



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**Grado de Ingeniería Informática - Ingeniería de Computadores**

## **Wave Function Collapse algorithm**

Realizado por

**Antonio Rodríguez Otero**

Dirigido por Fernando Sancho Caparrini  
Dpto. Ciencias de la Computación e Inteligencia Artificial  
Junio de 2022

## **Nota informativa**

Este documento ha sufrido una modificación de su formato original para adaptarlo a las normas de entrega, lo que afecta a la visualización de los vídeos demostrativos contenidos en él. Se recomienda encarecidamente consultar el documento original que está disponible por medio del siguiente enlace:

[https://docs.google.com/document/d/1YL8OXSaxX\\_CdqIWt2XX-SNkbGqYI1vv6J0iDYVvAiQo/edit?usp=sharing](https://docs.google.com/document/d/1YL8OXSaxX_CdqIWt2XX-SNkbGqYI1vv6J0iDYVvAiQo/edit?usp=sharing)

# Agradecimientos

Me gustaría aprovechar estas líneas para recordar a aquellas personas que me han apoyado y han estado conmigo durante estos cuatro años.

En primer lugar, agradecer a mis padres todo el apoyo, el cariño y la confianza que me han dado a lo largo de mi vida.

Agradecer también a mi tutor, Fernando Sancho Caparrini, por la oportunidad que me ha brindado para llevar a cabo este TFG.

# Resumen

En este TFG se presentará una explicación e implementación del algoritmo **Wave Function Collapse**, un algoritmo que busca satisfacer un conjunto de condiciones en un determinado espacio con el fin de generar contenido (principalmente visual) de manera procedural.

El proyecto se realizará en el motor gráfico comercial Unity, y el algoritmo será codificado en C#, el lenguaje por defecto utilizado para el scripting en Unity. Por ello, se hará una breve introducción a aspectos de Unity, así como a características de necesario conocimiento de C# que van a ser usadas en algunas partes del proyecto.

La implementación realizada contempla la visualización de los diferentes pasos claves del algoritmo y permite al usuario interaccionar libremente con él. Se proporcionará un entorno de ejecución del algoritmo, aunque se dejará abierto implementar nuevos entornos de ejecución bajo distintas configuraciones.

# **Abstract**

In this TFG an explanation will be made about the Wave Function Collapse algorithm that will be implemented later. This implementation will contemplate the visualization of the different key steps in the algorithm and will provide an interaction with the user. This algorithm seeks to satisfy a set of conditions in a given space.

The project will be carried out in the commercial graphic engine Unity and will be coded in C#, which is the Unity scripting language.

An execution environment for the algorithm will be provided, although it will be left open to implement new execution environments under different configurations.

In addition, a brief introduction to aspects of Unity will be made as well as features of necessary knowledge of C# that will be used in some parts of the project.

# Índice

<b>1. Introducción</b>	<b>7</b>
1.1 Procedural no es lo mismo que aleatorio	8
1.2 Motivación del proyecto	8
1.3 Objetivos	9
1.4 Alcance	10
1.5 Estructura del documento	10
<b>2. Estado del arte</b>	<b>12</b>
<b>3. Wave Function Collapse (WFC)</b>	<b>19</b>
3.1 Constraint Satisfaction Problems (CSPs)	24
<b>4. Implementación</b>	<b>26</b>
4.1 Tecnologías empleadas	26
4.1.1 Unity	26
4.1.1.1 Fundamentos	26
4.1.1.2 Prefabs	34
4.1.1.3 Componentes básicos	35
4.1.1.4 Importación de imágenes	37
4.1.2 C#	38
4.1.2.1 System.Linq y los métodos de extensión	39
4.1.2.2 Yield keyword y IEnumerable	39
4.1.2.3 Delegados y eventos	40
4.1.2.4 Tipos implícitos y explícitos	42
4.1.2.5 Conversión de tipos	42
4.1.2.6 Virtual	42
4.1.2.7 Override	42
4.2 Estructura del proyecto	43
4.3 Backend	44
4.3.1 WFC Basics	44
4.3.2 WFC Algorithm	48
4.3.2.1 Métodos de ejecución	50
4.3.3 Operaciones	53
4.3.4 Especificaciones	56
4.4 Entrada de usuario	60
4.4.1 Scriptable Objects	60
4.4.1.1 Serialización	63
4.4.2 Custom Editor	64
4.5 Frontend	66
4.5.1 UI	69
4.5.1.1 Interrupción de la ejecución de las mecánicas principales de la escena	71
4.5.2 Creación de la visualización de la función de onda	73
4.5.3 Comunicación con el algoritmo WFC.	76
4.5.4 Interacción con los módulos	77

<b>5. Resultado</b>	<b>81</b>
<b>6. Conclusiones</b>	<b>83</b>
Unity	83
C#	83
Implementación	84
Algoritmo WFC	84
Trabajo futuro	84
<b>Referencias</b>	<b>88</b>

# 1. Introducción

Podemos definir la **generación procedural** [41] como el método de creación de datos (generalmente, de contenido multimedia) de forma algorítmica, en lugar de hacerlo de forma manual diseñando explícitamente cada paso del proceso creativo. A lo largo del tiempo se han abarcado distintas soluciones algorítmicas enfocadas a la generación procedural en diferentes ámbitos, entre los que podemos destacar la música, la botánica, la generación de texturas, etc.

Esta generación procedural es especialmente beneficiosa en la industria de los videojuegos, una de las más grandes dentro del sector multimedia. Esto se debe a la complejidad y coste del desarrollo de un videojuego, consiguiendo, gracias a la generación procedural, automatizar la generación de mapas y otros contenidos, ajustándose a la verificación de ciertas reglas generales y, por tanto, reduciendo los tiempos de desarrollo de forma drástica. Con la mejora de las capacidades técnicas de consolas y ordenadores, la generación procedural ha saltado incluso de la etapa de diseño a la etapa de ejecución (en el momento en que el contenido se prepara en la máquina del jugador). Esta práctica es tan recurrente que incluso existe un género asociado a los juegos de mazmorras generadas proceduralmente, conocido como “roguelike”. Respecto al usuario, una de las principales ventajas de la utilización de la generación procedural en este tipo de género es la eliminación de la monotonía, ya que cada nueva partida es diferente. Entre los más conocidos en este género podemos encontrar juegos como “The Binding of Isaac” o “Hades”.

En el caso de juegos como Minecraft, la generación procedural de mundos crea una sensación de variedad y exploración con la que los mundos realizados a mano no pueden competir.



Este TFG se centra en la generación procedural de entornos mediante el algoritmo WFC, una algoritmo muy reciente que ha cogido fuerza a lo largo del tiempo hasta llegar incluso a ser implementado de forma nativa en la última versión del motor gráfico Unreal Engine.

The screenshot shows the Unreal Engine 5.0 Documentation homepage. At the top, there's a navigation bar with links for 'PRODUCTS', 'SOLUTIONS', 'LEARN', and 'MORE'. On the right side of the header is a search bar with a magnifying glass icon and a 'DOWNLOAD' button. Below the header, a sidebar on the left contains links for 'Unreal Engine 5 Documentation' (including 'What's New', 'Unreal Engine 5.0 Release Notes', 'Unreal Engine 5 Migration Guide', 'Beta Features', and 'Experimental Features'), a 'Filter pages...' dropdown, and a 'Search Documentation...' search bar. The main content area features the 'UNREAL ENGINE 5' logo at the top, followed by a large image of a city skyline with the text 'Wave Function Collapse' overlaid. Below the image, there's a section titled 'Wave Function Collapse'.

## 1.1 Procedural no es lo mismo que aleatorio

A menudo se tiende a asociar la generación procedural con el concepto de aleatoriedad. Esto, que a primera instancia puede parecer lógico, no es del todo cierto. Lo primero que hay que tener en cuenta respecto a la generación es que esta toma como punto de partida una serie de reglas básicas que regirán la generación procedural. Lo segundo que debemos tener en cuenta es que los ordenadores (actuales) precisamente no son máquinas muy aleatorias, es más, son justo lo contrario. Un ordenador es solo una máquina que sigue órdenes muy precisas que ya fueron programadas por alguien. Entonces, ¿cómo son capaces de generar resultados aleatorios los ordenadores? Sencillo, simplemente no lo hacen. Los ordenadores solo aparentan tener aleatoriedad mediante lo que se conoce como **números pseudo aleatorios**, los cuales no son aleatorios, son generados a partir de un algoritmo que consigue asemejarse bastante a la aleatoriedad real (en cuanto a la función de distribución que siguen).

Este hecho tiene una serie de importantes implicaciones, y es que, al ser solo una aleatoriedad aparente, podríamos replicar una generación procedural realizada con esa “aleatoriedad”. Desde el punto de vista práctico, la generación de números pseudo aleatorios es una función matemática concreta que usa un parámetro de entrada, llamado *seed* o *semilla*, de forma que, si el generador de aleatoriedad se ejecuta dos veces bajo la misma entrada, el resultado final será el mismo. Y este hecho se extraña a todos los algoritmos que hagan uso del generador aleatorio.

Gracias a esta seed, por ejemplo, en juegos que generan mundos proceduralmente a partir de una semilla, podemos, si nos gusta un mundo, ver su semilla y con ella volver a generar el mundo exactamente igual.

## 1.2 Motivación del proyecto

La industria de los videojuegos no deja de evolucionar y, con el paso del tiempo, el número de nuevos jugadores aumenta de manera notable. El sector ha acogido a 500 millones de nuevos aficionados en los últimos tres años y, ahora, el valor de la industria mundial de los videojuegos supera los 250.000 millones de euros, según un nuevo informe de la compañía Accenture realizado en 2021 [58].



Una de las fases del desarrollo de un videojuego que más tiempo y coste requiere es el diseño de niveles y la creación de escenarios. Esto hace de la generación procedural un método que para ciertos juegos puede ser útil aplicar, consiguiendo reducir tiempo de desarrollo, abaratar costes y facilitar el trabajo a los desarrolladores.

Al mismo tiempo, también se ha apreciado un auge en el mundo de la Inteligencia Artificial, provocado por el Big Data y la automatización de tareas para las que antes no se tenían mecanismos de generación adecuados.

Estos dos factores, junto con mi pasión propia hacia la industria de los videojuegos, me han hecho embarcarme en este proyecto.

## 1.3 Objetivos

En este proyecto tenemos varios objetivos principales:

- **Obtener conocimientos sobre la generación procedural de entornos.**

Para este objetivo haremos un recorrido por los diferentes métodos y algoritmos con los que se ha explorado la generación procedural, además de ilustrar algunas herramientas que se encuentran disponibles para este fin.

- **Entender el funcionamiento y fundamentos del algoritmo WFC.**

Se realizará una abstracción del algoritmo WFC para entender sus aspectos básicos mediante el uso de un ejemplo, el juego del Sudoku. Con este ejemplo se pretende obtener un entendimiento más sencillo del algoritmo WFC a través de las similitudes que comparte con mecanismos tradicionales de resolución para este famoso juego, además de introducir la terminología que debemos conocer.

- **Realizar una implementación del algoritmo WFC.**

La idea es realizar una implementación del algoritmo WFC en base a los fundamentos que lo componen y explicar su funcionamiento.

- **Introducirnos al motor gráfico Unity.**

Este proyecto usará Unity, un motor gráfico con el cual podremos visualizar el algoritmo WFC en funcionamiento. Ha sido elegido por una serie de características que se especificarán más adelante. Se busca realizar una introducción a esta herramienta y obtener conocimientos fundamentales de la misma.

- **Aprender aspectos particulares del lenguaje C#.**

C# es el lenguaje de programación sobre el que se ha implementado el algoritmo WFC, puesto que es el lenguaje base utilizado por Unity para el scripting. Con esta imposición se busca aprender aspectos particulares del lenguaje que nos resulten de utilidad para la implementación del algoritmo WFC.

- **Mostrar el funcionamiento del algoritmo WFC para una función de onda dada.**

Una vez realizada la implementación del algoritmo WFC, nos centraremos en la ejecución del algoritmo para una **función de onda** dada. Esta función de onda especificará una serie de reglas/restricciones establecidas, con el fin de visualizar el funcionamiento del algoritmo WFC.

## 1.4 Alcance

El alcance de este proyecto llega hasta la ejecución del algoritmo WFC bajo una función de onda que contará con una serie de restricciones de adyacencia en sus módulos, que en nuestro caso serán imágenes en un entorno 2D. La ejecución del algoritmo WFC debe ser visible por parte del usuario e incluso dar la posibilidad de interacción. El colapso de la función de onda acabará dando como resultado la generación de un mapa 2D.

Los conceptos involucrados en este párrafo serán expuestos con claridad en los siguientes capítulos del documento.

## 1.5 Estructura del documento

Este documento se ha estructurado en los siguientes capítulos:

- **Capítulo 2 - Estado del arte**

En este capítulo se abordan los principales métodos y algoritmos sobre los que se ha explorado la generación procedural. A su vez, también se muestran algunas herramientas desarrolladas con ese propósito.

- **Capítulo 3 - Wave Function Collapse (WFC)**

En este capítulo se realiza una descripción del funcionamiento del algoritmo WFC y se comentan los aspectos fundamentales que lo componen junto con su terminología. Además, se hace una pequeña introducción a los CSPs.

- **Capítulo 4 - Implementación**

Este capítulo es el núcleo del documento. Tomando como punto de partida el entorno donde implementaremos el algoritmo WFC, pasaremos a explicar cada una de las fases que se han desarrollado hasta el resultado final del proyecto.

- **Capítulo 5 - Resultado**

En este capítulo se refleja el resultado final de la implementación realizada y se hace una introducción al uso de la misma.

- **Capítulo 6 - Conclusiones**

En este capítulo se comenta la experiencia con el proyecto y se expondrán distintas observaciones para trabajo a futuro.

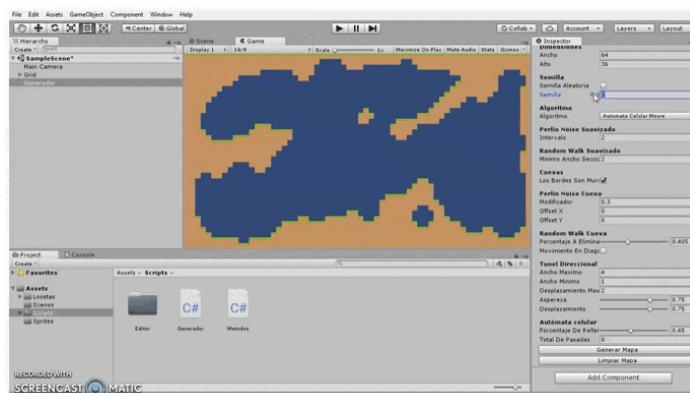
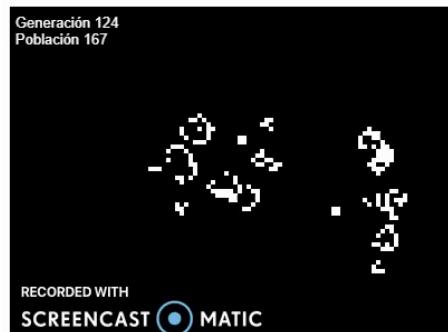
## 2. Estado del arte

La generación procedural es un ámbito en el que se lleva trabajando desde hace tiempo, siendo por lo tanto explorado con diferentes métodos y algoritmos según las necesidades requeridas. En este apartado del documento se pretende hacer un recorrido por los principales métodos y algoritmos que se han utilizado para la generación procedural, así como algunas herramientas desarrolladas con el fin de facilitar esta tarea. Entre los principales métodos y algoritmos podemos encontrar:

- **Autómatas celulares**

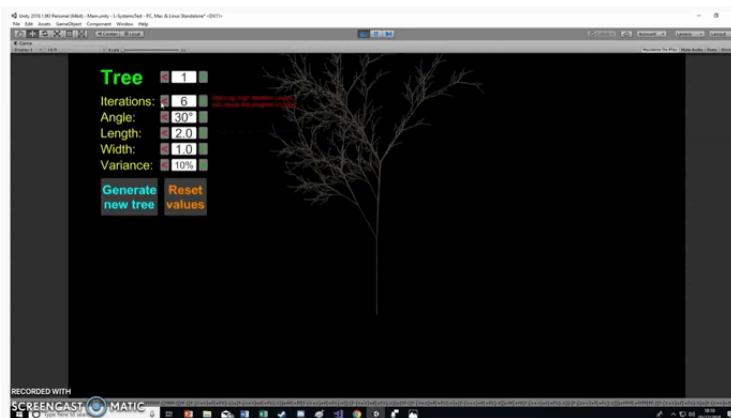
Surgen en la década de 1940 a partir de unos trabajos de John Von Neumann, quien intentaba modelar una máquina que fuera capaz de autorreplicarse.

Un autómata celular (AC) [50] es un modelo matemático y computacional para un sistema dinámico que evoluciona en pasos discretos. Es adecuado para modelar sistemas naturales que puedan ser descritos como una colección masiva de objetos simples que interactúen localmente unos con otros. Este autómata celular está compuesto por un conjunto de celdas o células que adquieren distintos estados o valores, siendo estos alterados de un instante a otro en unidades de tiempo discretas. De esta forma este conjunto de células logra una evolución según una determinada expresión matemática, que es sensible a los estados de las células vecinas y que se conoce como *regla de transición local*.



- **Sistemas-L**

Un *sistema-L* [48], o *sistema de Lindenmayer*, es una gramática formal (un conjunto de reglas y símbolos) principalmente utilizados para modelar el proceso de crecimiento de las plantas; puede modelar también la morfología de una variedad de organismos, además de poder ser utilizados para generar *fractales* autosimilares como los *Sistemas de Iteración de Funciones*. Los sistemas-L fueron introducidos y desarrollados en 1968 por el biólogo y botánico teórico húngaro Aristid Lindenmayer, de la Universidad de Utrecht.



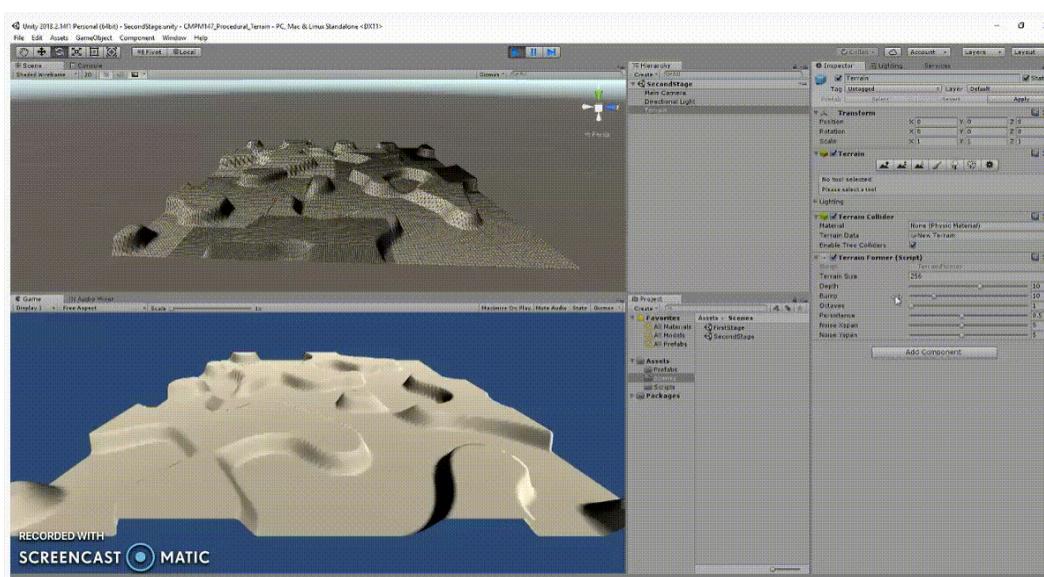
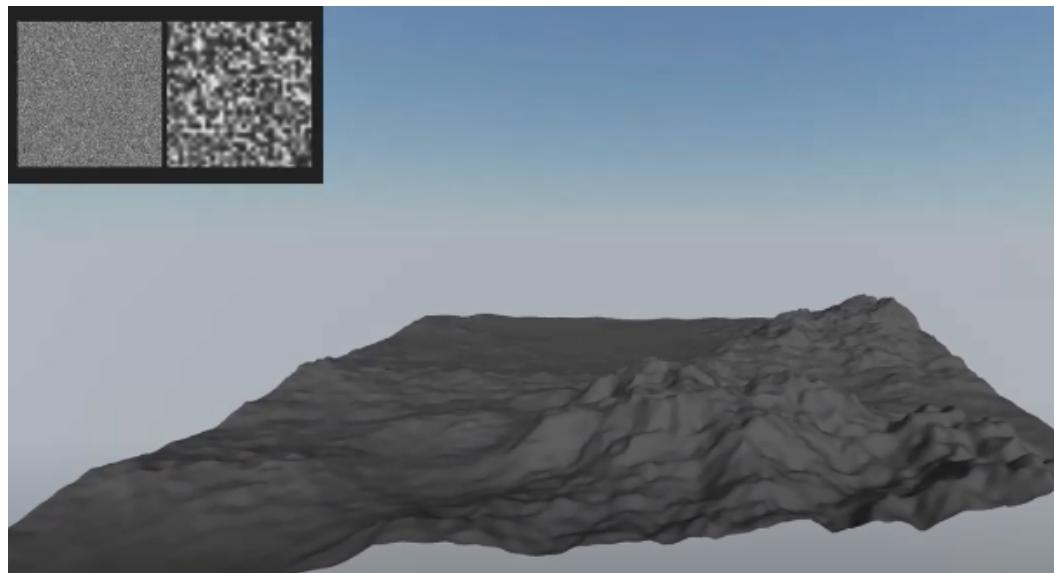
Un sistema-L está compuesto por:

- Un *alfabeto*, que es un conjunto de símbolos que contiene elementos que pueden ser reemplazados (*variables*).
- Un conjunto de símbolos que contiene elementos que se mantienen fijos (*constantes*).
- Una cadena de símbolos del alfabeto que definen el estado inicial del sistema (*inicio* o *axioma*).

- Un *conjunto de reglas o producciones* que definen la forma en la que las variables pueden ser reemplazadas por combinaciones de constantes y otras variables.

- **Perlin noise algorithm**

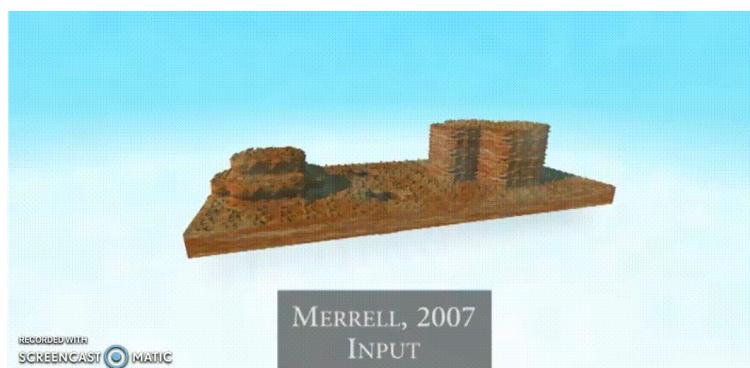
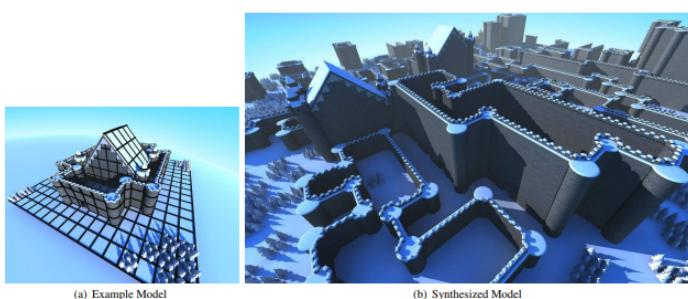
El ruido de Perlin es un popular algoritmo de generación procedural inventado por Ken Perlin. Se puede usar para generar elementos como texturas y terrenos. Se implementa normalmente como una función de dos, tres o cuatro dimensiones, pero se puede definir para cualquier número de dimensiones. Una implementación generalmente implica tres pasos: definir una cuadrícula de vectores de gradiente aleatorios, calcular el producto escalar entre los vectores de gradiente y sus compensaciones, y la interpolación entre estos valores.



En caso de interés en este algoritmo, existen diferentes artículos [42] donde se realiza una implementación detallada del ruido de Perlin.

- **Model Synthesis algorithm**

Model Synthesis [46] es una técnica para generar formas 2D y 3D complejas inspirada en la síntesis de texturas [52], a su vez siendo implementada de nuevo por el algoritmo WFC.



De hecho, el algoritmo WFC, que veremos en profundidad posteriormente en este documento, es tan similar a este método que solo encontramos dos diferencias entre ellos. Estas son:

- El orden de elección de las celdas.

En el caso del algoritmo WFC, la elección de las celdas se basa en la menor entropía, mientras que Model Synthesis se basa en el orden de línea de exploración.

- WFC trabaja con el modelo completo.

