UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

Creación de un videojuego experimental 2D

TRABAJO DE FIN DE GRADO

Doble grado en Ingeniería de Software y Tecnologías de la Sociedad de la Información

Curso académico 2019-2020

Autora: Marta García Pérez

Tutor: Jesús Mayor Márquez

RESUMEN

En este Trabajo de Fin de Grado se quiere proponer un algoritmo para la generación procedural de terreno para un escenario de un videojuego en dos dimensiones. La generación procedural consiste en la generación de contenido a través de un algoritmo, lo que nos permite dar lugar a creaciones de escenarios de lo más variadas y abundantes.

En nuestro caso, para el desarrollo de dicho algoritmo se ha utilizado la función de ruido de Perlin, que genera un conjunto de valores continuos en el espacio. Esta función se utiliza originalmente para la creación de efectos especiales, pero aquí ha sido utilizada para la distribución de los distintos elementos que conforman el mapa. Según el valor obtenido de la función en cada posición, se coloca un elemento distinto. Como la función mantiene continuidad entre sus valores, esto nos permite controlar las transiciones que queremos que haya entre los distintos elementos.

Se ha utilizado junto con un sistema inteligente de posicionamiento de tiles para mejorar la apariencia visual del terreno. Éste es capaz de mostrar una representación diferente de cada elemento según cuáles le rodeen, lo que además nos permite que el terreno cambie en caso necesario cuando un jugador interacciona son él.

Para poder mostrar el funcionamiento de este algoritmo, se ha desarrollado un videojuego 2D que nos permita el uso de este tipo de generaciones. Se trata de un juego para dos jugadores cuyo objetivo es recoger una mayor cantidad de tesoros y conseguir más puntos que el otro jugador. Para el desarrollo de la lógica de los personajes se ha utilizado un sistema de detección de colisiones para provocar distintos eventos que le permitan su interacción con el entorno.

Gracias a la función de ruido de Perlin se ha conseguido crear un videojuego que presente una gran cantidad de escenarios únicos que los jugadores pueden disfrutar en cada partida.

ABSTRACT

In this *Trabajo de Fin de Grado,* an algorithm for the procedural terrain generation for a two-dimensional videogame scenario is proposed. Procedural generation consists of creating content through the use of an algorithm, which allows us to generate a huge variety of scenarios.

For the development of this algorithm, it has been used the Perlin Noise function which generates a set of continuous values in space. This function is originally used for the creation of special effects, but this time, it has been used for the distribution of the elements that shape the map. Thus, depending on the value obtained from the function at a certain position, a different element is placed on the map, and while the function maintains its continuity between its values, we are able to control the transitions that we want between the different elements.

Moreover, an intelligent tile positioning system has been used in combination with this algorithm to improve the visual appearance of the terrain. This system has the capacity of showing a different representation of an element depending on the elements surrounding it. Also, this characteristic makes it possible for the terrain to change, if necessary, with the players' interactions.

In order to show the functionality of this algorithm, a 2D video game that allows this type of generation has been developed. This game is designed for two players whose main goal is to collect more treasures while earning more points than the other player. For the development of the characters' logic, a collision detection system has been implemented to invoke different events that would allow the characters to interact with the environment.

Thanks to the Perlin Noise function, it has been possible to create a video game with a great number of unique scenarios that players can enjoy in each game.

Índice de contenido

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

El mundo de los videojuegos tiene una gran relevancia en el mundo actual. Los hay de una gran variedad de tipos y temáticas, lo que ha logrado que se expandan y se abran un hueco en la vida de muchas personas.

El objetivo principal de los videojuegos es entretener, y a través de éste se pueden conseguir muchas otras metas, como enseñar o ayudar a desarrollar ciertas habilidades. Esto ha logrado que se haya creado una gran industria dedicada a su desarrollo y se distribuyan nuevos títulos continuamente, tanto por grandes empresas o corporaciones, como por pequeños grupos de personas que tienen una idea y quieren sacarla a luz.

Se producen continuos avances en este ámbito, que aparte de ayudar en el desarrollo de muchos otros campos, logran que se mejore la experiencia de uso. Además, permite que la tecnología necesaria en el desarrollo se expanda y sea accesible por un número mayor de personas.

No todos los videojuegos son para todos los gustos y es por ello que hay una gran variedad de temáticas. Incluso dentro de cada temática podemos encontrar distintos modos de juego. Pero hay algo que todos tienen en común: la existencia de un escenario.

Un escenario, lejos de ser simplemente una representación del entorno en la que se encuentra el personaje, en caso de haberlo, ofrece un amplio conjunto de interacciones y restricciones que se le presentan al jugador, y que le dificultan o asisten en la consecución de su objetivo.

Este escenario es, en gran medida, el responsable de garantizar el entretenimiento al jugador. Por esta razón, todos los videojuegos buscan tener un escenario de calidad que ayude a “atrapar” al usuario.

Esta concepción de calidad para un escenario varía en cada juego. En ocasiones, esta se basa en unos gráficos muy detallados, en otras, por la cantidad de interacciones que te permite llevar a cabo, e incluso algunos, simplemente por la claridad o simplicidad con la que te lo presentan.

Otro aspecto que se tiene en cuenta a la hora de desarrollar un videojuego, y en concreto un escenario, es la perspectiva se le quiere dar al jugador.

En concreto, en este Trabajo de Fin de Grado (en adelante TFG), se quiere desarrollar un juego simple, similar al juego “MineBombers”. En este juego, ambientado en una mina, el jugador tiene como objetivo conseguir más oro que el otro jugador o intentar agotar todas sus vidas para que éste no lo consiga. Para ello, debes abrirte paso a través del mapa, utilizando armas que te permiten destruir elementos que te bloquean en camino.

En este título se ofrece al jugador una visión del terreno desde arriba, lo que te permite analizar el mapa y tomar decisiones según tu entorno. Esto se consigue colocando la

cámara a una cierta distancia vertical sobre el terreno de juego, obteniendo así una vista similar a la que se tendría desde un helicóptero. Este tipo de perspectiva se llama vista cenital o *top-down*.

1.1 Planteamiento del problema

Se acaba de mencionar la importancia de tener un escenario de calidad, pero hay veces en las que tener un solo escenario no es suficiente. En muchas ocasiones, un jugador acaba abandonando un juego porque considera que ya ha explorado todo lo que su entorno podía ofrecerle y comienza a parecerle aburrido y repetitivo. Además de la posibilidad de que el jugador llegue a aprenderse el escenario y sepa que esperarse en cada momento de la partida. Esto provoca que no baste con tener un solo escenario de calidad, sino que necesitamos muchos, de manera que el jugador sienta que descubra un nuevo mundo cada vez que juegue.

La cantidad de contenido que ofrece es uno de los factores más importantes que se tienen en cuenta a la hora de decidir si merece la pena invertir en un juego concreto. A parte de la cantidad también es esencial la variedad. Luego si el mundo que se le presenta al jugador no resulta estimulante, éste se cansará rápido de ese contenido y no lo querrá continuar disfrutando. Para que evitar que esto ocurra, hay que crear contenido que resulte entretenido.

La creación de un escenario no es una tarea sencilla. Hay que llevar acabo al menos dos tareas fundamentales: su diseño y su desarrollo.

A la hora de diseñar el escenario, hay que determinar cuál es el objetivo que se quiere conseguir con él. Se deben tener en cuenta las restricciones y mecánicas a implementar en el juego, atendiendo a diferentes características que nos permitan alcanzarlo. Por ejemplo, en caso de que queramos construir un laberinto, y deseemos distintos niveles de dificultad para el jugador, podremos añadir elementos como enemigos que le dificulten el avance, o pistas que le ayuden a escoger el camino correcto.

Una vez se tenga el diseño acabado, hay que pasar a darle forma en el juego. Debes desarrollar las interacciones que hayas determinado en la fase anterior y crear y colocar cada uno de los elementos que lo conforman. Siguiendo el ejemplo del laberinto, además de crear el laberinto que hemos diseñado con todos sus materiales y dinámicas, también deberemos crear aquellos elementos que se relacionarán con el jugador y todas sus interacciones.

Volviendo al primer punto, un solo escenario no será suficiente. El objetivo del juego a desarrollar es simple, conseguir recolectar más tesoros que el otro jugador. Una vez los jugadores hayan recorrido el mismo mapa varias veces, encontrarán repetitivo el juego y sentirán que están jugando constantemente la misma partida.

Por ésta razón, necesitamos un gran número de escenarios variados que mantengan entretenido al jugador y le hagan disfrutar con cada nueva partida.

1.2 Objetivos

Con los problemas expuestos en mente, queremos alcanzar una solución que nos permita solventarlos de la mejor manera posible, creando una experiencia de juego agradable para el jugador.

Por ello, los objetivos que se establecen en este TFG son los siguientes:

- Utilizar técnicas de generación procedural de escenarios para la creación automática de nuevos mapas de juego.
- Desarrollar un algoritmo capaz de configurarse para que los mapas generados cumplan con unas ciertas pautas y se adapten al resultado que se pretende conseguir.
- Desarrollar un videojuego en dos dimensiones que nos permita utilizar este tipo de técnicas de generación procedural.
- Garantizar que en cada nueva partida se presente un nuevo escenario al jugador, de manera que no juegue dos veces el mismo mapa.
- Conocer la opinión de un grupo de muestra sobre los resultados obtenidos con el método de generación procedural desarrollado.

2 ESTADO DE LA CUESTIÓN

Para explicar claramente el desarrollo de este TFG, expondremos algunos puntos relacionados con el concepto de la generación procedural que vamos a utilizar y desarrollar, así como otros métodos de este mismo ámbito que se han utilizado en otros proyectos.

2.1 En qué consiste la generación procedural

La generación procedural o generación por procedimientos en un videojuego se puede definir como "la creación algorítmica del contenido de un juego con la intervención nula o limitada del usuario" [1].

Para una mejor compresión de la definición, aclaremos cada uno de los términos mencionados. Por creación algorítmica entendemos una creación guiada por un conjunto de reglas y procedimientos estipulados por el desarrollador para conseguir el contenido que se desea. El contenido, es todo aquello con lo que el usuario puede interaccionar, ya sea en el juego o durante el desarrollo. Por usuario en este caso, se refiere tanto el jugador como los desarrolladores del contenido, ya que ambos pueden participar en la generación. Y por su interacción, aquella información que debe o puede aportar el usuario para condicionar la generación del terreno, como por ejemplo el tamaño o la forma que tendrá el mapa. La cantidad de interacción necesaria determina el grado de independencia de la generación procedural [1].

Explicado de manera más simple, la generación procedural es aquella en la que el contenido se genera por medio de algoritmos, reduciendo así la necesidad de creaciones manuales.

Esto nos ofrece la posibilidad de crear una gran cantidad de contenido y de gran variabilidad, en un menor espacio de tiempo. En la mayoría de los casos, la cantidad de contenido que nos ofrece el algoritmo es infinita, si bien no todas las posibilidades nos resultan útiles a la hora de alcanzar nuestro objetivo. Para ello se utilizan algoritmos de ramificación y poda que nos ayudarán a descartar aquellos contenidos que no consideramos óptimos [2].

Además, como es el algoritmo de generación quien se encarga de realizar estas creaciones, la generación procedural nos permite no tener que guardar cantidades tan grandes de información como necesitaríamos originalmente en un desarrollo tradicional, ya que lo elementos serán generados en el momento de su uso. Por el contrario, nos exige una mayor velocidad de procesamiento para poder ofrecer simultáneamente una gran cantidad de recursos con buena calidad [2].

2.2 Historia de la generación procedural

La generación procedural no es un campo nuevo. Su desarrollo comenzó muchos años atrás con juegos como *Rogue* (1980), donde a través de caracteres ASCII, se representaba un conjunto de mazmorras que el jugador debía recorrer para encontrar un amuleto. Estas mazmorras, junto con los enemigos a los que te enfrentas y los objetos que recoges, son generadas de manera procedural, y cambian con cada nueva partida.

De este juego, surge el término "roguelike", que engloba a aquellos juegos que comparten una serie de características, como son la generación procedural de escenarios 2D, la muerte permanente del personaje (supone comenzar una nueva partida), una dinámica de juego basada en turnos, o que cada nivel esté compuesto de diversas salas conectadas [3] [4]. Esta línea de creación surge del intento de imitar la dinámica de "Rogue", y acaban surgiendo otros juegos como "Hack y Moria" [4]. Es importante destacar que antes de "Rogue" ya había algunos títulos que dejaban ver este tipo de técnicas, como "The Game of Dungeos" o "DND" (1974) y "Beneath Apple Manor" (1978).

En 1984, salió "Elite", que era capaz de generar 8 galaxias con 256 sistemas solares con hasta un máximo de 12 planetas cada uno, e información de cada uno de esos planetas.

Por estas fechas, la generación procedural sufre un pequeño parón debido a las mejoras en el almacenamiento de la información, ya es posible conservar mundos relativamente grandes en memoria.

Con Diablo, en 1995, se vuelve a introducir la generación procedural en los nuevos sistemas de videojuegos, con generaciones de mazmorras y de objetos. Con esto, la generación procedural quedó relegada a poco más que juegos RPG o roguelike.

En 2006, Dwarf's Fortress fue lanzado, y utilizaba esta tecnología para generar el escenario no solo a través de la distribución de materiales, sino simulando también su erosión debido al agua y viento, el tiempo, y la interacción entre distintas razas y ciudades [5].

En 2008 nos encontramos con "Spelunky" y, posteriormente, vio la luz "Minecraft", en 2009. Este último videojuego tuvo un gran éxito, y sus algoritmos ofrecen al jugador un mundo formado por cubos aparentemente infinito.

Ese mismo año, se lanza "Borderlands", donde se usa la aleatoriedad de la generación procedural para desarrollar millones de combinaciones para armas y objetos, y también ajusta la dificultad de los enemigos a nuestro nivel de juego [6].

En algunos juegos como "Proteus" (2013), comenzamos a ver el uso de audio procedural. También nos lo encontramos en "No Man's Sky" (2016), que presenta uno de mayores usos de generación procedural hasta la fecha. En este título se genera un universo entero con millones de planetas únicos, con sus propias combinaciones para generar fauna [7] y flora distinta en cada uno de ellos, además de generar sonido

procedural ambiente en cada planeta. Este título generó una gran expectación, pero desgraciadamente no tuvo el éxito esperado.

2.3 Usos de la generación procedural

Se ha visto que la generación procedural ofrece numerosas ventajas, y por tanto, es de esperar que se use en múltiples campos.

Como ya se ha mencionado, uno de sus usos principales es en el mundo de los videojuegos. La generación procedural se utiliza tanto para la creación de escenarios y mapas, como de distribución de elementos del juego e incluso en la creación de recursos nuevos. También se puede aplicar para ajustar la dificultad o complejidad del juego según el nivel del jugador que lo esté jugando [1].

En el campo cinematográfico, también se aprovecha esta tecnología con numerosas objetivos. Entre ellos, nos encontramos la generación de escenarios realistas, como pueden ser montañas o poblados. Dentro de estas generaciones, también se utiliza en la creación de texturas y otras decoraciones, que podrán usarse en la propia generación procedural o como en las creaciones manuales.

Por otro lado, se puede destacar su aplicación en la generación de grandes masas de personas. Como ocurre en la saga de "El Señor de los Anillos", en que la que se utilizó MASSIVE[1] para la creación de escenas de guerra entre grandes ejércitos [8].

La generación procedural también se ha ganado un hueco en el mundo del sonido, principalmente usado en los videojuegos, recibiendo el nombre de audio procedural. Una definición formal sería la dada por Paul Weir: "El audio procedural es la creación de sonido en tiempo real utilizando técnicas de sintetización como el modelado físico usando enlaces profundos con los sistemas de los juegos" [9]. Esto viene a significar que sonará diferente música dependiendo de los eventos que estén ocurriendo en el juego.

A un nivel muy limitado, esto simplemente podría ser, que al ocurrir un cierto evento en el juego, como el ser descubiertos por un enemigo, una música distinta comenzaría a sonar. Pero el verdadero audio procedural es aquel que se va generando a medida que avanzamos en el juego, y no está creado con anterioridad. La creación de estos sonidos vendrá dada por variables que determinan los desarrolladores, y dependiendo de los distintos eventos o reglas a los que se condicionen, se conseguirá un mejor nivel de adecuación del sonido respecto a lo que está pasando en cada situación [9].

Algunos reproductores de audio, hacen lo que podría considerarse inverso, sacar imágenes de la música que se está reproduciendo [2].

[1] MASSIVE: Multiple Agent Simulation System in Virtual Environment. Paquete software desarrollado por Stephen Regelous para la realización de efectos visuales.

2.4 Clasificación de algoritmos procedurales

Existen distintas clasificaciones para los algoritmos procedurales. A continuación, vamos a exponer una de las más extendidas para este tipo de algoritmos, en función del tipo de objetivo que se persiga con ellos [5] [10] [11].

2.4.1 Generación Aleatoria de Niveles en tiempo de ejecución

Es probablemente la técnica más conocida. Consiste en la generación del mapa del nivel mientras el juego está en marcha, ya sea mientras se juega el nivel o antes de jugarlo. Deben definirse claramente los límites del mapa, ya que normalmente el mapa generado debe almacenarse en memoria mientras dure la partida.

Una vez creado el mapa, se suele utilizar otros algoritmos para cerciorarse de que es "jugable", es decir, que no se producen situaciones en las que resultaría imposible al jugador alcanzar su objetivo. Este proceso de prueba, suele verse intensificado en mundos 3D, lo que hace que resulte en ocasiones demasiado complicado para ser factible o práctico. A este tipo de generación pertenecen juegos como "Rogue" o "Spelunky".

2.4.2 Diseño de contenido de niveles

En este caso la generación es de contenido de los niveles, ya sean enemigos u objetos, se encarga de distribuirlos por el mapa. También se utiliza para crear formas base del mapa que después editarán manualmente los diseñadores.

Suele ser una característica que se oculta al usuario, ya que su objetivo es reducir el esfuerzo que deben llevar a cabo los desarrolladores.

Actualmente se utiliza en la gran mayoría de juegos, como por ejemplo "Minecraft"; y en otras herramientas, como "SpeedTree", que se utiliza para creación y emplazamiento de árboles y arbustos.

2.4.3 Instanciación de entidades en el juego

Se trata de la instanciación de objetos del juego con características que los hagan diferentes al resto de sus mismas instancias. Es decir, tienes un objeto base, y el algoritmo se encarga con la modificación de ciertos parámetros de darle un toque original, de modo que lo percibas como parte del conjunto de instancias de ese objeto,

pero también como un elemento individual y distinto a los de su grupo. De este modo, se evita la dar impresión de repeticiones.

Se puede usar de manera complementaria a la técnica anterior, siendo aquella la que distribuye los objetos y esta última las que los personaliza.

Uno de sus usos puede ser la creación de bosques, donde se instancian un gran número de árboles y cada uno tiene que ser distinto al resto para darle realismo. Se utiliza en juegos como “Left for Dead” o “X-COM: UFO” que lo utilizan para dotar de diferencias a soldados enemigos o aliados.

2.4.4 Contenido con intervención del usuario

En este tipo de métodos, el usuario es el que guía a la generación a la hora de crear el elemento. Como puede ser en la generación de un personaje, indicándole el tipo de físico que debe tener.

El algoritmo le da la oportunidad al usuario de obtener lo que quiere sin el usuario tener conocimiento técnicos necesarios para conseguirlo. De esta manera, la creación de contenido se ve expandida al alcance de muchas más personas, que a su vez pueden compartir sus creaciones para que otros las utilicen o basen sus propias creaciones en éstas.

Esto además tiene una ventaja, y es que las creaciones del algoritmo podría decirse que pasan por un filtro de censura, ya es el propio usuario quien determina si algo es correcto o no. Esta técnica se utiliza en el juego “Spore” para permitir al usuario crear una criatura y el algoritmo será el encargado de dotarla de “vida”.

2.4.5 Sistemas dinámicos

Se utilizan para modelar aspectos más genéricos, como el tiempo atmosférico o el día y la noche, o incluso generar un flujo de conversación encadenando frases ya estipuladas, como sucede en “Façade”, un videojuego que ofrece una historia interactiva al usuario que éste va dirigiendo.

También es el tipo de algoritmo que se encargaría de generar el sonido procedural que se ha mencionado anteriormente.

2.4.6 Generación de puzles y argumento

Estos algoritmos, basados en su mayoría en grafos, y nos brindan la oportunidad de crear un gran número de puzles con distintas soluciones, lo que provoca que no podamos encontrar la solución de otra manera que no sea jugando.

Respecto a la narrativa de una historia en el juego, podría igualmente utilizarse para controlar distintos caminos argumentales. De esta manera, al "fracasar" en tu intento de pasar un nivel, no tendrías que volver a comenzar el nivel, directamente se te crearía uno nuevo. Y aplicar esto a la historia podría ofrecernos múltiples hilos que recorrer y alargar de este modo la vida útil del juego, ya que el jugador se encontraría nuevas historias al comenzar cada partida. Desgraciadamente, este control argumental conlleva demasiada dificultad a la hora de su desarrollo y no resulta rentable.

2.4.7 Generación dinámica de mundos

Es el caso de mapas que se crean durante el juego, no antes. Hacen uso de una semilla que se mantiene fija durante la partida, y con ella se calcula que elemento debe ir en cada posición.

De ese modo, el mundo no debe almacenarse en disco, salvo la porción de él que está siendo visualizada por el jugador en ese momento. De esta manera, mientras recorres mundo o vuelves sobre tus pasos, se volverá a calcular el terreno que debe mostrar cada vez. Únicamente las modificaciones que realice el jugador sobre el terreno deben almacenarse para poder posteriormente mostrarlas.

2.5 Diferentes métodos para la generación procedural

Actualmente hay muchos métodos que se utilizan para la generación procedural de escenarios. Dependiendo que es lo que queramos conseguir con ello, serán más útiles unos métodos u otros. Siguiendo la estructura y descripciones que se exponen en [12], vamos a centrarnos en la explicación de algunos métodos que permiten la generación de los siguientes tipos de terrenos.

2.5.1 Generación de laberintos

Los algoritmos que se consiguen con las siguientes técnicas, son laberintos perfectos, que tal y como se estipula en [12], deben cumplir:

- “El laberinto ha de tener una única entrada y una única salida (éstas se suelen considerar la celda inferior izquierda y superior derecha, respectivamente).
- Todas las celdas que lo componen han de ser alcanzables.
- El laberinto no puede contener bucles (sólo existe un camino para llegar desde una determinada casilla hasta a otra).”

A continuación explicamos algunos de los posibles algoritmos para la generación de estos laberintos.

Depth First Search

La matriz que compone el laberinto tiene todas las paredes de todas las celdas levantadas. Un elemento agente se encarga de recorrer la matriz “derribando” las paredes por las que pasa. No puede pasar dos veces por la misma casilla, por lo que si se encuentra sin poder avanzar, retrocederá las casillas necesarias hasta encontrar un nuevo camino. Este método produce pasillos largos y unos pocos y muy cortos sin salida, por lo que resulta fácil encontrar la solución. En caso de crear laberintos grandes, este algoritmo toma mucho tiempo para volver sobre sus pasos sin realmente generar nada del laberinto, por lo que puede resultar poco eficiente [12].

