UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Politécnica Superior de Gandia

Algoritmo procedural en videojuego de puzles. Caso de uso real en Unity

Trabajo Fin de Grado

Grado en Tecnologías Interactivas

AUTOR/A: Sirvent Sempere, Sergi

Tutor/a: Palacio Samitier, Daniel

CURSO ACADÉMICO: 2022/2023

Resumen

Creación e implementación de un algoritmo procedural en un videojuego de puzles creado con el motor *Unity*. El objetivo de este trabajo es investigar el funcionamiento de la generación procedural y mostrar su correcto funcionamiento en un videojuego.

El juego, llamado *Ice Tiles*, se basa en cruzar un tablero de casillas de hielo, en el cual algunas son de hielo fino y otras de hielo duro, indistinguibles entre sí. El jugador podrá ayudarse de la luz solar para derretir las casillas de hielo fino, dejando así al descubierto las casillas de hielo duro. El jugador deberá memorizar el camino para que cuando acabe el periodo de luz solar, que serán unos cuantos segundos, pueda cruzar a la siguiente isla sin pisar ninguna casilla de hielo fino. Si el jugador pisa una casilla de hielo fino, este caerá al mar, se reiniciará el nivel y se penalizará en la puntuación del jugador.

El algoritmo es totalmente regulable debido a que los puzles deberán tener dificultades distintas conforme el jugador aumente de nivel. Además, los caminos de hielo duro se generarán aleatoriamente (pero siempre con una solución viable) siempre que el usuario juegue un nivel, así se conseguirá que la experiencia sea diferente cada vez que el usuario entre a un nivel. Se pretende implementar enemigos y diferentes *Power ups*, los cuáles también variarán su comportamiento y generación dependiendo de la dificultad indicada al algoritmo procedural. El juego constará de interfaz y modelados básicos para una mayor inmersión en el juego.

Por tanto, el proyecto constará de una primera parte en la que se mostrará la investigación relacionada con los algoritmos procedurales en *Unity*, con su creación y desarrollo, y por otro lado se mostrará el funcionamiento de este en un videojuego.

Palabras clave

Aleatorio, caminos, algoritmo, *Unity*, regulable, casillas, videojuego, puzle.

Abstract

Creation and implementation of a procedural algorithm in a puzzle video game created with the Unity engine. The objective of this work is to investigate the operation of procedural generation and show its correct functioning in a video game.

The game, called 'Ice Tiles', is based on crossing a board of ice tiles, in which some are made of thin ice and others are made of hard ice, indistinguishable from each other. The player will be able to use sunlight to melt the thin ice squares, thus exposing the hard ice squares. The player must memorize the path so that when the period of sunlight ends, which will be a few seconds, they can cross to the next island without stepping on any square of thin ice. If the player steps on a tile of thin ice, the ice will fall into the sea, the level will be reset and the player's score will be penalized.

The algorithm is fully adjustable because the puzzles should have different difficulties as the player levels up. Also, hard ice paths will be generated randomly (but always with a viable solution) every time the user plays a level, thus making the experience different every time the user enters a level. It is intended to implement enemies and different 'Power ups', which will also vary their behavior and generation depending on the difficulty indicated to the procedural algorithm. The game will consist of interface and basic modeling for a greater immersion in the game.

Therefore, the project will consist of a first part that will show the research related to procedural algorithms in Unity, with its creation and development, and on the other hand, it will show how it works in a video game.

Keywords

Random, paths, algorithm, Unity, adjustable, squares, video game, puzzle.

Índice

Capítulo 1. Introducción.......... 7
1.1. Contexto de generación procedural.......... 7
1.2. Motivación.......... 7
1.3. Estructura de la memoria.......... 8
1.4. Fases de realización.......... 8
1.5. Objetivos.......... 9
1.5.1. Primarios.......... 9
1.5.2. Secundarios.......... 9
1.6. Relación del proyecto con los ODS.......... 10
Capítulo 2. Marco teórico.......... 11
2.1. Unity.......... 11
2.1.1 ¿Por qué *Unity*?.......... 11
2.2. Generación procedural.......... 12
2.2.1. WFC.......... 12
2.3. Lenguaje de programación C#.......... 13
2.4. Género puzle.......... 14
2.5. Scriptable objects.......... 15
2.5.1. Prefab.......... 15
2.5.2. Scriptable object vs prefab.......... 15
2.5.3. Proceso de creación de un scriptable object.......... 16
2.6. Patrón de diseño Singleton.......... 16
2.7. Conectividad 4-adyacencia.......... 17
Capítulo 3. Diseño e implementación del algoritmo.......... 18
3.1. Herramientas para el diseño, desarrollo e implementación.......... 18
3.1.1. Visual Studio en *Unity*.......... 18
3.1.2. Control de versiones.......... 18
3.1.3. Adobe Photoshop.......... 18
3.1.4. Draw.io.......... 19
3.1.4. Excel.......... 19
3.2. Introducción procedural y Sudoku.......... 19
3.2.1. Generación de mundos.......... 19
3.2.2. Sudoku procedural.......... 20
3.3. Visión general.......... 21
3.4. Función y parámetros de los scriptable objects.......... 21
3.5. Clases C#.......... 23
3.5.1. Tile.cs.......... 23

3.5.2. GameManager.cs 23
3.5.3. WorldController.cs 24
3.5.4. TileGrid.cs 25
3.5.5. PathGenerator.cs 25
3.5.6. WFC en PathGenerator.cs 27
3.6. Otras opciones (A*) 29
Capítulo 4. Caso de uso real 30
4.1. ¿Qué es *Ice Tiles*? 30
4.2. Proceso de implementación del algoritmo 30
4.3. Mejoras y cambios en *Ice Tiles* 31
4.3.1. Enemigos 31
4.3.2. *Power ups* 31
4.3.3. Islas contiguas 31
4.3.4. Selector de niveles y progresión 32
4.3.5. Sistema de puntuación y penalizaciones 32
4.3.6. Modelados 33
4.3.7. Sistema de sonidos 34
4.3.9. Mar 34
4.3.10. Animaciones 34
4.3.11. Casillas de hielo 34
4.3.12. Interfaz de usuario 35
4.4. Aspecto final 38
Capítulo 5. *Testing* 39
5.1. Pruebas con el algoritmo 39
5.1.1. Términos a definir 39
5.1.2. Relación de *crash* por tamaño 40
5.1.3. Relación de *crash* por tamaño con WFC 40
5.1.4. *Resets* medios por tamaño 41
5.1.5. Iteraciones medias por tamaño 42
5.1.6. Iteraciones por *reset* medias por tamaño 42
5.2. Pruebas con jugadores reales 43
Capítulo 6. Conclusiones 44
6.1. Trabajo futuro 44
Referencias 45
Bibliografía complementaria 47
Anexo 1. Relación del trabajo con los ODS 48

Ilustraciones, tablas y figuras

Tabla 1 - Tabla de fases de realización del proyecto 9
Tabla 2 - Resultados test jugadores reales 43

Ilustración 1 - Generación en Beneath Apple Mannor (1978) 12
Ilustración 2 - Ejemplo WFC en imágenes 13
Ilustración 3 - Comparación de código creado en lenguajes nativos de Unity 14
Ilustración 4 - Comparación de interfaces de Tetris 15
Ilustración 5 - Tipos de patrones de diseño y ejemplos 17
Ilustración 6 - Tipos de conectividad entre píxeles 17
Ilustración 7 - Logo creado en Adobe Photoshop 19
Ilustración 8 - Prueba de generación de mundo con WFC 20
Ilustración 9 - Sudoku procedural 20
Ilustración 10 – Proceso de creación de caminos válidos 21
Ilustración 11 - Parámetros que controlan los scriptable objects 22
Ilustración 12 - Diagrama UML de la clase Tile.cs 23
Ilustración 13 - Diagrama UML de la clase GameManager.cs 24
Ilustración 14 - Diagrama UML de la clase WorldController.cs 24
Ilustración 15 - Diagrama UML de la clase TileGrid.cs 25
Ilustración 16 - Camino válido generado en tablero 4x4 en Unity 26
Ilustración 17 - Camino inválido generado en tablero 4x4 en Unity 26
Ilustración 18 - Diagrama UML de la clase PathGenerator.cs 27
Ilustración 19 - Diagrama del proceso de creación dentro de PathGenerator.cs 28
Ilustración 20 - Ejemplo visual de proceso de validación de una casilla. 29
Ilustración 21 - Ejemplo práctico de la utilización del algoritmo A* 29
Ilustración 22 - Última versión de Ice Tiles antes de este proyecto 30
Ilustración 23 - Nivel desbloqueado y nivel bloqueado 32
Ilustración 24 - Menú de selección de nivel 32
Ilustración 25 - Diferentes puntuaciones de nivel 33
Ilustración 26 - Modelado 3D asignado al jugador 33
Ilustración 27 - Modelado 3D asignado a los enemigos 33
Ilustración 28 - Modelado 3D asignado a los power ups 34
Ilustración 29 - Modelado 3D asignado a las islas contiguas 34
Ilustración 30 - Comparación de las diferentes texturas del hielo 35
Ilustración 31 - SunTimeButton mejorado 35
Ilustración 32 - Menú de pausa 35
Ilustración 33 - Menú de ajustes 36
Ilustración 34 - Menú final de nivel 36
Ilustración 35 - Botón how to play 36
Ilustración 36 - Manuales how to play 37
Ilustración 37 - Pantalla de título 37
Ilustración 38 - Botones de navegación 38
Ilustración 39 - Indicador de power up freeSunMode 38
Ilustración 40 - Aspecto final nivel 6x6 en Ice Tiles 38
Ilustración 41 - Modelo de toma de datos para pruebas: 5.1.4, 5.1.5 y 5.1.6 41

Figura 1 - Relación crash por tamaño 40
Figura 2 - Relación crash por tamaño con WFC 41
Figura 3 - Test resets medios 42

Figura 4 - Test iteraciones medias 42
Figura 5 - Test iteraciones medias por reset.. 43

Capítulo 1. Introducción

1.1. Contexto de generación procedural

Este proyecto se basa en la generación procedural, también llamada generación por procedimientos. Según el artículo “Procedurally generated content: La revolución de los videojuegos es ahora” en Xataka [1], la generación procedural está basada en funciones iterativas. Esto significa que este tipo de generación se ejecutará hasta el momento en el que cumpla sus metas. Gracias a la época tecnológica en la que nos encontramos, estos procesos serán posibles, dándonos como resultado obras perfeccionadas por la tecnología.

Se puede utilizar generación procedural en el mundo de los videojuegos desde la creación de la banda sonora hasta el modelado de ciertos elementos del juego. Tal y como declara Lee,J en su artículo publicado en MakeUseOf [2], el jugador experimentará una sensación predecible en los videojuego hechos por desarrolladores humanos. Comenta que hoy en día gracias a la generación procedural se puede conseguir que el jugador sienta impredecible el funcionamiento del videojuego, ayudando a que sea más divertido y le suponga una experiencia jugable más gratificante.

1.2. Motivación

Como se ha expresado en el apartado anterior, la generación procedural forma parte del futuro de la industria del videojuego e incluso del presente. La industria del videojuego se ha convertido en los últimos años en una de las industrias con mayores beneficios e ingresos del mundo, incluso llegando a superar a industrias muy arraigadas en la sociedad como el cine y la música (incluso de manera conjunta) tal y como expone Jorge Arias en su artículo publicado en TheObjective [3].

Por mi parte he de decir que mi amor por los videojuegos es un sentimiento que me ha acompañado desde que recibí como regalo de cumpleaños mi primera *GameBoy.* Desde ese momento no he parado de interesarme por este mundo y desde hace unos años, por la industria del videojuego. Siempre he querido saber cómo era el proceso de desarrollar un videojuego. Gracias a mi grado y a cierta tendencia autodidacta he descubierto este proceso y he de decir que me ha fascinado.

Mi principal motivación en este proyecto es poder desarrollar una herramienta, que se adecúe a la actualidad de la industria en la que deseo trabajar en un futuro, mostrada en un caso de uso real, en este caso, un videojuego de puzle desarrollado por mí.

También cabe mencionar que ser capaz realizar un proyecto desde cero de manera individual y que resulte en un proyecto que puede llegar a encajar en la industria, me motiva a seguir creciendo como desarrollador.

1.3. Estructura de la memoria

La estructura del presente documento es:

- Capítulo 1: Introducción.

En este capítulo se introduce la tecnología principal utilizada en este proyecto. A su vez, también se pone en contexto el proyecto con la industria actual, las fases del proyecto y motivación.

- Capítulo 2: Marco teórico.

En este capítulo se da a conocer las bases teóricas del proyecto, haciendo hincapié en el entorno donde se ha desarrollado el proyecto y en las tecnologías utilizadas en su desarrollo.

- Capítulo 3: Diseño e implementación del algoritmo

En este capítulo se expone las decisiones de diseño tomadas para el desarrollo del algoritmo junto a las herramientas y proceso de implementación

- Capítulo 4: Caso de uso real.