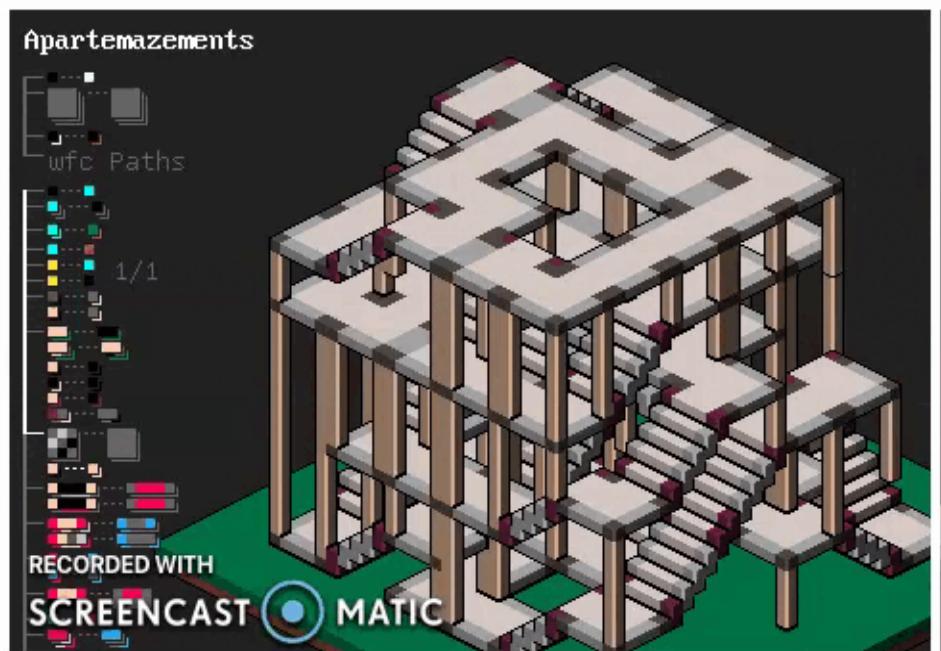
Este es un factor que se encuentra relacionado con la complejidad del problema que resuelven ambos algoritmos y que se explica posteriormente en este mismo documento.

Lo que se quiere reflejar con esta diferencia es que Model Synthesis subdivide el modelo con el que queremos trabajar en pequeños trozos. Esto nos permite obtener mejores resultados cuando los modelos con los que trabajamos son de mayor tamaño y, además, evitar contradicciones.

Como se puede ver, existen múltiples aproximaciones a la generación procedural, siendo aquí comentadas solo algunas de ellas. Al mismo tiempo se han desarrollado distintas herramientas con el objetivo de llevar a cabo generación procedural. Entre estas herramientas podemos encontrar:

- **MarkovJunior**

MarkovJunior (MJ) [54] es un lenguaje de programación probabilístico en el que los programas son combinaciones de reglas de reescritura y la inferencia se realiza a través de la propagación de restricciones. MarkovJunior lleva el nombre del matemático Andrey Andreyevich Markov, quien definió y estudió lo que ahora se conoce como algoritmos de Markov.

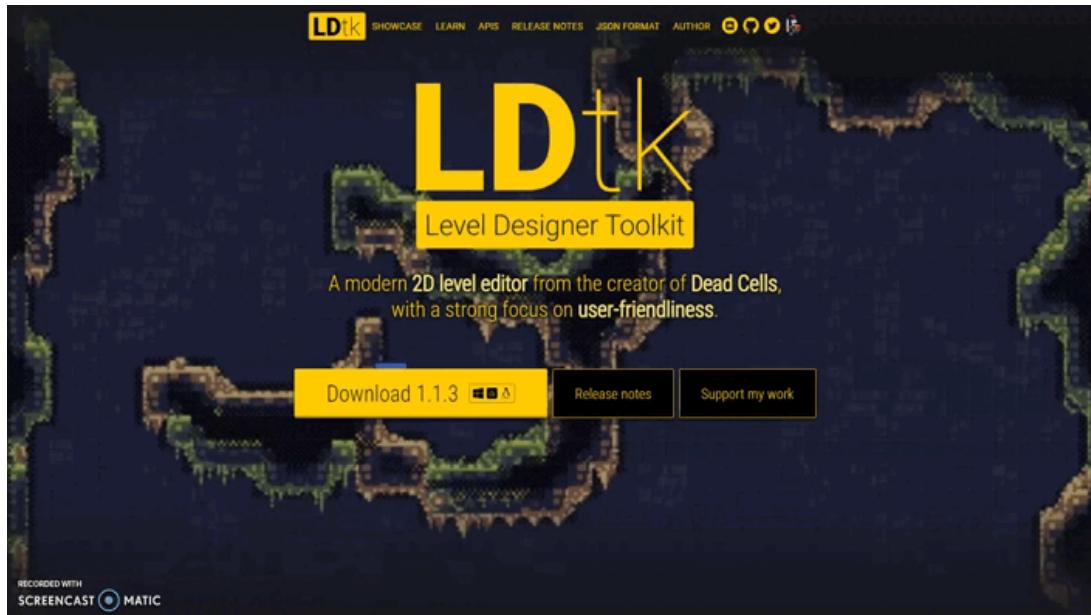


En cada paso de ejecución, el intérprete MJ encuentra la primera regla en la lista que tiene una coincidencia en la cuadrícula, encuentra todas las coincidencias para esa regla y aplica esa regla para una coincidencia aleatoria.

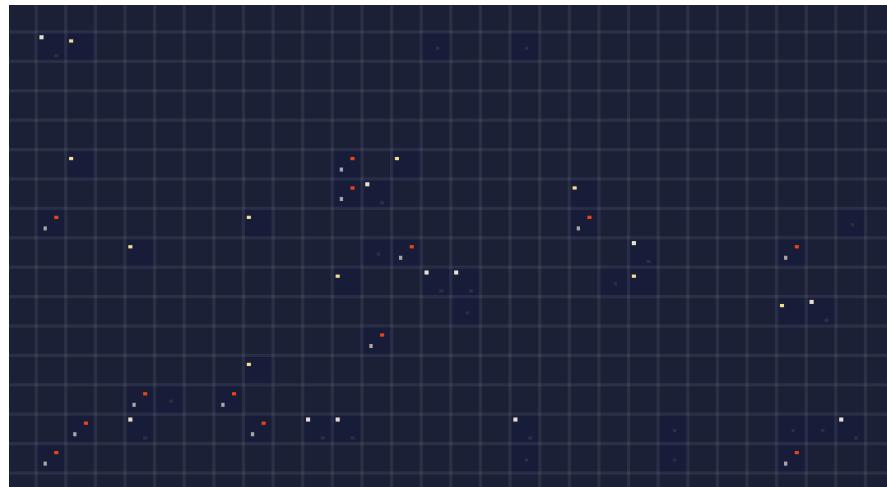


- **Level Designer Toolkit (LDtk)**

LDtk [55] es un editor de niveles 2D centrado en el fácil uso por parte del usuario y soportado en diferentes plataformas y lenguajes.



Una de las características principales que posee esta herramienta es la que denominan “Auto-rendering”, que nos permite establecer reglas simples desde un editor visual para que sean tomadas en cuenta en la generación del nivel.



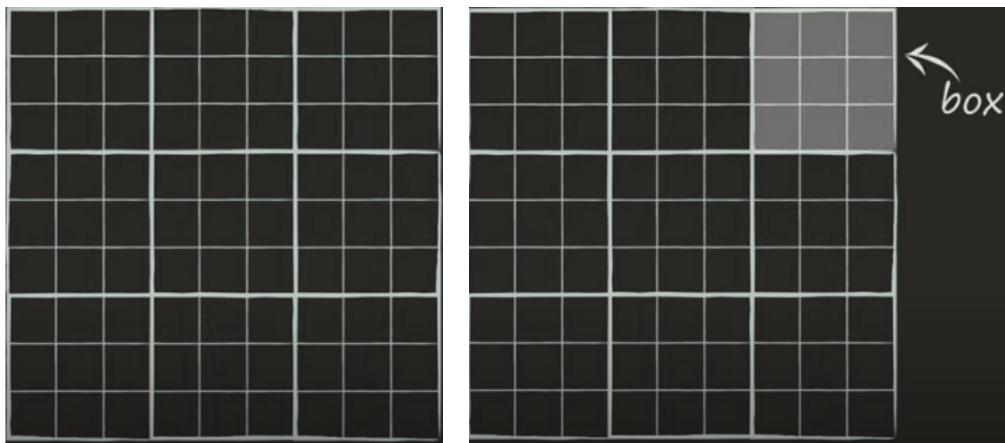
- **Generate Worlds**

Generate Worlds [45] es una herramienta centrada en la generación procedural de entornos 2D y 3D mediante el uso de reglas. La base de esta herramienta está inspirada por el algoritmo WFC, habiéndose llevado a cabo una serie de mejoras con el fin de obtener un mejor rendimiento.



### 3. Wave Function Collapse (WFC)

El Sudoku es un juego en solitario jugado en una **cuadrícula** de 9x9 que está a su vez **dividida en** 9 casillas de 3x3.



El objetivo es llenar cada espacio con un número entre 1 y 9. Hay una serie de **reglas o restricciones** que debemos seguir: cada fila, columna y casilla debe contener todos los números del 1 al 9, y ningún número puede aparecer dos veces en la misma fila, columna o casilla. Las siguientes figuras muestran un Sudoku correctamente rellenado:

6	1	5	8	3	9	2	4	7
7	8	2	6	1	4	5	3	9
9	3	4	5	7	2	8	6	1
8	7	9	1	4	6	3	2	5
1	2	6	7	5	3	9	8	4
5	4	3	2	9	8	7	1	6
4	9	7	3	8	1	6	5	2
2	5	8	4	6	7	1	9	3
3	6	1	9	2	5	4	7	8

6	1	5	8	3	9	2	4	7
7	8	2	6	1	4	5	3	9
9	3	4	5	7	2	8	6	1
8	7	9	1	4	6	3	2	5
1	2	6	7	5	3	9	8	4
5	4	3	2	9	8	7	1	6
4	9	7	3	8	1	6	5	2
2	5	8	4	6	7	1	9	3
3	6	1	9	2	5	4	7	8

6	1	5	8	3	9	2	4	7
7	8	2	6	1	4	5	3	9
9	3	4	5	7	2	8	6	1
8	7	9	1	4	6	3	2	5
1	2	6	7	5	3	9	8	4
5	4	3	2	9	8	7	1	6
4	9	7	3	8	1	6	5	2
2	5	8	4	6	7	1	9	3
3	6	1	9	2	5	4	7	8

6	1	5	8	3	9	2	4	7
7	8	2	6	1	4	5	3	9
9	3	4	5	7	2	8	6	1
8	7	9	1	4	6	3	2	5
1	2	6	7	5	3	9	8	4
5	4	3	2	9	8	7	1	6
4	9	7	3	8	1	6	5	2
2	5	8	4	6	7	1	9	3
3	6	1	9	2	5	4	7	8

Cuando observamos un puzzle de Sudoku estando vacío, cada cuadrado en la cuadrícula tiene el potencial de ser cualquiera de los 9 números posibles.

1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9

Podríamos decir que hay una **superposición** de los 9 estados posibles ocupando ese cuadrado a la vez, pero usualmente el puzzle de Sudoku no empieza estando vacío. En su lugar hay ya unos pocos números rellenos que han **colapsado** a una única posibilidad. Este **colapso** afectará por supuesto al espacio posible de las **celdas vecinas** así que podemos **propagar** ese conocimiento a esas celdas afectadas.

1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
4	5	6	4	5	6	4	5	6	4	6	7	4	5	6	4	5	6
7	8	9	7	8	9	7</td											

Después de **propagar** todas las restricciones impuestas por nuestras **condiciones iniciales**, el siguiente paso lógico a seguir es buscar la celda con el menor número de estados restantes posibles, es decir, la celda con la **menor entropía**, y **colapsar** esa celda a una única posibilidad una vez más, **propagando** el resultado a las celdas afectadas.

2 3 4	2 3 4	3 5 6	1 4 6	5 6 7	1 4 5	2 3 4	1 2 3	1 2 4
6 7 8	5 6 7	7 8 9	7 8 9	8 9	6 8 9	6 7 9	4 6 9	6 7
9	8							
2 4 6	2 4 6	6 7 9	1 4 6	6 7 9	3	2 4 6	8	5
7 9	7		7 9			7 9		
3 4 6	3 4 5		4 6 7		4 5 6	3 4 6	3 4 6	4 6 7
7 8 9	6 7 8	1	8 9	2	8 9	7 9	9	
1 2 3	1 2 3	3 6 8	5	3 6 8	7	2 3 4	2 3 4	2 4 6
6 8	6 8		9		9	6 8 9	6 9	8
2 3 6	2 3 5		2 3 6	3 6 8	2 6 8	1	2 3 5	2 6 7
7 8	6 7 8	4	8 9	9	9	6 9	8	
1 2 3	3 5 6		1 2 3	3 6 8	1 2 4	2 3 4	2 3 4	2 4 6
6 7 8	9	7 8	4 6 8		6 8	5 6 7	5 6	7 8
5	1 4 6	6 8 9	2 6 8	6 8 9	2 6 8	2 4 6	7	3
8			9		9	8		
3 4 6	3 4 6		3 6 7		5 6 8	4 5 6	4 5 6	4 6 8
7 8 9	7 8	2	8 9		9	8		
1 3 6	1 3 6	3 6 7	2 3 6		2 5 6	2 5 6	1 2 5	9
7 8	7 8	8	7 8	4	8	8	6	

La razón por la que priorizamos las celdas de **menor entropía** es porque hay menos **estados** para elegir y es menos probable que hagamos una mala decisión. Continuaremos este proceso de iterar sobre el puzzle **colapsando** y **propagando** hasta que todas las celdas estén **colapsadas** a una única posibilidad y el puzzle sea resuelto. También es posible que la elección de menor entropía nos lleve hasta una configuración para la que no hay solución válida, en cuyo caso habría que retroceder (backtracking) hasta el momento de la elección y tomar otro número como camino de construcción. Obsérvese que el resultado final no tiene porqué ser único (e incluso no tener solución en absoluto), depende de las restricciones iniciales impuestas.

Como el lector está esperando una explicación de cómo funciona el algoritmo WFC, y no una de como jugar al Sudoku, vamos a saltar a la similitud que existe entre el método expuesto para resolver Sudokus y el algoritmo que nos interesa.

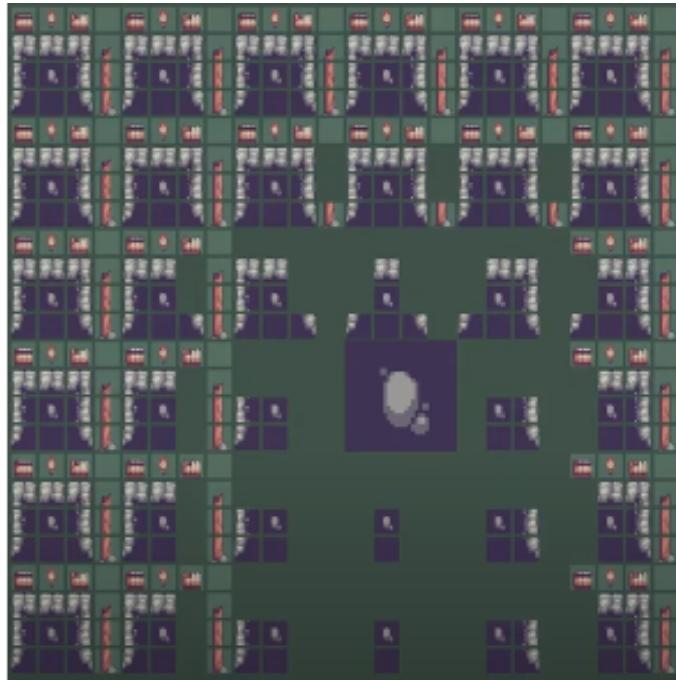
El algoritmo WFC es un solucionador procedural que toma una **cuadrícula de celdas** donde cada una de ellas ocupa una **superposición** de todos los posibles estados que puede tomar. Llamamos a esto **función de onda**.



A cada **estado** potencial para la celda lo llamaremos **tile** o **módulo**. Este **estado** viene con su propio conjunto de **reglas de adyacencia**.



De la misma manera que un humano realiza un puzzle de Sudoku, el algoritmo WFC buscará la **celda de menor entropía** (o elegirá una aleatoriedad si la menor entropía es común a varias **celdas**). Posteriormente, el algoritmo WFC colapsará la celda encontrada a un único **tile** o **módulo**. Entonces el algoritmo WFC pasará a **propagar** las consecuencias de colapsar esa **celda** a las otras celdas.



Este proceso se realizará hasta que todas las celdas de la cuadrícula estén colapsadas. Entonces podremos decir que la función de onda ha colapsado.



Como se puede ver, el algoritmo WFC es bastante sencillo y, debido a que tiene más que un parecido razonable con el proceso de resolver un rompecabezas de Sudoku, es aún más fácil de explicar e intuir.

Ahora que tenemos una buena idea de en qué consiste el algoritmo WFC, podemos ver que nos estamos enfrentando a un problema **SAT** (Boolean Satisfiability Problem => **SATISFIABILITY** => **SAT**). Este es el problema de determinar si existe una interpretación que satisfaga una fórmula booleana dada. Este tipo de problema es el primero para el que

demostró que estaba en la clase de los problemas **NP-completos**. Esto significa que todos los problemas de clase de complejidad NP que incluyen un amplio rango de decisiones naturales y problemas de optimización son los más difíciles de resolver como **SAT**. No existe algoritmo conocido que sea eficiente (los algoritmos eficientes caen dentro de la que se denomina problemas **P**) para resolver cualquier problema **SAT** y, de manera general, se cree que tal algoritmo no existe. La determinación de si **P** y **NP** son iguales es uno de los problemas abiertos más famosos y fundamentales de la actualidad, en lo que se denomina el problema **P versus NP**.

En este proyecto se explicará cómo se ha implementado el algoritmo WFC para la generación procedural de mapeados en 2D y compartiré mi viaje por las diferentes partes que componen esta implementación.

Usé **Unity 2020.3.27f1** con el fin de crear un entorno visual e interactivo con el que poder trabajar con el algoritmo WFC.

## 3.1 Constraint Satisfaction Problems (CSPs)

Los problemas de satisfacción de restricciones (CSPs) [56] son preguntas matemáticas definidas como un conjunto de objetos cuyo estado debe satisfacer una serie de restricciones o limitaciones. Los CSPs representan las entidades en un problema como una colección homogénea de restricciones finitas sobre variables que se resuelve mediante métodos de satisfacción de restricciones. Los CSPs a menudo exhiben una alta complejidad, requiriendo una combinación de métodos heurísticos y de búsqueda combinatoria para ser resueltos en un tiempo razonable. El problema SAT visto anteriormente es un caso particular de CSP.

Formalmente, un problema de satisfacción de restricciones se define como un triple  $\{X, D, C\}$  donde:

- $X = \{X_1, \dots, X_n\}$  es un conjunto de variables.
- $D = \{D_1, \dots, D_n\}$  es el conjunto de sus respectivos dominios de valores.
- $C = \{C_1, \dots, C_n\}$  es un conjunto de restricciones.

A un lector familiarizado con los CSPs (Constraint Satisfaction Problems) le debe resultar evidente que llenar un mundo finito de tiles es un CSP. En un CSP tenemos un conjunto de variables, un conjunto de valores que cada variable puede tomar (esto es lo que se denomina dominio) y un conjunto de restricciones. Para nosotros, las variables son una localización en el mapa, el dominio de cada variable es la lista de tiles o módulos que esta puede tomar, y las restricciones se basan en que cada módulo o tile cuenta con una serie de vecinos con los que puede encajar en el mapa.



Intuitivamente, el problema de crear correctamente un mapa es que los tiles o módulos pueden codificar dependencias arbitrarias de largo alcance, lo que a menudo nos lleva a tiempos de ejecución exponencial. La esperanza para luchar contra esos tiempos de ejecución es que podamos crear mundos interesantes mediante la búsqueda acelerada por heurística o adoptando la estrategia de “divide y vencerás”.

## 4. Implementación

### 4.1 Tecnologías empleadas

#### 4.1.1 Unity



Unity [6] es una herramienta de desarrollo de videojuegos creada por la empresa Unity Technologies. Entre sus principales características se encuentran:

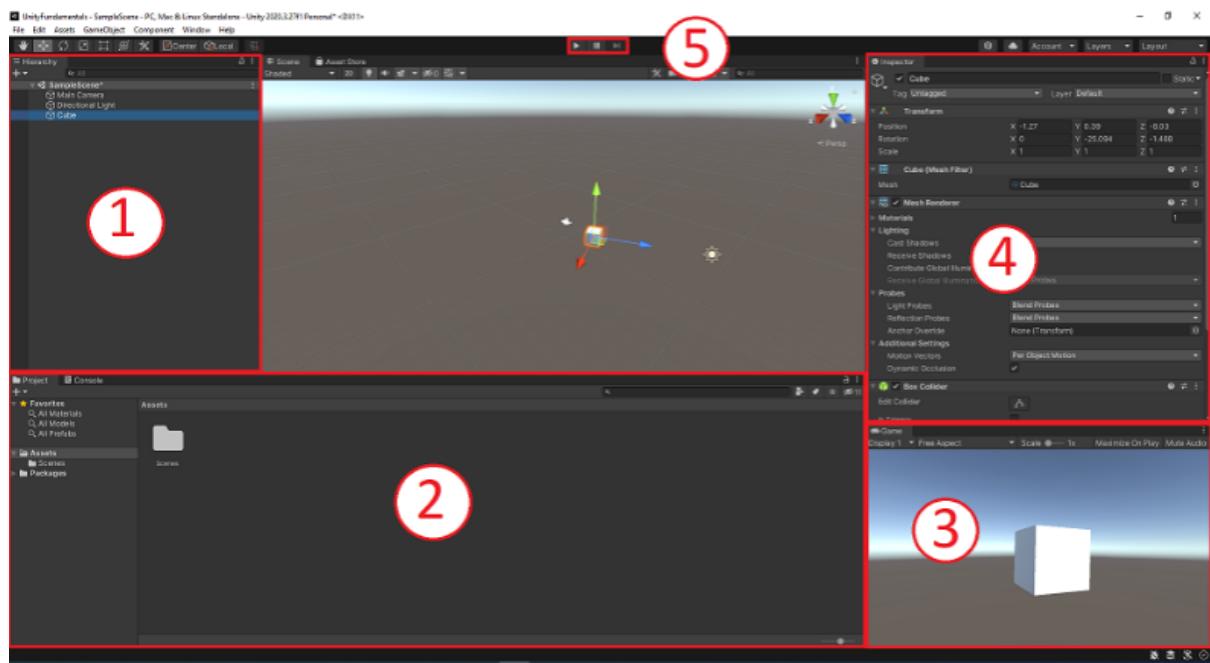
- Soporte de exportación a una gran cantidad de plataformas.
- Facilidad de uso.
- Versatilidad para realizar proyectos en 2D y 3D.
- Gran comunidad de desarrolladores independientes.
- Uso de C# como lenguaje de scripting.

Existen otras herramientas similares, como pueden ser Unreal Engine o Godot. Unreal Engine es, junto con Unity, una de las herramientas más usada para este tipo de propósito aunque tiene un mayor enfoque en proyectos 3D además de tener como lenguaje de scripting C++. Godot, al igual que Unity, es más versátil para proyectos en 2D y 3D, pudiendo hacer scripting en JavaScript, aunque en cualquier caso tiene una menor comunidad de desarrolladores que Unity, pero es un proyecto completamente Open Source. Todo lo comentado me ha llevado a elegir Unity como herramienta para este proyecto.

A continuación se pretende hacer una introducción sin demasiada profundidad a los elementos más importantes que componen esta herramienta, ya que es necesario conocerlos y tenerlos en cuenta para la implementación del algoritmo WFC que se desarrollará posteriormente.

##### 4.1.1.1 Fundamentos.

Al iniciar un proyecto con la herramienta podemos observar un layout igual o similar al mostrado a continuación. Un proyecto en Unity es el equivalente a lo que podemos encontrar en otros entornos de desarrollo denominados como **solución**.