Algoritmo de Prim

Igualmente con todas las paredes levantadas, este algoritmo comienza en una casilla aleatoria. Se tienen dos listas de casillas, las visitadas y las no visitadas. Cuando se visita una casilla, esta se añade a la lista de visitadas. Al visitar una casilla, se elige aleatoriamente a que casilla vecina visitar, y se derriba el muro que las une. La nueva casilla se añade a las visitadas. Después se escoge otra casilla aleatoria de la lista de visitadas y se repite el proceso, escogiendo de nuevo aleatoriamente a que vecino visitar. El algoritmo acaba cuando todas las casillas han sido visitadas. Este método provoca muchos cruces de caminos con pasillos cortos y sin salida, pero la solución al laberinto suele ser igualmente corta [12].

División Recursiva

Se comienza con las celdas de la matriz sin paredes, y lo que se hace es añadir un muro que divida la matriz en dos, no necesariamente del mismo tamaño. A continuación se abre un hueco en ese muro eliminando la pared de una de las celdas que lo forman. Después, se escoge la mitad más pequeña para volver a repetir el proceso, y la mitad grande se guarda en una pila. Una vez se ha continuado con el algoritmo hasta llegar a una división en la que cada mitad ocupe una celda, se extrae una de las partes

almacenadas en la pila para volver a repetirlo. Cuando ya no queden más partes en la pila se habrá acabado el laberinto. Los resultados de este método son muy impredecibles, pudiendo ser demasiado simples o muy complejos [12].

A continuación, en la Figura 1 podemos encontrarnos un ejemplo con laberintos resultantes de la aplicación de cada uno de los métodos que acabamos de explicar.

Figura 1. Ejemplo de laberintos generados por los algoritmos descritos [12]

2.5.2 Generación de mazmorras

Un mapa de mazmorras se caracteriza por estar formado por un conjunto de salas conectadas entre sí por un conjunto de pasillos. El personaje deberá recorrerlas recolectando objetos y enfrentándose a enemigos.

Partición binaria

Este método es similar al de división recursiva que hemos descrito antes para laberintos, pero en este caso se fija un umbral para el tamaño de cada "mazmorra". Se estipula el tamaño del mapa y este se va dividiendo en dos partes, escogiendo la más pequeña para volver a dividirla y la más grande almacenándola para posteriormente dividirla. En este caso, las divisiones se continúan hasta que el tamaño de las partes es inferior al umbral establecido y hasta que la pila quede vacía. Una vez la matriz se ha segmentado, en cada una de las celdas que se han creado, se crea una sala, de cualquier tamaño pero siempre respetando los límites de la celda. Con todas las salas creadas se pasa a conectarlas a través de pasillos. Primero se interconectan las salas "hermanas", es decir, aquellas que se generaron a través de la misma división. Si comparten pared, se hace a través de un pasillo recto, y si no se usará uno con forma de Z. A continuación subimos un nivel de división y conectamos las zonas que sean "hermanas". Se continúa subiendo niveles hasta que se han unido las dos mitades iniciales [13].

Basada en un agente

En una matriz definida, un agente va avanzando en línea recta en una dirección aleatoria. Por cada casilla que avance en línea recta, la probabilidad de que haga un giro aumenta. También por cada casilla que avance, aumenta la probabilidad de que el agente cree una sala de dimensiones también aleatorias donde se encuentre. Cuando realiza un giro o crea una mazmorra, las probabilidades de que ese mismo hecho vuelva a ocurrir se reducen a 0, para volver a aumentar de nuevo con el desplazamiento del agente. Este método genera mapas muy caóticos, lo que le da un aspecto muy realista al mapa, pero es también impredecible y tendente a generar mapas no válidos con mazmorras superpuestas [14].

2.5.3 Generación de terrenos 2D

Son terrenos representados por casillas, y cada casilla es ocupada por un elemento distinto. Estos elemento variarán dependiendo el terreno que se quiere construir, pero algunos básicos son agua, tierra o hierba.

2.5.4 Basado en un autómata cellular

Un autómata celular se representa con una matriz de tamaño fijo, en la que cada casilla está coloreada de un color, cada uno simbolizando un estado. Se comienza con todas las celdas inicializadas a un estado aleatorio. A partir de ahí, cada celda ve influenciado su estado por el estado de sus celdas vecinas, siguiendo unas reglas definidas. Hay muchos tipos de autómata. Se puede variar la definición de vecino, considerando como tales a las inmediatamente adyacentes que formen una cruz, o todas las adyacentes que la rodean; o incuso considerar también como vecinos a aquellas que están separadas una fila más. También se deben definir el número de estados, en caso de que sean dos, al autómata se le llama binario, y sus colores sería blanco y negro. Un autómata celular puede tener todas las dimensiones que se quiera, uno de dimensión uno sería una fila de celdas. Basándose en uno de estos autómatas celulares, se puede asignar cada estado un tipo de material que representar en ese lugar. El aspecto de este escenario vendrá determinado por el número de iteraciones que se realicen, el número de estados y las reglas que se definan [15]. En caso de que se quiera utilizar este algoritmo para la generación de mazmorras, se escogería un autómata binario, de manera que tenga estado “tierra” o “vacío”.

Diagramas de Voronoi

Estos diagramas han sido utilizados con una amplia variedad de objetivos, como el análisis espacial, análisis de asentamientos urbanos, geología y ecología [16]. Para su desarrollo, se parte de una matriz delimitada y se distribuyen aleatoriamente unas semillas. Ahora se generan regiones que estarán formadas por todos los puntos que se encuentran más cerca de una semilla que de cualquier otra. De esta manera cada semilla dará lugar a una región distinta. Como las semillas se colocan de manera aleatoria, algunos resultados no son óptimos, ya que si algunas semillas caen muy juntas sus regiones no tendrán una extensión realmente útil. Para evitar estos casos, a la hora de distribuir estas semillas se usan otros algoritmos para corregir su posición. También se suele usar en combinación con otros algoritmos para determinar de qué tipo de material es cada celda o región [12].

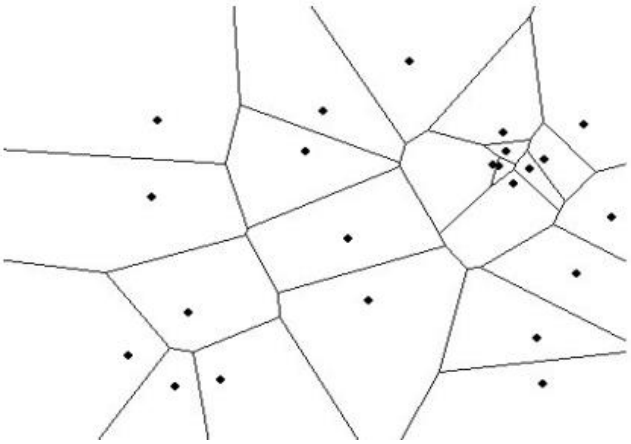


Figura 2. Representación de un Diagrama de Voronoi [16]

En la Figura 2, vemos un ejemplo de generación de estas regiones según unos puntos repartidos aleatoriamente. Los límites se han establecido por los puntos equidistantes a las semillas que se encuentran más cercanas. Como puede verse en la mitad derecha, hay algunos puntos que han sido colocados en zonas muy cercanas. Éstos, dependiendo del resultado que queramos conseguir, podrían no ser válidos, como se ha mencionado antes, y deberían ser recolocados.

2.5.5 Generación de terrenos en 3D

Este es el tipo de terreno más común en los videojuegos de hoy en día. Cuentan con las tres dimensiones presentes en el mundo real, luego deben tener en cuenta una gran cantidad de aspectos.

Basado en agentes

El uso de este método depende del terreno que se quiera generar. A cada agente se le asigna un elemento con el que trabajar y van actuando uno seguido de otro, luego dependiendo del número de elementos que queramos, habrá un número diferente de agentes distintos. Además, dependiendo del terreno que se quiera conseguir se establecerán a los mismos unas normas distintas que seguir. Este método proporciona a los desarrolladores más control e influencia en el terreno que se va a generar. Uno de los métodos más conocidos es el de **Doran y Parberry** [17], que genera terreno de estructura similar a una isla. Éste define tres tareas para la generación: "Coastline", en la que a través de distintos agentes se define el límite del terreno; "Landform", con un mayor número de agentes simultáneos se define el relieve del terreno; y "Erosion", fase en la que se crean ríos a través del terreno, tantos como agentes se utilicen en esta fase. Para realizar estas fases, identifican seis tipos de agentes:

- **Coastline agents**: son los primeros en actuar. Se parte con todo el mapa a nivel del mar y estos agentes se encargan de elevar el terreno. Se comienza con un solo agente que se irá multiplicando dependiendo del tamaño del mapa. Cada agente tiene un número de acciones que efectuar y después desaparecen. Se conseguirá un terreno más detallado si trabajan muchos agentes con un menor número de acciones, ya que cada uno se ocupará de una región menor.
- **Smoothing agents**: actuando en continuación a los anteriores, estos agentes se encargan de suavizar los cambios bruscos de altura. Se colocan en puntos aleatorios y van deambulando calculando nuevos valores de altura para los puntos que visitan. Esta nueva altura se calcula en base a la media de los puntos que rodean a éste.
- **Beach agents**: se encarga de dar forma a las zonas de playa. Se sitúan en los límites de la costa y reducen la altura de los puntos que visitan a un número aleatorio menor que un máximo establecido. Esto evita que haya montañas cerca de las playas.
- **Mountain agents**: estos agentes se encargan de elevar zonas del terreno por encima del nivel del mar para crear montañas. Al igual que los agentes de playa, éstos se distribuyen aleatoriamente por el terreno y realizan un número estipulado de pasos aleatorios. Su actuación va seguida de agentes suavizantes y de playa, para reducir los picos bruscos que se hayan podido generar. También se le suele aplicar una capa de ruido para volver a obtener alguno detales que se pierden por el suavizado.
- **Hill agents**: similares a los agentes de montaña, pero trabajan con menores alturas, rangos de aleatoriedad más pequeños y no pueden crear elevaciones que resalten.
- **River agents**: cada uno de estos agentes dará lugar a un río. Establecen un punto en una montaña y otro en el océano. Comenzando por el océano, comienzan a avanzar hacia el punto de la montaña. Con su avance van

creando una cuña de erosión por la que circula el río, y ésta se irá haciendo más estrecha y pronunciada a medida que se acerca a la montaña. También se establece una longitud mínima que debe tener un río.

Programación genética de terrenos

Se parte de algún terreno producido por algún otro medio, y se le aplican algoritmo genéticos para propiciar la evolución de este terreno hasta cumplir con una serie de requisitos. A partir de un mismo escenario de partida, pueden generarse distintos mapas finales dependiendo el proceso de evolución que haya seguido cada uno. Hay una gran cantidad de algoritmos genéticos que pueden ser usados.

También tiene una función de puntuación o aceptación que indica cómo de cerca está el territorio de cumplir con las expectativas impuestas, y determina si considera válido el terreno generado.

Un ejemplo es el método propuesto por Walsh y Glade, que utilizan el algoritmo genético para determinar los valores de los parámetros como la escala, datos atmosféricos o nivel del agua, de su generador de terreno. Cada uno de estos valores era representado con una cadena de 8 bits, y su mutación consistía en cambiar uno de esos valores. A la hora de cruzarlo, determinaba un punto que dividía la cadena en dos e intercambiaba una parte de los padres para dar lugar al nuevo elemento valor [18].

Basado en fractales

Observando en la naturaleza encontramos este tipo de formaciones, en los que una forma se repite dentro de otra con un tamaño menor. A la hora de generar terrenos realistas podemos usar igualmente este método.

Figura 3. Generación del fractal del helecho de Barnsley. [16]

Se suele conseguir a través de llamadas recursivas a una función que genera la forma que queremos conseguir, variando cada vez los parámetros que hacen reducir su

tamaño [16]. Hay muchos algoritmos diferentes que se pueden usar para generar fractales, pero uno de los más simples y más usados en los videojuegos es el **algoritmo Diamond-Square** [17]. Este método se comienza con el paso Diamond. Se establecen cuatro puntos con valores aleatorios que formarán un cuadrado. Y a continuación, se calcula el punto medio a través de la intersección de las diagonales del cuadrado. A este nuevo punto se asigna el valor medio de los valores de las esquinas más un número aleatorio. Después, se realiza el paso Square. Se hayan los puntos que se encuentran en el centro de los lados del cuadrado, y a estos nuevos puntos contendrán el valor obtenido de la media de los extremos de ese lado y el punto central, más un número aleatorio. Con todos los nuevos puntos generados se repiten de nuevo estos dos pasos. El rango de valores que puede tomar el número aleatorio determinará cuan escarpado será el terreno. Esto tiene una limitación, y es que se necesita aplicar sobre un rejilla de 2^N + 1, luego no puede utilizarse cualquier dimensión de terreno. [19].

También tenemos los Lindenmayer Systems o L-Systems, que se utiliza para la generación realista de plantas y fractales. Este método se basa en la reescritura de cadenas, de manera que tiene elementos que pueden cambiarse y otros que se mantienen fijos. Tiene una serie de reglas establecidas que va siguiendo sucesivamente para dar lugar a la cadena final que será representada a través de otros métodos. Este tipo de sistemas nos permiten una amplia configuración, por lo que son muy útiles y permiten realizar generaciones con un amplio rango de complejidad [16]. Una manera simple para trabajar con la representación de los L-systems es los “Turtle Graphics” que se basan en el desplazamiento de una “tortuga” que va dibujando líneas por donde pasa. También puede utilizarse para desarrollar un mapa de alturas, un concepto que desarrollaremos a continuación. Para esto, se establece a cada símbolo de la gramática del L-System un valor de altura, a la hora de generar el terreno, este valor se verá influenciado por los que tiene alrededor [20].

Mapa de alturas

Es una de las opciones más comunes, sirve como base para desarrollar el terreno. Consiste en una matriz que almacena la altura de cada uno de los puntos que contiene. El problema con este método es la elección de la altura de cada punto. Es por ello, que no se considera tanto un método, sino una herramienta para la aplicación de otros métodos. A la hora de utilizarla, hay que escoger algoritmos que produzcan datos similares a los del mundo real o al resultado que queremos obtener. Se pueden usar muchos de los métodos descriptos anteriormente.

Otra opción que tenemos es la **interpolación de terrenos**. Se comienza dando valores aleatorios a algunos puntos del mapa. Para que su apariencia resulte más fluida y realista, se realizan interpolaciones entre esos valores, de manera que obtenemos valores intermedios entre ellos. Hay diferentes maneras de conseguir esta interpolación. Una de las más simples, es la **interpolación bilinear**. Se consigue calculando la media

entre dos puntos, primero para el eje x y después para el eje y. Para que el resultado sea menos recto, se realiza una media ponderando los dos valores de los que partimos. Este método nos da perfiles muy escarpados, con muchos picos pronunciados y laderas muy rectas, que pueden no ser apropiados para lo que queramos conseguir. También tenemos la **interpolación bicúbica**. Muy similar a la anterior, pero en vez de calcular la interpolación a través una media ponderada, se utiliza alguna función que genere una forma de S. De esta menara, las laderas y montañas quedan mucho más suavizadas [17].

2.6 Método implementado: Perlin Noise

Tras haber hecho un recorrido por algunos de los algoritmos más utilizados en la generación procedural, se ha escogido el algoritmo Perlin Noise o ruido de Perlin como el método más apropiado para la creación del terreno que pretendemos conseguir.

Función de ruido

Entendemos por ruido a un conjunto de valores aleatorios. Es un fenómeno que se produce en muchos ámbitos y tiene distintas representaciones, como el sonido que se obtiene cuando se selecciona en la radio un canal con una frecuencia en la que no hay ninguna estación emitiendo, o la imagen formada por puntos blancos y negros que se obtenía en televisiones antiguas cuando sintonizabas un canal sin transmisión [21].

En nuestro caso, el ruido serán unos valores aleatorios que se almacenan en una línea o cuadrícula. Para aclarar más su definición, a continuación exponemos algunas propiedades que presentan las funciones de ruido [22]:

- Se definen correctamente en cualquier lugar en un espacio 3D.
- Sus valores varían en un rango conocido, generalmente [-1, 1].
- Es limitada de banda, ya que la frecuencia de su contenido o sus valores es limitada.

Los elementos naturales tienen pequeñas variaciones que las hacen únicas, esto hace que las creaciones producidas por ordenador sean demasiado regulares. El ruido puede permitirnos añadir esas características que las hagan parecer realistas.

Para que el ruido sea realmente útil en la mayoría de los casos, no vale con que devuelva valores aleatorios cualesquiera. Como ocurre con el ruido blanco, que presenta valores aleatorios que no guardan ninguna correlación entre ellos. El que exista esa relación ayuda a amoldar mejor los resultados de la función al objetivo que queremos conseguir, y por ello permite obtener resultados más realistas. Aquí es donde destaca el algoritmo Perlin Noise.

2.6.1 Algoritmo Perlin Noise

Este algoritmo es desarrollado por Ken Perlin en 1983 para generar representaciones realistas de elementos como fuego, humo o nubes en efectos especiales.

Esta función de ruido genera valores pseudoaletorios que guardan continuidad entre ellos. En la Figura 4 podemos ver un ejemplo producido por el algoritmo de ruido Perlin.

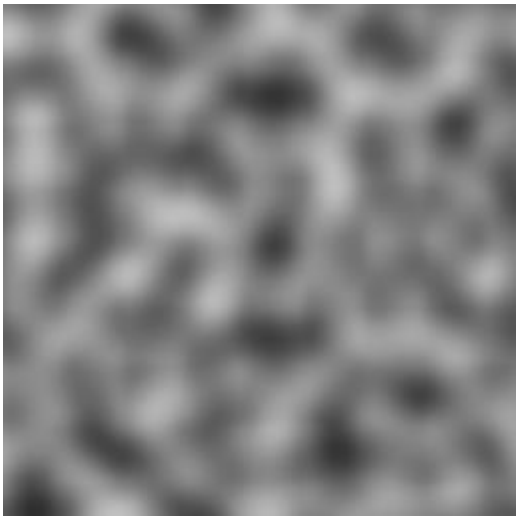


Figura 4. Representación 2D de valores producidos por la función Perlin Noise

Cada uno de los distintos tonos de grises que vemos nos indica los diferentes valores producidos por el algoritmo en esos puntos. Podemos apreciar la continuidad en los valores que ofrece el algoritmo gracias a los degradados que observamos, sin que haya cambios bruscos en el paso de blanco a negro o a la inversa. También vemos como a simple vista no se produce ningún patrón repetitivo.

La función se basa en una matriz para asignar los valores que corresponden a cada punto, y a partir de ésta lleva a cabo tres fases. Para desarrollar estas fases vamos a basarnos en una matriz de dos dimensiones, ya que es la que se utilizará en el desarrollo de este proyecto:

1. Definición de la matriz: en esta fase se crea la matriz. A cada vértice de cada celda de la matriz se le asigna un gradiente. Un gradiente viene a ser un vector normalizado de las mismas dimensiones que la matriz, en este caso 2. Este gradiente debe ser un vector aleatorio, y por ello se formará a partir 2 números pseudoaleatorios en el rango de [0,1]. Después se reposicionarán con rango [-1,1] y por último se normalizará el vector resultante [23].
   Para dimensiones mayores se pueden utilizar otros métodos para la obtención de estos valores con mejores resultados, como el método de MonteCarlo. También a la hora de reducir el coste computacional, se pueden almacenar valores de gradientes ya calculados en tablas hash y simplemente acceder a su valor a la hora de asignarlos o utilizarlos [23].

La razón de que este algoritmo presente “suavidad” en su continuidad viene dada por la utilización de estos gradientes. También son los responsables de que esta función sea limitada de banda, ya que al indicar la dirección en la que aumenta el valor, está determinando también en qué dirección decrece, es decir, en sentido contrario. Esto hace que no se produzcan frecuencias bajas, ya que la curva de valores baja y sube en espacios relativamente amplios, por lo que limita la frecuencia del contenido.

2. Producto escalar: ahora mismo tenemos cuatro vectores gradientes que parten de los vértices de la celda que contiene al punto que queremos evaluar, y lo que queremos obtener es un número real. La forma que propone Perlin de obtener este resultado, comienza por generar unos vectores dirección que partan desde cada una de las esquinas de la celda hasta el punto que queremos evaluar. Llegados a este punto tenemos dos vectores que tienen su origen en cada una de las esquinas, como se puede apreciar en Figura 5.

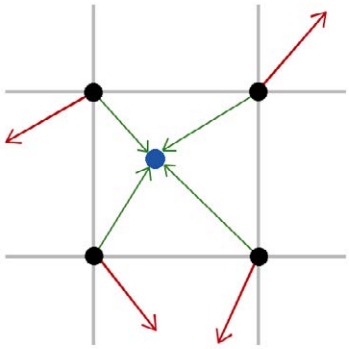


Figura 5. Representación de los vectores gradiente (en color rojo) y dirección (color verde) calculados por el algoritmo de Perlin

Ahora para obtener el valor real que queremos, realizamos el producto escalar de los dos vectores que tenemos. Esta operación nos da como resultado el número real que corresponderá al valor del vértice que estamos evaluando. Este paso se repite con todos los vértices que tengamos que rodeen a ese punto, en este caso, tendremos como resultado cuatro números reales aleatorios, ya que se han obtenido a través de un vector también aleatorio [23].

Como estos valores obtenidos provienen de un producto escalar, si el punto que queremos evaluar se encuentra justo en el vértice de la casilla, su valor será cero, ya que el vector dirección calculado también tendrá valor cero. Por la misma razón, podemos determinar que el número obtenido será negativo si ambos vectores, dirección y gradiente, apuntaban en direcciones opuestas [23].

3. Interpolación: en este último paso, queremos dar continuidad en el espacio a los valores discretos que hemos obtenido en el paso anterior. Se pueden usar distintos tipos de interpolación, como la lineal, trigonométrica, de Lagrange o con splines [24]. La interpolación lineal consiste en hallar la recta que pasa por los puntos que conocemos, de manera que podamos dar un valor aproximado de aquellos que no. Ésta consume pocos recursos computacionales, pero los resultados no son del todo

"naturales" [25], por lo que es mejor usar otros métodos de interpolación o incluso alguna función que suavice el resultado. Ken Perlin originalmente utilizaba para la interpolación en espacios 3D, la función:

$$f(t)=3t^2-2t^3$$

El problema con esta función se ocasiona con su segunda derivada:

$$f''(t)=6-12t$$

Esta segunda función no presenta valor cero en los puntos t=0 y t=1 [26], que son los valores extremos que se obtienen con la aplicación del Perlin Noise. Esto ocasiona que se produzcan discontinuidades a la hora de aplicarse a caras de celdas cúbicas adyacentes, ya que las uniones de estas caras pueden quedar marcadas por esas irregularidades. Es por ello, que más tarde pasó a usar en su lugar el polinomio de quinto grado:

$$f(t)=6t^5-15t^4+10t^3$$

Esta nueva función, presenta valor cero en las segundas derivadas de estos puntos extremos, luego a la hora de usar algunas técnicas de tratamiento de gráficos, los resultados obtenidos son mejores [27].

Para obtener mejores resultados, se pueden generar distintas capas con este algoritmo y combinarlas para formar una sola imagen. A cada una estas capas se las llama octavas, y tienen diferente frecuencia y amplitud entre ellas. En nuestro caso de dos dimensiones, la frecuencia equivaldría al número de valores aleatorios que se calculan para el mapa, y la amplitud, la cantidad de representaciones que tienen eso valores.

2.6.2 Variaciones y usos

Este algoritmo puede ser utilizado para generar valores en múltiples dimensiones. En el apartado anterior nos hemos basado en el ejemplo de las dos dimensiones, que puede ser utilizado como ya hemos mencionado, para crear texturas para efectos especiales u otros elementos. En el caso de una dimensión, se puede utilizar para simular letra escrita a mano, aportándonos las variaciones de dirección debe tener el trazo [25]. Y para tres dimensiones, tenemos por ejemplo la creación de escenarios.

