música**, **historia** y **misiones** e incluso las propias **reglas** del juego. Aspectos que quedarían fuera de esta definición serían básicamente el mismo motor del juego o la inteligencia artificial, aunque para esta última existen muchos estudios y métodos de aprendizaje automático que en cierta manera se asemejan a la definición de generación procedimental de contenido, pero realmente no generan nuevo contenido en sí.

Otro aspecto del **contenido** que se genere es que debe hacer el juego “**jugable**”. Se debe poder terminar un nivel generado o utilizar un objeto generado con estas técnicas, que tenga una **utilidad** dentro del tipo de asset al que pertenece. Si es un arma, deberíamos poder usarla para luchar contra enemigos, si es una decoración de escenario, debería estar en el contexto adecuado, situada por el nivel o habitación en una posición razonable.

### Videojuegos

Este es un término mucho más difícil de definir, ya que existen muchos géneros que podrían decirse que están al borde de lo que es un videojuego o cualquier otro tipo de obra audiovisual interactiva.

Una de las definiciones más clásicas detalla un videojuego como

*“Dispositivo electrónico que permite, mediante mandos apropiados, simular juegos en las pantallas de un televisor o de un ordenador.”*

De la RAE: [http://buscon.rae.es/drae/srv/search?val=videojuego](http://www.google.com/url?q=http%3A%2F%2Fbuscon.rae.es%2Fdrae%2Fsrv%2Fsearch%3Fval%3Dvideojuego&sa=D&sntz=1&usg=AFQjCNE3L1aW_y2K_ElQdWGhC8HTasHjxg)

En la Wikipedia tenemos una definición más preciso

*“Un videojuego o juego de video es un juego electrónico en el que una o más personas interactúan, por medio de un controlador, con un dispositivo dotado de imágenes de vídeo. Este dispositivo electrónico, conocido genéricamente como «plataforma», puede ser una computadora, una máquina arcade, una videoconsola o un dispositivo portátil (un teléfono móvil, por ejemplo). Los videojuegos son, hoy por hoy, una de las principales industrias del arte y el entretenimiento.”*

Pero tampoco es el objetivo de este documento indagar más en la definición de videojuego, pero si en el papel de la generación procedimental de contenido para estos y como afecta los distintos aspectos de jugabilidad y diversión.

Por lo tanto la generación procedimental implica el uso de procedimientos o algoritmos computacionales para crear algo en un videojuego. Más concretamente algunos ejemplos de esto pueden ser:

* Una herramienta que crea mazmorras para un juego de aventuras como ***The Legend of Zelda*** sin la intervención de la entrada de parámetros por parte del usuario.
* Un sistema que genera un tablero de juego con cierta combinación de reglas y parámetros. El algoritmo escoge sobre unas reglas y parámetros base para crear nuevo contenido usando la combinación de estos.
* Un motor que funciona como middleware con otro motor de juego para poblar de vegetación un mundo virtual, independientemente de cómo haya sido creado este mundo.

### ¿Y por qué generar contenido de manera procedimental?

Ahora que sabemos los conceptos básicos de lo que significa la generación procedimental de contenido, tenemos que preguntarnos por qué y para qué queremos usar técnicas de este tipo cuando podríamos simplemente diseñar el contenido.

Una de las razones más obvias surge de esto mismo, generar contenido **manualmente** conlleva, normalmente, tener a un **diseñador** o **artista** para hacer esto, con el consiguiente **coste** de mantenimiento de esa persona en cuanto a sueldo y tiempo, ya que algo creado por alguien suele tardar más que si lo hiciera un algoritmo. Dependiendo del **tipo de proyecto**, las **plataformas** donde se quiera publicar y el **presupuesto** del equipo de desarrollo en el que nos encontremos esto puede ser realizable o simplemente imposible de soportar.

Hablando de plataformas, un buen ejemplo de lo tenemos en la era de principios de los **80**, cuando las **computadoras** caseras eran tan **limitadas técnicamente**, sobretodo en espacio en disco, lo que hacía imposible incluir todos los recursos o “assets” para hacer juegos medianamente grandes. De hecho el uso de técnicas procedimentales en videojuegos viene de esto mismo, y como veremos algunos juegos de entonces, como Rogue, fueron los precursores de la generación de contenido de este tipo.



Los BBC Micro fueron una serie de microcomputadores que se hicieron bastante populares al principio de la década de los 80 en el Reino Unido. Tuvieron varios lanzamientos de videojuegos que generaban contenido con técnicas procedimentales.

Hoy en día ya no tenemos esas limitaciones técnicas, a no ser que las queramos imponer nosotros mismos, ni siquiera en dispositivos móviles, que hace años que incluyen memoria lo suficientemente rápida y amplia como para incluir el contenido necesario para el juego. Por ello actualmente la generación de contenido procedimental es algo que viene ligado al mismo diseño del juego. Como es el caso de este mismo proyecto, el propio diseño del juego favorece el uso de estas técnicas para generar una infinidad de situaciones que son lo suficientemente distintas como alargar su vida durante cientos de horas. Y los ejemplos más representativos de esto puede que sean **Minecraft** y **Spenlunky**, juegos muy diferentes en su jugabilidad y ritmo, pero que independientemente del número de veces que se jueguen, siempre pueden brindar una experiencia diferente a la anterior.

Pero esto no solo viene ligado a lo que el diseñador del juego quiere, sino a cómo, hoy en día, se consume el contenido audiovisual. En una era donde estamos saturados con entretenimiento desde los móviles, computadores y demás dispositivos electrónicos conectados a internet, la manera en la consumimos contenido es mucho más rápida que antes, no tenemos el tiempo suficiente para probar todo, y mucho menos para verlo o jugarlo de principio a fin. Por esto tiene más sentido la creación de obras que se puedan consumir de forma episódica y de forma auto conclusiva. En el caso de los videojuegos, que cada partida tenga un inicio y un fin, independientemente de que luego el usuario pueda ver un progreso global a lo largo de las diferentes sesiones. En este sentido la generación de contenido mediante técnicas procedimentales son realmente adecuadas para producir una experiencia distinta, si no única, en cada partida.

### Contenido a medida del usuario

Usando métricas y redes neuronales y midiendo como el jugador responde antes ciertas situaciones, el nuevo contenido generado podría ser manipulado dependiendo de los gustos y necesidades del jugador o mejorar su aprendizaje y adaptabilidad a las mecánicas del juego. De la misma manera puede ayudar a la creatividad, produciendo situaciones radicalmente diferentes a las que podríamos esperar en algo creado manualmente por un humano, ofreciendo una solución válida pero, a su vez, inesperable.

Por supuesto no creo que haga falta ni decir que estas técnicas generan reticencia entre los diseñadores y artistas, a los que les hace perder el control creativo sobre ciertos elementos del videojuego. Este es uno de los claros motivos por lo que algo como la generación procedural de texturas, que hace unos años prometía ser una buena solución a este tedioso proceso de creación de recursos, ha caído en el olvido cuando hoy en día las herramientas de diseño proporcionan una facilidad de uso que permite trabajar muy rápidamente a los artistas y con total control creativo.

Pero el uso de técnicas procedimentales también puede ayudar a entender el diseño. Al diseñar esos mismos procedimientos estamos siendo afrontados con las limitaciones, reglas y problemas que un diseñador o artista tiene que afrontar a la hora de trabajar, nos ayuda a entender este proceso de creación manual de contenido porque tenemos que tener en cuenta y entender este proceso manual para poder automatizarlo.

### Propiedades deseadas en la generación del contenido

Las soluciones basadas en generación procedimental de contenido tienen una serie de propiedades deseables o requeridas que pueden ser diferentes para cada aplicación, algunas de estas propiedades comunes pueden ser:

* **Velocidad**: los requerimientos de velocidad puede variar notablemente según el rango de la aplicación. Normalmente es necesario que esta sea superior a como trabajaría un humano si está realizando un trabajo que se considera similar, pero esto no es tan importante si lo que requiere el proyecto es la creación de contenido de manera creativa y que solo se puede proporcionar por estos mecanismos algorítmicos.
* **Confiabilidad**: el generador de contenido debe garantizar que este se crea dentro de unos criterios de calidad. Debe cumplir unas reglas o parámetros, que en mayor parte son establecidas de manera fija en el propio algoritmo, aunque también puede ser el caso en el que el usuario tenga algo de influencia sobre estos, siempre de manera inconsciente o indirecta.
* **Control**: sería deseable proporcionar cierto control sobre este, permitiendo especificar ciertos aspectos iniciales del mismo contenido que se va a crear. Por ejemplo estaríamos hablando de indicar que paleta de colores o texturas a utilizar en cierta situación, si las rocas deberían ser más suaves o agudas, el rango de tamaños que estas pueden tomar, etc. Pero demasiado control rompe el propio concepto y nunca debe caer en lo predecible. Cuando el resultado de un proceso de generación procedimental es demasiado predecible entonces es que se está haciendo mal.
* **Expresividad y diversidad**: medir la expresividad es difícil y generar contenido que es diverso a la vez que cumple cierta calidad no es nada trivial. Entre generar el contenido de manera totalmente aleatoria y probablemente de poca calidad o generar contenido muy predecible hay que encontrar un compromiso intermedio, con suficiente diversidad a la vez que calidad.
* **Creatividad y credibilidad**: generalmente queremos que el contenido no parezca creado por un generador automático, sino que parezca diseñado por una persona.

### Tipos de generación procedimental

Con la gran variedad de soluciones de generación procedimental de contenido existentes se hace necesario realizar una clasificación según similitudes en los problemas que intentan solucionar. En **PCG Wiki** destacan una serie de artículos de **Andrew Doull**, diseñador de videojuegos, en los que propone una taxonomía que intenta ser definitiva. Otros proponen clasificaciones alternativas, aunque en ambos casos encontramos prácticamente los mismos conceptos. Vamos a estudiar la propuesta de Andrew Doull.

#### Generación aleatoria de niveles en tiempo de ejecución

Generación en tiempo de ejecución hace referencia a la generación de contenido que se produce mientras la **aplicación** o el juego está **en marcha**, independientemente de que se produzca antes de cargar un nivel o durante el progreso de este.

En generación de niveles requiere que estén bien **definidos los límites** de este así como los parámetros base para el algoritmo ya que normalmente el juego debe almacenar el nivel completo y es necesario definir sus límites de manera concreta.

Dentro de los **niveles** las distintas **zonas** de este deben estar **siempre conectadas** de alguna manera, lo que hace que las reglas de generación sean más complejas ya que deben asegurar este requerimiento.

La **conectividad entre distintos niveles** se puede realizar de varias manera, desde el uso de puertas o escaleras a un paso entre niveles sin límites visibles, con lo que da la apariencia de ser un mundo continuo. Tanto la **conectividad** de zonas dentro de un nivel como entre distintos niveles suelen estar representados por algún tipo de **grafo**.

Es el tipo más general y en lo primero en lo que se piensa cuando se habla de generación procedimental de contenido en videojuegos. Un ejemplo clásico de esto es el juego Rogue, de 1980, que implementa un generador de mazmorras con habitaciones y pasillos.

#### Diseño de contenido de niveles

Se trata del uso de técnicas de generación para crear los recursos del juego, como pueden ser enemigos o armas. El objetivo es ayudar al diseñador y programador a generar contenido rápidamente y poblar el nivel atendiendo a los requerimientos del juego. Se trata de un tipo de generación bastante útil en proyectos de pocos recursos, desarrolladores independientes con equipos de pocas personas, donde pueden dejar la creación del contenido relacionado a objetos y enemigos al generador, pudiendo así abarcar un mundo más grande sin tener que crear y posicionar cada elemento manualmente. Esto es muy típico en juegos de tipo sandbox donde el mundo es tan grande y con tanto contenido que se hace mucho más conveniente que hacerlo manualmente.

Ejemplos clásicos de este tipo de generación es el uso de técnicas de mapas de alturas mediante ruido de **Perlin**, como el utilizado en **Minecraft**.

#### Generación dinámica de mundos

Con generación dinámica nos referimos a la que se hace “al vuelo” durante la partida. No confundir con la generación en tiempo de ejecución, en este caso nos referimos a contenido que se crea al comenzar o durante una partida pero que no se almacena permanentemente por el juego.

Para generar este contenido se hace uso de una semilla o un hash que permanece constante para una partida o mundo, de esta manera se puede revisitar sin que se produzcan cambios en este. Por otro lado hay que tener en cuenta que el jugador puede producir cambios permanentes en el mundo, estos cambios sí que se almacenan en disco, pero el mundo inicial en sí no es necesario ya que la semilla permite reconstruirlo cada vez. Nada generado procedimentalmente es guardado permanentemente en disco.

Una vez generada la semilla se expande la creación mediante el uso de un generador de números semi-aleatorios. Este generador de números toma la semilla como base y permite al algoritmo tomar decisiones sobre el tipo de contenido a crear basándose también las reglas establecidas por el diseñador.

Normalmente el diseño de estos algoritmos se produce de una manera jerárquica, donde se comienza con una decisión aleatorio en base a la semilla y el resto de contenido se genera en cadena a partir de cada paso anterior, por lo que un cambio en las etapas iniciales lleva a una generación totalmente diferente. Esta estructura de tiene el problema de hacer difícil el ajuste del contenido hasta ciertas etapas finales de la generación, donde empieza a tomar la forma final.

