Escuela Politécnica Superior

Videojuego basado en la generación procedimental de mazmorras

Grado en Ingeniería Multimedia

Trabajo Fin de Grado


Universitat d'Alacant
Universidad de Alicante

- Justificación y Objetivos

- Agradecimientos

- Dedicatoria

- Citas

- Índices

- Cuerpo del documento

  + Introducción

  + Marco teórico, estado del arte

  + Objetivos

  + Metodología

  + Cuerpo del trabajo

- Conclusiones

- Bibliografía y referencias

- Anexos

Agradecimientos

Aprovecho para darle las gracias a mis tutores, Francisco José Gallego y Faraón Llorens, por aceptar mi propuesta de trabajo y por sacar algún hueco para atenderme siempre que han podido, ya sea en persona como por correo.

Y principalmente a mi padre, sin el cual me habría sido imposible realizar este Grado en Ingeniería Multimedia. Gracias por su apoyo en todos los sentidos y por su paciencia durante estos años de estudio.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

1.1. 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 alternativas que puedan sustituir a los procesos de desarrollo de contenido manuales. En este aspecto cabe destacar a Minecraft debido a que, desde su lanzamiento, ha 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 en 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 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 proyecto propongo un estudio de técnicas de generación procedimental como las utilizadas en Minecraft, Spelunky o Diablo, 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.

1.2. 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 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.

1.3. Estructura del documento

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

El documento consta de 7 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.
- **Capítulo 6**: Herramientas utilizadas en la planificación y desarrollo del proyecto.
- **Capítulo 7**: Valoración final y conclusiones.
- **Anexo. Documento de diseño del videojuegos (GDD):** Como anexo al final de este documento, se incluye el llamado documentos de diseño del videojuego. 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.

2. Generación procedimental de contenido

2.1. 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**.

2.2. ¿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, 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.

2.3. 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.

2.4. ¿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 recurso 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.

2.5. 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 de videojuego lo describe la RAE como:

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

En la **Wikipedia** tenemos una definición más precisa:

"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 cómo 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.

2.6. ¿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 en sustitución de simplemente diseñar el contenido manualmente.

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 lo tenemos en la era de principios de los 80, cuando las **computadoras** caseras eran tan **limitadas técnicamente**, sobre todo en espacio en disco, lo que hacía imposible incluir todos los recursos o "assets" para hacer juegos medianamente grandes. El uso de técnicas de generación de contenido surge de esta necesidad de superar o bordear las barreras técnicas de la época.

Figura 2.1: 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 existen esas limitaciones técnicas (a no ser que se quieran imponer voluntariamente), ni siquiera en dispositivos móviles, y el uso de estas técnicas de generación de contenido son más por conveniencia que por limitación del hardware. 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.

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.

2.7. 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 procedimental 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.

2.8. 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.

2.9. 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 (Andrew Doull 2007), diseñador de videojuegos, en los que propone una taxonomía que intenta ser definitiva. Otros proponen clasificaciones alternativas (Togelius et al. 2011), aunque en ambos casos encontramos prácticamente los mismos conceptos. Vamos a estudiar la propuesta por Andrew Doull.

2.9.1. 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 maneras, 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.

2.9.2. 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 («Género de videojuegos» 2015) 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 (Tulleken 2009), como el utilizado en Minecraft.

2.9.3. 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.

2.9.4. Instanciación de entidades del juego

Se trata de una técnica cada vez más común por la que se hace uso de middleware para modificar dinámicamente ciertos parámetros de las entidades de la aplicación y crear la mayor variedad posible de una misma entidad, reduciendo 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 juegos como X-COM: UFO Defense que generan nombres aleatorios para los soldados de manera que el jugador los percibe como seres individuales, con personalidad.

2.9.5. 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.

2.9.6. 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. («Radiant AI» 2014) 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.

2.9.7. 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.

2.10. 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 sub-sistemas 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 procedimental, 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.

3. Generación de contenido en videojuegos

Ya hemos visto que la generación de contenido puede tomar muchas formas y sentidos, ahora vamos a realizar un repaso a la historia de los videojuegos más populares que, de alguna manera u otra, hacen uso de las ventajas de la generación de contenido mediante procedimientos, no simplemente para reducir el trabajo de tareas tediosas, sino para aportar algo interesante a la jugabilidad o para superar las limitaciones que ciertas situaciones imponen.

3.1. Superando las limitaciones

3.1.1. 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 procedimental de contenido. De entre estos quizá el ejemplo más memorable es Rogue.

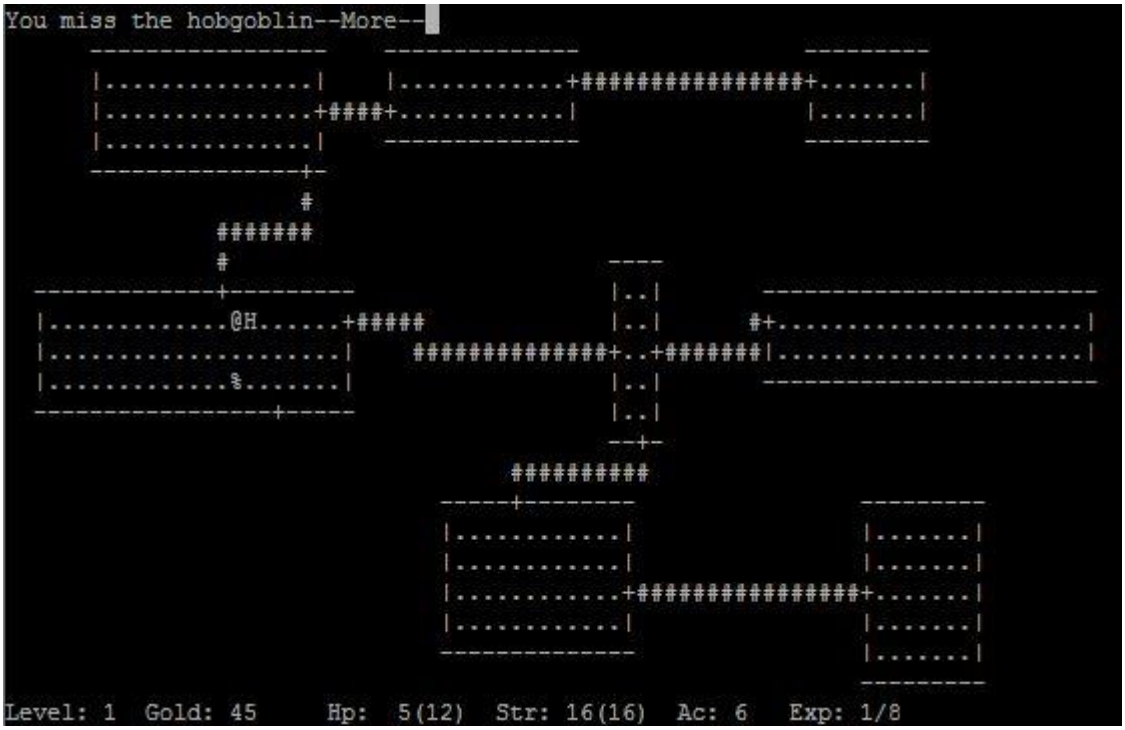


Figura 3.1 Rogue, 1980

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 guardado. 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.

3.1.2. 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 procedimental proporciona muchas horas de juego.

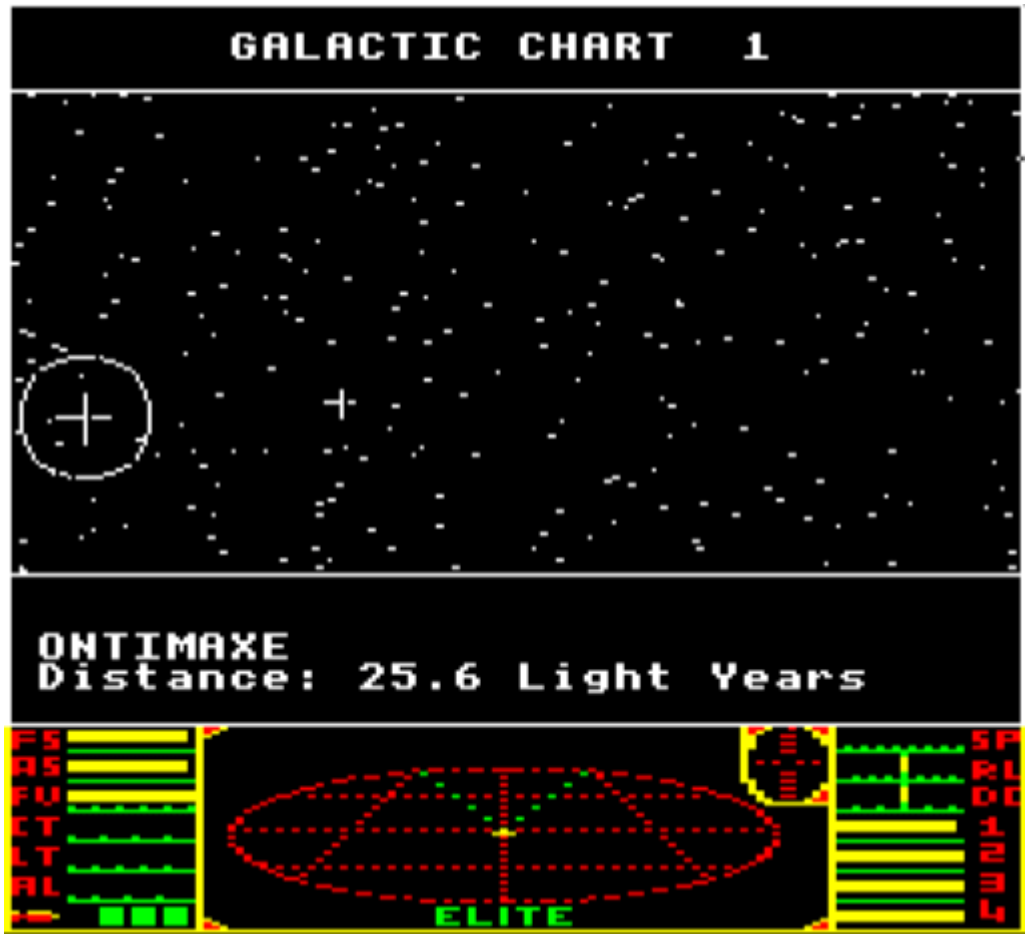


Figura 3.2 Elite, 1984

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 («The History of Elite» 2009) 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.

3.1.3. 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.

Figura 3.3 The Sentinel, 1986

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 procedimental, 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.

3.2. Una nueva generación

3.2.1. 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.

Figura 3.4 Diablo, 1996

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.

3.2.2. 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 procedimentales. No solo los mundos se crean de esta manera, sino que los movimientos de las criaturas en sí son animadas procedimentalmente, aun 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.

Figura 3.5 Spore, 2008

Por otro lado incluso la música de Brian Eno es creada usando un compositor algorítmico, lo que se conoce como música generativa («Generative music» 2015), término popularizado por él mismo. De esta manera la música se puede adaptar sobre la marcha a la gran variedad de situaciones.

3.3. La era independiente

3.3.1. 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 procedimentalmente 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**.

Figura 3.6 Spelunky, 2008

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.

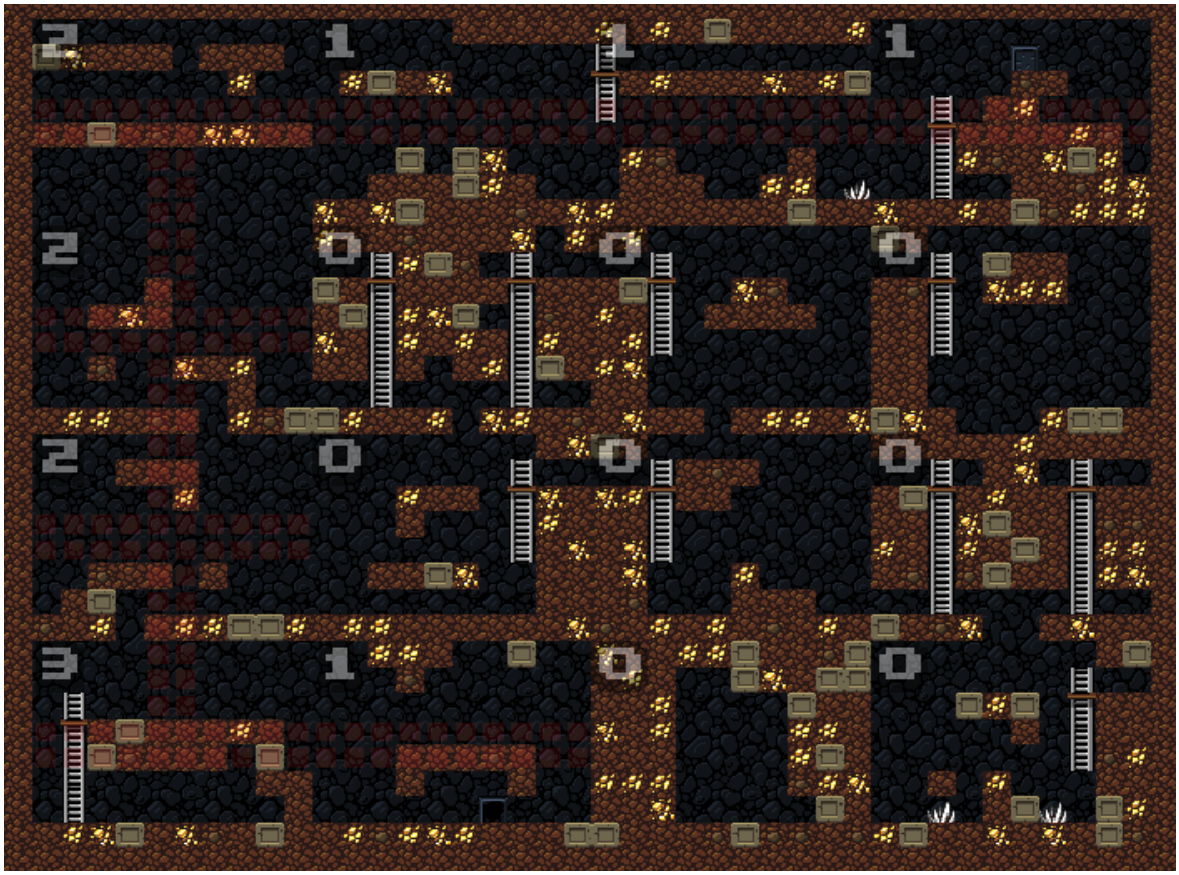


Figura 3.7 Spelunky, 2008

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.

3.3.2. 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 procedimental de mundos ya que es algo mucho más claro y una gran parte de lo que le aporta la diversión.

Figura 3.8 Minecraft, 2009

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 un artículo (Notch 2011) de su blog, 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.

Figura 3.9 Con un simple plano de ruido no es posible determinar ciertas estructuras

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.

3.3.3. 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.

Figura 3.10 The Binding of Isaac, 2011

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.

3.3.4. 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 procedimentales tanto para la vida animal como para los mismos planetas y distribución de estos.

Figura 3.11 No Man's Sky, 2015

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.

Figura 3.12 No Man's Sky. 2015

A fecha en la que se escribió este documento no se puede encontrar apenas información sobre las técnicas que utilizan para generar todo este contenido, solo he podido deducir, por entrevistas, que utilizan algo similar a lo visto en Spore para la generación de especies, y probablemente el resto se componga de un conjunto de técnicas ya conocidas.

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.

4.1. 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.

4.1.1. 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 **ge**n**erador 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**.

$$X_{n+1} = (aX_n + c) \bmod m$$

Donde X_0 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.

4.1.2. 4.1.2 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.

4.2. Introduciendo algunos conceptos

Vamos a estudiar un poco más a fondo algunos de los algoritmos para la generación procedimental 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**.

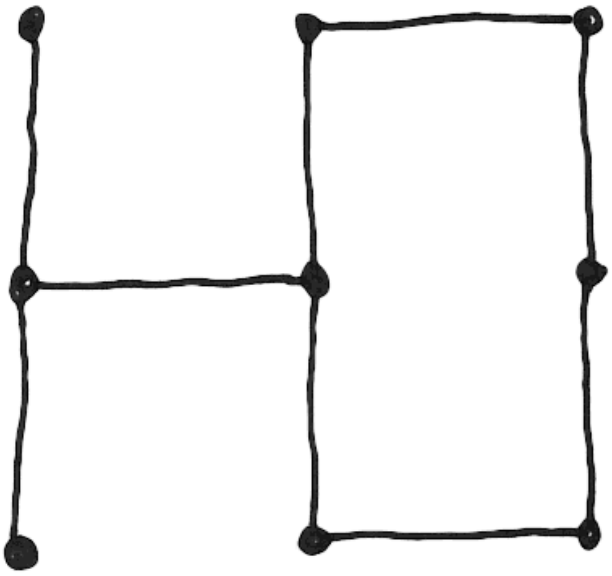


Figura 4.1 Grafo conexo

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

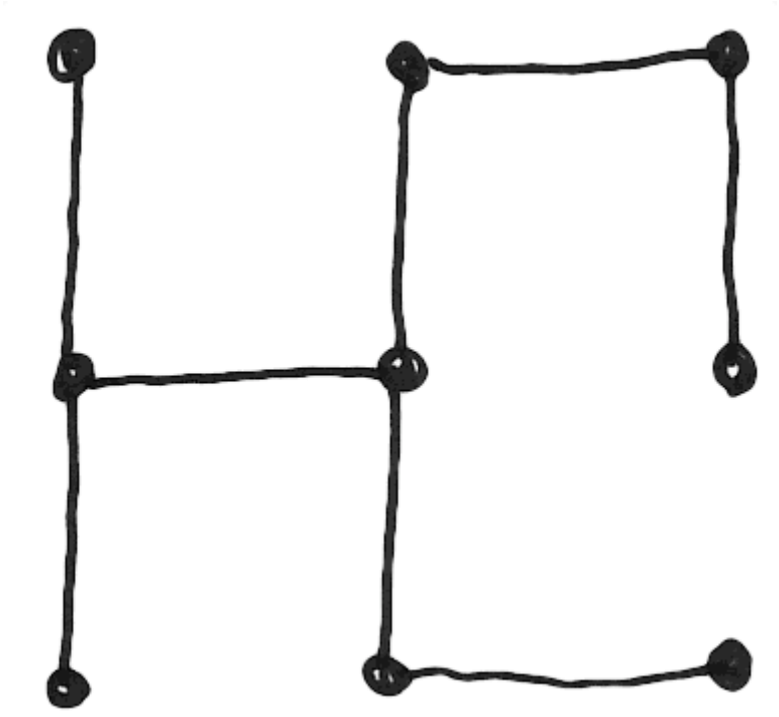


Figura 4.2 Árbol

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)**.

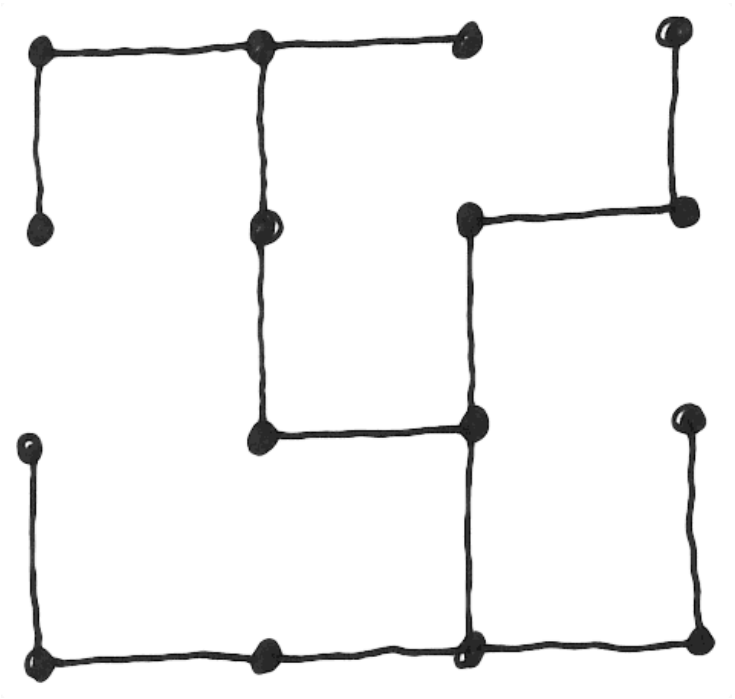


Figura 4.3 Árbol de expansión

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

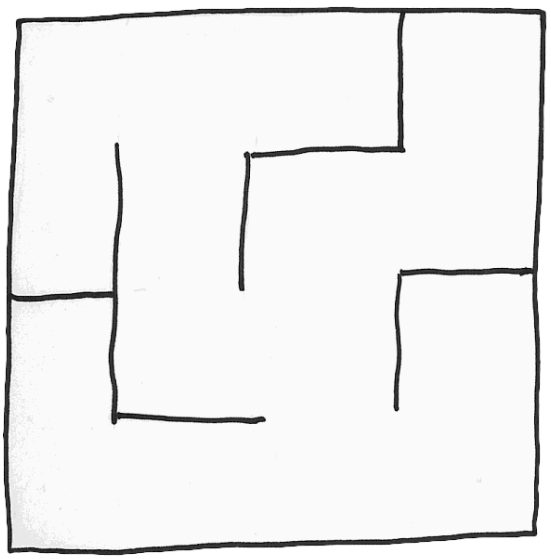


Figura 4.4 Laberinto

4.2.1. Aldous-Broder

Dos investigadores, **D. Aldous y A. Broder**, trabajando independientemente, estaban estudiando los árboles de expansión cuando diseñaron el siguiente algoritmo (Jamis Buck 2011):

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**.