Para aportar detalle a un tronco de un árbol seguramente no se quieran los mismos resultados que a la hora de crear nubes. En cualquiera de estas dimensiones, también se pueden asignar distintos valores para cada una de las octavas que utilizamos para conseguir distintas texturas que podremos utilizar para diferentes cosas.

Un uso más concreto que pueden tener, son los "Perlin Worms" o gusanos de perlin. Estos suelen ser utilizados para generar cuevas, ya que los segmentos que crea siempre son conexos entre sí. Se basa en el algoritmo de Perlin para determinar la dirección en

la que avanzará el gusano de manera continua y aleatoria [28]. Podemos observar un ejemplo de estos “gusanos” en Figura 6.

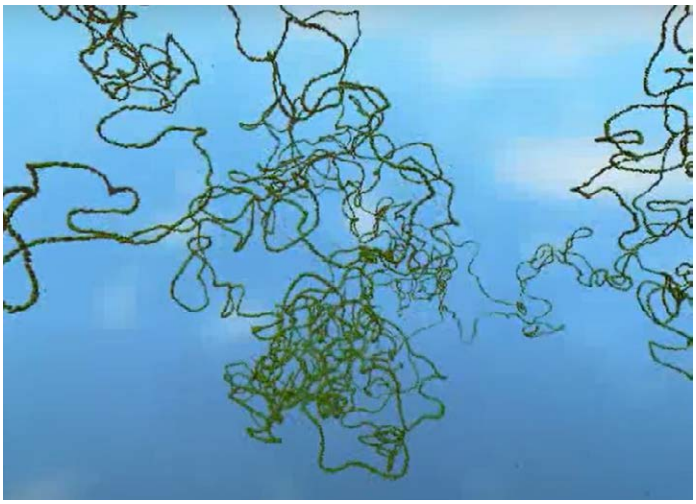


Figura 6. Ejemplo de gusanos de Perlin

En 2001, el mismo Ken Perlin, sacó a la luz Simplex Noise. Este algoritmo reduce la complejidad del algoritmo original y por tanto reduce sus tiempos de ejecución. También permite usarlo en dimensiones mayores con un menor coste computacional y es más sencillo de implementar a través de hardware. Todo esto es fundamentalmente gracias al cambio de figura a la hora de asignar gradientes, ya que la cuadrícula que usa deja de ser de cuadrados para pasar a ser de la forma más simple y compacta que mejor se adapte en cada dimensión, en el caso de las dos dimensiones es el triángulo. Esto nos permite tener que calcular una menor cantidad de ellos por cada punto que nos interese ya que hay menos puntos de los que depende [27]. También se debe a la forma de hallar la continuidad, que en vez de a través de una interpolación, realiza un sumatorio ponderando las influencias de cada uno de los vértices cercanos al punto a evaluar, lo que reduce notablemente el coste computacional. Aun así, este algoritmo es menos usado que el original, posiblemente debido a la dificultad que supone comprender su funcionamiento y que esta versión del algoritmo se encuentra actualmente patentada [29].

2.7 Mine Bombers

En este Trabajo de Fin de Grado se va a intentar desarrollar una versión simplificada del juego Mine Bombers para aplicación de nuestro algoritmo de generación procedural.

El Mine Bombers es un juego lanzado en 1995 por la desarrolladora finlandesa Skitso Productions para equipos DOS y MS-DOS. En 2003, la empresa decidió añadirlo a *freeware* para su distribución gratuita [30].

Está ambientado en una mina, en la que cada jugador controla un minero que se encuentra atrapado en ella tiene que abrirse camino. Tiene dos modos de juego: un jugador y multijugador.

En el modo para un jugador o “Single player”, el usuario tiene que recorrer el mapa intentado llegar hasta la puerta que le conducirá al siguiente nivel. Hay quince mapas distintos que tiene que explorar, repletos de diferentes tipos de monstruos que intentarán acabar con él. El jugador tiene que ir descubriendo el camino que seguir hasta alcanzar la puerta, y mientras los recorre podrá ir recolectando dinero, vidas y diferentes elementos del equipo que le ayuden a explorar y acabar con los monstruos.

En el modo multijugador, se puede jugar hasta cuatro personas. El objetivo del juego puede determinarse antes de comenzar la partida, de manera que se proclame vencedor el que más dinero o tesoros haya recogido por el campo, o el que más veces haya eliminado a los otros jugadores. Cada jugador comienza en una esquina del mapa y va avanzado por el terreno para ir recolectando dinero, que le servirá para comprar más armas. Cuando un jugador acaba con otro, también recibe un botín. La partida acaba cuando se hayan recogido todos los tesoros o solo quede como mucho un jugador en pie.

Figura 7. Pantalla de título Mine Bombers

El juego permite configurar muchos aspectos, como el número de dinero y tesoros que se reparten, el número de rondas que compondrá la partida, tiempo límite de cada ronda, número de jugadores, y otros aspectos de la propia partida, como la cantidad total de daño que hacen las armas, si se permite la venta de armas para conseguir más dinero o si el mapa se jugará a oscuras, limitando tu visión. También se permite cambiar las teclas para el manejo de cada jugador.

Antes de comenzar cada partida, nos aparece una pantalla de compra de armas y herramientas. Cada jugador puede decir que dinero gastar del que tiene recolectado y comprar el armamento que desee para usarlo en esa partida. En caso de que se

arrepienta de realizar alguna compra, puede volver a vender lo que ha comprado, pero por una cantidad inferior de dinero.

El juego cuenta con unos cuantos mapas creados por los desarrolladores para que jueguen los jugadores, e incluso llegó a haber un editor de mapas para que el usuario pudiese crearlos y compartirlos [31].

Los jugadores pueden decidir ponerse apodos, y el juego guarda puntuaciones y estadísticas de ellos que pueden consultar. Si juegas en el modo de un jugador, si obtienes una puntuación lo suficientemente alta, tu nombre puede llegar a aparecer en la lista de mejores puntuaciones que guarda el juego.

3 DESARROLLO

A continuación, se expondrán las herramientas utilizadas en el desarrollo de este proyecto, así como los puntos y elementos más importantes que lo componen y las principales decisiones tomadas relativas a su funcionamiento.

3.1 Herramientas utilizadas

En esta sección se describirán brevemente la funcionalidad y características de las principales herramientas que han sido utilizadas para el desarrollo de este Trabajo de Fin de Grado.

3.1.1 Unity 3D

Unity es un motor de videojuegos desarrollado por Unity Technologies. Un motor de videojuegos nos permite crear, diseñar y determinar el funcionamiento de un videojuego. Está compuesto por dos motores distintos: el motor gráfico, que se encarga del renderizado y cualquier otro aspecto que tenga que ver con los gráficos a través de información visual y espacial; y el motor físico, que simula las leyes físicas existentes a través de múltiples cálculos que buscan dar el mayor realismo posible.

Este motor de videojuegos, nos permite realizar desarrollos de videojuegos tanto 2D como 3D, además de otras funciones como animaciones. También se puede usar en combinación con muchos otros programas, como Blender o Maya para el modelado.

Unity está disponible para Windows, MacOS y Linux, y permite realizar desarrollos para un gran número de plataformas, como dispositivos móviles, PCs con distintos sistemas operativos, Smart TVs, consolas como la PlayStation 4, la Nintendo Switch o La Xbox One, y dispositivos de realidad extendida [32].

Su primera versión fue lanzada en una Conferencia Mundial de Desarrolladores de Apple en 2005, y desde entonces han lanzado múltiples versiones que han ido añadiendo multitud de características.

Este motor de videojuegos tiene distintos planes para que puedas elegir el que mejor se adapte a tus necesidades, desde un gratuito para uso personal, hasta uno de pago de dedicado a grandes empresas, pasando por otros para empresas más pequeñas o la versión para estudiantes.

En este TFG, se ha utilizado la versión Unity 2019.2.12f1 con el plan para estudiantes para el desarrollo del videojuego 2D y la aplicación del algoritmo de generación

procedural desarrollado. Los principales componentes o ventanas que se han utilizado y se consideran más relevantes son:

- **Scene**: es la ventana interactiva existente con el mundo que estás creado. Se utiliza para posicionar todos aquellos elementos que queramos en la escena [33]. Como nuestro videojuego se basa en un escenario en dos dimensiones, se ha trabajo con esta ventana configurada para mostrarnos el escenario desde el eje Z, de manera que nuestra visión se ve limitada a dos dimensiones, las determinadas por el eje X y eje Y.
- **Game**: esta ventana te muestra el resultado final de tu creación visto desde la cámaras que posiciones en escena, es decir, lo que vera el jugador al jugar el juego [33]. Esta ventana te ofrece múltiples opciones que afectan a la visualización del juego, como determinar el tamaño de pantalla o elegir si reproducir el sonido del juego. En esta ventana se han probado toda funcionalidad y cambios realizados durante el desarrollo.
- **Hierarchy**: muestra todos los objetos que se encuentran presentes en la escena, luego según lo que ocurra durante la ejecución su contenido puede variar, apareciendo nuevos y desapareciendo otros [33]. También muestra las relaciones que existan entre objetos padre e hijo, esto son agrupaciones de objetos que se realizan buscando la dependencia del hijo respeto algunas características del padre, como puede ser su posición.
- **Inspector**: muestra todas las propiedades y componentes de cualquier objeto o recurso seleccionado. Permite añadir nuevos componentes que añadan mayor funcionalidad al objeto, así como editar sus características. En caso de que el objeto tenga un script, permite ver y modificar el valor de variables públicas, salvo que se indique lo contrario.
- **Project**: en esta ventana se muestran todos los recursos que tienes en tu proyecto. En la parte izquierda se muestra la estructura que presentan todas las carpetas de tu proyecto, y en la derecha el contenido de la carpeta en la que te encuentres. Cuando pulsas un recurso que se encuentre en escena, esta ventana te mostrará automáticamente su ubicación, luego tienes la opción de bloquear la posición en la que te encuentres para que eso no pase. También se puede cambiar el tamaño en el que te muestra los elementos e incluso tener una lista de favoritos para poner acceder rápidamente.
- **Console**: muestra los mensajes informativos, de advertencias o errores que genera Unity, así como aquellos que hayas indicado en código que se muestren. Te permite aplicar filtros para mostrar solo uno o varios tipos de mensajes.
- **Tile Palette**: nos permite el manejo de las tiles, un recurso de que hablaremos más adelante. Es parecida a una herramienta simple de dibujo que nos permite definir un conjunto de estas baldosas que queremos utilizar, para colocarlas en escena, borrarlas o moverlas.
- **Animation**: permite crear y modificar animaciones directamente en Unity para utilizarlas en el juego [33]. Se ha utilizado para crear todas las

animaciones de los personajes de las que se hablará en una sección posterior. Te ofrece opciones de animación como la velocidad de reproducción de la misma o indicar si esta se reproducirá en bucle.

- **Animator**: se utiliza para crear y editar los controladores de las animaciones [34], que son los que determinan que animación se ejecuta en un objeto según unas condiciones determinadas. Permite determinar que parámetros determinarán esas condiciones, definir distintas capas de animación y determinar la relación que se quiere entre clips de animaciones.

3.1.2 Visual Studio

Visual Studio es un IDE (Integrated Development Environment) desarrollado por Microsoft que facilita el desarrollo de software. En este TFG ha sido utilizado como entorno de desarrollo para los script que dotan de funcionalidad al juego, ya que es el entorno de programación predeterminado por Unity y resulta una opción muy completa.

Nos permite desarrollar soluciones o proyectos para múltiples destinos: Windows, móviles, aplicaciones web o incluso bases de datos de SQL Server o Azure SQL. Utiliza la plataforma .NET que le permite desarrollar código sin compilarlo a lenguaje máquina hasta el momento de ejecutarlo, luego el código desarrollado puede ser independiente de la plataforma, pudiendo ser usado desde cualquiera compatible con .NET. Con este entorno de desarrollo podemos editar código, depurarlo, compilarlo o desarrollar conjuntos de pruebas [35]. Permite una fácil colaboración en caso de que haya varios desarrolladores, permitiendo además usar un control de versiones.

Acepta programar en una gran cantidad de lenguajes, como C++, C# (el cuál ha sido utilizado para el desarrollo de este proyecto), Java, Visual Basic .NET o Python.

Puede añadirse una gran cantidad de funcionalidad adicional a la que viene por defecto, como puede ser herramientas para el diseño del sistema, que te permiten crear distintos diagramas e incluso crear las clases partiendo de ellos.

Actualmente existen tres ediciones distintas para que elijas la que mejor se adapte a lo que necesites. La edición Community es gratuita y tiene instalación modular por lo que puedes instalarte solo aquellas funcionalidades que vayas a usar, mientras que Professional y Enterprise son de pago, pero ofrecen más características, como un mejor depurado del código.

3.1.3 Adobe Photoshop

Adobe Photoshop es uno de los programas de edición de imagen más extendido del mundo. Fue desarrollado por Adobe Inc en 1990 [36].

Cuenta con una gran variedad de herramientas para el tratamiento de imágenes, lo que hace de él un programa realmente completo, usado no solamente por fotógrafos profesionales, sino también por muchos tipos de artistas y un público más general.

La gran mayoría de los gráficos de este proyecto han sido creados, entera o parcialmente, con la ayuda de esta herramienta de edición.

3.1.4 Github

Github es una plataforma dirigida al desarrollo colaborativo, donde se almacenan proyectos para que sean accesibles en todo momento por cualquier persona autorizada. En 2018 fue comprada por Microsoft.

Utiliza el sistema de control de versiones Git, lo que nos aporta una gran variedad de herramientas, como pueden ser el flujo de trabajo en ramas, que permite realizar desarrollos simultáneos sobre un mismo proyecto, o incluso recuperar versiones anteriores de nuestro código en caso de que las necesitemos. Entre otras muchas cosas, cuenta con herramientas de revisión del código o de seguimiento de problemas que puedan surgir en el proyecto por parte de cualquiera de los miembros del equipo.

Esta plataforma es comúnmente usada para el desarrollo de software libre, ya que permite la contribución de muchas personas en un solo proyecto [37].

Para este proyecto sea ha usado Git BASH que permite la gestión de los repositorios desde una ventana de comando. Se ha elegido esta versión debido a la rapidez con la que se pueden realizar las operaciones más básicas, como comprobar el estado de tu rama de trabajo respecto a la del repositorio de referencia, descargar a tu ordenador el contenido del repositorio, añadir para subir al repositorio los archivos que desees, y finalmente incluir los cambios que hayas realizado al repositorio para que todos puedan acceder ellos.

Se ha realizado una acción de *commit* o confirmación de cambios al repositorio por cada cambio importante o añadido en la funcionalidad al juego, indicando cuál a ha sido éste, de manera que si en algún momento hubiese que acceder a alguna de aquellas versiones, se pueda decidir a cuál de ellas interesa acceder.

3.2 Desarrollo del juego

En este apartado, vamos a desarrollar los puntos más importantes para el desarrollo de este proyecto, explicando las distintas mecánicas de funcionamiento que tienen algunos elementos y como encajan finalmente todos ellos para permitirnos jugar el juego que queremos ofrecer.

Éste muestra un escenario generado de manera procedural ambientado en una mina, donde se encontrarán dos personajes que deben ir recorriendo el terreno recogiendo tesoros. A la hora de avanzar por él, los jugadores tienen la opción de picar terreno o de disparar, tanto al terreno como al otro jugador.

3.2.1 Personajes

Se trata de un juego para dos jugadores, luego habrá dos personajes. Como no se pretende hacer un uso comercial de este juego, se han seleccionado como gráficos para su representación, unos *sprites*[2] ya existentes pertenecientes a un juego de la saga "Pokemon". En caso de que algún día se quisiese comercializar el juego, estos diseños deberán verse reemplazados por otros distintos. Para el jugador uno, se han mantenido las imágenes originales, y para el segundo jugador, se han cambiado el color base rojo por uno amarillo, para facilitar la distinción entre ellos.

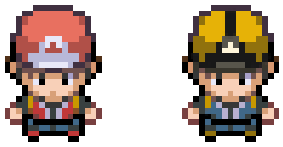


Figura 8. Personajes que representan a Player 1 y Player 2

El "jugador 1" utilizará para desplazarse las teclas de las flechas del teclado y realizará un disparo presionando la tecla *Ctrl* derecha. Su campo de juego será representado en la mitad derecha de la pantalla.

Por otro lado, el "jugador 2" utilizará para deslazarse el conjunto de las teclas "ASDW" y la barra espaciadora para disparar. Su visión del campo de juego se encontrará en la parte izquierda de la pantalla.

Esta división de la pantalla, se ha realizado colocando dos cámaras, una que siga a cada jugador y especificando en sus propiedades en qué parte de la pantalla debe ser representada su imagen.

A excepción de las teclas que utilizan cada uno, el funcionamiento de ambos personajes es el mismo. A continuación mostramos en la Figura 9 la jerarquía interna de los componentes de un personaje.

[2] Imagen de mapa de bits.

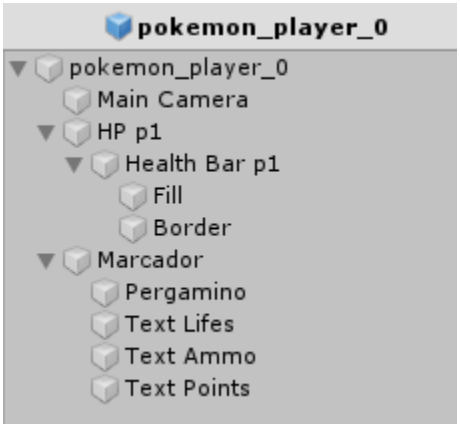


Figura 9. Jerarquía interna del prefab del Player 1

Como podemos ver, cada personaje tiene su propia cámara que le seguirá durante su desplazamiento por el mapa de la partida. Justo debajo, el elemento "HP p1" es un *canvas*[3] que contiene la barra de salud del personaje que aparecerá sobre él en todo momento. Y por último tenemos otro *canvas* "Marcador" que contiene los textos que se muestran durante la partida con las vidas restantes del jugador, la munición que le queda y los puntos que lleva acumulados. Para mejorar su apariencia, se le ha establecido una imagen de un pergamino de fondo, al que se le ha reducido la opacidad para que permita ver el terreno que hay tras él. Este cartel a parecerá en la esquina superior izquierda en la mitad de la pantalla correspondiente a ese jugador.

El primer elemento que vemos y que incluye a todos los demás es el *gameObject*[4] del propio personaje. Este contiene los distintos componentes necesarios para que realice su función correctamente. Entre ellos nos encontramos un *rigidbody*[5] para poder aplicar físicas sobre él y un *box collider*[6] *2D* que nos permitirá detectar las colisiones del personaje con su entorno e identificar con qué elementos se han producido de manera que pueda actuar en consecuencia con ello. Además contendrá el *script* "Player" del que hablaremos a continuación.

Lógica de los personajes

Para llevar a cabo la lógica de los personajes, se ha optado por dividir el código en dos clases, un "PlayerController" y un "Pawn". El primero (en nuestro caso se llama "Player") recoge los input producidos por el jugador y le dice al segundo ("PlayerMovement"), que acción debe realizar el personaje [38].

Ya que los dos personajes realizarán las mismas acciones, se quiere reutilizar el mismo código en ambos. Para ello, en el controlador se ha utilizado una enumeración de valores

[3] Área donde todos los elementos de interfaz de usuario deben estar [42].
[4] Son los objetos básicos en Unity. En sí mismos no son útiles, pero funcionan de contenedores para los componentes que implementan toda la funcionalidad [43]
[5] Componente que permite al objeto aplicarle físicas [44].
[6] Define la forma del objeto respecto al tratamiento de colisiones físicas [45].

(*enum*) con dos opciones, “Red” y “Yellow” una para cada jugador. De esta manera dependiendo de qué valor tenga el personaje que esté ejecutando la acción, se atienda a unos valores u otros en algunas variables. Como por ejemplo a qué ejes de movimiento se debe acceder para desplazar al personaje, o a la hora de almacenar las puntuaciones, guardarlas en la posición correcta.

El código que correspondería a “PlayerMovement”, se encarga de hacer moverse al personaje y de que ejecute cualquier acción que le diga el controlador. Modificará las variables correspondientes al personaje, así como gestionar las animaciones que se reproducen en él. Cada una de las funciones que contiene esta clase se encarga de una acción, como mover al personaje, dispara, picar o para de picar.

Animaciones

A la hora de dar dinamismo a los personajes, se ha utilizado el sistema de animaciones que tiene Unity. Éste nos permite crear pequeñas animaciones mediante la secuenciación de distintas imágenes, y seleccionar cuando reproducir estas animaciones según el valor de distintas variables.

Con la herramienta “Animation”, se crean los distintos clips de animación que vamos a utilizar. En este caso queremos doce animaciones distintas para cada personaje, que se pueden dividir en tres grupos: el personaje estático en el sitio, el personaje andando y el personaje picando. Cada uno de estos tres grupos está compuesto de cuatro animaciones, una para cada sentido de desplazamiento: izquierda, derecha, arriba y abajo. En todas ellas se seleccionan los distintos *sprites* que formarán los fotogramas de la animación, y se les asigna una velocidad de reproducción.

Una vez tenemos las animaciones listas, con la herramienta “Animator” pasamos a determinar cuándo se deben reproducir cada una de estas animaciones. Para determinar el estado del personaje y mostrar la animación adecuada a la acción que esté realizando, hemos definido cuatro variables, que dependiendo su valor nos llevarán de una animación a otra:

- **Vertical**: que nos indicará la cantidad de desplazamiento que se produce en el eje vertical por parte del jugador. Si su valor es negativo nos indica que se está desplazando hacia la abajo, y si es positivo, hacia la arriba.
- **Horizontal**: que nos indicará la cantidad de desplazamiento que se produce en el eje horizontal por parte del jugador. Si su valor es negativo nos indica que se está desplazando hacia la izquierda, y si es positivo, hacia la derecha.
- **Speed**: muestra si el personaje se está desplazando. Si su valor es igual a 0 significa que el personaje no está produciendo ningún desplazamiento y se encuentra estático en el sitio.
- **Pickaxe**: indica si el personaje está picando algún material. Su valor será “true” si el personaje está destruyendo algún bloque, y “false” en caso contrario.

En la Figura 10 podemos ver el árbol de ejecución de las animaciones. En él se encuentran la mayoría de las animaciones creadas para cada personaje representadas en cada uno de los recuadros. Cada uno de los bloques “Mov_XXX”, siendo “XXX” cualquiera de las cuatro posibilidades de dirección, es un subárbol, que contiene en su interior la animación del personaje estático y el personaje andando (véase Figura 11), y reproduce una u otra en función de la variable “Speed”. Las flechas simbolizan las transiciones que se producen entre las animaciones, condicionadas por las variables que acabamos de explicar más arriba. Para los subárboles de movimiento, el cambio de uno a otro se produce con el cambio de los valores de las variables horizontal y vertical, de manera que se reproduzca aquel que sea más acorde con el desplazamiento que esté realizando el personaje. De cada uno de estos bloques, sale otra transición a la animación del personaje picando. Esta pasará a reproducirse cuando la variable “Pickaxe” tenga valor *true*, y dejará de reproducirse, cuando pase a *false*, dando paso nuevamente a las animaciones de movimiento.