#### Instanciación de entidades del juego

Se trata de una técnica cada vez más común por la que se hace uso de middleware para cambiar dinámicamente ciertos parámetros de las entidades de la aplicación para crear la mayor variedad posible de una misma entidad y reducir la repetición. Esto se puede ver claramente en sistemas naturales como bosques, donde cada árbol puede provenir de una entidad base pero con ciertas variaciones sobre esta.

Se suele hacer uso de números aleatorios de manera que el juego no necesita almacenar la información de cada uno de los objetos, sino simplemente de las entidades base.

Esta manera de instanciar objetos crea una **sensación genérica** en estos, no son reconocibles como seres individuales y pertenecen a un grupo de objetos del mismo tipo. Para evitar esto hay juego como **X-COM: UFO Defense** que generan nombres aleatorios para los soldados de manera que el jugador los percibe como seres individuales, con **personalidad**.

#### Contenido con intervención del usuario

Con la llegada de internet el contenido creado y compartido por los usuarios ha aumentado en el contexto de los videojuegos. Pero el problema con esto es que un contenido de calidad requiere de ciertos conocimientos del usuario, un entendimiento técnico mediano que normalmente no tiene. Esto puede una barrera de entrada para muchos usuarios.

El contenido con intervención del usuario pretende reducir esta barrera permitiendo la elección sobre una serie de parámetros, que luego generan contenido de mediante algoritmos. Dando un gran rango de elección al usuario permite que este pueda crear elementos únicos sin la necesidad de conocimientos de la materia.

El ejemplo más representativo es el juego **Spore**, que permite al usuario crear una especie animal y seleccionar su aspecto en distintas etapas de su evolución y es el juego el que genera las animaciones de esta, el mundo adecuado y el contenido de este.

#### Sistemas dinámicos

Sistemas dinámicos como la meteorología, el comportamiento y la inteligencia artificial de un grupo de entidades pueden ser también generados mediante algoritmos.

En el caso particular de la inteligencia artificial esto permite que las entidades puedan responder de manera dinámica a la situación que propone el juego y el mismo jugador.

Un ejemplo de esto es la **Radiant A.I.** desarrollada para el juego de Bethesda, **The Elder Scrolls IV: Oblivion**. Este sistema permite a los NPC (personajes no jugables) generar objetivos sobre la marcha, como “Comer en esta ciudad a las 2:00PM”, o incluso generar misiones para el jugador de manera dinámica.

#### Generación de puzles y tramas/argumentos

A los niveles más simples, la generación procedimental se puede utilizar para generar distintos códigos para las puertas u otros elementos para puzles de manera que el usuario no pueda simplemente consultar una guía que le procure directamente esta información. Pero se puede hacer incluso más, extendiendo los puzles, dividiéndolos en distintas fases, pudiendo utilizar la aleatoriedad para colocar cada parte en un lugar o situación diferente cada vez, como la colocación de llaves que abren cofres o puertas.

La combinación con procesamiento de lenguajes naturales permite crear argumentos e historias de manera dinámica, pero normalmente la complejidad de este proceso hace que no merezca la pena y es más recomendable el uso de argumentos sencillos que dejen al jugador que sea el mismo el que rellene los elementos más complejos de este con el uso de su imaginación.

### Herramientas para la generación de contenido

Cuando hablamos de “sistemas” en la generación de contenido, no podemos referirnos a otra cosa que software que permite establecer una relación entre el proceso de diseño del juego y el programador o diseñador. Algunos programas ayudan solo a la creación de pequeñas porciones del contenido, son **especialistas** de un dominio, otros son más **generales**, pero suelen estar **compuestos** por diversos subsistemas especialistas. Algunos son interactivos, otros no. Algunos son totalmente autónomos, otros necesitan la intervención humana.

Son herramientas que proporcionan al diseñador una manera diferente de intervenir en el proceso creativo del desarrollo del videojuego, pudiendo incluso definir nuevo materiales sobre los que trabajar y que solo podrían haber sido creados mediante procesos programáticos. Al fin y al cabo muchos de estos sistemas tiene como objetivo mejorar el flujo de trabajo del diseñador, como cualquier otra herramienta, pero agregando una automatización de ciertos procesos que proporcionan cierta incertidumbre y variedad de manera similar a como lo haría el propio diseñador. Un ejemplo lo tenemos en sistemas como SketchaWorld, que permite el diseño de ciudades y terrenos, automatizando procesos repetitivos o tediosos como la escultura del terreno y permitiendo al diseñador centrarse en la distribución y alturas de las principales zonas, el posicionamiento de objetos importantes, etc.

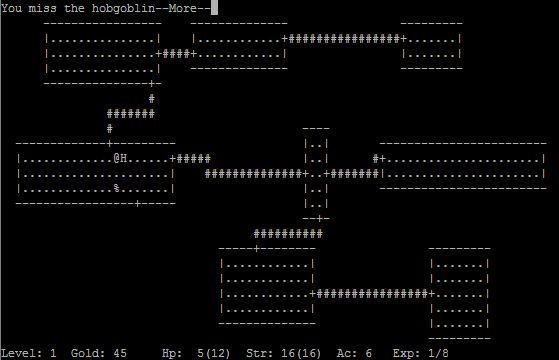
Un diseñador de contenido procedural, refiriéndonos a la persona que interactúa con el sistema generador, no está, como ya hemos explicado, desprovisto totalmente de interactividad con este, y no solo debe ser diseñador del contenido del mismo juego, sino también de las reglas sobre las que el sistema generador se va basar, es el encargado de proporcionar a esas herramientas con el “conocimiento” suficiente para crear contenido de manera coherente con el contexto del juego.

# CAPÍTULO 3. GENERACIÓN DE CONTENIDO EN VIDEOJUEGOS

## Los limitados comienzos

### Rogue (1980)

Al contrario de hoy día en el que tenemos disponibles gigabytes de memoria para mantener los recursos de alta calidad de un videojuego, en la historia temprana de los videojuegos no era aceptable debido a las grandes limitaciones de memoria, por lo que podemos encontrar muchos juegos de aquella época que utilizan de alguna manera alguna técnica de generación procedural de contenido. De entre estos quizá el ejemplo más memorable es **Rogue**.



Este juego utiliza caracteres ASCII para dibujar los todos elementos visuales y aplica algoritmos de generación procedimental de mazmorras y de colocación de objetos. Esto proporciona una jugabilidad casi infinita, donde dos partidas nunca son iguales. De hecho, utiliza el concepto muerte permanente, que simplemente quiere decir que cuando el jugador pierde toda su vida debe reiniciar el juego desde el inicio, no existen los puntos de guardados. Esto se lo puede permitir debido a que la generación de las mazmorras asegura que sea casi imposible encontrar el mismo resultado en dos partidas distintas.

Su popularidad le ha llevado a acuñar el término del subgénero de videojuegos “Rogue-Like”. La influencia de Rogue la hemos podido ver en una gran variedad de videojuegos publicados posteriormente y hasta el día de hoy, como Diablo.

Rogue utiliza un sistema de colocación de habitaciones de tipo aleatorio pero basándose en un grid de 3x3 habitaciones. Dentro de cada casilla del grid se crea una habitación de dimensiones aleatorias (dentro de unos márgenes) y las habitaciones en casillas adyacentes se unen mediante pasillos. La entrada y la salida se escogen antes de crear los pasillos ya que solo tendrán una unión con como máximo con otra habitaciones. Finalmente se colocan objetos y enemigos en las habitaciones de manera aleatoria pero respetando ciertas reglas y con esto se completa la mazmorra.

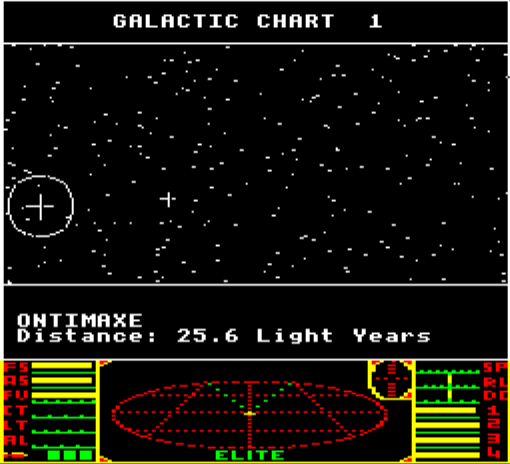
### Elite (1984)

Desarrollado por Acornsoft y publicado en 1984 para la computadora BBC Micro y la familia Electron, Elite es un juego que mezcla la simulación de pilotaje de naves espaciales con el comercio espacial en un entorno estilo sandbox donde el jugador escoge planetas que visitar y en la vecindad de estos puede encontrar piratas a los que enfrentarse, objetos como meteoritos que pueden dañar la nave.

La nave entonces se puede atracar en los planetas y realizar negocios con el dinero de las recompensas obtenido al eliminar piratas y otras actividades.

Además de su excelente implementación de gráficos 3D, de tipo wireframe, que en su momento impresionó porque exprimía al máximo las modestas máquinas donde se publicó, donde realmente destaca en comparación con el resto de juegos de comercio espacial era su universo generado procedimentalmente, incluyendo las posiciones de los planetas, nombres, políticas y descripciones.

Es un juego con una entrada de dificultad un poco dura, pero una vez te haces a los controles es bastante gratificante y su naturaleza procedural proporciona muchas horas de juego.



El éxito de este juego hizo que fuera portado, con una rebaja considerable en el aspecto técnico, a la Commodore 64 y otras plataformas como el Apple II. En Marzo de 2008, la [revista Next Generation](http://en.wikipedia.org/wiki/Next_Generation_%28magazine%29) lo declaró el número 1 de los mejores juegos de la década de los 80, definiéndolo como predecesor de lo que fueron más tarde juegos como la saga Wing Commander o Grand Theft Auto.

### The Sentinel (1986)

Creado por Geoff Crammond y publicado bajo el sello de Firebird en 1986 para varias máquinas como la BBC Micro, Comodore 64, Amstrad CPC, ZX Spectrum, Atari ST, Amiga y PC. Se trata de uno de los primeros juegos que incluían un aspecto 3D con polígonos rellenos.

En este juego tomas el papel de un robot con habilidades telepáticas con las que puedes recoger y colocar objetos del entorno desde una vista en primera persona. El objetivo es el de acabar con el centinela que está en la parte más alta del nivel. Este centinela gira constantemente vigilando el terreno, si te descubre comienza a absorber tu energía. Hay que llegar a la cima y disparar a la base en la que se asienta el centinela, pero el jugador por puede avanzar simplemente utilizando las teclas, sino que tiene que encontrar los robots esparcidos por el terreno que permiten transportarse a ellos, lo cual hace que la energía disminuya, pero se puede recuperar absorbiendo objetos del entorno, como árboles.

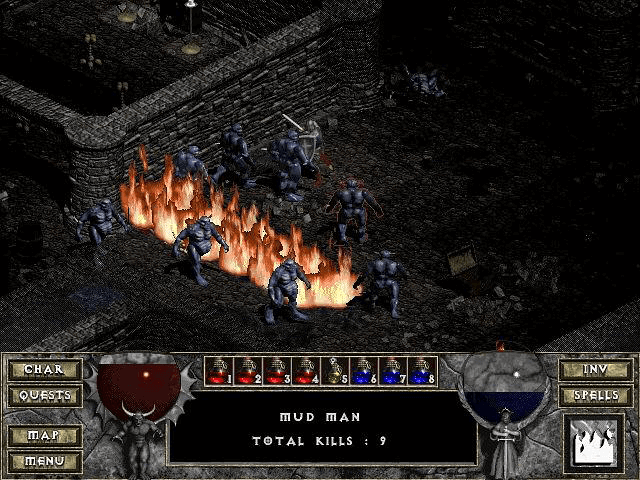


Claramente las limitaciones de memoria de los microcomputadores de 8 bits eran un impedimento para almacenar los 10.000 mundos que presumía tener. Aquí es donde entraba la generación de contenido procedural, donde cada mundo se generaba desde un pequeño paquete de datos: un número de 8 dígitos obtenido al terminar el mundo anterior. Estos dígitos eran diferentes según el éxito del jugador en ese mundo, dependiendo de la cantidad de energía con que terminara el objetivo principal, por lo que se intentaba balancear la dificultad del juego en base a esto. Debido a que no todos los mundos podían ser testeados durante el desarrollo, dejaron la posibilidad de volver a un mundo anterior (usando los números de 8 dígitos) y completándolo con una cantidad diferente de energía.

## Una nueva generación

### Diablo (1996)

Publicado 16 años después de Rogue, **Blizzard Entertainment** condujo el género de los rogue-like a la era moderna con **Diablo**. Se trata de un RPG de acción que implementó elementos procedimentales de una manera tan espléndida que los jugadores comenzaron a pasar cientos de horas jugándolo sin llegar a aburrirse.



En concreto, hay dos aspectos a destacar de la generación procedimental en Diablo:

* Al igual que sus antecesores, como Rogue, la estructura de las mazmorras se generaba de manera aleatoria pero en vez de simples caracteres ASCII este introducía gráficos 2D imitando una perspectiva tridimensional isométrica de gran detalle.
* La generación de ítems también era aleatoria, introduciendo una categoría de colores que clasificaba los objetos por rareza donde las estadísticas de estos se generaban “al vuelo” en el momento de creación.

