Diseño y generación de mapas inteligentes en videojuegos

David Campos Escalera

26/06/2019

Resumen– Cuando en un videojuego se quiere ampliar el abanico de situaciones jugables sin necesidad de diseñar cada elemento de forma individual, se recurre a los algoritmos de generación automática.

En este trabajo se ha desarrollado un videojuego y la generación procedural de sus entornos jugables. El juego es un videojuego de acción-sigilo en 2D donde los elementos principales del juego (número de enemigos, diseño de las pantallas) se adaptan a medida que avanzan los niveles según la dinámica de juego del jugador en niveles anteriores. Para ello se han utilizado algoritmos de generación procedural como el Colapso de la Función de Onda [1].

A partir de los algoritmos implementados se generan entornos jugables variados que crean experiencias diferentes en cada partida y se adaptan a tu forma de jugar.

Palabras clave– Colapso de la Función de Onda, Aprendizaje Computacional, Inteligencia Artificial, Generación procedural, Máquina de estados finitos , Diseño de videojuegos

Abstract– When in a video game you want to expand the range of playable situations without having to design each element individually, you resort to automatic generation algorithms.

In this paper I have developed a video game and the procedural generation of its playable environments. The game is a 2D action-stealth video game where the main elements of the game (number of enemies, level design) are adapted as the levels advance according to the player's game dynamics in previous levels. For this purpose, procedural generation algorithms have been used, such as the Wave Function Collapse [1].

Based on the implemented algorithms, different playable environments are generated which result in different experiences in each game, adapted to your way of playing.

Keywords– Wave Function Collapse, Machine Learning, Artificial Intelligence, Procedural Generation, Fine-State Machine, Game Design

1 Introducción

CREAR videojuegos es caro, desde que se imaginan hasta que se implementan, pasando por el diseño y otras tantas fases de desarrollo [2]. Es por ello que las empresas tratan de abaratar costes allá donde pueden. Una de las soluciones que se ha barajado desde prácticamente los inicios de esta industria es la generación de entornos jugables (mapas y sus elementos, como enemigos u objetos) de forma automatizada, reduciendo el número de personas necesarias para diseñar cada elemento que compone el entorno jugable y la consecuente experiencia final del jugador. Además de abaratar costes, la generación automatizada de entornos permite a equipos pequeños crear productos con combinaciones de elementos de juego prácticamente inabarcables, que hacen que cada partida sea diferente y que juegos de dimensiones modestas expandan

• E-mail de contacto: david.campose@e-campus.uab.cat
• Mención realizada: Computación
• Trabajo tutorizado por: Debora Gil Resina (Ciencias de la computación)
• Curso 2018/19

sus horas útiles de forma indefinida.

Uno de los mayores representantes de este tipo de juego es The Binding of Isaac [Figura 1]. En este videojuego debemos avanzar por diferentes salas que se seleccionarán aleatoriamente entre una lista de salas prediseñadas. También se generarán enemigos en posiciones predefinidas en función de la sala que haya aparecido. Se estima que ha vendido entre dos y cinco millones de unidades, además de haber conseguido picos de hasta 70.000 jugadores simultáneos y mantener una media de 5.000 jugadores diarios ocho años después de su lanzamiento original.

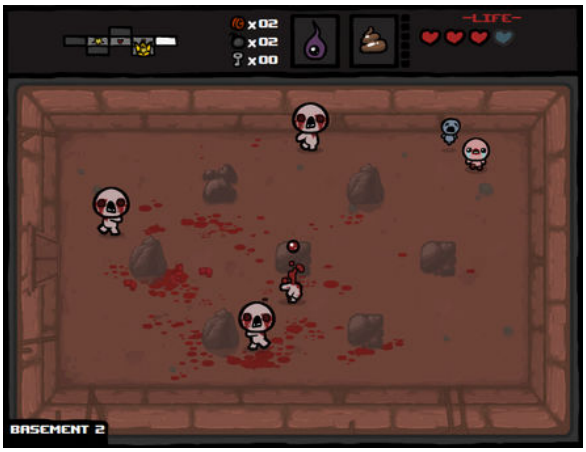

Fig. 1: Pantalla de juego en The Binding of Isaac

Dentro de los modelos de generación de entornos los mas usuales son la generación aleatoria, por procedimientos o procedural y la combinación de ambas [Figura 2].

Por un lado, la generación aleatoria es aquella donde el resultado siempre dependerá de una probabilidad sin seguir ninguna regla. Puede coincidir que se dé el mismo resultado, pero tan sólo se trataría de una coincidencia. Esto permite crear entornos muy diferentes entre sí, pero tiene la pega de poder generar entornos caóticos donde no se pueda jugar o que rompan por completo la experiencia del jugador [3]. Es ahí donde entra la generación procedural.

En la generación procedural se establecen una serie de reglas más complejas según las cuales ciertos tipos de interacciones no podrán darse. Esto significa que para una misma operación, con los mismos datos de entrada, la salida siempre será igual. De esta forma se nos permite tener una serie de interacciones mucho más controladas que utilizando la generación aleatoria.

Sin embargo, si únicamente se utilizara la generación procedural los entornos jugables serían siempre los mismos. Es por eso que se utiliza en combinación con la generación aleatoria, de forma que estaremos generando objetos de forma aleatoria, siempre acotados por los límites y reglas que impone la proceduralidad. Estas técnicas se utilizan especialmente en la generación de mapas. Un ejemplo serían aquellos juegos donde el mapa está compuesto por pequeñas secciones de mapa prediseñadas. Mediante la generación procedural se establecerían las secciones que pueden ubicarse en cada posición del mapa de forma que las conexiones entre zonas sean coherentes, y con la generación aleatoria se generaría una de las secciones de entre las que pasaron por los filtros de la proceduralidad.