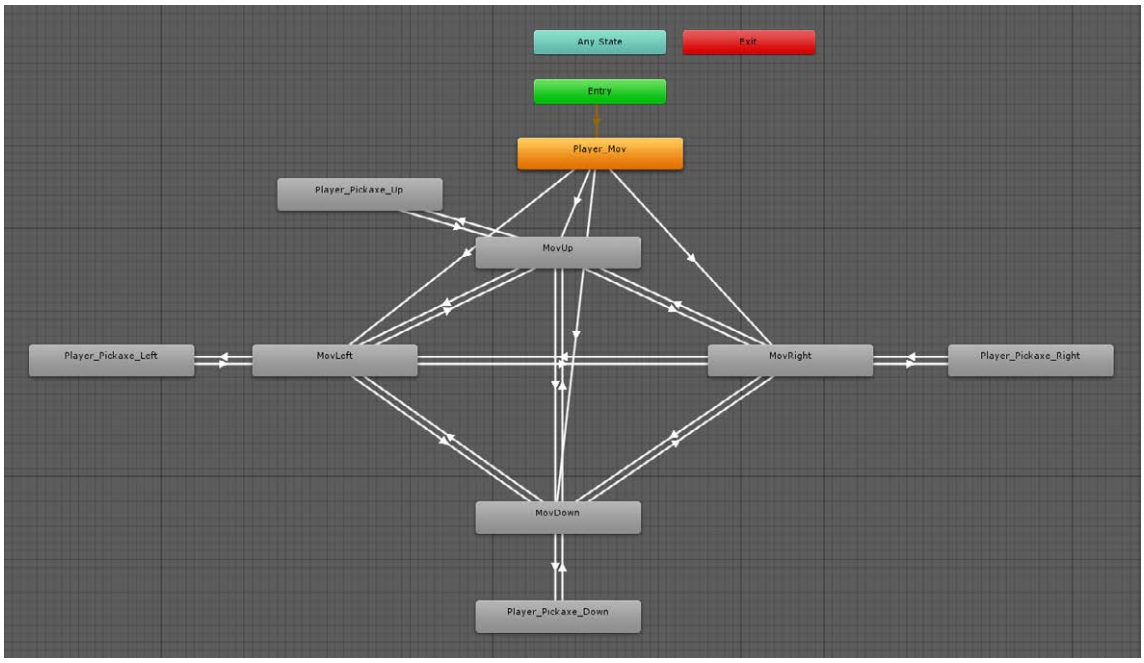


Figura 10. Árbol de animación de los personajes

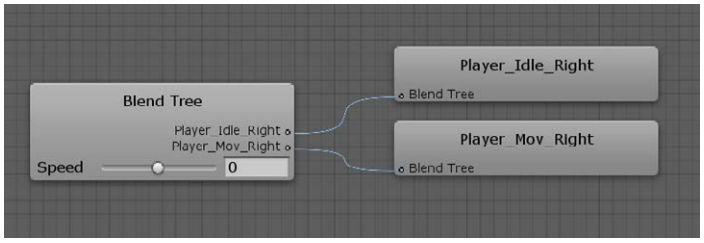


Figura 11. Subárbol de animación de MovRight

Finalmente para que el personaje represente estas animaciones, se le añade el componente “Animator” a su GameObject. A partir del script que se encarga del movimiento del personaje, se cambiarán los valores de las variables, que como acabamos de describir, controlan el paso de una animación a otra.

Vida de los personajes

En el juego se ha establecido que cada personaje tiene tres vidas, y por cada una de estas tres vidas, tiene doscientos puntos de salud. El valor de la cantidad de vidas restantes que tiene el jugador se almacena en el GameManager, del que ya hablaremos más detenidamente en secciones posteriores. La cantidad de puntos de salud que tiene el personaje en ese momento se almacena en la clase Player.

Esta clase, cada vez que recibe que el jugador ha sufrido algún daño, decrementa el valor de la variable y llama a su barra de salud para que represente esta nueva cantidad. Para su funcionamiento, la barra de salud tiene una clase a parte, "HealthBar", que simplemente se encarga de representar en la barra los valores que se le indican. Esta barra puede pasar por tres colores dependiendo de la salud restante del personaje: verde, amarillo y rojo.

Cada vez que el personaje resulte herido, además de disminuir su vida, se comprueba si el jugador ha sido abatido. Para ello, se consulta los puntos de vida restantes de jugador, y si son menores o iguales que cero, se destruye al personaje, y se resta una vida de las que tiene almacenadas en el GameManager. En caso de que le queden más vidas, el personaje reaparecerá en otro lugar del mapa, con sus puntos de vida regenerados. En caso contrario, el jugador ha perdido la partida, y ya no volverá a reaparecer su personaje.

Mecánica de disparo

Para hacer disminuir la vida del otro jugador más rápido, los personajes tienen la opción de disparar. Al igual que para desplazarse, cada uno tiene su propia tecla para efectuar el disparo. Cuando se presiona esa tecla, la clase Player comprueba si el personaje tiene munición para poder disparar. En caso de tenerla, decrementa la munición en uno, y llama a PlayerMovement para que realice la acción de disparar pasándole el proyectil que debe instanciar. Esto último podría permitirnos elegir distintos tipos de munición que disparar, aunque en nuestro caso solo se ha creado un tipo de proyectil.

La clase comprobará en qué dirección está mirando el personaje o en qué dirección se está desplazando en ambos ejes, x e y. Dependiendo de los resultados de cada eje, se imprimirá una fuerza al proyectil en una dirección distinta, además de modificar el punto desde el que se realiza el disparo. Finalmente para que el proyectil no se quede "contaminando" la escena, se destruirá el objeto pasados dos segundos o cuando haya impactado con algún objeto.

Recolección de tesoros

El objetivo del juego consiste en recoger tantos tesoros como puedas antes de quedarte sin vidas. Cuando una instancia de un tesoro entra en contacto con el *collider* del personaje, éste comprueba que se trata de un tesoro, lo recoge, y se suma a su puntuación global de la partida.

Una vez recogido el tesoro, desaparecerá del mapa y se anotará que se ha recogido un tesoro más a la cuenta total. Esto nos permitirá saber cuándo se han recogido todos los tesoros.

Mecánica de picar

Para poder avanzar por el terreno de juego, el jugador puede decidir destruir elementos de tierra y roca que le bloquean el camino, de los cuales se hablará más adelante. Para ello, aparte de disparar, tiene la opción de picar.

Cuando el *collider* del personaje entra en contacto con alguno de estos materiales "picables", se pasa a reproducir la animación del personaje picando, y se comienza a restar puntos de salud al elemento que esté intentado destruir. Cuando el jugador deja de estar en contacto con ese elemento o el elemento queda destruido, se vuelve a pasar a la animación del personaje andando o quieto sin picar.

Figura 12. Sprite de player 1 picando

3.2.2 Proyectil

También se ha creado un *prefab* para el elemento proyectil y cuenta además con su propia clase. En ella se especifica el daño que produce a un personaje o un bloque cuando impacta contra él.

A parte del daño que produce el impacto directo, a modo de simular los destrozos que produciría la explosión del proyectil, también se establece un daño menor que se aplicarán a los objetos cercanos al lugar del impacto. Cuando el proyectil impacta, aplica el daño al objeto contra el que ha impactado y posteriormente se hace uso de la función "Physics2D.OverlapCircle" para herir a los elementos circundantes. Esta función nos permite determinar un radio de un círculo y almacenar en una lista todos aquellos elementos que se encuentren dentro de ese círculo. Ahora solo debemos ir recorriendo

esa lista y producir el daño a cada uno de los objetos que se encuentren en ella. Mientras se recorre esta lista, se comprueba que el objeto que estamos hiriendo no es el mismo que con el que hemos golpeado inicialmente, de este modo evitamos herir dos veces al mismo objeto.

El proyectil también tiene su propia animación para representar la explosión. Esta comienza a reproducirse cuando se produce el impacto. Como una vez impactado el proyectil debe destruirse, se hace uso de una *corutina*[7] que nos permita esperar el tiempo que dure la animación de la explosión antes de destruir el objeto, de otro modo esta animación no llegaría a reproducirse, ya que el objeto que debe representarla estaría destruido.

Figura 13. Secuencia de animación de explosión del proyectil

3.2.3 Game Manager

El GameManager es un elemento muy común en el mundo del desarrollo de videojuegos. Se encarga de almacenar variables y procedimiento que son comunes para todo el juego y queremos que sean accesibles en cualquier momento. Para evitar que haya errores y duplicados, es fundamental que solo se permita tener una sola instancia del mismo que debe ir pasando de una escena a otra sin ser destruida.

Para este proyecto se ha optado por la utilización del patrón *Singleton*. Este es un patrón de diseño simple que nos permite asegurar que solo vaya a ver una instancia del GameManager que creemos. Hay distintas maneras, muy similares entre ellas, de implementar este patrón, en nuestro caso se ha creado como atributo privado una instancia del objeto y que sólo podrá será accesible a través un método público *get*. La instancia se crea con la propia carga del código del GameManager, de este modo siempre tendremos nuestra instancia lista para ser accedida. Además de esto, se hace que se cree el propio objeto del GameManager, almacenado en un prefab, en escena antes de que ésta comience su carga, y provocar que este objeto no sea destruido en el cambio de una escena a otra, de manera que sea el mismo objeto el que existe durante todo el transcurso de la partida. De esta manera ya tenemos nuestra clase GameManager configurada para ser consultada en cualquier momento.

Como ya hemos dicho, esta clase está pensada para almacenar datos y funciones que puedan ser utilizadas en todo momento, luego los datos que se han decidido declarar en nuestro caso son:

[7] Función que tiene la habilidad de pausar su ejecución y devolver el control a Unity para luego continuar donde lo dejó en el siguiente frame [46].

- Puntos recolectados por los jugadores: se trata de un array de una dimensión con dos posiciones, la primera del jugador 1 y la segunda de jugador 2. Ambas posiciones comienzan inicializadas a 0.
- Vidas restantes de cada jugador: también es un array de una dimensión con dos posiciones, la primera del jugador 1 y la segunda de jugador 2. Ambas posiciones comienzan inicializadas a 3.
- Figuras de los personajes: dos gameObjects correspondientes a los prefabs de los dos personajes.
- Figura de los tesoros: un gamebject correspondiente al prefab de los tesoros que deben recolectar los jugadores.
- Panel de Game Over: un canvas con la pantalla final que presenta las puntuaciones de los jugadores al acabar la partida.
- Capas del mapa: dos tilemaps correspondientes a Tilemap_Floor y Tilemap_Main que conforman el mapa.
- Vida de los obstáculos del mapa: un array de dos dimensiones de tipo float que almacena la vida restante de la tierra y roca que tiene que ir destruyendo el jugador.
- Ancho: valor de tipo entero que indica el ancho que tiene el mapa de juego en número de tiles.
- Alto: valor de tipo entero que indica el alto que tiene el mapa del juego en número de tiles.
- Numero de tesoros repartidos: valor de tipo entero que indica el total de tesoros que se han repartido por el mapa de juego.
- Numero de tesoros encontrados: valor de tipo entero que indica el número total de tesoros que han encontrado ambos jugadores.
- Controlador de musica: un audioControl para determinar que pista de música debe sonar en cada momento.
- Generator del terreno: para poder llamar a sus funciones y que se encargue de generar el mapa cuando corresponda.

Nuestro GameManager cumple con numerosas funciones, pero todas ellas se pueden agrupar en dos grandes categorias:

- Preparar el conjunto del terreno de juego: es decir, se encarga de que la creación del terreno se realice correctamente. Para ello, se encarga de establecer cuál es el ancho y alto que debe tener el mapa de juego. Cuando llega el momento también se encarga de hacer que el generador de terreno inicie la creación de las capas que conforman el mapa. Una vez esto está hecho, se encarga de repartir tantos tesoros como correspondan, dependiendo de las dimensiones del terreno, que nos los iremos encontrando según recorramos el mapa, y comprueba que están siendo colocados en posiciones adecuadas y accesibles para el jugador. Para considerar una posición como adecuada, esta no debe ser agua profunda o lava, ya que la primera zona no puede atravesarse y la segunda supondría al jugador sacrificar

puntos de salud; tampoco debe haber un jugador inicialmente en esa posición ni otro tesoro en las casillas adyacentes.

- Controlar el flujo de la partida: entre otras cosas, el GameManager constantemente está comprobando si se dan cualquiera de las condiciones para que se termine la partida, esto es cuando alguno de los dos jugadores se queda sin vidas o si se han recogido todos los tesoros que se han distribuido por el mapa. Si se da por concluida la partida, el GameManager se encarga de hacer aparecer la pantalla de Game Over con las puntuaciones. Por otro lado, cuando recibe que un jugador ha perdido todos sus puntos de salud, se encarga de restarle una vida a su cantidad total. Después, pasa a recolocar ese personaje en el terreno de juego. A la hora de recolocar un personaje calcula las coordenadas a través de dos números pseudoaleatorios calculados dentro del rango del tamaño del mapa y comprueba si esa posición es válida. Para que una posición de un personaje sea válida, debe de no haber instanciada en ella agua profunda o lava, ni roca ni tierra de la capa superior, además de no encontrarse ningún tesoro en esa casilla o las adyacentes. De no ser válida, calcula dos números nuevos y vuelve a probar.

Fuera de estas dos grandes funciones del GameManger, ésta también se encarga de realizar la carga de las escenas en el momento debido y de controlar el sonido del juego.

3.2.4 Implementación de la generación del escenario

Este punto se podría considerar el eje central del trabajo. Queremos generar un escenario procedural que sea compatible con los objetivos del juego. Para conseguirlo, se ha decidido usar el algoritmo Perlin Noise en combinación con el uso de *Rule Tiles*.

Rule Tiles

Un elemento tile es una casilla que se puede colocar tantas veces quieras a lo largo de un terreno para formar un escenario. Se deben colocar en un *tilemap*, que es el que se encarga de almacenar la información sobre la colocación de esas tiles, su renderizado y muchos otros componentes. Este tilemap, debe utilizarse junto con un elemento de tipo *Grid* o rejilla que permita posicionar en ella las tiles en la escena [39].

Los elementos de tipo Tile forman parte de un paquete de elementos extras (2DExtras) que se añaden a los ya existentes en Unity. En este paquete se incluyen distintos tipos de Tiles: animated tile, pipeline tile, random tile, rule override tile, rule tile, terrain tile, weighted random tile.

Para la creación del escenario hemos usando exclusivamente RuleTiles. Este tipo de tiles, nos permite definir una serie de reglas que deben cumplir las casillas adyacentes para determinar que representación de la tile queremos poner en un lugar concreto. Cada

rule tile está compuesta de una lista de tiles, cada una con un sprite asociado, y estas son las distintas formas que puede adoptar esa rule tile. Por cada una de estas opciones, se establecen una serie de condiciones que se deben dar para su colocación. En este proyecto se ha trabajado con una versión modificada de las ruletiles iniciales, que nos permiten definir los siguientes tipos de reglas o condiciones:

- Don't care: no se trata realmente de ninguna condición. Es la que se aplica por defecto si no marcas ninguna otra regla. No evalúa esa casilla a la hora determinar si esa tile va en esa posición.
- This: comprueba si en esa posición hay instanciada una tile del mismo tipo que la que estas intentado colocar. Se representa con una flecha verde
- Not this: comprueba que en esa posición no haya instanciada una tile perteneciente al mismo tipo que la que está intentando instanciar. Se representa con una cruz roja.
- Specific tile: comprueba que en esa posición haya instancia una tile de un tipo concreto. Se representa con una doble flecha gris.

Como tipo de tile entendemos a la propia rule tile y el conjunto de tiles que tiene como representaciones. Por ejemplo, tenemos la rule tile "Gound_Water". Ésta representa aquellas casillas en las que hay agua, y también aquellas en las que el agua entra en contacto con la tierra, por lo que distintos sprites, cada uno con su propia tile, formarán parte de esta rule tile, para adecuarse a la forma en la que rodee la tierra al agua. Dependiendo cual sea la situación una representación distinta de este tipo de casilla deberá aparecer.

En la Figura 14 podemos ver un ejemplo de un conjunto de reglas definidas para una tile concreta. Podemos ver que se trata de un gráfico que representa una superficie de agua rodeada de tierra que realiza un giro, parecido a un meandro de un río. La cuadricula que vemos al lado del sprite, es la representación gráfica de donde se aplicarán las reglas y que reglas son. El sprite que queremos colocar, estaría en la posición central de la cuadrícula. Como hemos dicho antes, las flechas verdes representan que en esa posición debe haber el mismo tipo de tile que la que queremos colocar. En este caso se trata de una tile "Ground_Water", luego tanto en la posición superior a ésta, como en la de la izquierda, debe haber instanciadas también otras ruletiles "Ground_Water". Las dobles flechas grises simbolizan la condición de que haya una tile específica. Ese tile que queremos encontrar se define en las líneas que observamos en medio y que viene precedidas por "MatchX", siendo X un número que indica la posición en la que se está estableciendo esa condición. En este caso, vemos que las tres flechas grises están exigiendo que haya rule tiles del tipo "Ground_Regular" en las posiciones: superior izquierda, en el lado derecho y en la que se encuentre justo debajo. En la Figura 15 podemos ver un ejemplo de su uso en la que se cumplen todas estas condiciones.

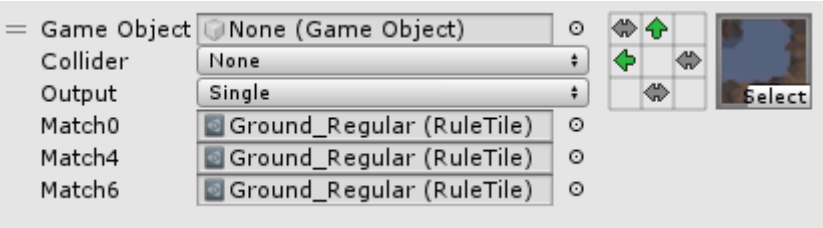


Figura 14. Ejemplo de configuración de Rule Tile Ground_Regular

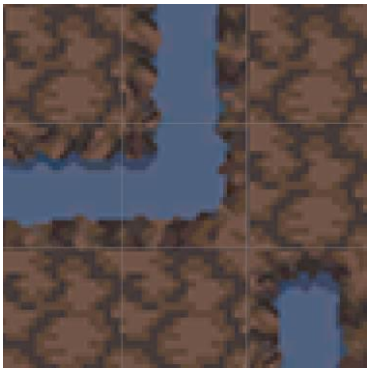


Figura 15. Ejemplo de uso de Rule Tile Ground_Regular

En la Figura 14 podemos ver otros campos que no se han mencionad aún. El primero es la posibilidad de adjudicar a esta tile concreta un GameObject que nosotros queramos, lo que nos permite dotarla de mayor funcionalidad. Justo debajo vemos que nos deja seleccionar si queremos o no establecer un *collider* a esta casilla, y en caso de quererlo de qué tipo, que se ajuste al *sprite* o a la casilla de la rejilla (Grid). A continuación, vemos otro desplegable que nos permite elegir el tipo de representación que tendrá la tile cuando se cumplan sus condiciones. Las opciones son Single, que es la que hemos usado en la mayoría de las tiles y que señala que solo tiene una representación posible; Random, que nos permite especificar varias representaciones posibles y la probabilidad de que aparezca una u otra, esta la hemos utilizado en alguna ocasión para aportar una mayor heterogeneidad al terreno; y por último, Animation, que nos permite reproducir una pequeña animación en el lugar [40].

Para el desarrollo de este juego, se han creado nueve ruletiles distintas y tres tilemaps en los cuales se repartirán estas rule tiles y juntos conformarán el escenario completo. Cada uno de estos tilesmaps contendrá un nivel de escenario: el más bajo de todos “Tile_Floor”, contendrá las tiles que formarán el paisaje base que se va descubriendo según se vayan destruyendo elementos; el segundo “Tilemap_Main”, es el que contiene los elementos que bloquean la vista del terreno base y que hay que destruir; y por ultimo “Tilemap_Frame” que contiene tiles que crean un marco en los límites del terreno para que el jugador no pueda salir de él.

Se quiere representar el entorno de una cueva, luego se han creado nueve tipos de ruletiles que representan cada uno un elemento distinto del paisaje y su unión con otros elementos. Para la capa “Tilemap_Floor” podemos distinguir:

- **Ground_Regular**: representa la parte del terreno en la que el suelo es tierra. Es el elemento base de esta capa con el que el resto de elementos se acoplan, ya que sirve de intermediario en las transiciones de uno a otro. Esta rule tile solo tiene una representación, ya que serán el resto de elementos los que contienen las uniones con este material.
- **Ground_Water**: representa las zonas del terreno que tienen agua y sus uniones con la tierra. Estas casillas permiten el desplazamiento del jugador sobre ellas pero reduciendo su velocidad para simular la dificultad de avanzar en el agua. Esta rule tile tiene cerca de cincuenta representaciones para cubrir todas las opciones que se pueden contemplar dada la disposición del escenario, es decir, con ocho tiles alrededor por tile.
- **DarkWater_Water**: representa la zona con agua profunda y su unión con las zonas de agua poco profunda. Es la única tile de esta capa que no presenta uniones con la tile Ground_Regular directamente. Esta tile no permite el paso del jugador a su interior, bloqueándole su avance en caso de que intente avanzar en su dirección.
- **Ground_Sand**: muestra las zonas con suelo arenoso y su unión con las zonas de tierra. No produce ningún efecto en el jugador, es un elemento decorativo.
- **Ground_Lava**: representa zonas de lava en el terreno y su unión con las zonas de tierra. Estas tiles permiten el avance del jugador sobre ellas, pero disminuyen sus puntos de vida durante el tiempo que transcurre en su superficie. Para poder ocasionar este efecto, llevan como componente un *gameObject* que contiene el script "TileDamage" que controlará quien entra en la zona delimitada por el *collider* contenido en este objeto.

Figura 16. Representación básica de los elementos de la capa Tilemap_Floor

En la capa "Tilemap_Main", podemos encontrar las siguientes rule tiles que pueden ser destruidas por el jugador tanto picando como disparando:

- **AboveEmpty**: esta tile contiene una imagen completamente transparente y representa una posición en la que no se encuentra ningún elemento de esta capa. Esta tile podría haberse omitido y simplemente no instanciar ninguna tile en estas posiciones, pero ha decidido implementarse para facilitar la configuración de las otras tiles de la misma capa.
- **AboveGround**: representa aquellas posiciones en la que hay tierra que el usuario puede picar para abrirse paso por el terrreno, y todas sus uniones con las tile AboveEmpty y AboveStone. Como en este caso son dos los elementos con lo que tiene que tratar la unión, esta tile es la que contiene un mayor número de reglas.

Para permitir ser picada y gestionar su destrucción, esta tile tiene asociada un gameObject con el script “TileHealth”.

- **AboveStone**: muestra las zonas del terreno compuestas por roca y sus uniones con AboveEmpty para marcar el fin de esas zonas. Este material también puede ser picado por el jugador para avanzar por donde desee, pero es más resistente que la tierra. Al igual que la tile AboveGround, esta tiene asociada un gameObject con el script “TileHealth” para para controlar cuando debe destruirse.

Este script “TileHealth” se encarga de gestionar la vida de las tiles. Cuando se instancia, la inicializa al valor que le corresponde dependiendo el tipo de material que sea y almacena su valor en el GameManager. Cuando una tile es atacada, se encarga de sustraer la vida correspondiente al ataque y actualizar también los valores que contiene el GameManager. Para representar el daño sufrido y la vida que tienen, se tienen dos *prefabs* distintos para cada uno de los dos tipos de tiles. Cuando la tile ha alcanzado los dos tercios de vida que originalmente tenía, se cambia en la posición en la que está la tile por la tile de vacío y se instancia sobre ella el primer *prefab* de destrucción (X-Desty-1, siendo X el elemento que representa); cuando ya solo le queda un tercio de vida, se destruye este *prefab* y se instancia otro que representa una mayor erosión (S-Desty-2); y cuando finalmente se ha quedado sin puntos de vida, se destruye este último tile y se deja en esa posición la instancia de la tile de vacío (AboveEmpty).