4.2.2. 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 una 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 será un pasillo
    if (neighbor == null) {
      neighbor = CreateCell(coordinates);
      CreatePassage(currentCell, neighbor, direction);
      activeCells.Add(neighbor);
    }
    // Si la celda adyacente ya existía entonces crea una pared entre las
dos
    else {
      CreateWall(currentCell, neighbor, direction);
    }
  }
  else { // Borde del mapa, crea siempre una pared
```

```
    CreateWall(currentCell, null, direction);
  }
}
```

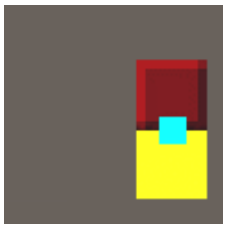
Para entenderlo mejor vamos a ver los primeros pasos de manera visual.

Celda en lista de activas.

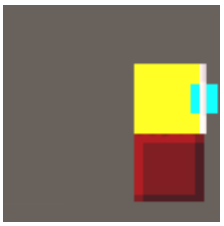
Celda finalizada. Sus 4 lados han sido inicializados

Celda actual y lado sobre el que se actúa en cian.

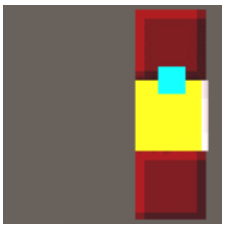
1. Escoge celda aleatoria inicial y la incluye en lista de activas.



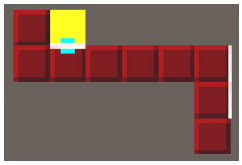
2. Escoge un lado de manera aleatoria y agrega la siguiente celda.



3. Escoge un lado de nuevo y como es límite de mapa pone una pared.

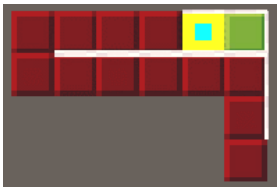


4. Continúa escogiendo otro lado. En este caso no visitado, por lo que crea un pasillo.



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.

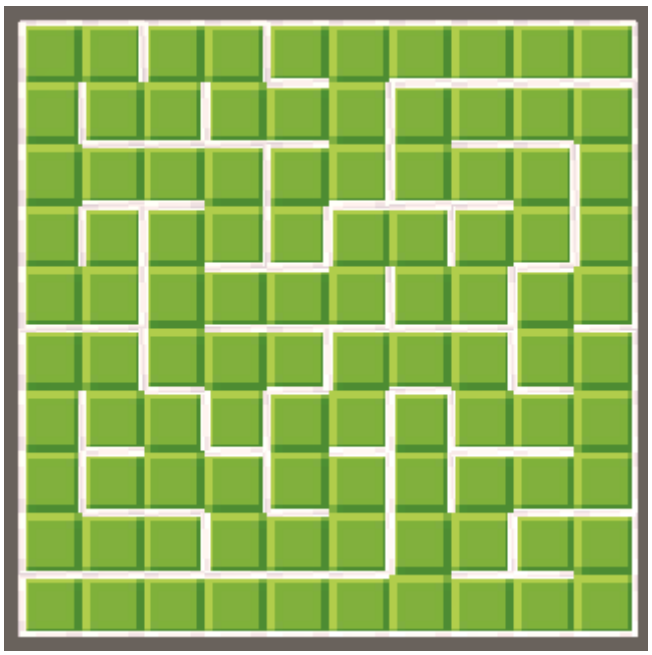


Figura 4.5 Laberinto final usando el algoritmo de Growing Tree

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.

4.2.3. 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, pero como veremos

hay otros algoritmos que son más adecuados y rápidos para generar mapas grandes de estilo mazmorra.

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**.

4.3. 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» 1970)**,** que despertó el interés en el entorno académico.

4.3.1. 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 sobre-población. 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.

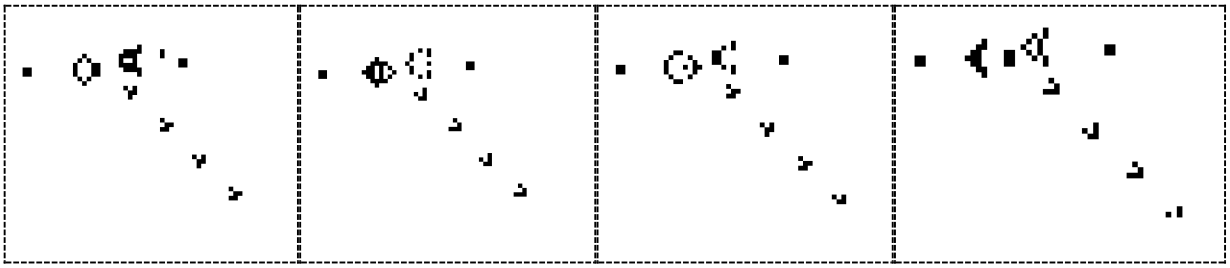


Figura 4.6 Cuatro generaciones del juego de la vida de Conway

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.

4.3.2. Autómata celular en generación de cuevas

Jim Badcock publicó hace unos años su aplicación de los autómatas celulares en la creación procedimental de cuevas (Jim Babcock [sin fecha]), 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);
      }
    }
  }
}
```

2. Recorremos el mapa de nuevo ahora aplicando las reglas del autómata celular:
   a) Si la celda es pared y tiene menos de 3 vecinas de tipo pared, entonces se convierte en espacio vacío.
   b) 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 número de paredes adyacentes a esta posición
        int numWalls = GetAdyacentWalls(c, r);
        // La celda actual es pared
        if (grid[c, r] == 1) {
          // Si paredes adyacentes <= 2, se vacía
          if (numWalls <= 2) {
            grid[c, r] = 0;
          }
        }
        else {// Es vacío, si >= 5 paredes adyacentes, se convierte en
pared
          if (numWalls >= 5) {
            grid[c, r] = 1;
          }
        }
        c++;
      }
      r++;
    }
  }
}
```

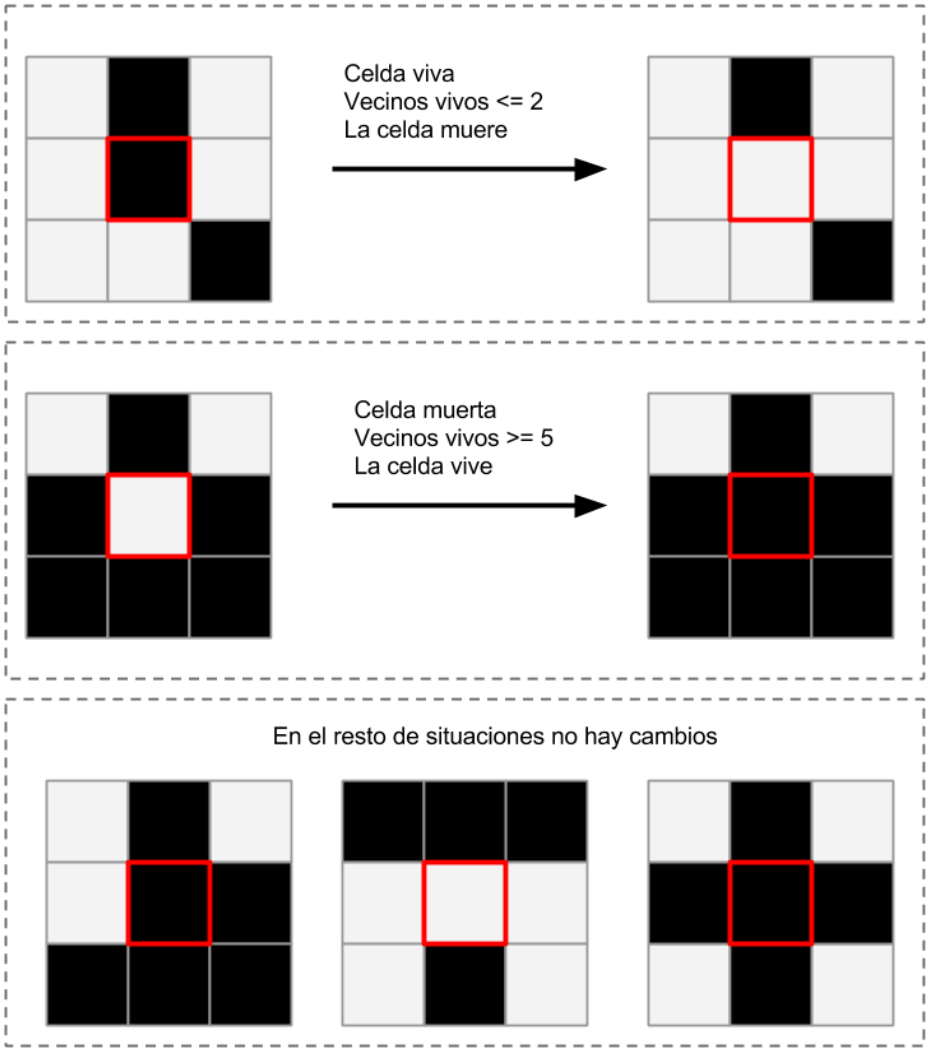


Figura 4.7 Reglas del autómata celular

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:

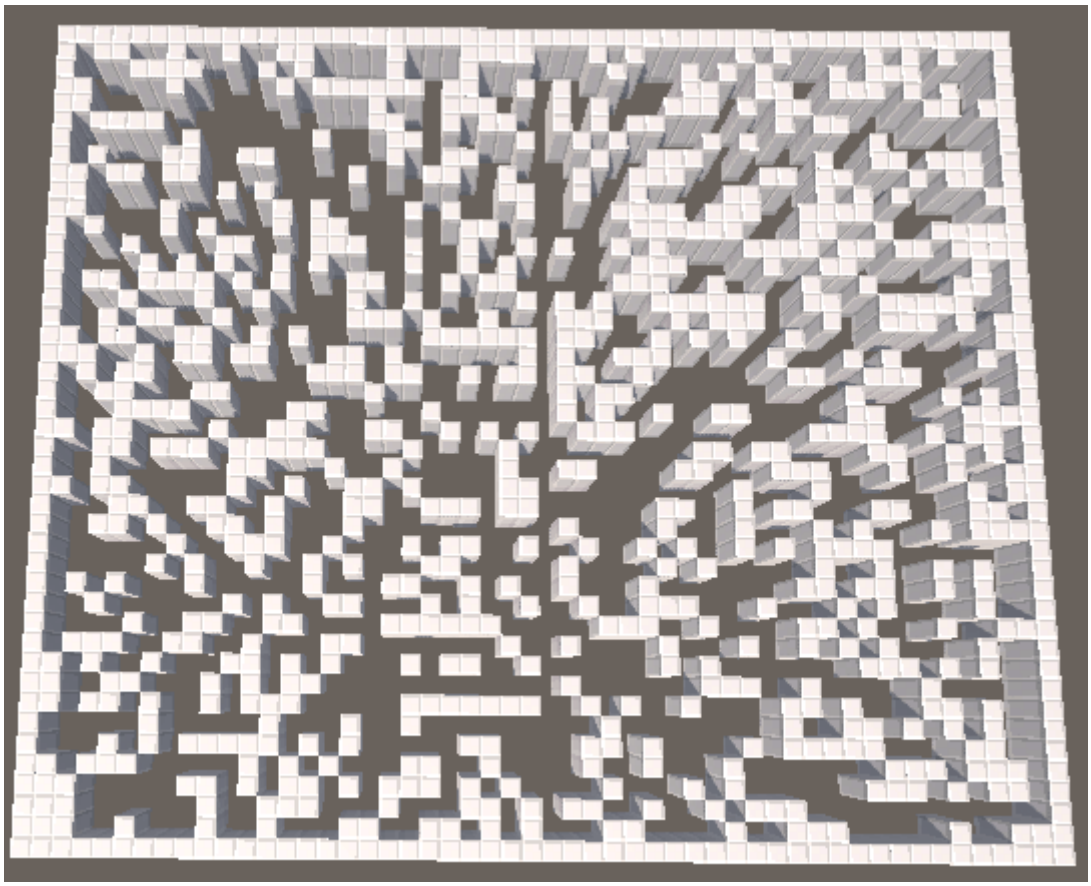


Figura 4.8 Tablero inicial generado aleatoriamente con una probabilidad de paredes del 35%.

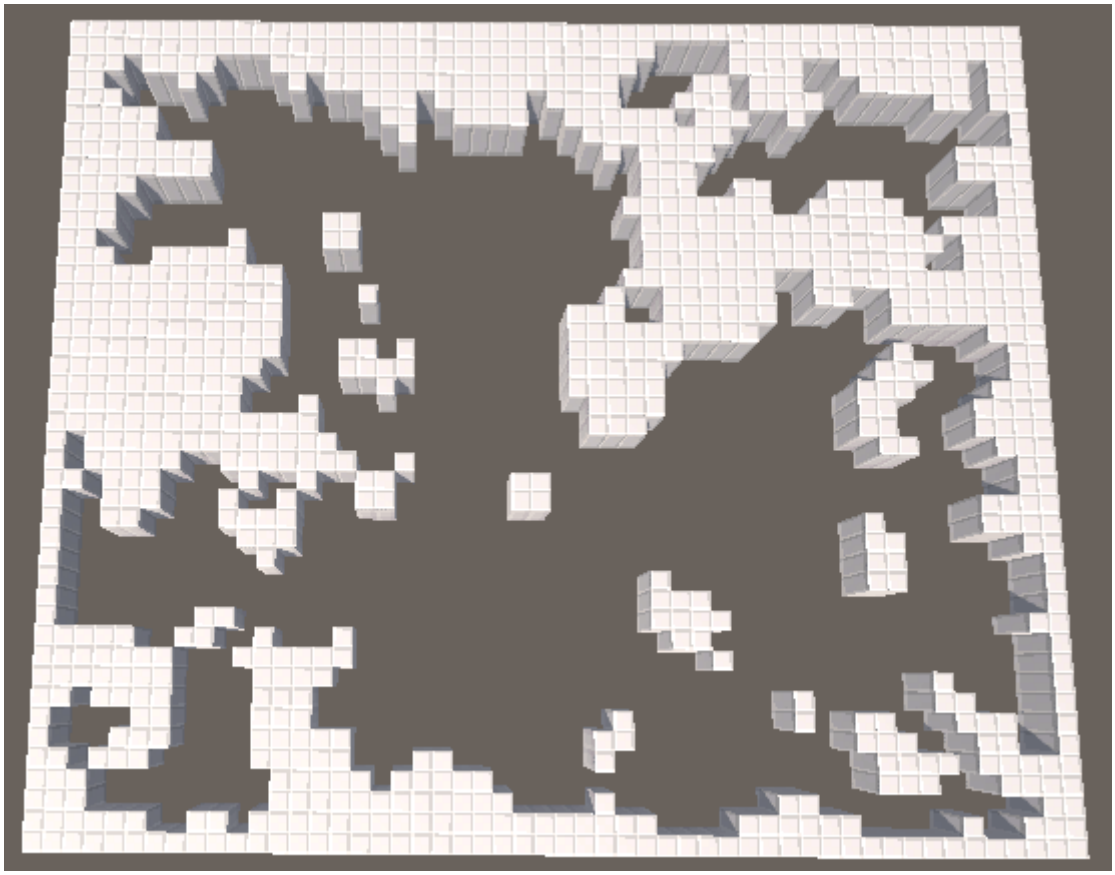


Figura 4.9 Resultado al realizar un solo pase

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.

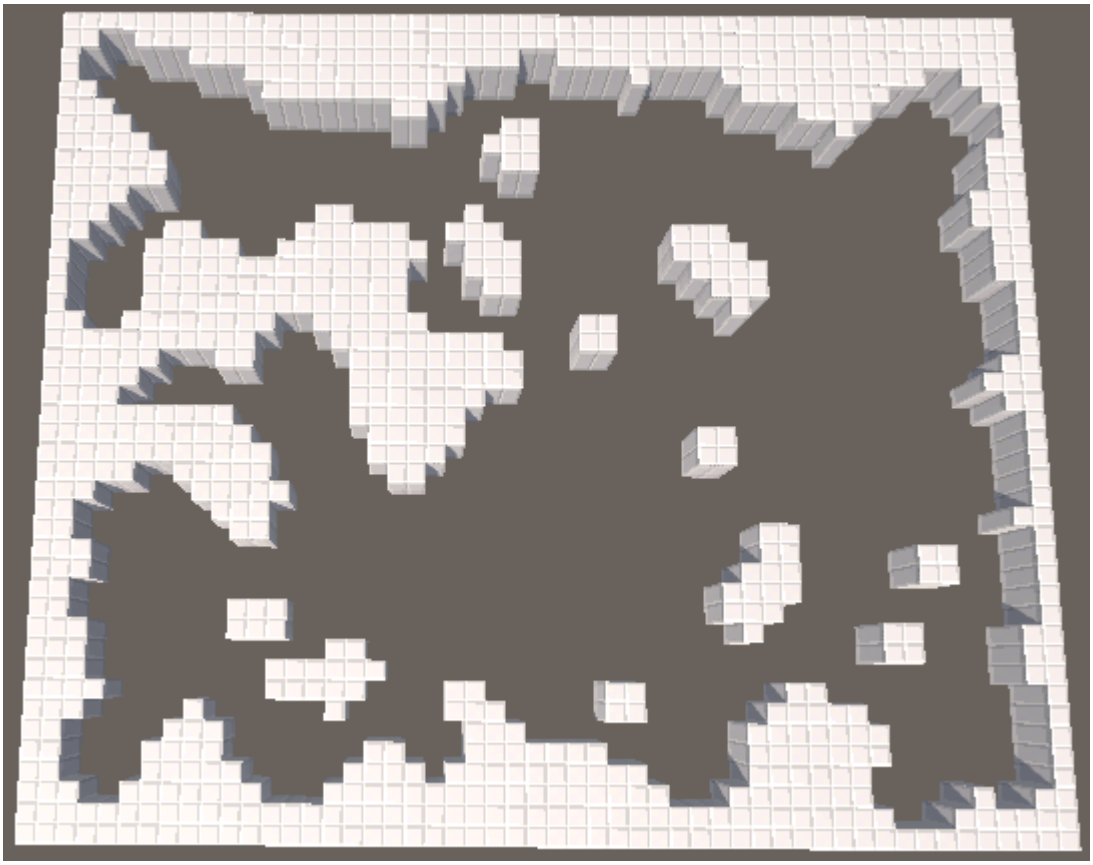


Figura 4.10 Resultado con dos pases

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.

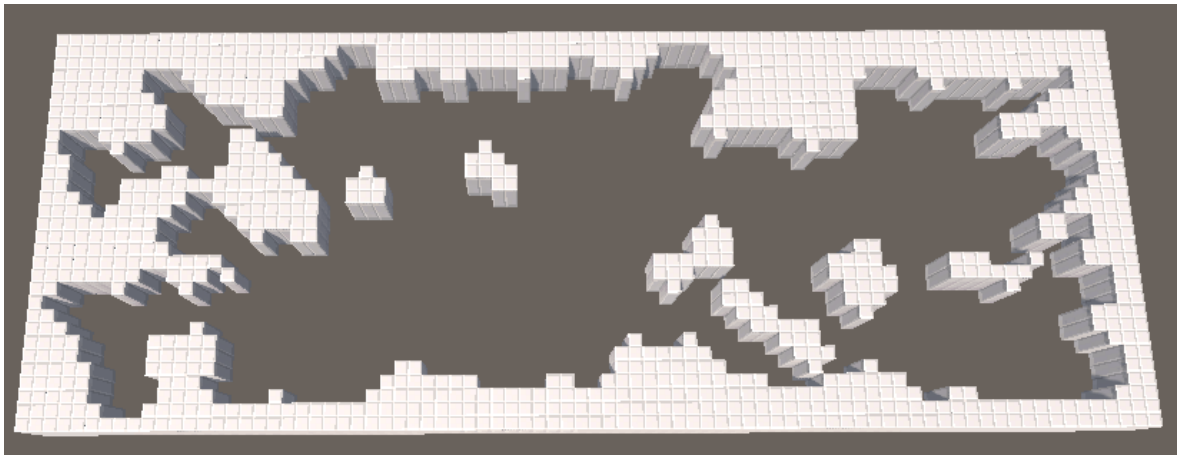


Figura 4.11 Los mismos parámetros pero con diferentes dimensiones. Conseguimos conectar todos los espacios pero el centro está demasiado despoblado

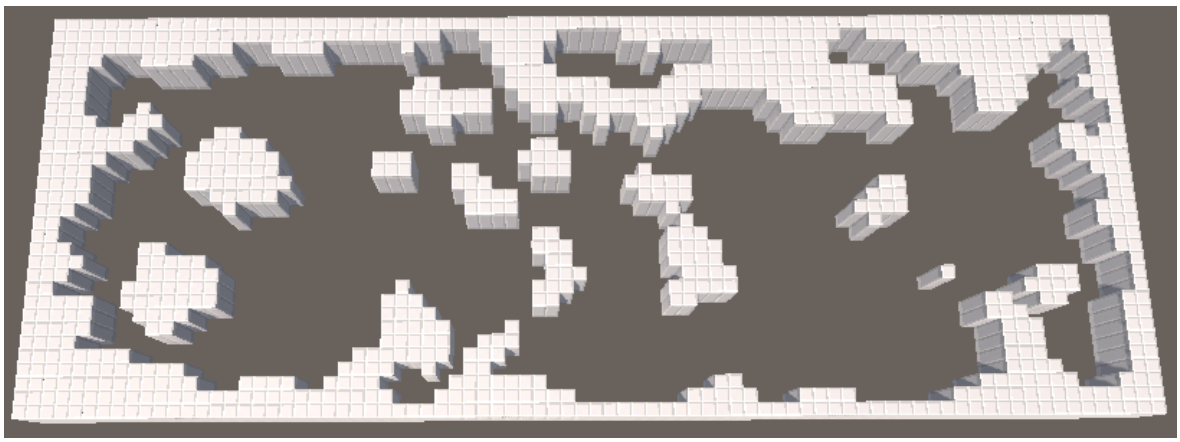


Figura 4.12 Si subimos la probabilidad de paredes al 40% y al realizar dos pases obtenemos un resultado mejor

4.3.3. 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:

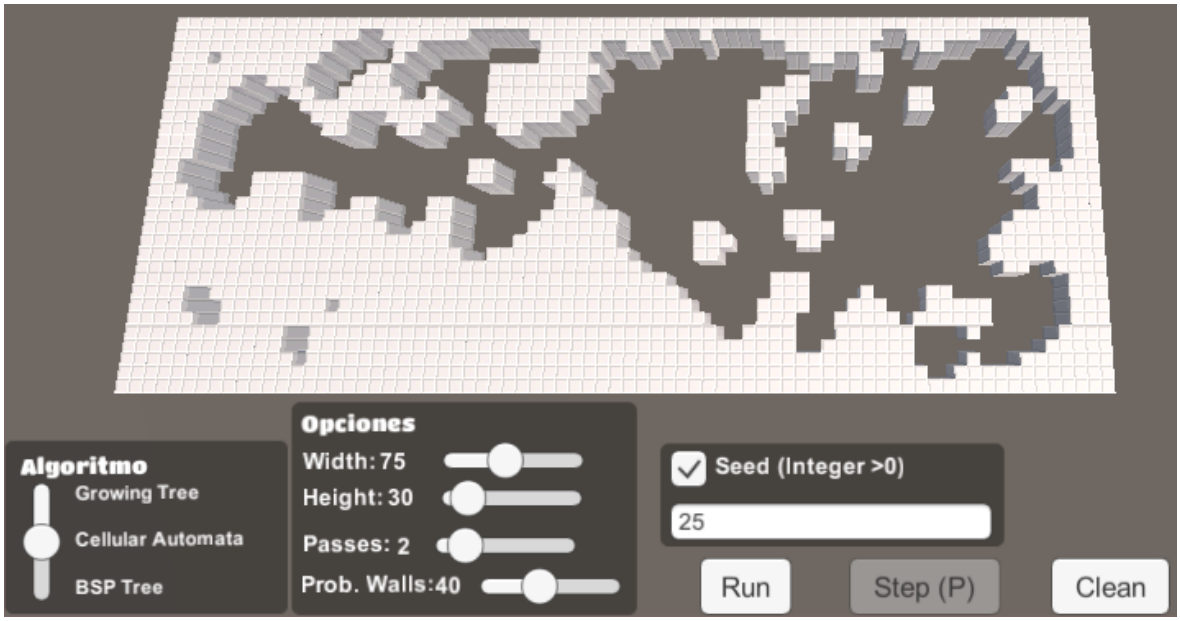


Figura 4.13 Autómata celular generado mediante semilla fija

4.3.4. 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:

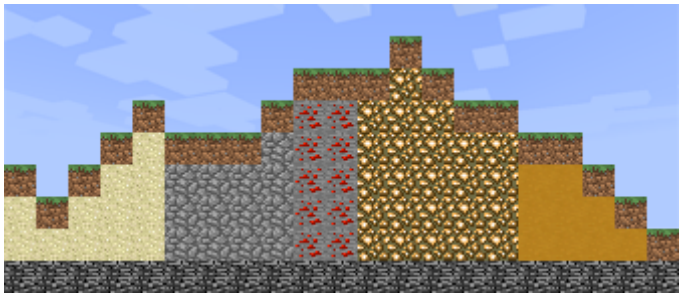


Figura 4.14 Distribución de bloques por capas de profundidad

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.

4.4. 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.*

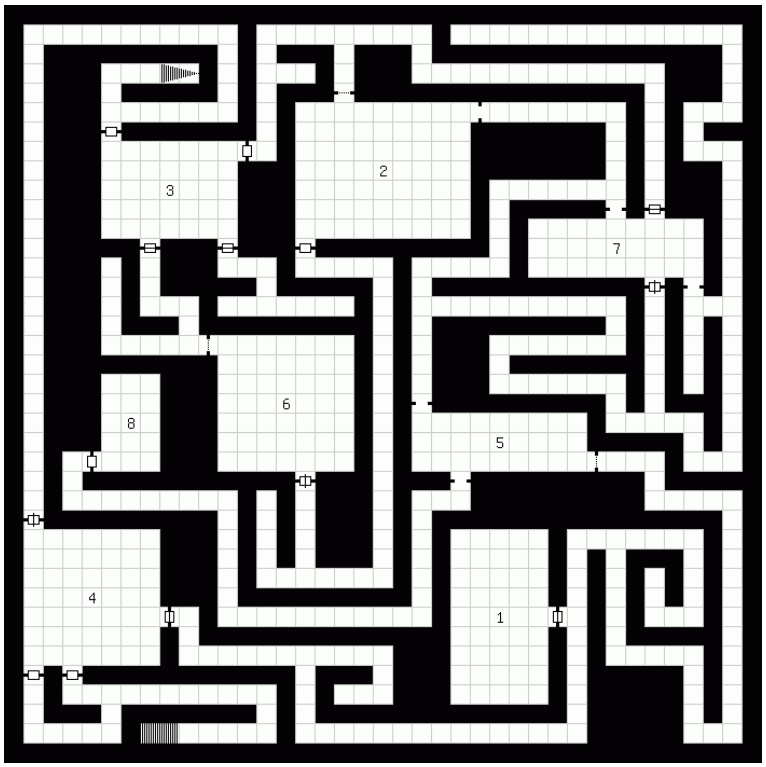


Figura 4.15 Mazmorra generada mediante posicionamiento aleatorio

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.

4.4.1. 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.

4.5. 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**.

4.5.1. ¿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.

Figura 4.16 Mapa inicial con el nodo raíz

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.

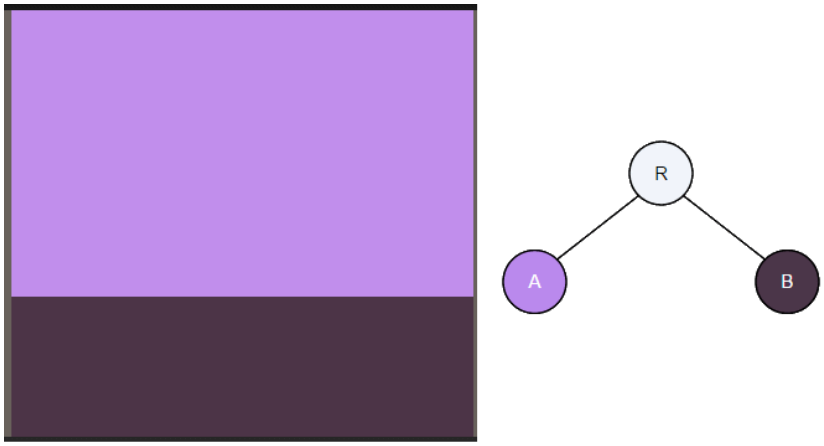


Figura 4.17 El primer corte divide la mazmorra en dos secciones dentro del árbol

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.

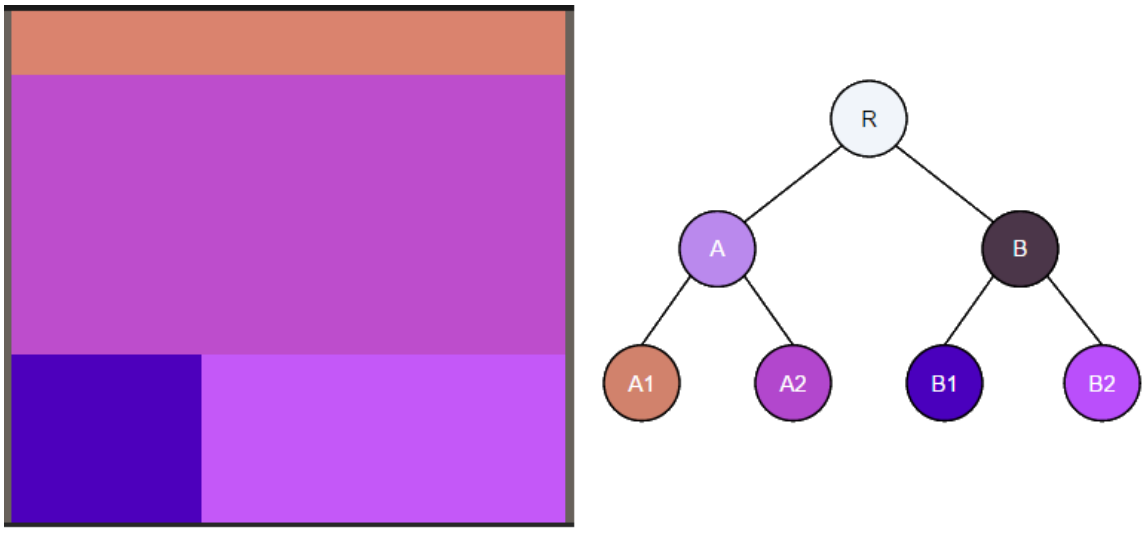


Figura 4.18 Estado después de realizar la segunda serie de cortes

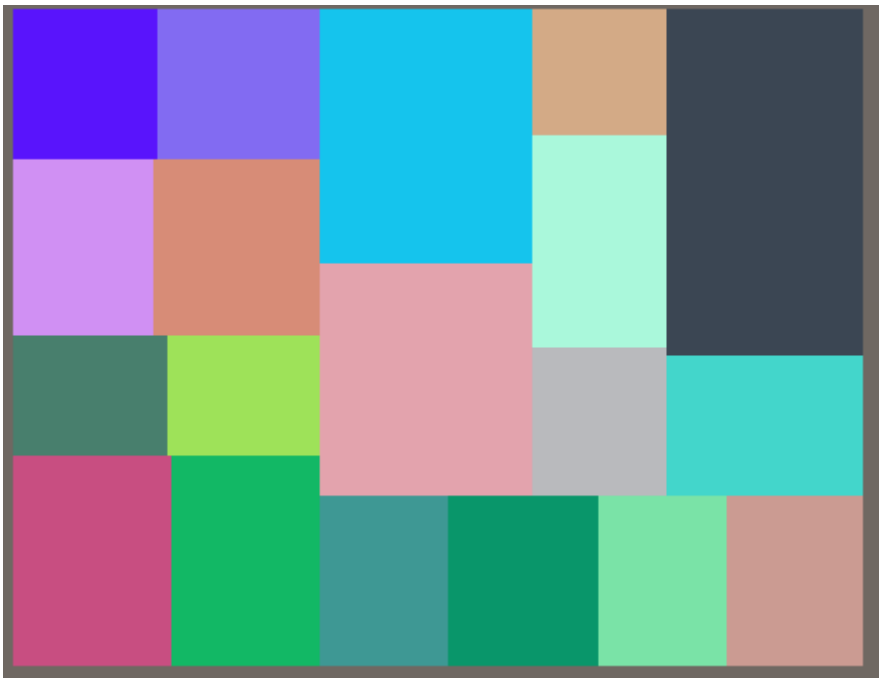


Figura 4.19 Subdivisión final del espacio completo de la mazmorra

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.

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.

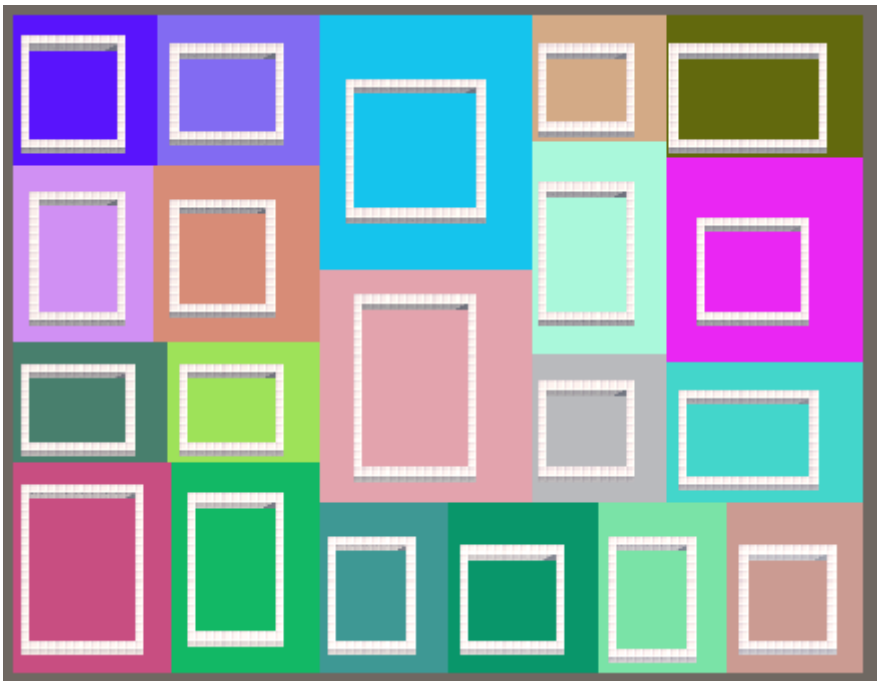


Figura 4.20 Habitaciones

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.

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.

Figura 4.21 Disposición final con los pasillos uniendo las habitaciones

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.

4.5.2. 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
```