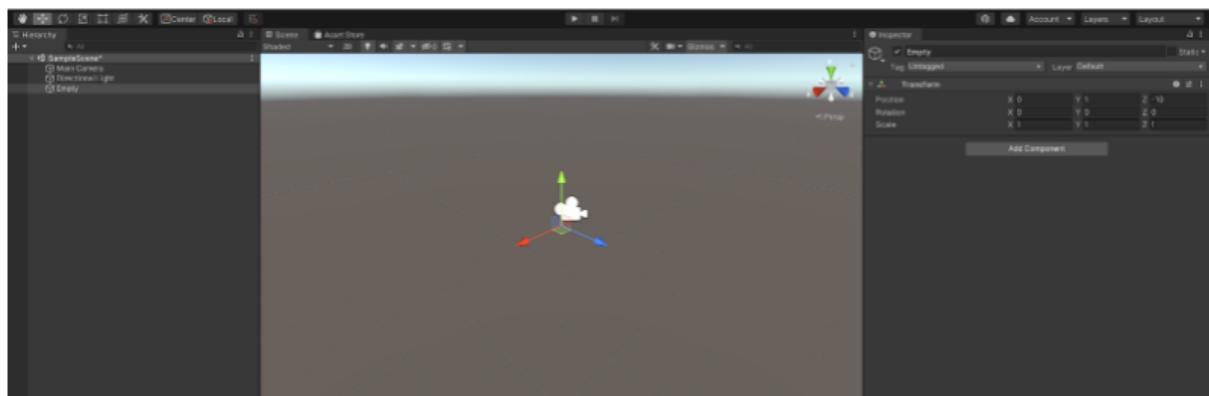
Unity por defecto crea lo que se denomina una escena. Este término hace referencia a lo que sería el entorno de trabajo dentro de un proyecto. Esto nos lleva por ejemplo, si tuviésemos un juego con varios niveles, a dividir esos mismos niveles en diferentes escenas.

Como se puede ver en la imagen anterior hay una serie de pestañas/secciones que son de vital importancia conocer para trabajar con la herramienta. Siguiendo los índices de la imagen, estas son:

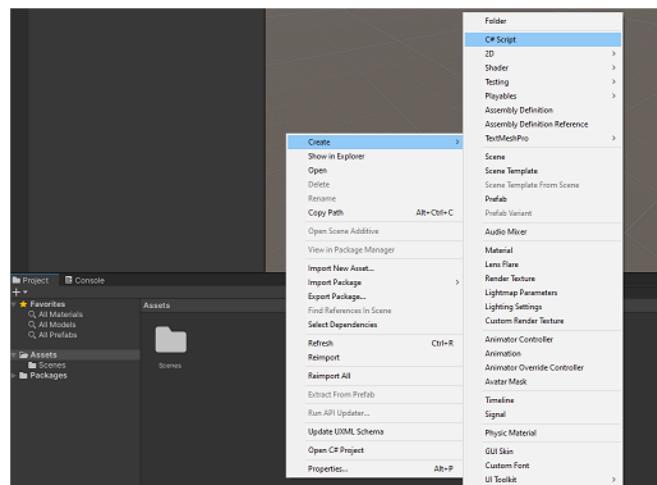
- **(1) Ventana de jerarquía:** Es el lugar donde tendremos todos los objetos del juego que formen parte de la escena actual.
- **(2) Ventana de ficheros del proyecto:** Este es el lugar del layout donde podemos ver almacenados los diferentes elementos que componen nuestro proyecto, como pueden ser imágenes, elementos de configuración, animaciones, etc.
- **(3) Ventana de juego:** En esta ventana podremos ver la ejecución de la escena.

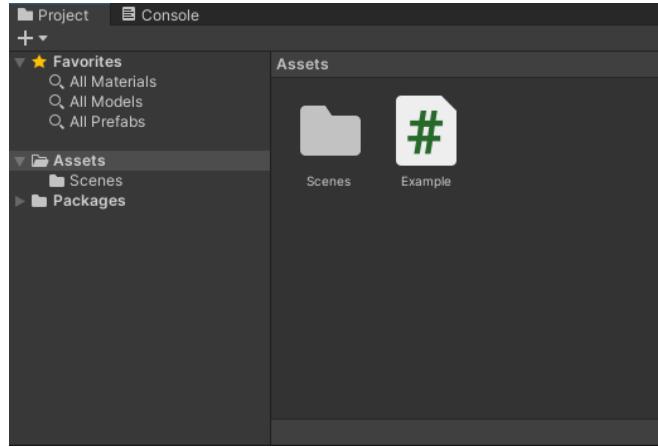
- **(4) Ventana de inspección:** Esta es una pestaña contextual que nos dará información o funcionalidad sobre un elemento del proyecto seleccionado.
- **(5) Botones de ejecución de la escena:** Estos botones controlan la ejecución de la escena, siendo el primero para iniciar la ejecución de la misma, el segundo para poner en pausa la ejecución y el tercero para avanzar la ejecución de la escena fotograma a fotograma cuando la ejecución está en pausa.

Los objetos de una escena en Unity cuentan con lo que se denominan componentes [9], teniendo cualquier objeto de la escena como mínimo un componente llamado "**Transform**", que indica la posición, rotación y escala de ese objeto en el mundo. Estos componentes son los que definirán el comportamiento del objeto en la escena.



Unity proporciona una serie de componentes por defecto, pero podemos añadir componentes propios y definir el comportamiento a medida de un objeto de la escena mediante scripting.





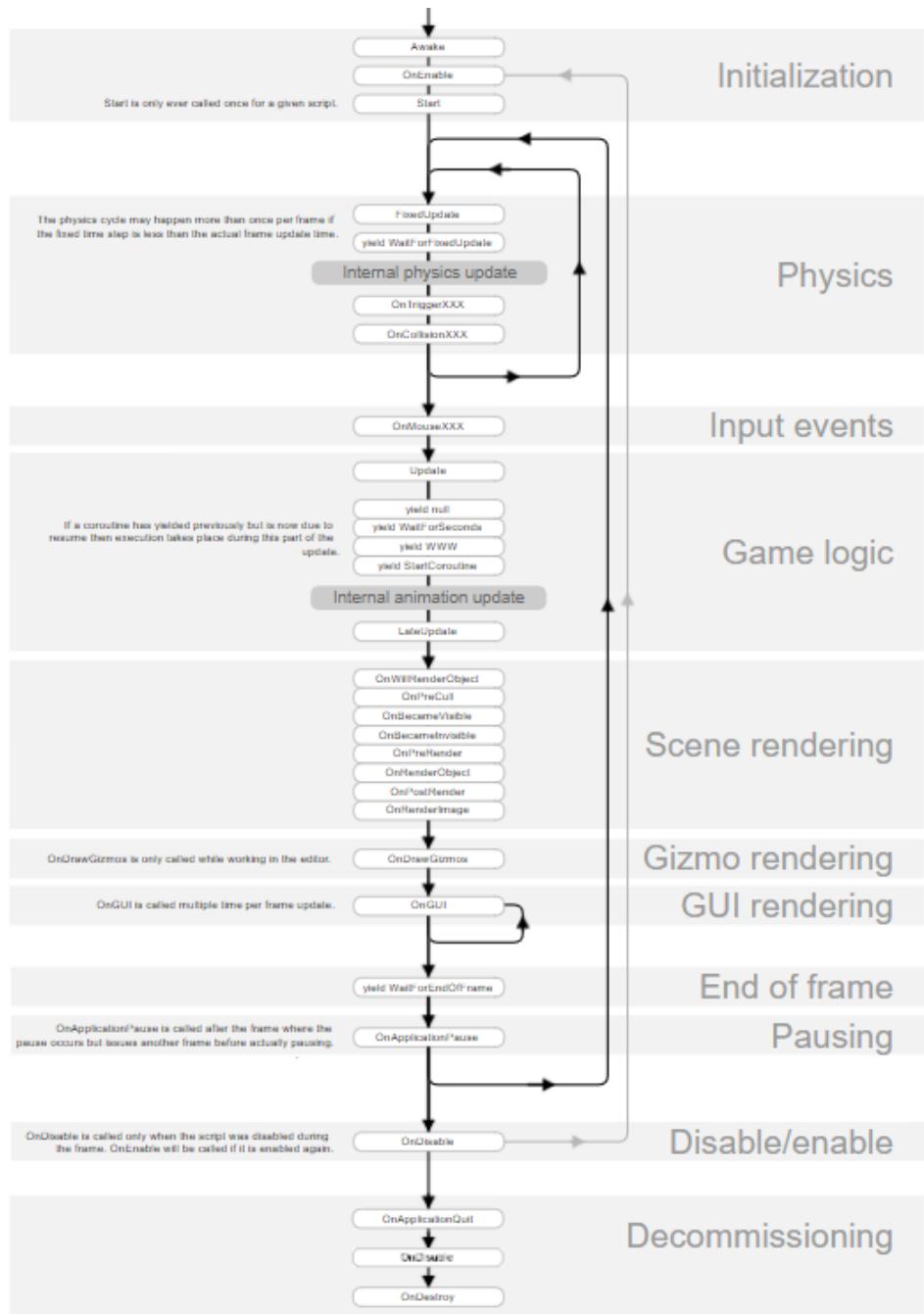
Al crear un script en Unity se cuenta, por defecto, con la siguiente estructura:

```
Example.cs  X
Assembly-CSharp
Example

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Example : MonoBehaviour
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10
11
12
13      // Update is called once per frame
14      void Update()
15      {
16          this.transform.Rotate(new Vector3(0f, 0f, 100f) * Time.deltaTime);
17      }
18 }
```

The screenshot shows the Unity Editor's code editor with the file 'Example.cs' open. The code defines a class 'Example' that inherits from 'MonoBehaviour'. It contains two methods: 'Start()' and 'Update()'. The 'Update()' method rotates the object's transform by 100 degrees on the Z-axis every frame. Unity's code completion and documentation are visible as tool tips above the code.

Lo primero que podemos observar es que se crea una clase con el nombre del script creado y que hereda de otra clase llamada *MonoBehaviour*. Esta es la clase proporcionada por Unity como fuente de herencia para un objeto de la escena que tiene un comportamiento. En consecuencia, de esta herencia surgen una serie de métodos sobre los que podemos definir el comportamiento del objeto en un determinado momento del ciclo de vida del script dentro de Unity.



Por el momento, solo nos interesa conocer los siguientes métodos dentro de este ciclo de vida [5]:

- **Start**

Es un método que solo se ejecuta una vez antes de la actualización del primer frame o fotograma y que se usa normalmente con propósitos de inicialización de parámetros.

- **Update**

Es un método que se llama una vez por cada frame o fotograma y es el que nos permite proporcionar un comportamiento específico a un objeto a lo largo de la ejecución de la escena. Aquí es necesario hacer un inciso para hablar del concepto de frame o fotograma y lo que se denomina FPS (fotogramas por segundo).

Un frame o fotograma no es más que una imagen procesada por el ordenador y que es mostrada por la pantalla. Llamamos, por lo tanto, fotogramas por segundo a la cantidad de imágenes que podemos mostrar en un segundo. Esto es necesario saberlo pues tiene una serie de implicaciones respecto a Unity y a los objetos de la escena.

Teniendo en cuenta lo comentado anteriormente es necesario poner atención en los siguientes puntos:

- **La complejidad computacional del código introducido en el método Update afecta a los FPS.**

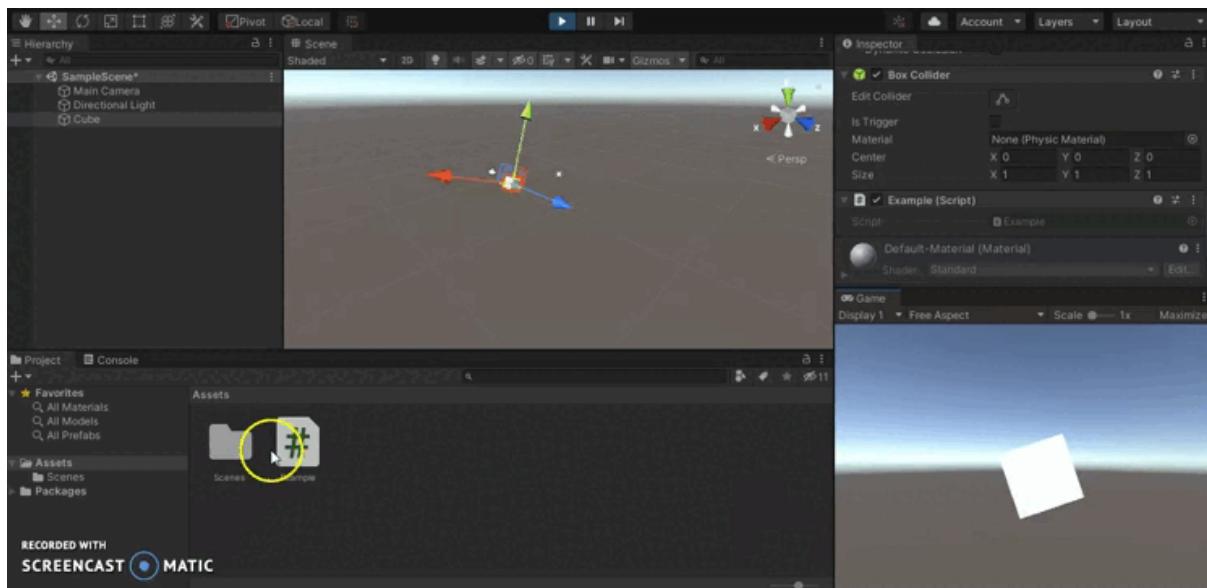
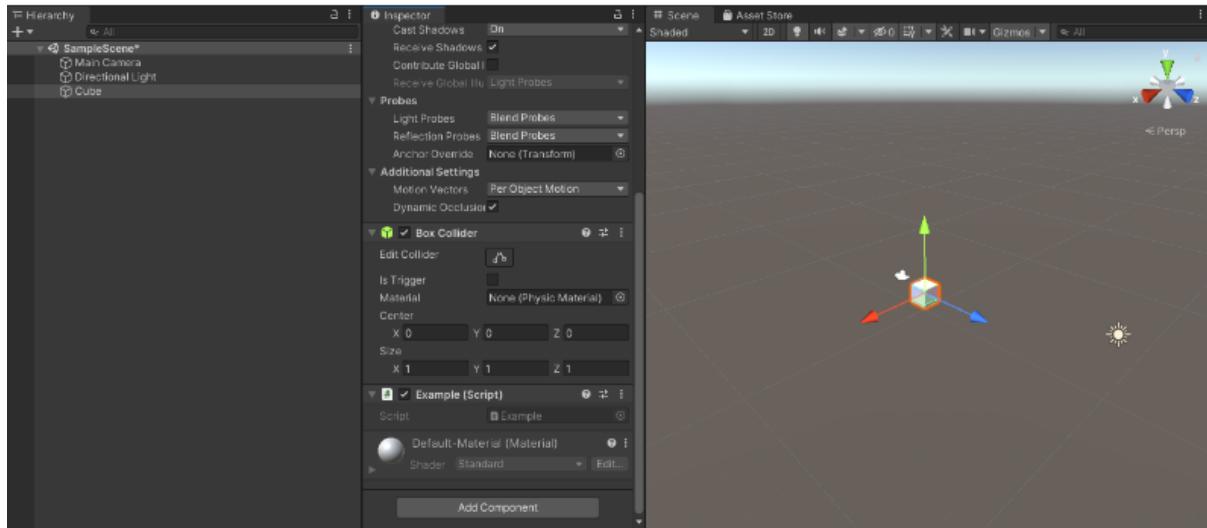
Hemos dicho que el método Update asociado a un objeto de nuestra escena se ejecuta una vez por frame y, por lo tanto, si introducimos un código de una cierta complejidad computacional en él, esto nos llevará a intervalos entre fotogramas superiores ralentizando así la actualización de los diferentes objetos de la escena y la interfaz con el usuario. Para ello Unity nos proporciona las corrutinas de las que hablaremos más adelante, y que nos permitirán distribuir la ejecución de un determinado código a lo largo de varios frames. En el caso de este proyecto esto será necesario, puesto que el algoritmo WFC es de un alto costo computacional.

- **La cantidad de FPS depende de la potencia del equipo que ejecuta la escena.**

Esto nos lleva a que equipos más potentes son capaces de mostrar más fotogramas por segundo, realizando por tanto más llamadas al método Update que equipos menos potentes. Es relevante conocer esto puesto que si, por ejemplo, queremos desplazar un objeto en nuestra escena durante la ejecución de la misma y no contemplamos este suceso, podremos experimentar cómo, dependiendo del equipo, el objeto posee un desplazamiento diferente, ya que irá en concordancia con los FPS que tengamos en un momento de la ejecución. Como lo normal es querer el mismo comportamiento para cualquier equipo sin estar ligado a los FPS, Unity nos provee del **Time.deltaTime**, que es el intervalo en segundos entre el frame actual y el anterior, permitiendo establecer una velocidad para un desplazamiento, una rotación, etc. de manera fija.

```
this.transform.Rotate(new Vector3(0f, 0f, 100f) * Time.deltaTime);
```

Una vez realizado el script deseado simplemente lo añadimos como componente del objeto que deseemos.



En un script, al añadir un atributo público, será editable desde el inspector de Unity una vez que se asocie el script a un objeto de la escena, como se puede ver a continuación.

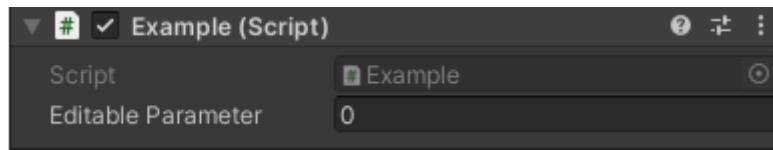
```

public class Example : MonoBehaviour
{
    public int editableParameter;

    // Start is called before the first frame update
    @ Mensaje de Unity | 0 referencias
    void Start()
    {
    }

    // Update is called once per frame
    @ Mensaje de Unity | 0 referencias
    void Update()
    {
        this.transform.Rotate(new Vector3(0f, 0f, 100f) * Time.deltaTime);
    }
}

```



Al igual que se puede hacer con un tipo simple como un entero, esto también se puede hacer para otros tipos de elementos más complejos, como puede ser un objeto de la escena, lo que resulta muy útil en muchas ocasiones. Además, no solo se puede hacer con atributos definidos como públicos, aunque en estos casos se tendrá que marcar de manera específica que se quiere sacar el atributo al editor de Unity.

```

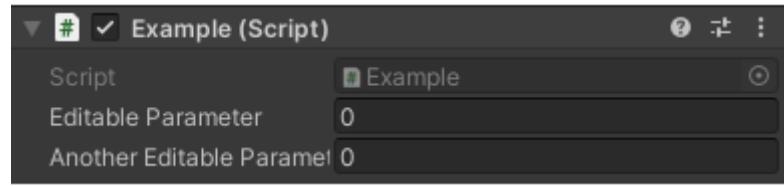
public class Example : MonoBehaviour
{
    public int editableParameter;
    [SerializeField]
    private int anotherEditableParameter;

    // Start is called before the first frame update
    @ Mensaje de Unity | 0 referencias
    void Start()
    {

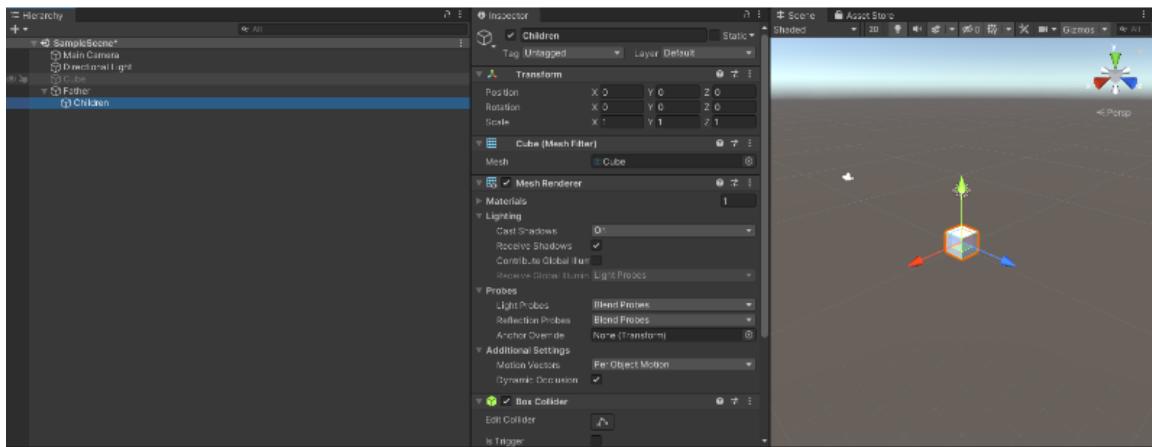
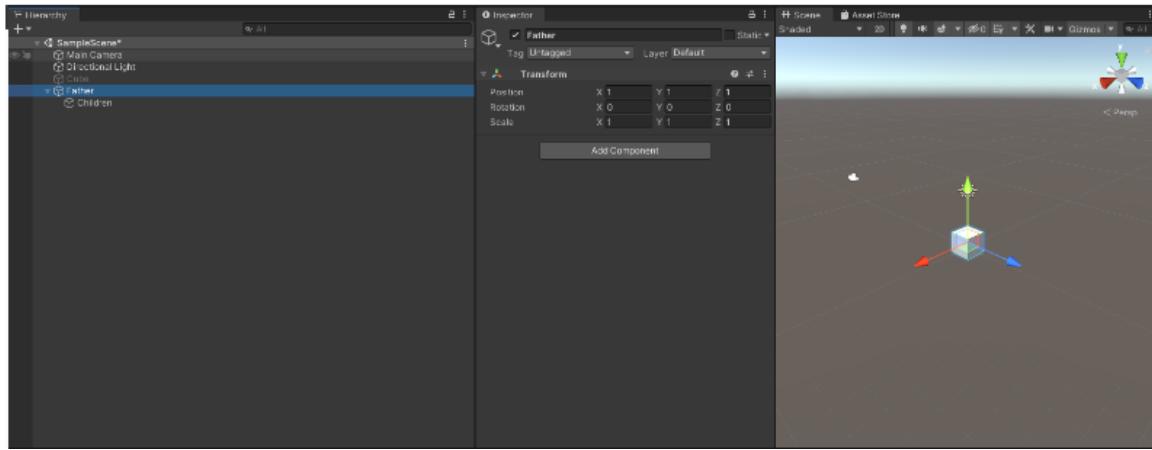
    }

    // Update is called once per frame
    @ Mensaje de Unity | 0 referencias
    void Update()
    {
        this.transform.Rotate(new Vector3(0f, 0f, 100f) * Time.deltaTime);
    }
}

```

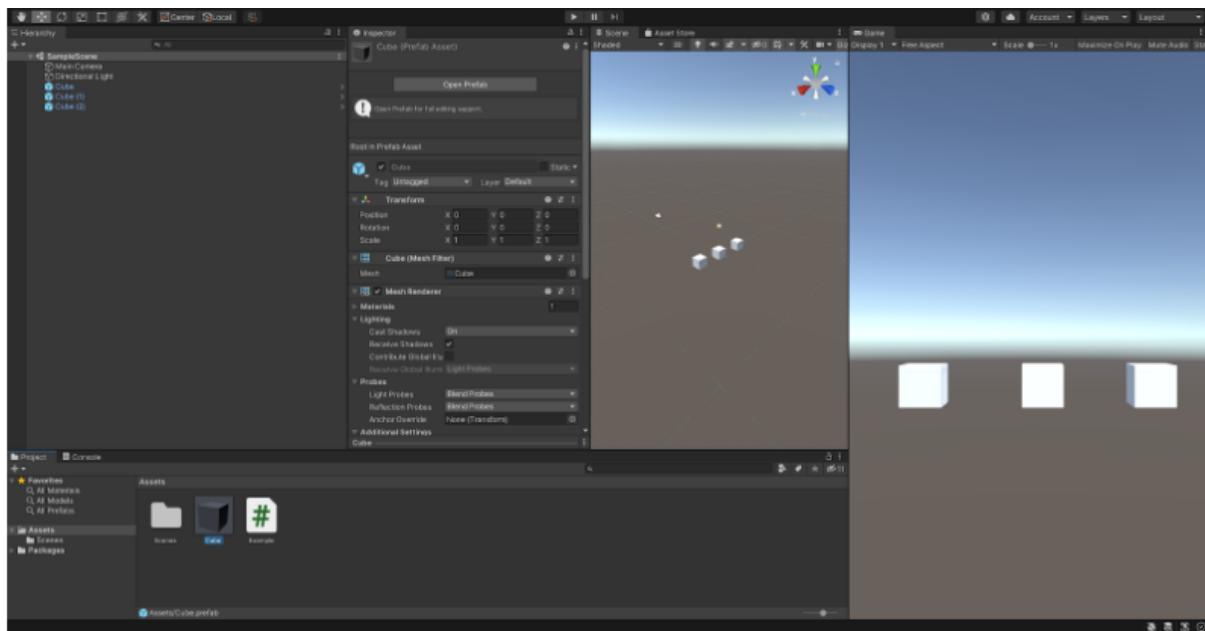


Una cosa que se debe conocer es que dentro de los objetos de la escena se pueden establecer jerarquías. En estas jerarquías que se establecen en la forma padre-hijo, los hijos trabajan con posiciones relativas a los padres, como se puede ver en la siguiente imagen.



#### 4.1.1.2 Prefabs

Los prefabs [10] en Unity, como su nombre indica, son objetos de la escena fabricados previamente. Esto es útil para instanciar múltiples objetos de la escena con una misma funcionalidad/comportamiento, como por ejemplo puede ser una moneda interactiva en un juego.