En este capítulo se explica en profundidad en qué proyecto real se implementará el algoritmo y se expone como ha cambiado desde el principio hasta la finalización del proyecto.

- Capítulo 5: *Testing*.

En este capítulo se mostrarán todas las pruebas a las que el algoritmo ha sido sometido para su correcto funcionamiento. Tanto pruebas matemáticas y de rendimiento hasta pruebas con jugadores reales.

- Capítulo 6: Conclusiones.

En este último capítulo se expone una visión y valoración general del proyecto junto a una pequeña reflexión sobre el posible trabajo futuro relacionado con este proyecto.

1.4. Fases de realización

Para el desarrollo del proyecto se han planteado 5 fases diferentes de realización. Se detallan en la *Tabla 1*:

Fases	Fecha prevista	Descripción
Investigación sobre algoritmos procedurales	Diciembre	Investigar que son exactamente, donde se utilizan
Investigación desarrollo del logaritmo	Enero	Investigar como poder desarrollar un algoritmo procedural
Desarrollo del algoritmo	Febrero y Marzo	Desarrollar mi propio algoritmo procedural adecuándolo a mi objetivo
Implementación de algoritmo en Unity	Abril y Mayo	Implementar el algoritmo procedural en el videojuego prototipo Ice Tiles
Fase de testeo y completar documentación	Junio y Julio	Testear el algoritmo y el videojuego Acabar la documentación del proyecto

Tabla 1 - Tabla de fases de realización del proyecto

Cabe destacar el nombre “*Ice Tiles*” en la fase de Implementación de algoritmo en *Unity*, este nombre es el elegido para el videojuego sobre el cuál operará el algoritmo creado en este proyecto, a su vez, “*Unity*” es el nombre del entorno en el que se desarrollará dicho videojuego. La explicación más detallada de este entorno se encuentra en el siguiente capítulo.

1.5. Objetivos

1.5.1. Primarios

El principal objetivo de este proyecto es poder crear e implementar un algoritmo procedural en un caso de uso real, en este caso un videojuego creado en *Unity*. El algoritmo será creado con tecnología procedural ya que en la industria de los videojuegos está en alza.

Este objetivo también incluye formarme y aprender sobre este tipo de tecnología, ya que antes de empezar este proyecto conocía varios juegos que estaban desarrollados con generación procedural, pero no era capaz de imaginar como estaban hechos.

1.5.2. Secundarios

Como objetivo secundario resaltaría el hecho de acompañar de una manera correcta al algoritmo. Antes de este proyecto, el videojuego *Ice Tiles* era simplemente un proyecto de *Unity* con una mecánica simple y una generación de caminos estática. Con este proyecto se pretende crear una demo jugable de este juego añadiendo como gran novedad la creación de caminos y niveles de manera procedural.

En resumen, aproximar *Ice Tiles* lo máximo posible a una demo jugable de un videojuego independiente del mercado gracias a la implementación de la generación procedural y de la mejora del entorno del proyecto.

1.6. Relación del proyecto con los ODS

En el *Anexo 1,* al final del documento, se encuentra la relación de este proyecto con los Objetivos de Desarrollo Sostenible. He realizado este apartado para profundizar sobre el ODS que siento que es el único que tiene relación con este proyecto

. Al tratarse de un proyecto centrado en una tecnología actual, con gran proyección futura, el único ODS que encaja y que pienso que tiene relación es el ODS9. Este proyecto busca crear una solución innovadora para el problema de crear puzles de manera procedural.

Capítulo 2. Marco teórico

2.1. Unity

Este apartado del marco teórico está dedicado a *Unity* [4]. Unity es un motor gráfico de juegos que fue presentado por primera vez en 2005. Desarrollado por la empresa *Unity Technologies*. Es popular ya que es gratuito y muchos estudios independientes, es decir, estudios de videojuegos con pocos recursos, lo utilizan para desarrollar sus videojuegos. *Unity* se trata de un motor multiplataforma, esto quiere decir que se pueden desarrollar en el mismo entorno videojuegos o aplicaciones para diversas plataformas.

Las plataformas son:

- Plataformas móviles
- Plataformas de escritorio
- Plataformas web
- Consolas
- Plataformas de realidad aumentada y virtual

Además de estas características, *Unity* cuenta con una plataforma llamada *Unity Asset Store* [5]. Esta plataforma da la posibilidad a los usuarios de publicar sus contenidos relacionados con *Unity*, como pueden ser modelados 3D, animaciones, código concreto, materiales… De esta manera otros usuarios tienen acceso a este contenido a cambio de dinero o de manera gratuita.

Desde 2005 *Unity* ha tenido diversas versiones, en las que se han añadido diversas funcionalidades en cada una de ellas. En este proyecto se ha utilizado la versión 2021.3.26f1. Se trata de una versión de 2021. He elegido esta versión ya que se trata de una LTS. LTS son las siglas para *Long Term Support.* Con este tipo de versiones, el desarrollador se asegura que el entorno está protegido de fallos y que cuenta con una gran variedad de manuales y tutoriales relacionados con las funcionalidades de la versión. Eligiendo una versión más antigua aseguras el soporte y el buen funcionamiento del sistema a cambio de perder funcionalidades nuevas, que tal vez no se usen en el proyecto.

Cabe destacar que *Unity* también cuenta con versiones no gratuitas [6], en las cuáles el usuario abona una suscripción mensual. Estas versiones cuentan con herramientas más avanzadas de desarrollo y sobre todo de analítica. Para este proyecto en concreto este tipo de versiones no es necesaria, ya que se trata de un proyecto pequeño y personal.

2.1.1 ¿Por qué *Unity*?

Por último quiero recalcar que he elegido *Unity* por diversas razones:

- Documentación: *Unity* es un motor que cuenta con una gran cantidad de documentación oficial sobre su funcionamiento. Esto hace que destaque por encima de grandes rivales como puede ser *Unreal* [7].

- Confianza en el entorno: *Unity* cuenta con una interfaz muy intuitiva que ayuda al desarrollador a crear de una manera simple y rápida.

- Conocimientos del lenguaje: *Unity* ofrece la posibilidad de desarrollar en C#, lenguaje de programación orientado a objetos, en el cual me siento muy cómodo.

- Motor estudiado en el grado: Gracias a que durante el grado hemos desarrollado dos proyectos con *Unity*, he tenido la posibilidad de poder consultar todos los recursos que los profesores nos facilitaron en su momento. Además de poder tener los dos proyectos desarrollados como referencia para desarrollar.

2.2. Generación procedural

Tal y como se ha expuesto en la introducción de esta memoria, la generación procedural tiene una gran importancia en el futuro y presente de la industria del videojuego. Pero ya que es la tecnología principal en este proyecto, es necesario profundizar en ella.

Según el artículo *"Procedural generation"* en Wikipedia [8], la generación procedural es un método para crear datos mediante algoritmos, siendo totalmente opuesto a la creación manual. Tal y como se describe en el artículo, los primeros usos de este tipo de generación en el mundo de los videojuegos se remontan al videojuego *Beneath Apple Manor(1978)* y a *Rogue(1980)*. Estos juegos utilizaban la generación procedural para crear mazmorras, monstruos, tesoros… De esta manera se conseguía dar al jugador sensación de aleatoriedad real.

La *Ilustración 1* muestra una de las generaciones de mazmorras que se podían encontrar en el videojuego *Beneath Apple Mannor(1978):*

Ilustración 1 - Generación en Beneath Apple Mannor (1978)

A su vez, la generación procedural puede ser aplicada de diferentes maneras, mediante diferentes procesos. Para ello, en este proyecto se ha utilizado el WFC(*Wave function collapse*).

2.2.1. WFC

El *Wave function collapse* o Colapso de función de onda [9] es un algoritmo utilizado en un principio en imágenes, pero también utilizado en generación procedural. Producirá generación procedural siempre que se le indique un patrón o una serie de reglas.

Según Wikipedia [10], el colapso de la función de onda se puede emplear como una técnica computacional utilizada en la generación de procedimientos para generar estructuras o patrones complejos y no repetitivos.

El proceso comienza con un patrón semilla pequeño, que luego se expande iterativamente seleccionando y "contrayendo" las probabilidades de los elementos vecinos hasta que se completa la estructura completa.

El algoritmo asegura que la salida resultante sea única y no repetitiva al colapsar las probabilidades de tal manera que los elementos vecinos sean siempre compatibles entre sí. Este algoritmo está basado en la mecánica quántica, pero resulta de gran utilidad en la generación procedural.

A continuación en la *Ilustración 2* se muestra el ejemplo más común de uso de esta tecnología en imágenes:

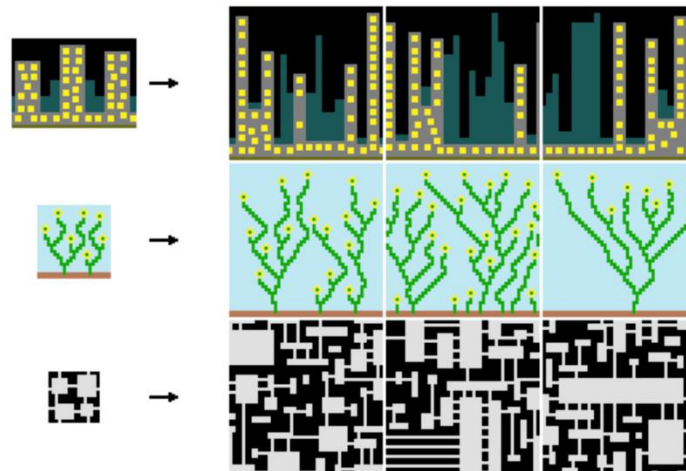

Ilustración 2 - Ejemplo WFC en imágenes

Como se puede observar, el algoritmo recibe un patrón y crea de manera aleatoria (pero bajo unas determinadas reglas impuestas por el desarrollador) una nueva imagen a partir del patrón.

En el caso concreto de este proyecto, se utilizará esta tecnología ya que el algoritmo será el encargado de crear caminos aleatorios dentro de los niveles del juego. Estos caminos no siempre serán válidos cuando se generen porque deberán seguir unas determinadas reglas. Por tanto, se tiene un objetivo que es crear un camino aleatorio y un conjunto de reglas determinadas que favorezcan a la generación correcta y deseada del camino.

2.3. Lenguaje de programación C#

C# es un lenguaje de programación orientado a objetos. Esta afirmación significa que el diseño del software de este lenguaje se organiza mediante objetos y no mediante funciones puras como podría ser un lenguaje de programación funcional.

Las aplicaciones creadas con C# son ejecutas con .NET [, esto permite que sean seguras y sólidas en su ejecución. :NET es una plataforma de código abierto que pertenece a Microsoft, la cual permite crear aplicaciones de todo tipo como: escritorio, web, móvil, etc.

Según Academia Android [11], *Unity* soporta tres grandes lenguajes de programación de manera nativa. Estos lenguajes son: C#, *Boo* y *UnityScript.* La opción recomendada de uso es C#, ya que es el lenguaje en el cual están creadas el mayor porcentaje de aplicaciones *Unity* (más de 80%) y además es el que cuenta con un mayor soporte y documentación oficial. Además, tal y como indica el artículo mencionado anteriormente, *Boo* se encuentra descontinuado y *UnityScript* está cerca de serlo.

Por estos motivos C# ha sido el lenguaje utilizado para crear el código de todo el proyecto.

A continuación, se muestra en la *Ilustración 3* una comparativa de código creado en los tres diferentes lenguajes que soporta *Unity* de manera nativa:

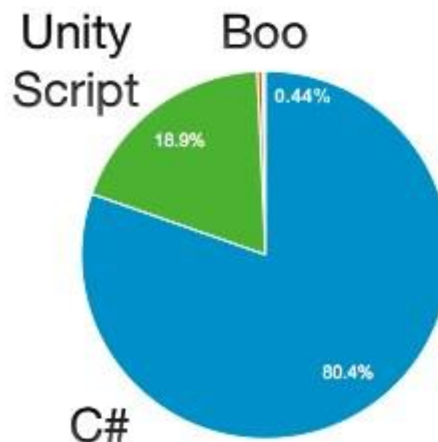

Ilustración 3 - Comparación de código creado en lenguajes nativos de Unity

2.4. Género puzle

Cabe destacar el género puzle ya que el videojuego en el cual se implementará el algoritmo procedural es de género puzle.

El género puzle destaca por querer desafiar al jugador mediante acertijos o rompecabezas en el propio juego, incluyendo ciertas veces factores de lógica e incluso razonamiento. Se podría resumir como un desafío intelectual.

El género puzle se dio a conocer en el mundo con el juego de Tetris [12]. Tetris es un juego creado por Alekséi Pázhitnov en 1984. Es un juego conocido por todos por ser desafiante y a su vez divertido.

Y por último cabe destacar que el juego donde se implementará el algoritmo desarrollado para este proyecto se trata de un juego de puzles, pero en concreto, para ejercitar la memoria. El jugador deberá memorizar los caminos que el algoritmo construya, para así poder completar el nivel sin morir. De esta manera damos un objetivo al jugador y a su vez ponemos a prueba su mente mientras se divierte.