Figura 4.22 Generado utilizando semilla fija

4.5.3. 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.
- Asegura la conectividad en toda la mazmorra.
- 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 puede ser demasiado similar, 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.

4.6. 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.

4.6.1. 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).

$$D_{puntuación} = 20 * N_{murciélagos} + 10 * N_{serpientes} + ...$$

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.

Figura 4.23 Dificultad del 10%, 50% y 90%. La densidad de puntos indica la cantidad de enemigos

4.6.2. 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.

4.6.3. 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.

4.6.4. 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.

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 como anexo 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.

5.1. El juego

Con gran inspiración en el juego de 2008, Spelunky, este proyecto se presenta como una implementación 3D de este, pero en un entorno más similar a Rogue (1980) o The Legend Of Zelda (en sus versiones clásicas 2D), donde encontramos habitaciones y pasillos más diferenciados, de construcción humana, en contraste con los entornos del estilo de cuevas naturales de Spelunky. Esto permite implementar trampas específicas para pasillos/habitaciones, que resultarían más difícil de colocar en una disposición irregular de tipo cueva.

El personaje principal se mueve por pasillos y habitaciones, evitando trampas, recogiendo tesoros y eliminando enemigos. El **objetivo** es llegar al final de la mazmorra en el menor tiempo posible y con la mayor cantidad de tesoro.

El juego se compone de las siguientes características:

- Habitaciones conectadas entre sí por pasillos.
- Objetos para mejorar o recuperar la vida.
- Enemigos que patrullan habitaciones.

- Trampas como paneles de pinchos, fosos, bolas rodantes, etc. en habitaciones y pasillos.
- **Generación procedimental**: el juego implementa algunos de los algoritmos estudiados en los capítulos anteriores.

5.2. 5.2 Motor de juego

El juego ha sido desarrollado sobre el motor de juegos Unity 3D debido a que proporciona una gran cantidad de herramientas integradas en un editor visual.

El proyecto se estructura por directorios, comenzando por la base llamado **Assets**, desde donde cuelgan el resto, que son:

- **Animations**: contiene las distintas animaciones, tanto de los sprites como de los objetos 3D, en un formato interno de Unity. También se pueden encontrar los controladores para la activación de estas animaciones. Un objeto puede implementar un controlador con varias animaciones.
- **Audio**: ficheros de sonido y música.
- **Fonts**: fuentes o tipografías que vemos en los distintos aspectos del HUD y menús.
- **Materials**: materiales en formato interno de Unity que van ligados a los modelos 3D, el skybox, partículas, etc. No todos los materiales se encuentran en esta carpeta, ya que los que son importados desde blender se mantienen en la siguiente carpeta.

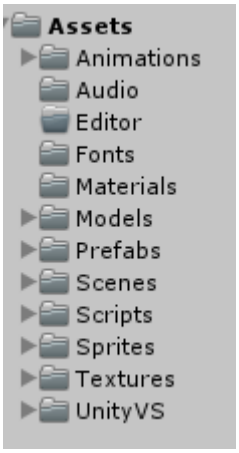


Figura 5.1 Sistema de directorios del proyecto

- **Models**: modelos 3D base que se utilizan para crear los Prefabs de los objetos 3D. La mayoría de estos modelos están en el formato nativo de Blender (.blend) y se pueden abrir con este sin ningún problema. También contiene los materiales asociados a estos objetos.

- **Prefabs**: son el concepto más importante en Unity, se tratan de objetos que sirven como prototipos para instanciar los objetos durante la ejecución el juego. Se crean en su mayoría a partir de los modelos 3D y agregando Scripts que definen su comportamiento.
- **Scenes**: formato de Unity para guardar las distintas escenas del juego.
- **Scripts**: son los ficheros de texto en formato C# donde se encuentra el código del juego que define el comportamiento de los distintos objetos e interacciones. Aquí se definen las distintas clases que se van a nombrar en las siguientes secciones.
- **Sprites**: tiras de imágenes de tipo sprite para los elementos del HUD, menús y personajes del juego.
- **Textures**: imágenes que se usan como texturas para los objetos 3D del juego.

5.3. Generación de mazmorras

Vamos a ver el proceso de generación completo de las mazmorras del juego, y vamos a separarlo en dos partes; la generación y distribución de las habitaciones y pasillos, y la generación del contenido de estas.

5.3.1. 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**.

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, tomando como base el lado en el que se sitúa el nodo.

A continuación vamos a ver los pasos que he seguido para generar una mazmorra completa, las clases y métodos involucrados, sin profundizar demasiado en estos,

pero dando un idea general sobre el flujo que sigue el código de esta implementación.

5.3.2. Parámetros iniciales y unidades

Las clases y objetos que involucran el generador de la mazmorra se pueden encontrar principalmente en las carpetas **Prefabs/DungeonGenerator** y **Scripts/DungeonGenerator**.

- **DungeonGenerator**: es el objeto principal, contiene los grids del nivel, el árbol BSP y es la clase encargada de llamar a los métodos para cortar, generar habitaciones, conectarlas y generar el contenido de la mazmorra.
- **Grid**: define un array bidimensional sobre el que vamos a realizar las subdivisiones del mapa y donde se van a definir los distintos tipos de tiles. En DungeonGenerator vamos a encontrar un grid para el nivel donde se escribirán las casillas de habitaciones y pasillos, por su identificador, y otro grid donde se dibujarán el resto de elementos del juego, como trampas, objetos y enemigos, también definidos por identificadores.
- **Vector2i**: la librería de Unity nos proporciona clases vector de 2 o 3 dimensiones de tipo de datos float, pero durante la generación de la mazmorra vamos a trabajar con una rejilla de posiciones enteras, por lo que he creado esta clase que representa un vector bidimensional de enteros. Incluye métodos de conversión con los vectores de Unity para la etapa final donde instanciamos los objetos a coordenadas del mundo 3D.
- **BSPTree**: clase que representa el árbol binario de partición. Contiene el nodo raiz de este para acceder al resto del árbol navegando por los hijos.
- **BSPNode**: representa un nodo del árbol. Incluye parámetros como posición y dimensiones, rama y profundidad dentro del árbol. También describe métodos para dividir estos nodos y generar sus hijos siguiendo las reglas del algoritmo de partición. Cada nodo representa una zona cuadrada o rectangular dentro del grid.

- **Room**: los nodos del árbol tienen una posición y dimensiones, pero las habitaciones dentro de estos se definen por esta clase, con su posición, dimensiones y un identificador que las distingue únicamente.
- **Digger**: durante la generación de pasillos necesitamos alguna manera de recorrer el espacio entre las habitaciones, creando paredes y espacios de tipo suelo entre estas. La clase Digger se encarga de esto.

Como vamos a trabajar con rejillas, la mejor manera de definir un sistema de unidades estándar es simplemente utilizando la equivalencia con la rejilla del editor de unity. Una celda de los grids de nivel y de objetos equivale a una celda del grid de Unity, que es una unidad de este.

5.3.3. 5.3.3 Generación del árbol

Esta primera parte del proceso es tal y como habíamos descrito en el estudio de este algoritmo, pero vamos a ver cada paso en este contexto concreto.

Antes de comenzar debemos definir una serie de parámetros que describan las características de la mazmorra a generar.

- **DUNGEON_WIDTH/DUNGEON_HEIGHT**: dimensiones de la mazmorra.
- **MIN_ROOM_SIZE**: tamaño mínimo del espacio que ocupará una habitación dentro de un nodo.
- **MARGIN**: Margen entre el borde de los espacios de los nodos y la habitación contenida en estos. Se refiere al margen por cada una de las 4 direcciones. Este margen permite crear pasillos y evita habitaciones demasiado juntas.

- **SEED**: en principio será la clase **Random** de Unity la que decida una nueva semilla en cada generación de mazmorra. Esta semilla se basa en la hora local del entorno donde se ejecuta la aplicación.

1. Creamos un grid de enteros del tamaño indicado en las dimensiones de la mazmorra, este es el **levelGrid**. Inicializamos toda la rejilla al valor de tipo vacío.

2. Creamos el árbol BSP, generando su nodo raíz. Posicionamos el nodo raíz en el centro del mapa.

3. Nos movemos al siguiente nodo por la izquierda. Si se trata de un nodo hoja, no tiene hijos, podemos realizar la partición. Si tiene hijos, vamos al siguiente por la izquierda, llamando al mismo método de forma recursiva y pasando este nodo hasta que lleguemos a un nodo hoja.

3.1 Partición: El mismo nodo (BSPNode) implementa un método por el que decide, con un 50% de probabilidad, si se divide (Split) por el eje 'x' o el 'z'.

3.2 Dividir (SplitX/SplitZ):

a. Primero se comprueba si el área del nodo permite dividir porque cumple un espacio mínimo para generar dos subdivisiones. El espacio mínimo se define como el tamaño mínimo de la habitación más el margen a cada lado multiplicado por dos.

Espacio mínimo del nodo = (MIN_ROOM_SIZE + MARGIN * 2) * 2

b. Si el nodo cumple este requisito entonces se escoge un punto aleatorio entre este mínimo y la longitud del nodo menos este tamaño mínimo. En la siguiente imagen se puede ver más claramente.