### Spore (2008)

Nos movemos más de una década para hablar de Spore, un juego estilo god-like, que es el término que se refiere a los juegos en los que no tomas el control de un personaje en el mundo, sino que tomas el papel de creador, como un dios que desde una lugar apartado maneja los hilos de ciertos elementos del juego, moldeando así su evolución.

El objetivo de Spore es la de crear un organismo multicelular que irá evolucionando, sobreviviendo al entorno o muriendo en el intento. El jugador decide, antes de comenzar la partida, las distintas etapas de evolución del organismo, o para decirlo más claro, que características físicas desarrollará en cada etapa. El juego conduce al jugador durante el desarrollo de esta especie en un mundo donde puede interactuar con otras tribus de especies diferentes, luchar bestias salvajes, desarrollar el aspecto social e inteligente de su especie e incluso realizar exploración espacial en etapas futuras.

El juego se desarrolló en Maxis, creadores de Los Sims, entre otros, con el diseñador principal Will Wright y publicado por Electronic Arts. Fue un éxito de crítica.



Spore destaca por su uso masivo de la generación de contenido dinámico utilizando técnicas procedurales. No solo los **mundos** se crean de esta manera, sino que los movimientos de las **criaturas** en sí son **animadas procedimentalmente**, aún dando al jugador grandes posibilidades de creación, pudiendo agregar o quitar miembros a su gusto. Para ello el creador de especies proporciona una gran variedad de miembros predefinidos que se pueden colocar como se desee, es el mismo jugador el que decide qué puede ser más útil a nivel evolutivo, o simplemente crear algo totalmente bizarro y comprobar cómo se desarrolla en el entorno.

Por otro lado incluso la **música** de **Brian Eno** es creada usando un compositor algorítmico, lo que se conoce como [**música generativa**](http://en.wikipedia.org/wiki/Generative_music), término popularizado por él mismo. De esta manera la música se puede adaptar sobre la marcha a la gran variedad de situaciones.

## La era independiente

### Spelunky (2008)

Creado por **Derek Yu** y publicado como freeware para sistemas Windows, se trata de un juego independiente de exploración de cuevas 2D al puro estilo Indiana Jones. El objetivo es el de llegar al final de cada nivel evitando trampas y criaturas, al mismo tiempo que se van recogiendo tesoros y salvando princesas perdidas. Sigue la premisa del clásico juego Spelunker de 1983 pero en un mundo creado proceduralmente en cada nueva partida.

Se hizo un remake en 2013 pero aún se puede encontrar la versión original freeware incluyendo el código fuente para **Gamemaker**.



Para generar cada pantalla o nivel se crea una rejilla y se divide en secciones o habitaciones formadas por varios tiles. A cada habitación se le asigna un tipo o rol, por ejemplo:

*0 - habitación que en principio no tiene salida y no forma parte del camino solución.*

*1 - habitación que tiene salida a la izquierda y derecha.*

*2 - habitación con salida izquierda, derecha e inferior. Si encima se une otra del mismo tipo entonces esta también tendrá salida superior.*

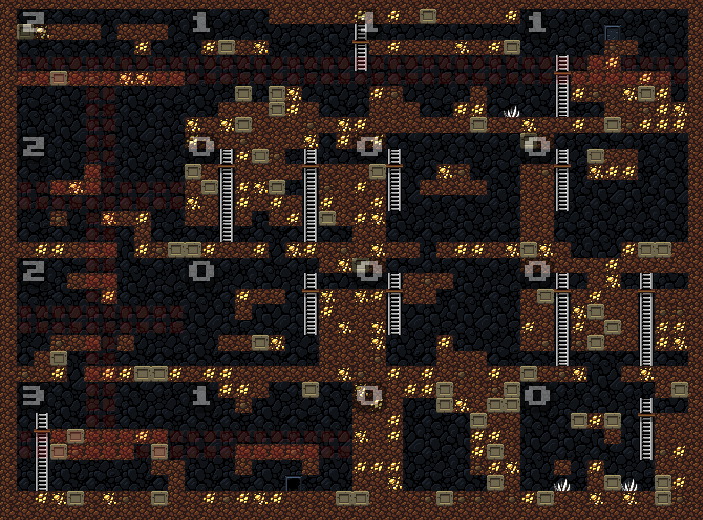
*3 - habitación con salidas izquierda, derecha y superior.*

Primero hay que colocar una habitación de inicio en la zona superior de la rejilla de habitaciones, generalmente del tipo 1 o 2.

Cada vez que se coloca una habitación por defecto es de tipo 1. Entonces de manera aleatoria (utilizando distribución uniforme de 1 a 5) se escoge la siguiente dirección. Para 1 o 2 se mueve a la izquierda, para 3 o 4 a la derecha. En estos casos ya tenemos la habitación 1 lista con salidas a estos dos lados. Ahora, cuando obtenemos un 5 entonces hay que cambiar la habitación de tipo 1 por otra de tipo 2 que siempre tiene una salida inferior.

Una vez nos movemos a la siguiente habitación realizamos lo mismo, pero esta vez miramos primero si la anterior era tipo 2 y nos hemos movido hacia abajo, caso en el que escogemos por defecto una habitación de tipo 2 o 3.

Cuando llegamos a la última fila y la siguiente dirección que obtenemos es hacia abajo, entonces podemos colocar la salida y ya tenemos el camino completo. El resto de habitaciones que queden vacías se rellenan con un tipo 0.



La disposición interna de cada habitación se basa es una mezcla de habitaciones predeterminadas con métodos probabilísticos para determinar el tipo de determinados bloques. Es decir, comenzamos escogiendo una habitación de entre unas 16 predeterminadas, siempre que estas cumplan los requisitos según el tipo, como ya hemos visto. Luego, para esta disposición escogida tenemos ciertos bloques que están marcados para que sean calculados con algún tipo de probabilidad, por ejemplo, si escogemos una habitación de tipo 0, que es una habitación que está en principio cerrada, habría algunos bloques en los bordes de esta con un 30% de probabilidad de que sean movibles, con lo que desde una habitación adyacente se podría acceder moviendo este bloque. O puede que en los bloques de la zona inferior alguno tenga la probabilidad de convertirse en pinchos.

Finalmente dependiendo de las habitaciones adyacentes que se unan podemos determinar, también por métodos probabilísticos, que tipo de criaturas o trampas incluir en cada habitación.

### Minecraft (2009)

Minecraft es un juego de estilo sandbox creado originalmente por Markus “Notch” Persson que más tarde formó la compañía Mojang. Este juego brinda al usuario con unos aspectos de creatividad y construcción sobre un mundo compuesto por cubos 3D con texturas que le da un aspecto simple. Este mundo permite una infinidad de tareas a realizar, como la exploración de nuevos espacios y cuevas, minería, construcción o combate. Pero todas estas mecánicas se presentan de una manera simple y abierta para que el mismo usuario sea capaz de construir sus propias aventuras en los diferentes modos de juego.

Es posiblemente, junto a Diablo, el juego más popular de la lista, pero el caso de Minecraft destaca más en el sentido de generación procedural de mundos ya que es algo mucho más claro y una gran parte de lo que le aporta la diversión.

El jugador comienza creado un nuevo mundo, este se genera a partir de una semilla, que es una cadena de caracteres que se puede introducir manualmente o dejar al mismo juego que cree una aleatoria. A partir de esta semilla se crear un mundo completo con campos, bosques, montañas, mares, cuevas, animales y demás elementos naturales. Además podemos encontrar lo que se llaman biomas, que representan zonas de distinto clima, como pueden ser zonas desérticas, nevadas, montañosas, praderas soleadas...



Cuando un usuario introduce una semilla al crear una partida obtiene siempre exactamente el mismo mundo en su estado original. Este es uno de sus puntos fuertes, puesto que la comunidad de usuarios comparte semillas y coordenadas en ese mundo donde se pueden encontrar formaciones interesantes y extrañas.

Por entrar un poco más en detalles, como el propio Notch explica en [este artículo](http://notch.tumblr.com/post/3746989361/terrain-generation-part-1), el mundo de Minecraft no es realmente infinito, a pesar de que no haya límites propios. Esto es debido a que el mundo se renderiza en trozos (chunks) de 16x16x128 bloques. El offset o separación entre estos bloques se basa en enteros de 32 bits en un rango de -2 billones a +2 billones. Al pasar ese rango el juego comienza a sobrescribir los antiguos chunks con nuevos y cuando pasas cierta distancia, los bloques que usan enteros para su posicion comienzan a actuar de manera extraña e inestable.

En cuanto a la generación del terreno en sí, al inicio del desarrollo se hacía uso de un mapa de alturas 2D utilizando el **ruido de Perlin** para obtener la “forma” del mundo. De hecho se usaban varios de estos mapas para obtener detalles como la elevación, la “rugosidad”, y otro para detalles locales concretos (como biomas). Pero aunque se trataba de un método sencillo y muy rápido, el hecho de ser 2D tenía la desventaja de generar formaciones simples y aburridas, por ejemplo no se podían crear salientes de montañas, donde en una misma zona tendríamos dos alturas diferentes.



Así que en vez de mapas 2D, se comenzó a utilizar algo así como ruido de Perlin 3D, esto quiere decir que, en vez de tratar el ruido como simple altura sobre el terreno, lo que se tenía en cuenta era la densidad del ruido. De este modo cualquier valor por debajo de 0 sería aire, y por encima de 0 tendríamos terreno.

### The Binding of Isaac (2011)

Otro juego que voy a nombrar brevemente es **The Binding of Isaac** diseñado por **Edmund McMillen**, conocido por el exitoso **Super Meat Boy**. Se trata de un juego al más puro estilo **The Legend Of Zelda** en sus versiones clásicas 2D, donde encontramos mazmorras con habitaciones contiguas e independientes, algunas con roles específicos como las que guardan un ítem especial tipo mapa, llave o jefe.



El proceso de creación de las habitaciones es incluso más simple y trivial que en otros juegos como el clásico Rogue, pero el verdadero interés se encuentra en la gran variedad y probabilidad de encontrar los diferentes tipos de objetos, que hacen que una partida pueda resultar más complicadas que otras.

Mi interés por este juego reside en que es similar a lo que he desarrollado para este proyecto, donde tenemos una jugabilidad estilo Zelda, con un estilo visual similar, pero en mi caso incluyo una generación o distribución de habitaciones y pasillos más variable, como veíamos en Rogue.

### No Man’s Sky (2015)

Para finalizar con estos análisis vamos a echar un vistazo a lo que está por venir, más en concreto al prometedor **No Man’s Sky**.

**No Man’s Sky** se presentó por primera vez en los premios Spike's VGX de Diciembre de 2013 como, en principio, un exclusivo para Playstation 4. Está siendo desarrollado por los británicos Hello Games, conocidos por Joe Danger. Pero esta nueva creación, que tiene como fecha de lanzamiento Mayo de 2015, es algo completamente distinto a Joe Danger.

No Man’s Sky se presenta como un juego de aventuras de ciencia ficción de tipo sandbox donde el jugador podrá explorar una gran variedad de mundos, llenos de vida, criaturas, océanos profundos. Además incluye otro aspecto de exploración y batallas espaciales.

En cierta manera algo como lo que hemos visto con Elite (1984) pero elevado a la máxima potencia, uniendo elementos de generación de criaturas y otro tipo de vida, todo esto utilizando técnicas procedurales tanto para la vida animal como para los mismos planetas y distribución de estos.



En este caso no se trata de que cada nueva partida sea diferente, sino de un **mundo persistente** donde todos los jugadores juegan online pero aparecen en lugares tan apartados que será raro encontrarse con otros jugadores a no ser que estos compartan una posición.

Para poner en perspectiva la inmensidad del universo que quieren presentar, podremos encontrar la cifra exacta de **18.446.744.073.709.551.616 planetas** y para pasar en cada uno un segundo harían falta **585 mil millones** de años. Sin embargo, la inmensa mayoría, similar a un universo real, no serán planetas muy interesantes y solo un 10% de estos tendrán vida, lo cual ya es un número vertiginoso de planetas.



De momento no he podido encontrar mucho de las técnicas que utilizan para generar todo este contenido, solo he podido ver que utilizan algo similar a la creación de especies en Spore pero de manera automática y simplemente han [comentado en alguna entrevista](https://www.youtube.com/watch?v=9NWpdyQXzHw) que realmente no utilizan nada demasiado complicado para generar tal inmensidad de mundos, probablemente sea una conjunción de muchas de las técnicas que ya hemos visto en otros juegos.

# CAPÍTULO 4. ALGORITMOS

Una vez entendido el concepto de generación procedimental de contenido y su implicación en el mundo de los videojuegos, nos adentramos en los detalles de las técnicas y algoritmos que permiten producir contenido en la generación de sistemas de cuevas y mazmorras.

## Clasificación

Antes hemos expuesto una clasificación de los tipos de generación de contenido, y ahora es necesario presentar otra clasificación de los algoritmos involucrados en estas técnicas.

Estos se pueden categorizar atendiendo a dos conceptos: cómo generan y en qué filosofía se basan.