En este trabajo se ha desarrollado un videojuego del género acción-sigilo con generación procedural y aleatoria de sus entornos jugables teniendo en cuenta las dinámicas del jugador en niveles anteriores para conseguir entornos personalizados.

Fig. 2: Criatura y entorno generados mediante aleatoriedad y proceduralidad en *No Man's Sky*

2 Estado del arte

El estado del arte dentro de la generación de entornos en videojuegos se centra principalmente en el uso de las generaciones aleatoria y procedural que ya se han mencionado durante la introducción. Estos son los sistemas que he utilizado en el desarrollo de mi generador de entornos. Utilizo el Colapso de la Función de Onda, tecnología popularizada en 2018 que consiste en una serie de algoritmos de generación procedural basados en las mecánicas de la física cuántica para observar y por tanto definir el estado (el elemento que se posicionará) en las diferentes casillas que componen el mapeado del juego. Además de ello se utilizan otra serie de algoritmos de generación procedural que serán presentados más adelante.

Otra manera de gestionar los entornos de juego para hacerlos dinámicos es la jugabilidad emergente. Esta consiste en dejar interactuar diferentes mecánicas jugables en entornos previamente diseñados, de forma que el jugador podrá resolver una misma situación de distintas maneras, creando así experiencias diferentes para cada persona o partida.

El problema de la jugabilidad emergente es que requiere que los juegos tengan suficiente contenido como para sentir variedad en sus interacciones y evitar repetirse, de forma que es algo que sólo pueden alcanzar los proyectos más grandes como *Fortnite* o *Red Dead Redemption 2*, de forma que no ha conseguido establecerse como estándar en la industria.

Si bien juegos actuales como los citados son los que han llevado la jugabilidad emergente a los oídos del público mayoritario, fueron los videojuegos multijugador masivos en

línea los que abrieron paso a esta forma de diseño, con EVE Online como su mayor exponente [Figura 3].

Fig. 3: Batalla de Asakai, surgida de forma emergente en EVE Online

3 OBJETIVOS

El objetivo de este trabajo es crear un videojuego de acción-sigilo con mapas generados en base a las dinámicas del jugador en su última partida con tal de adaptarnos a él, utilizando la generación procedural y aleatoria.

A partir de aquí diferenciaremos entre los objetivos planteados para la creación del videojuego (3.1) y los correspondientes a la generación del entorno de juego (3.2).

3.1. Creación del juego

El objetivo es crear un juego del género acción-sigilo en 2D con vista *top-down* en tercera persona, que además sea divertido a la vez que desafiante.

Nuestra meta es avanzar en un mapa eliminando a diferentes enemigos hasta encontrar a su jefe. Deberemos acabar con él para superar el nivel y pasar al siguiente, que será más complicado (habrá más enemigos y se generarán menos objetos donde esconderse). Hay seis niveles, que se desbloquean a medida que superes los anteriores.

El juego es Jugador contra Entorno (es decir, contra la máquina). Inicialmente se planteó que fuera un juego de Rol, pero tras implementar gran parte de las mecánicas que lo habrían definido dentro de ese género y ver que no era divertido se desestimó la idea para dar paso a la acción-sigilo, que parece funcionar mucho mejor.

No se busca hacer un videojuego accesible para todos los públicos, sino orientarlo a los jugadores más experimentados o casuales.

El objetivo inicial era desarrollar una cámara y personaje controlables mediante el teclado, enemigos (normales o jefe, con sus respectivas máquinas de estados finitos) y otros elementos estáticos como árboles, paredes, arbustos y lámparas.

3.2. Generación entorno de juego

En cuanto a la generación del entorno, que será posterior a la creación del juego, nuestro objetivo será generar entornos de juego distintos según las acciones del jugador en su última partida utilizando algoritmos que junten proceduralidad con aleatoriedad.

Para generar estos entornos modificaremos los elementos estáticos y dinámicos del mapa, que será una matriz cuadrada contenida en el objeto *Level Generator* [Figura 4], a cada nueva partida.

Los elementos estáticos (árboles, hierba, paredes y lámparas) serán los primeros en generarse siguiendo la idea propuesta por el Colapso de la función de onda, que utilizaremos para definir el número, tipo y disposición de objetos en la matriz de juego. Las probabilidades de que se posicione uno u otro elemento dependen de factores como el tiempo pasado en hierba la partida anterior o los elementos cercanos a cada posición de la matriz.

Los elementos dinámicos (enemigos) se generarán posteriormente en posiciones aleatorias de la matriz, dando mayor importancia a aquellas donde el jugador haya pasado más tiempo en la partida anterior. El número de estos dependerá de cuántos hayamos eliminado o si morimos en la última partida.

Fig. 4: Representación gráfica del objeto *Level Generator*

4 METODOLOGÍA

Este es un proyecto donde los tiempos de cómputo son prácticamente inexistentes. Cualquier implementación que se realice se podrá probar al instante. Pero más allá de esta ventaja, nos encontramos con que no sólo debemos programar el código correctamente sino que además el diseño de los elementos y sus interacciones debe sumar al juego, y no restar. Teniendo esto en cuenta, los objetivos quedan supeditados a las distintas decisiones de diseño que vayamos tomando a lo largo del desarrollo.

Los objetivos planteados nos obligan a tener que crear el juego en primer lugar, y seguir con la generación del entorno jugable. Dentro de cada uno de estos objetivos, el orden que sigamos de cara al desarrollo se irá entremezclando al interactuar unos objetivos con otros (por ejemplo, no podemos desarrollar los arbustos sin modificar el personaje jugable).