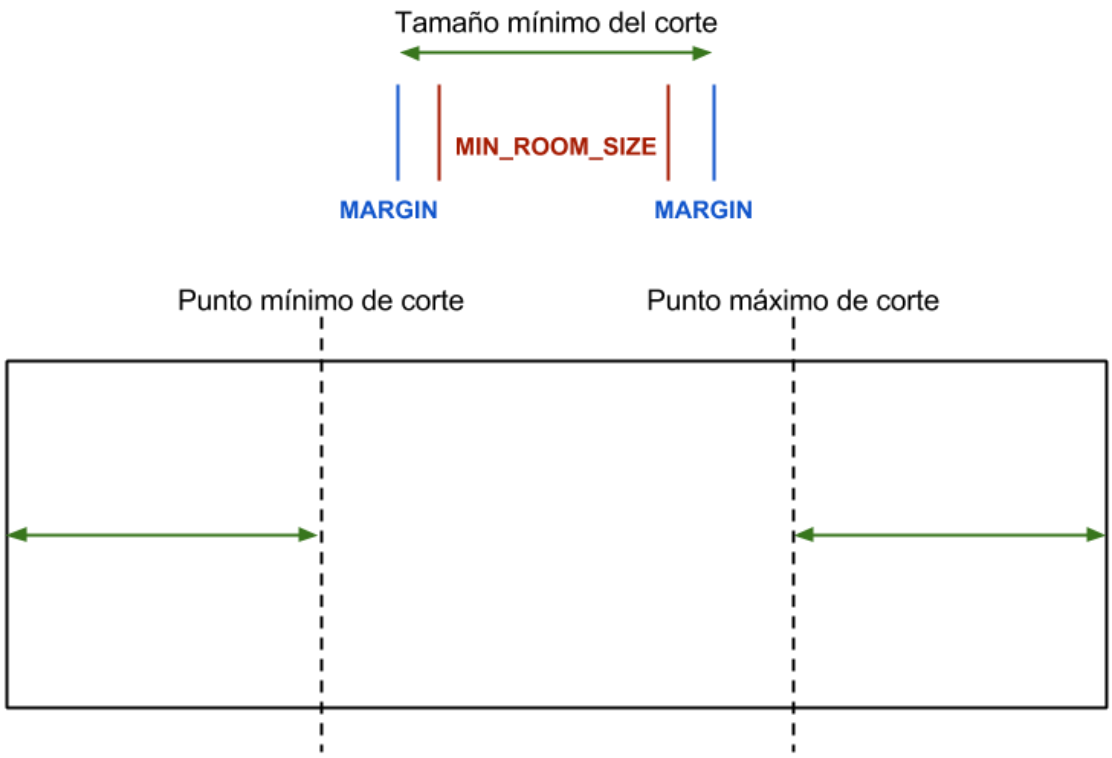


Figura 5.2 Todos los puntos comprendidos entre el mínimo y máximo de corte son candidatos a ser escogidos de manera aleatoria

Figura 5.2: Todos los puntos comprendidos entre el mínimo y máximo de corte son candidatos a ser escogidos de manera aleatoria.

Del valor anterior creamos un nodo hijo izquierdo del nodo actual. Obtenemos el espacio restante y creamos el nodo derecho.

4. Después de cortar el nodo izquierdo volvemos hacia arriba y pasamos a la parte derecha, realizando los mismos pasos del punto 3.

6. Iterar 3 y 4 durante x ciclos. El número de ciclos dependerá del tamaño de la mazmorra habitaciones. En este caso concreto se comienza con una mazmorra de dimensiones 50x50, tamaño mínimo de habitación de 14 unidades de grid y se realizan 8 vueltas. Conforme se avanzan niveles y aumenta el tamaño de la mazmorra se van agregando más vueltas para evitar habitaciones demasiado grandes por falta de divisiones.

5.3.4. 5.3.4 Creando las habitaciones

De la fase anterior hemos obtenido un árbol binario que podemos recorrer de manera recursiva, llegando hasta las hojas donde vamos a crear las habitaciones.

El método **AddRoom** del DungeonGenerator instancia el objeto base para la habitación en la posición del nodo y le asigna unas dimensiones aleatorias entre el tamaño mínimo de la habitación (MIN_ROOM_SIZE) y el tamaño máximo de espacio disponible menos el margen.

Size x = Random (MIN_ROOM_SIZE_SIZE, NODE_SIZE - MARGIN)

Size z = Random (ROOM MIN_ROOM_SIZE SIZE, NODE_SIZE - MARGIN)

Teniendo la posición y dimensiones de la habitación ahora recorremos el grid del nivel estableciendo los tiles de suelo y paredes. Los tiles se definen con enumerados que determina un identificar para cada elemento de la mazmorra. También definimos un tipo para las habitaciones, de manera que podemos saber si se trata de una entrada, salida, pasillo o una habitación normal en la que colocar contenido.

```
// Tipos de habitaciones

public enum RoomType {

  DEFAULT,

  ENTRANCE, // Habitacion inicial donde comienza el jugador

  EXIT,     // Habitacion final de la mazmorra

  PASSAGE   // Pasillo que une dos habitaciones

}

// Tipos de tiles y numero identificador base de habitaciones

public enum TileType {
```

```
  // Vacio

  EMPTY = 0,

  // Suelo

  FLOOR = 1,

  // Items

  COIN = 10,

  CHEST,

  POTION_HEALTH,

  // Trampas

  TRAP_SPIKES_FLOOR,

  TRAP_SPIKES_WALL,

  TRAP_ARROWS,

  TRAP_ROCK,

  ENEMY_CRAB,

  ENEMY_GOBLIN,

  EXIT,

  // Numero base para habitaciones y pasillos

  ROOM_ID = 50

}
```

5.3.5. Conectando las habitaciones

El proceso de conexión de habitaciones se realiza primero 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.

Pero esto solo no permite la conexión completa de la mazmorra, y debemos subir hacia los nodos intermedios del árbol, asignándoles una habitación y después realizando el mismo proceso para conectarlos con sus nodos hermanos.

Así que una vez conectadas un par de habitaciones debemos subir hacia su nodo padre y escoger que habitación de sus dos nodos hijos se le va a asignar, nótese que no estamos generando una habitación nueva, sino que se abrirán nuevas conexiones o pasillos sobre las que ya estaban creadas en un principio.

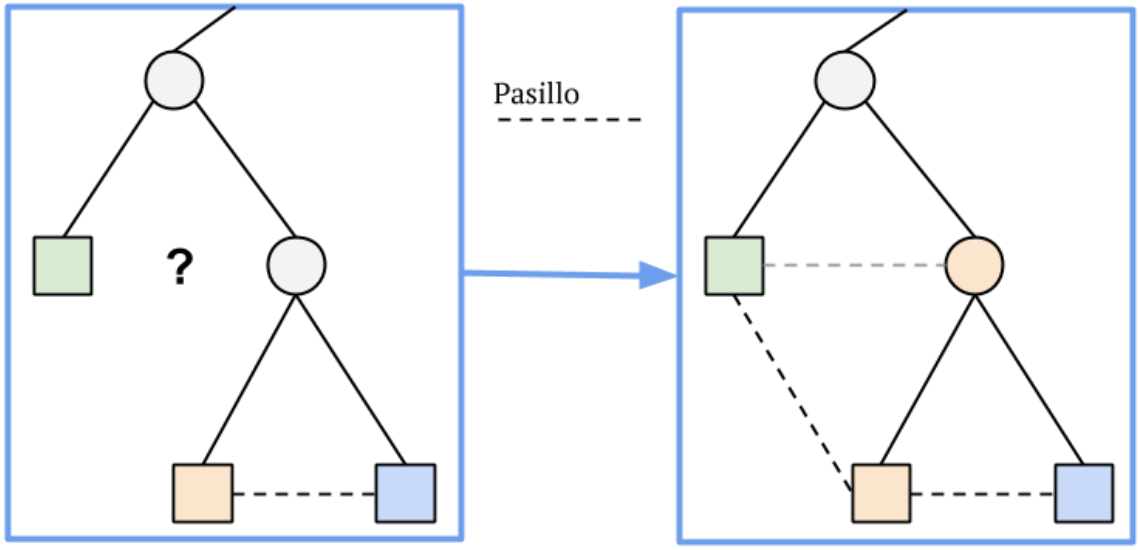


Figura 5.3 A. Los nodos hermanos con habitación ambos se unen sin problema pero los nodos del nivel anterior no tienen forma de hacerlo porque falta una habitación. B. Utilizando algún criterio de selección se asigna al nodo padre una habitación hija y se une con el hermano. No se crean habitaciones nuevas, sino que se unen con las que ya existen.

Este paso de elección es importante ya que genera distintos resultados según como se escojan las habitaciones. Para este juego en concreto he implementado un par de métodos distintos. Ambos obtienen buenos resultados para este proyecto en concreto pero hay que hacer notar sus diferencias:

Elección tipo aleatorio. Si simplemente escogemos una habitación con un 50% de probabilidad vamos a obtener resultados más inesperados ya que podemos encontrarnos con situaciones en las que un pasillo atraviese otra habitación y solape otros pasillos creados con anterioridad.

En la imagen vemos como en el caso más extremo la habitación escogida para el nodo A se une con la escogida para el nodo B, eliminando los pasillos que se habían creado anteriormente.

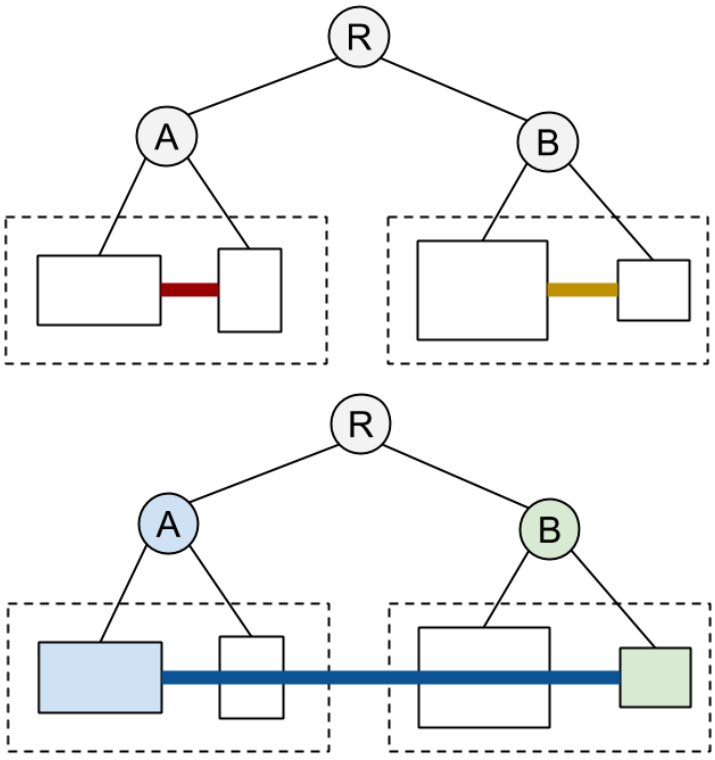


Figura 5.4 La elección aleatoria produce situaciones en las que los pasillos atraviesan habitaciones y solapan otros pasillos

Figura 5.4: La elección aleatoria produce situaciones en las que los pasillos atraviesan habitaciones y solapan otros pasillos.

Visualmente no se producen cambios en las habitaciones implicadas o los pasillos que las unen, pero al repasar el grid del escenario donde se asigna un identificador a cada casilla veremos que estos dos pasillos que unen las tres habitaciones tendrán el mismo número ya que los dos pasillos que había antes han sido solapados.

En este caso en concreto no es de importancia, ya que el único uso que se le da a los identificadores de los pasillos es la diferenciación con respecto a las habitaciones, pero no necesitamos identificarlos independientemente.

Figura 5.5 Porción de una mazmorra con pasillos generados utilizando el método aleatorio. Se obtienen bucles entre habitaciones y “giros” entre habitaciones

Elección por proximidad al centro. Se trata de una técnica muy simple pero efectiva a la hora de evitar que los pasillos se solapen. Se calcula la distancia de cada habitación candidata al centro de la mazmorra. La habitación con una distancia menor es la escogida. Esto determina, en prácticamente casi todos los casos, que la habitación más hacia el interior es la escogida y se evitan los problemas del método aleatorio.

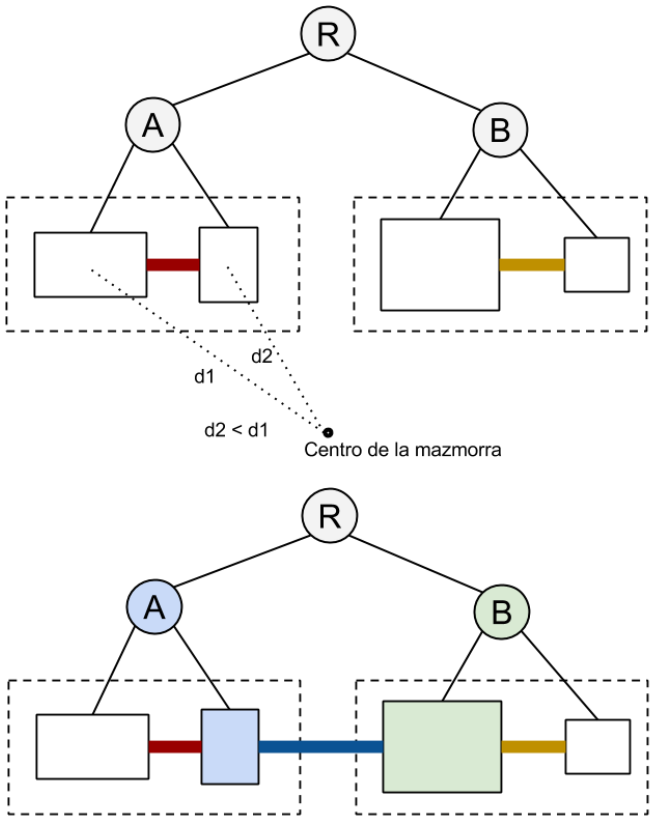


Figura 5.6 Calculando la distancia al centro nos aseguramos de que siempre se unan las habitaciones más cercanas

Este método es superior al anterior cuando queremos diferencia claramente cada pasillo para asignarle un identificador diferente y conocer la conectividad entre las habitaciones en el mismo proceso de creación del pasillo. El resultado que se obtiene visualmente es más regular que en el caso anterior, evitando la posibilidad de que pasillos atraviesen habitaciones. Es más previsible que el método anterior pero también menos interesante en cuanto a variedad.

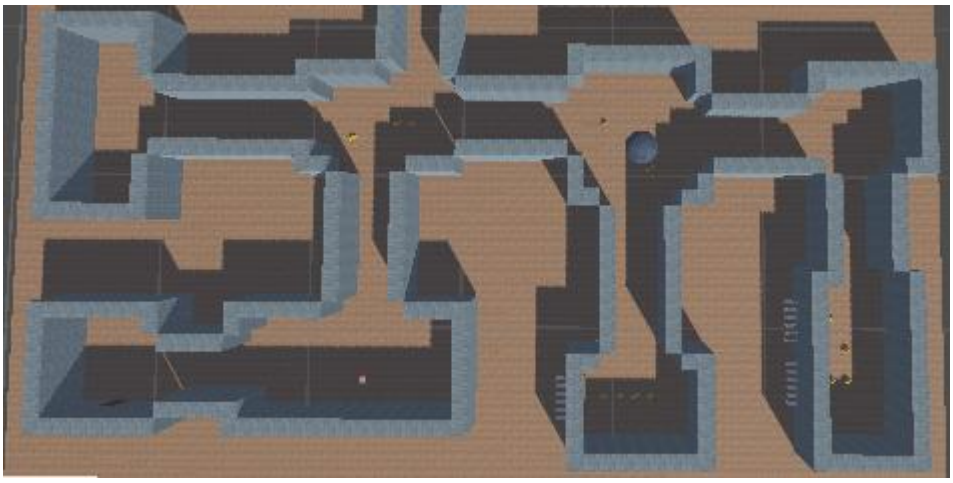


Figura 5.7 Típico resultado al utilizar el método de conexión de pasillos por distancia hacia el centro. Habitaciones y pasillos claramente separados

¿Qué método utilizamos?

Ambos, ya que ambos son interesantes en su modo y ninguno interfiere en la futura generación de contenido de las habitaciones y pasillos. En vez de desechar uno de los métodos vamos a seleccionar, en cada generación de mazmorra, uno de los dos de manera aleatoria, lo cual puede aportar incluso más variedad a la jugabilidad.

5.3.6. Limpiando con autómata celular

Después de haber estudiado los distintos algoritmos e implementado el caso ya explicado, quise agregar un poco más de variedad a las formas de la mazmorra, que eran totalmente rectangulares.

Para ello se realiza un paso adicional para hacer una limpieza de **celdas** "**solitarias**" que puedan haber quedado en la creación de los pasillos y para redondear esquinas dentro de la misma mazmorra

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.

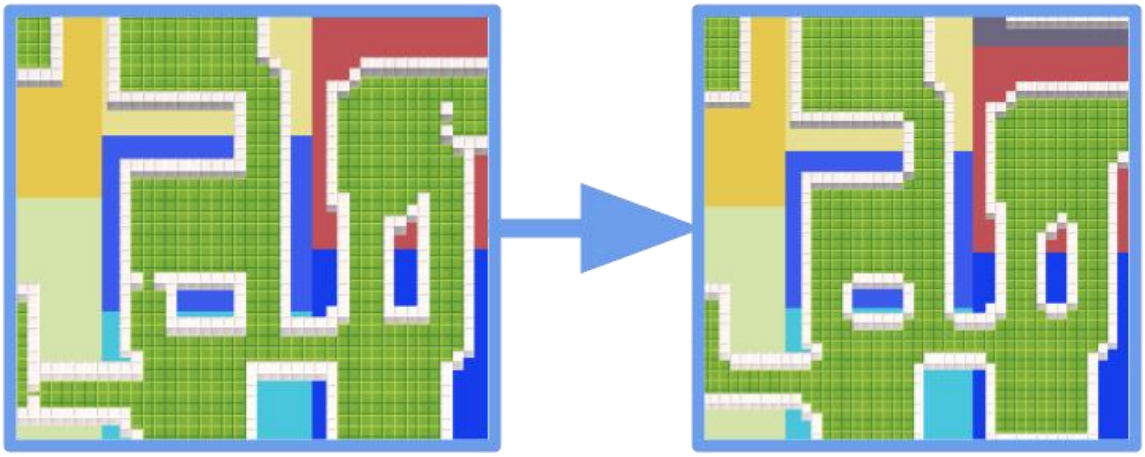


Figura 5.8 Observamos cómo algunos pasillos se terminan de abrir, celdas solitarias se limpian y se redondean los bordes internos de la mazmorra

5.4. Generando el contenido de la mazmorra

5.4.1. Entrada y salida

Antes de empezar a generar contenido debemos determinar la habitación de entrada y la de salida. Para ello nos servimos del mismo árbol, que ya en el primer corte divide la mazmorra en dos zonas separadas, por lo que podemos escoger, por ejemplo, la rama izquierda para colocar la entrada y la derecha para la salida.

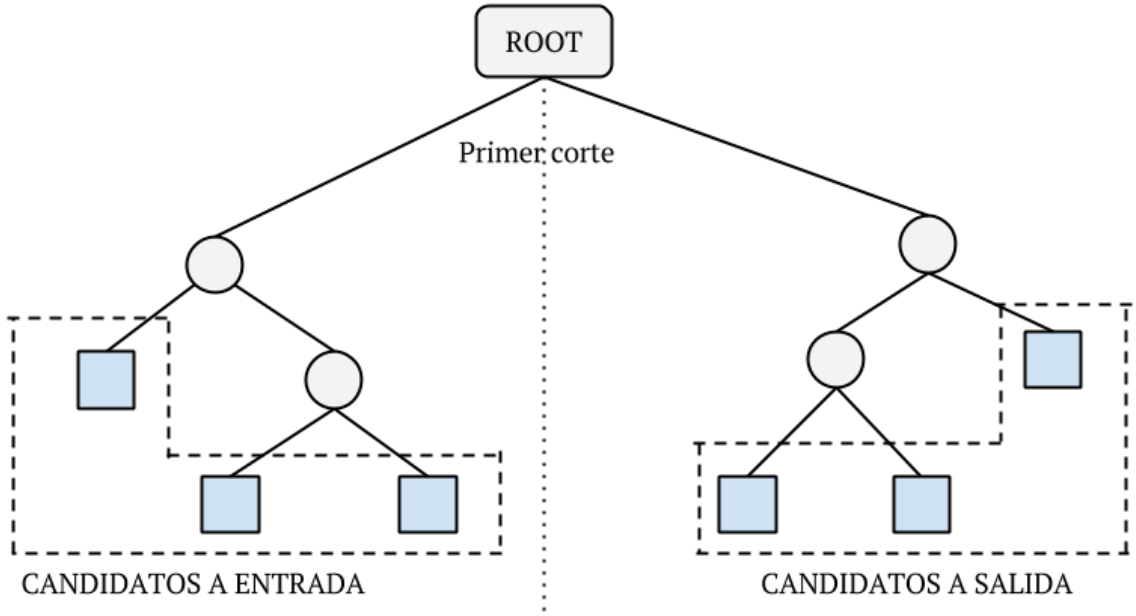


Figura 5.9 El primer corte determina los grupos de candidatos a entrada y salida

La colocación de la **entrada** se hace simplemente escogiendo un nodo **al azar**.

Para escoger la salida tenemos que tener en cuenta que no queremos que se genere justo al lado de la entrada, ya que aunque el primer corte asegura que están en dos mitades distintas de la mazmorra, esto no quiere decir que al unirlas mediante pasillos no se vayan a unir directamente la habitación de entrada con la de salida, ya que al menos habrá un pasillo uniendo dos habitaciones de cada rama.

Por ello para determinar la salida vamos a tomar la habitación de entrada y medir la distancia desde esta a todas las habitaciones del lado derecho del árbol. La habitación más lejana será la de salida. Las pruebas han demostrado que esto es un método que funciona siempre, dejando suficiente espacio entre las dos habitaciones. También cabe recordar que no es necesario que la salida sea la última habitación, o la que se encuentre en la esquina más lejana, ya que el objetivo del juego es recoger la mayor cantidad de tesoro antes de salir.

5.4.2. Generación de objetos

Una vez tenemos la distribución de habitaciones y pasillos y hemos establecido el comienzo y final del nivel podemos poblar la mazmorra con contenido, vamos a ver qué criterios he seguido para esto.

El método para rellenar la mazmorra es el más simple de los presentados en el estudio de algoritmos de generación de contenido, basado en un conjunto de condiciones y probabilidades. Se escoge esta algoritmo no solo por su simpleza, sino porque parece el más adecuado para un juego de este estilo, similar a Spelunky, donde la dificultad se determina por la cantidad y valor en dificultad de los enemigos y trampas y depende del número de niveles avanzados en el juego.

Antes de nada cabe decir que los valores de probabilidad presentados a continuación se basan en la experiencia de las pruebas realizadas durante el desarrollo del juego.

Es importante establecer un orden en la generación de los distintos tipos de contenidos. Tenemos objetos, trampas y enemigos. Los objetos se crean los primeros, ya que de estos dependen las trampas y enemigos.

La generación de **objetos** se produce recorriendo las habitaciones de la mazmorra y haciendo un simple lanzamiento de dados en cada una de estas, mediante un sistema de **probabilidades**, siguiendo los siguientes criterios:

- Las **monedas** aparecen en habitaciones con un **60%** de probabilidad. Son el ítem más común. Una vez seleccionada esta probabilidad se pueden crear entre 1 y 6 monedas por habitación, dependiendo en las dimensiones de la habitación, cuanto más grande sea la probabilidad se mueve hacia un mayor número de monedas.