### Cómo generan. Secuencial frente a mapeado

Por **generación secuencial** entendemos a un grupo de técnicas que generan contenido con una estructura lineal intrínseca. El ejemplo más claro es la generación de una secuencia de Fibonacci, donde comenzamos con una semilla y el resto de números son generados de manera secuencial a partir del número anterior.

**0, 1** (semillas) -> **1** (0 + 1) -> **2** (1 + 1) -> **3** (1 + 2)...

La implementación más simple de un generador de números pseudo-aleatorios es probablemente el **generador secuencial congruente**, que básicamente se basa en la idea de multiplicar el último número con un factor **a**, sumar una constante **c** y obtener el módulo sobre **m**.

**Xn+1 = (aXn + c) mod m**

Donde **X0** será la semilla inicial.

El uso de números **pseudo-aleatorios** está presente prácticamente en todos los algoritmos para la generación de contenido, pero para algunos es especialmente determinante, como en el de **autómata celular** que vamos a estudiar, donde la semilla y resultado inicial determinará los siguientes niveles de generación, y donde cada paso depende del resultado anterior, en forma de secuencia.

Por otro lado podemos diferenciar la generación que se realiza sobre mapas 2D o 3D, de tipo **mapeado**. En este caso estamos hablando de generación a distintos niveles, como por ejemplo la que vemos al utilizar mapas de alturas mediante ruido de Perlin, generando distintas octavas de ruido en distintas capas que finalmente forman un mapa único.

No podemos decir que secuencial y mapeado sean conceptos excluyentes y generalmente un mapeado requiere una generación secuencial de las distintas capas que lo componen.

### En qué se basan. Teleológico frente a Ontogénico

El acercamiento **teleológico** a la generación de contenido hace referencia a la imitación de los procesos físicos reales que siguen los sucesos naturales. Por ejemplo, si se quiere generar un terreno con valles y montañas, sería el proceso de imitar el movimiento de placas tectónicas y luego utilizar un algoritmo como el **Rain Drop Algorithm**, que imita la caída de gotas de lluvia para erosionar el terreno, aplicándolo sobre un mapa de alturas.

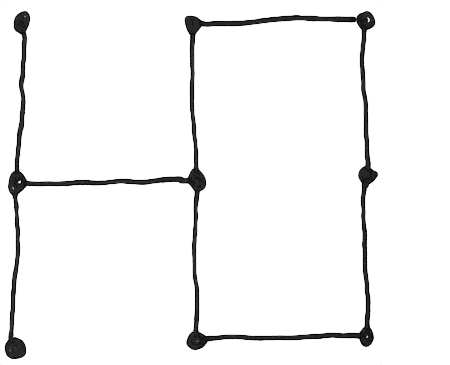
Por otro lado el método **ontogénico** utiliza otras técnicas que no tienen nada de similitud con los procesos físicos naturales, como cuando se hace uso de ruido de **Perlin** para generar terreno.

## Introduciendo algunos conceptos

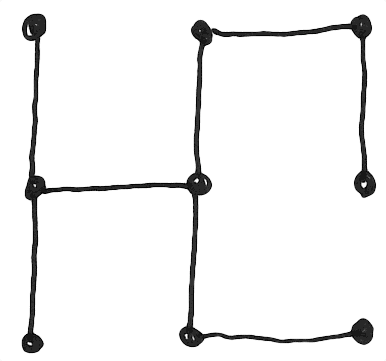
Vamos a estudiar un poco más a fondo algunos de los algoritmos para la generación procedural de mazmorras, pero tenemos que empezar por lo básico, un laberinto.

Existen decenas de algoritmos para la resolución y creación de mazmorras de manera procedimental, y algunos de los factores importantes que debemos tener en cuenta para escoger uno son el coste temporal, la variedad de resultados que nos permite obtener, la complejidad del mismo algoritmo y la personalización que podamos realizar sobre este para adaptarlo a nuestro proyecto.

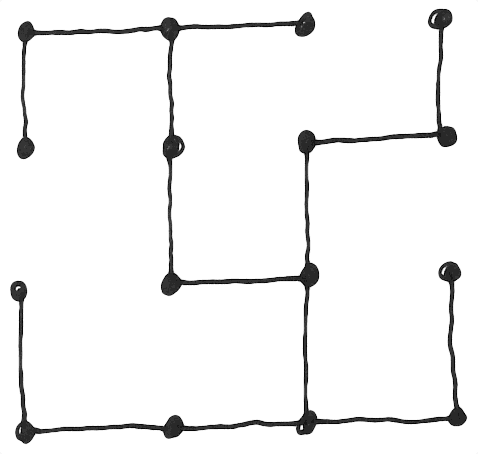
Antes de pasar al algoritmo conocido como Growing Tree debemos repasar rápidamente ciertos conceptos básicos. Por un lado tenemos los **vértices** o **nodos**, que en el caso que nos compete, pueden ser llamados **celdas**. Estos nodos se unen mediante **aristas**, que es básicamente una línea. Una colección de vértices y aristas es lo que llamamos **grafo**. Si desde una nodo podemos alcanzar cualquier otro nodo del grafo siguiente las aristas entonces decimos que es un **grafo conexo**.



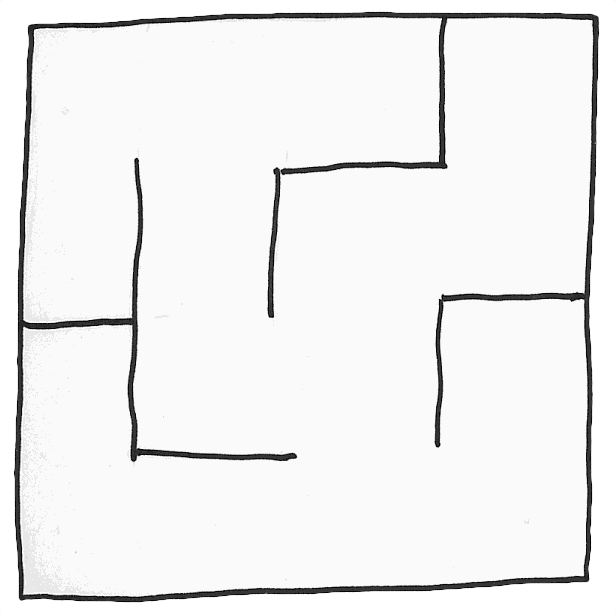
Si eliminamos los ciclos del grafo, obtenemos un grafo acíclico, una cuando es grafo acíclico es conexo, lo que tenemos es llamado un **árbol**.



Un grafo puede estar compuesto por varios árboles, pero existe uno que comprende todos los nodos de un grafo, tenemos un **árbol de expansión (Spanning Tree)**.



Y los árboles de expansión son la misma esencia de la estructura de los algoritmos de generación de laberintos.



### Aldous-Broder

Dos investigadores, **D. Aldous y A. Broder**, trabajando independientemente, estaban estudiando los árboles de expansión cuando diseñaron el siguiente algoritmo:

1. Escoger un vértice cualquiera.

2. Escoger otro vértice aleatorio entre los vecinos de este. Si el nodo no ha sido visitado con anterioridad, moverse a este y agregarlo, junto a la arista, al árbol de expansión.

3. Repetir el paso 2 hasta que todos los vértices hayan sido visitados.

Un algoritmo extremadamente simple que selecciona cualquiera de todos los posibles árboles de expansión del grafo con la misma probabilidad. También hay que decir que es una técnica muy ineficiente, ya que su naturaleza aleatoria a la hora de escoger el nodo hace que se puedan volver a visitar los mismos vértices una y otra vez.

Posteriormente este método fue mejorado por otros como el algoritmo de Wilson, entre otros, pero vamos a pasar directamente al que nos interesa estudiar, que entra dentro de los algoritmos que hacen uso de la técnica de escoger un nodo aleatoriamente en cada paso, también conocido como **Drunken Walk**.

### Growing Tree

Entre los algoritmos de generación de laberintos, el llamado Growing Tree es quizás el más personalizable. La premisa básica es la de escoger un nodo del grafo aleatoriamente y agregarlo a una lista de "celdas activas". En cada paso posterior miramos a uno de los nodos de la lista y agregamos uno de sus vecinos **no visitados**. Si el nodo no tiene más vecinos sin visitar, lo quitamos de la lista y probamos con otro nodo. El proceso termina cuando la lista se queda vacía.

Debemos que tener en cuenta que cada celda o nodo tiene 4 bordes que tocan con otros nodos o con el exterior de la mazmorra, por lo que cada nodo deberá ser visitado 4 veces. Esto no se produce en la misma pasada, sino que al introducirlos en la lista y luego hacer el backtracking nos vamos asegurando que ese nodo tiene aún bordes libres, en caso contrario lo podemos sacar y lo damos como cerrado.

Cuando estamos comprobando un vecino debemos determinar si se ha de crear un pared según el caso. Para los bordes que den al exterior de los límites de la mazmorra simplemente creamos la pared, cuando nos movemos a una nueva celda no visitada antes, entonces simplemente creamos pasillo entre estas, pero cuando nos topamos con otro nodo que ya está en la lista de activos entonces creamos una pared en ese borde y nos movemos a otro de los vecinos. Este mismo proceso nos permite que siempre se pueda alcanzar cualquier celda desde otra.

Un aspecto interesante es como el uso de distintas heurísticas para seleccionar un nodo de la lista de activos cambia el comportamiento de este algoritmo. Por ejemplo, si escogemos el nodo más reciente, el último que se agregó a la lista, obtenemos un comportamiento de pila recursiva. Este comportamiento es el mismo que encontramos en otro algoritmo llamado **Recursive Backtracker**. Si escogemos un nodo aleatoriamente, entonces tenemos un comportamiento del estilo del algoritmo de **Prim**.

En la implementación que he realizado sobre Unity podemos ver el comportamiento al usar el nodo más reciente agregado a la lista.

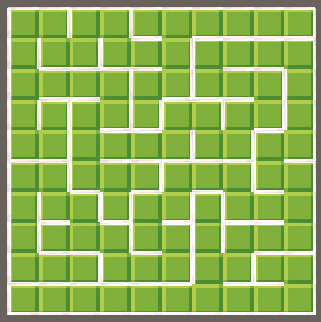
|  |
| --- |
| **CreateDungeon**() {  *// Inicializa el laberinto de 10x10*  size.x = 10;  size.z = 10;  cells = **new** Cell[size.x, size.z];  *// Lista de celdas activas*  List<Cell> activeCells = **new** List<Cell>();  *// Selecciona la primera celda al azar y la introduce en la lista de activas*  Vector2i randomCoor = **new** Vector2i(Random.Range(0, size.x), Random.Range(0, size.z));  activeCells.Add(CreateCell(randomCoor));  *// Mientras existan celdas activas*  **while** (activeCells.Count > 0) {  **DoNextStep**(activeCells);  } }  **DoNextStep**(List<Cell> activeCells) {  *// Obtiene la siguiente celda de la lista de activos*  *// si se escoge de manera aleatoria el comportamiento es de tipo Prim*  int currentIndex = GetNextIndex(activeCells);  Cell currentCell = activeCells[currentIndex];  *// Comprueba si la celda ya ha sido inicializada en sus 4 lados*  *// si es asi, la elimina de la lista de activos*  **if** (currentCell.IsInitialized()) {  activeCells.RemoveAt(currentIndex);  **return**;  }  *// En caso contrario escoge aleatoriamente un lado no inicializado*  t\_Direction direction = currentCell.RandomUninitializedDirection();  Vector2i coordinates = currentCell.coordinates + direction.ToIntVector2();  *// Si esta dentro de los limites del mapa*  **if** (ContainsCoordinates(coordinates)) {  *// Recoge la celda adyacente en el lado seleccionado*  Cell neighbor = GetCell(coordinates);  *// Si el vecino no existe entonces crea una celda y la agrega a la lista de activos*  *// El camino entre estas dos celdas sera un pasillo*  **if** (neighbor == **null**) {  neighbor = CreateCell(coordinates);  CreatePassage(currentCell, neighbor, direction);  activeCells.Add(neighbor);  }  *// Si la celda adyacente ya existia entonces crea una pared entre las dos*  **else** {  CreateWall(currentCell, neighbor, direction);  }  }  **else** { *// Borde del mapa, crea siempre una pared*  CreateWall(currentCell, **null**, direction);  } } |

Para entenderlo mejor vamos a ver los primeros pasos de manera visual.

|  |  |  |
| --- | --- | --- |
| Celda en lista de activas. | Celda finalizada. Sus 4 lados han sido inicializados. | Celda actual. El cuadro de color cian indica el lado sobre el que se está actuando. |

|  |  |  |  |
| --- | --- | --- | --- |
| 1. Escoge celda aleatoria inicial y la incluye en lista de activas. | 2. Escoge un lado de manera aleatoria y agrega la siguiente celda. | 3. Escoge un lado de nuevo y como es límite de mapa pone una pared. | 4. Continúa escogiendo otro lado. En este caso no visitado, por lo que crea un pasillo. |
| 5. Después de varios pasos intenta visitar un lado que ya tiene celda y crea una pared entre las dos. | 6. Continúa creando paredes entre las celdas ya visitadas hasta que llega a un rincón y completa una celda por los 4 lados. | 7. Finaliza la celda y comienza a hacer backtracking desapilando de las celdas activas. | 8. Va cerrando celdas activas hasta que llega a una con lados sin procesar y continúa apilando nuevas casillas. |

Finalmente obtenemos un laberinto completo donde cada celda puede ser alcanzada desde cualquier otra, solo habría que situar las entradas y salidas y estaría terminado.