#### 4.1.1.3 Componentes básicos.

Hay una serie de componentes básicos para los objetos de la escena que debemos conocer para este proyecto. Son los siguientes:

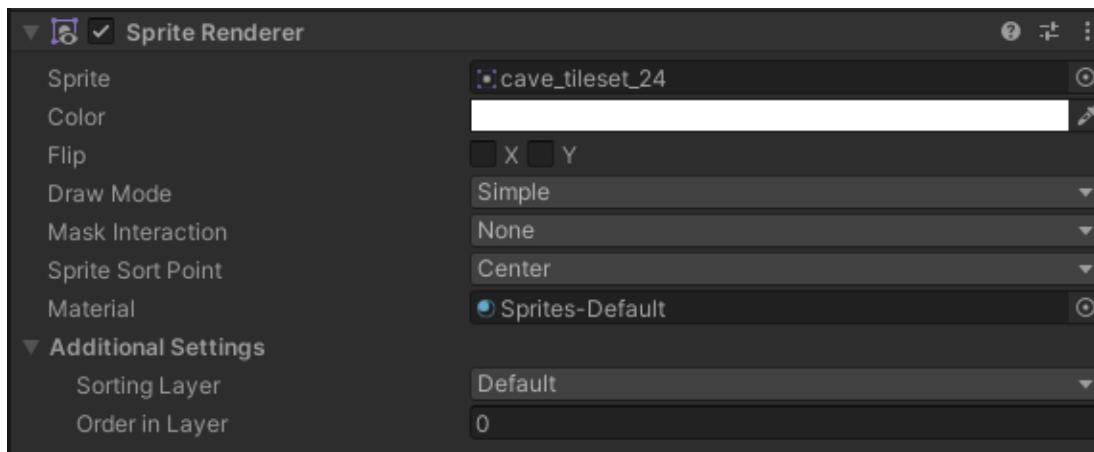
- **Transform**

Este es el único componente obligatorio para un objeto de la escena. Especifica su posición, rotación y escala en el mundo.



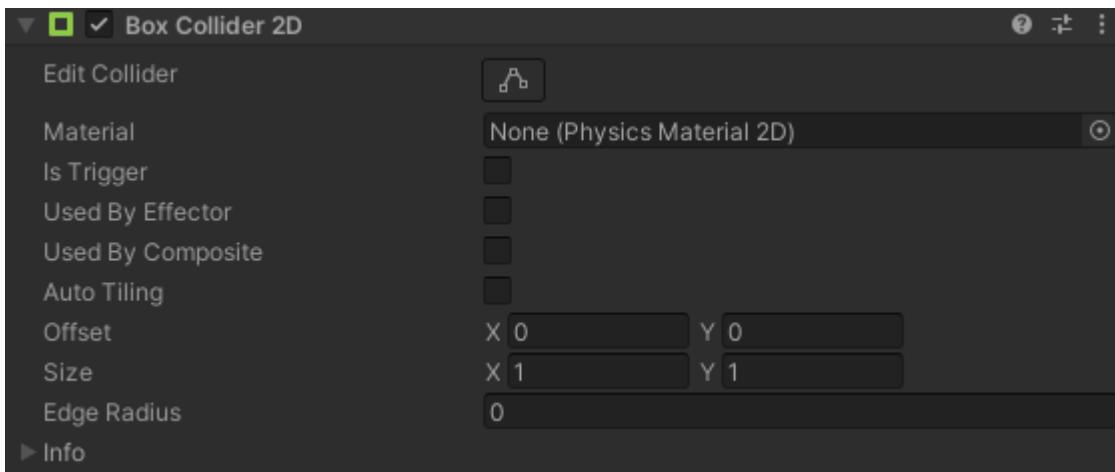
- **Sprite Renderer**

Para renderizar una imagen en la escena es necesario que el objeto donde se quiera visualizar cuente con este componente, en el que se especificará la imagen a renderizar.



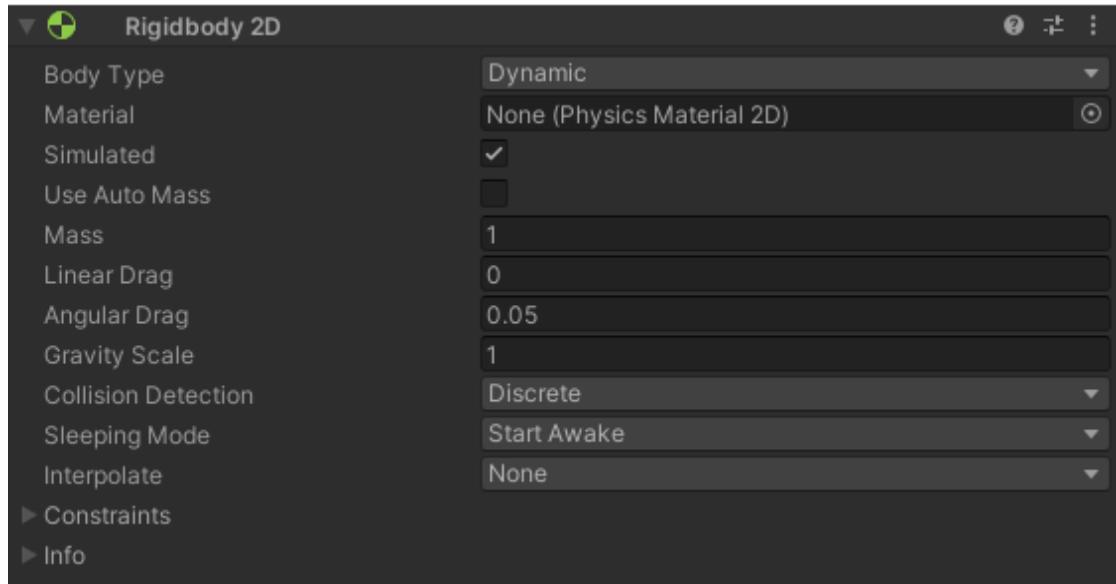
- **Box Collider 2D**

Este componente es uno de los tipos de collider que nos proporciona Unity. Dispone de una forma de caja y nos permite que un objeto de la escena se tenga en cuenta en el sistema de colisiones proporcionado por Unity. Este componente es necesario, en nuestro caso, para la interacción del ratón del usuario con una determinada imagen, siendo estas imágenes los módulos con los que trabajará el algoritmo WFC.



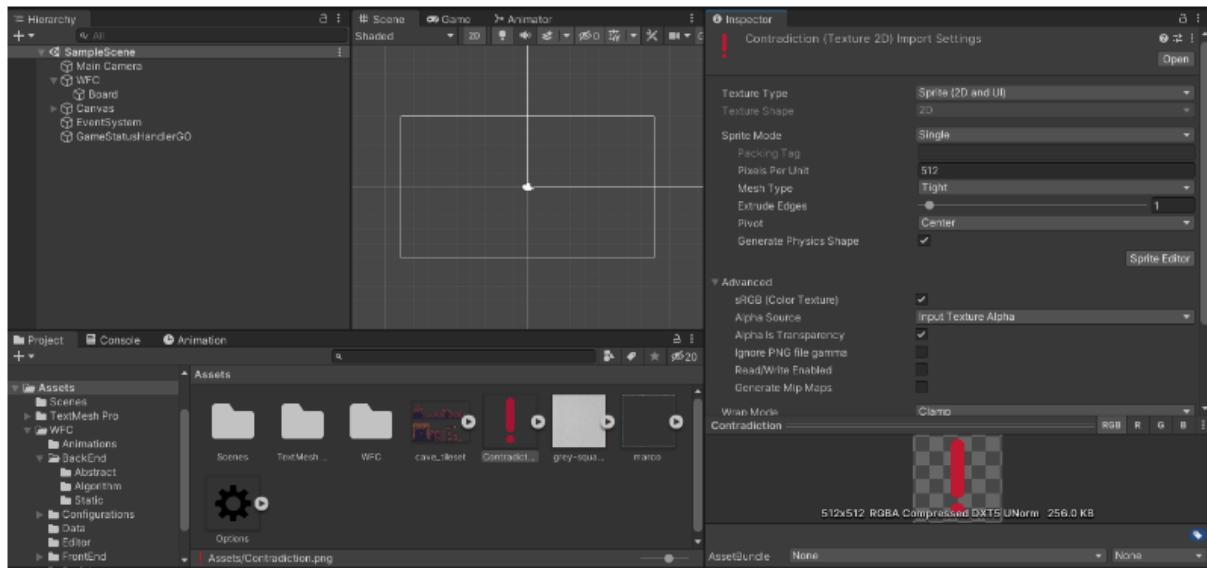
- **Rigidbody 2D**

Este componente nos permite introducir un objeto 2D al sistema de físicas de Unity, consiguiendo que este sea afectado por fuerzas como, por ejemplo, la gravedad.



#### 4.1.1.4 Importación de imágenes

Para usar una imagen en un proyecto en Unity simplemente bastará con introducirla a la carpeta de Assets del proyecto. Una vez la imagen se encuentre en el proyecto, al seleccionarla podemos decidir cómo queremos que esta se importe.



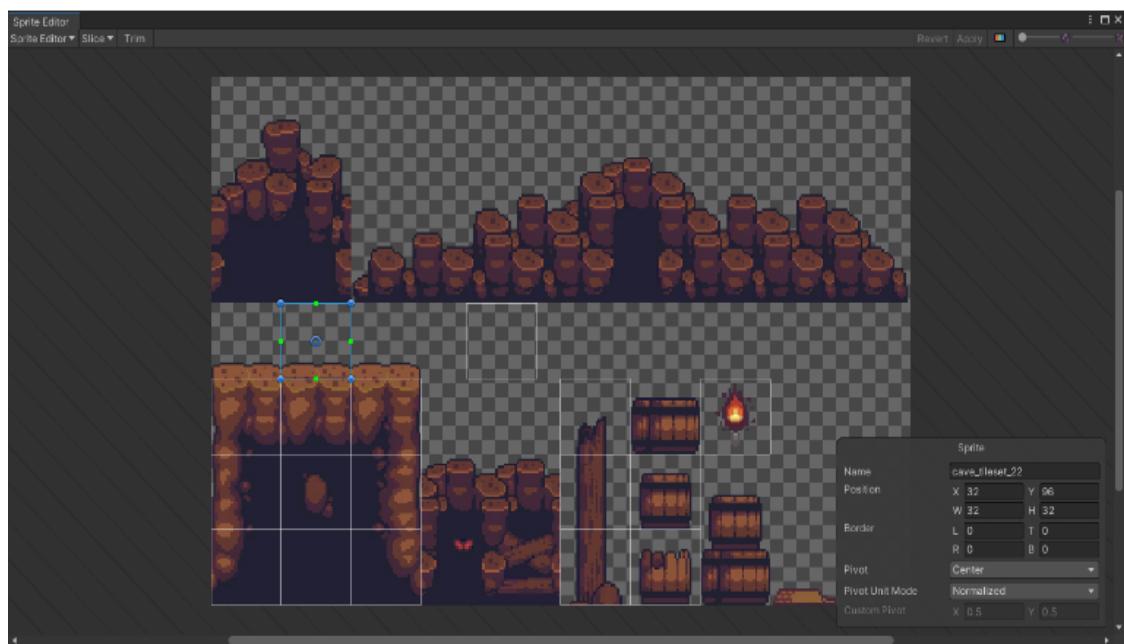
Entre las opciones de importación de la imagen, hay dos opciones básicas e importantes a tener en cuenta. Son las siguientes:

- **Sprite Mode**

Indica a Unity cómo es el tipo de imagen que estamos importando.

**¿Contiene una única imagen? ¿La imagen a su vez contiene varias imágenes?**

En el caso de contener varias imágenes, Unity nos proporciona lo que llama "Sprite Editor".



Esto nos permite, entre otras cosas, seleccionar las distintas imágenes contenidas, como se puede ver en la imagen anterior.

- **Pixels Per Unit (PPU)**

Indica a Unity cuántos píxeles se van a dibujar en una unidad del grid Unity. La unidad de Unity la podemos entender como un cuadrado de 1x1 metro. En la imagen perteneciente al apartado anterior se puede ver cómo se quiere importar una imagen de 512x512 píxeles. Esto implica que si la importamos especificando que queremos 512 píxeles por unidad, la imagen va a ocupar en el mundo un cuadrado de 1x1 metro.

## 4.1.2 C#



C# [28] es el lenguaje de programación utilizado para el scripting en Unity, el cual cuenta con una serie de particularidades propias que se han usado para este proyecto y que se detallarán a continuación para un mejor entendimiento del código en ciertas partes del proyecto.

### 4.1.2.1 System.Linq y los métodos de extensión

Para facilitar el manejo de todas las celdas/slots que componen nuestro espacio se ha implementado una clase estática. La clase estática usa la extensión System.Linq [33], que sería el equivalente de C# a los Stream de Java, con el fin de ayudarnos a trabajar con listas mediante programación funcional. Además, se ha utilizado una peculiaridad de C#, que es lo que denominan *métodos de extensión* [35]. Estos métodos de extensión nos permiten "agregar" métodos a los tipos existentes sin crear un nuevo tipo derivado, recompilar o modificar de otra manera el tipo original.

```
using System.Linq;

/// <summary> Clase estatica para el manejo de listas de MapCell mediante progra ...
0 referencias
public static class MapCells
{
    /// <summary> Dada una lista de MapCell nos da informacion sobre la celda con la ...
    1 referencia
    public static MapCell GetMinEntropyCell(this List<MapCell> mapCells) {
        return mapCells.Where(mapCell => mapCell.GetEntropy() > 1).OrderBy(mapCell => mapCell.GetValidOptions().Count).FirstOrDefault();
    }

    /// <summary> Dada una lista de MapCell y unas coordenadas se intenta encontrar ...
    1 referencia
    public static MapCell GetCellByCoords(this List<MapCell> mapCells, Vector3Int coords) {
        return mapCells.Where(mapCell => mapCell.GetCoords().Equals(coords)).FirstOrDefault();
    }

    1 referencia
    public static bool AreMapCellsCollapsed(this List<MapCell> mapCells) {
        return mapCells.All(mapCell => mapCell.IsCollapsed());
    }

    /// <summary> Dada una lista de MapCell nos da informacion sobre si alguna celda ...
    0 referencias
    public static bool SomeContradiction(this List<MapCell> mapCells) {
        return mapCells.Any(mapCell => mapCell.IsContradicted());
    }
}

MapCell minEntropyCell = this.wfc.mapCells.GetMinEntropyCell();
```

#### 4.1.2.2 Yield keyword y IEnumerable

En C# existe una palabra reservada llamada `yield` [29]. Esta palabra reservada es usada en instrucciones como `yield return <expression>`, lo cual nos permite guardar el contexto actual de una parte del código a ejecutar dentro de una función, con el objetivo de devolver información y volver a ese mismo contexto en una llamada posterior a esa misma función. Esta instrucción tiene sus limitaciones, puesto que las funciones que hagan uso de esa instrucción están obligadas a devolver unos tipos de valores concretos, siendo uno de ellos `IEnumerable<T>`. Básicamente, este tipo de valor nos indica que el valor devuelto por la función es enumerable, es decir, se puede devolver de uno en uno y es lo que nos permite poder iterar una serie de elementos posteriormente mediante la estructura `foreach`. Por otro lado, la instrucción `yield break` permite indicar que ya no hay más elementos sobre los que iterar.

```
public override IEnumerable<WFCProgress> CollapseMinEntropyMapCell()
{
    MapCell minEntropyCell = this.wfc.mapCells.GetMinEntropyCell();
    if (minEntropyCell == null)
    {
        yield return new WFCUncollapsible("Imposible encontrar solucion para el WFC actual (Aplicar reset)");
        yield break;
    }
    else {
        foreach (WFCProgress progress in this.CollapseMapCell(minEntropyCell)) {
            yield return progress;
        }
    }
}
```

#### 4.1.2.3 Delegados y eventos

Un delegado [30] es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Al crearse una instancia del delegado se le puede asociar cualquier método que cuente con la cabecera especificada para el delegado e invocarlo posteriormente haciendo uso del delegado.

```
public delegate void Del(string message);

public static void DelegateMethod(string message){
    Console.WriteLine(message);
}

Del handler = DelegateMethod;
handler("Hello Word");
```

Esto, entre otras cosas, nos permite trabajar con eventos, los cuales son un tipo especial de delegado que nos permite notificar un suceso y manejarlo en consecuencia para todo aquel que esté interesado en ser notificado ante la aparición de ese suceso. A continuación, se muestra un ejemplo simple de cómo se utilizan los eventos.

```
public class Publisher{

    //Signature for the event.
    public delegate void EventHandler();

    //Event
    public event EventHandler ProcessCompleted;

    public void StartProcess(){

        Console.WriteLine("Process Started");
        //some code here...
        OnProcessCompleted();
    }

    //Notifier function
    protected virtual void OnProcessCompleted(){

        //Execute all handler functions registered for the event
        ProcessCompleted?.Invoke();
    }

}
```

```
public class Listener{

    public static void Main(){

        Publisher publisher = new Publisher();
        publisher.ProcessCompleted += ProcessCompletedHandler;//Event subscription
        publisher.StartProcess();

    }

    //Specific Handler for the event
    public static void ProcessCompletedHandler(){
        Console.Write("Process Completed");
    }

}
```

A la hora de notificar que ha sucedido un evento, existe el operador condicional `?.` que nos permite evitar errores de referencias nulas, por lo que su uso es altamente recomendable.

#### 4.1.2.4 Tipos implícitos y explícitos

C# es un lenguaje tipado que nos permite definir una variable en los métodos de dos maneras distintas.

```
var i = 10; // Implicitly typed.  
int i = 10; // Explicitly typed.
```

Una variable con tipo implícito sigue siendo fuertemente tipado exactamente igual que si se hubiera declarado de manera explícita, solo que es el compilador quien determina el tipo de la variable.

#### 4.1.2.5 Conversión de tipos

En C# existe el operador `as` [38], que permite convertir explícitamente una expresión a un tipo determinado si su tipo en tiempo de ejecución es compatible con ese tipo.

```
E as T
```

Donde `E` es una expresión que devuelve un valor y `T` es el nombre de un tipo o un parámetro de tipo.

#### 4.1.2.6 Virtual

La palabra clave `virtual` [39] se usa para modificar una declaración de método, propiedad, indizador o evento y permitir que se invalide en una clase derivada.

```
public virtual double Area()  
{  
    return x * y;  
}
```

#### 4.1.2.7 Override

El modificador `override` [40] es necesario para ampliar o modificar la implementación abstracta o virtual de un método, propiedad, indexador o evento heredado.

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    public override int GetArea() => _side * _side;

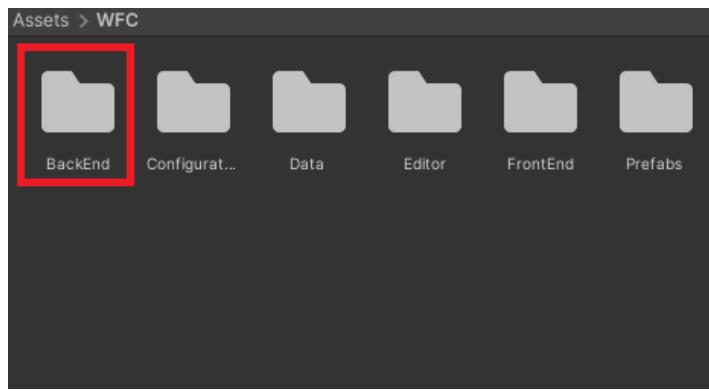
    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
```

## 4.2 Estructura del proyecto

El proyecto se ha dividido en tres ramas de desarrollo bien diferenciadas:

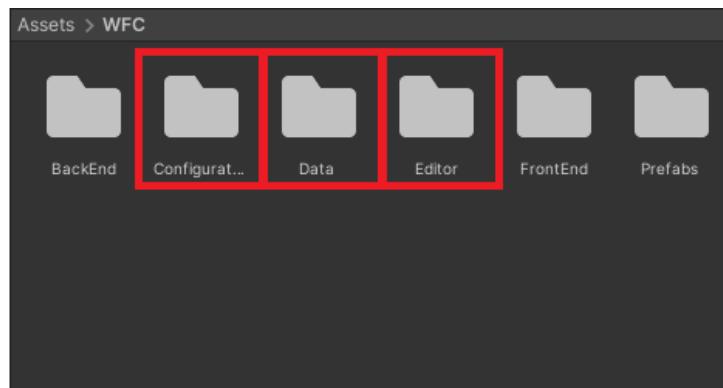
- **Backend**

Esta rama del desarrollo es el núcleo de este proyecto, pues es donde se implementa el algoritmo WFC, así como todos los elementos de los que se compone.



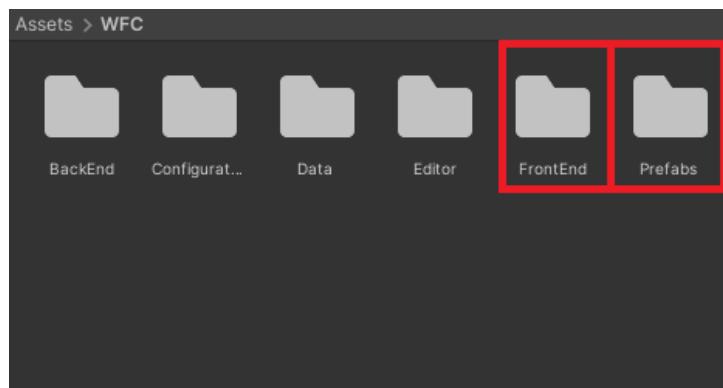
- **Entrada de usuario**

Esta rama del desarrollo se centra en gestionar la entrada del usuario para el algoritmo WFC.



- **Frontend**

Esta rama del desarrollo se centra en la visualización del funcionamiento del algoritmo WFC.



## 4.3 Backend

Esta rama de desarrollo, como se ha dejado ver previamente, es la más importante del proyecto. Aquí se implementa el algoritmo WFC, así como todos los elementos de los que se compone, como es el grid, celdas o slots y los módulos o tiles que estas/os tienen como estados posibles. La idea de este apartado es mostrar los aspectos básicos para entender cómo se ha implementado el algoritmo WFC y los diferentes elementos que lo componen. Para ello este apartado se ha dividido a su vez en una serie de subapartados que te guiarán por aspectos fundamentales de la implementación.

### 4.3.1 WFC Basics

Hay tres aspectos básicos que componen el algoritmo WFC. Son los siguientes:

- Grid

Este elemento hace referencia al entorno donde trabaja el algoritmo WFC, es decir, sería la función de onda que queremos que colapse, y simplemente consta de una lista de celdas o slots que dependen de las dimensiones en las que trabajamos.



- Celda o Slot

Este elemento hace referencia a una parte de nuestro entorno donde queremos que se defina un estado.



```
public abstract class MapCell
{
    protected Vector3Int coords; // Coordenadas de la celda
    protected List<Module> options; //Lista de opciones que puede tomar la celda
    protected List<Module> invalidOptions; //Lista de opciones invalidos para la celda
    protected bool renderizable;
}
```

Consta de los siguientes atributos:

- **coords**: Identifica la parte del grid/entorno a la que pertenece.
- **options**: Representa la superposición de todos los estados.
- **invalidOptions**: Lista donde se irán introduciendo los estados que queden descartados para la celda o slot.
- **renderizable**: Variable de control para la representación de la celda/slot en el Frontend.

Además de los métodos de acceso a sus atributos cuenta con los siguientes métodos:

- **GetValidOptions**

A partir de los módulos disponibles para la celda y los módulos definidos como inválidos nos proporciona la lista de módulos válidos a los cuales la celda puede colapsar.

```
public List<Module> GetValidOptions()
{
    if (options == null || invalidOptions == null)
    {
        Debug.LogError("MapCell_GetValidOptions: options property or invalidOptions property mustn't be null");
        return null;
    }

    List<Module> validOptions = options.Except(invalidOptions).ToList();
    return validOptions;
}
```

- **GetEntropy**

Este método nos devuelve la entropía de la celda que definimos previamente como el número de estados que puede tomar la celda para un momento dado de la función de onda.

```
public int GetEntropy()
{
    return this.GetValidOptions() == null ? -1 : this.GetValidOptions().Count;
}
```

- **IsCollapsed**

Este método nos permite determinar si una celda ha colapsado. El colapso lo especificamos como que la celda solo tiene un módulo válido al que colapsar.

```
public bool IsCollapsed()
{
    return this.GetValidOptions().Count == 1;
}
```

- **IsContradicted**

Este método nos permite determinar si en una celda existe una contradicción. La contradicción de una celda la especificamos como que la celda no cuenta con ningún módulo válido al que colapsar.