En la *Ilustración 4* podemos ver la primera interfaz del videojuego Tetris en 1984 y a su lado la interfaz del juego en el año 2020:

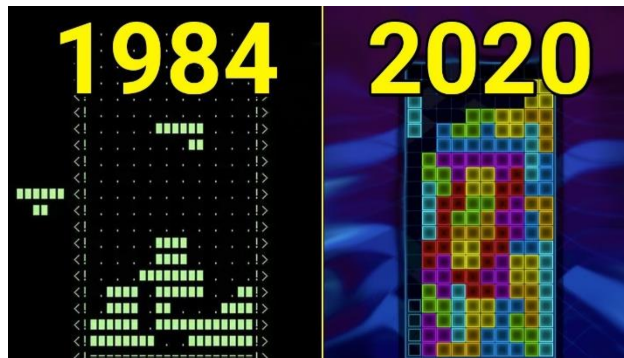

Ilustración 4 - Comparación de interfaces de Tetris

2.5. Scriptable objects

Es necesario nombrar a los *scriptable objects* en este capítulo ya que son una de las tecnologías más potentes que posee *Unity* y han sido de gran ayuda y utilidad para la realización de este proyecto.

Según el manual oficial de *Unity* [13], un *scriptable object* es un gran contenedor de datos que se guarda como archivo en su proyecto, evitando las copias de valores y reduciendo el uso de memoria innecesario. Los *scriptable objects* almacenan simplemente un tipo de datos.

2.5.1. Prefab

Para entender el funcionamiento de los *scriptable objects* cabe definir el término *Prefab* [14]. El término *Prefab* en Unity hace referencia al objeto que se almacena como un archivo del proyecto con el objetivo de ser reutilizado varias veces dentro de un mismo proyecto. Es decir, si al crear un objeto dentro de un proyecto de Unity existe la necesidad de duplicar dicho objeto en varias ocasiones, la opción más inteligente es crear un *Prefab* de ese mismo objeto. Por tanto, el *Prefab* necesitará estar siempre asociado a un objeto.

2.5.2. Scriptable object vs prefab

Los *scriptable objects* son de utilidad cuando se quiere evitar duplicar datos y cuando se necesita almacenar un tipo de datos en concreto. En cambio, los *prefabs* son de gran utilidad cuando se quiere crear un objeto a modo de plantilla y a partir de ese momento crear copias de él.

En este proyecto el algoritmo crea los niveles a partir de unos parámetros que serán especificados en el *Capítulo 3:Diseño e implementación del algoritmo* de este documento. Es cierto que cada nivel contará con elementos muy similares pero el hecho de contener parámetros diferentes para cada uno de ellos hace que el *scriptable object* sea la manera adecuada de abordar este problema.

Con esta elección se ahorra espacio en la memoria y se evita crear objetos de niveles duplicados en los que simplemente cambien ciertos parámetros del algoritmo. El entorno del nivel siempre será el mismo, pero gracias a los parámetros incluidos en los *scriptable objects* el algoritmo podrá crear caminos diferentes de una manera más rápida y sencilla.

2.5.3. Proceso de creación de un scriptable object

Cabe destacar que el proceso de creación de los *scriptable objects* es muy sencillo:

1. Se diseña un *script* de *Unity* (hoja de código C#) con los parámetros (vacíos) que queremos que tenga nuestro *scriptable object*.

2. Se crea un archivo de tipo *scriptable object* (que tendrá el nombre que le hayamos dado al *script* anterior) y se rellenarán los parámetros vacíos con los valores que le interesen al desarrollador. En este momento el desarrollador es libre de volver a crear otro *scriptable object* con los mismos parámetros que el anterior, pero con valores diferentes si es necesario.

Este es el proceso que se ha seguido durante el desarrollo del proyecto para crear los diferentes niveles que contiene el videojuego final.

2.6. Patrón de diseño Singleton

El patrón *Singleton* [15] es un patrón de diseño creacional que garantiza que tan solo exista un objeto de su tipo y proporciona un único punto de acceso a él para cualquier otro código. Este patrón facilita el acceso global desde varios puntos del código a una misma instancia, de esta manera, se convierte en una tarea sencilla acceder desde cualquier punto del código a información almacenada en ésta.

Los patrones de diseño o patrones de diseño software son herramientas que ayudan a los desarrolladores a poder organizar, mantener y mejorar la eficiencia de sus proyectos.

Existen muchos patrones de diseño, pero podemos dividirlos en tres grandes grupos [16]:

- Patrones creacionales: Proporcionan diversos mecanismos de creación de objetos. Aumentan la flexibilidad y reutilización del código.

- Patrones estructurales: Facilitan las composiciones de clases y las estructuras de objetos. El concepto de herencia es muy utilizado en este tipo de patrones.

- Patrones de comportamiento: Se ocupa de la comunicación entre objetos de la clase. Se utiliza principalmente para detectar y manipular patrones de comunicación entre objetos de una clase.

En la *Ilustración 5* se muestran diferentes ejemplos de patrones de diseño divididos por tipos. Incluyendo al patrón *Singleton* que ha sido utilizado en este proyecto.

Ilustración 5 - Tipos de patrones de diseño y ejemplos

2.7. Conectividad 4-adyacencia

El término de 4-adyacencia [17] procede del campo del tratamiento digital de imagen. Se trata de un término que tiene como objetivo separar objetos de una escena, en concreto píxeles. Existen varios tipos de conectividad, la Ilustración 6 muestra tres tipos de conectividad entre píxeles distintos, en concreto, la conectividad, expresa la proximidad entre píxeles de la imagen binaria.

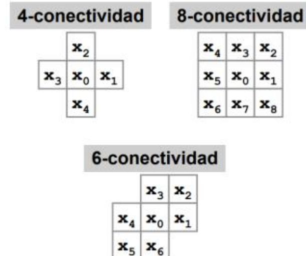

Ilustración 6 - Tipos de conectividad entre píxeles

En este proyecto, no se utiliza el concepto de píxeles, sino de casillas. Se utilizará este concepto para propagar la información y así poder generar caminos de manera procedural. En concreto el tipo de conectividad utilizado en el proyecto es 4-conectividad.

Capítulo 3. Diseño e implementación del algoritmo

3.1. Herramientas para el diseño, desarrollo e implementación

3.1.1. Visual Studio en *Unity*

El entorno de desarrollo y compilación de código *Visual Studio* [18], es una herramienta muy potente utilizada por los desarrolladores para crear todo tipo de aplicaciones. Fue creada en el 1997 con el nombre de *Visual Studio 97.*

Actualmente también se trata del entorno integrado de *Unity*, es decir, cualquier script creado en *Unity* es editado por el usuario mediante Visual Studio. Aunque no sea tan completo como las versiones de pago de Visual Studio, la versión integrada en *Unity* nos permite utilizar los *scripts* de manera sencilla e intuitiva dentro del editor.

En concreto, en este proyecto se ha utilizado este entorno junto al lenguaje C# para desarrollar el código. La versión 16.11.8 de *Visual Studio* ha sido la utilizada en este proyecto.

3.1.2. Control de versiones

Durante el desarrollo se ha utilizado como herramienta de control de versiones Git. En concreto el programa *GitHub desktop.* Aunque se trata de un proyecto individual, el control de versiones aporta seguridad y control al proyecto.

También ha sido una herramienta muy útil ya que me ha permitido trabajar en el proyecto desde diferentes dispositivos gracias a estar subido en un repositorio en *Github* [19].

Por último cabe destacar que en el año 2023 alrededor de unos 100 millones [20] de desarrolladores han utilizado Git en sus proyectos, por tanto, es una herramienta de vital importancia en el desarrollo software, por tanto es buena práctica guardar tus progresos con esta potente herramienta.

3.1.3. Adobe Photoshop

Programa de edición de imágenes creado por Adobe. El uso de este programa en este proyecto ha sido meramente visual. Se ha utilizado para transformar y crear la interfaz de usuario del videojuego en el que el algoritmo está incluido. Se han creado los menús, los botones incluso el logo del propio juego gracias al uso de este programa.

A continuación, en la *Ilustración 7* se muestra el logo del videojuego creado con *Adobe Photoshop* utilizado en este proyecto:

Ilustración 7 - Logo creado en Adobe Photoshop

3.1.4. Draw.io

La página web Draw.io ha sido utilizada durante este proyecto para la creación de cualquier esquema visual referente al proyecto. Tanto los diagramas de clase como los diagramas de flujo expuestos en este documento están realizados en esta página web gratuita.

3.1.4. Excel

Herramienta de *Microsoft Office* utilizada para monitorizar las pruebas referentes al algoritmo. En *Excel* se han recogido los diferentes datos que han ayudado a mejorar el comportamiento y el funcionamiento del proyecto.

También ha ayudado a generar varias gráficas, gracias a los datos recogidos, que han sido de gran ayuda para comprender el comportamiento del algoritmo en diferentes situaciones de manera más visual y clara. Se profundiza más sobre estos datos y pruebas en el *Capítulo 5: Testing* de este mismo documento.

3.2. Introducción procedural y Sudoku

Para la correcta utilización de la generación procedural gracias al colapso de función de onda, se han realizado dos pruebas con el objetivo de comprender el alcance y el funcionamiento de esta tecnología en concreto.

3.2.1. Generación de mundos

El repositorio *Generating Procedural Game Worlds with Wave Function Collapse* [21] nos facilita ciertas partes de código tutorizado que nos permiten aprender sobre el funcionamiento del colapso de función de onda.

En la *ilustración 8* se muestra una de las pruebas en *Unity* que realicé gracias a este código. La prueba consiste en dibujar un patrón e indicar al algoritmo ciertos parámetros para que este genere una especie de mundo a partir del patrón dibujado.

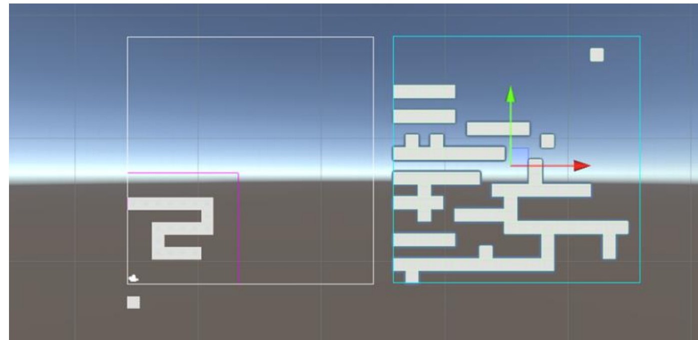

Ilustración 8 - Prueba de generación de mundo con WFC

3.2.2. Sudoku procedural

Aunque la prueba de generación de mundo expuesta en el anterior apartado fue de gran ayuda, necesitaba aprender la propagación entre casillas. Para ello, el ejercicio más común en estos casos, para aprender sobre esta característica del colapso de función de onda, es realizar un Sudoku.

En la *Ilustración 9* se muestra una pequeña demo creada por mí. En esta demo el Sudoku que se presenta funciona con colapso de función de onda, ya que al rellenar cualquiera de las casillas del juego, se actualizan las afectadas por esa decisión. Esta prueba me ayudó mucho a entender el funcionamiento de esta tecnología, y a encontrar un punto por donde unificarla con mi algoritmo.

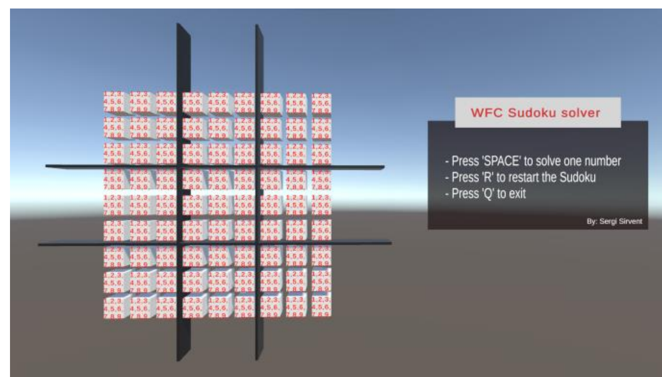

Ilustración 9 - Sudoku procedural

3.3. Visión general

A continuación, en la *Ilustración 10,* se muestra una visión general del proceso de creación de caminos indicando todos los elementos que forman parte del mismo:

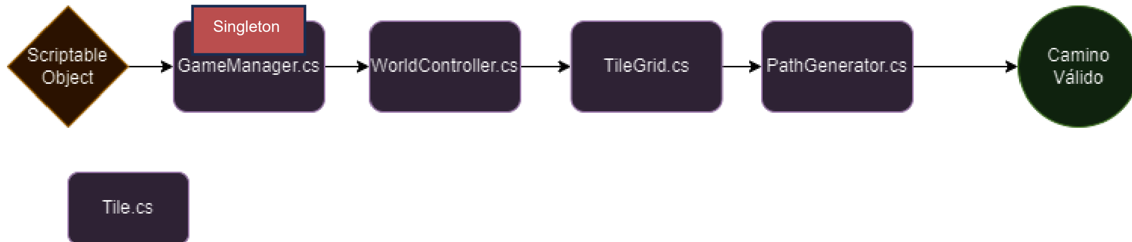

Ilustración 10 – Proceso de creación de caminos válidos

Como se puede observar el proceso está compuesto por cuatro *scripts* de lenguaje C#, es decir, cuatro clases C# (color morado), el *scriptable object* que proporcionará la información al principio del proceso (color amarillo) y finalmente el camino válido que será el resultado de todo el proceso (color verde). Cabe destacar la clase *Tile.cs* que se trata de la clase que controla los atributos de cada casilla que se genera.