Además, durante el desarrollo de este trabajo se ha ido dejando probar el producto a distintas personas (nombradas en los agradecimientos) que han reorientado los objetivos y la clase de videojuego que se estaba creando, modificando

así las tareas a medida que se recibían opiniones para retroalimentar el proyecto.

Es por eso que he optado por la metodología ágil eXtreme Programming [Figura 5] [4], que me permite modificar las tareas y objetivos a medida que termino iteraciones.

Las iteraciones consisten en el análisis, diseño, desarrollo y pruebas de los diferentes elementos en que se han deconstruido en los objetivos. A medida que terminaba las iteraciones valoraba y decidía si debía redefinir ese objetivo. Al final de cada semana revisaba el progreso del proyecto con tal de seguir la planificación prevista o modificarla según exigiera la situación.

Gracias a utilizar esta metodología pude modificar los objetivos según veía que me hacía falta para tener un producto que cumpliera con mis expectativas. Desde cambios pequeños en ciertos elementos jugables, hasta haber pasado de desarrollar el videojuego en 3D a 2D o cambiar el motor en que se iba a desarrollar.

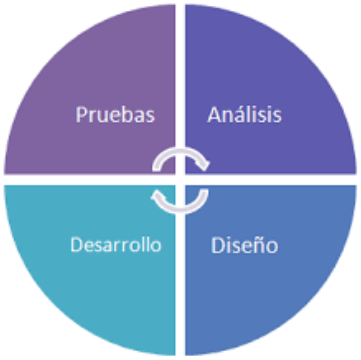

Fig. 5: Metodología eXtremme Programming

Las principales herramientas utilizadas han sido:

- **Overleaf**. Utilizado para redactar la documentación del proyecto y los distintos informes. Es una plataforma gratuita que no requiere licencias y nos ofrece un sistema de autoguardado.
- **Game Maker Studio 2**. Motor gráfico que he utilizado para el desarrollo del videojuego. A diferencia de Unity, este motor no cuenta con la opción de trabajar en tres dimensiones, pero en su lugar es mucho más potente para el uso de las dos dimensiones. Su lenguaje es Game Maker Language, basado en C++. Otro cambio respecto a Unity es que este motor sí que requiere de un pago inicial. Este es de 99$ en su versión completa más básica. Este motor incluye su propio editor de imágenes que a su vez nos permite crear animaciones. Esta elección se debe a que el motor cumple con todo lo que se necesita para desarrollar un proyecto de este calibre y es donde más cómodo me siento trabajando.
- **Entorno Google**. Además de Drive para almacenar archivos, herramientas como el buscador de Google o YouTube me han proporcionado una grandísima parte de los conocimientos necesarios para poder desarrollar este proyecto. A eso debemos sumar la herramienta Draw.io, integrada en el entorno Google, que he utilizado para crear diferentes diagramas.
- **Trello** [5]. Herramienta utilizada para llevar cierto orden respecto a las tareas que se han desarrollado a lo largo del proyecto. Creé distintas listas como 'Por hacer', 'Hecho' o 'En progreso', asignando las diferentes tareas a cada lista.

Por último, hacer mención a las dos webs que he utilizado prácticamente a diario a lo largo de todo el proyecto; el foro de la comunidad de GameMaker [6], donde prácticamente cualquier duda sobre el motor parece estar resuelta, y la guía de referencia de GameMaker Studio 2 [7], donde se encuentran explicaciones para todas las funciones y formas de funcionar de GameMaker Studio Language.

Sin estas dos webs, ambas oficiales y propiedad de la empresa desarrolladora del motor (*YoYo Games*), este proyecto no habría podido llegar a buen puerto.

5 Desarrollo

El proyecto está compuesto por una serie de elementos que forman la base del videojuego, en forma de objetos estáticos o dinámicos, y la generación del entorno jugable mediante una matriz de posiciones donde asignaremos los diferentes objetos que hayamos diseñado.

5.1. Elementos del videojuego

Se empezará explicando los elementos principales que componen el videojuego. Cada elemento dentro del motor gráfico Game Maker Studio 2 está representado por un objeto. Aquellos elementos de menor interés no serán tenidos en cuenta.

Los objetos en este entorno constan de una parte gráfica (formada por imágenes llamadas *sprites*), unas propiedades y un código propio. Es desde este código donde podremos crear interacciones entre los diferentes objetos.

Las dimensiones utilizada de forma estándar para la creación de la mayoría de objetos que forman el videojuego es de 16x16 píxeles.

Diferenciaremos entre objetos estáticos, que serán aquellos que una vez creados no modificarán su posición, y dinámicos, que podrán desplazarse por la matriz cuadrada donde se jugarán las partidas.

5.1.1. Elementos estáticos

- **Arbusto**. Objeto atravesable por el jugador. Cuenta con una animación compuesta por cuatro *sprites* distintos que crean la sensación de movimiento. Nos permite escondernos dentro y no ser detectados por el enemigo.

- **Pared**. Objeto no atravesable. Consta de una sola imagen estática. Hace de cobertura para el jugador, de forma que este podrá ocultarse tras las paredes sin ser detectado por los enemigos.
- **Árbol**. El tronco del árbol hereda las propiedades de la pared, de forma que tampoco es atravesable y servirá para ocultarse, además de contar con una única imagen. Además del tronco del árbol también estarán las hojas de este, que son un elemento meramente estético con una animación que consta de cinco *sprites* que tratarán de conseguir la sensación de movimiento.
- **Lámpara**. Un objeto del mismo tamaño que los muros y el tronco de los árboles. Cumple la misma función que las paredes permitiendo que nos podamos ocultar de los enemigos tras ella. Además cuenta con un área circular que ilumina a su alrededor y que no nos permitirá escondernos en arbustos. Este área parpadeará cada 30 a los 60 segundos, creando la animación en que la lámpara falla esporádicamente.