```
public bool IsContradicted()
{
    return this.GetValidOptions().Count == 0;
}
```

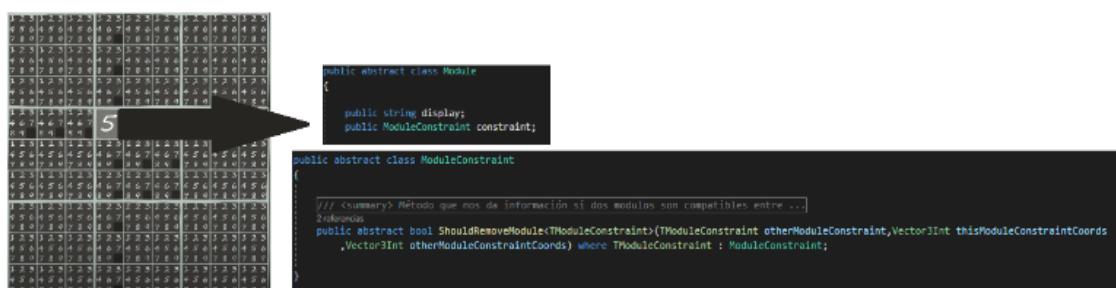
- **GetMapCellNeighborsCoords**

Este método nos permite obtener, a partir de la celda y el grid al que pertenece, las coordenadas de las celdas vecinas.

```
public abstract List<Vector3Int> GetMapCellNeighborsCoords<TMapGrid>(TMapGrid mapGrid) where TMapGrid : MapGrid;
```

- **Módulo o Tile**

Este elemento representa un estado que puede tener una celda o slot y contiene la restricción que debe cumplir el módulo o tile en el grid/entorno.



Como se puede ver solo se han hecho referencia a clases abstractas debido a que desde el punto de vista del algoritmo WFC es indiferente si estamos trabajando en un entorno 2D o 3D ya que solo verá una serie de restricciones que se deben cumplir en un determinado espacio. Esto nos permite abstraernos de implementaciones inferiores a las clases vistas anteriormente que definirán qué tipos de entorno, módulos y restricciones queremos.

Partiendo de lo anterior, estamos listos para generar distintos módulos con distintas restricciones que deben ser cumplidas en el espacio donde se aplicará el algoritmo WFC.

## 4.3.2 WFC Algorithm

Hay dos aspectos fundamentales que debemos tener en cuenta del algoritmo WFC para este proyecto:

1. **Es costoso computacionalmente.**
2. **Debe coexistir con una visualización en la capa Frontend.**

Por el momento nos centraremos en el segundo punto, que es la coexistencia con la capa Frontend. La clave de este punto se basa en que queremos que el usuario interactúe con el algoritmo WFC, así que tenemos que permitir observar el estado del grid en un determinado paso del algoritmo. Esto lleva a poder realizar dos acciones:

1. **Parar momentáneamente la ejecución del algoritmo en un pase clave del mismo.**

Para cubrir este punto haremos uso de **yield**, explicado anteriormente en los aspectos característicos de C#, y que nos permitirá precisamente lo que buscamos. Esto nos lleva a que los diferentes pasos del algoritmo tengan esta estructura.

```
/// <summary> Metodo que intenta colapsar la función de onda.
5 referencias
public abstract IEnumerable<WFCProgress> Collapse(bool collapsedRequired);

/// <summary> Metodo que intenta colapsar la celda de la cuadricula con la menor ...
3 referencias
public abstract IEnumerable<WFCProgress> CollapseMinEntropyMapCell();

/// <summary> Metodo que intenta colapsar una celda de la cuadricula.
3 referencias
public abstract IEnumerable<WFCProgress> CollapseMapCell(MapCell mapCell);

/// <summary> Metodo que intenta colapsar una celda a un modulo.
4 referencias
public abstract IEnumerable<WFCProgress> CollapseMapCellToModule(MapCell mapCell, Module module);

/// <summary> Metodo que intenta propagar informacion entre las celdas de la cu ...
3 referencias
public abstract IEnumerable<WFCProgress> Propagate(MapCell source);

/// <summary> Metodo que intenta eliminar una opcion disponible de colapso en un ...
4 referencias
public abstract IEnumerable<WFCProgress> RemoveMapCellOption(MapCell mapCell, Module module);

/// <summary> Metodo que intenta eliminar una opcion disponible de colapso en un ...
2 referencias
public abstract IEnumerable<WFCProgress> RemoveMapCellOptionFromUser(MapCell mapCell, Module module);

/// <summary> Metodo que intenta establecer la funcion de onda a su estado inici ...
4 referencias
public abstract IEnumerable<WFCProgress> Reset();

/// <summary> Metodo que nos da informacion sobre la funcion de onda con la que ...
11 referencias
public abstract MapGrid GetMapGrid();

}
```

## 2. Devolver la información que necesitamos para visualizar el estado de nuestra función de onda al usuario.

Para esto creamos una serie de clases que nos permita identificar qué acción clave ha sucedido dentro del algoritmo y guardar la información relevante respecto a esa acción.

```
public abstract class WFCProgress { }

3 referencias
public abstract class WFCVisualizableProgress : WFCProgress {
    public MapCell mapCell;
    public Module module;
}

10 referencias
public class WFCMapCellModuleRemoved : WFCVisualizableProgress
{
    3 referencias
    public WFCMapCellModuleRemoved(MapCell mapCell, Module module)
    {
        this.mapCell = mapCell;
        this.module = module;
    }
}

11 referencias
public class WFCMapCellCollapsed : WFCVisualizableProgress
{
    2 referencias
    public WFCMapCellCollapsed(MapCell mapCell, Module module)
    {
        this.mapCell = mapCell;
        this.module = module;
    }
}
```

Como hemos dicho previamente, el algoritmo WFC debe funcionar independientemente de si estamos trabajando en un entorno 2D o 3D ya que solo verá una serie de restricciones que se deben cumplir en un determinado espacio. Debido a esto creamos una clase genérica para el algoritmo WFC que se abstraiga del entorno en el que trabaja.

```
public class WFCRoutines<TMapGrid> : WFCRunner where TMapGrid : MapGrid
{
    public TMapGrid wfc;

    2 referencias
    public WFCRoutines(TMapGrid wfc)
    {
        this.wfc = wfc;
    }
}
```

#### 4.3.2.1 Métodos de ejecución

Hemos dejado de lado anteriormente el costo computacional que tiene el algoritmo WFC, que también afecta a la coexistencia con la capa Frontend y a los FPS. Para ello se han desarrollado dos modos de ejecución para el algoritmo WFC.

```
public static class ExecutionHandler
{
    /// <summary> Metodo para ejecutar el algoritmo WFC bajo las corrutinas de Unity ...
    7 referencias
    public static WFCProgressObserver RunAsCoroutine(this IEnumerable<WFCProgress> operation, MonoBehaviour context, float timeBudgetPerFrame = .1f) ...
    /// <summary> Metodo para ejecutar el algoritmo WFC en un frame a traves de iter ...
    1 referencia
    public static void RunAsRoutine(this IEnumerable<WFCProgress> operation, Action<WFCProgress> progressHandler = null) ...
}
```

- **Ejecución como rutina simple**

Este modo de ejecución es el más básico posible y a la vez el menos recomendable de usar, puesto que es equivalente a ejecutar el algoritmo WFC dentro del método Update de un objeto de nuestra escena, provocando que se ralentice toda la parte visual y los inputs del usuario.

```
public static void RunAsRoutine(this IEnumerable<WFCProgress> operation, Action<WFCProgress> progressHandler = null) {
    foreach (WFCProgress progress in operation) {
        progressHandler?.Invoke(progress);
    }
}
```

- **Ejecución como corrutina**

Para mejorar el rendimiento de cara al usuario que trabajará con el algoritmo WFC desde la capa Frontend en Unity existen lo que se denominan *corrutinas* [25]. Estas nos permiten repartir una tarea a realizar en distintos frames y las podemos ver como un hilo de ejecución distinto al actual.

```

public static WFCProgressObserver RunAsCoroutine(this IEnumerable<WFCProgress> operation, MonoBehaviour context, float timeBudgetPerFrame = .1f) {
    WFCProgressObserver wfcProgressObserver = new WFCProgressObserver();

    IEnumerator Routine() {
        yield return new WaitForEndOfFrame(); // Da la oportunidad al llamador de registrar metodos para los eventos generados
        var t = Time.realtimeSinceStartup;

        foreach (WFCProgress progress in operation) {
            wfcProgressObserver.NotifyOnProgress(progress);

            var d = Time.realtimeSinceStartup - t;
            if (d > timeBudgetPerFrame) {
                yield return new WaitForEndOfFrame();
                t = Time.realtimeSinceStartup;
            }
        }

        wfcProgressObserver.NotifyOnComplete();
    }

    Coroutine coroutine = context.StartCoroutine(Routine());
    return wfcProgressObserver;
}

```

Al usar corrutinas es necesario establecer un mecanismo de comunicación para observar el algoritmo WFC puesto que, como hemos dicho, podemos ver las corrutinas como un nuevo hilo de ejecución diferente. Para esto vamos a usar los eventos de C#.

```

public class WFCProgressObserver
{
    //Delegados
    public delegate void OnProgressHandler(WFCProgress progress);
    public delegate void OnCompletedHandler();
    public delegate void OnSelectedModuleHandler(WFCMapCellCollapsed selection);

    //Eventos
    public event OnProgressHandler onProgress;
    public event OnCompletedHandler onComplete;
    public event OnSelectedModuleHandler onSelectedModule;

    //Notificadores
    i referencia
    public void NotifyOnProgress(WFCProgress progress) {
        onProgress?.Invoke(progress);
        switch (progress) {

            case WFCMapCellCollapsed selection:
                NotifyOnSelectedModule(selection);
                break;

            default: break;
        }
    }

    i referencia
    public void NotifyOnComplete() {
        onComplete?.Invoke();
    }

    i referencia
    public void NotifyOnSelectedModule(WFCMapCellCollapsed selection) {
        onSelectedModule?.Invoke(selection);
    }
}

```

### 4.3.3 Operaciones

A continuación, vamos a desgranar la implementación del algoritmo WFC a través de sus diferentes operaciones.

- **Colapsar**

Esta operación se encarga de aplicar de manera completa el algoritmo WFC hasta colapsar la función de onda o encontrar una contradicción. Además, cuenta con un parámetro que determina si se quiere obligar a la función de onda a colapsar de manera que, si se encontrara una contradicción, la función de onda volvería a su estado inicial para volver a intentar ser colapsada.

```
public override IEnumerable<WFCProgress> Collapse(bool collapseRequired)
{
    while (!this.wfc.mapCells.AreMapCellsCollapsed())
    {
        foreach (WFCProgress progress in CollapseMinEntropyMapCell())
        {
            yield return progress;

            switch (progress) {
                case WFCUncollapsible wfcUncollapsible:
                    yield break;

                case WFCContradiction contradiction:
                    if (!collapseRequired) yield break;

                    foreach (WFCProgress anotherProgress in Reset())
                    {
                        yield return anotherProgress;
                    }
                    break;

                case WFCMapCellCollapsed mapCellCollapsed: break;
                case WFCMapCellModuleRemoved mapCellModuleRemoved: break;
                default: break;
            }
        }
    }

    yield return new WFCCollapsed();
}
```

La idea de esta operación es ir colapsando la celda de menor entropía en un momento dado de nuestra función de onda, propagando la información y restricciones de este colapso al resto de celdas, repitiendo este proceso hasta que todas las celdas se encuentren con un estado definido, es decir, que todas las celdas estén colapsadas.

- **Colapsar celda de menor entropía**

Esta operación se encarga de colapsar la celda de menor entropía a un módulo cualquiera que no haya sido definido como inválido para la celda en cuestión. La celda con menor entropía la definimos previamente como la celda que en un momento dado de nuestra función de onda cuenta con un menor número de estados posibles.

```
public override IEnumerable<WFCProgress> CollapseMinEntropyMapCell()
{
    MapCell minEntropyCell = this.wfc.mapCells.GetMinEntropyCell();
    if (minEntropyCell == null)
    {
        yield return new WFCUncollapsible("Imposible encontrar solucion para el WFC actual (Aplicar reset)");
        yield break;
    }
    else {
        foreach (WFCProgress progress in this.CollapseMapCell(minEntropyCell)) {
            yield return progress;
        }
    }
}
```

- **Colapsar celda**

Esta operación se encarga de hacer colapsar una celda a un módulo cualquiera que no haya sido definido como inválido para la celda en cuestión.

```
public override IEnumerable<WFCProgress> CollapseMapCell(MapCell mapCell)
{
    List<Module> mapCellValidOptions = mapCell.GetValidOptions();
    int moduleToCollapseIndex = Random.Range(0, mapCellValidOptions.Count);
    Module moduleToCollapse = mapCellValidOptions[moduleToCollapseIndex];
    return this.CollapseMapCellToModule(mapCell, moduleToCollapse);
}
```

- **Colapsar celda a un módulo**

Esta operación se encarga de colapsar una celda a un módulo en concreto.

```

public override IEnumerable<WFCProgress> CollapseMapCellToModule(MapCell mapCell, Module module)
{
    List<Module> mapCellOptions = mapCell.GetOptions();
    List<Module> mapCellInvalidOptions = mapCell.GetInvalidOptions();
    List<Module> newInvalidOptions = mapCellOptions.Where(moduleOption => !mapCellInvalidOptions.Contains(moduleOption))
        .Where(moduleOption => !moduleOption.Equals(module)).ToList();

    mapCellInvalidOptions.AddRange(newInvalidOptions);
    foreach (Module removedModule in newInvalidOptions) {
        yield return new WFCMapCellModuleRemoved(mapCell, removedModule);
    }

    yield return new WFCMapCellCollapsed(mapCell, module);
    foreach (WFCProgress progress in this.Propagate(mapCell)) {
        yield return progress;
    }
}

```

- Propagación

Esta es la operación más importante del algoritmo, pues es la encargada, como su nombre indica, de la propagación de restricciones a lo largo del grid.

```

public override IEnumerable<WFCProgress> Propagate(MapCell source)
{
    Stack<MapCell> mapCellStack = new Stack<MapCell>();
    //HashSet<MapCell> mapCellsVisited = new HashSet<MapCell>();
    //mapCellsVisited.Add(source);
    mapCellStack.Push(source);

    while (mapCellStack.Count > 0)
    {
        MapCell currentMapCell = mapCellStack.Pop();
        foreach (Vector3Int neighborMapCellCoords in currentMapCell.GetMapCellNeighborsCoords(this.wfc))
        {
            MapCell neighborMapCell = this.wfc.mapCells.GetCellByCoords(neighborMapCellCoords);
            //if (mapCellsVisited.Contains(neighborMapCell)) continue;

            List<Module> currentMapCellValidOptions = currentMapCell.GetValidOptions();
            List<Module> neighborMapCellValidOptions = neighborMapCell.GetValidOptions();

            foreach (Module neighborMapCellValidOption in neighborMapCellValidOptions)
            {
                bool toRemove = !currentMapCellValidOptions.Any(currentMapCellValidOption =>
                    currentMapCellValidOption.constraint.ShouldRemoveModule(neighborMapCellValidOption.constraint,
                    currentMapCell.GetCoords(),
                    neighborMapCellCoords) == false);

                if (toRemove)
                {
                    foreach (WFCProgress progress in RemoveMapCellOption(neighborMapCell, neighborMapCellValidOption)) {
                        yield return progress;
                    }
                    mapCellStack.Push(neighborMapCell);
                }
            }//Fin foreach

            if (neighborMapCell.IsContradicted())
            {
                yield return new WFCContradiction(neighborMapCell, null, currentMapCell);
                yield break;
            }
            if (neighborMapCell.IsCollapsed())
            {
                yield return new WFCMapCellCollapsed(neighborMapCell, neighborMapCell.GetValidOptions()[0]);
            }

            //mapCellsVisited.Add(neighborMapCell);
        }//Fin foreach
    }//Fin while
}

```

Lo que hacemos es crear una pila donde se almacenan las celdas a las que se le ha eliminado, como mínimo, un módulo como posibilidad de colapso. Tomando una celda de la pila obtenemos las celdas vecinas a esta y para cada una de ellas vemos los módulos disponibles para colapsar. Para cada módulo, determinamos si hay alguna posibilidad de que sea válido en el espacio o debe ser eliminado por restricciones. La propagación habrá terminado cuando la pila se encuentre vacía.

- **Eliminar módulo como opción de colapso para una celda**

Esta operación se encarga de que una celda no pueda colapsar a un determinado módulo.

```
public override IEnumerable<WFCProgress> RemoveMapCellOption(MapCell mapCell, Module module)
{
    mapCell.GetInvalidOptions().Add(module);
    yield return new WFCMapCellModuleRemoved(mapCell, module);
}
```

- **Eliminar módulo como opción de colapso para una celda, siendo módulo y celda marcados por el usuario**

Esta operación se encarga de que una celda no pueda colapsar a un determinado módulo, siendo estos dos parámetros marcados por el usuario y conllevando por tanto una propagación de este suceso.

```
public override IEnumerable<WFCProgress> RemoveMapCellOptionFromUser(MapCell mapCell, Module module)
{
    mapCell.GetInvalidOptions().Add(module);
    yield return new WFCMapCellModuleRemoved(mapCell, module);
    foreach (WFCProgress progress in this.Propagate(mapCell)){
        yield return progress;
    }
}
```

- **Reiniciar**

Esta operación se encarga de devolver la función de onda a su estado inicial.

```

public override IEnumerable<WFCProgress> Reset()
{
    foreach (MapCell mapCell in this.wfc.mapCells)
    {
        mapCell.GetInvalidOptions().Clear();
    }
    yield return new WFCReset();
}

```

#### 4.3.4 Especificaciones

Hemos visto de manera abstracta todo lo que necesitamos para tener una función de onda sobre la que aplicar el algoritmo WFC. Ahora llega el momento de especificar, mediante herencia, la función de onda con la que queremos trabajar. Nuestro objetivo es trabajar con un entorno 2D con una serie de imágenes que tienen una serie de restricciones de adyacencia que cumplir. Sabiendo esto se han implementado las siguientes clases:

- **MapGrid2D**

Como queremos un entorno 2D, se ha creado un grid específico para ello, donde especificaremos su dimensión (anchura, altura) e instanciamos las celdas que lo van a componer con los respectivos módulos disponibles.

```

public class MapGrid2D : MapGrid
{
    /// <summary> Anchura de la cuadricula.
    public int width;

    /// <summary> Altura de la cuadricula.
    public int height;

    /// <summary> Constructor para una cuadricula en un entorno de dos dimensiones.
    public MapGrid2D(int width,int height,List<Module> optionsPerMapCell) {
        this.width = width;
        this.height = height;
        this.mapCells = new List<MapCell>();
        CreateMapGrid2D(optionsPerMapCell);
    }

    /// <summary> Metodo auxiliar para general el grid con las celdas correspondientes.
    protected virtual void CreateMapGrid2D(List<Module> optionsPerMapCell) {
        for (int x = 0; x < this.width; x++) {
            for (int y = 0; y < this.height; y++) {

                Vector3Int mapCellCoords = new Vector3Int(x, y, 0);
                MapCell2D mapCell2D = new MapCell2D(mapCellCoords, optionsPerMapCell,true);
                mapCells.Add(mapCell2D);
            }
        }
    }

    public override Vector3Int GetDimensions()
    {
        return new Vector3Int(this.width, this.height, 0);
    }
}

```

- **MapCell2D**

Al igual que para el grid, se ha creado un tipo de celda específica para trabajar en un entorno 2D con el objetivo de definir qué celdas vecinas tiene una celda ubicada en una posición del grid.

```
public class MapCell2D : MapCell
{
    /// <summary> Constructor para una celda en un entorno de dos dimensiones.
    5 referencias
    public MapCell2D(Vector3Int coords, List<Module> options, bool renderizable)
    {
        this.coords = coords;
        this.options = options;
        this.invalidOptions = new List<Module>();
        this.renderizable = renderizable;
    }

    public override List<Vector3Int> GetMapCellNeighborsCoords<TMapGrid>(TMapGrid mapGrid)
    {

        bool hasLeftNeighbor;
        bool hasRightNeighbor;
        bool hasTopNeighbor;
        bool hasBottomNeighbor;

        List<Vector3Int> neighborsCoords = new List<Vector3Int>();

        MapGrid2D mapGrid2D = mapGrid as MapGrid2D;
        if (mapGrid2D == null) return null;

        Vector3Int currentPosition = this.GetCoords();

        hasLeftNeighbor = currentPosition.x - 1 >= 0 && currentPosition.y >= 0 && currentPosition.y < mapGrid2D.height ? true : false;
        hasRightNeighbor = currentPosition.x + 1 < mapGrid2D.width && currentPosition.y >= 0 && currentPosition.y < mapGrid2D.height ? true : false;
        hasTopNeighbor = currentPosition.y + 1 < mapGrid2D.height && currentPosition.x >= 0 && currentPosition.x < mapGrid2D.width ? true : false;
        hasBottomNeighbor = currentPosition.y - 1 >= 0 && currentPosition.x >= 0 && currentPosition.x < mapGrid2D.width ? true : false;

        if (hasLeftNeighbor) neighborsCoords.Add(new Vector3Int(currentPosition.x - 1, currentPosition.y, currentPosition.z));
        if (hasRightNeighbor) neighborsCoords.Add(new Vector3Int(currentPosition.x + 1, currentPosition.y, currentPosition.z));
        if (hasTopNeighbor) neighborsCoords.Add(new Vector3Int(currentPosition.x, currentPosition.y + 1, currentPosition.z));
        if (hasBottomNeighbor) neighborsCoords.Add(new Vector3Int(currentPosition.x, currentPosition.y - 1, currentPosition.z));

        return neighborsCoords;
    }
}
```

- **AdjacencyImageConstraint2D**

Esta clase se encarga de implementar la restricción de adyacencia de una imagen.



```

public class AdjacencyImageConstraint2D : ModuleConstraint
{
    /// <summary> Imagen sobre la que se aplica la restricción de adjacencia.
    public Sprite sprite;

    /// <summary> Imágenes compatibles a la izquierda.
    public List<Sprite> validLeftNeighbors;

    /// <summary> Imágenes compatibles a la derecha.
    public List<Sprite> validRightNeighbors;

    /// <summary> Imágenes compatibles arriba.
    public List<Sprite> validTopNeighbors;

    /// <summary> Imágenes compatibles abajo.
    public List<Sprite> validBottomNeighbors;

    /// <summary> Constructor de restricción de adjacencia con imágenes.
    3 referencias
    public AdjacencyImageConstraint2D(Sprite sprite ,List<Sprite> validLeftNeighbors, List<Sprite> validRightNeighbors,
        List<Sprite> validTopNeighbors, List<Sprite> validBottomNeighbors)
    {
        this.sprite = sprite;
        this.validLeftNeighbors = validLeftNeighbors;
        this.validRightNeighbors = validRightNeighbors;
        this.validTopNeighbors = validTopNeighbors;
        this.validBottomNeighbors = validBottomNeighbors;
    }

    public override bool ShouldRemoveModule<TModuleConstraint>(TModuleConstraint otherModuleConstraint, Vector3Int thisModuleConstraintCoords,
        Vector3Int otherModuleConstraintCoords)
    {
        AdjacencyImageConstraint2D otherAdjacencyImageConstraint2D = otherModuleConstraint as AdjacencyImageConstraint2D;
        if (otherAdjacencyImageConstraint2D == null) return true;

        bool shouldRemoveOtherModule = false;

        if (thisModuleConstraintCoords.x > otherModuleConstraintCoords.x)

            shouldRemoveOtherModule = !this.validLeftNeighbors.Contains(otherAdjacencyImageConstraint2D.sprite);

        else if (thisModuleConstraintCoords.x < otherModuleConstraintCoords.x)

            shouldRemoveOtherModule = !this.validRightNeighbors.Contains(otherAdjacencyImageConstraint2D.sprite);

        else if (thisModuleConstraintCoords.y > otherModuleConstraintCoords.y)

            shouldRemoveOtherModule = !this.validBottomNeighbors.Contains(otherAdjacencyImageConstraint2D.sprite);

        else if (thisModuleConstraintCoords.y < otherModuleConstraintCoords.y)

            shouldRemoveOtherModule = !this.validTopNeighbors.Contains(otherAdjacencyImageConstraint2D.sprite);

        return shouldRemoveOtherModule;
    }
}

```

- **Module2D**

Al igual que se generó una clase abstracta para representar un módulo de nuestra función de onda, se ha vuelto a generar otra clase abstracta para los módulos que vayan a trabajar en un entorno 2D.

```
public abstract class Module2D : Module {
    /// <summary> Imagen del modulo.
    public Sprite sprite;
}
```

- **AdjacencyImageModule2D**

Esta clase implementa un módulo para un entorno 2D con restricciones de adyacencia respecto a la imagen del mismo.