Modificando este algoritmo incluso podríamos hacer que se generen habitaciones en vez de pasillos. El uso de este método similar al **Recursive Backtracker** nos permite determinar y marcar cuando una habitación se ha completado, momento en el que se comienzan a desapilar nodos y retoma el camino hacia la creación de otra habitación. Pero la cosa se complica si queremos tener habitaciones y pasillos en ese mismo mapa, por ello este algoritmo no es de mucha utilidad más allá de la generación de laberintos.

### Conclusión

El algoritmo de Growing Tree es interesante en el sentido en que se puede personalizar y ampliarlo para generar algo más que laberintos, ensanchando los pasillos o uniendo los adyacentes para generar habitaciones y obtener una estructura más similar a lo que busco para el videojuego que quiero crear, pero como veremos hay otros algoritmos que son más adecuados y rápidos para generar mapas grandes.

Podemos decir finalmente que este algoritmo tiene sus **ventajas**:

* Está basado en técnicas muy simples.
* Es flexible y puede comportarse como otros algoritmos según su implementación.

Pero también hay algunas **desventajas**:

* No es el algoritmo más rápido de por sí, y habría que utilizar una mezcla con árboles de partición (BSP, Quadtree) para generar laberintos realmente grandes.
* Generalmente solo se puede utilizar para la creación laberintos, otro tipo de estructuras como mazmorras o cuevas están fuera del alcance o requieren una gran cantidad de **personalización extra**.

## El autómata celular

El segundo tipo de algoritmo que vamos a estudiar entra dentro de los algoritmos utilizados para representar sistemas naturales, como cuevas, bosques o manadas de animales.

El concepto de **autómata celular** fue presentado originalmente en los **años 40** por Stanislaw Ulam y John Von Neumann cuando trabajaban en el laboratorio nacional de Los Alamos. Fue estudiado ocasionalmente durante las siguientes dos décadas pero no fue hasta los 70 en que **John Horton Conway** creó el “**Juego de la vida” (**[**Conway’s Game Of Life**](http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)**)** que despertó el interés en el entorno académico.

### Conway’s Game Of Life

Este juego consiste en una serie de puntos llamados células que evolucionan con el tiempo basándose en la interacción con sus vecinas. Si una célula tiene **menos de** **dos** vecinas, entonces **muere**, debido a la falta de población, con **dos o tres** vecinas se mantiene viva durante esa generación, si tiene **más de tres** entonces **muere** por sobrepoblación. Si una célula muerta (un espacio vacío) tiene exactamente tres vecinos, entonces esta se convierte en una célula viva, representando un proceso de reproducción.

Antes de comenzar el juego se establecen ciertas condiciones para la situación inicial, pero a partir de ahí el jugador no tiene más interacción con este, el juego evoluciona por sí solo.

|  |  |  |  |
| --- | --- | --- | --- |
|  | C:\Users\Ruben\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tmp-1.gif | C:\Users\Ruben\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tmp-2.gif | C:\Users\Ruben\AppData\Local\Microsoft\Windows\INetCache\Content.Word\tmp-3.gif |

Durante los **años 80 Stephen Wolfram** se ocupó del estudio del autómata celular sobre una sola dimensión, y en 2002 publicó su libro “**Un nuevo tipo de ciencia**”, donde argumenta que este tipo de métodos pueden ser utilizados en otros campos de la ciencia como la criptografía.

### Autómata celular en generación de cuevas

**Jim Badcock** publicó hace unos años su aplicación de los autómatas celulares en la creación procedimental de cuevas, mostrando su potencial para la representación de sistemas naturales, al menos como base, ya que el proceso requiere un refinamiento posterior.

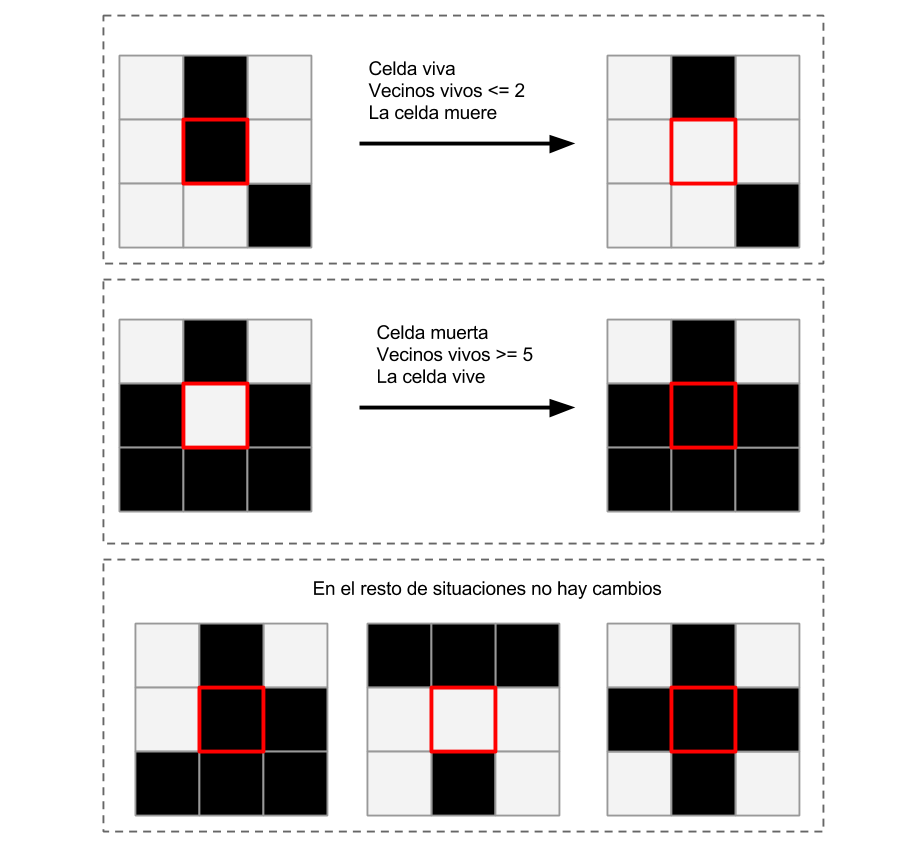
El **proceso** consiste en lo siguiente:

1. Generamos un lienzo inicial utilizando la técnica de **ruido blanco**, es decir, dividimos el mapa en celdas y las recorremos decidiendo aleatoriamente si esta se rellena o se queda vacía. Como no queremos completa aleatoriedad establecemos un porcentaje que indique la probabilidad de que esta se convierta en pared. Por ejemplo, decimos que la probabilidad es del 40%, generamos un número aleatorio entre 0 y 100 y si este es inferior o igual al 40 entonces generamos una pared.

|  |
| --- |
| **RandomByProbability**(int probability) {  int randomN = Random.Range(1, 101); *// 1 - 100*  **if** (probability >= randomN) {  **return** 1;  }  **return** 0; } **FillRandom**() {  *// 0 = vacio, 1 = pared*  **for** (int r = 0; r < height; r++) { *// c = column, r = row*  **for** (int c = 0; c < width; c++) {  *// Los bordes son pared*  **if** (c == 0 || c == width - 1 || r == 0 || r == height - 1) {  grid[c, r] = 1;  }  **else** {  *// Aleatorio entre 0 y 100 con x% de probabilidad de pared*  grid[c, r] = **RandomByProbability**(wallProbability);  }  }  } } |

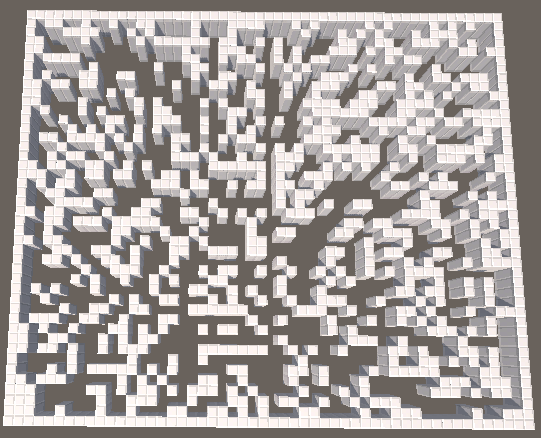
1. Recorremos el mapa de nuevo ahora aplicando las reglas del autómata celular:
   1. Si la celda es pared y tiene menos de 3 vecinas de tipo pared, entonces se convierte en espacio vacío.
   2. Si la celda está vacía y tiene 5 o más vecinas de tipo pared, entonces esta también se convierte en pared.

|  |
| --- |
| **CreateDungeon**(){  *// Crear el mapa inicial de ruido usando la probabilidad de paredes*  **FillRandom**();  int r, c; *// r = fila, c = columna*  **for** (int t = 0; t < passes; t++) {  r = 0;  **while** (r < height) {  c = 0;  **while** (c < width) {  *// Obtenemos numero de paredes adyacentes a esta posicion*  int numWalls = **GetAdyacentWalls**(c, r);  *// La celda actual es pared*  **if** (grid[c, r] == 1) {  *// Si paredes adyacentes <= 2, se vacia*  **if** (numWalls <= 2) {  grid[c, r] = 0;  }  }  **else** {*// Es vacio, si >= 5 paredes adyacentes, se convierte en pared*  **if** (numWalls >= 5) {  grid[c, r] = 1;  }  }  c++;  }  r++;  }  } } |

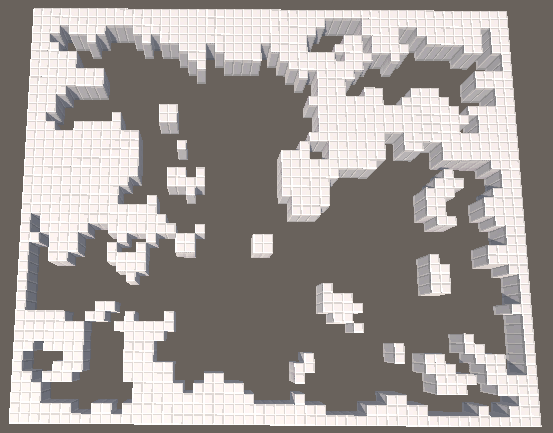


1. Cuando hemos recorrido todas las celdas el algoritmo termina, aunque es posible realizar más iteraciones, refinando el resultado para eliminar celdas solitarias o que sobresalen.

Vamos a ver ejemplos generados a partir de la implementación que he realizado sobre Unity:

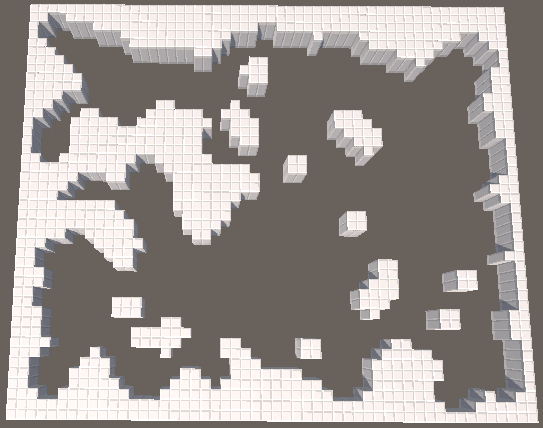


Tablero inicial generado aleatoriamente con una probabilidad de paredes del 35%.

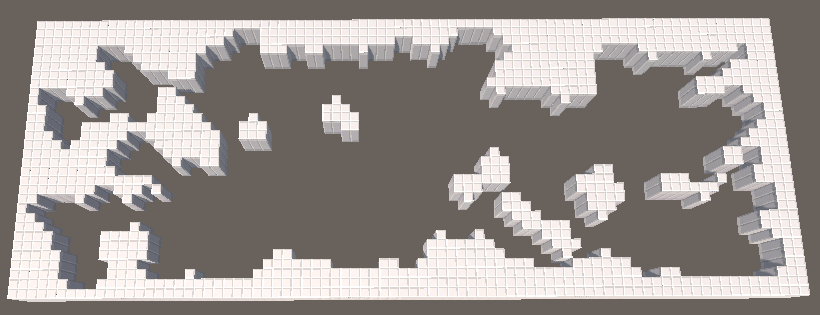


Este es el resultado al terminar el algoritmo con **un solo pase**, podemos ver que se generan zonas o espacios abiertos distinguibles unos de otros. También observamos uno de los problemas del algoritmo, y es que es poco consistente, y muy frecuentemente obtenemos zonas inconexas que deberemos procesar posteriormente para unirlas al sistema principal de habitaciones. A veces estas zonas desconectadas están separadas simplemente por distancias de 1 o 2 celdas, pero como podemos ver en este mismo ejemplo, en la zona superior derecha tenemos una zona relativamente grande con una separación de la zona principal de al menos 3 celdas, por lo que resultaría difícil determinar programáticamente como esa zona debería unirse.