5.1.2. Elementos dinámicos: personaje controlable

El personaje controlable se trata de un objeto que cuenta con las siguientes variables:

- **Posición X**.
- **Posición Y**.
- **Velocidad horizontal**.
- **Velocidad vertical**.
- **Velocidad máxima**.
- **Aceleración**.
- **Ángulo del *sprite***.
- **Flechas**.
- **Escondido**.
- **En luz**.
- **Cámara**.

Se puede mover utilizando las flechas del teclado o el estándar WASD. Su velocidad y dirección dependerá de las variables de velocidad y aceleración, que modificarán la posición actual del personaje. La aceleración hace que empezar y terminar el movimiento no sea instantáneo, sino que la velocidad aumente o decremente a lo largo del tiempo.

Cuando se encuentre con un objeto estático no atravesable frenará hasta estar pegado a este, también reduciendo su velocidad mediante la aceleración y no parando en seco. Si en algún momento atraviesa uno de estos objetos (lo cual es frecuente al tratarse de colisiones *pixel perfect*) se moverá al jugador a la posición más cercana que no colisione con ningún objeto.

La representación gráfica del jugador rotará siempre apuntando hacia la posición del puntero, utilizando para esto la variable del ángulo del *sprite*.

Se podrá atacar con un arco utilizando el botón derecho del ratón. Mientras esté apretado, apuntaremos, hasta que dejemos de pulsarlo y la flecha se libere. Inicialmente se tendrán tres flechas. Estas desaparecerán al colisionar con enemigos (a los que eliminarán) o colisionarán con objetos estáticos del mapa quedándose çlavadas.[en] estos. Podremos recuperar estas últimas pasando por encima de ellas.

También se podrá atacar con una espada utilizando el botón izquierdo del ratón, siempre que no estemos apuntando con el arco. Si la espada incide con un enemigo, este será eliminado.

Además podrá ocultarse de los enemigos estando en el interior de arbustos siempre que estos no estén iluminados, entrando en estado .[es]condido". Si el arbusto donde tratamos de escondernos está iluminado por las lámparas se anulará la ocultación.

Por último, la cámara seguirá la posición del personaje controlable en todo momento, además de añadir otros efectos como zooms cuando combatimos con enemigos o un ligero temblor al apuntar.

5.1.3. Elementos dinámicos: enemigos normales

Estos cuentan con una máquina de estados finitos que define sus estados.

Su estado inicial es 'Patrulla'. En este estado patrullarán una zona definida por su punto de aparición y otro punto aleatorio dentro de los límites de la matriz que forma el mapa.

Si el jugador entra en alguna de sus dos áreas de influencia [Figura 6] y no está tras ningún objeto no atravesable o escondido y no iluminado, el enemigo lo detectará y pasará al estado 'Siguiendo', en que perseguirá al jugador hasta alcanzarlo y pasar al estado 'Combate', en que atacará con su espada (de forma que si impacta al jugador lo eliminará).

Si durante el estado 'Siguiendo' deja de ver al jugador, volverá al estado 'Patrulla'. También lo hará tras atacar.

Fig. 6: Áreas de influencia de los enemigos

El movimiento de los enemigos utiliza unas funciones de Game Maker Studio 2 que nos permiten crear un área sobre

la que estos podrán desplazarse. Este área nos permite definir qué objetos en ella no serán atravesables, y asignar las coordinadas a las que queremos que se dirija el objeto enemigo (que será su patrulla o el personaje controlable en caso de haberlo detectado).

Los enemigos aumentarán su velocidad cuando detecten al personaje controlable.

5.1.4. Elementos dinámicos: enemigo final

El enemigo final funciona de la misma forma que lo hacen los enemigos normales, pero con pequeñas modificaciones.

Cuenta con una burbuja a su alrededor que le permite resistir dos golpes, siendo esta burbuja eliminada en el momento en que la espada o una flecha del jugador la impacte.

También hace uso de una máquina de estados finitos. En su caso, si detecta al jugador y hay enemigos vivos, o si se elimina la burbuja que lo protege, pasará al estado "Huir."[ei]rá hasta el enemigo normal más cercano, persiguiéndolo indefinidamente.

Si eliminamos a todos los enemigos y nos detecta se quedará quieto, con un efecto que trata de imitar un temblor asociado al miedo, y se quedará pasivo ante nosotros. No nos atacará, tan sólo esperará hasta que acabemos con él.

5.2. Generación entorno de juego

Para entender cómo funciona la generación de los entornos donde jugaremos las distintas partidas es importante saber que utilizaremos una matriz de 40x40 casillas como lienzo en blanco [Figura 7]. Estas casillas tendrán la dimensión estandarizada para los distintos elementos del juego, 16x16 píxeles.

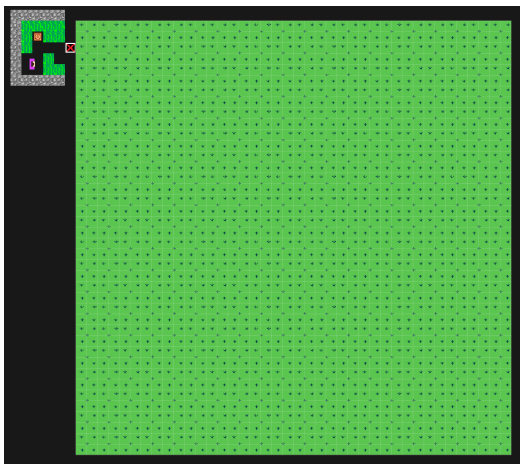

Fig. 7: Mapa previo a la generación del entorno

Toda la generación del entorno estará sujeta al objeto *Level Generator* que se nombró en objetivos. En este se almacenará el código necesario para crear cada partida.