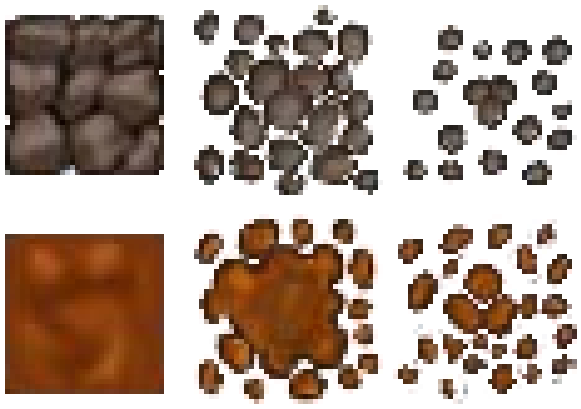


Figura 17. Representación de la destrucción de los elementos tierra y roca de la capa Tilemap_Above

Finalmente para la capa de “TileFrame”, tenemos una única rule tile que nos permite delimitar el terreno accesible para el jugador. Esta ruletile, “WallBrick” no puede ser destruida por el personaje y se colocará rodeando el escenario como si se tratase de un marco.

Dentro de cada rule tile se establece una prioridad a la hora de evaluar si una representación de esta tile puede ser colocada o no. Esta prioridad viene determinada por el orden en el que las tiles están colocadas en la lista de representaciones para cada rule tile. Esta característica se ha aprovechado para reducir el número de reglas que establecer y de representaciones que crear. En especial en las rule tiles de “AboveGround” y “AboveStone”, se han dejado solamente las reglas del tipo “This” esenciales, y de esta manera ambos elementos solo tienen en cuenta la existencia del

elemento vacío a su alrededor, y adoptarán la representación que corresponda sin tener en cuenta a que elementos se están uniendo después, y dejando que el límite del cambio de terreno, en caso de haberlo, sea marcado por los contornos del elemento roca.

Uso del Perlin Noise

Ya hemos explicado en anteriores secciones el funcionamiento del algoritmo de Perlin para conseguir ruido que mantiene la continuidad entre sus valores. Para su utilización en este proyecto se ha escogido utilizar la función que hace uso de este algoritmo en la librería "Mathf" del propio Unity.

Esta función calcula un plano con valores de ruido de Perlin. Recibe como parámetros dos valores decimales de tipo float, que son las coordenadas del punto del plano que va a consultar para devolver su valor de ruido.

Esta función aplica una función hash a los valores que se pasan como parámetros para consultar una tabla con gradientes precalculados, luego al pasar unas mismas coordenadas, la función te devolverá siempre el mismo valor. Para solventar este posible inconveniente y así evitar repeticiones, se hace uso de un número pseudaleatorio dentro de un rango muy amplio, de manera que el conjunto de puntos consultados cambie con cada ejecución de la función. De esta manera el mapa generado siempre será distinto.

Comencemos explicando la función en la que utilizamos directamente la función del ruido de Perlin. En esta función, se calcula un número pseudoaleatorio que utilizaremos como hemos explicado antes para variar la parcela del plano de la que obtenemos los valores del algoritmo de Perlin. Recibe como parámetro una matriz con las mismas dimensiones que el mapa de juego, y se encarga de almacenar en ella los valores que utilizaremos para crear el terreno.

Esta matriz podría considerarse un tipo de mapa de altura, un concepto que ya hemos explicado anteriormente. Pero en este caso, en vez de almacenar la altura que tendrá cada uno de los puntos, como nuestros terreno es únicamente de dos dimensiones, lo que almacena será el valor que nos indicará que elementos instanciar.

Como se ha contado anteriormente, hay tres tilemaps distintos para cada una de las capas de terreno. La capa base "Tilemap_Floor" utilizará dos capas de valores Perlin para generar el terreno y que resulte más realista. El primer mapa de valores que se inicializa, se usará para distribuir las zonas de lava, tierra y agua poco profunda; y el segundo mapa se utiliza para determinar donde se instancia agua profunda y arena. Recordemos que la función de Perlin nos da valores en el rango [0,1], luego según el valor que se obtiene en cada posición se instanciará una tile distinta según esta distribución:

- El valor obtenido en el primer mapa está comprendido entre [1, 0.7): se instanciará lava en esa casilla.

- El valor obtenido en el primer mapa está comprendido entre [0.7, 0.4]: se instanciará el elemento tierra. Dentro de este rango, si el valor está a su vez entre los valores (0.5 , 0.6), se consultará el valor que se obtiene en la misma posición en el segundo mapa, y si resulta ser mayor de 0.5, se instancia el elemento arena.
- El valor obtenido en el primer mapa está comprendido entre [0, 0.4): se instancia agua. Dentro de esta sección, si el valor es menor que 0.2, se consultará el valor que se obtiene en el segundo mapa, y si éste es menor a 0.5, se instancia agua profunda.

Para la capa de "Tilemap_Main", se utiliza una sola capa de ruido. Y según sus valores la distribución resultante sería:

- El valor obtenido se encuentra en [0, 0.5): se instancia la tile que representa el vacío. De esta manera garantizamos que haya suficiente terreno que recorrer sin necesidad de que el personaje tenga que destruir obstáculos y el juego resulte más dinámico.
- El valor obtenido se encuentra en (0.7, 0.8] o [0.9, 1]: se instancia el elemento roca. Este es el elemento menos común de esta capa, ya que es el elemento más duro y que más dificulta el avance del jugador. Además para que el mapa sea visualmente más atractivo, se busca que esté rodeado del elemento tierra de esta misma capa. Además con el hueco que hemos creado en el rango, buscamos que también se encuentre tierra en el interior de grandes aglomeraciones de roca.
- El valor obtenido se encuentra entre [0.5, 0.7] o (0.8, 0.9): se instancia el elemento tierra. Este elemento también servirá de obstáculo para el jugador.

Por último, la capa "Tilemap_Frame" no utiliza el algoritmo de Perlin para formarse, simplemente recorre las posiciones límite del terreno de juego e instancia elementos formando el muro.

Los límites que se han establecido para cada rango han sido determinados tras varias pruebas y eligiendo aquellos que otorgasen al mapa un mejor aspecto. Finalmente, nuestra llamada a la función de ruido es:

```
mapArray[x, y] = Mathf.PerlinNoise(x * scale + random, y * scale + random);
```

Como vemos para cada componente de la coordenada usamos tres valores. El primero, son los correspondientes x e y, que son las posiciones de la matriz donde estamos almacenando los valores, de esta manera conseguimos que los valores sean todos continuos dentro del mapa de ruido consultado. El segundo, es un valor decimal que se ha establecido para determinar la diferencia entre los valores obtenidos en consultas para posiciones continuas, algo parecido a la escala que tiene los mapas. Este valor multiplica las coordenadas que se están pasando para ampliar o reducir el abarque del campo consultado, de esta manera, los valores que se obtienen son menos o más similares, respectivamente. El último valor, es el número aleatorio del que hemos hablado antes que determina la zona que vamos a consultar para evitar duplicidad.

En la Figura 18 podemos ver tres mapas calculados con el mismo número aleatorio pero al que se ha variado la escala de consulta. El mapa del centro es el que tiene la escala elegida 0.13, mientras que al mapa de la izquierda se le ha reducido su valor hasta 0.6, poco más de la mitad. Como vemos en este mapa las zonas generadas con un mismo elementos son mucho mayores, esto se debe a que los números consultados tenia valores mucho más similares, es como si hubiésemos hecho "zoom" en una zona concreta del mapa anterior. El mapa de la derecha, ha visto prácticamente duplicada su escala, con un valor de 0.3. Aquí vemos como las agrupaciones son mucho menores, ya que los valores consultados están más alejado unos de otros, luego el resultado es como si hubiésemos alejado la vista respecto al mapa central.

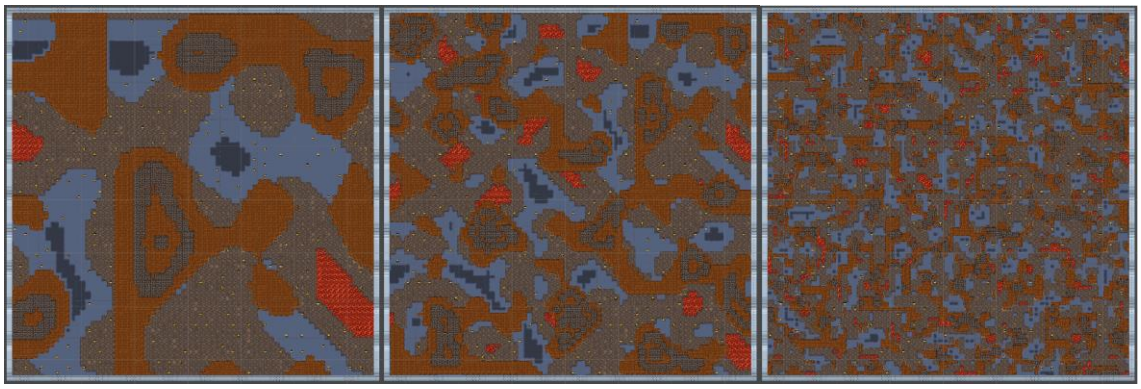


Figura 18. Comparativa de terrenos según la escala escogida

3.2.5 Sonido

La banda sonora del juego ha sido creada por Alejandro Arbelo Martín específicamente para este título, y se compone de tres temas diferentes presentes cada uno en una pantalla distinta: pantalla inicial, escena de juego y pantalla con las puntuaciones. Cada uno de estos, está dividido a su vez en dos pistas, una inicial que introduce la canción, y un bucle que se repite indefinidamente.

Se ha agregado al GameManager un componente *AudioSource* que se encarga de la reproducción de las pistas de audio que contiene. El manejo de este componente, se realiza a través de un script que también contenido en el GameManager. Esta clase tiene un atributo para almacenar cada una de las seis pistas de audio y dos métodos públicos, "playCave" y "playScoreMusic", que podrán ser llamados desde el GameManager a la hora de cargar las pantallas de juego y de puntuación respectivamente. Estas funciones llaman a una función privada indicándole por parámetro cuales son las pistas que debe reproducir. Esta otra función, se encarga de pausar la canción que esté sonando en ese momento, y de comenzar a reproducir la canción inicial correspondiente. Haciendo uso de una corrutina, espera a que esta pista acabe para comenzar a reproducir la segunda parte del tema en bucle.

3.2.6 Ajustes

Se ha creado una pantalla de ajustes que permita al usuario determinar el tamaño del mapa de juego que se va a crear (véase Figura 22). Este panel está compuesto por un *canvas* que se hace visible al pulsar el botón "Settings" que tenemos en la pantalla inicial antes de comenzar a jugar. En esta nueva pantalla podemos introducir el número de tiles de ancho y alto que queramos que tenga el mapa. A la hora de introducir los números, se comprueba que estos sean mayores o iguales a 30, que es la medida que se ha considerado mínima para que se genere un mapa cómodo de juego. De no ser así, al pulsar el botón "Save" que se encuentra debajo, un texto amarillo nos lo indicará en pantalla. Si cumple con estos requisitos, la función que llama el botón se encargará de almacenar los valores en el GameManager y de devolvernos a la pantalla inicial para que podemos comenzar el juego. También tenemos la opción de salir sin establecer el tamaño. De no haber establecido un tamaño antes de comenzar el juego, se creará un mapa con el tamaño por defecto, que se ha establecido que sea 70.

El número de tesoros que se instanciarán es (ancho * alto)/15. Como ya se ha contado antes, los tesoros no pueden colocarse adyacentes unos a otros, lo que quiere decir que no puede haber más de un tesoro por cada nueve casillas. Se ha decidido que haya un tesoro por cada quince, de manera que no ocupen un porcentaje de terreno demasiado grande.

3.3 Diagrama de clases

Ahora que ya se conoce la utilidad de cada uno de los elementos del juego, pasemos a determinar qué clases componen a cada elemento, y cuáles son las principales relaciones entre ellos.

A continuación, en la Figura 19, se muestra un diagrama con las clases simplificadas con algunas de estas relaciones existentes. De esta manera se pueden ver de manera más clara cuál es el propósito de la interacción de los elementos principales del juego.

Las clases señalizadas en color naranja son la que representan al elemento jugador, la amarilla es el proyectil que dispara el jugador, las verdes son las que encontramos en las baldosas de terreno, y las azules se encargan de aspectos de control del juego.

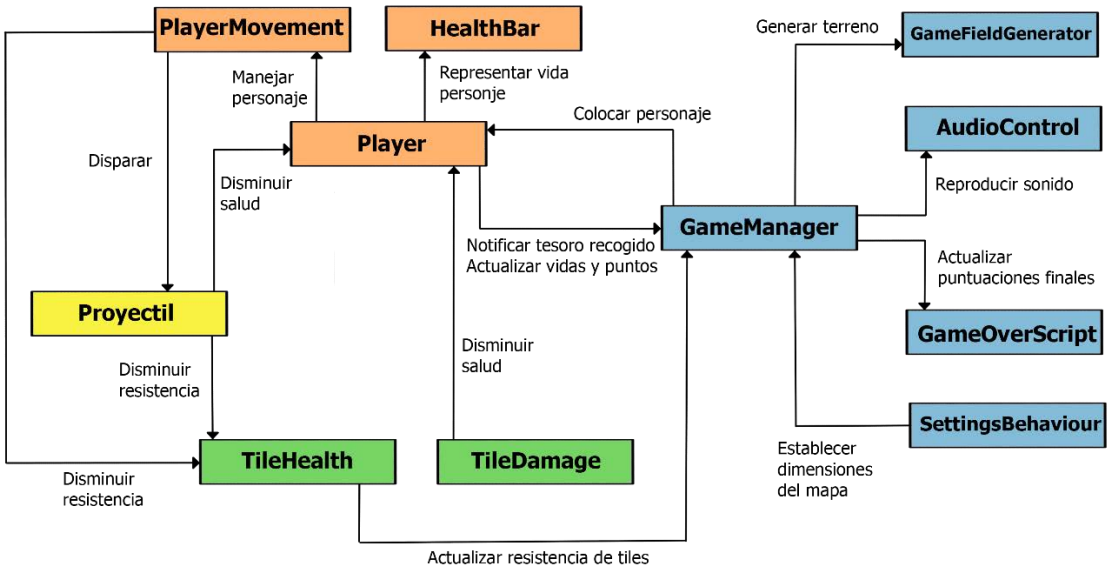


Figura 19. Diagrama con semántica de relaciones entre clases

El motor de videojuegos que se ha escogido, Unity, permite utilizar C# como lenguaje de programación a la hora de escribir el código que vayamos a utilizar en el juego. Para poder comunicarse con la mayoría de eventos y objetos en la escena, la clase debe heredar de Monobehaviour. Esta clase ofrece funciones como Start, Update y Awake, que se pueden encontrar en varias de las clases del diagrama de la Figura 20. La función Awake se ejecutará justo antes de que se comience a reproducir la escena en la que se encuentra el objeto con este método. La clase Start, será llamada con la llegada del primer frame[8] de ejecución de la escena, y no volverá a ejecutarse. Y la función Update, está constantemente ejecutándose con cada frame del videojuego.

El funcionamiento del objeto del personaje se basa en tres clases: Player, PlayerMovement y HealthBar:

- **Player**: esta clase realiza la función de *controller*, que registra las entradas producidas por el usuario y le dice a la clase PlayerMovement que acciones debe realizar el personaje. Contiene una instancia de esta clase como atributo para poder hacer uso de sus métodos públicos.
- **PlayerMovement**: esta clase realiza la función de *pawn*, ejecutando las acciones que le indique Player. Se encarga principalmente del manejo de las animaciones del personaje, y de provocar las consecuencias de esas acciones.
- **HealthBar**: esta clase simplemente se encarga de la representación de la vida del jugador en una barra de salud que se encuentra encima suya. Al

[8] Cada actualización del contenido de la pantalla.

igual que PlayerMovement, los métodos de ésta son llamados desde Player para que realice su función.

Como vemos el manejo de la lógica del personaje se realiza principalmente desde Player, y esta se encarga de llamar a las otras dos clases cuando sea necesario.

Hay dos clases encargadas del funcionamiento de las tiles del terreno:

- **TileHealth**: esta clase se encuentra en todas las baldosas destructibles de la capa superior. Se encarga de la gestión de la resistencia de la baldosa, tanto de tipo tierra como roca. Cada vez que la baldosa recibe daño, se accede a su método público *damage* para provocárselo.
- **TileDamage**: esta clase se encuentra en las baldosas del elemento lava, y se encarga de producir daño al jugador mientras se encuentre en su posición. Detecta cuando un jugador ha entrado en su zona, y comienza a aplicarle daño, llamando a la función correspondiente del personaje, hasta que deja de estar en contacto con ella.

Una única clase se encarga del elemento proyectil, y es la clase homónima **Proyectil**. Ésta podrá ser reutilizada en caso de que se creen más tipos de munición, determinando a través de una enumeración del proyectil del que se trata. Cuando se produce una colisión determina el tipo de objeto con el que ha impactado y le resta los puntos de salud correspondientes. Después pasa a herir todos aquellos elementos que se encuentren cercanos al punto de colisión.

La generación del terreno se realiza exclusivamente con la clase **GameFieldGenerator**. Esta clase contiene todos los tipos de tiles que formarán el terreno, además de los tres tilemaps que las contendrán. En este caso vamos a detallar el orden de llamada de sus funciones para aclarar más el funcionamiento de la generación del terreno. La función *iniciar* será llamada por el GameManager para dar comienzo a la generación, y ésta realizará una llamada a las dos funciones *prepare* (*prepareFloor* y *prepareAboveFloor*), que cada una comenzará la generación del terreno correspondiente. Estas llamarán a la función *initTileArray* que almacenará en las matrices destinadas a ello los valores de Perlin Noise que vamos a usar. Posteriormente, llamarán a la función *PopulateTilemap* que se encargará de instanciar los elementos en el mapa siguiendo esos valores de ruido. Y ya por último, con *frameMap*, se crea el marco que delimita el terreno.

La clase **GameManager** guarda todos los datos que pueden ser accedidos por todas la clases del juego. Como vemos tiene un atributo privado llamado instancia, que es el utilizado para la implementación del patrón Singleton. El método *GetInstance* puede ser llamado por cualquier clase para acceder a la instancia y con ello a los datos de este objeto. El objeto que contiene este script se mantiene durante toda la ejecución del programa, y es por ello que nos sirve para pasar datos de una escena a otra, como puede ser el tamaño de tablero que estipulamos en la pantalla de ajustes para que sean utilizados en la escena de juego. Sus métodos se encargan de posicionar los elementos en el mapa y de cargar las escenas y sonidos correspondientes a cada momento.

La banda sonora del juego, se controla desde el GameManager, pero la clase encargada de su correcta reproducción es **AudioControl**. Contiene como atributos todas las pistas de sonido, y según la función que invoque el GameManager, reproducirá unas u otras.

La pantalla de ajustes del inicio del juego, tiene su propia clase también. Esta es **SettingsBehaviour**, y se encarga de mostrar y ocultar el panel según corresponda y de llamar a la función del GameManager para que establece las nuevas dimensiones del tablero.

Por último, para la pantalla de puntuaciones, la clase **GameOverScript**, contiene un único método que es llamado por el GameManager para establecer los textos con las puntuaciones correctas una vez finaliza la partida.

En la figura 20, podemos ver el conjunto de las clases que se han ido explicando, con algunos de los atributos y métodos más importantes mencionados.

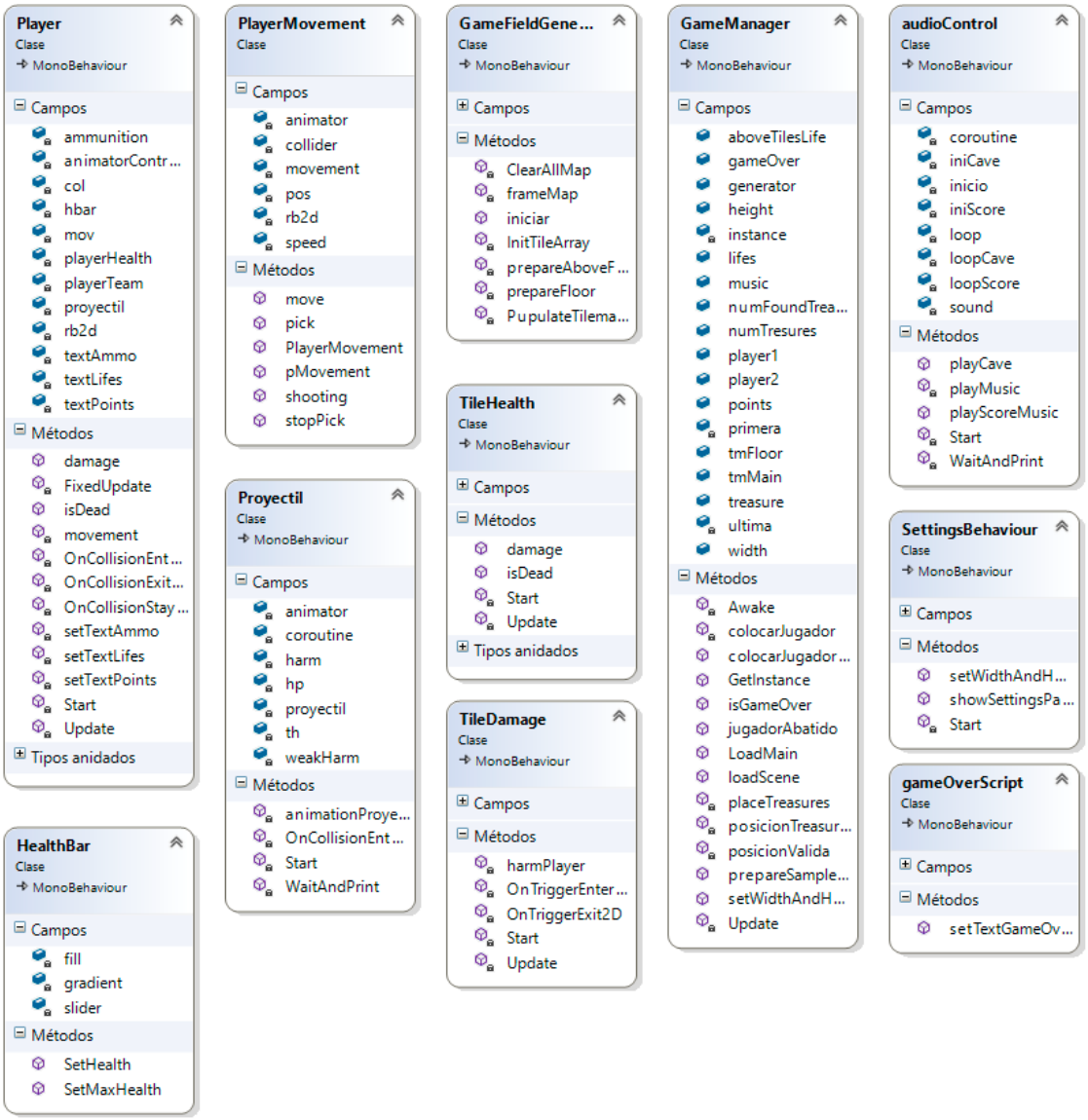


Figura 20. Diagrama de clases simplificado

4 RESULTADO

Para mostrar el resultado obtenido con este proyecto, vamos a detallar el trascurso normal de una partida, desde que se inicia el juego hasta que finaliza. El jugador debe pasar por tres pantallas distintas: la pantalla de inicio, la pantalla del propio juego y la pantalla final con las puntuaciones.