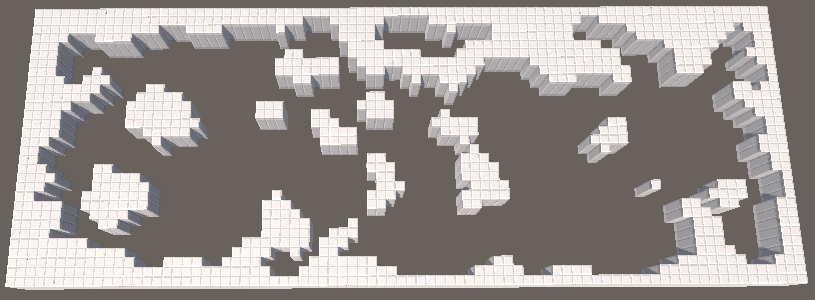
Por otro lado esto puede ser interesante para juego como **Minecraft**, donde podemos encontrar en cuevas, con mucha frecuencia, zonas totalmente aisladas a las que se debe acceder usando el pico o la pala. En ese caso específico este algoritmo parece actuar correctamente.



Aquí tenemos otro ejemplo, esta vez con **dos pases**. Ahora hemos conseguido que todos los espacios estén conectados de alguna manera. También se han eliminado celdas sueltas de una unidad al realizar este segundo pase, lo cual puede ser beneficioso para eliminar ruido innecesario pero al mismo tiempo perjudicial porque le quita un poco de la aleatoriedad que podemos encontrar en las cuevas naturales. Otro problema de realizar este segundo pase es que perdemos la distinción entre las distintas secciones o habitaciones de la cueva, por lo que nos queda un espacio abierto con algunos rincones más cerrados.



Los mismo parámetros pero con diferentes dimensiones. Conseguimos conectar todos los espacios pero el centro está demasiado despoblado.



Si subimos la probabilidad de paredes al 40% y al realizar dos pases obtenemos un resultado mejor.

### Semillas

Debido a que la generación de ruido al inicio se hace aleatoriamente utilizando una semilla, es posible especificar esta y los parámetros de probabilidad de paredes e iteraciones para obtener siempre el mismo resultado. En los tests incluidos con el proyecto podemos comprobar esto, por ejemplo, si introducimos los siguientes parámetros:

Width = 75

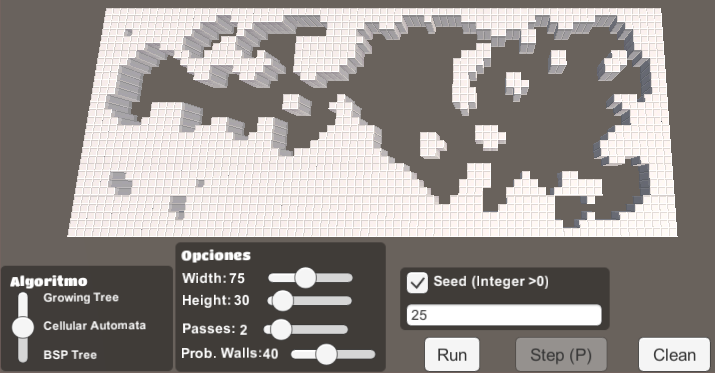
Height = 30

Passes = 2 (Iteraciones de autómata que se van a realizar)

Prob. Walls = 40 (Probabilidad inicial de que se creen paredes)

Seed = 25

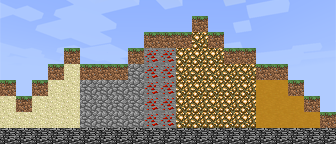
El resultado debería verse así:



### Conclusión

Haciendo varias pruebas puedo llegar a la conclusión de que los mejores resultados los obtengo con una probabilidad de obtener paredes del 35% al 40% y uno o dos pases según este porcentaje, cuanto más se acerca al 40% mejores resultados obtendremos con dos pases que solo con uno.

Uno de los principales problemas con este algoritmo, como ya hemos visto, es el poco control sobre el resultado final, demasiadas aleatoriedad en la creación de los distintos espacios que componen la cueva. Esto dificulta también la decoración de esta, ya que es más difícil identificar cada parte de la cueva como esquinas, cámaras aisladas, tipo de materiales de roca, etc., por lo que tendríamos que determinar el tipo de material o bloque según la profundidad de la cueva, ya sea en vertical o en horizontal. Un ejemplo de esto es Minecraft, que distribuye los bloques basándose en la profundidad de bloques desde una capa base y también en la distribución horizontal en zonas que dan al exterior:



En cualquier caso este tipo de estructuras no son las que quiero para juego que voy a desarrollar, ya que busco una mejor distinción entre habitaciones y pasillo, estructuras del estilo de templos, hechos en parte por humanos. Pero no voy a descartar completamente esta técnica, ya que veremos más adelante que es útil para refinar y darle un toque más natural a los resultados del siguiente algoritmo que vamos a estudiar.

## Generación de mazmorras. Posicionamiento aleatorio.

Pasando a los generadores de **mazmorras** más específicamente del estilo que podemos encontrar en **Rogue**, es más simple es el que se basa en la colocación de habitaciones sobre un plano, de manera aleatoria, y dejando al programador la reorganización y conexión de estas.

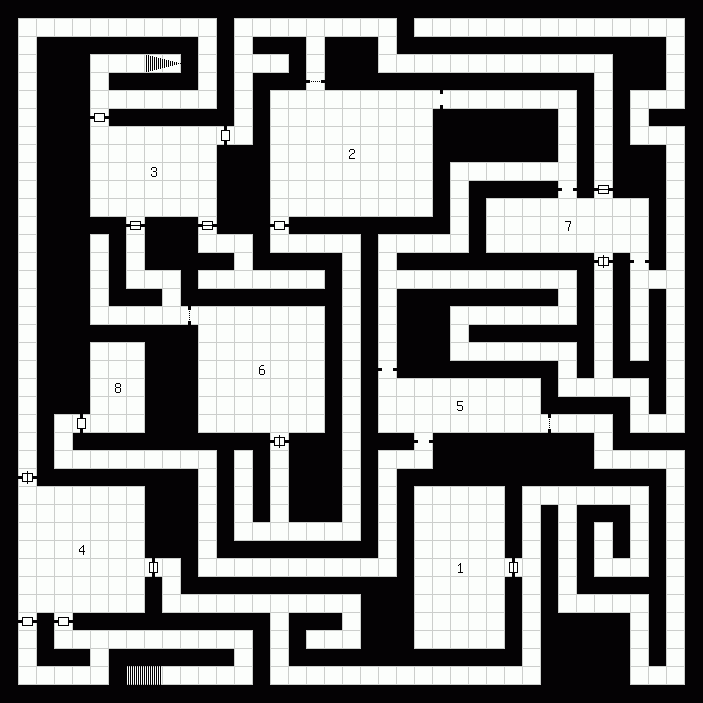
Se trata de un algoritmo que tiene una descripción bastante sencilla pero cuya implementación requiere mucha más personalización por parte del programador. Estos son los pasos que generalmente se deben seguir en este tipo de implementación:

*1. Inicializar el mapa llenándolo con tiles.*

*2. Crear un total de* ***N*** *habitaciones en posiciones y con dimensiones aleatorias. Algunas se van a solapar, pero esto se tratará en pasos posteriores.*

*3. Crear pasillos para unir pares de habitaciones. Se deben de tener en cuenta las habitaciones que se usarán como entrada y salida para no unirlas directamente. Los pasillos se pueden crear de forma directa o escogiendo direcciones aleatoriamente hasta que llegue a su destino, dándole así un aspecto irregular. Para esto es recomendable usar la técnica del* ***Drunken Walk****.*

*4. Realizar una limpieza a habitaciones solapadas, ya sea uniéndolas en una sola o limpiando los tiles que sobresalgan alrededor de estas.*



Como hemos podido comprobar, este algoritmo requiere mucha intervención del programador para ajustar los resultados, y aun así encontramos problemas en la creación de los pasillos que acaban en rincones sin salida, con que habría que realiza otro pase para eliminar estas irregularidades, más trabajo extra que no forma parte del algoritmo y que se deja en manos de la persona que lo implementa. Los solapamientos de los pasillos se pueden arreglar directamente uniéndolos.

Pero tampoco tenemos que olvidarnos de la segunda parte del proceso de generación, en la que se puebla la mazmorra con objetos y enemigos y este método nos proporciona de por sí una estructura adecuada para esto, con lo que se debería implementar un método para generar un grafo que represente las habitaciones como nodos y los pasillos como aristas. Con esta estructura podemos conocer las habitaciones adyacentes y los caminos entre estas aplicando, por ejemplo, el algoritmo de **Dijkstra**. Esto sería especialmente útil a la hora de encontrar caminos más cortos para la IA y la colocación de objetos y enemigos de manera más inteligente y no tan aleatoria.

### Conclusión

Se trata de un método con una definición muy sencilla pero que requiere de mucho trabajo extra para obtener resultados correctos.

* Se basa en técnicas sencillas.
* Requiere demasiada personalización extra por parte del programador, sobretodo en la creación de pasillos.
* De por sí no proporciona una estructura de datos (grafo).
* Los problemas de solapamiento no son fáciles de solucionar.

## Generación de mazmorras. Particionado BSP.

Esta técnica utiliza el particionado binario para subdividir un espacio utilizando hiperplanos. Las subdivisiones obtenidas son representadas mediante una estructura de datos de tipo árbol, conocida como **BSP Tree o Árbol BSP**.

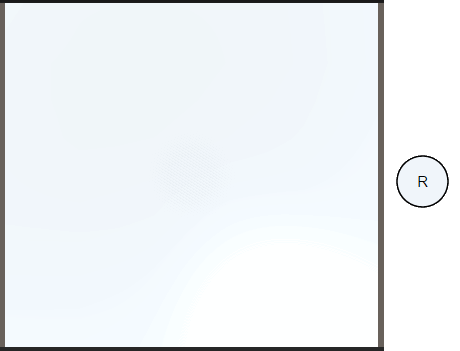
### ¿Por qué utilizar un árbol BSP?

Cuando vamos a generar un mapa de mazmorra, hay muchas maneras de hacerlo, simplemente podríamos generar rectángulos de tamaños y en posiciones aleatorias y crear una habitación en cada uno, pero esto puede llevar a muchos problemas, como la superposición de habitaciones, espacios entre estas demasiado arbitrarios y extraños. Si entonces queremos pulir el algoritmo y arreglar estos problemas la complejidad de la solución se vuelve grande y es más difícil de depurar nuevos problemas.

La misma estructura de un **árbol binario** nos permite dividir el espacio de manera más o menos regular, manteniendo la consistencia entre el espacio entre habitaciones, los tamaños de estas y permitiéndonos unir mediante pasillos basándonos directamente en la unión de los mismos nodos del árbol.

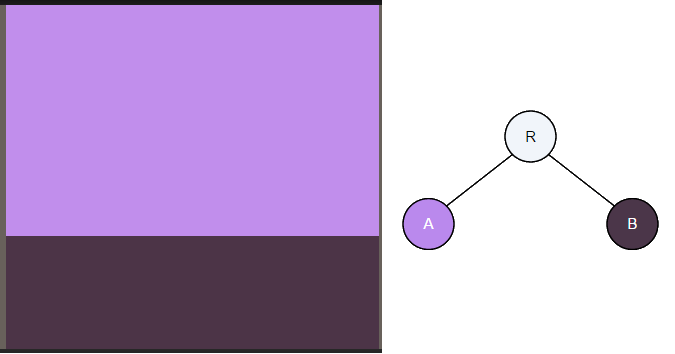
El método procedería de la siguiente manera:

**1.** Creamos un plano completo sobre el que vamos a generar las habitaciones de la mazmorra. Este espacio completo es la raíz del árbol.

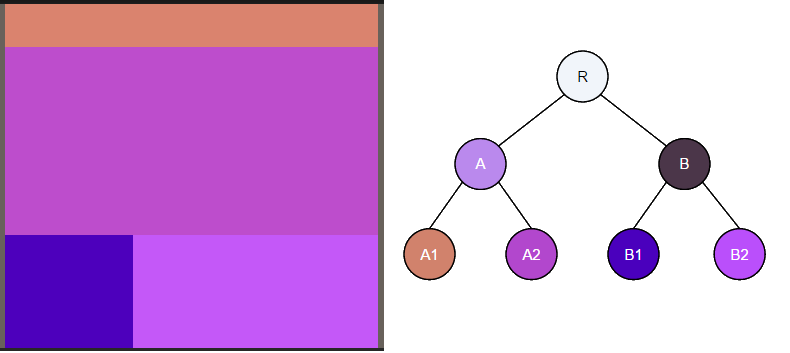


**2.** Escogemos una orientación aleatoria, horizontal o vertical, sobre la que vamos a partir.

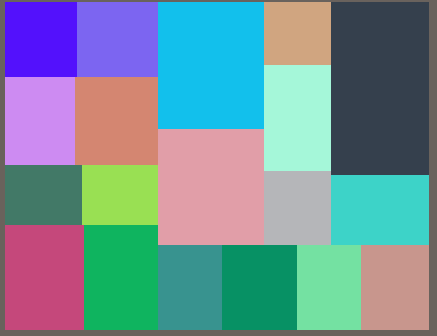
**3.** Escogemos un punto en x (horizontal) o en y (vertical) según la orientación escogida y partimos el espacio en dos sub-mazmorras.



**4.** Seguimos subdividiendo esas sub-mazmorras generadas pero teniendo cuidado que las divisiones no sean demasiado cerca del borde, ya que debemos ser capaces de poder incluir una habitación en cada una de estas divisiones.