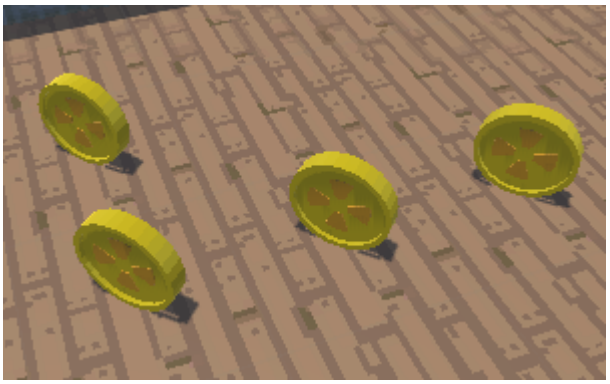


Figura 5.10 Monedas

- Los **cofres** aparecen con un **20%** de probabilidad inicial, por habitación. Siendo esta afectada por la cantidad de cofres que ya se han creado en esa misma mazmorra. Cada cofre nuevo que se crea **reduce** en un **5%** la probabilidad de creación de cofre en la siguiente habitación.

Factor de reducción (**R**) = 5%, **Probabilidad inicial** (**Pi**) = 20%

NÚMERO DE COFRES (N)	**PROBABILIDAD (Pi – R * N)**
0	20%
1	15%
2	10%
3	5%
4	0%

Tabla 1 Probabilidad de aparición de cofre

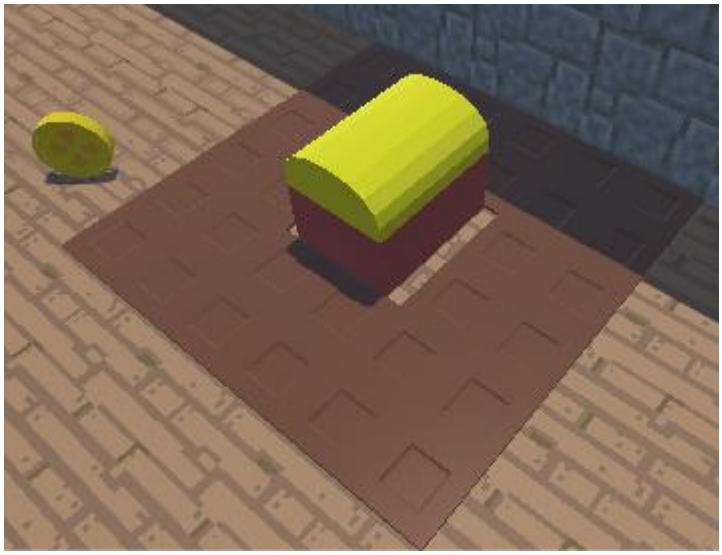


Figura 5.11 Cofre

- Las **pociones** se crean también con un **20%** de probabilidad inicial que se **reduce** en un **10%**, por lo que solo podremos encontrar 2 de estas por mazmorra.

Factor de reducción (**R**) = 10%, **Probabilidad inicial** (**Pi**) = 20%

NÚMERO DE POCIONES (N)	**PROBABILIDAD (Pi – R * N)**
0	20%
1	10%
2	0%

Tabla 2 Probabilidad de aparición de pociones

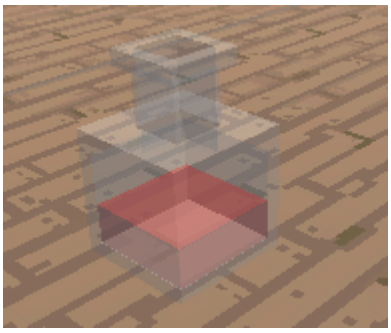


Figura 5.12 Poción de vida

El posicionamiento de los objetos dentro de la habitación se escoge de manera aleatoria, manteniendo un espacio con respecto a la pared, de manera que luego sea más fácil colocar trampas alrededor de estos.

Internamente se mantiene una rejilla, **objectGrid**, donde cada casilla se marca con un entero indicando el tipo de objeto en ese tile.

5.4.3. Generación de trampas

Las trampas se generan en dos fases, una para los pasillos y otra para las habitaciones.

Una vez se ha terminado la generación de objetos en una habitación, lo que se hace es recorrer las casillas del grid de objetos que pertenecen al área de esa **habitación**, según el objeto que se encuentre se actúa de las siguientes maneras:

- **Moneda:** las monedas cercanas a alguna pared provocarán que se generen trampas de **pinchos de pared** en los tiles de pared cercanos a estas. Se crean tres trampas por moneda, por lo que se producen combinaciones interesantes cuando hay varias monedas cercanas a las paredes de la habitación. Obliga al jugador a tener cuidado con el momento en que se acerca a recoger la moneda.
- **Cofre**: cuando se encuentra un cofre siempre va a estar protegido por algún tipo de trampa. En este caso serán los **pinchos de suelo** o el **dispensador de**

flechas. En el primer caso se genera pinchos en todas las casillas adyacentes a la que contiene el cofre, obligando a saltar hacia este para abrirlo. En el segundo caso se crean dispensadores de manera que siempre se protegen al cofre, no pudiéndose acercar el jugador sin activar al menos uno de estos.

La segunda fase de la generación de trampas es la que se realiza sobre los pasillos. Esto ocurre una vez terminada la población de las habitaciones, y se recorre la rejilla del escenario determinando los pasillos según las casillas contiguas de estos. Distinguimos los pasillos de las habitaciones por el número de estas, si el identificador de la casilla es mayor que el identificador base (ROOM_ID = 50) más el número de habitaciones creadas, entonces se trata de un pasillo.

Una vez que se encuentra un pasillo se cuentan las casillas contiguas de este para determinar su longitud, si es mayor de 10 casillas entonces es candidato para colocar una trampa de roca rodante.

En el resto de pasillos más pequeños pueden aparecer dispensadores de flechas, siempre en grupos de mínimo 3. Para mantener el pasillo superable sin recibir daño se utilizan los parámetros de comienzo y recarga de la trampa para sincronizarlos. Una vez se coloca el primero el momento de comienzo del siguiente se retrasa un poco para que las flechas no se disparen al mismo tiempo. El jugador debe estar atento al patrón de las trampas para pasarlas en el momento adecuado.

5.4.4. Generación de enemigos

Finalmente creamos enemigos en algunas de las habitaciones.

Durante la creación de los propios ítems en cada habitación y después de colocar las trampas en esta procedemos a colocar enemigos siguiendo los siguientes criterios:

- Los enemigos tienen la capacidad de patrullar alrededor de un cierto punto, por lo que se hace perfecto para las habitaciones con un cofre. Si hay un

cofre en la habitación entonces hay una de probabilidad de que el cofre esté patrullado por un enemigo. La probabilidad depende del nivel/número de la mazmorra, comenzando en la primera mazmorra con un 50% de probabilidad de que un cofre tenga a un enemigo patrullándolo, y subiendo un 5% cada vez que se avanza de nivel hasta llegar a un 90% de probabilidad.

- Cuando en la habitación hay menos de 3 monedas entonces hay un 20% de probabilidad de que podamos encontrar un enemigo. Si se encuentran 3 o más monedas entonces habrá un 100% probabilidad de que aparezca un enemigo.
- Cuando una habitación está vacía o es la habitación final de mazmorra entonces siempre habrá al menos un enemigo. Dependiendo del tamaño de la habitación podrá haber hasta 4 enemigos en esta.

5.5. Dificultad de la mazmorra

Para determinar la dificultad de la mazmorra como habíamos estudiado en el capítulo anterior vamos a utilizar el método de **Spelunky**.

Hacemos una suma ponderada de los enemigos, es decir, un valor que les damos según su dificultad multiplicada por la cantidad de estos en el nivel. Entonces se suman y finalmente les restamos el valor de las pociones.

AGENTE	**VALOR**
Cangrejo	5
Goblin	10
Pociones	5

Tabla 3 Relaciones de agente-valor

De esta manera podemos obtener un valor de dificultad de la mazmorra

Dificultad = número cangrejos x 5 + número goblins x 10 – número pociones x 5

Este valor toma sentido al compararlo con distintos niveles y la experiencia del usuario. De esta manera podemos determinar una escala de dificultad.

ESCALA DE DIFICULTAD

ESTA PARTE ESTÁ POR TERMINAR MIENTRAS REALIZO PRUEBAS SUFICIENTES PARA DETERMINAR UNA ESCALA DE DIFICULTAD

5.6. Elementos visuales

5.6.1. Sprites 2D/Billboards

Como se explica en el documento de diseño adjunto, se quiere obtener un estilo visual con personajes 2D sobre entornos tridimensionales, tal y como presentan algunos juegos de la época la Playstation, principalmente de estilo acción aventuras y rol.

A pesar de ser principalmente un motor de juegos 3D, Unity tambien porporciona herramientas para 2D, por lo que tenemos el objeto Sprite que podemos incluir en el entorno 3D y con una cámara en perspectiva sin ningún problema.

Al ser objetos 2D nos debemos asegurar de que siempre miren hacia la cámara. Para ello se ha creado un script ObjectLookAtCamera.cs que gira un objeto hacia esta. Utilizando el vector forward del objeto, que indica la dirección local del objeto "hacia delante" de este, podemos igualarlo al de la cámara.

```
// Obtenemos el vector forward de la cámara

Vector3 forwardDirection = cameraInstance.transform.forward;

// Cancelamos la direccion y

forwardDirection.y = 0;

// Establecemos el vector forward del objeto que va a ser girado

transform.forward = forwardDirection;
```

El problema con esto viene cuando tenemos, por ejemplo, un enemigo que se mueve automáticamente, pero al girar la cámara se ve afectado por este script y cambia su dirección de movimiento. Para solucionar esto simplemente separamos el objeto padre del objeto sprite, y es a este último al que le asignamos el script.

Figura 5.13 El objeto y el sprite se separan pero están enlazados

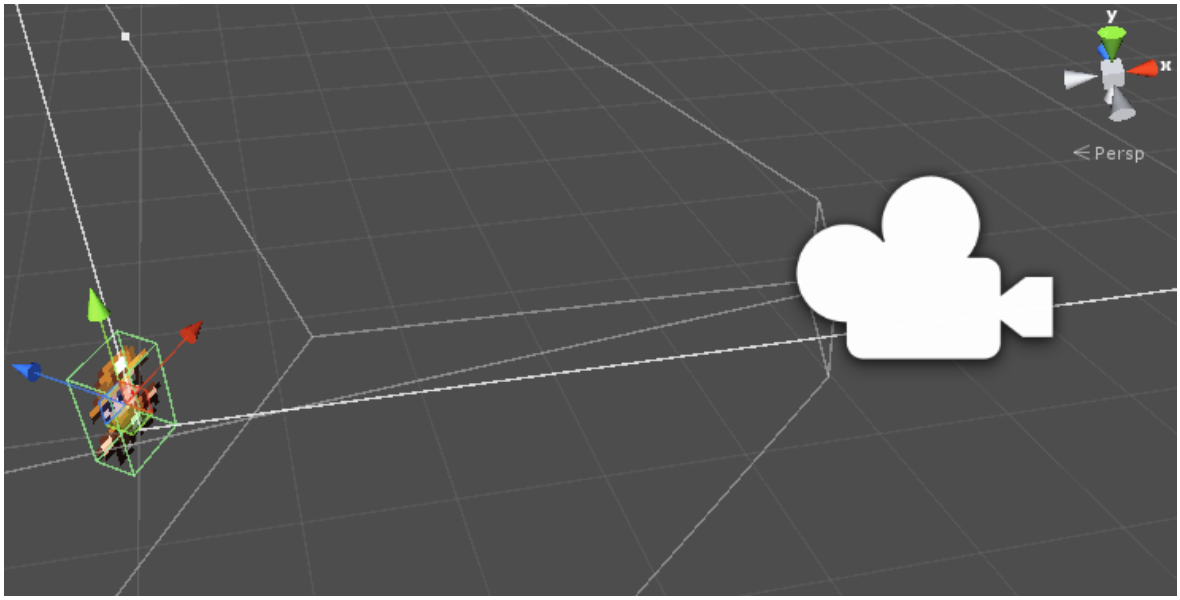


Figura 5.14 El sprite mira hacia la cámara rotando en el eje ‘y’

Hablando sobre las cajas de colisión, a pesar de tratarse de sprites, no podemos utilizar los box colliders 2D que proporciona Unity para este tipo de objetos ya que las físicas 2D y 3D no interactúan entre ellas. Por ello asignamos cajas de colisión 3D, ajustando su escalado en 'z' para obtener los mejores resultados según cada objeto. Si la caja tiene una profundidad demasiado pequeña entonces vamos a ver problemas como que los sprites desaparecen a pegarse a una pared, ya que el sistema de colisiones permite que estas penetren un poco antes de expulsarlos fuera, si la caja es demasiado grande entonces la percepción de la profundidad se vuelve confusa para el jugador.

Después de realizar muchas pruebas se ha encontrado que los mejores valores se encuentran entre **0.3 y 0.5** unidades. En este juego lo que queremos conseguir con estos valores es, sobre todo, que el jugador no se frustre al intentar golpear a los enemigos, y que tampoco de la sensación de que puede dañarlos prácticamente desde cualquier ángulo.

Figura 5.15 El ataque del jugador instancia una caja de colisión que daña a los enemigos

5.6.2. Modelos 3D

Para la creación de los modelos 3D se ha utilizado la herramienta Blender. Podemos encontrarlos en la carpeta Models del proyecto, pudiendo abrirlos directamente con Blender, ya que Unity permite leer este formato directamente.

Los objetos de esta carpeta incluyen sus propios materiales y texturas, mantiendo de esta manera, su sincronización entre los dos sistemas.

Para evitar problemas con las diferencias de escala entre Blender y Unity me he asegurado de crear los objetos en Blender con las unidades adecuadas, sabiendo que una unidad de la rejilla de Blender corresponde con una unidad en Unity, por lo que he tenido esto en cuenta y escalado los objetos en Blender antes de importarlos. Por ejemplo, un cubo en Blender ocupa por defecto 4 unidades de rejilla, pero en Unity solo una, por lo que para las trampas he tenido que reducir a la mitad el tamaño de estas para que entren en una casilla de la rejilla del mapa.

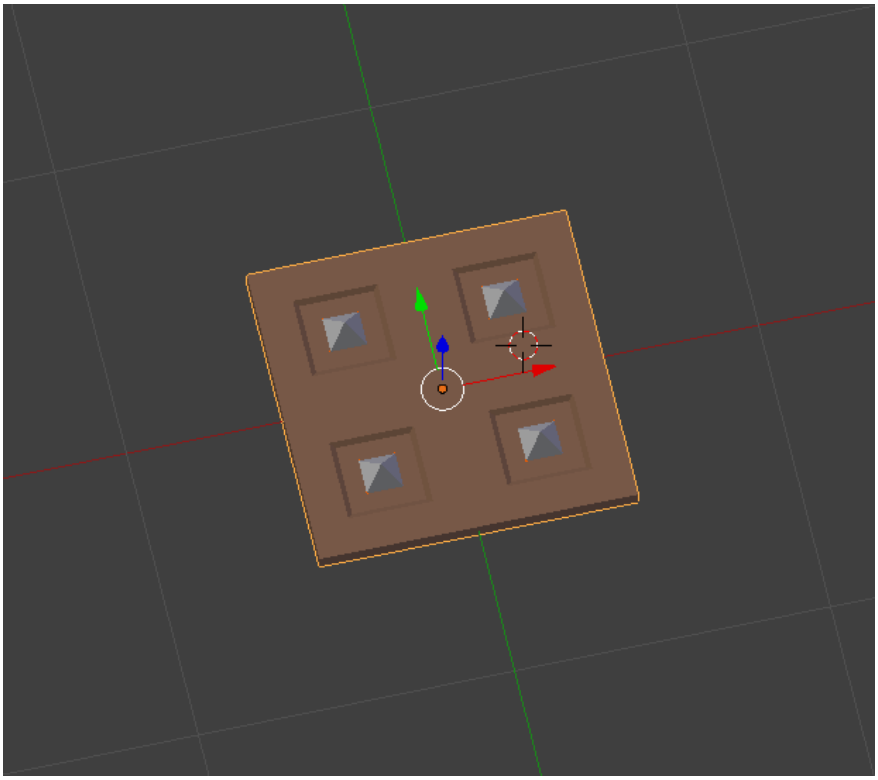


Figura 5.16 Las trampas ocupan una unidad de rejilla pero se pueden agrupar varias para cubrir una zona más amplia

5.6.3. Texturas en elementos dinámicos

Uno de los retos que surgen al trabajar en un entorno que se genera de manera dinámica es la imposibilidad de crear todos sets de escenario de antemano y simplemente posicionarlos en tiempo de ejecución. Esto es obvio para elementos como las paredes o el suelo, que parten de un objeto base con un escalado de una unidad en sus tres dimensiones y que finalmente son re-escalados para cubrir el espacio de esa pared o zona del suelo o techo. Aquí vemos como, sin aplicar ningún tipo de tiling al material de la pared, la textura se estira o se comprime para cubrir cada cara del objeto.

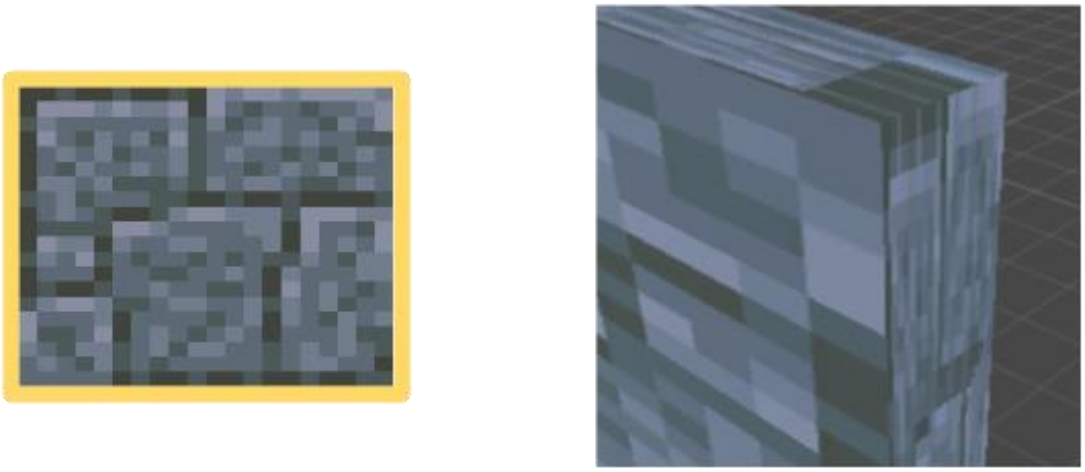


Figura 5.17 Textura sobre una pared con un tiling de 1x1

Por ello es necesario establecer el tiling al valor correcto una vez se ha terminado de escalar la pared hasta sus dimensiones finales. Unity permite modificar estos valores desde el mismo editor, y mediante código podemos acceder al material y modificar el tiling:

```
GetComponent<Renderer>().material.mainTextureScale =
new Vector2(transform.localScale.x, transform.localScale.y);
```

Pero al hacer esto nos encontramos con el siguiente resultado:

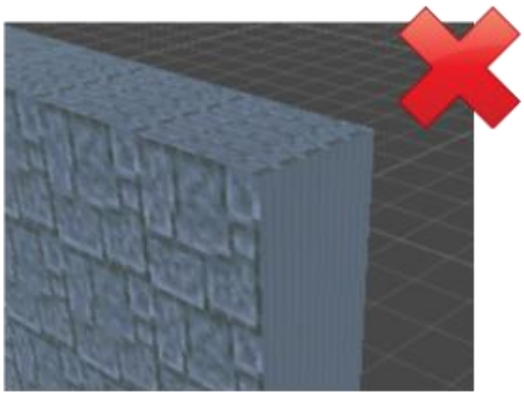


Figura 5.18 Tiling aplicado teniendo en cuenta solo el escalado en x,y

El tiling se ha aplicado correctamente sobre un lado de la pared, pero no sobre el resto. Esto se debe a que las texturas son elementos 2D, y el material que nos presenta Unity por defecto aplica la escala a todas las caras del objeto 3D por igual.

Para arreglar este problemas debemos obtener el escalado del objeto en sus 3 ejes y aplicarlo sobre las texturas de cada cara por separado. El componente Mesh de los objetos de Unity nos permite acceso a cada una de las caras de este y desde donde podemos modificar sus UV para aplicar la orientación correcta de la cara, para finalmente establecer el material a modo de repetición y obtener el resultado correcto.

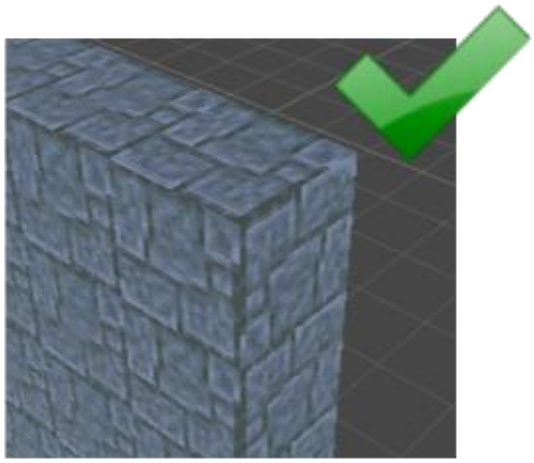


Figura 5.19 Tiling aplicado por caras

5.6.4. Animaciones

Las animaciones tanto para los modelos 3D como para los sprites 2D en Unity se agrupan dentro de un mismo sistema, lo que facilita establecer los triggers de cada estado, independientemente del tipo de objeto.