También debemos tener en cuenta que los algoritmos utilizados para la generación de los objetos estáticos y dinámicos no están relacionados (si bien los segundos dependen de los primeros, ya que no podremos generar objetos dinámicos en casillas ya ocupadas por objetos estáticos no atravesables).

5.2.1. Contadores de tiempo

De cara a la generación del entorno jugable tendremos en cuenta distintos tiempos que se almacenarán a medida que juguemos una partida para ser aplicados en la generación de la siguiente.

Cada 60 fotogramas (que son los fotogramas que hay en un segundo) se sumará 1 a los distintos contadores afectados en cada momento. Estos son:

- **Tiempo total**. En esta variable se almacenará la duración total de la partida.
- **Tiempo en hierba**. Cuenta el tiempo que hemos pasado dentro de arbustos durante la última partida.
- **Tiempo en casilla**. En un rango de 64x64 píxeles (o 4x4 casillas) se sumará el tiempo a cada una de las casillas de la matriz que entren en este rango. Se ha decidido hacer así y no calculando el tiempo dedicado a cada casilla específica por tal de que los enemigos se generen en posiciones aproximadas a nuestro anterior camino, y no exactamente en este (ya que esta es la función del tiempo en casilla; la generación de los enemigos).

Posteriormente, conociendo el tiempo total de una partida y el dedicado a estar en el interior de los arbustos podremos disminuir o aumentar la probabilidad de aparición de arbustos en la siguiente partida.

Lo mismo pasa con el tiempo en casilla, que nos permitirá generar un mapa de probabilidades en la matriz de 40x40 de cara a la generación de los enemigos. Aquellas posiciones donde se haya invertido más tiempo será en las que mayor probabilidad de aparición de enemigos habrá.

5.2.2. Inicialización del entorno

El primer paso que se llevará a cabo para generar el entorno de juego será crear los muros que delimitaren el espacio del juego. Estos serán creados mediante dos bucles for (uno para los muros horizontales, otro para los verticales) que recorrerán todo el contorno del mapa.

Una vez hecho esto crearemos la matriz de 40x40 casillas donde se generará el entorno, e inicializaremos todos los valores que almacenarán la distintas casillas:

- **Observado**. Utilizaremos esta variable para saber si ya se ha observado una posición de la matriz o no. Por observar entendemos que ya se haya asignado un objeto a

la casilla, aunque este no sea nada. Inicialmente todas las casillas de la matriz permanecerán sin observar.

- **Objeto asignado**. Almacena el nombre del objeto que se generará en cada casilla. Inicialmente será nulo, ya que no habrá ningún objeto asignado.
- **Peso**. Valor utilizado para conocer cuál es la casilla de la matriz que observaremos en cada iteración. El peso se modifica a medida que iteramos con el Colapso de la Función de Onda. Inicializaremos todas las casillas con un valor aleatorio entre 0 y 0'5.
- **Probabilidad hierba**. Probabilidad de que aparezca hierba en la casilla que estemos observando. Esta probabilidad se calculará utilizando el tiempo en hierba que calculamos la partida anterior. Por defecto, la probabilidad de que aparezca hierba en una casilla es del 0'5 %, desde dónde sólo podrá disminuir.
- **Probabilidad pared**. Probabilidad de que aparezca una pared en la casilla que estemos observando. Por defecto, la probabilidad es del 0'5 %.
- **Probabilidad árbol**. Probabilidad de que aparezca un árbol en la casilla que estemos observando. Por defecto, la probabilidad es del 0'08 %.
- **Probabilidad lámpara**. Probabilidad de que aparezca una lámpara en la casilla que estemos observando. Por defecto, la probabilidad es del 0'03 %.
- **Probabilidad nada**. Probabilidad de que no aparezca nada en la casilla que estemos observando. La probabilidad de que esto suceda es del 0'75 %.
- **Peso enemigo**. Este peso, similar al anterior, se utilizará para saber las casillas donde generaremos a los enemigos. Se calcula a partir de la suma de un valor aleatorio entre 0 y 2'1 y el tiempo que dedicamos a cada casilla en la partida anterior entre el tiempo total de la partida, de forma que podrá sumar hasta 1 a la casilla en cuestión.

Una vez inicializados estos valores para cada casilla de la matriz todavía faltarán por calcular las variables relacionadas con el nivel de dificultad.

La primera de estas será el número de enemigos que se generarán. Para calcularla seguiremos la siguiente fórmula:

- **var probEnemies = 2 - (global.enemiesKilled / global.nEnemies);** Se calcula un coeficiente a partir de restar a 2 el número de enemigos hayamos matado en la partida anterior, entre el número de enemigos totales.
- **global.nEnemies = (ceil(probEnemies * 4)) + (global.actualLvl * 2);** Se multiplica un número de enemigos predefinido por la probabilidad calculada anteriormente y se redondea al alza. A esto se le suman dos enemigos por cada nivel de dificultad. Este valor se asigna a la variable que controla el número de enemigos totales de la partida.
- **global.enemiesKilled = 0;** Una vez ya se ha utilizado el valor almacenado con los enemigos eliminados la última partida, lo volvemos a igualar a cero para empezar la siguiente.