3.4. Función y parámetros de los scriptable objects

Los *scriptable objects* son un tipo de archivo definido anteriormente en el documento (Capítulo 2, apartado 2.5). Contiene los parámetros elegidos por el desarrollador para crear el camino en cada nivel, su función es pasar los parámetros elegidos a la clase *GameManager.cs*.

El objetivo de este apartado es mostrar y profundizar en todos los parámetros que contienen los scriptable objects que componen los diferentes niveles, ya que de esta manera se explica a qué nivel de personalización puede llegar el algoritmo.

En la *Ilustración 11* se muestra un diagrama UML (*Unified Modeling Language*) [22] con los correspondientes parámetros de los *scriptable objects*. Se ha elegido este formato ya que se trata de un lenguaje unificado para mostrar diagramas de sistemas. Se han recalcado en rojo los parámetros que intervienen directamente en la creación del camino.

Scriptable object
- levelNumber : int
- gridEqualSize : int
- minPathLength : int
- powerUpsNumber : int
- enemyNumber : int
- sunTimeModeDuration : int
- enemyTimeToReachPos : float
- cameraPos : Vector3
- cameraRot : Vector3
- deadPenal : int
- sunTimeUsesPenal : int

Ilustración 11 - Parámetros que controlan los scriptable objects

Los parámetros de personalización que incluyen los scriptable objects son los siguientes:

- *levelNumber*: permite indicar el número de nivel.

- *gridEqualSize*: permite indicar mediante un número el tamaño del cuadrado que actuará como tablero. En la versión actual los tableros tienen forma cuadrada, por tanto, al introducir solo un número es suficiente información para la generación. Por ejemplo, si el desarrollador introduce un 4, el tablero contaría con 16 casillas (más las 2 de control). Este parámetro influye en la generación de caminos.

- *minPathLength*: permite indicar el número de casillas mínimo que debe tener el camino generado. Tal y como se ha expuesto en este mismo capítulo, para que el camino sea considerado válido, debe superar dos reglas. Este parámetro permite personalizar el valor de la regla de longitud mínima. Este parámetro influye en la generación de caminos.

- *powerUpsNumber*: permite indicar el número de *power ups* que aparecerán en el nivel.

- *enemyNumber*: permite indicar el número de enemigos que se generarán en el nivel.

- *sunTimeModeDuration*: permite personalizar la duración del *sunTimeMode*.

- *enemyTimeToReachPos*: permite controlar la velocidad de los enemigos.

- *cameraPos* y *cameraRot*: permiten personalizar la posición y la rotación de la cámara respectivamente.

- *deadPenal* y *sunTimeUsesPenal*: permite personalizar el valor de las penalizaciones que recibirá el jugador cuando cometa errores.

Los parámetros que no están relacionados con la creación de caminos hacen referencia a funcionalidades y entidades implementadas relacionadas con el caso de

uso real, es decir, forman parte del videojuego que envuelve al algoritmo. Se profundiza más sobre ellas en el *Capítulo 4: Caso de uso real* de este mismo documento.

3.5. Clases C#

3.5.1. Tile.cs

Tile.cs es la clase encargada de almacenar la información necesaria de cada casilla que conforma el tablero y el camino. La *Ilustración 12* muestra el diagrama de la clase.

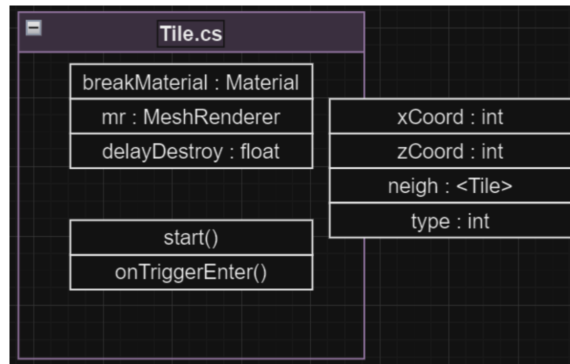

Ilustración 12 - Diagrama UML de la clase Tile.cs

3.5.2. GameManager.cs

Esta clase se trata de la instancia estática encargada de formar el *Singleton*. El *Singleton* es un patrón de diseño creacional ya definido anteriormente en este documento (*Capítulo 2: Marco teórico, apartado 2.6*). Su función es proporcionar información a todos los componentes del código de una manera sencilla y rápida, en este caso, le proporcionará la información de los parámetros elegidos a *WorldController.cs*. Cabe destacar que esta clase no solo almacena información referente a la creación del algoritmo, contiene información sobre todo el funcionamiento del proyecto. La *Ilustración 13* muestra el diagrama de la clase.

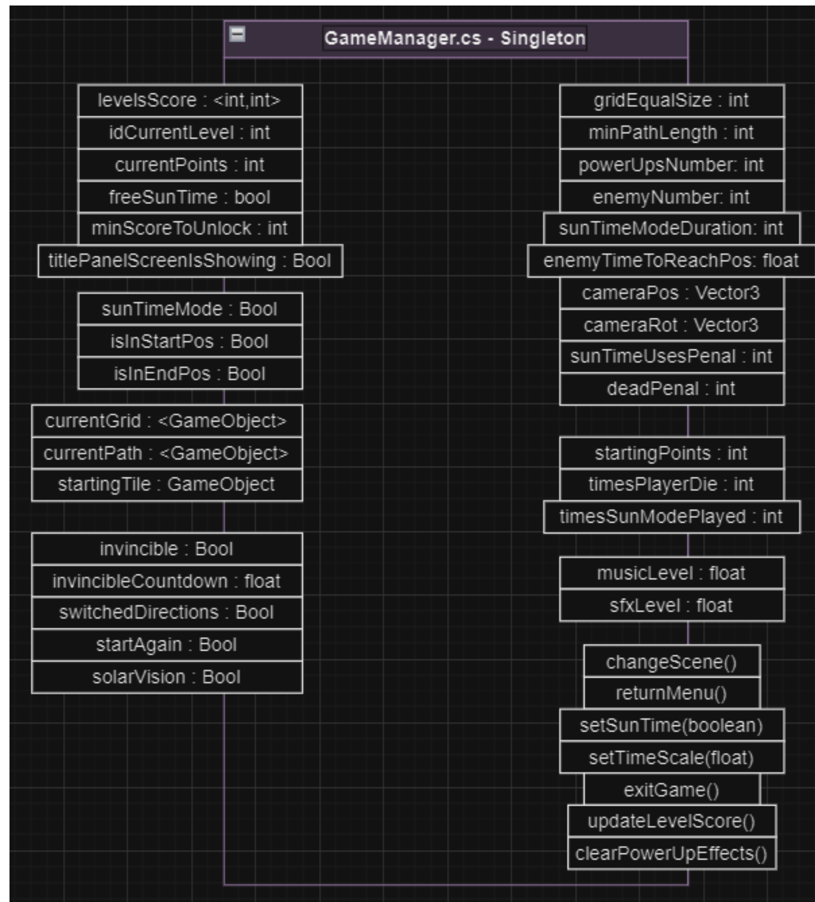

Ilustración 13 - Diagrama UML de la clase GameManager.cs

3.5.3. WorldController.cs

Clase encargada de leer la información de *GameManager.cs* y empezar el proceso de creación del tablero donde más tarde se creará el camino. También es el encargado de colocar los elementos de la escena en posición inicial. La *Ilustración 14* muestra el diagrama de la clase.

Ilustración 14 - Diagrama UML de la clase WorldController.cs

3.5.4. TileGrid.cs

Clase encargada de crear el tablero cuadrado. Se encarga de crear el tablero, pero vacío, es decir, deja el entorno listo para que la siguiente clase del proceso tenga la mayor facilidad para crear el camino. A destacar de esta clase que también añade la casilla inicial y la casilla final, lugares donde el jugador inicia y acaba el nivel respectivamente, llamadas casillas de control. La *Ilustración 15* muestra el diagrama de la clase.

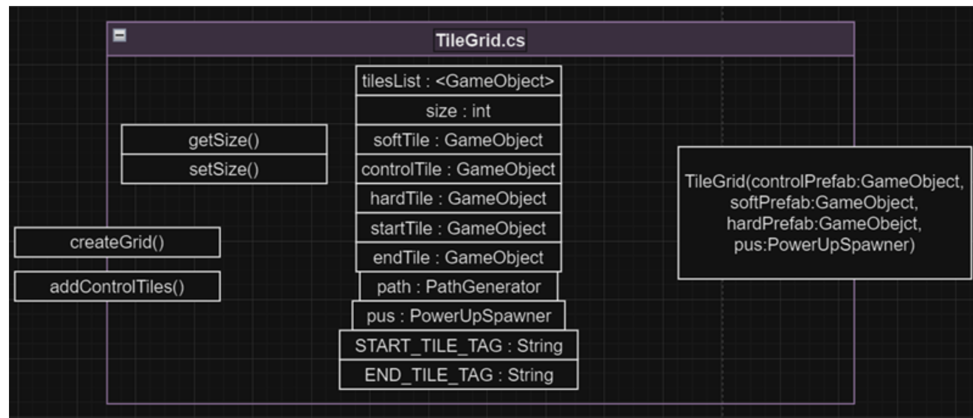

Ilustración 15 - Diagrama UML de la clase TileGrid.cs

3.5.5. PathGenerator.cs

Clase más importante del proceso. Es la encargada de generar proceduralmente un camino válido que conduzca al jugador desde la casilla inicial a la casilla final generadas anteriormente. Para que se considere un camino válido, esta clase deberá comprobar que el camino cumple dos reglas:

- Única solución: el camino comunica las casillas de control mediante un único camino. El jugador sólo tiene una opción para cruzar, no existen bifurcaciones. Gracias a la introducción del Colapso de Función de Onda, definido anteriormente en este documento (*Capítulo 2: marco teórico, apartado 2.2.1*), esta regla es controlada de manera eficiente. Esta regla es fija para todos los niveles, sin importar los demás parámetros dados por el *scriptable object*.

- Longitud mínima: mediante esta regla comprobamos que el camino creado está formado como mínimo por un número de casillas concreto. Este número variará dependiendo del nivel, por esta razón este valor viene dado en los parámetros establecidos en el *scriptable object*.

Una vez generado el camino, si éste cumple con las dos reglas, se almacenan las coordenadas del camino y se genera sobre el tablero creado en la clase *TileGrid.cs*. En el caso de que el camino incumpla alguna de estas reglas, se reiniciará el proceso desde el principio de la clase *PathGenerator.cs*, es decir, justo después de acabar con la función de la clase *TileGrid.cs*.

En la *Ilustración 16* se muestra un nivel con tamaño 4x4 creado en *Unity* después de la creación del camino válido, indicándose en color azul oscuro las casillas pertenecientes al camino generado, en azul claro las casillas del tablero que no pertenecen al camino y en rojo las casillas de control:

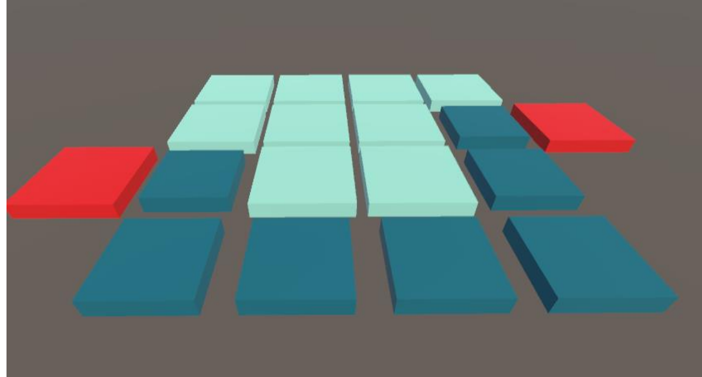

Ilustración 16 - Camino válido generado en tablero 4x4 en Unity

En la *Ilustración 17* se muestra un tablero de las mismas características que la *Ilustración 16*, pero con un camino inválido generado ya que tiene más de una solución posible para el jugador, por tanto, incumple una de las dos reglas establecidas y no se considera válido. Esta imagen fue capturada antes de la implementación de la regla de única solución, muestra el motivo por el cuál fue introducida esta segunda regla al algoritmo.