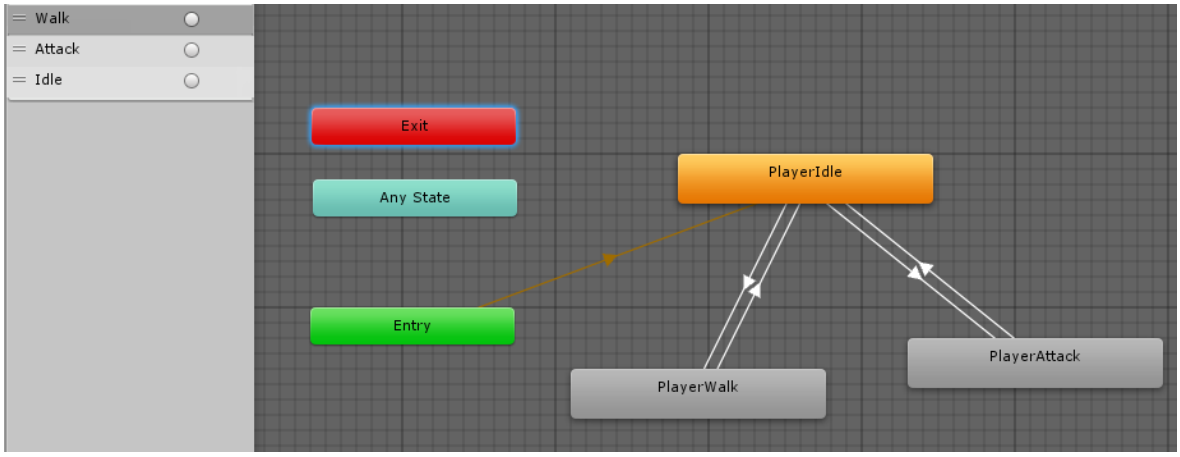


Figura 5.20 Controlador de animaciones del jugador

5.6.5. Menús y HUD

Una de las ventajas del uso de un motor como Unity es que incluye un buen sistema de interfaz, con botones, checkboxes, sliders y lanzamiento de eventos. En el juego podemos encontrar diversos menús con un estilo visual común. Además se han creado diversos scripts para separar el comportamiento y eventos de cada uno de estos:

- **UIManager**: maneja la activación de botones para cambio de escenas y el menú de pausa.
- **UIHelper**: permite a un objeto de tipo texto actualizarse a sí mismo cuando está asociado a un slider.
- **UIMenuHelp**: muestra/oculta el menú de ayuda en la escena de tests de generación.
- **PauseMenu**: maneja la lógica del menú de pausa.

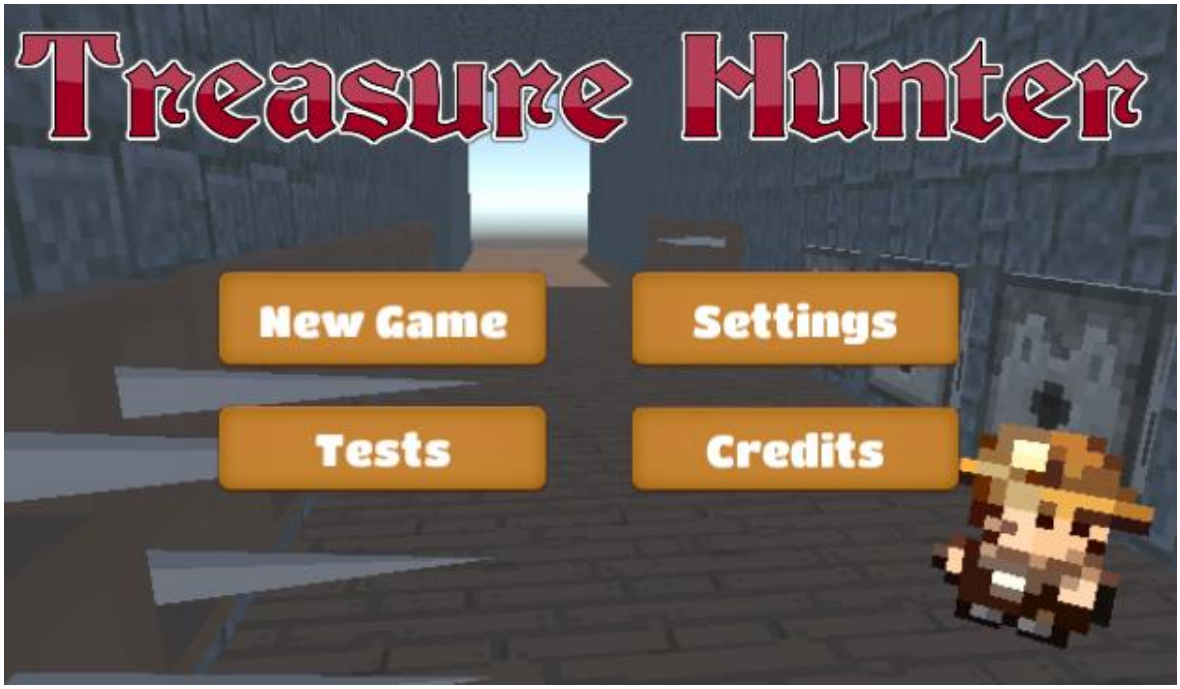


Figura 5.21 Menú principal

En cuanto al HUD, los elementos que vemos durante la partida y que se refieren al jugador, encontramos el script Hud.cs que se encarga de actualizar los elementos visuales cuando se produce algún cambio en estos.

Por ejemplo, se ha creado sistema de vida por corazones que se actualiza con la vida del jugador.

Figura 5.22 Sistema de corazones de vida

5.7. Inteligencia artificial

Los enemigos implementan una inteligencia artificial muy básica basada en una máquina de estados. Estos pueden estar detenidos en un lugar, patrullar alrededor de un punto, perseguir al jugador y atacarle.

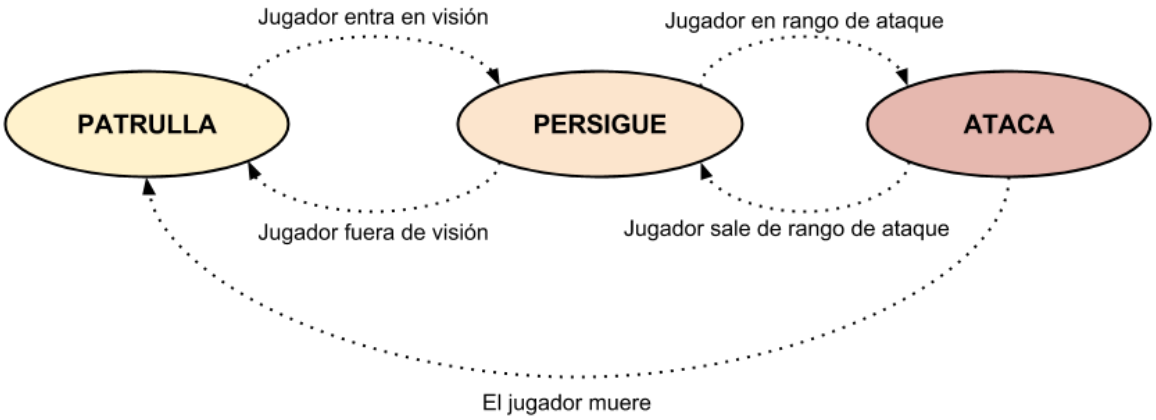


Figura 5.23 Sistema de estados de inteligencia artificial

Al tratarse de una implementación sencilla, se ha desarrollado directamente sobre la clase **Enemy** (Scripts/Enemies/Enemy.cs). Basándose en su situación, el agente realiza una decisión en cada actualización y activa la acción del estado correspondiente.

```
void Update()

{

  DecidirAccion();

  switch (estado)

  {

    case AIState.Idle:

      Espera();

      break;

    case AIState.Patrolling:

      Patrulla();
```

```
      break;

    case AIState.Chasing:

      Persigue();

      break;

    case AIState.Attacking:

      Ataca();

      break;

    case AIState.Dead:

      AnimacionMuerte();

      break;

  }

}
```

Tanto en modo de patrulla como persecución se establecen unos objetivos hacia los que moverse de forma directa. Como nos encontramos en un entorno de tipo mazmorra, con muchas paredes alrededor, y para evitar que los enemigos se encuentren constantemente estancados con estas o con los diversos objetos de las habitaciones, este decide otro objetivo cuando se realiza una colisión con una pared, con lo que en el peor de los casos este se estaría moviendo de un lado a otro de la habitación sin problema.

5.8. Cámaras

5.8.1. Principal

Se trata de una cámara estándar de proyección en perspectiva que se sitúa detrás del jugador a modo de tercera persona. La cámara puede girar sobre el eje vertical para orientarse mediante el **ratón** o el **teclado** (teclas **J y L**). Este giro se realiza de manera suave, aplicando una aceleración y fricción al movimiento. El código para este comportamiento lo podemos encontrar en el script **CameraLookAt.cs**.

También se incluye un script de **cámara libre**, pero este está orientado exclusivamente al testeo del juego. Se puede alternar entre la cámara fija y la libre mediante la tecla **C**.

Como se trata de un entorno cerrado y una cámara en tercera persona, las **paredes** pueden **bloquear la vista**. Por ello se ha implementado un script con el que esta cámara utiliza raycasting para detectar los objetos de escenario entre ella y el jugador y modifica los materiales de estos objetos obstructores para que se vean semitransparentes. Cuando estos objetos dejan de estorbar se restauran a su material por defecto.

Figura 5.24 Vista del jugador a través de una pared

A grandes rasgos, el proceso que vemos en el fichero **CameraClearVision.cs** es como sigue. Tenemos en cuentas que el script va asignado a la cámara, por lo que la palabra **this** va referida a esta:

```
// Lanza un rayo desde la camara hacia el jugador y recoge los hits

Vector3 direction = player.transform.position - this.transform.position;

RaycastHit[] hits = Physics.RaycastAll(this.transform.position, direction);



// Recorremos la lista de objetos que intersectan

for (int i = 0; i < hits.Length; i++)

{
```

```
  if (hits[i].transform.tag == "Wall")

  {

    // Si la pared no estaba en la lista cache, se agrega

    // y se modifica su material

    if (cachedObjects.Find(item => item == hits[i].transform.gameObject) ==
null)

    {

      Material objectMaterial =
hits[i].transform.gameObject.GetComponent<Renderer>().material;

      // Cambiamos el material para que sea semitransparente

      // Utilizamos un material basado en partículas que proporciona Unity

      objectMaterial.shader = Shader.Find("Particles/Additive");

      if (objectMaterial.HasProperty("_Color"))

      {

        objectMaterial.color = new Color(30, 30, 30);

      }

      cachedObjects.Add(hits[i].transform.gameObject);

    }

  }

}
```

5.8.2. Minimapa

Durante el transcurso de una partida podemos encontrar una pequeña aerea del mapa, mostrando la posición del jugador. Esto no es más que una cámara secundaria de proyección ortográfica, con unas dimensiones de viewport a un 30% del total de la pantalla, con el flag de solo profundidad, con lo que no se ve el skybox y se realiza un culling sobre los elementos de la capa Escenery y Minimap.

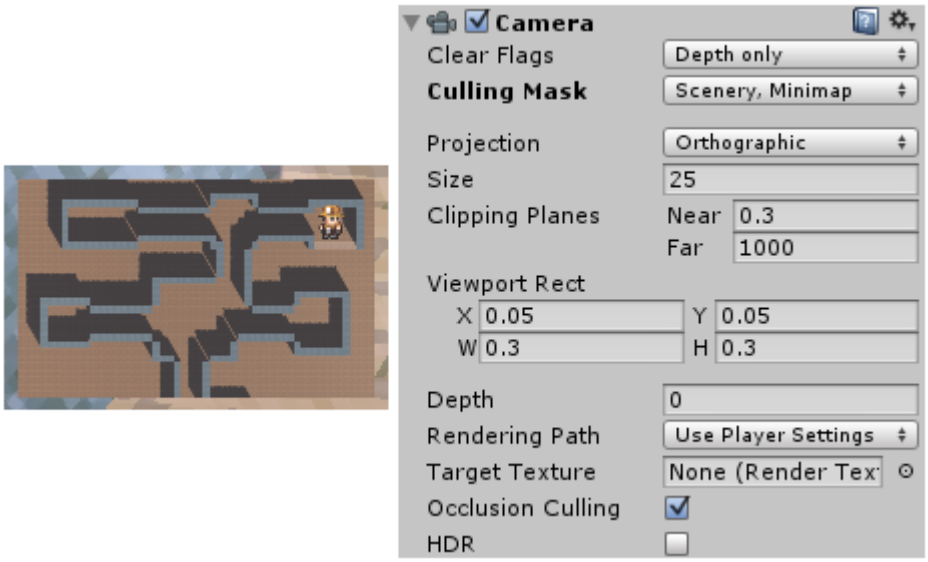


Figura 5.25 Configuración de la cámara ortográfica del minimapa

El culling nos permite mostrar solo los elementos que se encuentran en las capas de escenario y minimapa, es decir, el suelo, paredes y el sprite que muestra la posición del jugador. Este sprite es simplemente un Quad situado delante de la cámara, luego el script **CameraMinimap.cs** es el encargado de modificar la posición de este para coincidir con la del jugador.

5.9. Sonido y música

La mayor parte de sonidos y música del juego ha sido obtenido en Freesounds y sitios similares, así como la herramienta online **bfxr** para generar alguno sonidos como el de salto.

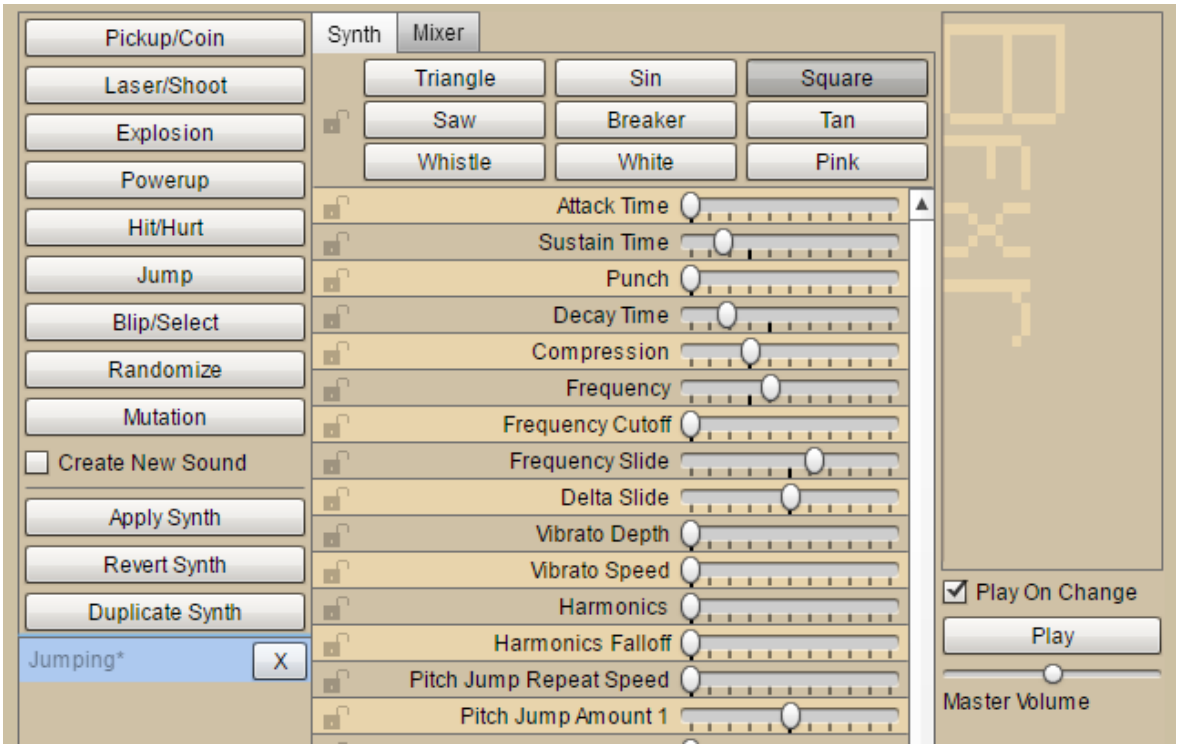


Figura 5.26 Bfrx proporciona gran variedad de controles para generar sonidos únicos

Por otro lado el sistema de sonido de Unity nos permite establecer sonidos de manera posicional, es decir, que varían en volumen según la posición del jugador con respecto a estos.

5.10. Problemas encontrados durante el desarrollo

El desarrollo del proyecto no ha estado falto de dificultades. A continuación presento algunos de los problemas que considero interesantes mencionar y presento mi solución a estos.

Billboards

Aunque hemos visto lo sencillo que es crear un sprite de estilo billboard que siempre mira hacia la cámara, llegar a ese resultado conllevó diversos dolores de cabeza. Por un lado tenemos que tener en cuenta el sistema de unidades de Unity, donde la escala de los elementos 2D no tienen una escala que se corresponda 1 a 1 con la de los objetos 3D, donde, por ejemplo, un cubo por defecto es de 1x1x1 equivalente al espaciado en la rejilla del editor visual. Pero los sprites tienen un escalado expresado en píxels de la ventana 2D. Por ello se hace necesario escalarlos para obtener un tamaño deseado, en este caso en particular son de 5x5x1.

Esto puede traer problemas al aplicar rotaciones sobre el objeto padre que contiene el sprite, ya que el error de las operaciones se acumula y produce un efecto de deformación sobre el mismo sprite. Por ello es importante realizar el escalado sobre el mismo objeto Sprite y no sobre su padre, con lo que el error no se acumula de uno a otro.

Por otro lado tenemos el problema de las sombras. El SpriteRenderer de Unity no funciona por defecto con las sombras y los personajes se ven fuera de lugar en el entorno 3D, haciendo difícil la orientación del jugador sobre el suelo, sobre todo en los saltos. La solución más simple que se ha utilizado es crear objetos anidados que representan la sombra y con un script mantenerlos siempre en el suelo. En este caso es efectivo ya que la orientación de la luz es siempre la misma.

Blender vs Unity

Otro de los dolores de cabeza es la conexión entre Unity y Blender, desde el cual se han creado algunos de los modelos del juego y animaciones como la apertura del cofre.

El primero de los problemas es la importación de las animaciones. Unity es capaz de importar animaciones realizadas en Blender, pero importa al menos un valor para

cada parámetro del objeto, es decir, posición, rotación y escalado. En el caso del cofre solo quería que se abriera la tapa, por lo que en Blender solo se había animado la rotación de esta. Pero al importarlo a Unity se crean puntos para la posición y el escalado, y el problema con esto es que al reproducir la animación el objeto se mueve al origen de coordenadas, ya que la posición de la animación es esa. La solución a esto es bastante sencilla, simplemente hay que crear un objeto vacío y colocar el modelo 3D importante dentro de este, de esta manera la animación se reproduce con respecto a este objeto padre y se mantiene en la posición de origen de coordenadas locales a este.

Por otro lado tenemos el problema de que, al realizar modificaciones en el fichero del modelo de Blender, queremos que se actualicen los objetos del juego que usen este modelo, pero para ello hay que mantener el modelo dentro del objeto lo más puro posible, es decir, si tenemos que incluir algún script en el objeto, nunca se debe hacer sobre el sub-objeto el modelo 3D ya que se pierde la conexión con el original.

6. Metodología de trabajo

6.1. Herramientas

6.1.1. 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.

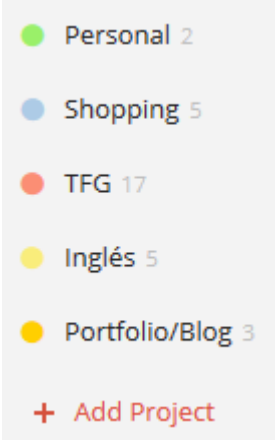


Figura 6.1 Todoist permite crear listas por proyecto

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.

Figura 6.2 Se pueden crear grupos para organizar tareas del mismo tipo

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

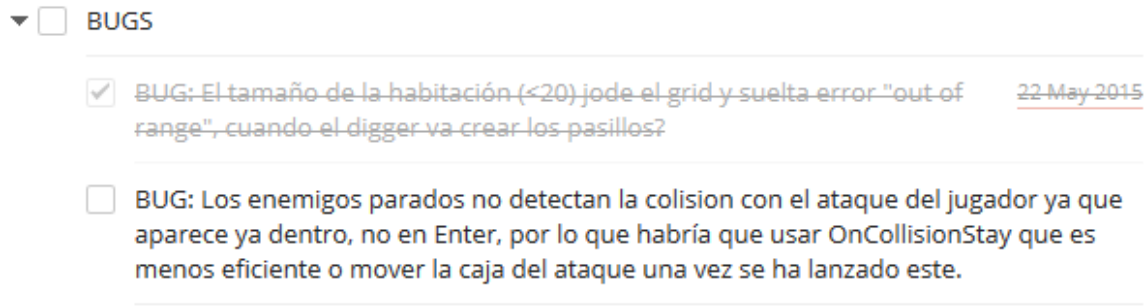


Figura 6.3 También se pueden establecer fechas límite para las tareas

6.1.2. Gestión de productividad: 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.

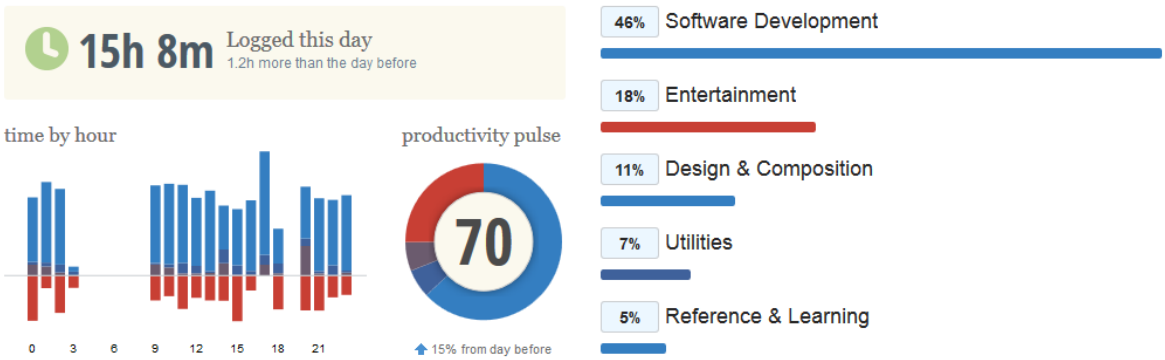


Figura 6.4 RescueTime proporciona información sobre productividad

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.