Resultado de la segunda iteración.



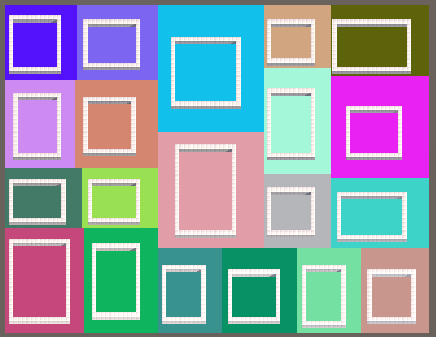
Resultado de la última iteración. Cada uno de estos espacios incluirá una habitación.

**¿Y cuándo nos detenemos?**

Tenemos varias opciones:

* En cada iteración comprobamos si quedan espacios que puedan ser divididos y en las nuevas áreas se puedan crear al menos una habitación, seguimos dividiendo hasta que no queden espacios que cumplan esta condición. El problema con este método es que al final obtenemos áreas con muy dimensiones similares, por lo que las habitaciones también serán prácticamente iguales, ya que estamos asegurándonos en cada iteración de que realizamos al menos un corte hasta que no hay espacio para más.
* Otra solución es establecer un número de iteraciones fijo dependiendo del tamaño de la mazmorra, con lo que obtendremos áreas divididas al máximo posible, pero otras que se podrían dividir al menos una vez más, pero se quedan enteras y de esta manera se puede crear un habitación más grande y alargada o una habitación pequeña pero un pasillo largo que la une con otra sección.

**5.** Cuando tenemos las divisiones para incluir una sola habitación en cada una de estas, comenzamos a construir la mazmorra. Creamos una habitación de tamaño aleatorio dentro de cada división, es decir, cada hoja del árbol, teniendo en cuenta los límites del espacio.



Como vemos las habitaciones dentro de cada espacio se generan en una posición y con dimensiones aleatorias dentro de sus límites, por lo que podemos encontrar habitaciones pequeñas en espacio relativamente grande. Haciendo estos parámetros ajustables podemos obtener infinidad de resultados diferentes.

**6.** Para construir los pasillos que unen las habitaciones recorremos el árbol conectando cada nodo hoja con su hermana. Los nodos hoja son los nodos que no tienen más hijos, al final de cada rama del árbol.

El resultado final para el ejemplo que hemos estado viendo sería como se ve en la siguiente imagen.



El utilizar un árbol binario nos asegura que después de realizar la primera partición vamos a obtener al menos dos hojas en dos ramas distintas que podemos utilizar como habitaciones de inicio y de final, estando siempre separadas por la raíz, por lo que nunca estarán conectadas directamente y se deberá recorrer primero la mayor parte del resto de habitaciones, algo que es esencial para el tipo de juego que queremos realizar en este caso.

Como alternativa podemos utilizar un **QuadTree** para generar la mazmorra usando este tipo de técnica. En ese caso tendríamos cuatro nodos por cada partición y puede acelerar el proceso de creación, pero la diferencia es realmente mínima en estos casos.

### Semillas

Al igual que hemos visto con la generación con autómata celular, el uso de semillas también es de interés en este algoritmo ya que los cortes de cada nodo se realizan utilizando números aleatorios, por lo que podemos establecer una semilla inicial y obtener los mismos resultados cada vez, siempre que usemos también los mismos parámetros de tamaño de mazmorra y de habitaciones. Aquí podemos ver un ejemplo que se puede probar en la sección de tests del proyecto.

Width: 70

Height: 40

Room size: 12 (Tamaño mínimo de corte)

Seed: 96



### Conclusión

A diferencia del autómata celular, el método de particionado binario nos proporciona una distribución de habitaciones más adecuada para videojuegos donde las mazmorras son edificios construidos por manos humanas, no naturales. Estas habitaciones tienen que tener formas y estar espaciadas de manera más regular, con pasillos que las unan y con, al menos, dos habitaciones con solo una conexión, que podemos utilizar para colocar la entrada y la salida de la mazmorra. Se trata de un tipo de distribución que podemos encontrar en juegos como Zelda o Diablo.

Por el lado bueno:

* El resultado es más previsible que con otros algoritmos, sabiendo siempre que obtendremos habitaciones con unas dimensiones y separación entre estas similares pero variables.
* Es un algoritmo rápido ya que en unos pocos pasos tenemos suficientes nodos en el árbol como para generar una mazmorra de tamaño considerable, siempre dependiendo de los parámetros variables que proporciona el programador.
* El mismo proceso de corte nos proporciona un grafo con el que podemos identificar las habitaciones. De esta manera es más sencillo unirlas mediante pasillos utilizando nodos hermanos, asignarles roles y llenarlas de contenido posteriormente.
* No se producen bucles entre habitaciones adyacentes.
* Es bastante conveniente para la mejora de rendimiento ya que el árbol puede ser utilizado para determinar que habitaciones se renderizan en la escena en cada momento, así como para limitar la gestión de colisiones a la habitación/nodo visible.

Pero estas ventajas también hacen que este sea un algoritmo con un propósito más concreto.

* La distribución de habitaciones que proporciona es bastante equitativa, por lo que no puede representar el caos de un sistema natural como lo hace el método del autómata celular.
* La forma de las habitaciones, al contrario que el autómata celular, se deja enteramente al programador. Esto se ha comentado dentro de las ventajas como previsibilidad, pero también agrega trabajo extra al programador.

## Generación de contenido en mazmorras

Cuando se habla de generación procedimental de mazmorras normalmente se habla de la estructura de esta, la distribución, forma y conectividad de las habitaciones y los niveles. Pero cuando se llega al apartado de poblar la mazmorra se suele pasar por alto o simplemente se sugiere la creación de contenido de manera aleatoria a gusto diseñador.

Esto se puede entender ya que la generación del contenido dentro del nivel es algo que depende enormemente de las peculiaridades de cada juego, y lo más normal es que sea el diseñador el que decida cuales son las reglas de creación y el programador el que se encargue de diseñar un método para aplicar estas reglas. Pero estos métodos pueden estar basados en ciertas estructuras de datos o patrones de diseños.

Por ello, se hace difícil encontrar información al respecto, pero después de investigar ciertos artículos sobre el tema, presento aquí mi propia clasificación e interpretación de estos.

### Algoritmos basados en conjunto de condiciones.

Se tratan de los métodos más generales y simple. Se basan simplemente en generar un conjunto de reglas a las que el algoritmo generador de contenido tiene que atenerse. Luego se utiliza una distribución personalizada que determina el propio diseñador del juego para mantener el equilibrio deseado en la distribución de los objetos y enemigos por las habitaciones. Se podría dividir el nivel en áreas que abarquen, por ejemplo, 2x2 habitaciones, y entonces colocar los objetos de manera aleatoria evitando repetición dentro de esas áreas.

Las condiciones las determina el mismo diseñador mediante un sistema de probabilidades y hacen referencia a que por ejemplo, un nivel solo contenga un boss y siempre esté en la última habitación, que las habitaciones de alrededor de esta tengan más tesoro y enemigos que el resto, o que haya más probabilidad de encontrar pociones al inicio del nivel.

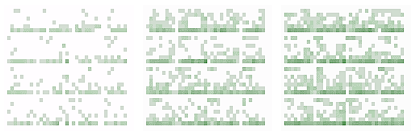
Es algo bastante común el utilizar un **grafo** para determinar los **pesos** de cada elemento a colocar y así afectar directamente a la dificultad del nivel. Si seguimos con el ejemplo de Spelunky, los murciélagos tienen un mayor peso que las serpientes, ya que pueden volar y seguir al jugador. Esto permite, en niveles más avanzados, escoger entre las entidades de mayor peso, así aumentando la dificultad del nivel.

La dificultad se puede medir utilizando una suma ponderada de valores normalizados (estableciendo mínimos y máximos).

Dpuntuación = 20 \* Nmurciélagos + 10 \* Nserpientes + ...

Los pesos de cada entidad se deciden mediante pura experimentación sobre los distintos niveles del juego.

Un estudio sobre un clon de Spelunky muestra claramente como la dificultad al en distintos niveles la determina la cantidad de enemigos en el nivel.



Dificultad del 10%, 50% y 90%. La densidad de puntos indica la cantidad de enemigos.

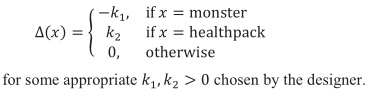
### Algoritmo basado en propagación de restricciones.

Se basa en la definición de **dominios finitos** que representan una serie de valores o atributos para los nodos del grafo que representa la topología del nivel, es decir, habitaciones, pasillos y otros espacios. Estos atributos están restringidos por una serie de reglas, que dividimos en **restricciones de cardinalidad** y **restricciones** **puntuales**.

Las **restricciones de cardinalidad** hacen referencia a los límites por los que los nodos pueden ser etiquetados con un valor específico, por ejemplo, decimos que en el nivel debe haber un único jefe, al menos 10 enemigos normales y como máximo 20, una cierta cantidad de tesoros, etc.

Las **restricciones puntuales** son más específicas e indican, por ejemplo, que la última habitación debe ser un jefe, o que las habitaciones al lado del jefe deben tener enemigos.

Finalmente se utiliza un **sistema de puntuación** mapeando cada uno de estos dominios a valores numéricos.



Con este sistema de puntuación asociado a los distintos dominios del nivel se puede hacer una evaluación de los caminos posibles en este y determinar si existe un equilibrio adecuado o es necesario ajustar el contenido de ciertos dominios.

La **propagación de** **restricciones** asegura que la modificación del contenido de una habitación propague los cambios para mantener el resto de habitaciones en el camino dentro de los parámetros que determina el dominio al que pertenecen.

### Personas procedimentales.

Un método más experimental para la población de mazmorras es el uso de **personas**. Una persona representa un arquetipo de posible jugador con su propio comportamiento y toma de decisiones. Con esta persona se trata de **evaluar** si un nivel es jugable y si es posible sobrevivir al peor escenario de este. Esta evaluación modifica el contenido del nivel adaptándolo al tipo de jugador al que está dirigido. Este método requiere de una pre-inicialización del contenido del escenario, para la que se puede utilizar una serie de reglas simples y basadas en generación de números aleatorios dentro de unas restricciones.

Podríamos tener una persona que represente a un guerrero y cuyo objetivo será matar el mayor número de monstruos. Si en la evaluación del escenario por esta persona se llega a la conclusión de que no es posible cumplir los objetivos en el peor de los casos, entonces se hacen las modificaciones adecuadas y se siguen haciendo pruebas hasta que el resultado sea el esperado.

La persona deberá decidir cuál es el mejor **camino** dentro de la mazmorra dependiendo de su personalidad, si es un **guerrero** escogerá el camino con mayor **número de monstruos**, si lo que quiere es recolectar **tesoros** entonces intentará **evitar los monstruos** pero tendrá que tomar ciertos **riesgos** dependiendo de si la recompensa lo vale. Por ejemplo, un cazador de tesoros se enfrentará a dos monstruos si estos están en el camino de un tesoro bastante valioso y no hay camino alternativo. Esto se conoce como **función de utilidad**, y determina la determinación del agente a tomar ciertos **riesgos** en la búsqueda de cierta **recompensa**.

Para ello se hace uso de **algoritmos evolutivos** que afectan a una serie de **perceptores lineales (redes neuronales)**, cuyos valores de entrada son parámetros como la salud del jugador, el cofre más cercano que no involucre combatir, la distancia a la salida, etc.

Estos perceptores permiten evaluar la situación y tomar una decisión, y esta decisión permite a su vez la evolución de los mismos perceptores, que establecen nuevos pesos para estos en un proceso de **auto-adaptación**.

Aunque el uso de estos perceptores entra dentro de la rama de la inteligencia artificial y redes neuronales, su combinación con algoritmos evolutivos produce un tipo de personas llamadas personas procedimentales que actúan sobre el nivel produciendo resultados diferentes en cada partida.

### Métricas. Experiencia de usuario.

La experiencia del jugador durante el transcurso del juego proporciona información valiosa que puede ser utilizada para ajustar la dificultad de los niveles. Los tipos de objetos y su posicionamiento en los niveles pueden variar con cierta consistencia basándose en unos parámetros que dependen directamente del número de nivel o la profundidad dentro de este, pero la frecuencia de objetos y fuerza de los enemigos puede ser afectada por los resultados del jugador en niveles anteriores. Resultados como el tiempo en terminar un nivel, el número de enemigos que se han eliminado, las veces que ha muerto o los puntos que ha conseguido pueden servir de base para ajustar la dificultad de niveles posteriores.

# CAPÍTULO 5. DESARROLLO DE UN VIDEOJUEGO

Después de haber estudiado distintos algoritmos para la generación de mazmorras y de su contenido vamos a entrar en detalles sobre el desarrollo del videojuego, qué algoritmos se han escogido y cómo se han implementado para este determinado juego.

Antes de continuar leyendo se recomienda hacer un repaso rápido al documento de diseño adjuntado con el proyecto donde se detallan las peculiaridades del juego, ya que aquí solo nos vamos a detener en explicar en cómo se ha realizado la implementación de lo detallado en ese documento.