Pantalla de inicio

La pantalla de inicio (véase Figura 21) es la primera imagen que se muestra al jugador, y se carga con la escena inicial del juego. Está compuesta por un *canvas* que tiene de fondo una imagen decorativa de producción propia, y dos botones, que también cuentan con una imagen propia simple para cambiar su apariencia.

La interacción con esta pantalla se realiza a través del ratón, pudiendo pulsar cualquier de los dos botones presentes. Ambos botones al ser seleccionados, llaman a una función distinta que realiza una acción determinada.

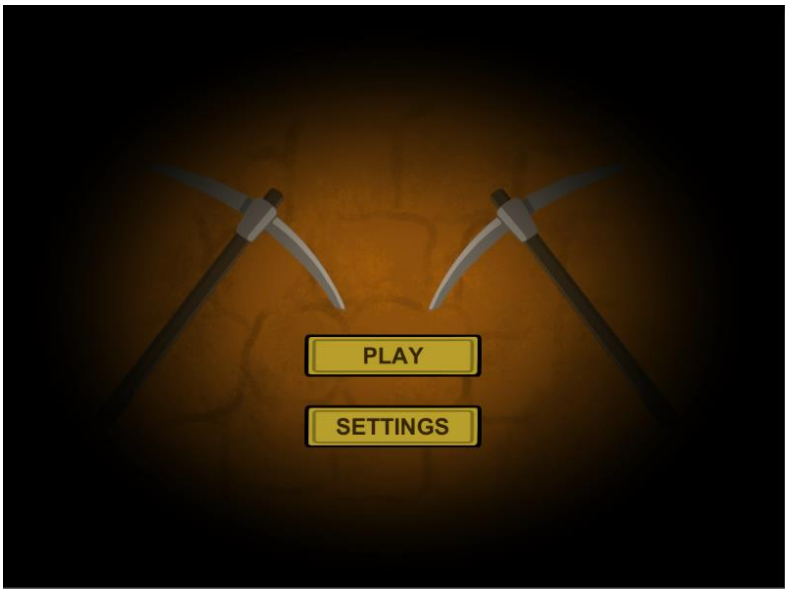


Figura 21. Pantalla de inicio del juego

Al pulsar el botón “Settings”, la función invocada, hace aparecer otro *canvas* por encima del actual que nos muestra una nueva pantalla (véase Figura 22). Este nuevo *canvas*, está compuesto por la misma imagen de fondo, y por dos botones con la misma apariencia que los anteriores, además de tres campos de texto estáticos, y dos campos de texto para introducir información. Estos dos últimos, nos permiten introducir a través de teclado dos valores numéricos, que indican las dimensiones que queremos que tenga el campo de juego que vamos a jugar, en el primero se introduce el ancho y en el

segundo el alto. Los números introducidos serán la cantidad de rule tiles que se instancien para esa propiedad, de manera que si introducimos “50” y “60”, el terreno de juego a generar, tendrá 50 rule tiles de ancho y 60 rule tiles de alto.

Como ya hemos dicho, tenemos dos botones. El primero, que contiene el texto “Exit”, al ser pulsado, llama a una función que nos devuelve a la pantalla inicial en la que nos encontrábamos antes, sin que se guarde ninguno de los valores introducidos. El segundo botón, indicado con el rótulo “Save”, invoca a otra función que comprueba si los valores introducidos son mayores que los límites que se han impuesto, en nuestro caso 30. De no cumplir este requisito, aparecerá un texto amarillo en la pantalla que nos indicará que debemos cambiar los valores por unos nuevos. Si finalmente se cumple la condición, la función guarda los valores introducidos en el GameManager para que puedan ser utilizados a la hora de crear el terreno, y nos devuelve a la pantalla inicial.

De nuevo en esta pantalla, podemos pulsar el botón “Play” que hará cargar una nueva escena que contendrá el terreno de juego con las medidas que hemos introducido. En caso de que no se hayan introducido nuevos valores, el juego utilizará los valores por defecto, que generará un terreno de juego con setenta rule tiles de ancho y de alto.

Durante todo este proceso, suena de fondo el tema de la banda sonora asociado a estas pantallas, tanto la inicial como la de ajustes.

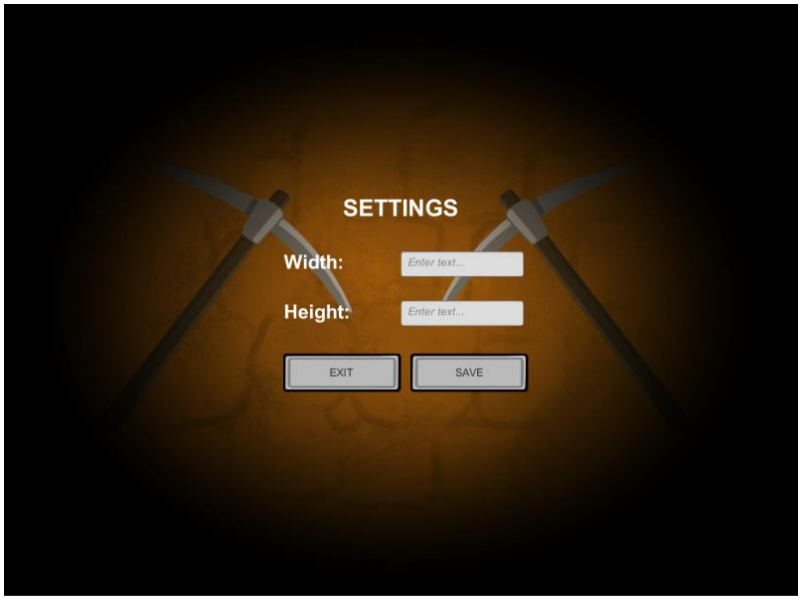


Figura 22. Pantalla de ajustes del juego

Pantalla de juego

Esta es la pantalla que nos muestra el desarrollo de la partida y que permite a los usuarios jugar el juego. Con la llegada a esta escena, comienza a sonar el tema sonoro

principal del juego, que se repetirá lo que dure la partida. Como vemos en la Figura 23, la pantalla se divide en dos mitades: la mitad derecha es para el jugador 1 y la parte izquierda para el jugador 2.

Figura 23. Captura de la pantalla de juego

En cada una de estas mitades, nos encontramos en la esquina superior izquierda un recuadro que nos indica las vidas y munición restantes, y puntos recolectados por ese jugador. Encima de cada personaje podemos ver que aparece su barra de vida, ésta disminuirá cuando un disparo, propio o del otro jugador, impacte contra el personaje, o camine por encima de lava. Mientras se va vaciando la barra cambia su color de verde, a amarillo y a rojo. Una vez se ha vaciado por completo, el jugador reaparece en otro punto del mapa, y aparecerá que tiene una vida menos en el recuadro.

El jugador 1 utilizará las flechas de teclado para desplazarse y el *Ctrl* derecho para disparar, mientras que el jugador 2, utilizará las teclas “AWSD” y la barra espaciadora para efectuar el disparo.

En la Figura 23 podemos ver las tres capas de terreno que se han creado. En la parte superior de la mitad del jugador 1, podemos ver una porción del marco que rodea el tablero. En la pantalla del jugador 2, en la parte inferior, vemos tierra y roca de la capa superior que el jugador puede decidir picar. Y bajo los pies de ambos jugadores, vemos agua y tierra pertenecientes a la capa de suelo que se va descubriendo al destruir la capa superior. También podemos ver los tesoros repartidos por la escena y que los jugadores recogerán para acumular puntos al pasar por donde se encuentran.

En la Figura 24 vemos como el terreno va cambiando según el jugador va picando su superficie. Se puede apreciar como gracias al funcionamiento de las rule tiles el aspecto de cada baldosa afectada se modifica según su entorno para ofrecer un mejor aspecto visual.

Figura 24. Secuencia de alteración del terreno ocasionada por el jugador

Por último, mostramos una visión del mapa de juego completo generado por el algoritmo desarrollado basado en Perlin Noise en la Figura 25. Se puede ver como se han generado zonas aleatorias con los distintos elementos configurados. No se aprecia que ninguna de ellas siga ningún patrón preestablecido, ni que formen conjuntos demasiado cuadriculados que acaben resultando monótonos para el usuario. También se puede ver la colocación de ambos jugadores y el campo de visión de cada uno sobre terreno señalizado por un rectángulo blanco.

Cómo vemos en la capa superior siempre se respeta que hay tierra rodeando a la piedra, y en la capa base vemos que el agua profunda siempre se encuentra dentro del agua poco profunda, y la arena y la lava dentro de zonas de tierra. De esta manera se garantiza que no se produzcan combinaciones imposibles que no queremos que se produzcan.

Durante el desarrollo de la partida, los jugadores irán desplazándose por el terreno de juego recolectando el mayor número posible de tesoros para conseguir más que el otro jugador. Cada jugador podrá disparar proyectiles hasta que se le agote la munición. Con estos disparos puede restar puntos de vida al otro jugador en caso de que impacte contra él o en sus alrededores más inmediatos. Si un jugador se acerca lo suficiente a una elementos de la capa superior, comenzará a picar ese material, de esta manera, mientras se mantenga intentando avanzar en esa dirección, irá restan puntos de resistencia a ese material. Cuando se le agoten los puntos a ese material ese será destruido y el jugador podrá avanzar por dónde éste se encontraba. Una vez se hayan recolectado todos los tesoros repartidos por el mapa o alguno de los dos jugadores se quede sin vidas, se pasa a la pantalla con las puntuaciones.

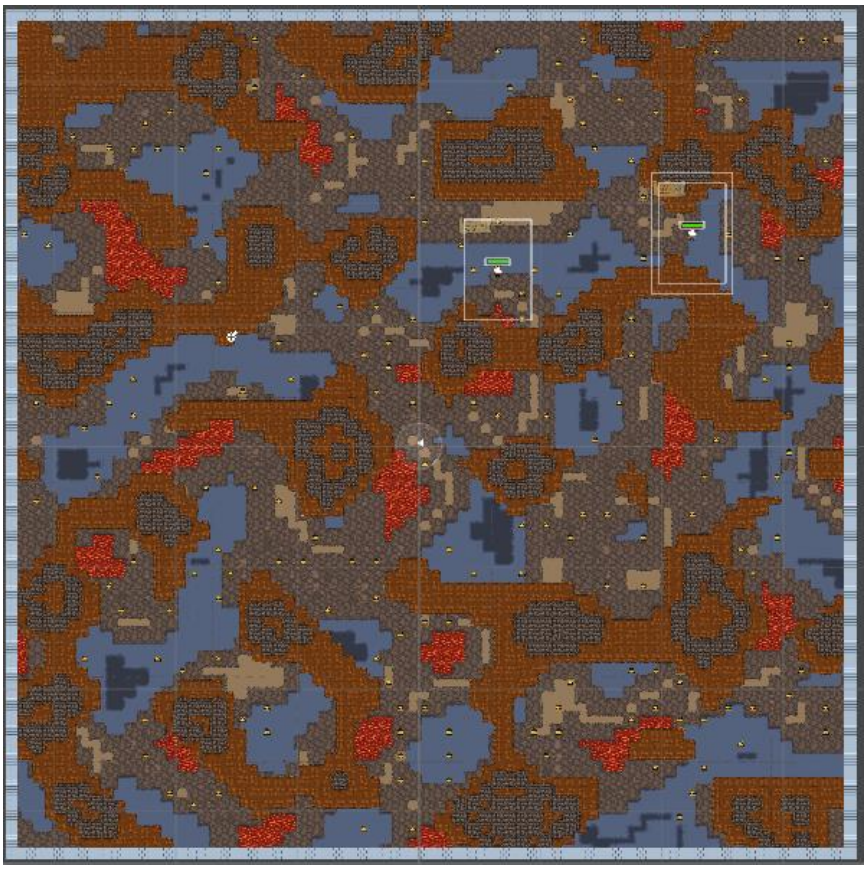


Figura 25. Ejemplo de mapa de juego completo generado por el algortimo desarrollado

Pantalla final

Esta pantalla se muestra cuando se ha acabado la partida. Con la llegada de esta última pantalla, comienza a sonar la última pista de la banda sonora. Se hace aparecer sobre el terreno de juego un canvas con un fondo marrón con opacidad reducida, y sobre el que se muestran las puntuaciones obtenidas por ambos jugadores, cada una en la mitad de la pantalla correspondiente a ese jugador.

Además de las puntuaciones, cuyos valores se obtienen del GameManager, se muestra un título que anuncia “Game Over”, que indica que se ha acabado la partida. Para señalizar claramente quien ha sido el ganador, aparece una corona en el lado del personaje que ha obtenido más puntos. En caso de que ambos jugadores tengan el mismo puntaje, aparecerá una corona en ambos lados, indicando un empate.

Esta pantalla permite al personaje que siga con vida, seguir desplazándose por el terreno de juego, realizando modificaciones sobre él y recogiendo nuevos tesoros, pero estos ya no se tendrán en cuenta en la puntuación establecida.

En la Figura 26, se puede ver el resultado de una partida en la que el jugador 1 ha obtenido una puntuación de 30 puntos y el jugador 2 una puntuación de 60. Como vemos, en el lado del jugador 2 nos encontramos la representación de una corona. Esta

es la corona que le proclama vencedor de la partida. Además vemos que la parte del canvas correspondiente a la mitad del jugador 1 se encuentra completamente opaca, esto nos indica que el jugador 1 ha perdido todos sus puntos de vida y no puede volver a reaparecer en el mapa. Al contrario, la parte del jugador 2 es translucida, lo que le permite ver el terreno de juego, y nos indica que puede desplazarse aún por él.

Figura 26. Pantalla de fin del juego mostrando las puntuaciones

4.1 Comparativa con Mine Bombers

Tal y como se indicó al comienzo de este TFG, nuestro juego aspira a ser una versión simplificada del juego original Mine Bombers. Al ser éste muy complejo en cuanto a la cantidad de opciones que presenta al jugador, no se han podido recrear todas ellas, pero si algunas de las más básicas.

Como ya sabemos, ambos juegos están ambientados en una mina, y los jugadores deben recorrer el mapa para recoger tesoros, atacar al otro jugador y conseguir más tesoros que él. En el juego original también teníamos la opción de proclamar vencedor a quien más veces derrotase al otro jugador, pero en nuestra versión se ha decidido que el objetivo era únicamente conseguir más puntos.

El juego original permite un modo multijugador con hasta cuatro jugadores, pero debido al tamaño del teclado, un número de jugadores superior a dos resulta incómodo. Debido a esto, en nuestra versión, se ha limitado el número de jugadores a dos, de manera que cada persona pueda jugar cómodamente en su parte del teclado.

Como vemos en la Figura 27, ambos juegos presentan escenarios vistos desde arriba, es decir, son de vista cenital. Pero aquí encontramos también una de las diferencias más notables entre ambos, ya que en el original se le presenta a ambos jugadores el mapa

completo de juego y ven sobre él moverse a sus personajes, pero para el desarrollado en este TFG, se ha decidido dividir la pantalla de manera que cada jugador vea solo la porción de terreno cercana a él. Esto provoca que los jugadores no puedan conocer como es el resto de terreno por adelantado sin haberlo recorrido antes.

Figura 27. Pantallas de juego del Mine Bombers y el juego desarrollado

Para poder avanzar por el terreno los jugadores pueden picar elementos que se encuentren. En el juego Mine Bombers deben picar material para avanzar, ya que todo el mapa está cubierto de él, pero no resulta pesado ya que la tierra resulta muy rápida de destruir, en cambio la piedra es más tediosa. En nuestro título, grandes zonas del terreno están sin tapar para que el jugador pueda moverse sin necesidad de estar picando, y en este caso la tierra resulta un poco más difícil hacer que desaparezca.

Ambos mapas tienen los elementos tierra y piedra que pueden ser destruidos por los jugadores. El juego Mine Bombers cuenta con muchos otros tipos de materiales con distintas funcionalidades, como por ejemplo un suelo que provoca una explosión al contacto. En nuestro caso, se ha decidido no crear tantos tipos de elementos en la capa superior y se ha ampliado la funcionalidad de la capa inferior, es decir, la que el jugador va descubriendo y por la que se desplaza. En el juego original, esta capa esta solo compuesta de tierra que forma el suelo, pero en nuestra versión, se han creado distintos elementos: tierra y arena, lava que resta vida al pasar sobre ella, agua que hace disminuir tu velocidad y agua profunda que no te deja pasar.

La principal diferencia entre ambos, es la que se buscaba con el desarrollo de este proyecto, la generación procedural del terreno. El título original cuenta con unos mapas preestablecidos por los desarrolladores para que jueguen los usuarios, e incluso había un editor para que ellos mismos creasen un mapa. Pero esto hace que una vez te has aprendido la disposición de los mapas que te ofrecen, tu única alternativa es crearte tus propios mapas, lo que supone un esfuerzo mayor por parte del jugador, y posiblemente tras unos pocos diseños, usuario perdería motivación. Además de que el jugador que lo ha diseñado conoce perfectamente todos los elementos que lo componen.

El algoritmo que hemos desarrollado nos generará mapas distintos que poder jugar en cada partida. De esta manera, ninguno de los jugadores sabrá que es lo que se va

encontrar al comenzar el juego. Además, se ha añadido la posibilidad de determinar cuáles son las dimensiones del campo, luego los jugadores podrán decidir sus valores dependiendo el tipo de juego que estén buscando, si una partida rápida o más larga, o incluso decidir crear una mapa muy grande y colaborar para encontrar todos los tesoros.

Esto último nos lleva a otra diferencia. En el juego original, si juegas al modo normal de juego viendo el mapa al completo, los tesoros repartidos que encuentran en él están visibles para el jugador, mientras que en nuestro juego, puedes encontrarte tesoros escondidos bajo elementos destruibles, lo que dificulta la tarea de encontrar todos ellos.

Otra característica adicional, es que en el juego original lo jugadores siempre comenzaban en las esquinas del tablero, en cambio en nuestra versión puedes comenzar en cualquier parte. Cada vez que el jugador vuelve a la vida, se recalcula un punto nuevo para colocar al personaje. Esto hace que cada vez que vuelvas a aparecer en el mapa no tienes que recorrer la misma zona que ya recorriste antes de agotarse tus puntos de vida.

5 CONCLUSIONES

En este trabajo de fin de grado se ha creado un videojuego con vista cenital para dos jugadores basado en el ya existente “Mine Bombers”. Al igual que en éste, el jugador debe abrirse paso por el terreno de una mina destruyendo bloques que le impiden su avance mientras va recolectando tesoros.

A diferencia del original, nuestro juego genera los escenarios de manera procedural gracias a la utilización del algoritmo de ruido de Perlin y el funcionamiento de las Rule Tiles de Unity para ofrecer una mejor apariencia visual y poder adaptarse a los cambios que se produzcan en él.

Se ha comenzado configurando las reglas de las rule tiles. Partiendo del elemento más abundante, la tierra de la capa base, y configurando sus uniones con el resto de materiales, el agua, la lava y la arena. Posteriormente se ha creado el agua profunda y su unión con el agua poco profunda. A la hora de crear esta capa, se comenzó configurando las reglas de las rule tiles para que tuvieran en cuenta qué elemento tenían adyacente, pero esto provocaba algunos problemas cuando el algoritmo posicionaba elementos cerca que no tenían ninguna relación. Como tanto el elemento lava, como arena y el agua tienen sus representación con la unión con el elementos tierra, realmente no es relevante que elemento se encuentra adyacente, ya que siempre habrá una pequeña representación de tierra separándolas. De este modo, aunque se encuentren adyacentes, por ejemplo, arena y lava, habrá un pequeño muro de tierra que separa ambos elementos. Teniendo esto en cuenta, se ha decidido transformar esas relaciones que tenían en cuenta que elemento estaba en contacto, por las que nos permitían cualquiera menos el que estamos intentado colocar, en los elementos lava y arena, ya que ninguno de los dos posee transiciones con ningún otro elemento que no sea tierra. De este modo, ambos elementos siempre mostrarán la representación que le corresponde independientemente de los elementos que el algoritmo le haya colocado en las casillas adyacentes.

Desgraciadamente no se ha podido seguir el mismo método con el agua poco profunda y agua profunda, ya que el agua poco profunda debe tener en cuenta si está dando paso a tierra o a agua profunda; y el agua profunda, solo puede transicionar con agua poco profunda. Debido a esto, en ocasiones puntuales se presentan en el mapa algunas representaciones del agua que rompen un poco con la continuidad general del terreno, ya que aparecerá la representación por defecto en vez de la que correspondería con su situación.

Para la creación de la capa superior, las tiles también deben tener en cuenta con qué material están en contacto, por lo que también se han utilizado las reglas teniendo en cuenta con que elemento están adyacentes. La mayor dificultad en esta capa ha sido la gran cantidad de representaciones y reglas necesarias para que el resultado quedase exactamente como debía, ya que como el jugador puede modificar el terreno, pueden darse posibilidades que inicialmente no se tenían, como que una tile de roca esté en

contacto con una tile vacía y una de arena simultáneamente. Esto nos da lugar a un número de representaciones necesarias muy alto. Además en la mayoría de los casos, el esfuerzo de realizarlas y configurarlas no resultaría recompensado en relación a la mejora en la apariencia que se consigue. Debido a esto, se ha optado por disminuir el número de representaciones que realizar aprovechándose del orden de evaluación de las reglas que realiza la rule tile. Se han colocado en las posiciones superiores, que son las primeras en ser evaluadas, aquellas que precisen de más condiciones para ser colocadas, luego las representaciones que están destinadas a situaciones concretas se colocarán correctamente. De esta manera, si no las cumplen dan paso a la posibilidad de otras con menos condiciones, que pueden ser utilizadas en situaciones más generales. Además, en aquellas en las que la tierra está en contacto con posiciones vacías, se han eliminado las condiciones que indicaban continuidad del elemento, para poder permitir esa representación aun cuando en esa posición se encuentre el elemento roca, ya que visualmente no resulta llamativo y nos permite usar representaciones visualmente más correctas para cada elemento. Pero en algunos casos se podía apreciar que no se había realizado esa representación del paso de un elemento a otro, luego se decidió crear y configurar aquellas casillas que resultaban más claras para ofrecer un mejor aspecto visual.

La disposición de cada ruletile en el mapa se ve determinada por el algoritmo Perlin Noise. Se ha hecho uso de la propia función de la que dispone Unity en una de sus librerías, de manera que se consulta a la función el valor que obtiene en un punto concreto y según el resultado obtenido se instancia un elemento determinado. Esta función tiene la limitación de que al consultar el mismo punto, siempre te devuelve el mismo valor. Para solventar esta inconveniencia, se calcula un número pseudoaleatorio dentro de un rango muy amplio para que no se repita fácilmente, y se suma a las coordenadas del valor que se quiere consultar. De esta manera se consulta una parte distinta del plano cada vez que se llame a la generación.

En el desarrollo de esta fase, se tienen que tener dos aspectos en cuenta: el rango de los valores que representan a cada elemento, que nos determinará qué cantidad de ese elemento aparecerá de manera general en el mapa; y la escala a la que queremos consultar el mapa con los valores de ruido, que nos permite decidir cuánto queremos que varíe un valor obtenido para una casilla y su adyacente, lo que nos sirve para ajustar el tamaño que queremos conseguir de las aglomeraciones que se forman de cada elemento. Ambos se han determinado a través de prueba y error hasta que se han obtenido unos resultados similares a los que se buscaba.