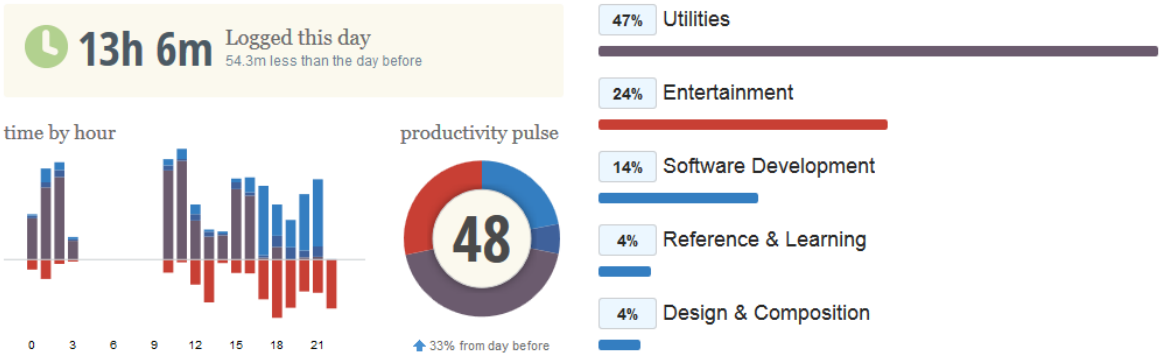


Figura 6.5 Los datos no siempre deben ser interpretados al pie de la letra

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.

7. Valoración final

A. Anexo: Game Design Document

A.1 Descripción y objetivos

Con gran inspiración en el juego de 2008, Spelunky, este proyecto se presenta como una implementación 3D de este, pero en un entorno más similar a Rogue (1980) o The Legend Of Zelda (en sus versiones clásicas 2D), donde encontramos habitaciones y pasillos más diferenciados, de construcción humana, en contraste con los entornos del estilo de cuevas naturales de Spelunky. Esto permite implementar trampas específicas para pasillos/habitaciones, que resultarían más difícil de colocar en una disposición irregular de tipo cueva.

El personaje principal se moverá por las distintas habitaciones, evitando trampas, recogiendo tesoros y eliminando enemigos. El objetivo es llegar al final de la mazmorra en el menor tiempo posible y con la mayor cantidad de tesoro.

Para las mazmorras del juego se intentarás cumplir los siguientes objetivos:

- Habitaciones conectadas entre sí por pasillos.
- Se generan procedimentalmente, siendo una mazmorra diferente en cada partida.
- Objetos para mejorar o recuperar la vida.
- Objetos especiales de mazmorra como puedan ser mapas, brújulas, llaves para abrir puertas o cofres...
- Enemigos.
- Trampas como paneles de pinchos, fosos, bolas rodantes...

Opcionalmente se consideran los siguientes objetivos:

- Varios tipos de armas, como arrojadizas.
- Jefes de mazmorra en la habitación final o en alguna habitación protegiendo un tesoro.

Importante notar que tanto la distribución de las habitaciones como de todo lo demás se generará de manera procedimental. Los elementos varían según la dificultad que se le quiera imprimir, que aumenta según se van superando mazmorras.

La vista del juego será 3D, pero con un aspecto pixelado estilo Minecraft, y se considera el uso de sprites para los personajes del juego.

A.2 Herramientas

A.2.1 Motor de juego

Debido a que el objetivo principal del juego es demostrar la generación procedimental de las mazmorras se va a utilizar Unity en su versión free como motor de juego para facilitar el desarrollo. Unity proporciona diversas herramientas integradas con un editor, estas incluyen un motor gráfico, motor de físicas, sistema de audio y sistema de GUI entre otros, suficiente para cubrir las necesidades de desarrollo.

A.3.2 Diseño de assets

Para los elementos 2D del juego como los botones de menú, títulos y resto de elementos del HUD, así como otros elementos 2D como texturas se utiliza The Gimp o Paint.net, ambos editores gratuitos. Para el diseño de los assets 3D se hace uso de Blender. Se utilizan recursos externos con licencia libre para todo lo que se considere adecuado.

7.1. Requisitos hardware y software

Estos son los requisitos que debe cumplir el sistema cliente para ejecutar el juego:

- Windows 7 o superior.
- Teclado y ratón o mando de juego con soporte para XInput.
- Sistemas basados en Linux.
- Tarjetas gráficas HD Intel 4000 o superior, NVIDIA o ATI.

7.2. Argumento

¡Aventuras, oro, gemas preciosas! Esa es la vida de Tass, el mejor saqueador habido y por haber de las tierras Niilunga.

Niilunga es un reino por el que han pasado numerosos reyes y reinas, culturas y civilizaciones de todos los tipos, y durante su período han dejado abandonados cantidad de tesoros, ocultos en las profundidades de los complejos templos y otros edificios.

Tu eres Tass, perdiste a tus padres cuando eras pequeño y desde entonces has tenido que buscarte la vida por tu cuenta cómo has podido, ahora dejas la sucia y… ciudad de Corinto y te alejas en las tierras de Niilunga buscando aventura. Pasas los días de ruina en ruina, recogiendo tesoros mientras esquivas trampas y te enfrentas a terribles criaturas. Pero algún día serás lo suficientemente rico como para retirarte para siempre y dedicar el resto de los días a vivir la vida de la mejor de las maneras.

¡Corre a la aventura, recorre las peligrosas mazmorras y recoge todos los tesoros que encuentres!

7.3. Jugabilidad y mecánicas

7.3.1. Objetivos

El juego consiste en atravesar un nivel de tipo mazmorra, compuesto por habitaciones y pasillos, desde una entrada hasta una puerta de salida. Por el camino se pueden recoger tesoros, que contribuyen a la puntuación del jugador, junto al tiempo en que se completa el nivel. Durante el transcurso del nivel se plantean trampas que sortear y enemigos contra los que luchar o evitar. El juego es de niveles infinitos y solo termina cuando el jugador pierde todas sus vidas o reinicia un nuevo juego.

7.3.2. Flujo de juego

Al tratarse de un juego de partidas rápidas, el flujo a través de este debe ser también rápido y muy modular. Queremos que el usuario pueda iniciar o continuar una partida y terminar un nivel en menos de 10-15 minutos o abandonarlo en cualquier momento, volviendo al mismo estado en el que estaba antes de comenzar a jugar ese nivel.

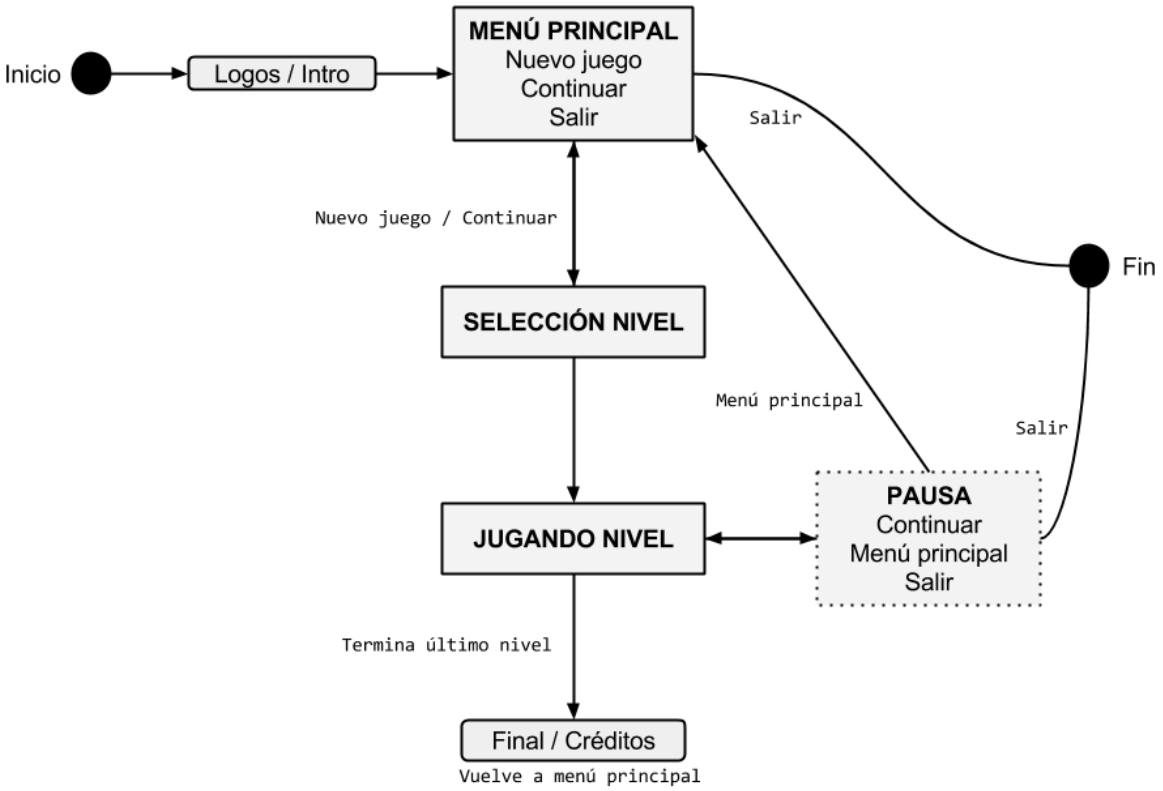


Figura 0.1 Flujo de pantallas

Dentro del nivel/mazmorra el objetivo es llegar a la salida consiguiendo por el camino la mayor cantidad de oro y joyas. No hay puntos de guardado o de control, cuando el jugador muere vuelve al inicio de la mazmorra.

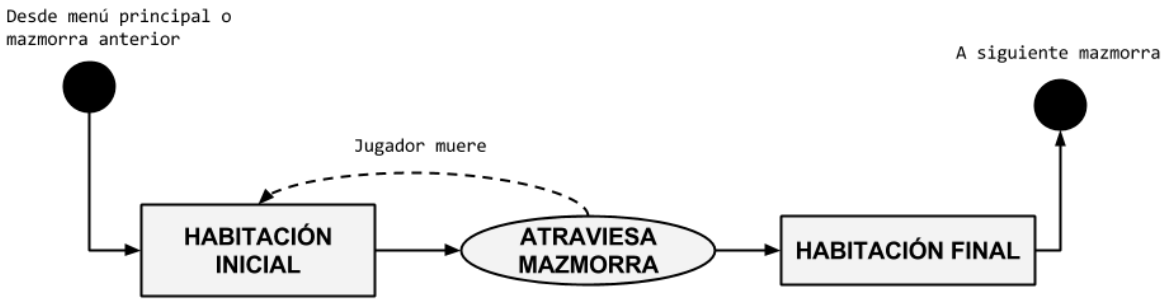


Figura 0.2 Flujo de mazmorra

7.3.3. Controles de jugador

El jugador maneja a un personaje en un entorno tridimensional y podrá realizar distintas acciones:

- **Movimiento**: El jugador puede mover en el plano x,z que representa el suelo en todas las direcciones disponibles dentro de este. Se utilizarán las flechas de teclado para este movimiento.
- **Correr**: Añade una velocidad extra al movimiento normal. Mientras el jugador se mueve se puede correr manteniendo pulsado la tecla Shift.
- **Saltar**: Con la tecla espacio el jugador puede realizar un salto para sortear trampas, su longitud dependerá de la velocidad de movimiento.
- **Atacar**: El jugador puede golpear enemigos o cofres para abrirlos. Se usa con la tecla Z o el botón izquierdo de ratón.

Para el resto de detalles sobre las mismas mazmorras, objetos y enemigos que se podrán encontrar vamos a extendernos un poco más en las siguientes secciones.

7.4. Mazmorras

Al tratarse de un juego de tipo dungeon crawler de tipo aventuras las referencias más representativas de lo que deseamos las podemos encontrar en la saga de videojuegos The Legend Of Zelda (los juegos clásicos en 2D) o la saga Diablo, juegos completamente diferentes en temática, pero que comparten esa jugabilidad sobre mazmorras de diversos estilos.

Pero no buscamos algo tan homogéneo como podríamos encontrar en el primer juego de la saga Zelda, donde las habitaciones son prácticamente del mismo tamaño y simplemente se sitúan una al lado de la otra. En este caso buscamos algo más desordenado, con pasillos que unan las habitaciones, con formas irregulares, pero sin llegar al aspecto de cueva natural. Algo similar a algunas de las mazmorras que podemos encontrar en Diablo.

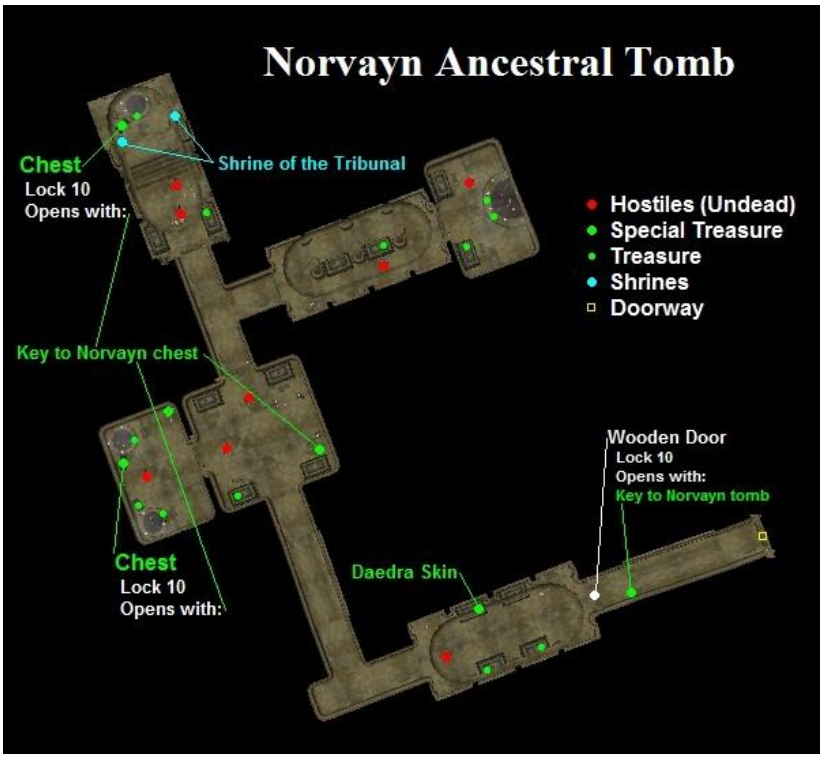


Figura 0.3 Mapa de Diablo donde podemos distinguir claramente un inicio y fin de mazmorra, habitaciones y pasillos que las unen.

Este tipo de distribución nos permite colocar ciertas trampas en pasillos, donde hacen mejor efecto que en habitaciones amplias, y dejar las habitaciones para otro tipo de trampas, enemigos y objetos.

Dejando esto claro, podemos definir el contenido que luego agregaremos a la mazmorra, podremos encontrar:

- **Entrada**. Habitación donde comienza el jugador.
- **Salida**. Habitación donde termina la mazmorra, se debe colocar alejada de la entrada. No tiene por qué ser la última habitación a visitar, sino que el jugador puede escoger terminar la mazmorra o seguir explorando un poco más.
- **Habitación normal**. Casi siempre contendrá algún enemigo y posiblemente trampas, pero en menor medida.

- **Habitación especial**. Son habitaciones que contienen algún objeto especial, como una llave que abre la puerta de otra habitación, o un cofre con tesoro extra o alguna poción.
- **Pasillo**. Une dos habitaciones u otro pasillo. No contiene ningún enemigo pero sí trampas, algunas de estas solo aparecen en pasillos.

7.5. Generación de contenido

El aspecto más importante del juego será la generación procedimental de contenido, y más en concreto de la distribución y aspecto de las habitaciones de las mazmorras, así como de los objetos, trampas y enemigos que vamos a situar en estas.

7.5.1. Habitaciones - Estructura de la mazmorra

Primero debemos generar la estructura de la mazmorra, es decir, decidir las dimensiones de esta y crear las habitaciones y pasillos respetando ciertos parámetros. Estos parámetros dependen del algoritmo que finalmente se seleccione para la generación, se baraja el uso de árboles binarios de partición, el método del autómata celular o simplemente la generación aleatoria de habitaciones sin solapamiento.

Independientemente del algoritmo utilizado, y teniendo como base la rejilla del editor de Unity, donde un objeto cubo ocupa una unidad de esta rejilla, establecemos unas medidas estimadas de la mazmorra y habitaciones según ascendemos en niveles.

Las dimensiones no son absolutas, sino que se escoge al azar entre el valor del nivel actual y el valor del siguiente. Por ejemplo, en el nivel 1 sería ancho entre 50 y 80 y alto entre 50 y 80.

1 unidad = 1 unidad/celda de la rejilla de Unity

Las dimensiones no son exactas para cada nivel, sino que se escoge al azar entre el valor del nivel actual y el valor del siguiente. Por ejemplo, en el nivel 1 sería ancho entre 50 y 80 y alto entre 50 y 80.

Nivel	Min. Dimensiones mazmorra	Dimensiones Habitación
1	50x50	15
2	80x80	15
3	100x100	12
4	105x105	15
5	110x110	14
6	115x115	14
7	120x120	12

Estos valores se han decidido después de realizar varias pruebas en Unity con mazmorras creadas manualmente. A partir del nivel 8 los valores serán prácticamente los mismos y la dificultad variará con el contenido de la mazmorra.

Se selecciona una habitación de entrada y una de salida antes de continuar generando el contenido dentro de las habitaciones.

7.5.2. Colocación de objetos

Lo primero después de la creación de las habitaciones es colocar los objetos sobre los que después nos basaremos para las trampas y enemigos. Tenemos distintos

tipos de objetos que se explican en su correspondiente sección, pero ya podemos establecer ciertas reglas:

- Nunca aparecen en pasillos.
- Hay objetos en todas las habitaciones, incluidas la entrada y la salida.
- Hay objetos comunes, como monedas de oro, y otros menos comunes, como cofres o llaves.
- Algunos objetos solo aparecen si son requeridos en la mazmorra, como las llaves.
- La generación de cierto tipos de objetos afectará a la generación de trampas y enemigos en esa habitación y posiblemente las habitaciones y pasillos adyacentes. Queremos que el acceso a ciertos objetos especiales sea más difícil.

7.5.3. Trampas

Las trampas son objetos animados distribuidos por toda la mazmorra con cierta aleatoriedad basándonos en los siguientes principios:

- Se pueden generar tanto en habitaciones como en pasillos.
- Dependiendo del tipo de trampa puede que algunas solo sean exclusivas para habitación o para pasillo.
- Si en una habitación podemos encontrar un cofre o un objeto especial, como puede ser una llave, es muy probable que alrededor de este se genera algún tipo de trampa, como los suelos de pinchos.
- No se deberían incluir más de 3 tipos de trampas en una misma habitación o pasillo.
- La cantidad varía dependiendo de la cercanía con la salida y habitaciones con objetos especiales.
- Las trampas no interactúan o dañan a enemigos.

Tales trampas aparecen en pasillos en tal medida, junto a cofres y en habitaciones especiales. Interactúan con otras trampas o con enemigos...

7.5.4. Enemigos

Finalmente se generan enemigos de varios tipos, como se describe más adelante, la colocación de estos es como sigue:

- Siempre aparecen en habitaciones, nunca en pasillos.
- Entre 1 y 5 como máximo por habitación.
- Toman precedencia sobre las trampas, pero pueden aparecer junto a alguna.
- Aparecen en más cantidad cuanto más cerca están de una habitación con un objeto especial o hacia el final de la mazmorra.
- No aparecen en la habitación de entrada pero pueden aparecer en la de salida.

7.6. Objetos

En general los objetos que podemos encontrar que no sean trampas o decoración de escenario se clasificará en pociones de vida o dinero, pero pueden estar contenidos en cofres u otros objetos. A continuación listo una posible serie de objetos que se incluyen en el juego:

Pociones. Se encuentran esparcidas por la mazmorra, aunque no son muy comunes, existen pociones de varios efectos. El efecto de las pociones se activa por contacto pero se estudiará el uso de un inventario para almacenarlas y que sea el mismo jugador el que las active manualmente.

- **Vida**. Recuperan la vida del jugador. Pueden recuperar la vida por completo o solo un corazón.
- **Invencibilidad**. Hacen que el jugador sea invencible durante cierto período de tiempo. Son poco comunes de encontrar y aparecen en zonas o mazmorras de nivel difícil, donde pueda haber pasillos con varias trampas.

- **Tesoro**. Esto representa los puntos en el juego, y el objetivo es obtener el máximo posible antes de terminar la mazmorra, ya que esto, junto con el tiempo, indicará la puntuación del usuario. Podemos encontrar distintos tipos de tesoro.
    - **Oro** (10 pts). Se representa mediante una moneda de oro, es el ítem más común del juego pero el que menor valor monetario tiene. Puede que algunos dejen una moneda de oro al morir.
    - **Gemas** (50 pts). Es más difícil de encontrar y en menor cantidades. Aparecen en ciertas habitaciones apartadas, normalmente junto a otro ítem de tipo poción.
    - **Diamantes** (100 pts). Los más valiosos del juego y difíciles de encontrar. Pueden aparecer en algún cofre y habitaciones apartadas, pero esto es menos común.
- **Cofre**. Los cofres pueden incluir ítems de tipo pociones o tesoro. Cuando incluyen tesoro es más probable de encontrar gemas que oro o diamantes. Puede que mezcle varios de estos.
- **Llave**. Algunas habitaciones están cerradas con llave. Para ello se debe encontrar una de estas llaves. Solo tienen un uso y habrá tantas llaves como puertas en la mazmorra. Suelen estar protegidas por enemigos o trampas.

7.7. Trampas

Uno de los aspectos más importantes del juego es el uso de trampas, posicionadas por toda la mazmorra en habitaciones o pasillos en distintas combinaciones como parte de las técnicas de generación procedimental de contenido.

Estas trampas deberán ser sorteadas por el jugador atendiendo a las características de estas, como su tipo, posición o combinación.

7.7.1. Características

Cada trampa presenta una construcción y comportamiento diferente basándose en diversos aspectos. Además la combinación de distintos tipos de trampas en una habitación o pasillo hace que el jugador deba replantear la estrategia en cada situación a pesar de ya conocer el funcionamiento de estas trampas individualmente.