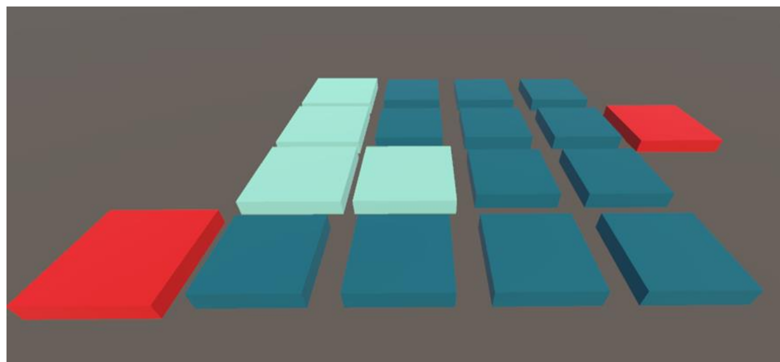

Ilustración 17 - Camino inválido generado en tablero 4x4 en Unity

La *Ilustración 18* muestra el diagrama de la clase *PathGenerator.cs*.

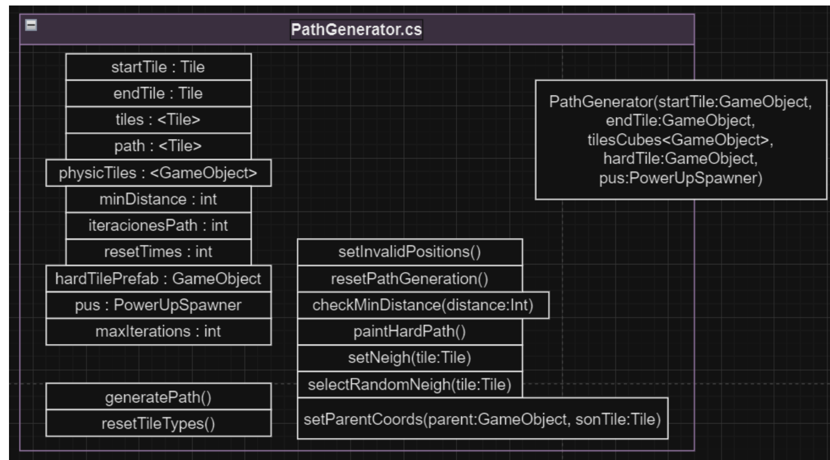

Ilustración 18 - Diagrama UML de la clase PathGenerator.cs

Como se puede ver en la *Ilustración 18,* la clase cuenta con diversos métodos privados que permitirán crear el camino, en el siguiente apartado (*3.5.6. WFC en PathGenerator.cs*) se expone el proceso de creación del camino que se produce en *PathGenerator.cs* gracias al Colapso de Función de Onda (WFC) y a la generación procedural.

3.5.6. WFC en PathGenerator.cs

Este apartado está enfocado en exponer el uso de la generación procedural en la generación de caminos de este proyecto. Como se ha expuesto en el *Capítulo 2: Marco teórico*, el Colapso de Función de Onda es utilizado en generación procedural, y, por consecuente, en este proyecto. Este proyecto centra el uso de esta tecnología en la clase *PathGenerator.cs* en concreto.

Para la correcta propagación de información mediante el colapso de función de onda (WFC) se utilizarán las casillas vecinas de cada casilla. Una casilla se considera vecina de otra si entra dentro de su 4-conectividad. El concepto de 4-conectividad ha sido definido en previamente en este documento (*Capítulo 2: Marco teórico, apartado 2.7*). Ha sido posible incorporar este concepto al proyecto ya que, al crear un tablero cuadrado con casillas cuadradas, se imita el modo en el que los píxeles son presentados en una imagen.

Con el término de conectividad aplicado al proyecto se muestra a continuación en la *Ilustración 19* un diagrama de flujo de la creación del camino de forma procedural:

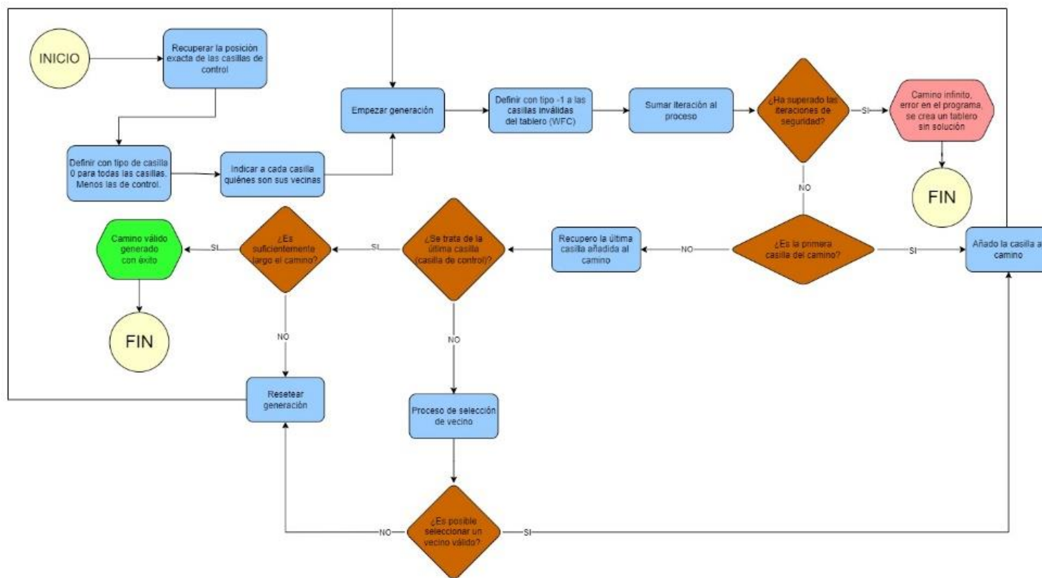

Ilustración 19 - Diagrama del proceso de creación dentro de PathGenerator.cs

Con el diagrama de la creación de caminos expuesto cabe recalcar ciertos términos para entender de una manera clara como se ha aplicado la generación procedural:

- Tipos de casilla: tal y como se indica en la *Ilustración 12* las casillas cuentan con un atributo llamado *type*. Este atributo tiene diferentes valores con diferentes significados:
  - 0: casilla válida para pertenecer al camino.
  - 1: casilla perteneciente al camino
  - -1: casilla no apta o inválida para pertenecer al camino
- ¿Cómo se detectan casillas inválidas?: pues gracias al método de la clase *Pathgenerator.cs* llamado *setInvalidPositions()*. Éste método detecta las casillas que pueden crear una segunda solución si llegan a ser parte del camino. En otras palabras, les adjudica el tipo de casilla -1 a las casillas que tengan dos o más casillas con tipo 1 (que pertenecen al camino) como vecinas. Gracias a este proceso, se descartan un gran número de casillas, y esto favorece a la selección de vecinos para continuar el camino. Este método de propagación de tipos es el que implementa el colapso de función de onda (WFC) e impide que los caminos rompan una de las reglas definidas en el apartado *3.5.5. PathGenerator.cs* de este mismo capítulo, la regla de que los caminos deben tener una única solución. En la *Ilustración 20* se puede ver de manera visual el proceso de validar o invalidar una casilla.

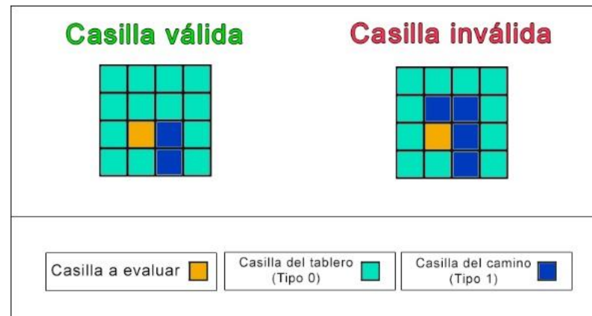

Ilustración 20 - Ejemplo visual de proceso de validación de una casilla.

- Iteraciones de seguridad: iteraciones máximas que puede soportar el entorno. Este número concreto de iteraciones se ha determinado después de los *test* aplicados al algoritmo. Se profundiza sobre ellas en el *Capítulo 5: Testing* de este mismo documento.

3.6. Otras opciones (A*)

Cabe destacar que el algoritmo de colapso de función de onda no fue la única opción posible para el desarrollo de este proyecto.

El otro gran algoritmo candidato fue el algoritmo A* [23]. El funcionamiento de este algoritmo se basa en la conexión de dos puntos de manera eficiente, basándose en la distancia entre ellos. Para evaluar esta distancia, este algoritmo utiliza costes. Se le asigna un coste a cada movimiento que el algoritmo deba realizar, de esta manera, en el momento en el que el algoritmo llegue al final del recorrido se asigna un coste a dicho recorrido. De esta manera, tras simular varios recorridos, el algoritmo escoge el recorrido con menos coste, y, por lo tanto, el más eficiente.

En la *Ilustración 18* se muestra un ejemplo práctico del uso de A*. Se asigna a las casillas con conectividad-4 un coste de 10 y a las casillas que se encuentran en diagonal un coste de $\sqrt{2}\ (1{,}4)$. Para facilitar el cálculo se han multiplicado los valores por 10. El objetivo es alcanzar el punto *B* desde el punto *A.*

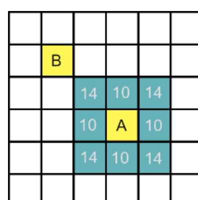

*Ilustración 21 - Ejemplo práctico de la utilización del algoritmo A**

Por tanto, ¿por qué este proyecto utiliza el colapso de función de onda? Esto es debido a que no se busca la eficiencia en este proyecto. El objetivo del algoritmo es crear caminos válidos de una distancia concreta, no óptima. Si el algoritmo genera siempre el camino óptimo posible, se pierden cantidad de caminos válidos y con este proyecto se busca hacer pensar al jugador que no va a encontrar nunca dos caminos iguales.

Capítulo 4. Caso de uso real

4.1. ¿Qué es *Ice Tiles*?

Este apartado está dedicado a introducir y exponer qué es *Ice Tiles. Ice Tiles* pretende ser un juego de puzles desarrollado en *Unity*, con cámara fija.

El jugador controla a un esquimal que debe superar una gran plataforma de hielo (dividida en casillas) para cruzar al otro lado de la isla en la que se encuentra. Para poder cruzar debe tener mucho cuidado de no pisar las capas de hielo fino, ya que, si lo hace, se caerá al frío mar.

Este juego surge de una idea propia en 2020, en época de cuarentena. La idea básica era crear un juego en el cual el jugador tuviera que memorizar un camino específico, en este caso, el camino formado por las casillas de hielo duro.

El proyecto siguió avanzando hasta el punto en el que se contaba con diez niveles construidos, un jugador con movimiento y la mecánica de que las casillas de hielo fino desapareciesen cuando se pulsaba un botón. A partir de este momento, aparté el proyecto y no volví a modificarlo.

En la *Ilustración 22* se puede ver una captura de pantalla de la última versión de *Ice Tiles* antes de que empezara a formar parte de este proyecto. En ella se puede observar el tablero, las dos casillas de control, los controles del jugador, el propio jugador que es una cápsula y por último el botón para ocultar el hielo fino y mostrar el camino.

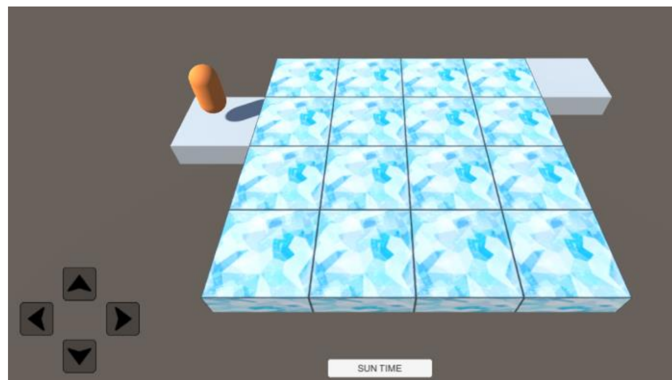

Ilustración 22 - Última versión de Ice Tiles antes de este proyecto

4.2. Proceso de implementación del algoritmo

Una vez decidido el objetivo de mi trabajo de fin de grado (crear un algoritmo procedural), pensé en alguna manera de mostrar el alcance de éste. La solución estaba en *Ice Tiles.*

Como se expone en el apartado anterior, *Ice Tiles* contenía diez niveles diferentes, pero esta versión tenía un gran problema. Dichos niveles se trataban de niveles preconstruidos, es decir, el camino de hielo duro siempre era el mismo. Aunque hubiese varios niveles distintos, con tamaños distintos, una vez el jugador superaba un nivel, no había ningún incentivo para que repitiera el nivel. Este fue el motivo para introducir una funcionalidad principal al algoritmo procedural. El algoritmo sería capaz de crear caminos diferentes cada vez que el usuario entrara a un nivel, controlado por diversos parámetros (*scriptable objects*).

De esta manera el juego adquiere la cualidad de rejugabilidad y su generación de niveles se convierte en impredecible gracias a la generación procedural.

Por tanto, gracias a retomar el proyecto abandonado *Ice Tiles*, el juego cuenta con una generación de niveles procedural, que mejora la rejugabilidad y por tanto divierte más al usuario y por otro lado, gracias a implementar el algoritmo en un proyecto de *Unity* como *Ice Tiles,* se tiene una herramienta para mostrar el potencial del algoritmo más allá de la teoría.