A la hora de usar el rango de valores, se han aprovechado rangos continuos entre sí para elementos que queremos que se presenten adyacentes, y rangos separados para evitar contactos no deseados, y otorgando un mayor número de valores para elementos que queremos que se encuentren presentes con más frecuencia en el mapa. De esta manera si hay dos elementos que no queremos que se encuentren juntos, les asignamos rangos muy separados para que sea altamente improbable que los valores correspondientes a esos elementos se encuentren adyacentes en el mapa.

En cuanto a la escala de la consulta, esto nos determina la diferencia de valor que nos proporcionan dos puntos continuos. Dependiendo el tipo de terreno que queramos generar, nos puede interesar que los valores cambien rápidamente para generar mapas menos suaves; o que los valore apenas varíen para que el resultado sea más uniforme. Esto lo conseguimos multiplicando por un número constante el valor de la coordenada que queremos consultar, y como la función nos permite realizar consultas pasándole como parámetros posiciones decimales, podemos utilizar cualquier valor que queramos para modificar la distancia de los puntos que consultamos.

Estos son los puntos principales con los que se ha caracterizado la generación procedural del terreno. Así como el uso de números pseudoaleatorios para la distribución de los elementos, como tesoros y personajes, que se encuentran en él.

Además de esto, al juego se le han añadido dos personajes, uno para cada jugador, que se desplazan por el campo de juego, pudiendo disparar proyectiles para restar puntos de vida al otro jugador y picar los elementos de la capa superior para abrirse camino. Se han establecido animaciones para cada una de estas acciones en los cuatro sentidos de desplazamiento básicos (arriba, abajo, izquierda, derecha). Para el manejo de estos personajes se ha decidido implementar un *controlador* para manejar las entradas producidas por el jugador, y un *pawn* que determinará las acciones que realice el personaje. También se han repartido tesoros por el mapa, comprobando que se colocan en posiciones a las que el jugador puede acceder, que éstos deben ir recolectando para conseguir puntos y ganar la partida.

Otro elemento importante es el Game Manager, implementado con la utilización del patrón *singleton*, que contiene datos que pueden ser consultados o modificados desde distintas escenas, como la puntuación del jugador o el número de vidas que tiene, y que además se encarga de manejar ciertos aspectos del juego, como la música de fondo que debe sonar en cada momento de la partida, o colocar los personajes cuando hayan sido abatidos.

Con todo esto se ha conseguido crear un juego completamente funcional para dos jugadores, cuyo escenario generado tiene una probabilidad extremadamente baja, prácticamente nula, de verse repetido en dos partidas distintas jugadas por el mismo jugador.

5.1 REFLEXIÓN SOBRE IMPACTOS SOCIALES Y MEDIOAMBIENTALES Y ASPECTOS DE RESPONSABILIDAD ÉTICA Y PROFESIONAL

En el desarrollo de este Trabajo de Fin de Grado, se ven afectados algunos aspectos sobre todo de índole social y ética. La generación procedural nos brinda la oportunidad de ofrecer muchísimo más contenido al jugador, por lo que éste se ve altamente beneficiado. Pero por otro lado, la generación de contenido está siendo producida por un algoritmo en vez de por una persona. Quizás algunos puedan pensar que el desarrollo

de este tipo de tecnologías, podría dejar sin su puesto de trabajo a muchos diseñadores y creadores. Realmente su trabajo se verá modificado con el uso de estos mecanismos, pero no será reemplazado. El diseñador es necesario para que cree componentes que use el algoritmo, además de que será él quien determine las restricciones y mecánicas que debe seguir el método que se vaya a usar. La generación procedural se convierte en una herramienta que ayuda al diseñador en su trabajo.

En cuanto a temas legales de protección intelectual, habría que señalar que se ha utilizado un personaje perteneciente a la compañía Nintendo de la saga Pokémon. Su uso se encuentra permitido pese a estar protegido, ya que se utiliza únicamente con propósitos docentes y de investigación.

Por otro lado, este tipo de técnicas se han aprovechado en el pasado para evitar que los equipos tuviesen que disponer de una gran cantidad de memoria para almacenar el juego, y permitirles de este modo ofrecer una gran cantidad de contenido al usuario. Actualmente, también podría utilizarse para hacer más accesible el recurso para equipos que disponen de menores capacidades de almacenamiento.

6 LÍNEAS FUTURAS

Tal y como se ha dejado ver a lo largo de este documento, se ha conseguido desarrollar un videojuego para dos jugadores que cuenta con las mecánicas básicas y cuyo escenario de juego se crea de manera procedural. De todos modos, hay aún muchas características adicionales que se pueden incluir en este proyecto para hacer que el juego resulte aún más entretenido y se mejore la creación procedural.

Una característica importante podría ser que el juego funcionase en línea, de manera que ambos jugadores no tendrían por qué estar en el mismo lugar en el momento de jugar una partida. Esta función nos permitiría añadir aún más jugadores simultáneos, ya que el número establecido en este proyecto de dos personajes, viene determinado principalmente por la comodidad a la hora de manejar los controles, pero si cada jugador dispone de sus propio teclado, este impedimento desaparece y podrían ampliarse las partidas para que apareciesen más personajes en un mismo mapa. Otra opción posible para añadir otro jugador más sin necesidad de usar distintos ordenadores, sería añadir un tercer personaje que se manejase a través del ratón, por ejemplo, haciendo que se desplace pinchando en alguna zona del mapa, picar manteniendo el botón derecho y dispara con *doble click*.

Otro aspecto importante es la manera de interaccionar entre esos jugadores, refiriéndonos con ello a la manera de hacer disminuir sus puntos de vida y tener más posibilidades de obtener una mayor puntuación que ellos. Por ello, estaría bien aumentar el catálogo de armas o herramientas que nos ayudan a que esto ocurra, replicando herramientas que hay en el juego original o diseñando unas nuevas. De esta manera se consigue aumentar las posibilidades del juego, haciendo que cada arma tenga un efecto distinto. En relación con esto, podríamos repartir objetos de munición por el campo de manera que los jugadores puedan volver a obtener la oportunidad de disparar en caso de que se les agotase su munición. Podría especificarse que cada tipo de arma tenga una munición distinta, de manera que solo se pueda usar si tienes la munición correspondiente, o tener una munición estándar que sirva para todas, pero que cada arma consuma una cantidad diferente, y que sea el jugador el que debe decir como utilizarla.

Para que a la hora de determinar el ganador el resultado sea más incierto, podrían incorporarse distintos tipos de tesoros, cada uno de ellos que otorguen distintas puntuaciones, de manera que un jugador que tiene una puntuación inferior en un momento dado pueda tener un golpe de suerte que le haga dar la vuelta a la partida. También podría establecerse un solo tesoro de una relevancia mucho mayor, de manera que pueda jugarse una partida que consista en encontrar ese tesoro, y que el jugador que lo encuentre antes gane.

Dentro del juego que tenemos con las características actuales, hay algunos aspectos aún pendientes que podrían ayudar a mejorar la experiencia de juego. Por ejemplo, como se ha mencionado en puntos anteriores, las combinaciones entre los elementos de tierra,

roca y vacío que se encuentran en la capa superior, eran demasiadas como para crearlas todas, luego como mejora se podrían ampliar el número de estas representaciones configuradas para que el aspecto visual sea mejor. También podrían añadirse efectos sonoros al realizar ciertas acciones, como disparar, picar o recoger un tesoro, o incluso efectos visuales como un brillo que rodee al personaje cuando ha recogido un tesoro, para que el jugador se sienta más metido en la partida.

Siguiendo el camino del sonido y enlazando con el tema principal de este proyecto, la generación procedural, podría aspirarse a incluir sonidos procedurales en el juego. Estas generaciones deberán ser muy básicas ya que el juego de por sí no cuenta con la complejidad suficiente para desarrollar algoritmos elaborados basados en muchos componentes distintos. Podría ser por ejemplo, que al acercarse un jugador a otro jugador, la banda sonora se acelerase o se subiesen los sonidos graves para transmitir más emoción; o incluso al estar cerca de un tesoro que se reproduzca algún sonido complementario. Por otro lado, si se quiere crear una base para el sonido que se reproduzca, podría tenerse en cuenta el número casillas que se crean de cada elemento, y que cada uno de ellos tenga alguna característica concreta, de manera que dependiendo el número de casillas que haya de ese elemento, el sonido que se cree será distinto. También se puede añadir más complejidad a los elementos existentes, para que presenten ciertas características que ayuden a determinar los sonidos que se deben reproducir, y de esta manera que la generación sea más compleja y específica para cada partida, o incluso en el caso de que se jugase en ordenadores separados, podría producirse para cada personaje una banda sonora distinta dependiendo, por ejemplo, de los elementos que esté visualizando en su campo de visión.

También podrían utilizarse otros métodos de generación procedural para crear nuevos escenarios, de manera que el juego intercale entre ellos o se combinen para dar lugar a escenarios mucho más diversos. Algunos de los otros métodos que podemos utilizar los hemos explicado con anterioridad, como pueden ser autómatas celulares o diagramas de Voronoi para determinar las regiones de cada elemento, o si buscamos una estética aún más distinta, se puede crear algún método basado en agentes. Cada uno de estos métodos nos permite determinar distintas configuraciones que nos dan un mayor rango de variabilidad entre los escenarios que crean. Incluso si se quiere ampliar la funcionalidad del juego, se pueden utilizar algoritmos para crear escenarios con forma de laberinto y dificultar aún más el avance de los jugadores, como se presenta en alguno de los escenarios del título original.

Las nuevas posibilidades que nos abren nuevos algoritmos, más el aumento en el número de armas, podrían darnos pie a tener nuevos elementos que conformen el mapa, de manera que algunos puedan solo ser destruidos con un tipo específico de munición, o requieran más daño. O simplemente más elementos que le den un nuevo aspecto visual.

Un último añadido al juego, podría ser limitar el campo visual que tiene el jugador del terreno. Esto ayuda a que el juego sea más entretenido porque no sabes con que te vas a encontrar inmediatamente después. Esto se puede conseguir simplemente limitando

los bordes de visión de la cámara, de manera que parezca que el jugador está sumido en la oscuridad, o incluso hacer que el mapa fuera completamente oscuro, pero configurarse la lava de manera que ésta emitiese luz, así como una antorcha que llevase el jugador. Con esta última manera, se conseguiría que haya zonas más iluminadas que otras, e incluso zonas completamente a oscuras, ya que algún elemento puede bloquear la llegada de luz a una región concreta, lo que además aumentaría el realismo de los gráficos del juego.

Estas son solo algunas ideas que se han contemplado como posibles vías de desarrollo de este videojuego, de manera que en un futuro pueda ofrecer al jugador aún un mayor entretenimiento.

7 ENCUESTA

Se ha decidido realizar una encuesta a un número reducido de personas para recibir su opinión sobre los resultados obtenidos con el desarrollo de este proyecto.

7.1 Objetivo de la encuesta

Con la realización de esta encuesta, se pretende obtener una opinión general del grupo encuestado sobre el videojuego desarrollado y la generación procedural conseguida, así como comparar algunos aspectos respecto al juego original.

7.2 Procedimiento de realización de la encuesta

A la hora de la comenzar la encuesta, se le ha sugerido al encuestado que realizase una lectura rápida de las preguntas que se le iban a presentar.

A continuación, se ha presentado al encuestado el juego original Mine Bombers, referido en la encuesta como Juego 1. Se le han explicado los controles pertenecientes a este juego, así como su objetivo y funcionamiento, y se le ha indicado que comenzase una partida. En caso de realizar la encuesta a más de una persona, se ha permitido que jugasen al juego dos encuestados simultáneamente, ya que se trata de un juego para dos jugadores. En caso de solo haber un solo jugador, la persona que realiza la encuesta ha tomado el papel del segundo jugador. Se les ha permitido jugar todas las rondas que deseasen, determinando como límite un mínimo de dos.

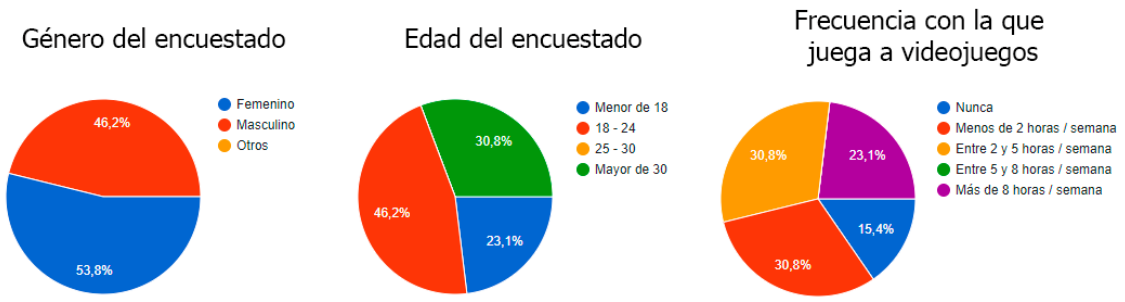
Una vez hayan considerado haber jugado suficientes partidas como para conocer el juego, se les ha presentado el desarrollado en este TFG, referido en la encuesta como Juego 2. Se les han explicado el objetivo y funcionamiento de este nuevo juego, y los controles pertenecientes a cada jugador, y se les ha permitido decidir el tamaño del terreno de juego. Al igual que en el otro juego, se ha impuesto un mínimo de dos partidas, pero tras ellas los encuestados pudieron decidir jugar todas la que quisieran.

Finalmente, tras haber tenido la oportunidad de jugar ambos juegos, se les ha vuelto a proporcionar la encuesta, realizada a través de “Google Forms”, para que aportasen su contestación. En el Anexo 1 puede encontrarse una plantilla de la encuesta con las distintas preguntas realizadas.

En caso de que el encuestado fuese menor de edad, se ha solicitado el consentimiento expreso a ambos padres o tutores del mismo antes de la realización de la encuesta.

7.3 Resultados obtenidos y su interpretación

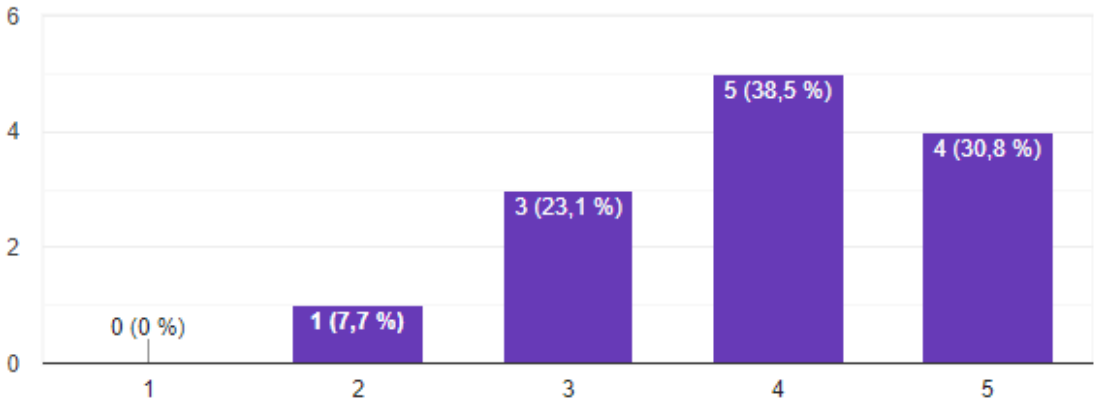
Se ha realizado la encuesta a un total de 13 personas. La encuesta comienza con preguntas sobre los encuestados para determinar a qué tipo de perfiles se realizan las preguntas. Se ha obtenido que de las 13 personas encuestas, 7 de ellas han sido mujeres y 6 hombres. En la Gráfica 1 se puede ver los gráficos que presentan las respuestas a las otras preguntas, la edad que tienen y la frecuencia con la que juegan a videojuegos. Como vemos se ha realizado la encuesta a perfiles dispares.



Gráfica 1. Graficas circulares con los datos de los encuestados

A continuación, se pregunta sobre distintos aspectos del juego desarrollado en este proyecto.

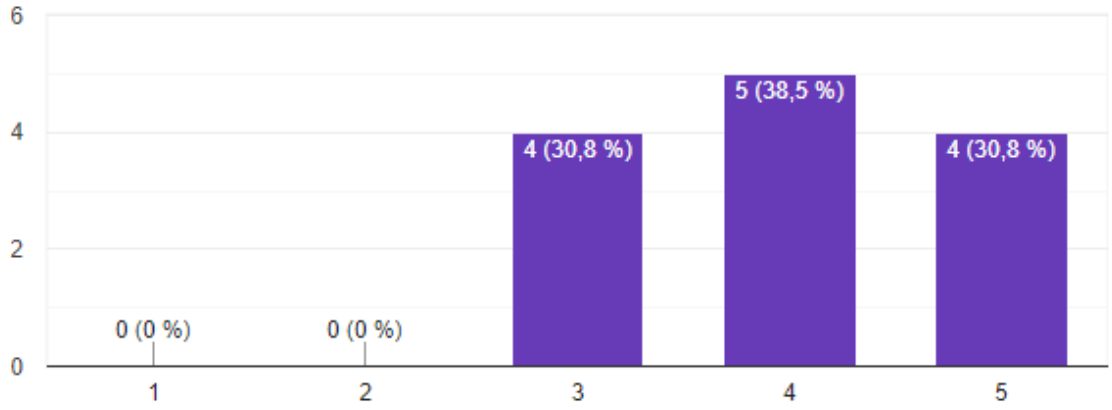
Valoración de la comodidad de los controles



Gráfica 2. Diagrama de barras con la valoración de la comodidad de los controles

Con esta pregunta, se pretende descubrir si los controles escogidos son los apropiados o se podría haber realizado otra combinación mejor. Como podemos ver, la mayoría de las personas encuestadas encuentran cómodos los controles elegidos, pero hay casi un cuarto de los encuestados que no lo ven así.

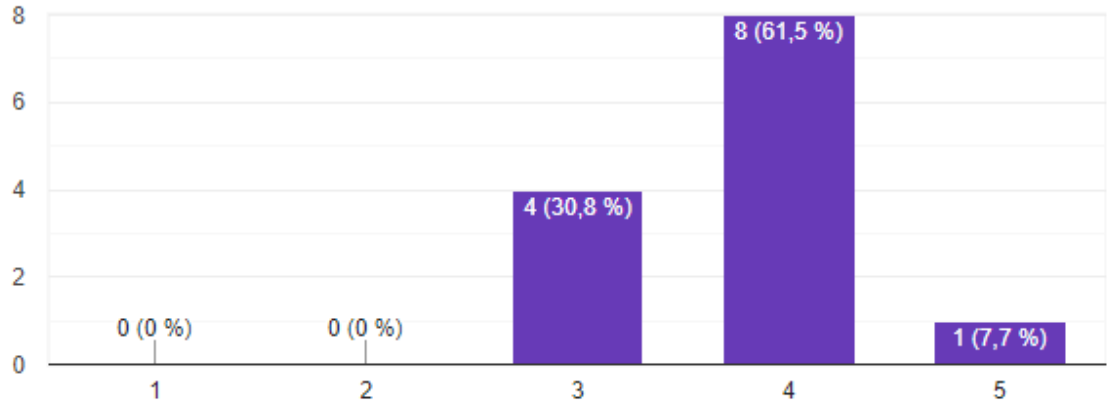
Valoración general de los gráficos



Gráfica 3. Diagrama de barras con la valoración general de los gráficos

Podemos observar como la valoración en general recibida sobre los gráficos es positiva, con esto podemos determinar que aún susceptibles de mejoras, los gráficos parecen ser aceptados por el grupo encuestado.

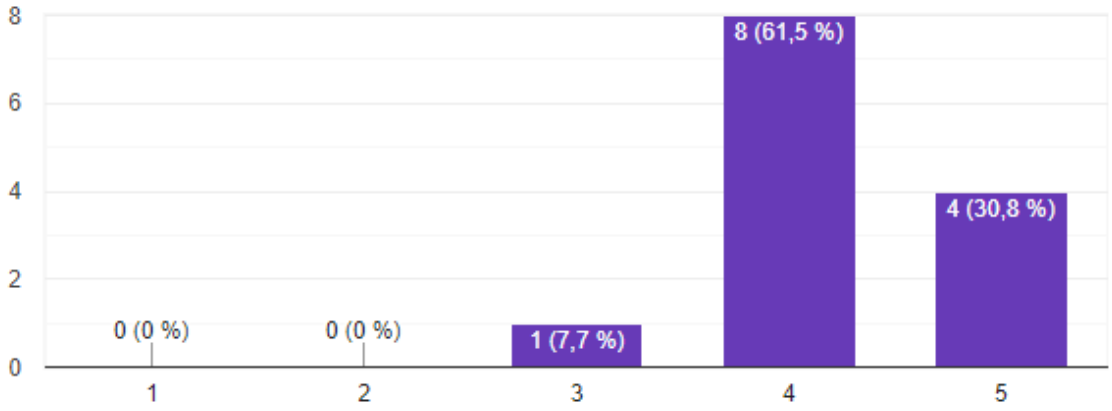
Valoración de la variedad de los mapas probados



Gráfica 4. Diagrama de barras con la valoración de la variedad de los mapas

Con esta pregunta, queremos determinar si el nivel de diferencia entre los mapas generados es suficiente para los usuarios. El grupo de encuestados ha ofrecido resultados en su mayoría positivos, pero solo uno lo considera muy variado y muchos aportan una respuesta más neutral, por ello, concluimos que para el grupo de prueba, los mapas generados podrían presentar una mayor variedad.

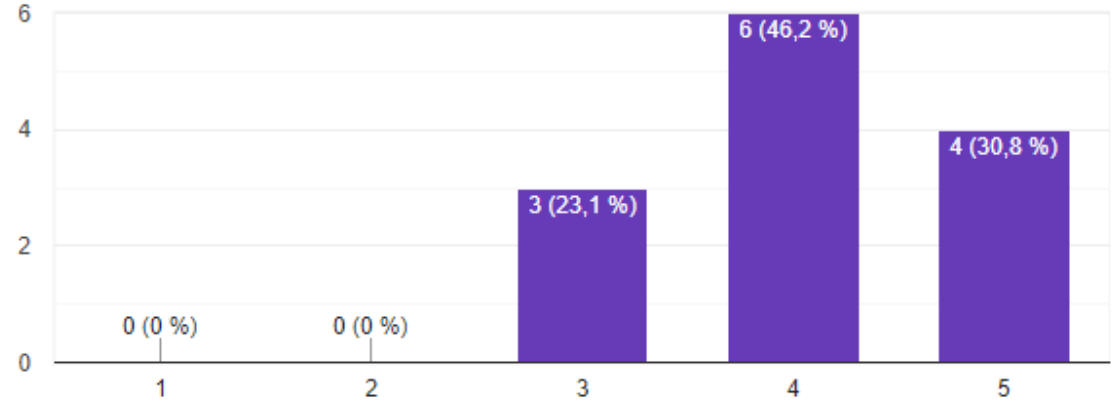
Valoración de la distribución de los elementos (tesoros y personajes) por el mapa



Gráfica 5. Diagrama de barras con la valoración de la distribución de los elementos

Como muestra el gráfico, el conjunto de los encuestados también considera apropiada en su mayoría la distribución de los elementos por el mapa, luego se trata de un aspecto con una valoración positiva.