```
public class AdjacencyImageModule2D : Module2D
{
    /// <summary> Constructor de modulo 2D con restriccción de adjacencia con imagene ...
    3 referencias
    public AdjacencyImageModule2D(Sprite sprite, AdjacencyImageConstraint2D adjacencyImageConstraint2D)
    {
        this.sprite = sprite;
        this.display = sprite != null ? sprite.name : null;
        this.constraint = adjacencyImageConstraint2D;
    }
}
```

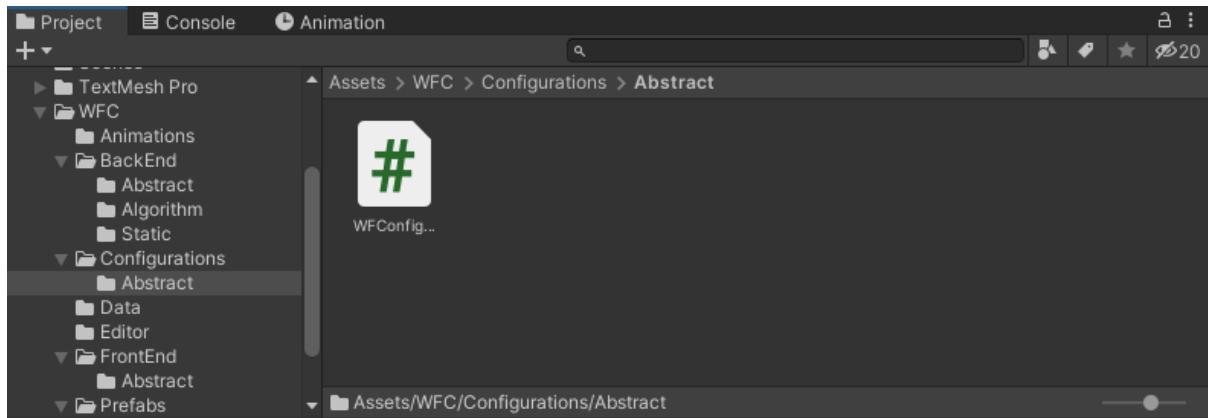
## 4.4 Entrada de usuario

En este punto tenemos todo lo necesario para ejecutar nuestro algoritmo WFC, pero no tenemos ninguna entrada por parte del usuario con la que trabajar. En este apartado vamos a solucionar ese aspecto con utilidades que Unity nos proporciona a través de su interfaz.

### 4.4.1 Scriptable Objects

Podemos ver un Scriptable Object [17] como un script que simplemente es usado como un contenedor y manejador de información sobre el cual crear diferentes objetos con diferentes datos, siendo este el medio que usaremos para almacenar las configuraciones deseadas sobre las que queremos ejecutar el algoritmo WFC.

Lo primero que se ha hecho es especificar aspectos comunes que deben tener las configuraciones que se vayan a crear mediante clases abstractas. Están ubicadas en el siguiente script:



Lo más básico que debe tener un objeto de configuración es la posibilidad de crear una función de onda sobre la que aplicar el algoritmo WFC en base a los datos de entrada.

```
public abstract class WFConfigObject : ScriptableObject
{
    /// <summary> Metodo que nos permite generar un entorno de ejecucion del algoritmo WFC ...
    3 referencias
    public abstract WFCRunner CreateSpace();
}
```

El siguiente paso es el almacenamiento de los módulos y restricciones de la configuración que componen la función de onda.

```
public abstract class WFConfigObject<TModule, TModuleConstraint, TMapGrid> : WFConfigObject where TModule : Module
    where TMapGrid : MapGrid where TModuleConstraint : ModuleConstraint
{
    /// <summary> Modulos de la configuracion.
    [SerializeField]
    public List<TModule> modules = new List<TModule>();

    /// <summary> Restricciones de los modulos de la configuracion.
    [SerializeField]
    public List<TModuleConstraint> constraints = new List<TModuleConstraint>();

    /// <summary> Metodo para obtener los modulos de la configuracion.
    0 referencias
    public virtual List<TModule> GetModules() {
        return modules;
    }

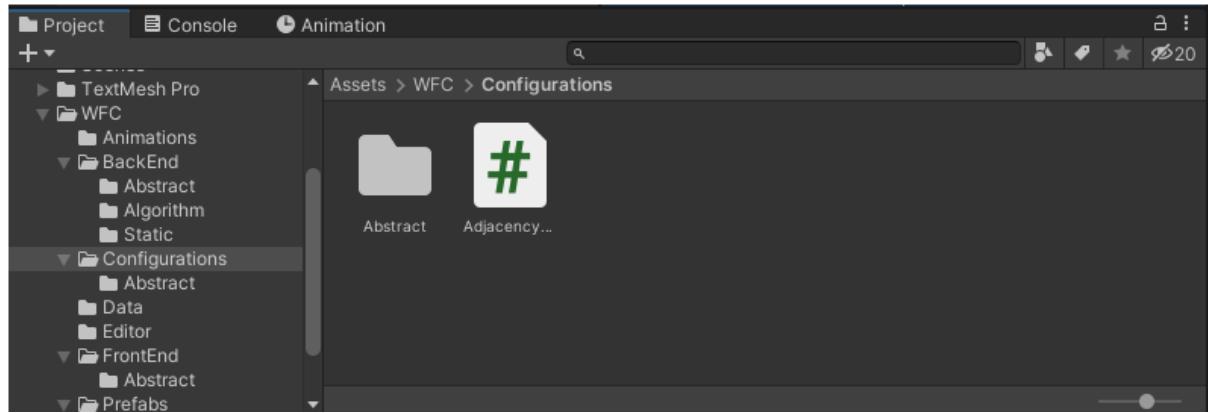
    /// <summary> Metodo para obtener las restricciones de los modulos de la configuracion.
    0 referencias
    public virtual List<TModuleConstraint> GetConstraints()...

    /// <summary> Metodo que genera la funcion de onda bajo el algoritmo WFC.
    2 referencias
    public abstract WFCRoutines<TMapGrid> CreateWFC();

    3 referencias
    public override WFCRunner CreateSpace()...
}
```

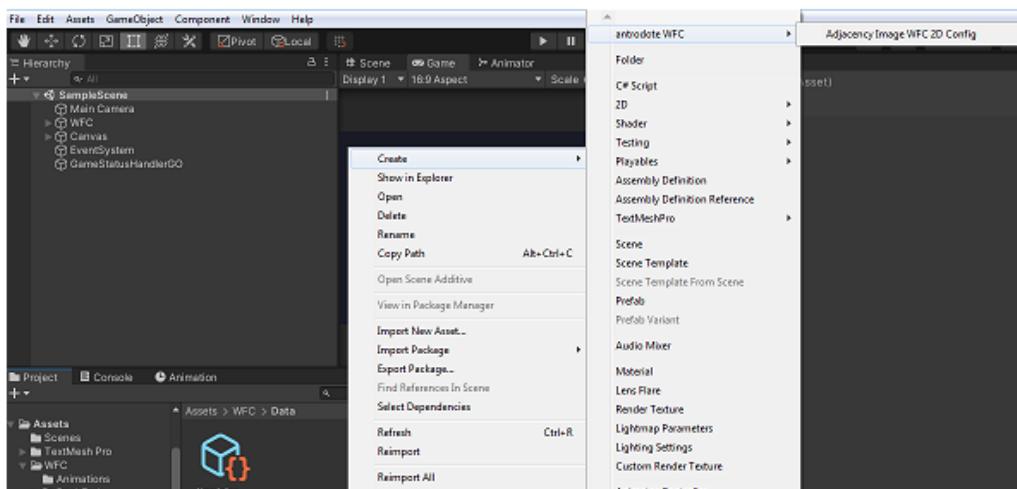
Unity nos permitirá ver estos objetos en el inspector e introducir los datos con los que queramos trabajar desde una visualización por defecto. A pesar de tener una visualización por defecto se pueden crear visualizaciones personalizadas, como haremos posteriormente con el fin de añadir botones y funcionalidades extra.

Hasta este momento todo ha sido comentado como con anterioridad, desde un punto de vista abstracto. Sin embargo, para crear una configuración debemos dejar esa abstracción de lado y concretar el tipo de módulo, restricciones y la cuadrícula sobre la que queremos que trabaje el algoritmo WFC. Para ello se ha creado un script con esa especificación.



En este script podemos establecer el tipo de módulo, restricción y cuadrícula mediante herencia, aunque lo primero que se hace es añadir un atributo `CreateAssetMenu`, que nos permite decir a Unity que debe añadir el Scriptable Object en cuestión como un objeto que puede ser creado bajo una ruta determinada en el menú de creación de Unity, así como establecer el nombre inicial que tendrán estos objetos en la creación.

```
[CreateAssetMenu(fileName = "New Adjacency Image WFC 2D Config", menuName = "antrodote WFC/Adjacency Image WFC 2D Config")]
(1) Script de Unity | 11 referencias
public class AdjacencyImageModule2DConfig : WFConfigObject<AdjacencyImageModule2D, AdjacencyImageConstraint2D, MapGrid2D>
```



El script añade otros parámetros no contemplados con las clases abstractas anteriores, como puede ser el tamaño del grid y su envoltura.

```
public class AdjacencyImageModule2DConfig : WFCConfigObject<AdjacencyImageModule2D, AdjacencyImageConstraint2D, MapGrid2D>
{
    /// <summary> Vector para la configuración de una cuadrícula de dos dimensiones.
    public Vector2Int size = new Vector2Int(0, 0);

    /// <summary> Restricción de adyacencia sobre imágenes para los límites de la cu ...
    public AdjacencyImageConstraint2D wrapper = new AdjacencyImageConstraint2D(null, new List<Sprite>(), new List<Sprite>(), new List<Sprite>(), new List<Sprite>());
}
```

Esta envoltura nos permite restringir los límites del grid para que ciertas celdas no puedan colapsar a ciertos módulos.

Además, el script debe encargarse de crear la función de onda al definir la siguiente función abstracta proveniente de la herencia.

```
public override WFCRoutines<MapGrid2D> CreateWFC()
{
    if (this.size.x == 0 || this.size.y == 0)
    {
        Debug.Log("Invalid size for the grid");
        return null;
    }

    for (int i = 0; i < modules.Count; i++)
    {
        modules[i].constraint = constraints[i];
    }

    List<Module> optionsPerCell = new List<Module>();
    optionsPerCell.AddRange(modules);
    MapGrid2D mapGrid = new MapGrid2D(this.size.x, this.size.y, optionsPerCell);
    WFCRoutines<MapGrid2D> wfcSolver = new WFCRoutines<MapGrid2D>(mapGrid);
    CreateWrapper(wfcSolver);
    return wfcSolver;
}
```

#### 4.4.1.1 Serialización

Si nos salimos de los tipos más primitivos y queremos usar nuestros propios objetos para almacenar información, la serialización es un aspecto crítico que hay que tener en cuenta ya que es fuente de errores a la hora de trabajar con Scriptable Objects. La serialización es la base del editor de Unity y básicamente consiste en un proceso de codificación de un objeto en un medio de almacenamiento pudiendo ser reconstruido en un momento posterior. Esta serialización cuenta con limitaciones como, por ejemplo, no poder usarse para clases abstractas, siendo algo a tener en cuenta puesto que, en caso de saltarnos esta limitación, la información se perderá una vez cerremos Unity o empecemos a ejecutar la escena.

```

public abstract class WFConfigObject<TModule, TModuleConstraint, TMapGrid> : WFConfigObject where TModule : Module
    where TMapGrid : MapGrid where TModuleConstraint : ModuleConstraint
{

    /// <summary> Modulos de la configuracion.
    [SerializeField]
    public List<TModule> modules = new List<TModule>();

    /// <summary> Restricciones de los modulos de la configuracion.
    [SerializeField]
    public List<TModuleConstraint> constraints = new List<TModuleConstraint>();
}

```

```

[System.Serializable]
11 referencias
public class AdjacencyImageModule2D : Module2D

```

```

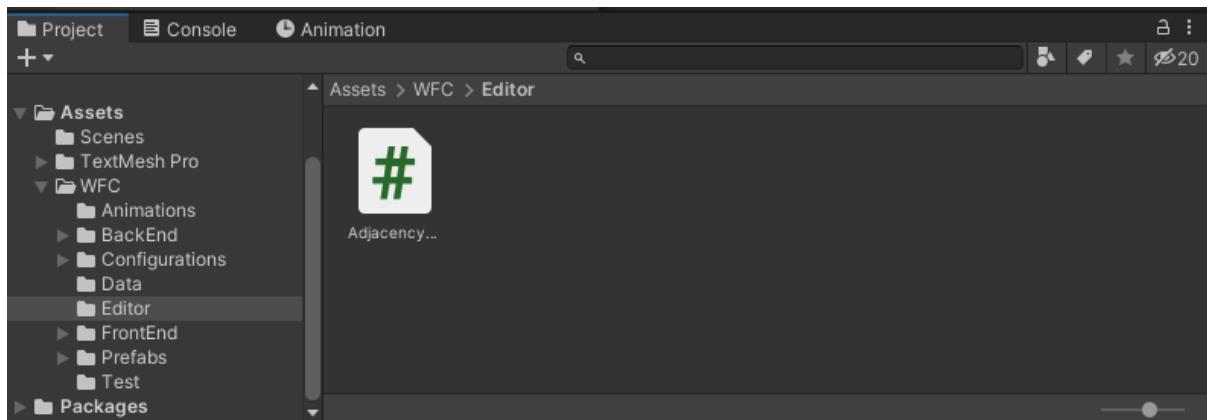
[System.Serializable]
10 referencias
public class AdjacencyImageConstraint2D : ModuleConstraint

```

La serialización es automática en atributos públicos y de tipos genéricos pero, si queremos trabajar con atributos como listas, es necesario especificar que son serializables mediante atributos en C#, siendo esto también aplicable para las clases como se puede ver en las imágenes anteriores. Se ha realizado una separación de los módulos y de sus restricciones en dos listas debido a las limitaciones de la serialización, ya que, puesto que los módulos hacen uso de abstracción en la especificación de restricciones tienen el impedimento de ser serializados al completo, aunque otra posibilidad para solucionar esto sería crear una clase intermedia con tipos definidos para que sea completamente serializable.

## 4.4.2 Custom Editor

Unity nos permite crear un editor personalizado para un Scriptable Object. Para ello se ha creado una carpeta llamada "Editor". Esta carpeta contiene el script que gestiona la visualización en el inspector de Unity del Scriptable Object que deseemos.



Para realizar la visualización customizada de un Scriptable Object es necesario que el script que gestiona esa visualización tenga la siguiente estructura.

```
[CustomEditor(typeof(AdjacencyImageModule2DConfig))]
@ Script de Unity | 0 referencias
public class AdjacencyImageModule2DEditor : Editor
```

Primero se le asocia el atributo de CustomEditor a la clase, indicando el Scriptable Object del que va a gestionar su visualización en el inspector. Por otro lado, la clase debe heredar de la clase Editor proporcionada por Unity.

Para definir la visualización deseada, es necesario definir en el inspector el siguiente método heredado de la clase Editor:

```
public override void OnInspectorGUI()
{
    AdjacencyImageModule2DConfig config = target as AdjacencyImageModule2DConfig;
    if (config == null) return;

    Rect totalVerticalRect = EditorGUILayout.BeginVertical(GUILayout.Height(100));

    DisplayMapGridWidthParameter(config);
    DisplayMapGridHeightParameter(config);
    DisplayMapGridWrapperParameter(config);
    ClearAllSpriteHandleButton(config);
    AddFromSpriteSheetHandleButton(config);
    LoadSpritesEventHandler(config);

    EditorGUILayout.EndVertical();

    DisplayConfigurationModules(config);
    AddNewModuleHandleButton(config);
}
```

La herencia también lleva a tener un atributo llamado target, que hace referencia al objeto que está siendo inspeccionado.

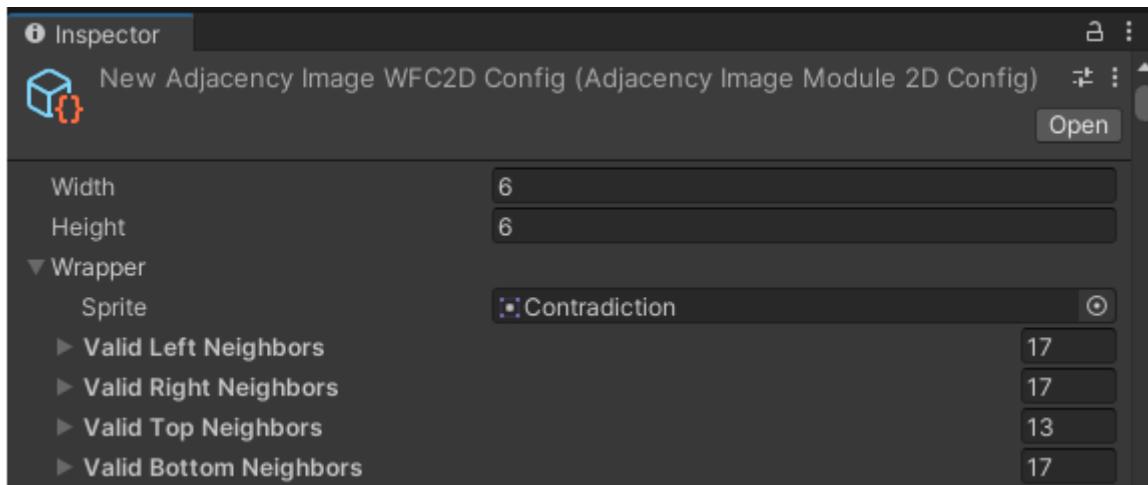
Unity proporciona muchas funciones para personalizar el editor asociado al objeto inspeccionado por lo que no vamos a entrar en profundidad en ellas y les remitimos a leer la documentación proporcionada por Unity al respecto en caso de que no se entienda qué hace alguna función en el código. Además, el código asociado para hacer un editor personalizado es largo y complejo. En cambio, algo que nos parece interesante exponer es cómo se pueden visualizar objetos complejos que son serializables. Esto lo permite un atributo proporcionado por la herencia de Editor, como se puede ver en el siguiente ejemplo:

```

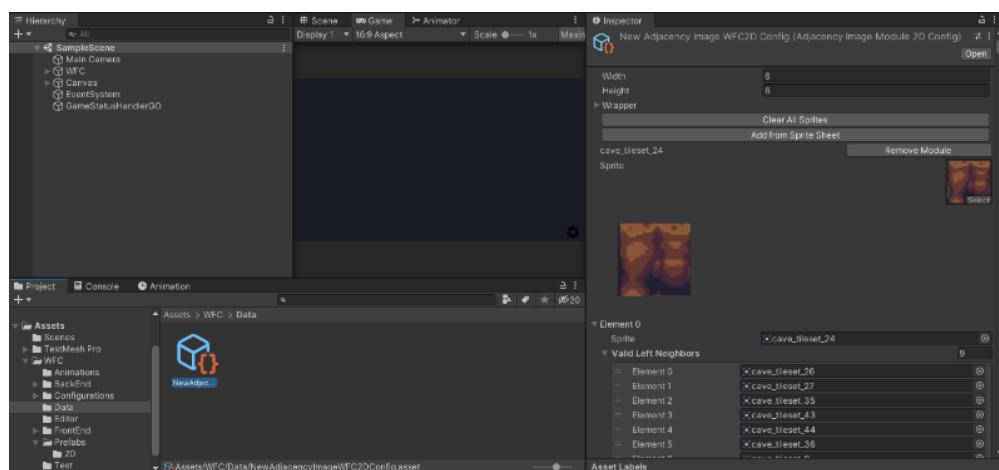
private void DisplayMapGridWrapperParameter(AdjacencyImageModule2DConfig config)
{
    SerializedProperty wrapperConstraint = serializedObject.FindProperty("wrapper");
    EditorGUI.BeginChangeCheck();
    if (EditorGUILayout.PropertyField(wrapperConstraint, true) && EditorGUI.EndChangeCheck())
    {
        Undo.RecordObject(target, "Change wrapper");
        EditorUtility.SetDirty(config);
    }
}

```

Como se puede ver, con el atributo `serializedObject` y su función `FindProperty` podremos encontrar una propiedad serializable compleja y mostrarla por el inspector.

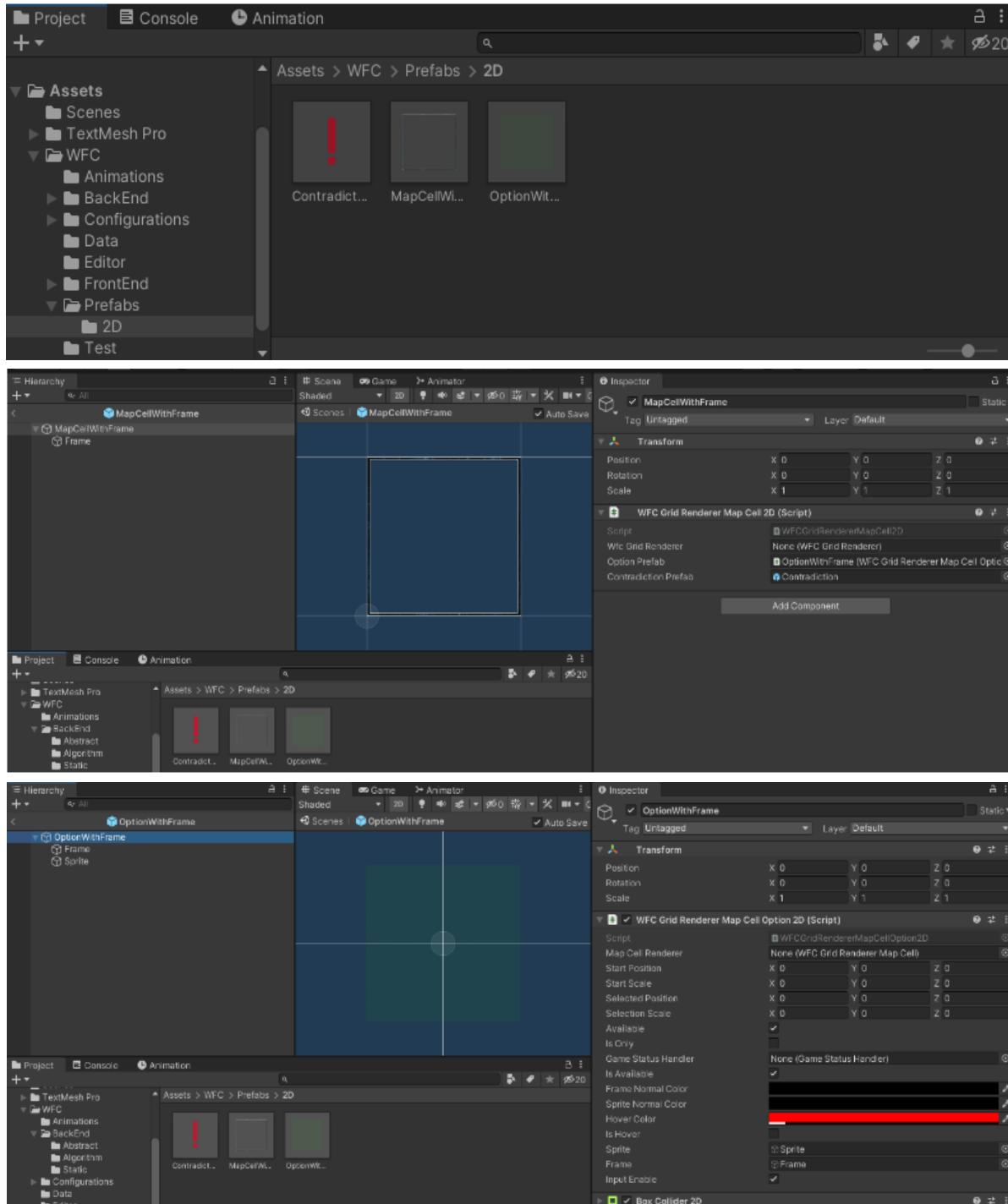


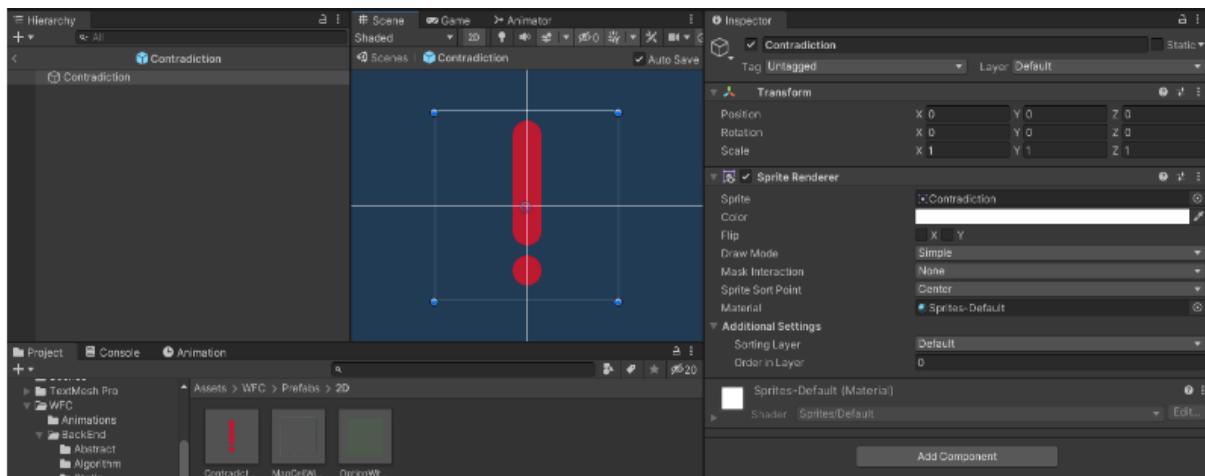
Después de haber realizado los pasos explicados anteriormente, se ha terminado generando la siguiente interfaz con el usuario para una configuración de módulos 2D con restricciones de adyacencia respecto a imágenes.



## 4.5 Frontend

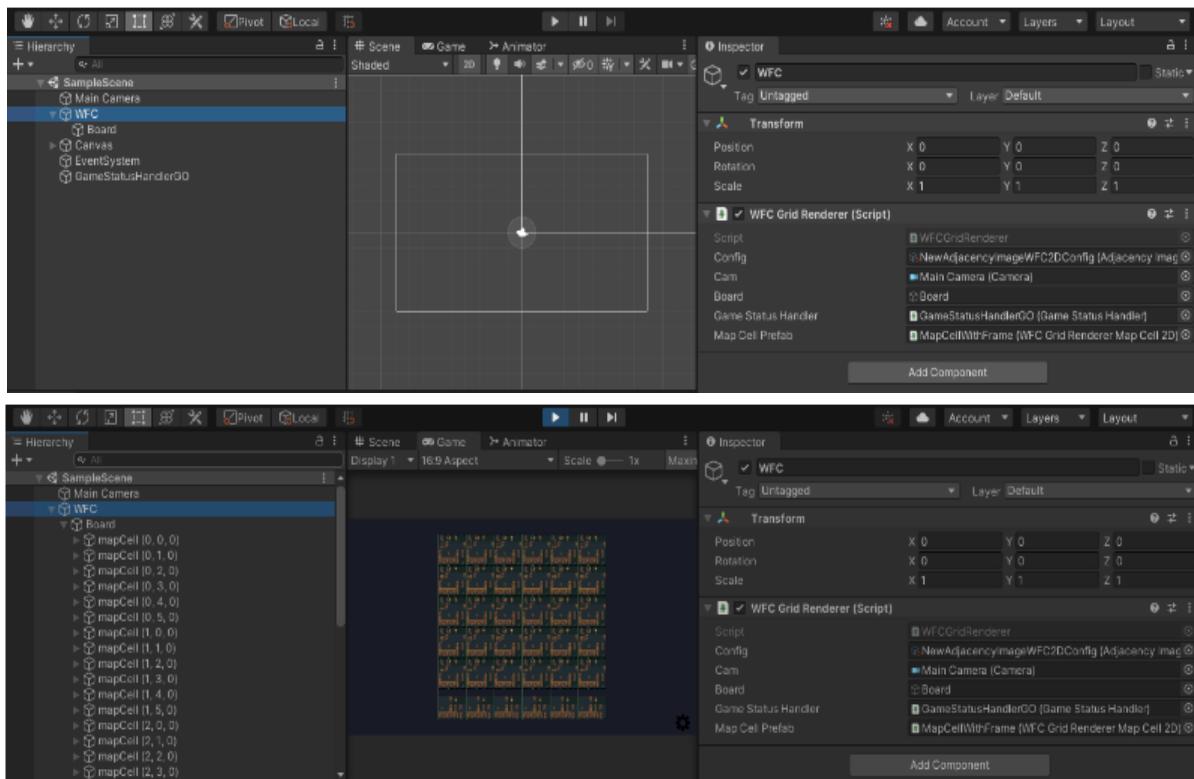
Nuestro objetivo en la capa de visualización es mostrar la función de onda y su progreso a lo largo de la ejecución del algoritmo WFC. Para ello se han creado los siguientes prefabs para representar una celda, una opción y una contradicción.



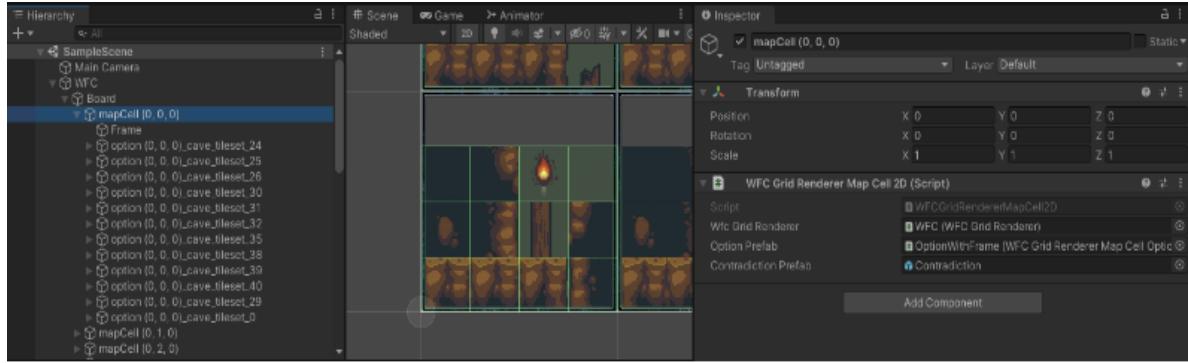


Estos prefabs, la configuración creada sobre la que ejecutar el algoritmo WFC y otros parámetros tomando como punto de partida el script **WFC Grid Renderer** son usados para crear el espacio de manera visual.

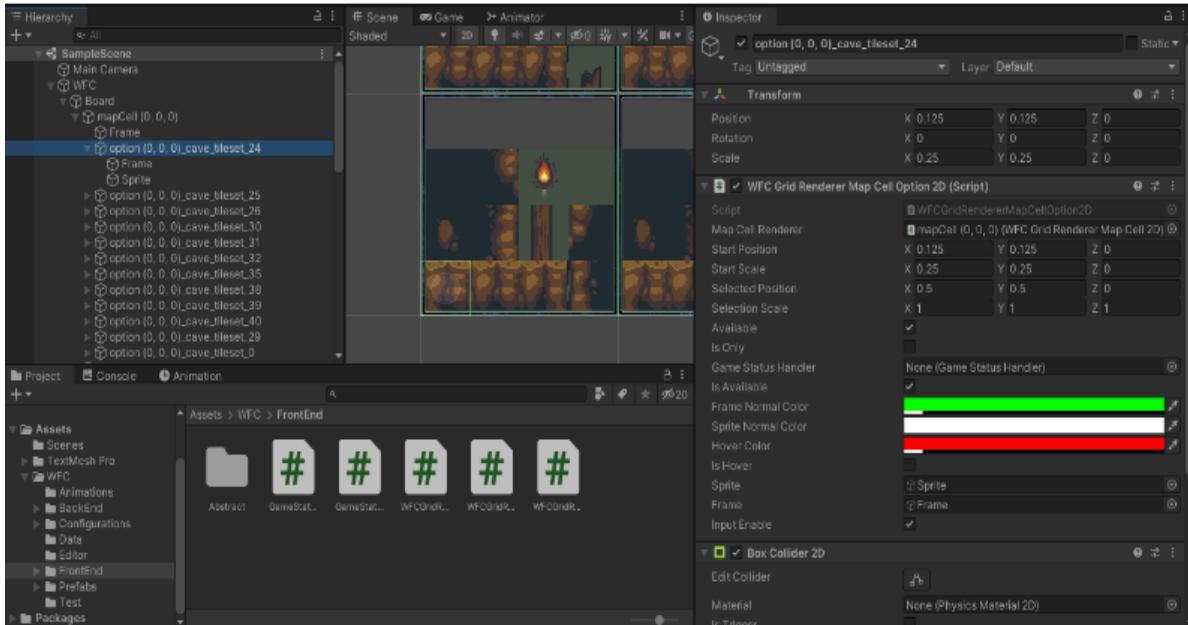
**WFC Grid Renderer** es el nexo de unión entre la visualización de la función de onda y el algoritmo WFC. Desde él se mandan a ejecutar las diferentes operaciones del algoritmo WFC y se actualiza la visualización de la función de onda en base al progreso del algoritmo. Además, gestiona la correspondencia entre una celda de la función de onda y su visualización.



Las celdas renderizadas tienen asociado un script que controla su visualización además de gestionar la correspondencia entre sus módulos disponibles y la visualización de los mismos.

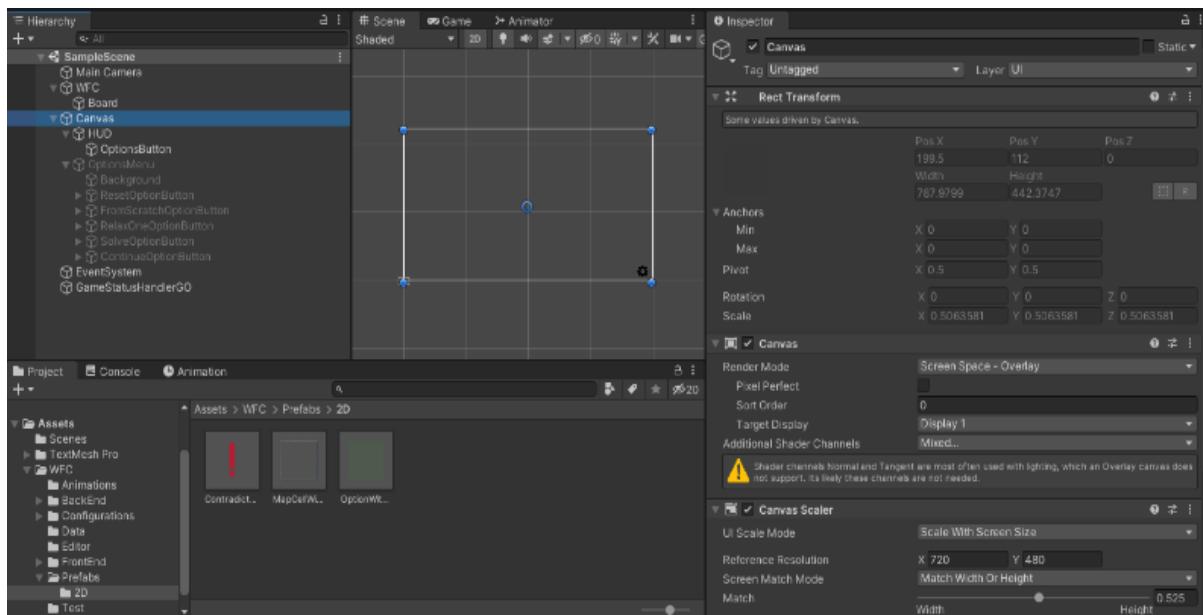


Del mismo modo, cada uno de los módulos también tiene asociado un script que controla aspectos como la interacción con el usuario, animación de descarte y *el colapso de la celda a él*.



## 4.5.1 UI

Se ha creado una interfaz de usuario para abarcar todas las operaciones del algoritmo WFC. Todo elemento de interfaz de usuario en Unity necesita como mínimo un Canvas [18] para ser renderizado. Si no existe previamente al crear el primer elemento de interfaz de usuario se genera automáticamente.



El canvas tiene varios componentes con algunas opciones en algunos de ellos que es necesario comprender y establecer. Estas opciones son:

- **Canvas -> Render Mode**

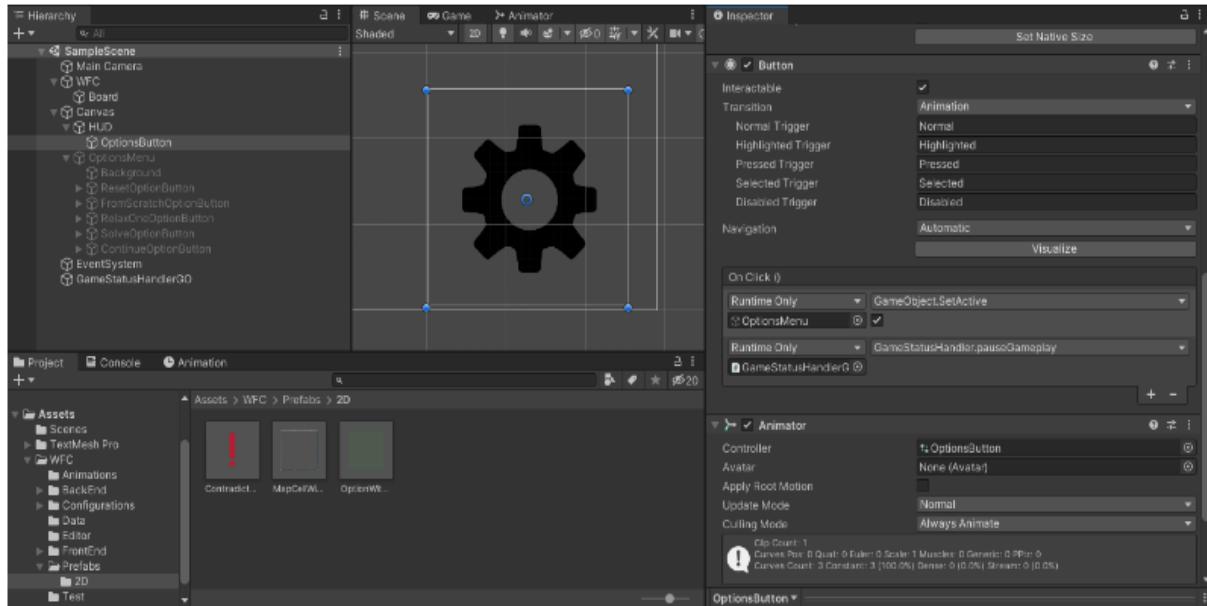
Esta opción, como su nombre indica, especifica el modo de renderizado del canvas, que en nuestro caso es **Screen Space - Overlay**, para que el canvas se superponga a los demás elementos de la escena.

- **Canvas Scaler -> UI Scale Mode**

Esta opción determina cómo se van a escalar los diferentes elementos de la interfaz, siendo en nuestro caso **Scale With Screen Size**, que se utiliza para que los elementos de la interfaz se adapten al tamaño de la pantalla tomando como referencia una determinada resolución.

El canvas se puede componer de varias interfaces para mostrar al usuario y la visualización de una u otra simplemente se basa en un sistema de agrupación de las diferentes interfaces a mostrar, donde estas agrupaciones se van activando y desactivando según los intereses que tengamos.

Los botones son los elementos de interfaz que vamos a utilizar para implementar funcionalidad en base a la interacción con el usuario. Cuando detectan el click del usuario nos proporcionan la posibilidad de ejecutar una acción, como se puede ver en la siguiente imagen.

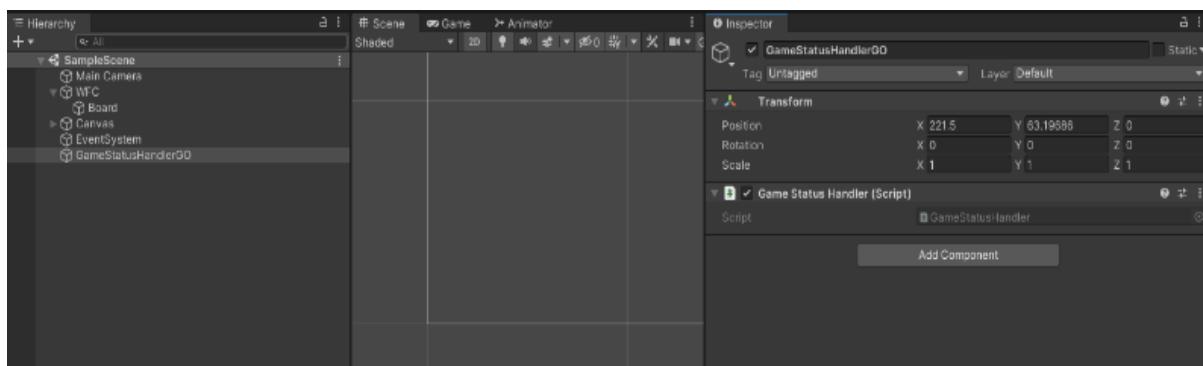


Teniendo esto en cuenta, se ha creado un botón en los controles del usuario que ejecutará cada operación del algoritmo WFC que ha quedado sin abarcar, tal y como se ve a continuación:



#### 4.5.1.1 Interrupción de la ejecución de las mecánicas principales de la escena

Al mostrar una interfaz de menú como la que se ha propuesto para el proyecto, es necesario desactivar cualquier interacción que exista fuera de esta. Esto ha llevado a crear un script que gestiona el estado de la escena que notifica cuando este estado cambia mediante el uso de eventos con el fin de desactivar o activar las interacciones principales con la función de onda.



```
public enum GameStatus
{
    Gameplay,
    Paused
}
```

```
public class GameStatusHandler : MonoBehaviour
{
    private GameStatus currentGameStatus;
    public delegate void GameStatusChangeHandler(GameStatus newGameStatus);
    private event GameStatusChangeHandler onGameStatusChanged;

    void Start()
    {
        this.currentGameStatus = GameStatus.Gameplay;
    }
}
```

```
public GameStatus GetGameStatus()...
0 referencias
public void PauseGameplay()
{
    this.SetGameStatus(GameStatus.Paused);
}
0 referencias
public void RenaudeGameplay()
{
    this.SetGameStatus(GameStatus.Gameplay);
}
2 referencias
public void SetGameStatus(GameStatus newGameStatus)
{
    if ( newGameStatus.Equals(currentGameStatus)) return;

    currentGameStatus = newGameStatus;
    onGameStatusChanged?.Invoke(newGameStatus);

}
1 referencia
public void SubscribeToGameStatusEvent(GameStatusChangeHandler gameStatusChangeHandler)
{
    onGameStatusChanged += gameStatusChangeHandler;
}
1 referencia
public void UnSubscribeToGameStatusEvent(GameStatusChangeHandler gameStatusChangeHandler)
{
    onGameStatusChanged -= gameStatusChangeHandler;
}
```

## 4.5.2 Creación de la visualización de la función de onda

Para comenzar a visualizar la función de onda tomamos como punto de partida el script **WFCGridRenderer**.

```
public class WFCGridRenderer : MonoBehaviour
{
    /// <summary> Configuracion de la que obtener la funcion de onda a la que se apl ...
    public WFCConfigObject config;

    /// <summary> Funcion onda sobre la que aplicar el algoritmo WFC.
    public WFCRunner wfc;

    /// <summary> Funcion onda auxiliar.
    public WFCRunner auxWFC = null;

    /// <summary> Camara de la escena.
    public Camera cam;

    /// <summary> Objeto de la escena que contendra la visualizacion de funcion de o ...
    public GameObject board;

    /// <summary> Controlador del estado del juego para activar o desabilitar la int ...
    public GameStatusHandler gameStatusHandler;

    /// <summary> Prefab para la visualizacion y control de las celdas de la cuadric ...
    public WFCGridRendererMapCell mapCellPrefab;

    /// <summary> Diccionario de correspondencia entre una celda de la cuadricula y ...
    public Dictionary<MapCell, WFCGridRendererMapCell> mapCellToRenderer = new Dictionary<MapCell, WFCGridRendererMapCell>();

    /// <summary> Variable para controlar si el algoritmo WFC se puede aplicar desde ...
    private bool canSolveFromScratch = true;
}
```

En el mismo momento del inicio de la ejecución de la escena, se creará la función de onda a partir de la configuración deseada.

```
public void InitWFC()
{
    this.wfc = config.CreateSpace();
    if (this.wfc == null) return;
    Debug.Log("Configuracion creada");
    SetCameraPosition(this.wfc.GetMapGrid().GetDimensions());
    this.InitGameObjects();
}

void Start()
{
    InitWFC();
}
```

Después, se instanciará en tiempo de ejecución la representación visual de cada celda de nuestra función de onda.

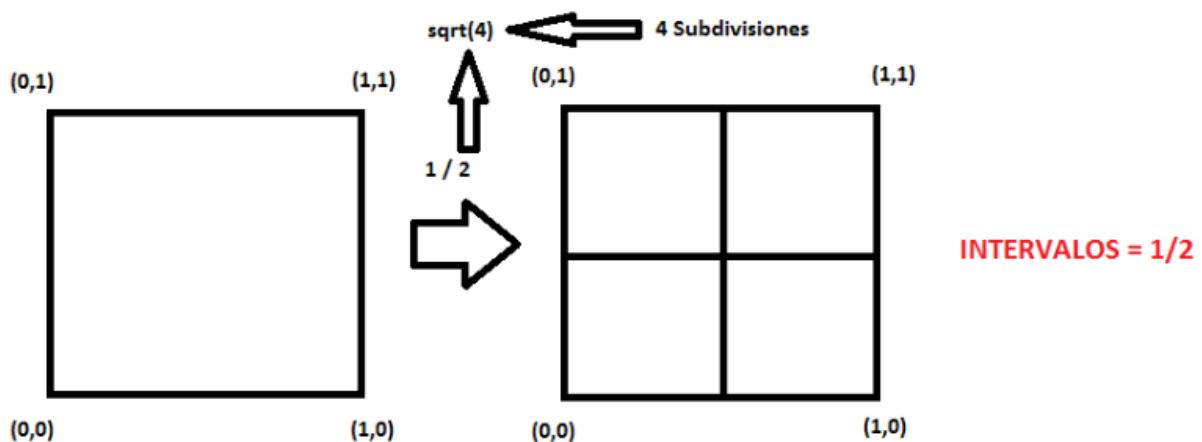
```

public void InitGameObjects()
{
    mapCellToRenderer.Clear();
    for (var i = 0; i < board.transform.childCount; i++)
    {
        Destroy(board.transform.GetChild(i).gameObject);
    }

    foreach (MapCell mapCell in this.wfc.GetMapGrid().mapCells.RenderizableMapCells())
    {
        WFCGridRendererMapCell mapCellInstance = Instantiate(mapCellPrefab, board.transform);
        mapCellInstance.transform.localScale = new Vector3(1, 1, 1);
        mapCellInstance.name = $"mapCell {mapCell.GetCoords()}";
        mapCellInstance.transform.localPosition = mapCell.GetCoords();
        mapCellToRenderer.Add(mapCell, mapCellInstance);
        mapCellInstance.OnCreated(mapCell, this);
    }
}

```

A la hora de crear la visualización en pantalla de una celda y querer visualizar también sus módulos disponibles, es necesario distribuirlos correctamente en el espacio de la celda en cuestión. Los módulos, que en nuestro caso van a ser imágenes, cuando sean seleccionados como colapso para una celda, ocuparán una unidad de Unity, que, en su esencia, es un cuadrado en el espacio. Antes de llegar a ese punto, todos los módulos disponibles para una celda deben compartir el espacio de esa unidad. Esto implica hacer una subdivisión de la unidad como si tuviésemos, por ejemplo, cuatro módulos disponibles por celda, tal cual se explica a continuación:



```

public override void OnCreated(MapCell mapCell, WFCGridRenderer wfcGridRenderer)
{
    this.wfcGridRenderer = wfcGridRenderer;
    this.mapCell = mapCell;
    List<Module> mapCellValidOptions = this.mapCell.GetValidOptions();

    int cellDivisor = Mathf.CeilToInt(Mathf.Sqrt(mapCellValidOptions.Count));
    float subCellsRange = 1f / cellDivisor;
    moduleToRenderer.Clear();

    float coordX = 0.0f;
    float coordY = 0.0f;
    int placedOptions = 0;
    foreach (Module module in mapCellValidOptions)
    {
        WFCGridRendererMapCellOption mapCellOptionInstance = Instantiate(optionPrefab, this.transform);
        float instanceCoordX = coordX + subCellsRange / 2;
        float instanceCoordY = coordY + subCellsRange / 2;
        mapCellOptionInstance.name = $"option {mapCell.GetCoords()}_{module.display}";
        mapCellOptionInstance.transform.localScale = new Vector3(subCellsRange, subCellsRange);
        mapCellOptionInstance.transform.localPosition = new Vector3(instanceCoordX, instanceCoordY, 0);
        placedOptions++;
        if (placedOptions >= cellDivisor)
        {
            coordX = 0.0f;
            coordY += coordY + subCellsRange;
            placedOptions = 0;
        }
        else
        {
            coordX += coordX + subCellsRange;
        }

        moduleToRenderer.Add(module, mapCellOptionInstance);
        mapCellOptionInstance.OnCreated(module, this);
    }
}

```

Respecto a la creación de la visualización de un módulo, no hay mucho que comentar. En la creación de esta visualización aprovechamos para establecer ciertos parámetros para su manejo, como puede ser coordenadas, si la celda colapsa al módulo o la suscripción a eventos de los que depende para su correcto funcionamiento.