Cabe destacar que el proyecto de *Ice Tiles* fue totalmente descartado y sustituido por un nuevo proyecto de *Unity*. Esta nueva versión de *Ice Tiles* la cual cuenta con la generación procedural está construida sobre un proyecto totalmente nuevo, pero se considera importante nombrar al proyecto anterior ya que la idea y las bases han sido totalmente extraídas de éste.

El siguiente apartado está destinado a mostrar las diversas mejoras que ha recibido el proyecto *Ice Tiles* para poder acercar lo máximo posible al algoritmo a un caso de uso real.

4.3. Mejoras y cambios en *Ice Tiles*

4.3.1. Enemigos

Pingüinos procedentes de las islas. Su función es eliminar al jugador y evitar que llegue a su destino de una manera sencilla. Si el jugador colisiona con alguno de estos enemigos, morirá y tendrá que volver a empezar el nivel desde la casilla inicial.

4.3.2. *Power ups*

Objetos opcionales que el jugador puede recoger durante su camino. Pueden ser beneficiosos o perjudiciales. Los efectos implementados son cuatro en total, siendo dos de ellos beneficiosos y dos de ellos perjudiciales:

- *StartAgain*: teletransporta al jugador al inicio del nivel.
- *SwitchedDirections*: invierte los controles para mover al jugador.
- *Invincible*: permite al jugador sr invencible durante un periodo corto de tiempo.
- *FreeSunTime*: cuando este *power up* es recogido, deja ver al jugador el camino de hielo duro durante un periodo corto de tiempo.

4.3.3. Islas contiguas

Las casillas de control de la versión anterior pasan a ser islas de nieve con temática ártica.

4.3.4. Selector de niveles y progresión

El juego cuenta con un menú en el que el jugador seleccionará el nivel que desea jugar. No todos los niveles estarán desbloqueados al principio del juego. El jugador deberá completar los niveles con cierta puntuación si quiere desbloquear el siguiente. En la *Ilustración 23* se muestran dos botones de acceso al nivel 1 y al nivel 2, estos botones son los que se encuentran en el menú de selección de nivel. Se puede ver claramente que el nivel dos no está desbloqueado ya que se ve translúcido. Del mismo modo, en la *Ilustración 24* se muestra el menú de selección de nivel.

Ilustración 23 - Nivel desbloqueado y nivel bloqueado

Ilustración 24 - Menú de selección de nivel

4.3.5. Sistema de puntuación y penalizaciones

Sistema con el cuál se puntúa al jugador al terminar un nivel. Este mismo sistema determina si al completar un nivel la puntuación es suficiente como para desbloquear el siguiente.

Se introducen las penalizaciones por determinadas acciones del jugador. En la versión actual, se penaliza al jugador por usar más de una vez el botón de descubrir el camino de hielo duro (*sunTimeButton*) en el mismo nivel y también se penalizará al jugador por morir. Tal y como está explicado en el manual dentro del juego, el jugador deberá obtener al menos 1 pez de 3 para poder desbloquear el siguiente nivel.

En la *Ilustración 25* se muestra de nuevo el botón de acceso al nivel 1, pero esta vez en todas las formas posibles de puntuación.

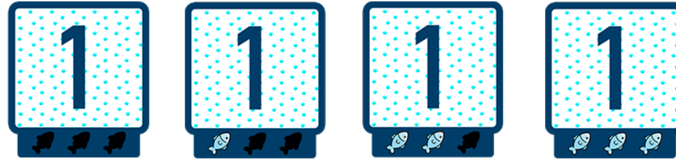

Ilustración 25 - Diferentes puntuaciones de nivel

4.3.6. Modelados

Se ha recurrido a la *asset store* de *Unity* para asignar un modelado 3D a los siguientes elementos. Estos elementos han sido los únicos del proyecto en los cuáles se ha recurrido a la *asset store* de *Unity*:

- Enemigo [24]
- Jugador [25]
- *Power Up* [26]
- Islas contiguas [27]

En las siguientes ilustraciones se exponen los diferentes modelados extraídos de la *asset store* de *Unity* que han sido implementados en el proyecto: modelado del jugador(*Ilustración 26*), modelado del enemigo(*Ilustración 27*), modelado del *power up*(*Ilustración 28*) y por último el modelado de las islas contiguas(*Ilustración 29*).

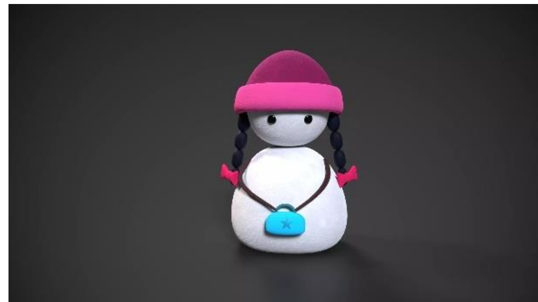

Ilustración 26 - Modelado 3D asignado al jugador

Ilustración 27 - Modelado 3D asignado a los enemigos

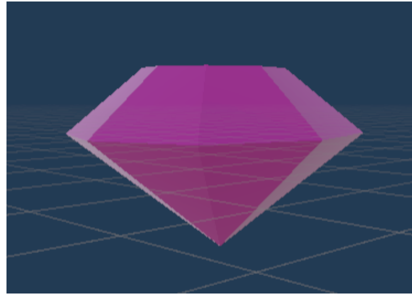

Ilustración 28 - Modelado 3D asignado a los power ups

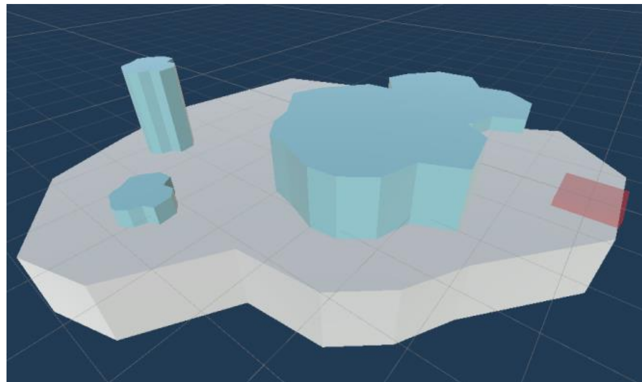

Ilustración 29 - Modelado 3D asignado a las islas contiguas

4.3.7. Sistema de sonidos

Las acciones realizadas durante el juego ahora cuentan con sonidos. También gracias a este sistema se ha implementado una música de fondo en el menú de selección de nivel. Los sonidos han sido extraídos de la página web de sonidos gratuitos *freesound* [28].

4.3.9. Mar

Las islas están situadas sobre un mar en movimiento implementado por mí. El mar utiliza dos texturas diferentes de agua y la tecnología de los *shaders* [29] de *Unity* para funcionar.

4.3.10. Animaciones

El juego ahora cuenta con animaciones básicas en ciertos elementos.

4.3.11. Casillas de hielo

El material de hielo ha sido mejorado respecto a la anterior versión. Además, cuando el jugador pisa alguna casilla de hielo fino, esta cambia de material y deja caer al jugador al mar. En la *Ilustración 30* se puede observar la nueva textura que se le ha asignado a las casillas de hielo y a su derecha, en la

misma ilustración, la textura a la que cambia la casilla de hielo fino antes de romperse cuando es pisada por el jugador.

Ilustración 30 - Comparación de las diferentes texturas del hielo

4.3.12. Interfaz de usuario

Quizá el cambio más grande respecto al proyecto anterior. Como se puede observar anteriormente en la *Ilustración 22* solo existe el botón encargado de mostrar el camino correcto al jugador. Estos elementos han sido elaborados en *Adobe Photoshop,* buscando siempre la cohesión temática entre ellos. Este cambio introduce los siguientes elementos para interfaz de usuario:

- Botón de mostrar camino (*sunTimeButton*): mejora visual al anterior botón existente. En la *Ilustración 31* se muestra el *sunTimeButton* mejorado:

Ilustración 31 - SunTimeButton mejorado

- Menú de pausa: menú que pausa el juego y permite volver al jugador al menú de selección de nivel o ajustar el sonido. En la *Ilustración 32* se muestra el menú de pausa:

Ilustración 32 - Menú de pausa

- Menú de ajustes: permite al jugador ajustar el nivel de volumen que crea conveniente tanto para los sonidos del juego como para la música ambiente. En la *Ilustración 33* se muestra el menú de ajustes:

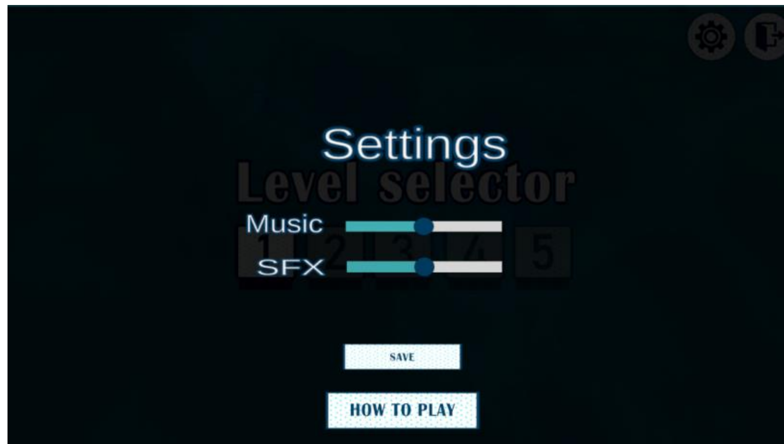

Ilustración 33 - Menú de ajustes

- Menú final de nivel: menú que es presentado al jugador cuando completa un nivel. Este menú indica al jugador la puntuación que ha obtenido en el nivel que acaba de completar y le permite volver al menú de selección de nivel. En la *Ilustración 34* se muestra el menú de final de nivel:

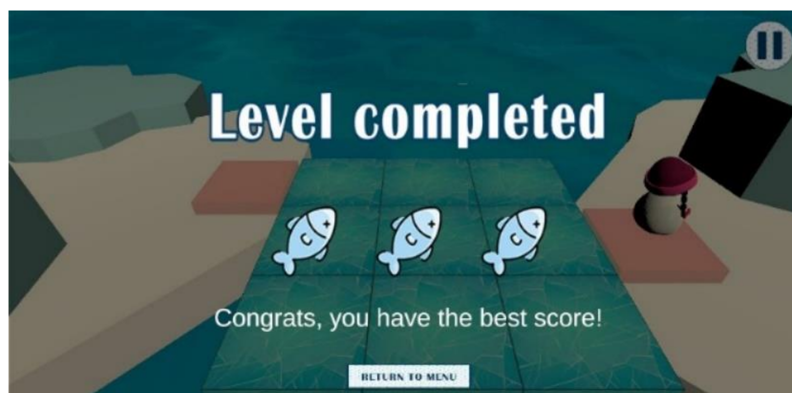

Ilustración 34 - Menú final de nivel

- Botón *how to play*: este botón permite acceder al jugador a al menú *how to play*. En la *Ilustración 35* se muestra el botón de *how to play*:

Ilustración 35 - Botón how to play

- Menú *how to play*: si el jugador accede a este menú podrá aprender mediante varios paneles, que actúan como manuales del juego. En la *Ilustración 36* se muestran los diferentes manuales que se muestran al jugador una vez presiona el botón de *how to play*:

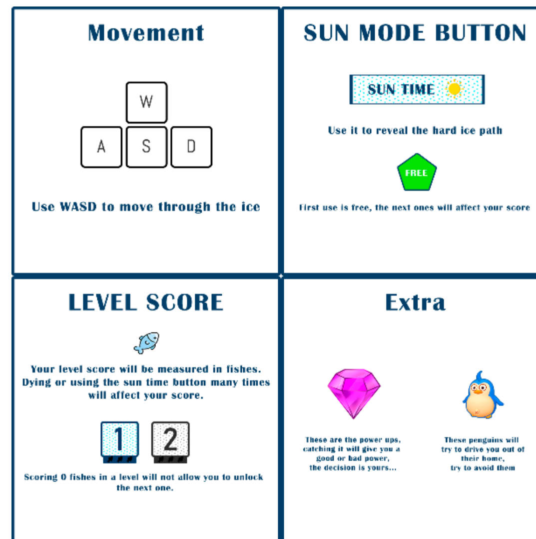

Ilustración 36 - Manuales how to play

- Pantalla de inicio: esta pantalla sirve como introducción al juego. Es lo primero que ve el jugador al iniciar el programa, se le presenta el logo del juego y se espera a que el jugador pulse cualquier botón para avanzar. En la *Ilustración 37* se muestra dicha pantalla:

Ilustración 37 - Pantalla de título

- Botones de navegación entre menús: estos botones tienen como objetivo facilitar la navegación por los menús del juego. En la *Ilustración 38* se muestran los principales de izquierda a derecha: botón de pausa, botón de salir, botón de ajustes.

Ilustración 38 - Botones de navegación

- Indicadores de *power up*: cuando el jugador recoja algún *power up* estos indicadores aparecerán en pantalla. En la *Ilustración 39* se muestra el indicador del *power up* de *freeSunMode*.