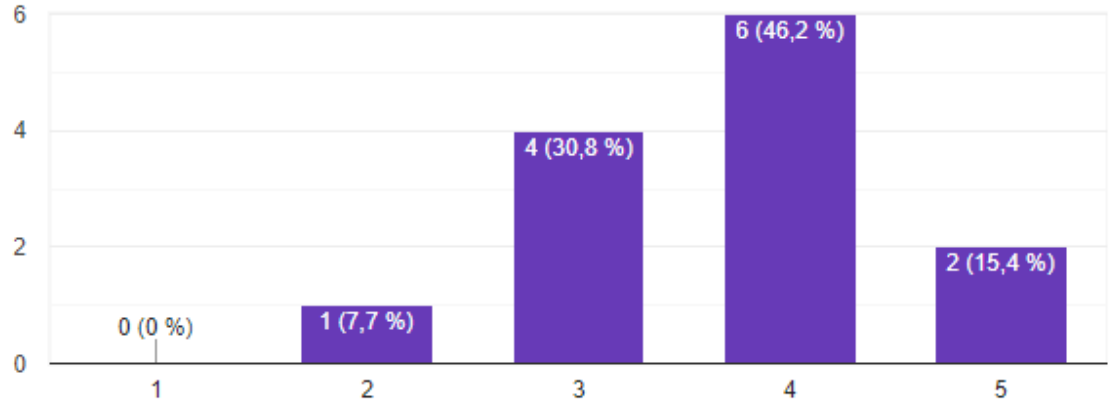
Valoración de la transición entre los elementos que conforman el mapa (por ejemplo, de lava a tierra)



Gráfica 6. Diagrama de barras con la valoración de la transición de los elementos

Las respuestas a esta pregunta nos indican la opinión de los encuestados respecto a la representación del paso de un elemento a otro. Como vemos, las respuestas en general nos indican que los encuestados están conformes con las transiciones realizadas.

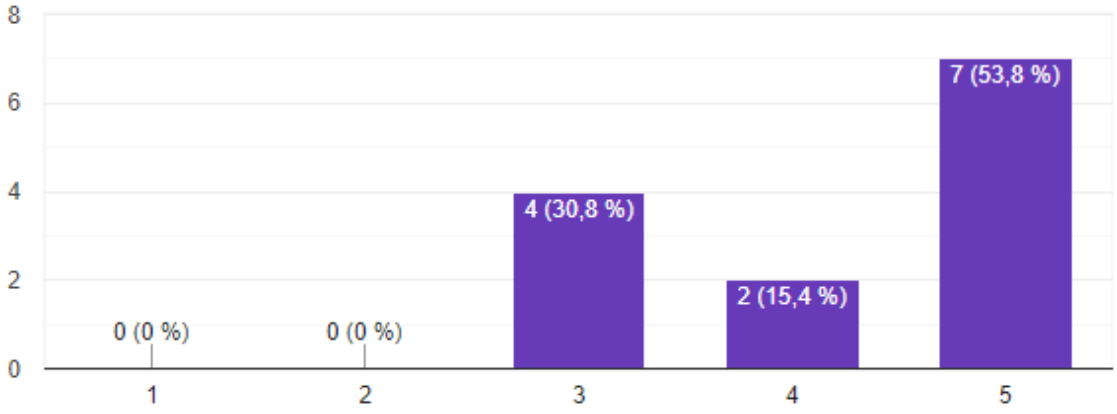
Valoración general de los mapas probados



Gráfica 7. Diagrama de barras con la valoración general de los mapas

Esta gráfica nos indica la valoración de los encuestados respecto a los mapas que han jugado. En este caso, las opiniones se encuentran más repartidas, pero más de la mitad de los encuestados opinaron que el mapa tiene una valoración positiva.

Valoración del nivel de entretenimiento experimentado



Gráfica 8. Diagrama de barras con la valoración del nivel de entretenimiento

En esta gráfica podemos ver como la gran mayoría de los encuestados consideran al videojuego desarrollado muy entretenido.

¿Qué aspecto(s) valoraría mejor del juego?

Algunas de las respuestas más relevantes, nos señalan como aspectos positivos: los tiempos de carga del juego, los diferentes efectos que tiene en el jugador los distintos elementos del terreno, la velocidad de movimiento del personaje, y además se indica que las partidas resultan entretenidas y dinámicas.

¿Qué aspecto(s) valoraría peor del juego?

Algunos de los aspectos peor valorados o que se han indicado que pueden mejorar son: los controles del jugador 2, el campo puede resultar repetitivo, utilizar mejores texturas para la mejor identificación de los elementos del mapa y la dificultad en el control de movimientos precisos con el personaje. Además, se sugiere la adición de una vista general del mapa para poder ubicarse.

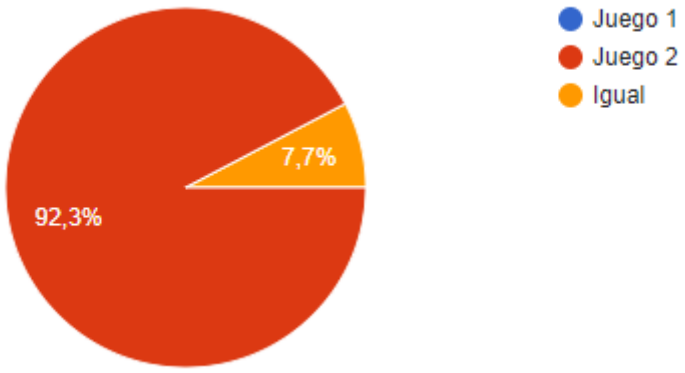
En la Tabla 1, podemos observar la media aritmética de las valoraciones obtenidas para cada aspecto encuestado sobre el Juego 2.

Tabla 1. Media aritmética de los aspectos encuestados del Juego 2

Aspecto del Juego 2	Valoración media obtenida
Comodidad de los controles	3.92
Valoración general de gráficos	4
Variedad de los mapas	3.76
Distribución de los elementos	4.23
Transición entre elementos	4.08
Valoración general de los mapas	3.69
Nivel de entretenimiento	4.23

Después se pide a los participantes que comparen distintos aspectos entre los dos juegos, el original y el desarrollado.

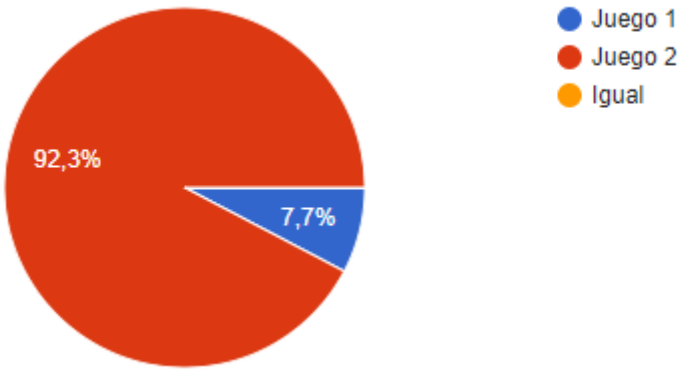
¿Qué juego considera que tiene una mejor apariencia visual?



Gráfica 9. Grafica circular de comparación de apariencia visual

Como vemos la mayoría de los encuestados considera que el juego desarrollado en este proyecto tiene mejor apariencia visual que el juego original.

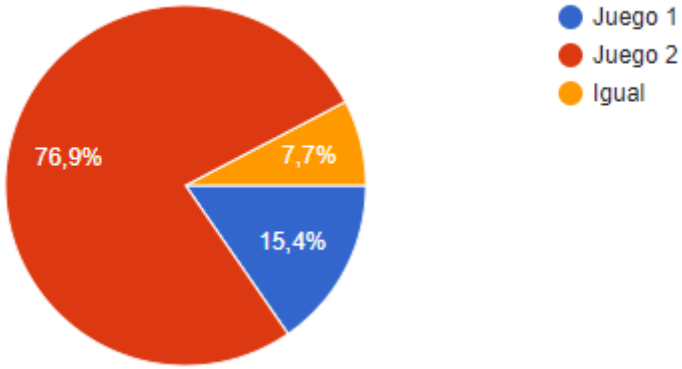
¿Qué juego le ha aportado una mejor experiencia de uso?



Gráfica 10. Grafica circular de comparación de experiencias de uso

Los resultados indican que los encuestados opinan en su mayoría que el juego desarrollado en este proyecto ofrece una mejor experiencia de uso.

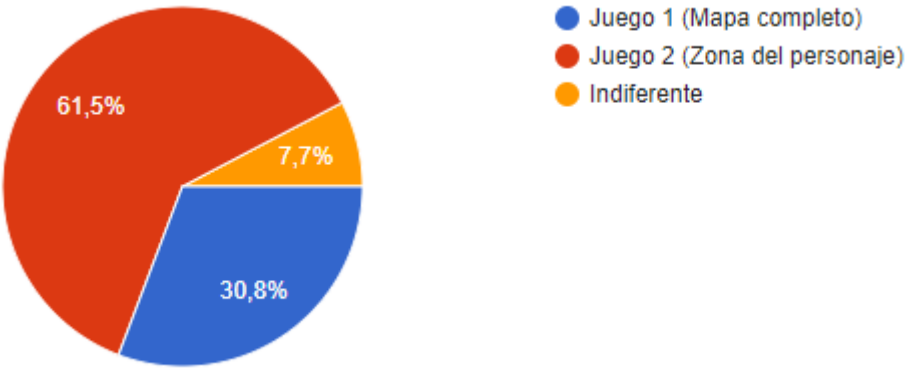
¿Qué juego le ha resultado más dinámico?



Gráfica 11. Grafica circular de comparación de dinamismo

Las opiniones se encuentran más repartidas, pero en general, los encuestados opinan que el juego desarrollado en este proyecto resulta más dinámico.

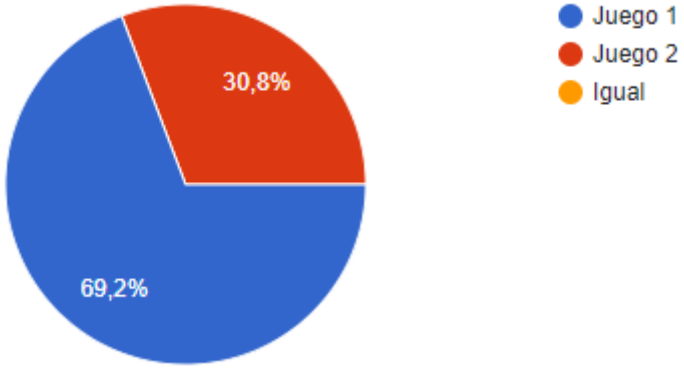
¿Qué tipo de vista de juego prefiere?



Gráfica 12. Grafica circular de comparación de tipo de vista preferida

Con esta pregunta se pretende determinar cuál es tipo de vista preferido por el usuario. Como vemos las opiniones son varias, pero más de la mitad de los encuestados prefiere la vista que le presenta la zona con los alrededores del personaje.

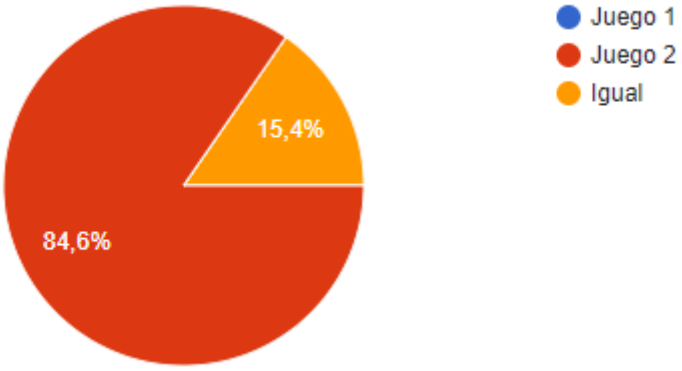
¿Qué juego considera que tiene un mayor nivel de dificultad?



Gráfica 13. Grafica circular de comparación del nivel de dificultad

En la gráfica podemos observar que a la mayoría de los encuestados le ha resultado más difícil de jugar el juego original frente al desarrollado en el proyecto.

¿Qué juego le parece más intuitivo?



Gráfica 14. Grafica circular de comparación del carácter intuitivo de los juegos

Vemos que a la gran mayoría de los encuestados el juego desarrollado en este proyecto les resulta más intuitivo de utilizar que el original.

Por último, se le da la oportunidad de añadir algún comentario general, entre los que podemos destacar: la banda sonora del juego resulta agradable, el juego desarrollado ofrece menor posibilidad de estrategia que el original y, como comentario general, el juego resulta muy entretenido.

7.4 Conclusiones de la encuesta

Tras la presentación y breve interpretación de los datos mostrados, podemos concluir que el juego en su conjunto resulta entretenido para los encuestados.

Sobre el terreno del mapa de juego, se concluye que, para un parte de los encuestados, la variedad entre los mapas generados no es muy grande. Por otro lado, se aplaude la variedad de acciones que desencadenan los elementos existentes en el mapa. También se considera que la distribución de los elementos por el mapa ha sido en general correcta y la representación del cambio de elementos ha sido buena.

Comparando con el juego original, el conjunto de encuestados, ha considerado que el videojuego desarrollado presenta una mejor apariencia visual del terreno, además de tratarse de un juego más simple y por ello más intuitivo y fácil de manejar, que les ha proporcionado en general una mejor experiencia de uso.

8 BIBLIOGRAFÍA

[1] J. Spinelli, «Generación Procedural Inteligente de Niveles Plataforma 2D utilizando Algoritmo Genéticos,» Ingeniería en Sistemas, Facultad de Ciencias Exactas, 2019.

[2] I. Fernández, «Procedurally Generated Content: la revolución de los videojuegos es ahora,» Xataka, 11 Agosto 2016. [En línea]. Available: https://www.xataka.com/videojuegos/procedurally-generated-content-la-revolucion-de-los-videojuegos-es-ahora-aunque-llevamos-40-anos-creandola. [Último acceso: Abril 2020].

[3] A. J. S. a. J. J. Bryson, «A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games,» 2014. [En línea]. Available: https://pdfs.semanticscholar.org/f69b/f76b77da89bfd7135b9c60d2b9b10fc1ac20.pdf. [Último acceso: Abril 2020].

[4] A. McHugh, «What is a Roguelike?,» Green Man Gaming, 11 Julio 2018. [En línea]. Available: https://www.greenmangaming.com/blog/what-is-a-roguelike/.

[5] D. Aversa, «Procedural Contents Generation,» Mayo 2015. [En línea]. Available: https://www.davideaversa.it/wp-content/uploads/2015/06/Procedural-Contents-Generation.pdf. [Último acceso: Abril 2020].

[6] Gallego, «La Revolución Procedural va llegar,» Vida Extra, 19 Junio 2014. [En línea]. Available: https://www.vidaextra.com/industria/la-revolucion-procedural-va-a-llegar. [Último acceso: Abril 2020].

[7] H. Alexandra, «A Look At How No Man's Sky's Procedural Generation Works,» Kotaku, 10 Octubre 2016. [En línea]. Available: https://kotaku.com/a-look-at-how-no-mans-skys-procedural-generation-works-1787928446. [Último acceso: Abril 2020].

[8] «Generación por procedimientos,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Generaci%C3%B3n_por_procedimientos#Cine. [Último acceso: Abril 2020].

[9] M. López, «Qué es el sonido procedural y cómo se utilizará en 'No Man's Sky',» Xataka, 29 Julio 2016. [En línea]. Available: https://www.xataka.com/videojuegos/que-es-el-sonido-procedural-y-como-se-utilizara-en-no-man-s-sky. [Último acceso: Abril 2020].

[10] A. Doull, «The Death of the Level Designer,» January 2008. [En línea]. Available: http://njema.weebly.com/uploads/6/3/4/5/6345478/the_death_of_the_level_designer.pdf. [Último acceso: Abril 2020].

[11] R. M. Vilar, «Videojuego basado en la generación procedimental de mundos o niveles,» Universidad de Alicante, Alicante, España, 2015.

[12] M. G. Hermida, «Estudio de Técnicas de Generación Procedural en el Mundo de los Videojuegos,» Escuela de Ingeniería de Telecomunicación, Universidad de Vigo, Vigo, 2018.

[13] «RogueBasin: Basic BSP Dungeon Generation,» Diciembre 2017. [En línea]. Available: http://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation. [Último acceso: Abril 2020].

[14] A. L. J. T. R. L. a. R. B. Noor Shaker, «PGCBook: Chapter 3: Constructive generation methods for dungeons,» [En línea]. Available: http://pcgbook.com/wp-content/uploads/chapter03.pdf. [Último acceso: Abril 2020].

[15] E. W. Weisstein, «MathWorld: Cellular Automaton,» [En línea]. Available: https://mathworld.wolfram.com/CellularAutomaton.html. [Último acceso: Abril 2020].

[16] H. M. George Kelly, A Survey of Procedural Techniques for City Generation, Dublin, Ireland: School of Informatics and Engineering, Institute of Technology.

[17] J. T. a. M. J. N. Noor Shaker, «PCGBook: Chapter 4:Fractals, noise and agents with applications to landscapes,» [En línea]. Available: http://pcgbook.com/wp-content/uploads/chapter04.pdf. [Último acceso: Abril 2020].

[18] F. Z. a. X. L. William L. Raffe, A Survey of Procedural Terrain Generation Techniques using Evolutionary Algorithms, Brisbane, Australia: School of Computer Science and Information Technology, RMIT University, 2012.

[19] J. M. Serrano, «Generación procedural de terreno,» 2015. [En línea]. Available: http://hdl.handle.net/10251/55287. [Último acceso: Abril 2020].

[20] A. S. Ibirika, Nuevo enfoque para la generación procedural, Bilbao: Universidad de Deusto, 2014.

[21] Red Blob Games, «Noise Function and Map Generation,» 31 Agosto 2013. [En línea]. Available: https://www.redblobgames.com/articles/noise/introduction.html. [Último acceso: Mayo 2020].

[22] P. Hanrahan, «Introduction to Computer Graphics: Using Noise,» Fall 1998. [En línea]. Available: http://www.graphics.stanford.edu/courses/cs248-98-fall/Assignments/Assignment4/noise.html. [Último acceso: Mayo 2020].

[23] «Perlin Noise: Part 2,» Scratchapixel 2.0, [En línea]. Available: https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2.

[24] «Lambda Lab 85,» 16 11 2014. [En línea]. Available: https://www.lanshor.com/ruido-perlin/.

[25] «Adrian's Soapbox,» 09 08 2014. [En línea]. Available: https://flafla2.github.io/2014/08/09/perlinnoise.html. [Último acceso: Mayo 2020].

[26] K. Perlin, Improving Perlin Noise, Media Research Laboratory, Dept. of Computer Science, New York University .

[27] S. Gustavson, “Simplex noise demystified,” Linköping University, Sweden, 2005.

[28] «Noise Function,» [En línea]. Available: https://www.cs.utexas.edu/~theshark/courses/cs354/lectures/cs354-21.pdf. [Último acceso: Mayo 2020].

[29] K. Perlin, «Standard for Perlin Noise». United States Patente US6867776B2, 01 08 2002.

[30] «Mine Bombers,» Wikipedia, 15 Agosto 2018. [En línea]. Available: https://fi.wikipedia.org/wiki/Mine_Bombers.

[31] «Mine Bombers Download (1996 Arcade action Game),» Old Games, [En línea]. Available: https://www.old-games.com/download/2521/mine-bombers.

[32] «Unity (motor de videojuego),» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Unity_(motor_de_videojuego).

[33] «Unity Documentation: The Main Windows,» Unity , 2016. [En línea]. Available: https://docs.unity3d.com/540/Documentation/Manual/UsingTheEditor.html.

[34] «Unity Documentation: The Animator Window,» Unity, [En línea]. Available: https://docs.unity3d.com/Manual/AnimatorWindow.html.

[35] «Microsoft Visual Studio,» Microsoft, [En línea]. Available: https://visualstudio.microsoft.com/es/vs/features/.

[36] «Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Adobe_Photoshop.

[37] L. Castillo, «Conociendo Github,» 2012. [En línea]. Available: https://conociendogithub.readthedocs.io/en/latest/data/introduccion/#:~:text=Para%20que%20sirve%3F-,%C2%B6,un%20fork%20y%20solicitar%20pulls..

[38] «Unreal Engine Documentation,» [En línea]. Available: https://docs.unrealengine.com/en-US/Gameplay/Framework/Controller/PlayerController/index.html. [Último acceso: Mayo 2020].

[39] «Unity Documentation,» [En línea]. Available: https://docs.unity3d.com/Manual/class-Tilemap.html.

[40] «Unity Learn,» [En línea]. Available: https://learn.unity.com/tutorial/using-rule-tiles#5cdedefbedbc2a0e45bdc4f1.

[41] J. T. a. M. J. N. N. Shaker, «Procedural Content Generation in Games: A Textbook and an Overview of Current Research,» Springer, 2016.

[42] «Unity Documentation,» [En línea]. Available: https://docs.unity3d.com/es/2019.4/Manual/UICanvas.html.

[43] «Unity Documentation,» [En línea]. Available: https://docs.unity3d.com/es/2018.4/Manual/class-GameObject.html.

[44] «Unity Documentation,» [En línea]. Available: https://docs.unity3d.com/es/2018.4/Manual/class-Rigidbody.html.

[45] «Unity Documentation,» [En línea]. Available: https://docs.unity3d.com/Manual/CollidersOverview.html.

[46] «Unity Documentation,» [En línea]. Available: https://docs.unity3d.com/es/2018.4/Manual/Coroutines.html.

ANEXO 1. PLANTILLA DE LA ENCUESTA

Rodee la opción más acertada para cada caso.

Datos sobre el encuestado

Género del encuestado * Femenino Masculino Otros

Edad del encuestado * Menor de 18 18 – 24 25 – 30 Mayor de 30

Frecuencia con la que juega a videojuegos *

Nunca Menos de 2 horas / semana Entre 2 y 5 horas / semana
Entre 5 y 8 horas / semana Más de 8 horas / semana

Preguntas sobre el Juego 2

Valoración de la comodidad de los controles

Muy incómodos 1 2 3 4 5 Muy cómodos

Valoración general de los gráficos

Muy baja 1 2 3 4 5 Muy alta

Valoración de la variedad de los mapas probados

Ninguna 1 2 3 4 5 Muy alta

Valoración de la distribución de los elementos (tesoros y personajes) por el mapa

Muy mala 1 2 3 4 5 Muy buena

Valoración de la transición entre los elementos que conforman el mapa (por ejemplo, de lava a tierra)

Muy mala 1 2 3 4 5 Muy alta

Valoración general de los mapas probados

Muy mala 1 2 3 4 5 Muy buena

Valoración del nivel de entretenimiento experimentado

Muy aburrido 1 2 3 4 5 Muy entretenido

¿Qué aspecto(s) valoraría mejor del juego?

_____
_____
_____

¿Qué aspecto(s) valoraría peor del juego?

_____
_____
_____

Preguntas de comparación entre ambos juegos

¿Qué juego considera que tiene una mejor apariencia visual?

Juego 1 Juego 2 Igual

¿Qué juego le ha aportado una mejor experiencia de uso?

Juego 1 Juego 2 Igual

¿Qué juego le ha resultado más dinámico?

Juego 1 Juego 2 Igual

¿Qué tipo de vista de juego prefiere?

Juego 1 (Mapa completo) Juego 2 (Zona del personaje) Indiferente

¿Qué juego considera que tiene un mayor nivel de dificultad?

Juego 1 Juego 2 Igual

¿Qué juego le parece más intuitivo?

Juego 1 Juego 2 Igual

Comentarios generales:

_____
_____
_____
_____