Por último calcularemos el número de objetos que posicionaremos en el mapa de forma previa al uso del Colapso de la Función de Onda. Esto se hace con el objetivo de que, a mayor sea la dificultad, menos objetos se generen en el mapa. La fórmula a seguir será:

- **var objPrefabsLevel = 24 - global.actualLvl*3;** Desde 21 objetos en el primer nivel de dificultad, generaremos 3 menos a cada nivel que aumentemos.

Una vez calculado este valor se utilizará para generar ese número de objetos aleatorios en posiciones aleatorias de la matriz. Según se generen estos objetos se recalcularán los pesos de cada casilla y aquellas que hayan recibido objetos pasarán a estar observadas. Los pesos de las casillas colindantes a las que han recibido objeto aumentarán, y las probabilidades de aparición de objetos también.

Una vez se ha inicializado y calculado todo lo expresado anteriormente, estamos preparados para utilizar el Colapso de la Función de Onda.

5.2.3. Generación del mapa. Colapso de la Función de Onda

El Colapso de la Función de Onda es un algoritmo donde lo primero que haremos será inicializar una matriz donde todas las casillas estén en un estado no observado.

En nuestro caso ya hemos creado esta matriz, y hemos añadido ciertos elementos a esta para que la generación del mapa tome en cuenta el nivel de dificultad.

El siguiente paso es recorrer toda la matriz en busca de la casilla no observada con el valor más elevado. Una vez sepamos qué posición es la indicada pasaremos a definir un estado en esta. Se iterará múltiples veces generando números aleatorios del 0 al 1 hasta que alguno de estos sea inferior a la probabilidad de generarse un objeto.

Cuando esto pase, asignaremos el objeto que haya tocado a la casilla que estamos observando, y esta pasará a considerarse observada y por tanto colapsada en un estado. Además de esto, los pesos y probabilidades de las casillas colindantes a la que se ha observado se modificarán de la siguiente manera:

- **Peso**. Al peso inicial (entre 0 y 0'5) se le sumará 0'25.
- **Probabilidad hierba**. A la probabilidad inicial, que era de máximo un 0'5 %, se le sumará un 2'5 % si en la casilla observada se ha generado hierba.
- **Probabilidad pared**. Si en la casilla observada se ha generado una pared, la probabilidad de pared en las casillas que la rodean aumentará en un 2'5 % sobre el

0'5 % inicial. Si en su lugar se ha generado una lámpara o un árbol, la probabilidad de pared se reducirá al 0 %.

- **Probabilidad árbol**. Si en la casilla observada se ha generado una pared o una lámpara, la probabilidad de árbol en las casillas que la rodean se reducirá al 0 %. Lo misma pasará si en la casilla observada se ha generado un árbol, pero en este caso afectará a un área de 5x5 casillas (tomando por centro la casilla observada).
- **Probabilidad lámpara**. Si en la casilla observada se ha generado una pared, un árbol o una lámpara, la probabilidad de lámpara en las casillas que la rodean se reducirá al 0 %.
- **Probabilidad nada**. Esta probabilidad se mantendrá igual en cualquier caso.

Se seguirá iterando con los nuevos pesos calculados (a esto se lo conoce como **propagación**, ya que se propaga nueva información conseguida en el anterior paso de observación) hasta que se haya observado cada casilla de la matriz, momento en que podremos decir que todas ellas han colapsado.

Con un objeto asignado a cada posición de la matriz, volveremos a recorrer esta creando los diferentes elementos.

5.2.4. Generación enemigos normales

Para generar a los enemigos normales utilizaremos los pesos para enemigos asignados a cada casilla de la matriz, cuyo cálculo se ha explicado en el apartado de **Inicialización del entorno**. También utilizaremos el número de enemigos que calculamos previamente.

Se recorrerá toda la matriz buscando la casilla con el peso para enemigos más elevado. Una vez la encontremos, se cambiará el valor de esta posición a 0 (para no volver a ser escogida) y se generará un enemigo en ella.

Después se definirá un punto aleatorio en el mapa, que será el que tendrá que patrullar el enemigo. Este punto estará siempre a un mínimo de 3 casillas (48 píxeles) de distancia del enemigo.

Terminaremos una vez se hayan generado todos los enemigos que debe haber en el mapa.

5.2.5. Generación enemigo final

Se calculará una media con los pesos para enemigos correspondientes a tres espacios de 15x15 casillas tal y como están delimitados en la figura que encontramos a continuación [Figura 8]. Utilizando esta media sabremos en cuál de las tres áreas se debe generar el enemigo final.

Dentro del área seleccionada buscaremos la casilla con mayor peso para enemigo, que es donde se generará el enemigo final. Tras esto se definirá un punto aleatorio dentro de este mismo área, que será el seleccionado para la patrulla del jefe.

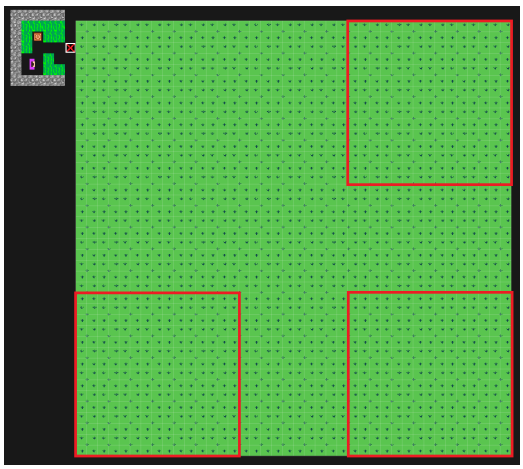

Fig. 8: Mapa con las zonas de generación del enemigo final marcadas

Con esto habremos terminado la generación del entorno de juego, personalizado en cada partida y teniendo en cuenta acciones del jugador en la última partida.