Ilustración 39 - Indicador de power up freeSunMode

4.4. Aspecto final

Este apartado está dedicado a exponer el aspecto final de que verá el jugador al jugar un nivel de *Ice Tiles.* En dicho aspecto están implementadas todas las mejoras nombradas anteriormente. A diferencia de la *Ilustración 22* la *Ilustración 40* muestra al jugador un entorno mucho mas cuidado, pulido y claro.

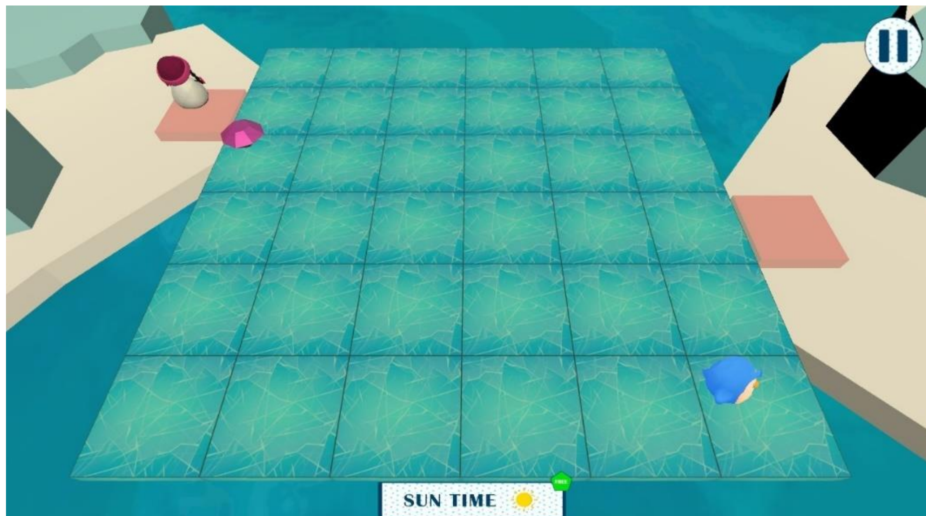

Ilustración 40 - Aspecto final nivel 6x6 en Ice Tiles

Capítulo 5. *Testing*

5.1. Pruebas con el algoritmo

El objetivo de este apartado es mostrar las pruebas que han sido realizadas respecto al funcionamiento y los límites del algoritmo.

Para demostrar que implementar el Colapso de función de onda (WFC) ha sido beneficioso para la generación, se han hecho pruebas sobre una versión antigua del algoritmo en la que los caminos se generaban aleatoriamente y, en caso de que este camino acabara en la casilla indicada, se comprobaba si el camino cumplía con las reglas establecidas (longitud mínima y única solución). A diferencia de la versión con colapso de función de onda (WFC), esta versión no descarta posibles casillas inválidas, por lo tanto, teóricamente debería tener un rendimiento peor que el de la versión definitiva que si que cuenta con esta propagación.

5.1.1. Términos a definir

Para la correcta explicación de las pruebas realizadas es necesario definir varios términos:

- *Crash*: se aplica el término *crash*, a cuando un programa informático deja de funcionar. También se puede aplicar a cuando se cierra inesperadamente o deja de contar con alguna de sus funcionalidades. Esto puede suceder por una sobrecarga de operaciones, fallos de programación, entre otras muchas razones.

- Iteración: se trata de una repetición, una reiteración. Dado que los caminos en este proyecto se crean llamando reiteradas veces a un método de la clase *PathGenerator.cs* (tal y como se define en el *Capítulo 3: Diseño e implementación del algoritmo* de este mismo documento), es necesario definir este término ya que cada llamada a este método se considera una iteración en la creación del camino.

- *Reset*: cuando el algoritmo detecta que el camino que ha creado no cumple con las reglas necesarias para ser válido, vuelve a empezar el proceso de creación desde el primer paso, esto se considera un *reset.*

- Iteraciones de seguridad: término creado para este proyecto. Cuando se empezaron las pruebas, se detectó que el programa *Unity*, soportaba aproximadamente 3700 iteraciones antes de sufrir un *crash.* Este *crash* en concreto cerraba repentinamente el programa y hacía saltar un error en pantalla cuando el programa se volvía a abrir. Por esta razón se tuvieron que definir las iteraciones de seguridad, este término hace referencia a un número concreto de iteraciones, 3500. Cuando el algoritmo supera dicho número de iteraciones y aún no ha conseguido un camino válido, genera un nivel sin solución posible e indica con un mensaje por consola que se han superado las iteraciones de seguridad. Este término ha sido de gran utilidad para detectar los límites del algoritmo.

Para realizar estas pruebas y comparaciones se ha creado una hoja de Excel. Los test realizados son los siguientes:

- Relación de tamaño con *crash*
- Relación de tamaño con *crash* con WFC
- *Reset*, iteraciones e iteraciones por *reset*
- *Reset*, iteraciones e iteraciones por *reset* con WFC

5.1.2. Relación de *crash* por tamaño

Para la realización de este test se ha indicado al algoritmo (en este caso en su versión de generación aleatoria sin WFC) que procediera con la generación de caminos válidos para diferentes tamaños, un total de 20 veces por tamaño. Cuando las iteraciones de seguridad eran superadas en cualquier intento, se contabilizaba un *crash* para ese tamaño.

En la *Figura 1* se muestra el porcentaje de *crash* en 20 muestras de manera visual en diferentes tamaños. Se puede observar que en el tamaño de 5x5 ya empiezan a haber *crashes*. También podemos concluir que esta versión del algoritmo no podrá crear tableros con dimensiones mayores a 15x15 ya que el 100% de los intentos ha sobrepasado las iteraciones de seguridad.

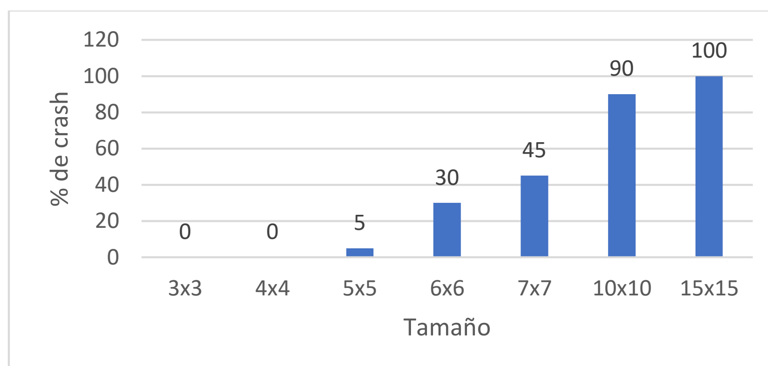

Figura 1 - Relación crash por tamaño

5.1.3. Relación de *crash* por tamaño con WFC

Este test es idéntico al del apartado anterior. La única diferencia es que el algoritmo sobre el que se hace esta prueba es el algoritmo en la última versión del proyecto, es decir, el algoritmo que está implementado en *Ice Tiles*.

En la *Figura 2* se puede observar que en este caso el tamaño límite de funcionamiento del algoritmo es en tableros de 100x100. Podemos observar que el sistema empieza a superar las iteraciones de control alguna vez a partir del tamaño de 20x20.

En el juego *Ice Tiles*, el máximo tamaño utilizado es 7x7, pero en esta prueba se han utilizado tamaños mayores a este porque el objetivo es ver el límite del algoritmo. A diferencia de la prueba anterior (mostrada en *Figura 1*), esta versión del algoritmo no dará ningún problema en *Ice Tiles*, ya que el porcentaje de *crash* en los tamaños utilizados en el juego es 0%.

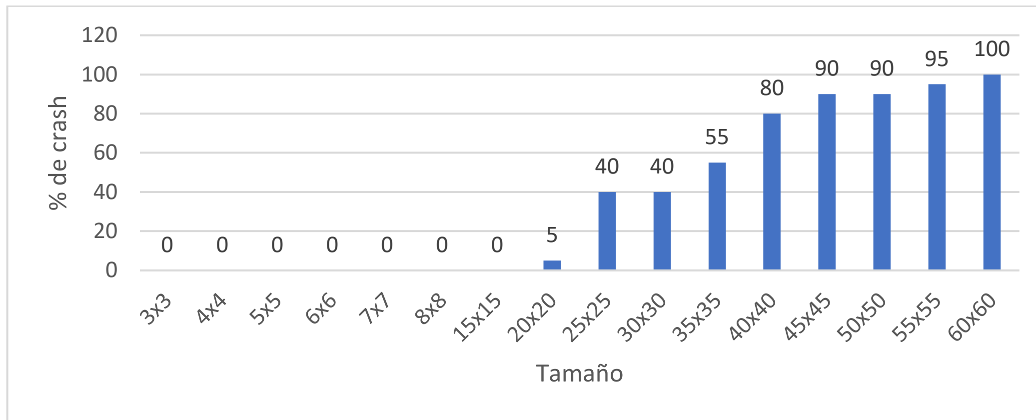

Figura 2 - Relación crash por tamaño con WFC

5.1.4. *Resets* medios por tamaño

Este test está orientado a comparar los *resets* medios que necesita el algoritmo basado en aleatoriedad contra el algoritmo basado en WFC en los diferentes tamaños de tablero que forman *Ice Tiles*. Para ello de nuevo se han tomado 20 muestras en cada uno de ellos para cada tamaño. En la *Ilustración 41* se puede observar un ejemplo de la toma de datos para esta prueba. Este modelo de toma de datos ayuda a obtener conclusiones tanto en la prueba de este apartado como en las pruebas de los apartados *5.1.5* y *5.1.6* de este mismo capítulo.

Intentos - 4x4 WFC	Resets	Iteraciones	Iteraciones por reset
1	12	130	10,83333333
2	14	146	10,42857143
3	5	58	11,6
4	2	23	11,5
5	2	26	13
6	5	53	10,6
7	6	67	11,16666667
8	9	89	9,888888889
9	23	261	11,34782609
10	16	167	10,4375
11	4	37	9,25
12	34	337	9,911764706
13	1	12	12
14	2	22	11
15	16	184	11,5
16	3	27	9
17	6	59	9,833333333
18	5	50	10
19	4	43	10,75
20	4	41	10,25
Media	8,65	91,6	10,71489422

Ilustración 41 - Modelo de toma de datos para pruebas: 5.1.4, 5.1.5 y 5.1.6

En la *Figura 3* se puede observar los resultados de esta prueba de una manera más visual. Podemos observar como todos los tableros generados con el algoritmo que cuenta con WFC tienen menos *resets* medios para crear caminos que los tableros creados con el mismo tamaño pero basándose en la aleatoriedad. Este test concuerda con la teoría de que el WFC facilita la generación. Gracias a este test podemos afirmar por ejemplo que el algoritmo con WFC realiza 6,95 *resets* de media para crear un camino válido en un tablero de 7x7 mientras que el algoritmo basado en aleatoriedad necesita 11,3 de media para crearlo en un tablero de tamaño 3x3.

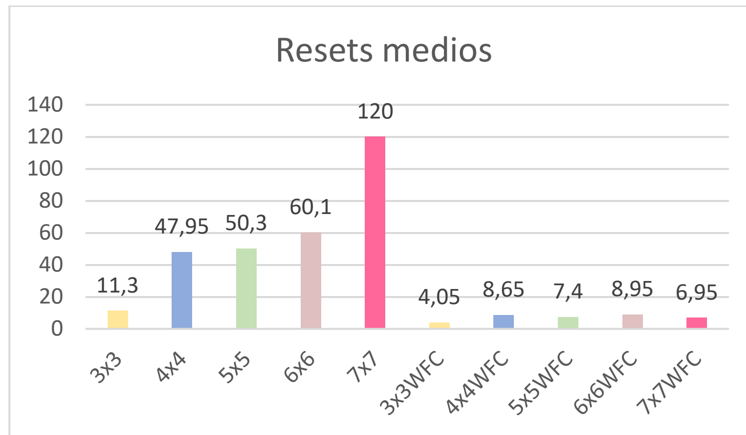

Figura 3 - Test resets medios

5.1.5. Iteraciones medias por tamaño

Para este test se han tomado los mismos datos que en el test del apartado anterior. En este caso, el objetivo de este test es comparar las iteraciones medias que necesitan las dos versiones del algoritmo para crear un camino válido en los diferentes tamaños de tablero que existen en *Ice Tiles*.

En la *Figura 4* se muestra el resultado de esta prueba de una manera más visual. Podemos observar como de nuevo las iteraciones medias necesarias para crear caminos con el algoritmo que utiliza WFC son mucho menores a las de la otra versión. Por tanto, gracias a esta prueba podemos afirmar por ejemplo que el algoritmo con WFC necesita 91,6 iteraciones para crear un camino válido en un tablero de 4x4, mientras que la versión sin WFC necesita 619,85.