## El juego

*Descripción básica del juego, objetivos, requisitos funcionales.*

## Herramientas y tecnologías

*Sobre qué motor de juegos se ha realizado y que otras herramientas o recursos se han utilizado para los assets y por qué.*

## Generación de mazmorras

*Esta sección falta ser ilustrada con código y algunas imágenes.*

### Árbol BSP

Para generar la estructura base de la mazmorra he escogido el algoritmo de generación mediante **árbol BSP**. La implementación de este algoritmo es algo más compleja que otros vistos, pero nos proporciona ciertas ventajas para la siguiente colocación de objetos.

Por un lado nos permite identificar fácilmente que nodos serán habitaciones, pero además podemos utilizar la división inicial de la raíz para establecer una dificultad distinta entre un lado del árbol y otro. Esto también nos permite escoger la entrada y salida de mazmorra, simplemente basándonos en el lado en el que se sitúa el nodo.

Si queremos adentrarnos incluso más, podemos establecer una dificultad según la profundidad del nodo en el árbol, los de niveles más altos tendrán probabilidades mayores de aparición de ciertos enemigos y trampas. Esto agregar una curva de dificultad progresiva según vamos avanzando dentro de la misma mazmorra.

Vamos a definir ciertos valores para acortar la descripción del algoritmo:

* **ROOM\_SIZE**: tamaño mínimo del espacio que ocupará una habitación.
* **NODE\_SIZE**: tamaño del espacio/nodo.
* **MARGIN**: Margen entre el borde de los espacios de los nodos y la habitación contenida en estos.

### Generación del árbol

**1.** Creamos un grid de enteros del tamaño indicado en las dimensiones de la mazmorra. Una casilla de este grid equivale a una casilla del grid de Unity, de una unidad, el tamaño por defecto de un cubo.

Inicializamos toda la rejilla al valor 0.

**2.** Creamos el árbol BSP comenzando con el nodo raíz. Posicionamos el nodo raíz en el centro del mapa.

**3.** Nos movemos al siguiente nodo por la izquierda. Si tienes hijos, vamos al siguiente por la izquierda, llamando al mismo método de forma recursiva pasando este nodo izquierdo. Cuando no se encuentre hijo, partimos (Cut).

**3.1** **Cut**: Decidimos con un 50% de probabilidad, si se divide (Split) por el eje 'x' o el 'z'.

**3.2 SplitX/SplitZ**:

**a.** Comprobamos que se puede hacer un corte, dejando sitio en ambos nodos resultante para introducir una habitación en cada uno. Para ello comprobamos si el tamaño del nodo es al menos ROOM\_SIZE \* 2.

**b.** Realizamos el corte, escogiendo un valor entre [ROOM\_SIZE, NODE\_SIZE.x/z - ROOM\_SIZE]. De esta manera dejamos al menos un mínimo del tamaño de la habitación. Restamos el valor obtenido al NODE\_SIZE y obtenemos así el tamaño de la partición.

**c.** Del valor anterior creamos un nodo a la izquierda del nodo actual. Obtenemos el espacio restante y creamos el nodo derecho.

**4.** Al volver pasamos al nodo derecho y realizamos los mismos pasos.

**6.** ITERAR 3-5 durante x ciclos. El número de ciclos dependerá del tamaño de la mazmorra habitaciones.

### Creando las habitaciones

**7.** Recorremos el árbol de forma recursiva y cuando llegamos a un nodo hoja, sin hijos, creamos una habitación en este, AddRoom(nodo).

**7.1.** **AddRoom**(nodo): Obtenemos la posicion de la habitación a partir de la del nodo, será la misma puesto que vamos a centrar la habitación en el espacio particionado.

**7.2.** Instanciamos el objeto base para la habitación en esta posición y le damos unas dimensiones aleatorias entre el tamaño mínimo de la habitación (ROOM\_SIZE) y el tamaño máximo de espacio disponible menos un margen que dejamos para que dos habitaciones adyacentes no aparezcan demasiado juntas.

Size x = Random(ROOM\_SIZE, NODE\_SIZE - MARGIN)

Size z = Random(ROOM\_SIZE, NODE\_SIZE - MARGIN)

**7.3.** Con la posición y dimensiones de la habitación ahora recorremos el grid estableciendo los tiles de suelo(1) y paredes(2).

### Conectando las habitaciones

**8.** Recorremos el árbol hasta las hojas, uniendo sus habitaciones con pasillos.

El proceso se realiza bajando hasta un nodo hoja con habitación, entonces se busca su hermano, que contendrá una habitación adyacente.

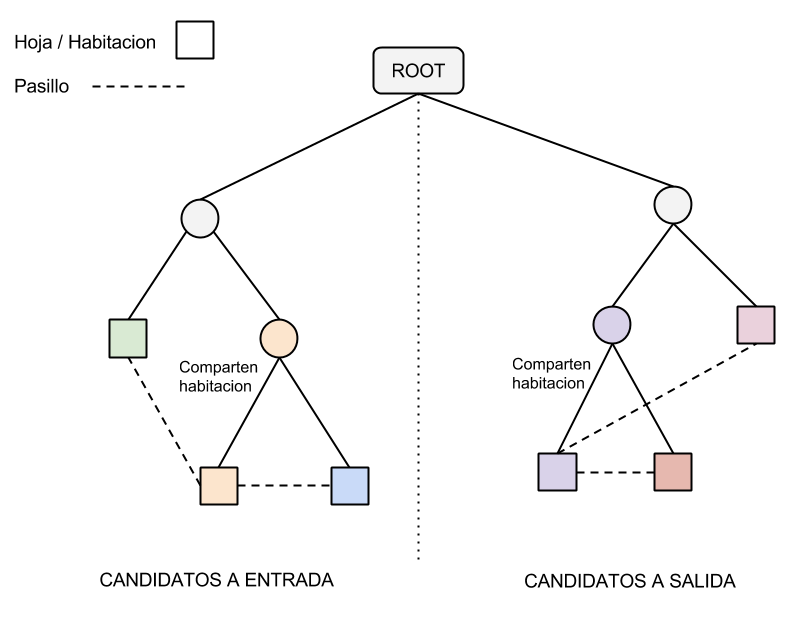
Entonces creamos un pasillo entre estas dos habitaciones mirando los lados adyacentes y seleccionando un punto de puerta aleatorio en la primera de las habitaciones. Desde este punto "cavamos" en línea recta hasta la otra habitación.

Una vez creado el pasillo subimos al nodo padre, que no tendrá asignada una habitación y le asignamos aleatoriamente una de entre los nodos hijo. De esta manera el nodo hermano de este nodo padre se conectará con una de los nodos hijos y realizando esto por todo el árbol mantenemos la conectividad en toda la mazmorra.

### Limpiando mediante autómatas celulares

**9.** Un paso adicional es hacer una limpieza de **celdas** "**solitarias**" que puedan haber quedado en la creación de los pasillos. Aplicamos una iteración usando el algoritmo del **autómata celular** de manera ligera, es decir, modificamos las reglas de eliminación de tiles para que solo sean eliminados aquellos que se han quedado aislados.

Esto además agrega más variedad a la geometría de las habitaciones y pasillos, creando un efecto de redondeo en las esquinas y abriendo un poco más algunos pasillos.



## Generando el contenido de la mazmorra

*Descripción del algoritmo implementado para poblar la mazmorra con objetos, trampas y enemigos. Ajustes de la dificultad de esta.*

## Inteligencia artificial

*Detalles sobre la implementación de la máquina de estados y el pathfinding básico de la inteligencia artificial de los enemigos.*

## Elementos visuales

### Menús y HUD

### Sprites 2D

### Modelos 3D

### Animaciones

## Cámaras

## Sonido y música

# CAPÍTULO 6. METODOLOGÍA DE TRABAJO

## Planificación

*Sección pendiente, el contenido está en un documento separado pero falta adaptarlo para incluirlo aquí.*

## Problemas

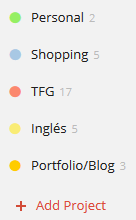
*Problemas encontrados durante el desarrollo del trabajo*

## Herramientas

### Gestión de objetivos: Todoist

Ya que se trata de un proyecto individual, no me era necesario utilizar herramientas avanzadas como Trello o Basecamp, que están orientadas a grupos de trabajo pequeños pero que se quedan un poco grande para una sola persona.

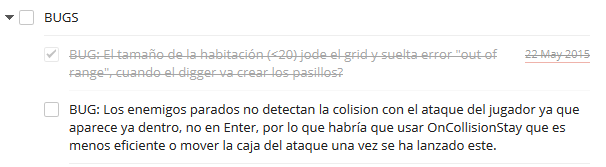
Por ello escogí una aplicación de gestión de listas de tareas. Todoist es una de las más populares y permite crear grupos convenientemente llamados proyectos, por lo que pude crear uno específico para este trabajo de fin de grado.



Dentro de estos grupos podemos crear tareas en modo de lista, pero lo que hace de Todoist una buena aplicación para gestionar proyectos individuales es el hecho de que permite crear grupos de tareas anidadas y establecer fechas para cada una de estas por separado o al grupo completo.

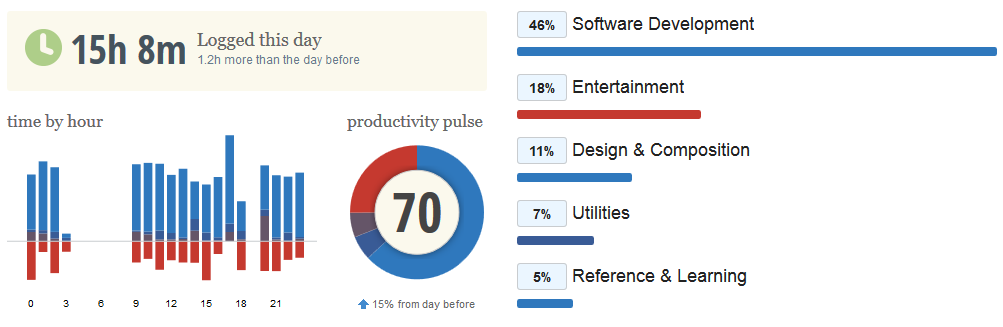


De esta manera podemos crear un grupo de tareas para la documentación, otro para los bugs, el diseño de assets, etc.



### Gestión de tiempo: Rescue Time

Rescue Time es una aplicación para la gestión del tiempo que hace un seguimiento de nuestros hábitos a lo largo del día, recogiendo información de sitios que visitamos, aplicaciones que usamos y tiempo que pasamos en estos. El objetivo es establecer un hábito de trabajo, intentando mejorar poco a poco cada día hasta conseguir un buen nivel de productividad.

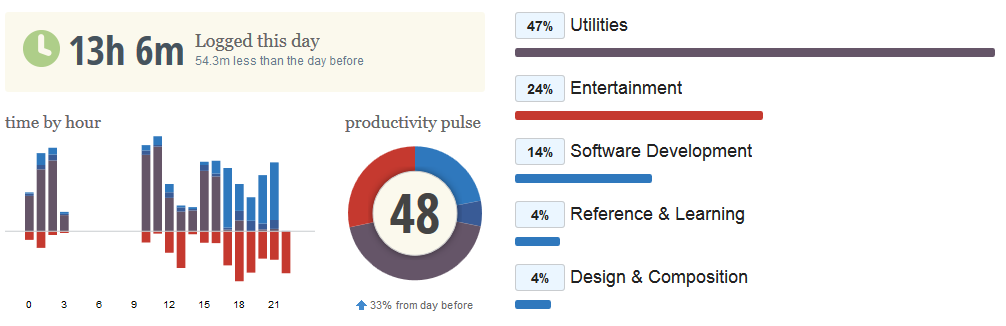


El software permite la configuración de objetivos diarios, estableciendo así una cantidad de horas máximas para entretenimiento, horas mínimas de desarrollo de software, de diseño, de estudio, etc.

La aplicación entonces te puede mostrar en cada momento el porcentaje y horas dedicadas a cada objetivo, si se están cumpliendo y la productividad general. Esta información se presenta por día, semana o mes y proporciona información extra con lo que podemos ver las etapas de mayor o menor productividad y actuar en consecuencia.

La desventaja de usar esta aplicación es que es necesario tener instaladas las correspondientes extensiones en cada uno de los navegadores que se usen de manera usual, así como la aplicación de escritorio y la de dispositivos móviles si queremos un total seguimiento de nuestros hábitos.

Además, el seguimiento de ciertas aplicaciones no siempre está dentro de las clasificaciones por defecto y obtenemos bajos niveles de productividad por esto mismo.



Aquí podemos ver un ejemplo de esto, donde Utilities es una clasificación neutral que no cuenta como productiva, pero que en este caso se trataban, en su mayoría, de aplicaciones utilizadas para trabajar en el proyecto pero que no eran reconocidas por la base de datos de Rescue Time.

En cualquier caso es una buen software para mejorar hábitos de trabajo y plantearte objetivos diarios que me ha venido bien en el último mes de trabajo donde tuve que centrarme para llegar a la entrega con tiempo.

*Falta incluir otras herramientas menores asociadas al proyecto, como el framework para el blog.*

# CAPÍTULO 7. VALORACIÓN FINAL Y CONCLUSIONES

*Reflexión sobre el desarrollo del trabajo.*

# BIBLIOGRAFÍA

*Referencias para el desarrollo del trabajo.*