6 Resultados

Se ha conseguido desarrollar todos los elementos propuestos, desde el videojuego de acción-sigilo hasta la generación procedural del mapa, pasando por todos los elementos de los objetivos.

Para la discusión de resultados volveremos a dividir estos en los dos objetivos principales del proyecto (creación del juego y generación del entorno) y terminaremos con una breve conclusión general de los resultados.

6.1. Creación del juego

Todos los elementos que se quisieron diseñar desde un inicio se han podido realizar correctamente. Además, todos interactúan correctamente entre ellos. Prácticamente no hay errores o *bugs*, y los pocos que no se ha logrado solucionar no impiden poder disfrutar de la experiencia.

Por todo ello, y como ha alcanzado su estado de madurez, el videojuego puede darse por completado. Si bien ha habido cambios, como pasar del 3D al 2D o que finalmente sea un juego de acción-sigilo, todos ellos se han realizado teniendo en cuenta las opiniones de terceras personas y la mía propia, constituyendo decisiones de diseño que son subjetivas y definen a qué clase de público se orienta el producto.

En este caso se ha orientado The Last Uprising (nombre definitivo del juego) hacia un público al que le gusten los juegos complejos de partidas muy rápidas y que requieren de mucha precisión y concentración. De otra forma, a poco que te vea un enemigo habrás muerto. Esta exigencia hace

que no sea un producto para cualquier persona, pero es la interpretación que he decidido darle.

Además de los objetivos que hubo inicialmente se han desarrollado otros como los menús [Figura 9], ciertos efectos para la cámara o la optimización en el rendimiento del juego, entre otras mejoras que abarcan incluso a los elementos presentes en los objetivos iniciales.

Fig. 9: Menú principal en The Last Uprising

6.2. Generación entorno de juego

Se ha conseguido generar diferentes mapas para cada partida [Figura 10] tomando en cuenta las acciones del jugador, haciendo uso de las generaciones procedural y aleatoria, con especial énfasis en el Colapso de la Función de Onda, del cual se ha cogido el concepto para hacer una adaptación propia dentro del lenguaje Game Maker Language.

Todos los objetivos respecto a la generación del entorno de juego se han cumplido correctamente. Cabe destacar que todavía se podría refinar más los patrones utilizados a la hora de decidir la disposición de los elementos, o cambiar el algoritmo para en lugar de generar los elementos uno por uno, generara pequeños trozos de mapa ya prediseñados que acabaran por ocupar el mapa entero. De esta forma se podría controlar mucho más el diseño de cada entorno de juego que se generara. Pero tampoco era un acercamiento que quisiera tomar, pues es el que suelen utilizar la mayoría de videojuegos que utilizan estas técnicas (ej. *Diablo*, *The Binding of Isaac*...) y prefería optar por algo diferente.

6.3. Opiniones de terceros

Durante el desarrollo se ha dejado probar el videojuego a distintas personas (citadas en los agradecimientos) que han ayudado a orientar el proyecto y buscar errores.

Además de esto, se ha pasado un cuestionario para la versión final del videojuego, de acción-sigilo y elevada dificultad, de partidas rápidas y dinámicas que se adaptan al jugador. El recibimiento ha sido muy positivo, especialmente entre personas experimentadas.

Este cuestionario se realizó con 6 jugadores; 2 experimentados, 2 casuales y 2 que no juegan a videojuegos. Las preguntas consistían en conocer la opinión sobre la dificultad y el entretenimiento del producto (con una puntuación del 1 al 10), además del tiempo que les había llevado superar todos los niveles y otras observaciones y opiniones finales.

Los dos jugadores experimentados contestaron a la pregunta sobre la dificultad con 7 y 8. Los casuales, con 8 y 10. Los novatos, ambos 10. En cuanto al entretenimiento todos respondieron entre 8 y 9, a excepción de uno de los no jugadores que puntuó con un 6.

En cuanto al tiempo que llevó superar todos los niveles, los jugadores experimentados tardaron 12 y 38 minutos. Los intermedios 1 hora y 23 minutos y 1 hora y 41 minutos. Entre los jugadores casuales, uno tardó cerca de 2 horas y el otro no consiguió superar todos los niveles (es el mismo que reportó un 6 en cuanto al entretenimiento).

La conclusión que se puede extraer de esta escueta etapa de testing es que el juego parece funcionar bien para el público al que va dirigido, pero que aún así se podrían tratar de adaptar para cualquier tipo de jugador y por tanto expandir el público objetivo.

7 Conclusiones

En la realización de este Trabajo de Fin de Grado se ha creado un videojuego y utilizado las tecnologías que conforman el estado del arte en lo referente a la generación de entornos jugables. Se han cumplido los objetivos propuestos inicialmente y el producto final parece agradar a la gente que ha podido probarlo.

La conclusión más importante que se extrae es la importancia de un buen diseño por encima de ofrecer más contenido con peor diseño. Un juego mal diseñado, especialmente utilizando técnicas de generación automática, puede ser muy aburrido o incluso imposible de jugar.

Esto último que se ha comentado es el motivo por el que no se ha implementado el Aprendizaje Computacional, ya que no había ningún lugar en la generación del entorno en que pudiera aportar más que otras técnicas.

Los elementos que peor trato han recibido son el apartado gráfico y sonoro, que aunque no tienen demasiado que ver con las competencias del grado, me gustaría haber tenido más tiempo para procurar ponerlos al nivel del resto del proyecto.