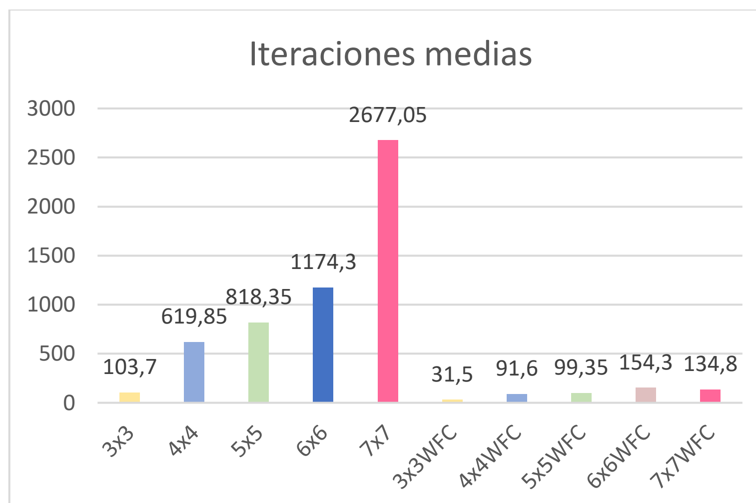

Figura 4 - Test iteraciones medias

5.1.6. Iteraciones por *reset* medias por tamaño

Este test relaciona los test expuestos en los dos apartados anteriores. Con este test se pretende comparar cuantas iteraciones medias realiza el algoritmo antes de descartar un camino, es decir, antes de hacer un *reset*.

En la *Figura 5* se muestran los resultados de esta prueba de manera más visual. Podemos observar cómo en esta ocasión la diferencia entre versiones del algoritmo no es tan diferenciada, pero gracias a esta prueba podemos afirmar que el algoritmo basado en WFC descarta de una manera más rápida los caminos inválidos.

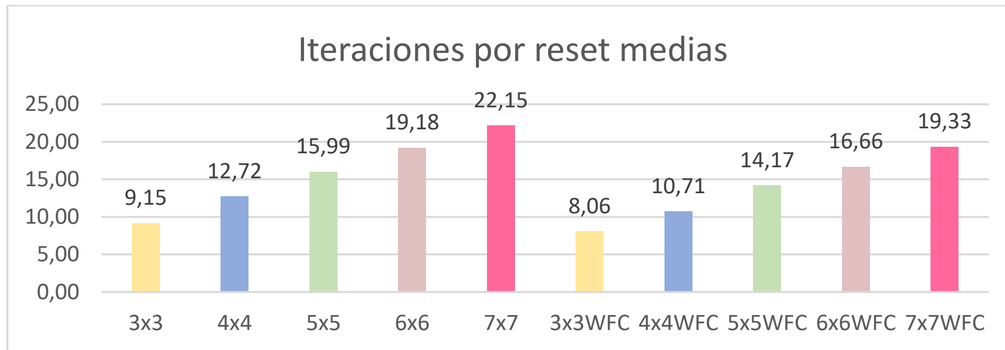

Figura 5 - Test iteraciones medias por reset

5.2. Pruebas con jugadores reales

Este apartado está dedicado a las pruebas realizadas con jugadores reales. El objetivo de estas pruebas es recibir información sobre la jugabilidad y la experiencia de los jugadores que prueben *Ice Tiles*. A diferencia de las pruebas del apartado *5.1* de este capítulo, estas pruebas no están dirigidas al funcionamiento del algoritmo, sino al funcionamiento global del proyecto, es decir, al caso de uso real, *Ice Tiles*.

Los resultados de esta prueba están expuestos en la *Tabla 2*. En esta tabla se expone el género, edad y experiencia con los videojuegos de cada sujeto. A su vez, también se expone las mejoras o los problemas detectados cuando jugaron a *Ice Tiles*. Se incluye finalmente una columna con un breve resumen de cómo se solucionaron estos problemas.

Edad	Género	Experiencia en videojuegos	Problema o mejora	Solución
54	Mujer	Sin experiencia	No familiarizado con el control WASD	Incluir el control con las flechas direccionales
22	Mujer	Jugadora ocasional	No saber el efecto de los *power ups*	Indicadores de *power ups*
19	Hombre	Jugador habitual	Falta de claridad en los menús de UI.	Cambio de color en el fondo cuando estos menús están abiertos.
52	Hombre	Jugador ocasional	Error al asignar la puntuación al acabar ciertos niveles.	Correción por código.
25	Mujer	Sin experiencia	Problemas con ciertos sonidos al recoger el *power up* de *switched directions*.	Correción por código.

Tabla 2 - Resultados test jugadores reales

Capítulo 6. Conclusiones

El principal objetivo de este proyecto era desarrollar un algoritmo procedural e implementarlo en un entorno real con el uso de *Unity*. Gracias a este documento he podido compartir el proceso que he podido seguir y la organización que he llevado a cabo para conseguir este proyecto.

Por lo tanto cabe destacar que se ha conseguido desarrollar un algoritmo que utiliza la tecnología procedural como base para la creación de caminos en un juego de puzles, todo esto desarrollado en el motor de videojuegos *Unity*.

Se podría concluir diciendo que el objetivo del proyecto ha sido completado junto con sus requisitos, aunque pienso que sigue habiendo margen de mejora. En el siguiente apartado se exponen las diferentes mejoras que pienso que pueden ayudar al proyecto a evolucionar y seguir mejorando.

6.1. Trabajo futuro

Como este proyecto termina con la implementación del algoritmo en un entorno real, el trabajo futuro pienso que principalmente debe estar enfocado a este aspecto, aunque pienso que la estructura del propio algoritmo siempre será algo a mejorar. A continuación expongo algunas propuestas como trabajo futuro relacionadas con el proyecto:

- Aumentar los parámetros del algoritmo.
- Aumentar el número tipos de *power ups* e incluir nuevos enemigos.
- Sustituir los modelados de la *asset store* por modelados propios.
- Cambiar el patrón de diseño Singleton por un patrón más estructurado.
- Mejora de los gráficos del juego.
- Publicar el juego en alguna plataforma de videojuegos (ej. Itch.io [30])

Referencias

[1] Fdez, I. (2016). Procedurally generated content: La revolución de los videojuegos es ahora (aunque llevamos 40 años. . . Xataka.

https://www.xataka.com/videojuegos/procedurally-generated-content-la-revolucion-de-los-videojuegos-es-ahora-aunque-llevamos-40-anos-creandola#:~:text=Procedural%2C%20dicho%20a%20lo%20bruto,%2Faaaaaaaa%2Faaaaaa%2F%3B%20etc%C3%A9tera (acceso Ago. 01 2023)

[2] Lee, J. (2014). How procedural generation took over the gaming industry. MUO.

https://www.makeuseof.com/tag/procedural-generation-took-gaming-industry/ (acceso Ago 01 2023)

[3] Arias, J. (2023, 4 junio). La industria de los videojuegos ya genera más ingresos que la música y el cine juntos. The Objective.

https://theobjective.com/economia/2023-06-04/videojuegos-ingresos-musica-cine/ (acceso Ago 01 2023)

[4] Wikipedia contributors. (2023). Unity (game engine). Wikipedia. https://en.wikipedia.org/wiki/Unity_(game_engine) (acceso Ago 02 2023)

[5] Unity Asset Store - The best assets for game making. (s. f.). Unity Asset Store. https://assetstore.unity.com/ (acceso Ago 02 2023)

[6] Technologies, U. (s. f.). Planes y precios. Unity.

https://unity.com/es/pricing#plans-individualsand-teams (acceso Ago 02 2023)

[7] Comunicacion. (2023, 23 marzo). Las diferencias entre Unity y Unreal. https://lboxacademy.es/blog/diferencias-entre-unity-y-unreal/ (acceso Ago 02 2023)

[8] Wikipedia contributors. (2023a). Procedural generation. Wikipedia. https://en.wikipedia.org/wiki/Procedural_generation#Video_games (acceso Ago 02 2023)

[9] The wavefunction collapse algorithm explained very clearly | Robert Heaton. (2018, 17 diciembre). Robert Heaton.

https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/ (acceso Ago 02 2023)

[10] Wikipedia contributors. (2023a). Wave function collapse. Wikipedia. https://en.wikipedia.org/wiki/Wave_function_collapse (acceso Ago 03 2023)

[11] Scripts y lenguajes de programación en Unity. (2015, 11 febrero). https://academiaandroid.com/scripts-lenguajes-programacion-unity/ (acceso Ago 03 2023)

[12] colaboradores de Wikipedia. (2023). Tetris. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Tetris (acceso Ago 03 2023)

[13] Technologies, U. (s. f.-b). Unity - Manual: ScriptableObject. https://docs.unity3d.com/Manual/class-ScriptableObject.html (acceso Ago 03 2023)

[14] Technologies, U. (s. f.-b). Unity - Manual: Prefabs. https://docs.unity3d.com/Manual/Prefabs.html (acceso Ago 03 2023)

[15] Singleton en C# / Patrones de diseño. (s. f.).

https://refactoring.guru/es/design-patterns/singleton/csharp/example (acceso Ago 03 2023)

[16] Canelo, M. M. (2023). ¿Qué son los patrones de diseño de software? Profile Software Services.

https://profile.es/blog/patrones-de-diseno-de-software/#Patrones_creacionales (acceso Ago 04 2023)

[17] Index of /~Neira/12082. (s. f.).

https://webdiis.unizar.es/~neira/12082/conectividad.pdf (acceso Ago 07 2023)

[18] colaboradores de Wikipedia. (2023b). Microsoft Visual Studio. Wikipedia, la enciclopedia libre.

https://es.wikipedia.org/wiki/Microsoft_Visual_Studio (acceso Ago 08 2023)

[20] Holcombe, J. (2023). Estadísticas clave de GitHub en 2023 (Usuarios, empleados y tendencias). Kinsta®.

https://kinsta.com/es/blog/github-estadisticas/ (acceso Ago 08 2023)

[21] Generar mundos con colapso de funciones de onda - PROCJAM tutorials. (s. f.). https://www.procjam.com/tutorials/es/wfc/ (acceso Ago 08 2023)

[22] Definición de UML | Diccionario SIG. (s. f.).

https://support.esri.com/es-es/gis-dictionary/uml#:~:text=%5Bprogramming%5D%20Acr%C3%B3nimo%20de%20Unified%20Modeling,los%20objetos%20en%20un%20sistema. (acceso Ago 04 2023)

[23] Pavgadmin. (2021). TilemAp-based A* algorithm implementation in Unity Game. Pav Creations.

https://pavcreations.com/tilemap-based-a-star-algorithm-implementation-in-unity-game/#how-a-star-algorithm-works-the-basics (acceso Ago 07 2023)

[24] Lovely Animals PACK | 3D Animals | Unity Asset Store. (2019, 8 mayo). Unity Asset Store. https://assetstore.unity.com/packages/3d/characters/animals/lovely-animals-pack-92629 (acceso Ago 08 2023)

[25] Cute Little Snow Girl (FREE) | 3D characters | Unity Asset Store. (2023, 5 enero). Unity Asset Store.

https://assetstore.unity.com/packages/3d/characters/cute-little-snow-girl-free231634#reviews (acceso Ago 08 2023)

[26] Simple Gems Ultimate Animated Customizable Pack | 3D Props | Unity Asset Store. (2018, 19 julio). Unity Asset Store.

https://assetstore.unity.com/packages/3d/props/simple-gems-ultimate-animatedcustomizable-pack-73764 (acceso Ago 08 2023)

[27] Low Poly Nature Pack RG | 3D Environments | Unity Asset Store. (2023, 11 mayo). Unity Asset Store.

https://assetstore.unity.com/packages/3d/environments/low-poly-nature-pack-rg-236313 (acceso Ago 08 2023)

[28] Freesound - Freesound. (s. f.).

https://freesound.org/ (acceso Ago 08 2023)

[29] Technologies, U. (s. f.-d). Unity - Manual: Shaders. https://docs.unity3d.com/Manual/Shaders.html (acceso Ago 08 2023)

Bibliografía complementaria

[19] Sergisirvent - Procedural_Algorithm_TFG_Repo. (s. f.). GitHub.
https://github.com/sergisirvent/Procedural_Algorithm_TFG_Repo (acceso Ago 08 2023)
[30] itch.io. (s. f.). Itch.io.
https://itch.io/ (acceso Ago 18 2023)

Anexo 1. Relación del trabajo con los ODS

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No procede
ODS 1. Fin de la pobreza				x
ODS 2. Hambre cero				x
ODS 3. Salud y bienestar				x
ODS 4. Educación de calidad				x
ODS 5. Igualdad de género				x
ODS 6. Agua limpia y saneamiento				x
ODS 7. Energía asequible y no contaminante				x
ODS 8. Trabajo decente y crecimiento económico				x
ODS 9. Industria, innovación e infraestructuras	x			
ODS 10. Reducción de las desigualdades				x
ODS 11. Ciudades y comunidades sostenibles				x
ODS 12. Producción y consumo responsables				x
ODS 13. Acción por el clima				x
ODS 14. Vida submarina				x
ODS 15. Vida de ecosistemas terrestres				x
ODS 16. Paz, justicia e instituciones sólidas				x
ODS 17. Alianzas para lograr objetivos				x