Pero debemos establecer una serie de premisas y parámetros para colocar las trampas en lugares adecuados y combinarlas, siempre permitiendo que puedan ser superadas sin recibir daño. Algunos de estos requisitos podrían ser:

- **Dificultad de mazmorra**. La dificultad o nivel de la mazmorra
- **Tipo de habitación**. Las habitaciones especiales con llaves u otros ítems especiales contendrán menos carga de trampas, pero los pasillos y habitaciones adyacentes o directamente conectados a estas serán, en contraste, las que más tengan.
- **Dimensiones de la habitación**. Los pasillos y las habitaciones tendrán trampas y combinaciones distintas como veremos en la descripción de las trampas.
- **Cercanía con otras trampas**. Las trampas del mismo tipo normalmente se pueden encadenar o combinar, como colocar varios suelos de pinchos seguidos en un pasillo. En estas combinaciones se deberán ajustar los tiempos de activación de cada una para que puedan ser superables sin que el jugador reciba daño.

7.7.2. Listado de trampas

Para entender mejor lo expuesto aquí se listan las trampas posibles y sus características y modos de funcionamiento.

- **Pared de pinchos**. Son placas que se posicionan en las paredes y de las que surgen un...
- **Suelo de pinchos**. Similares a las de pared, estas se sitúan en el suelo y contienen unos pinchos más pequeños que igual de dañinos.

- A diferencia de las anteriores estas pueden ser automáticas, activándose cada cierto tiempo, o de presión, activándose cuando el jugador pasa por encima. Tanto para uno como para otro modo tiene que haber un tiempo de "recarga".
- **Dispensador de dardos/flechas**. Se sitúan en la pared y se activan cuando el jugador pasa por delante o automáticamente cada cierto tiempo. En el primer caso también tienen un tiempo de recarga, por lo que si el jugador está delante de estas no se produce un disparo constante, sino que se hará cada 2 segundos. Estas trampas deben pasarse corriendo o lo más separado posible de estas, ya que los dardos es disparan a gran velocidad. En los pasillos se pueden combinar junto con paredes de pinchos para mantener al jugador centrado en el corredor de manera que no puede evitarlas simplemente pegándose a la pared contraria.
- **Bola rodante**. Las podemos encontrar exclusivamente en pasillos largos. Se trata de una roca gigante que rueda hacia el jugador normalmente matándolo directamente al contacto. Por ello no son difíciles de evitar y se puede escapar corriendo en dirección contraria y dejando que golpeen alguna pared. Pero pueden combinarse en pasillos que ya tienen otras trampas, obligando al jugador a volver a pasar sobre estas. Se activan al pasar por cierta zona del pasillo donde el jugador activa un trigger invisible que hace que la roca se mueva hacia este.

7.8. Enemigos

7.8.1. Inteligencia artificial

Los enemigos implementan una inteligencia artificial simple, basadas en estados y círculos de visión o cajas de colisión, según sea el caso, los comportamientos pueden ser los siguientes:

- **Patrullar**: El personaje patrulla una zona, moviéndose en una línea o en un rectángulo.

- **Perseguir**: El personaje persigue al jugador cuando este entra en su círculo de visión. Cuando el jugador se aleja demasiado entonces el enemigo vuelve a su zona de patrulla.
- **Atacar**: Cuando el jugador está suficientemente cerca del personaje, este ataca en la dirección del jugador. Tendrá un tiempo de espera entre ataque y ataque.

En principio estos tres estados se complementarán. Por ejemplo, un enemigo de tipo cangrejo patrullará de un lado a otro y cuando jugador entre en su rango de ataque entonces pasará al estado de atacar. Los **goblins**, además de patrullar, podrán perseguir al jugador cuando este entre en su rango de visión y atacará cuando entre en su rango de ataque. Si el jugador se aleja demasiado entonces este vuelve a su zona de patrulla.

7.9. Aspecto Visual

7.9.1. General

A pesar de que a lo largo de este documento he nombrado a Zelda y Spelunky como referencia para la jugabilidad y temática de este, el aspecto que vamos a escoger se desarrollará sobre un entorno 3D pero con ciertas peculiaridades. El entorno, como ya he dicho, será 3D, pero los personajes como el jugador y los enemigos serán sprites 2D. A esto se lo conoce como billboards y se suele utilizar para representar vegetación en videojuegos con gran cantidad de estos elementos, ya que en vez de un modelo 3D estamos trabajando sobre un quad mucho más sencillo.

El uso de billboards sobre personajes surge en juegos como Doom, donde los enemigos y los propios jugadores eran sprites que siempre miraban a la cámara y en algunas ocasiones, para dar la sensación de que se giraban simplemente se cambiaba el sprite, pero siempre eran objetos planos. No solo en Doom, sino que este estilo se ha utilizado en juegos de rol japonés como la saga Ys.

Figura 0.4 Ys Chronicles presenta sprites 2D en un entorno 3D

La elección de este estilo es primordialmente debido a que es mucho más fácil crear o incluso encontrar hojas de sprites que pueda usar en el juego, de buena calidad y que tengan un estilo similar entre sí para la creación de los personajes en comparación con tener que realizar modelos 3D y animarlos correctamente, lo cual no es el objetivo de este trabajo. Pero por otro lado también está mi interés en utilizar este estilo para darle un toque más de dibujo animado, que sea un juego 3D pero manteniendo el estilo alegre de un juego 2D como pueda ser Spelunky.

Para el resto de elementos 3D vamos a escoger un estilo sencillo, de **baja carga poligonal** para que encaje con el estilo general simple pero alegre. Por esto las **texturas** también tendrán un estilo **Minecraft**, basadas en arte de píxel.

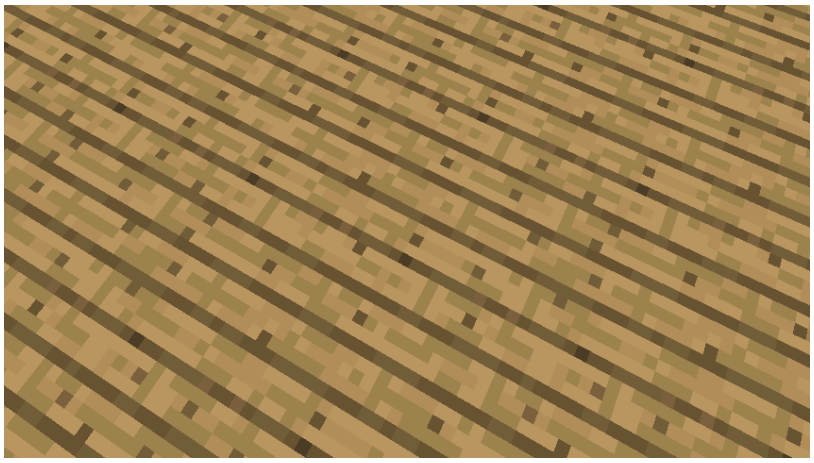


Figura 0.5 Textura de madera en minecraft, simple pero efectiva

En cuanto al estilo de colores y texturas en sí vamos a hacer referencia, de nuevo, a Minecraft, que tiene un aspecto pixelado, simple pero efectivo. También debemos tener en cuenta que estamos realizando un juego estilo Spelunky, por lo que la temática deberá estar reflejada en estas texturas, con el uso de roca, madera, cuero, moho y otros elementos que representen aspecto natural.

7.10. Perspectiva y cámaras

Escogemos una vista 3D con una cámara con proyección en perspectiva, dando efecto de profundidad ya que es importante poder medir correctamente las distancias para sortear trampas y golpear a enemigos.

La cámara principal seguirá al jugador con un efecto de muelle, con lo que no se desplazará inmediatamente a la posición centrada en el jugador sino que se realizará un movimiento retrasado suavizado mediante un interpolado lineal.

La cámara se situará frente al jugador, o detrás de este, a cierta distancia de separación. Esta distancia se podrá modificar dentro de ciertos límites. También

se podrá subir o bajar la vista dentro de ciertos límites pero en ningún caso se podrá girar en torno al eje vertical de manera manual.

Otras cámaras que se puedan incluir serán automáticas para resaltar ciertas situaciones, pero en ningún caso el jugador tiene control sobre estas. Estas situaciones se refieren a pequeñas escenas cinemáticas cuando el nivel comienza y se hace un sobrevuelo sobre el mapa o cuando el nivel termina y se hace un zoom al jugador. En la sección de animaciones de cámara se describen estas situaciones.

Se incluirá una cámara libre a propósitos de testeo, con un control similar al que implementa el mismo editor de Unity.

7.11. Interfaz

Cuando hablamos de interfaz vamos a describir, por un lado, el sistema de menús para acceder al juego, durante la pausa y en escenas intermedias, como las estadísticas de final de fase, y por otro lado, el sistema de HUD, es decir, la información que se muestra durante la partida indicando principalmente el estado actual del jugador.

7.11.1. Resoluciones

El juego se puede adaptar a distintas resoluciones y aspectos, pero se desarrollará teniendo en cuenta un aspecto de 16:9 y una resolución mínima de 960x540, por lo que no se asegura que los elementos de la interfaz se adapten correctamente en resoluciones más bajas.

7.11.2. Menús

Como ya hemos visto en la sección donde se explica el flujo de juego, vamos a tener un menú principal para escoger una nueva partida, opciones y tests, además de un menú de pausa. También se mostrará una pantalla al final de cada fase indicando

las estadísticas de esta. Estas pantallas compartirán un estilo homogéneo en los botones, tipografías y comportamiento. Los menús podrán ser navegados y activables tanto por ratón como por teclado.

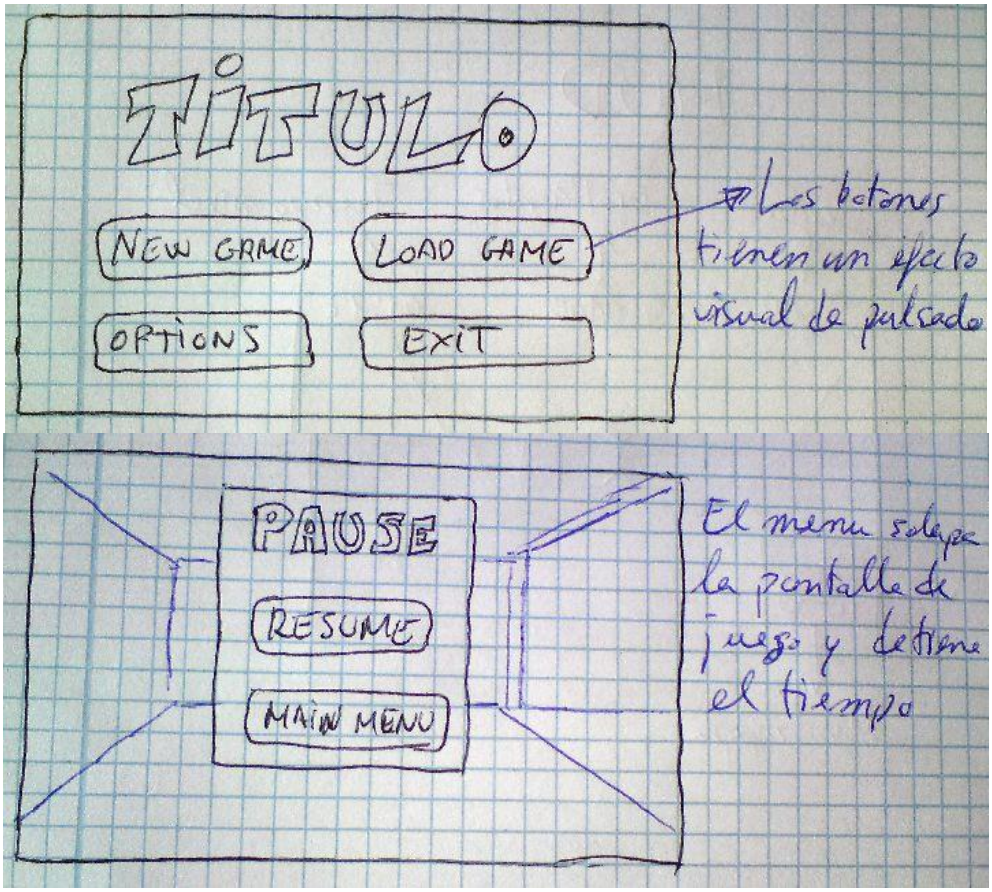


Figura 0.6 Pantalla de título y de pausa

7.11.3. Heads-up display

Durante la partida se mostrará, bordeando la pantalla, cierta información sobre el estado del jugador. Serán elementos no interactivos, pero se actualizarán automáticamente durante la partida. Este HUD contendrá los siguientes elementos:

- **Vida del jugador**. Representada mediante corazones. Indica la salud del jugador, la cantidad de golpes que puede recibir. Se actualizará cada vez que este reciba daño o se cure mediante algún objeto.

- **Oro/Puntos**. Cantidad de oro, gemas y otras joyas recogidas durante el nivel. Se mostrará el valor total de la suma de estas.
- **Tiempo de juego**. Tiempo que se lleva jugando ese nivel. En principio simplemente se utilizará para propósitos de speedruns. Pero es considerará aplicar recompensas (más puntos/fortunas) para los mejores tiempos.
- **Minimapa**. Muestra, a vista de pájaro, la porción de la mazmorra descubierta.

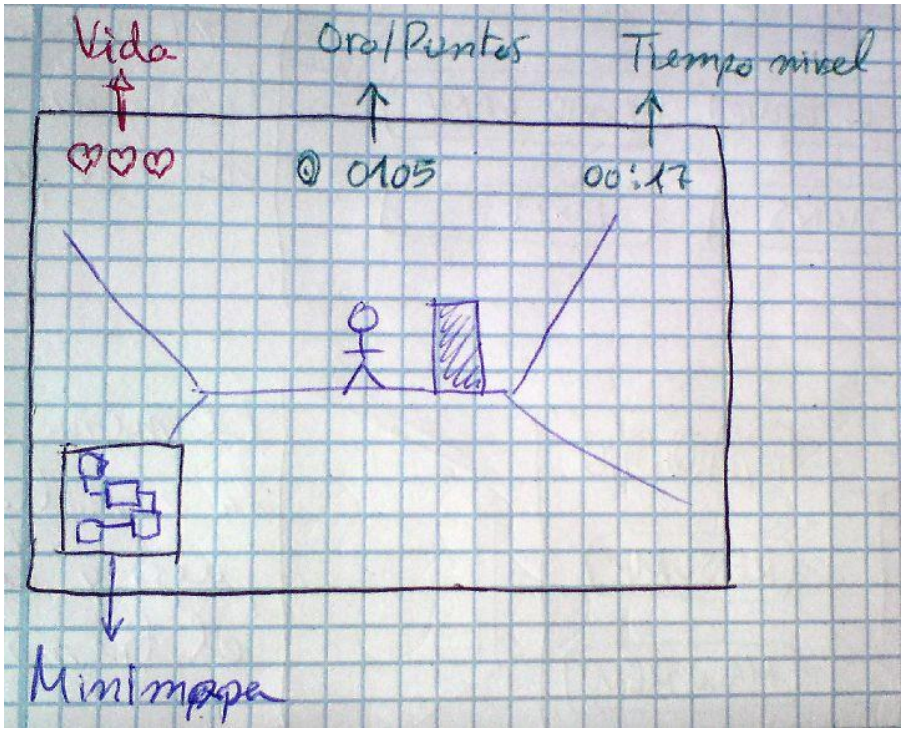


Figura 0.7 HUD

7.12. Animaciones

7.12.1. Objetos 3D

Los objetos interactivos 3D como puedan ser cofres o trampas serán animados, cuando sea posible, desde el mismo editor Unity 3D, ya que permite más flexibilidad y rapidez a la hora de ajustar tiempos. Para las trampas esto puede ser sencillo ya que, por ejemplo, las trampas de pinchos simplemente se basan en

animar la posición en un eje. Por otro lado trampas como el disparador de dardos o la roca rodante tendrán movimientos basados en físicas, es decir, no serán animaciones en sí sino que se les aplicará una fuerza física para definir su comportamiento.

Otros objetos como cofres en principio serán animados en Blender ya que se trata de objetos compuestos donde se anima la rotación utilizando un punto de anclado concreto. Esto hace que sea más conveniente realizar la animación previamente en Blender e importarla a Unity.

7.12.2. Personajes 2D

Para los personajes ya hemos definido que tendrán un aspecto 2D y se utilizarán sprites para representarlos en pantalla. Por ello se utilizar sprite sheets conteniendo todas estas animaciones, pero se tendrán que editar desde Unity, que proporciona un módulo de animación con el que podemos crear las distintas acciones por separado y luego utilizar un gestor basado en nodos para activar cada una de estas.

7.13. Sonido y Música

Durante el juego podemos escuchar los sonidos básicos del personaje que maneja el jugador como:

- **Salto**. Hace un pequeño gruñido o se escucha un sonido al estilo Super Mario Bros que representa el sonido del personaje separándose del suelo. Si al caer se utiliza un sonido deberá ser algo muy sutil, un sonido seco.
- **Golpea**. Se escucha el sonido del arma moviéndose. Además puede que se agregue un sonido de esfuerzo del mismo personaje. Cuando se produce contacto del arma contra un enemigo se escucha un sonido seco de golpe. En principio no se usará sonido al golpear otros objetos, pero se puede hacer una excepción si se necesita acentuar el sonido al abrir un cofre de un golpe.

- **Moverse y correr**. En principio se probará con sonido solo al correr, se puede probar con un sonido constante de pasos, muy sutil, o simplemente al arrancar el sprint.
- **Daño**. El personaje lanza un quejido al ser golpeado. Si el golpe produce la muerte entonces el sonido es más largo y dramático.

Los enemigos también pueden realizar sonidos al golpear, ya sea emitiendo un gruñido como con algún sonido del arma que utilicen. Algunos enemigos como los cangrejos o las arañas puede realizar un pequeño sonido de pasos al moverse, al descolgarse o realizar otras animaciones.

Otros sonidos provendrán de objetos, ya sean trampas como ítems para el usuario. Los sonidos deberán ser simples, en lo posible, y representar claramente el objeto y acción que se está produciendo.

Los botones de menú pueden incluir sonido al ser pulsados o cuando el ratón pasa por encima. Se tratará de sonidos de roca o madera moviéndose, como si se tratara de un mecanismo manual.

Durante la pausa durante el juego se reducirá el sonido del nivel, como la música, y se podrá escuchar claramente el sonido de los elementos del menú.

La activación/desactivación de menús puede incluir algún sonido simple indicando que se ha producido una acción.

Acceder a un nivel o comenzar un juego nuevo también podrá incluir un sonido de transición o pequeño clip de música.

Música

Se trata de un juego de espíritu arcade, por lo que la música deberá tener cierta marcha, incitando al jugador a estar atento, pero sin resultar frenética, un sonido funky. Tanto las canciones de los menús como las de mazmorras tendrán un estilo similar, intentando lograr una homogeneidad entre el cambio de escenas.

Los instrumentos son de estilo electrónico, recordando en cierta manera a la música chip, pero con un estilo más moderno.

7.14. Planificación

Finalmente expongo la planificación para el desarrollo del mismo videojuego. Mi intención aquí no es entrar en detalle con respecto a las fechas concretas y tampoco incluir la planificación de otros aspectos del estudio del trabajo de fin de grado, sino simplemente del orden de implementación de las características del videojuego expuestas en este documento de diseño.

Puntos del videojuego y orden de implementación.

Etapa 1

- Implementación del algoritmo de creación de habitaciones y pasillos. Pruebas con distintos parámetros y ajustes según el nivel de la mazmorra.
- Implementación del jugador, movimientos sobre un escenario creado manualmente para realizar pruebas los controles.
- Creación de assets, como el sprite del jugador y objetos.

Etapa 2

- Creación de sprites de enemigos.
- Creación de los modelos para las trampas y animación de estas.
- Implementación de la IA de los enemigos. Pruebas sobre un escenario creado manualmente.
- Implementación de la lógica de los objetos. El jugador interactúa con estos.
- Creación del flujo de pantallas, intersección entre estas y menús.

Etapa 3

- Implementación de la lógica de las trampas e interacción con el usuario.
- Estudio y desarrollo de la generación de objetos, trampas y enemigos sobre el escenario generado procedimentalmente.
- Integración final, pruebas, corrección de bugs y ajustes finales.

Bibliografía y referencias

ANDREW DOULL 2007. The Death Of The Level Designer - Procedural Content Generation Wiki. [en línea]. [Consulta: 21 junio 2015]. Disponible en: http://pcg.wikidot.com/the-death-of-the-level-designer.

Conway's Game of Life. En: Page Version ID: 666495124, *Wikipedia, the free encyclopedia* [en línea] 1970. [Consulta: 21 junio 2015]. Disponible en: https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=666495124.

Generative music. En: Page Version ID: 665392869, *Wikipedia, the free encyclopedia* [en línea] 2015. [Consulta: 21 junio 2015]. Disponible en: https://en.wikipedia.org/w/index.php?title=Generative_music&oldid=665392869.

Género de videojuegos. En: Page Version ID: 83140259, *Wikipedia, la enciclopedia libre* [en línea] 2015. [Consulta: 21 junio 2015]. Disponible en: https://es.wikipedia.org/w/index.php?title=G%C3%A9nero_de_videojuegos&oldid=83140259.

JAMIS BUCK 2011. Maze Generation: Aldous-Broder algorithm. *The Buckblog* [en línea]. Disponible en: http://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm.

JIM BABCOCK [sin fecha]. Cellular Automata Method for Generating Random Cave-Like Levels. [en línea]. Disponible en: http://www.jimrandomh.org/misc/caves.html.

NOTCH 2011. Terrain generation, Part 1. *Notch tumblr* [en línea]. Disponible en: http://notch.tumblr.com/post/3746989361/terrain-generation-part-1.

Radiant AI. En: Page Version ID: 599345000, *Wikipedia, the free encyclopedia* [en línea] 2014. [Consulta: 21 junio 2015]. Disponible en: https://en.wikipedia.org/w/index.php?title=Radiant_AI&oldid=599345000.

The History of Elite: Space, the Endless Frontier. [en línea] 2009. [Consulta: 21 junio 2015]. Disponible en: http://www.gamasutra.com/view/feature/3983/the_history_of_elite_space_the_.php.

TOGELIUS, J., YANNAKAKIS, G.N., STANLEY, K.O. y BROWNE, C. 2011. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 172–186.

TULLEKEN, H. 2009. How to Use Perlin Noise in Your Games. *Dev.Mag* [en línea]. [Consulta: 21 junio 2015]. Disponible en: http://devmag.org.za/2009/04/25/perlin-noise/.