En cuanto a las posibles líneas de futuro para este trabajo, se podrían mejorar los ya citados apartados gráfico y sonoro. También se podrían estudiar y establecer nuevas reglas a la hora de generar el entorno de juego para tener un diseño más controlado, o hacer uso de pequeñas habitaciones prediseñadas en lugar de tratar los elementos individualmente. Hay múltiples extensiones que podrían seguir ayudando a refinar el juego y sus algoritmos, pero no pienso que pueda evolucionar a ser otro tipo de experiencia. La idea del juego es una muy fija que funciona y que si se tratara de cambiar, no funcionaría, haciendo más rentable empezar un proyecto de cero reutilizando algunas de las

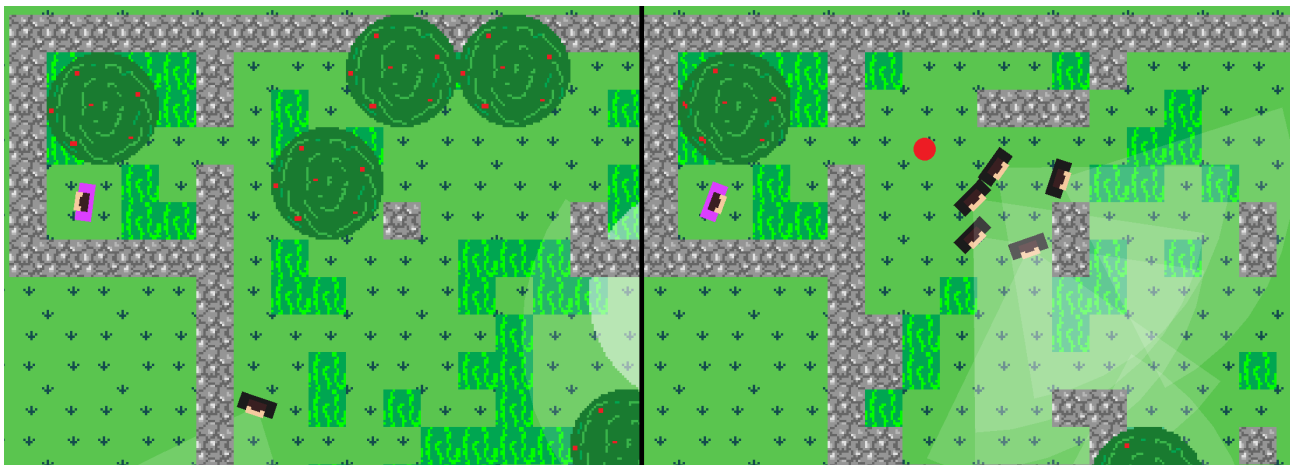

Fig. 10: A la izquierda, una partida empezada justo tras ejecutar el juego. A la derecha, otra tras morir en el punto indicado en rojo.

funciones de este (movimiento, colisiones, etc).

Por último, se podría tratar de hacer la experiencia más accesible para jugadores no experimentados a partir de que la generación de entornos ayudara más a estos jugadores (más lugares donde esconderse, menos enemigos o que estos aparecieran alejados de los caminos habituales del jugador, etc).

Agradecimientos

En primer lugar, agradecer a todos los profesores del Grado en Ingeniería Informática que durante estos años en que he cursado la carrera me han enseñado y hecho evolucionar ya no sólo a nivel académico y profesional, sino también personal. A todos ellos les agradezco y debo el haber llegado hasta este punto, en que tengo la oportunidad de seguir avanzando en mis metas con una base de conocimiento sólida.

De la misma manera, agradecer a mi tutora, *Debora Gil*, por haber hecho de golpe de realidad en los momentos en que no terminaba de encontrar el objetivo o alcance para mi proyecto.

También agradecer al profesor *Enric Martí*, quien me ayudó a dar el primer paso desde la casilla de salida y gracias a quien conseguí que me aceptaran la propuesta de este trabajo.

Agradecer a *Agustín Molina* por su apoyo emocional incondicional, sus aportaciones en cuanto a ideas para el diseño del videojuego y las incontables horas que ha dedicado a probar el juego, hacer *testing* y ofrecerme comentarios al respecto.

A la empresa en que trabajo, *HP Inc.*, por haberme permitido hacer tres semanas de vacaciones para terminar este trabajo, y a mis compañeros allí que me dieron todo el soporte y apoyo de que pudieron hacer acopio.

Por último, a RoboJIDA y sus integrantes (*Albert Villar*, *Iván Sánchez* y *Jordi Orihuela*), con quienes he dedicado una infinidad de horas en videollamada mientras ellos hacían sus Trabajos de Fin de Grado y yo el mío.

Por vuestras ideas, por vuestro apoyo, por vuestro conocimiento. Gracias a todos los que me han acompañado en este viaje.

Referencias

[1] M. Gumin, “Wave function collapse bitmap and tilemap generation from a single example with the help of ideas from quantum mechanics.” https://github.com/mxgmn/WaveFunctionCollapse. [Online; accedido el 12/06/2019].

[2] A. Fabrés, “Las fases del desarrollo de videojuegos.” https://bit.ly/2Xy56cH. [Online; accedido el 25/06/2019].

[3] P. Berger, “Is no man’s sky empty?.” https://bit.ly/2YfGwdM. [Online; accedido el 25/06/2019].

[4] “Extreme programming.” http://www.extremeprogramming.org. [Online; accedido el 6/03/2019].

[5] “Trello.” https://trello.com. [Online; accedido el 15/06/2019].

[6] “Game maker community.” https://forum.yoyogames.com/index.php. [Online; accedido el 15/06/2019].

[7] “Game maker studio 2 - language reference.” https://docs2.yoyogames.com/. [Online; accedido el 15/06/2019].