```

public override void OnCreated(Module module, WFCGridRendererMapCell mapCellRenderer)
{
    this.mapCellRenderer = mapCellRenderer;
    Module2D module2D = module as Module2D;

    if (module2D == null) return;
    if (this.frame == null || this.frame.GetComponent<SpriteRenderer>() == null) return;
    if (this.sprite == null || this.sprite.GetComponent<SpriteRenderer>() == null) return;
    if (this.GetComponent<BoxCollider2D>() == null) return;

    this.module2D = module2D;
    this.sprite.GetComponent<SpriteRenderer>().sprite = this.module2D.sprite;
    this.spriteNormalColor = this.sprite.GetComponent<SpriteRenderer>().color;
    this.frameNormalColor = this.frame.GetComponent<SpriteRenderer>().color;
    this.startPosition = this.transform.localPosition;
    this.startScale = this.transform.localScale;
    this.selectedPosition = new Vector3(0.5f, 0.5f, 0);
    this.selectionScale = new Vector3(1, 1, 1);
    this.isAvailable = true;
    if (this.mapCellRenderer.wfcGridRenderer.gameStatusHandler == null) return;
    this.mapCellRenderer.wfcGridRenderer.gameStatusHandler.SubscribeToGameStatusEvent(OnGameStatusChanged);

}

2 referencias
public void OnGameStatusChanged(GameStatus newGameStatus)
{
    this.inputEnable = newGameStatus.Equals(GameStatus.Gameplay) ? true : false;
}

```

### 4.5.3 Comunicación con el algoritmo WFC.

Desde la capa de visualización, la comunicación con el algoritmo WFC se hace desde el script **WFCGridRenderer**. Este manda a ejecutar en segundo plano la operación del algoritmo WFC deseada y mediante el uso de eventos es notificado del progreso del algoritmo. En esa notificación el script actualiza la visualización de la función de onda en base al progreso del algoritmo.

- **Solicitudes de operación**

```
public void RelaxOne() {
    this.canSolveFromScratch = false;
    this.wfc.CollapseMinEntropyMapCell().RunAsCoroutine(this).onProgress += HandleWFCProgress;
}

/// <summary> Metodo que manda al algoritmo WFC a colapsar nuestra funcion de on ...
0 referencias
public void Solve() {
    if (!canSolveFromScratch) return;
    this.wfc.Collapse(true).RunAsCoroutine(this).onComplete += HandleCompletedWFC;
    this.canSolveFromScratch = false;
}

/// <summary> Metodo que manda al algoritmo WFC a colapsar nuestra funcion de on ...
1 referencia
public void SolveFromUser()
{
    this.canSolveFromScratch = false;
    this.auxWFC = this.wfc;
    this.wfc.Collapse(false).RunAsCoroutine(this).onProgress += HandleWFCProgress;
}

/// <summary> Metodo que manda al algoritmo WFC a eliminar un modulo de una celd ...
1 referencia
public void RemoveModule(MapCell mapCell, Module option) {
    this.canSolveFromScratch = false;
    this.wfc.RemoveMapCellOptionFromUser(mapCell, option).RunAsCoroutine(this).onProgress += HandleWFCProgress;
}

/// <summary> Metodo que manda al algoritmo WFC a colapsar una celda a un modulo ...
1 referencia
public void SelectionModule(MapCell mapCell, Module option) {
    this.canSolveFromScratch = false;
    this.wfc.CollapseMapCellToModule(mapCell, option).RunAsCoroutine(this).onProgress += HandleWFCProgress;
}

/// <summary> Metodo que manda al algoritmo WFC a volver el estado inicial de nu ...
0 referencias
public void Reset_WFC() {
    this.wfc.Reset().RunAsCoroutine(this).onProgress += HandleWFCProgress;
}
```

- Manejadores de progreso

```
private void HandleCompletedWFC() {
    foreach (MapCell mapCell in wfc.GetMapGrid().mapCells.RenderizableMapCells())
    {
        foreach (Module invalidModule in mapCell.GetInvalidOptions())
        {
            mapCellToRenderer[mapCell].RemoveOption(invalidModule);
        }

        mapCellToRenderer[mapCell].SelectOption(mapCell.GetValidOptions()[0]);
    }
}
```

```
private void HandleWFCProgress(WFCProgress progress)
{
    switch (progress)
    {
        case WFCMapCellCollapsed wFcMapCellCollapsed:
            mapCellToRenderer[wFcMapCellCollapsed.mapCell].SelectOption(wFcMapCellCollapsed.module);
            break;

        case WFCMapCellModuleRemoved wFcMapCellModuleRemoved:
            mapCellToRenderer[wFcMapCellModuleRemoved.mapCell].RemoveOption(wFcMapCellModuleRemoved.module);
            break;

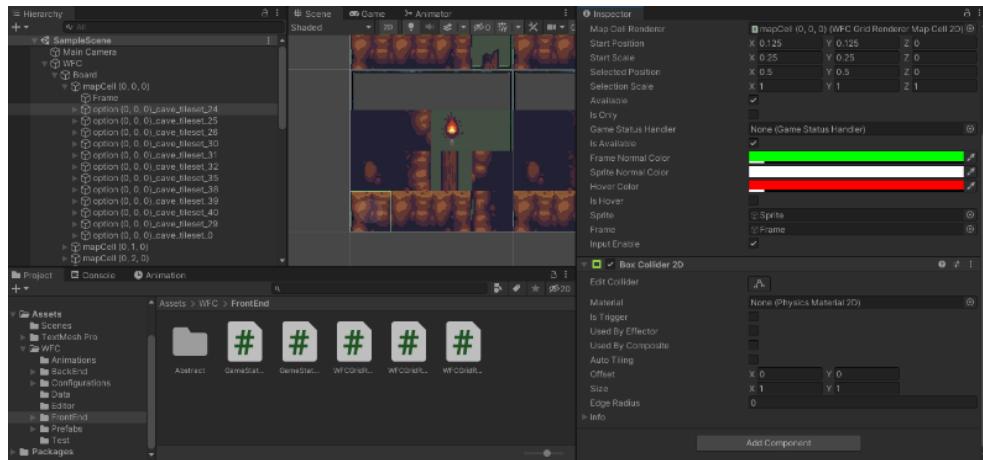
        case WFCUncollapsible wFcUncollapsible:
            List<MapCell> contradictedMapCells = this.wfc.GetMapGrid().mapCells.ContradictedMapCells();
            if (contradictedMapCells == null || contradictedMapCells.Count < 1) return;
            foreach (MapCell contradictedMapCell in contradictedMapCells)
            {
                mapCellToRenderer[contradictedMapCell].ShowContradiction();
            }
            break;

        case WFCReset wFcReset:
            foreach (MapCell mapCell in this.wfc.GetMapGrid().mapCells.UnRenderizableMapCells())
            {
                wFc.CollapseMapCell(mapCell).RunAsRoutine(null);
            }
            InitGameObjects();
            this.canSolveFromScratch = true;
            break;

        case WFCContradiction wFcContradiction:
            if (this.auxWFC == null)
            {
                mapCellToRenderer[wFcContradiction.mapCell].ShowContradiction();
            }
            else
            {
                this.wfc = this.auxWFC;
                this.auxWFC = null;
                this.SolveFromUser();
            }
            break;
    }
}
```

#### 4.5.4 Interacción con los módulos

Todos los módulos renderizados tienen, aparte del script que controla su visualización e interacción, el componente Box Collider 2D.



Este componente es necesario si queremos detectar eventos del ratón sobre el módulo renderizado y actuar respecto a estos eventos. Al tener este componente añadido a un módulo renderizado, simplemente basta con definir una serie de métodos al script que controla el renderizado del módulo. Estos métodos, al igual que los métodos Start y Update, son proporcionados por Unity para todo objeto que tenga un comportamiento en la escena. Dichos métodos son los siguientes:

```
private void OnMouseEnter()
{
    if (!this.inputEnable) return;
    this.isHover = true;
    if (this.frame == null || this.frame.GetComponent<SpriteRenderer>() == null || isOnly) return;
    if (this.sprite == null || this.sprite.GetComponent<SpriteRenderer>() == null || isOnly) return;
    this.frame.GetComponent<SpriteRenderer>().color = this.hoverColor;
    this.sprite.GetComponent<SpriteRenderer>().color = this.hoverColor;
}

@ Mensaje de Unity | 0 referencias
private void OnMouseExit()
{
    if (!this.inputEnable) return;
    this.isHover = false;
    if (this.frame == null || this.frame.GetComponent<SpriteRenderer>() == null || isOnly) return;
    if (this.sprite == null || this.sprite.GetComponent<SpriteRenderer>() == null || isOnly) return;
    this.frame.GetComponent<SpriteRenderer>().color = this.frameNormalColor;
    this.sprite.GetComponent<SpriteRenderer>().color = this.spriteNormalColor;
}
```

```

private void OnMouseOver()
{
    if (isOnly) return;
    if (!this.inputEnable) return;

    if (this.GetComponent<BoxCollider2D>() == null || !this.GetComponent<BoxCollider2D>().enabled) return;

    if (Input.GetMouseButtonUp(1)) //Click Derecho
    {
        this.mapCellRenderer.wfcGridRenderer.RemoveModule(this.mapCellRenderer.mapCell, this.module2D);
    }
    else if (Input.GetMouseButtonUp(0)) // Click Izquierdo
    {
        this.mapCellRenderer.wfcGridRenderer.SelectionModule(this.mapCellRenderer.mapCell, this.module2D);
    }
}

```

Como se puede ver en la captura anterior, se han definido dos acciones para que el usuario interactúe con los módulos de una celda. Dichas acciones son:

- **Selección del módulo como colapso de la celda a la que pertenece**

Esta acción, que está definida en el click izquierdo del ratón del usuario, provocará la llamada al algoritmo WFC con el fin de colapsar la celda al módulo seleccionado. Cuando el algoritmo haya colapsado la celda al módulo y lo notifique, se refrescará el renderizado del módulo en cuestión con esta porción de código perteneciente al método Update que gestiona su visualización.

```

this.transform.localPosition = Vector3.Lerp(this.transform.localPosition, this.selectedPosition, Time.deltaTime);
this.transform.localScale = Vector3.Lerp(this.transform.localScale, this.selectionScale, Time.deltaTime);
this.sprite.GetComponent<SpriteRenderer>().color = this.spriteNormalColor;

if (this.frame != null) this.frame.SetActive(false);
this.mapCellRenderer.DisableFrame();

```

Simplemente consiste en adecuar su posición y escala mediante una interpolación para que ocupe la celda a la que pertenece de manera completa.

- **Eliminación del módulo como colapso de la celda a la que pertenece**

Esta acción, que está definida en el click derecho del ratón del usuario, provocará la llamada al algoritmo WFC con el fin de eliminar el módulo como opción de colapso para la celda a la que pertenece. Cuando el algoritmo haya realizado la operación correspondiente y lo notifique, se refrescará el renderizado del módulo en cuestión con esta porción de código perteneciente al método Update que gestiona su visualización.

```

        if (!available && isAvailable && !isOnly) {
            this.BlowUp();
            isAvailable = available;
        }

        if (transform.position.y < 0) {
            Destroy(this.gameObject);
        }
    }

public void BlowUp() {
    this.isHover = false;

    if (this.GetComponent<BoxCollider2D>() == null || !this.GetComponent<BoxCollider2D>().enabled) return;
    if (this.frame != null) this.frame.SetActive(false);
    this.GetComponent<BoxCollider2D>().enabled = false;
    Rigidbody2D body = this.GetComponent<Rigidbody2D>() == null ? gameObject.AddComponent<Rigidbody2D>() : this.GetComponent<Rigidbody2D>();
    body.isKinematic = false;
    body.AddForce(
        new Vector2(UnityEngine.Random.Range(1, 3) * (UnityEngine.Random.value > .5f ? 1 : -1),
        UnityEngine.Random.Range(4, 5)), ForceMode2D.Impulse);
    body.AddTorque(UnityEngine.Random.Range(1, 4) * (UnityEngine.Random.value > .5f ? 1 : -1),
    ForceMode2D.Impulse);
}

```

En la eliminación del módulo simplemente se hace una pequeña animación mediante el sistema de físicas de Unity. Esta animación consiste en añadir al módulo en cuestión el componente Rigidbody y aplicarle una fuerza y torque de manera aleatoria. Después, debido a la fuerza de la gravedad de Unity, el objeto acabará cayendo al vacío y se acabará eliminando cuando su posición en el eje Y sea menor a cero.

Hay otro método proporcionado por Unity para los objetos de la escena con comportamiento que es llamado cuando estos son destruidos. Se muestra a continuación:

```

void OnDestroy()
{
    if (this.mapCellRenderer.wfcGridRenderer.gameStatusHandler == null) return;
    this.mapCellRenderer.wfcGridRenderer.gameStatusHandler.UnSubscribeToGameStatusEvent(OnGameStatusChanged);
}

```

Este método ha sido aprovechado para que el objeto pueda desuscribirse del evento que nos informa del estado de la escena y que controla si puede haber interacción del usuario o no.

Es necesario aclarar que el usuario solo manda a ejecutar una operación del algoritmo WFC y que es el progreso de algoritmo WFC, junto con el manejador del mismo, el que se hace cargo de refrescar la visualización de la función de onda en concordancia con el algoritmo. A continuación se muestra el flujo para visualizar el colapso de una celda a un módulo.

```
private void HandleWFCProgress(WFCProgress progress) {
    switch (progress) {
        case WFCMapCellCollapsed wFCMapCellCollapsed:
            mapCellToRenderer[wFCMapCellCollapsed.mapCell].SelectOption(wFCMapCellCollapsed.module);
            break;
    }
}

public void SelectOption(Module selectModule) {
    if (!moduleToRenderer.ContainsKey(selectModule)) return;
    moduleToRenderer[selectModule].Select();
}

public void Select() {
    isOnly = true;
}

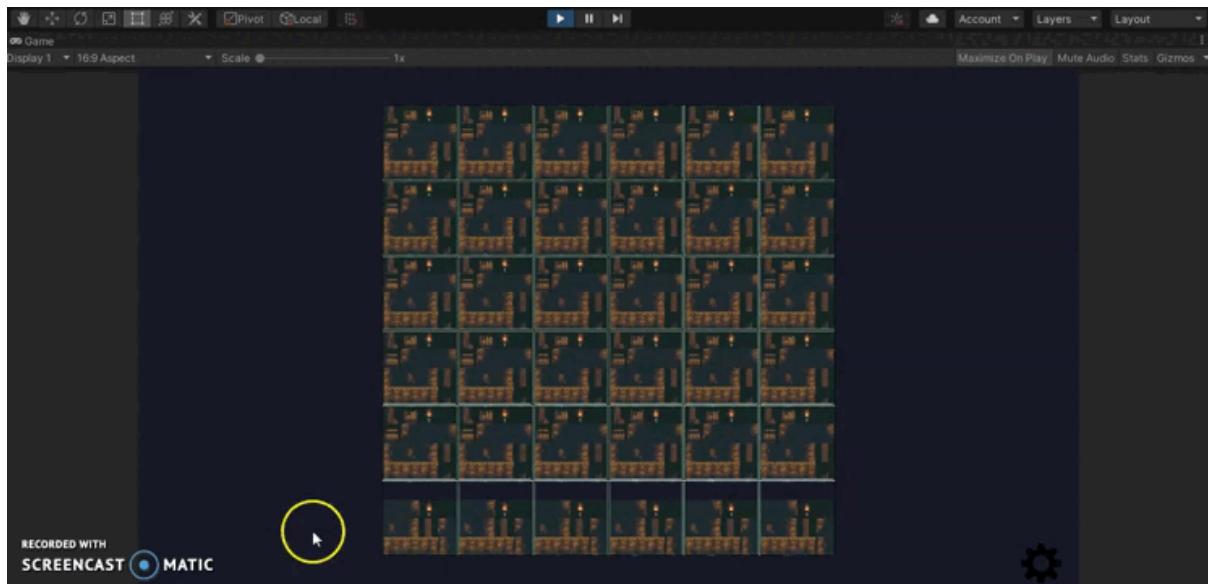
if (!isOnly) return;

this.transform.localPosition = Vector3.Lerp(this.transform.localPosition, this.selectedPosition, Time.deltaTime);
this.transform.localScale = Vector3.Lerp(this.transform.localScale, this.selectionScale, Time.deltaTime);
this.sprite.GetComponent<SpriteRenderer>().color = this.spriteNormalColor;

if (this.frame != null) this.frame.SetActive(false);
this.mapCellRenderer.DisableFrame();
```

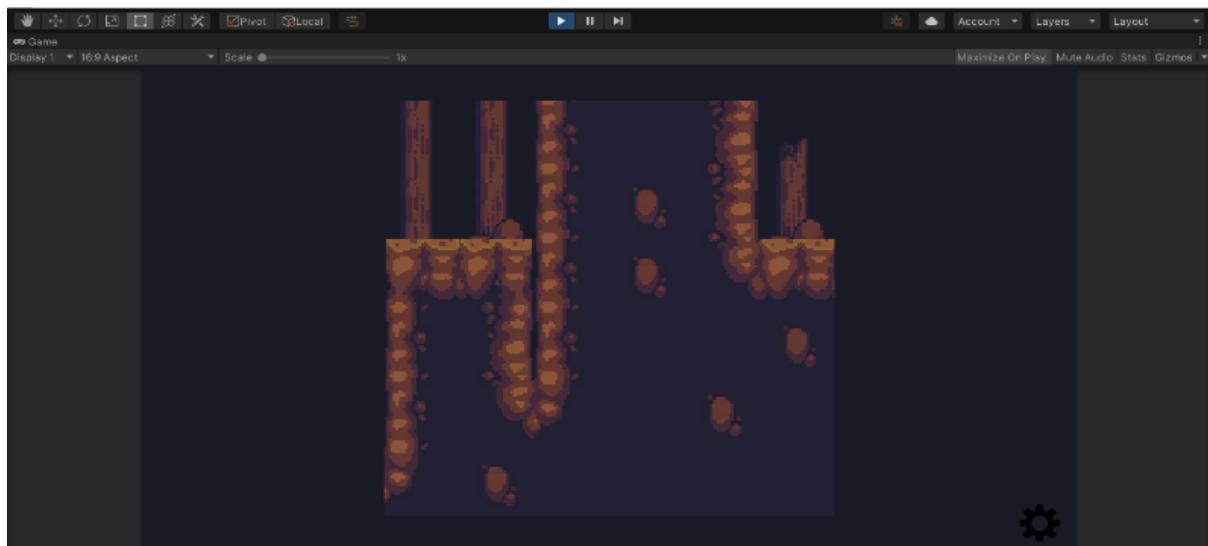
## 5. Resultado

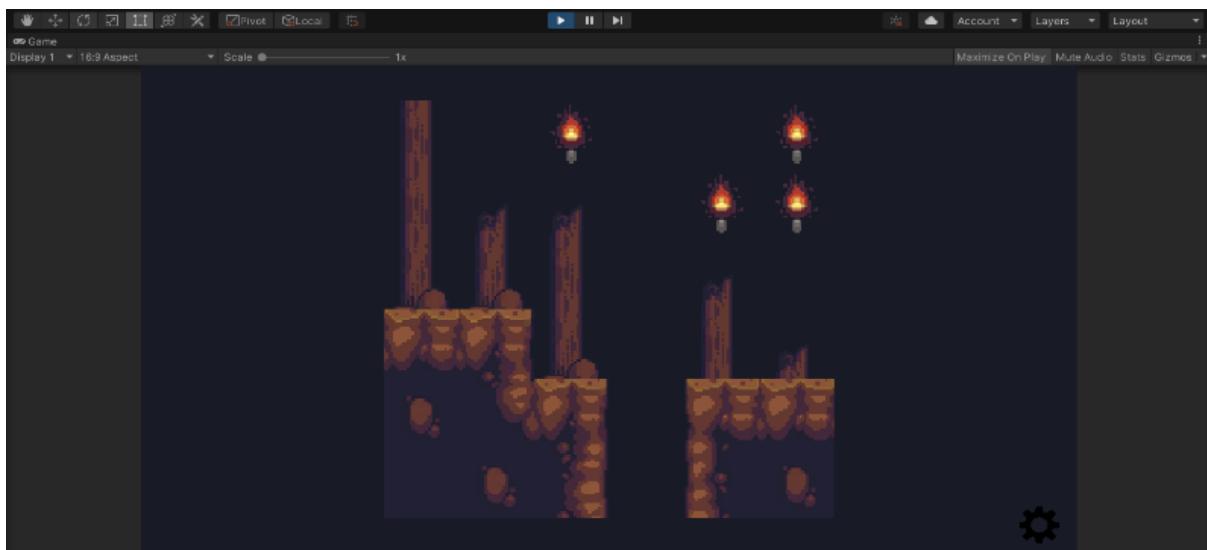
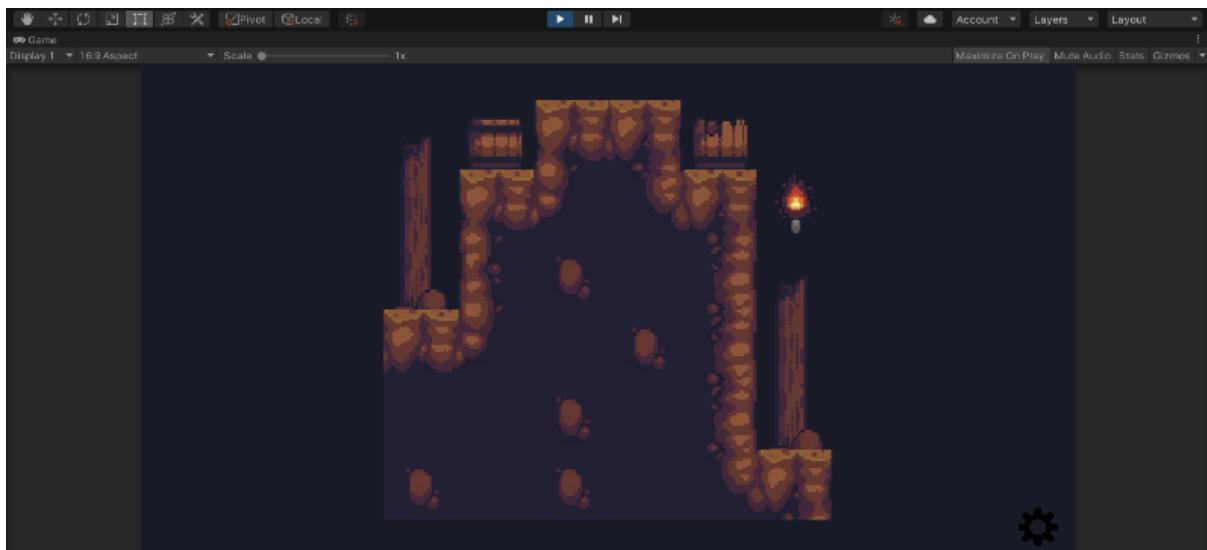
Con la implementación de todo lo comentado previamente obtenemos el siguiente resultado final.



Los controles son simples. Al hacer click izquierdo sobre un módulo hará que la celda a la que pertenece colapse a él, mientras que si hacemos click derecho lo estamos descartando como opción de colapso para la celda a la que pertenece. Las demás operaciones de interacción con el algoritmo WFC se encuentran disponibles desde la UI.

A continuación se muestran ejemplos del colapso de nuestra función de onda.





Cabe destacar que lo mostrado en las imágenes anteriores son simplemente el colapso de funciones de ondas para una configuración de adyacencia de imágenes que han sido seleccionadas para ilustrar el funcionamiento del algoritmo WFC en un ejemplo. Esto es extremadamente positivo, pues como se ha mostrado anteriormente, nada nos impide crear nuevas configuraciones de adyacencia de imágenes con otro conjunto de imágenes distinto.

También debe quedar claro que se ha dejado preparado un entorno de abstracción de tal manera que la inclusión de nuevos módulos con distintas restricciones sea relativamente sencilla.

Llegados a este punto, donde se ha explorado el algoritmo WFC, podemos decir que el algoritmo WFC no es más que un algoritmo satisfactor de restricciones, siendo esto útil no solo en el ámbito de generación de entornos gráficos. Lleva a que, por ejemplo, podamos resolver diferentes problemas, como puede ser el Sudoku, el cual fue utilizado para introducir el algoritmo WFC en este mismo documento o el famoso problema de las ocho reinas [57].

## 6. Conclusiones

Llegado a esta parte del documento, se va a recoger y realizar una valoración final del proyecto para poder, tomando perspectiva, conocer hasta qué punto se han conseguido los objetivos planteados inicialmente y poder plantear, así mismo, posibles líneas futuras de desarrollo que quedan abiertas, así como la proyección del proyecto realizado. Además, también se expondrá la opinión personal del proyecto en las diferentes fases de las que se ha compuesto, junto con una valoración del algoritmo WFC para la generación procedural de entornos.

Para ello, antes de nada, me gustaría hablar del entorno de trabajo en el que se ha implementado el algoritmo WFC.

### **Unity**

Entre las conclusiones que puedo sacar a través de la experiencia de haber utilizado este motor gráfico me gustaría destacar las siguientes:

- Es un motor gráfico que cuenta con una interfaz sencilla y amigable con el usuario.
- Es una herramienta muy versátil, permitiendo proyectos de múltiples ámbitos (2D, 3D, VR, ...).
- La orientación en componentes para los objetos que forman parte de la escena la considero muy intuitiva.
- Es un motor gráfico con muchas fuentes de aprendizaje y una comunidad enorme que no duda en ayudar.

Pese a que el motor haya sido usado para este proyecto y se haya realizado de manera satisfactoria obteniendo una gran experiencia, soy consciente de que puede que este motor no sea recomendable para proyectos de mayor presupuesto o muy ambiciosos, debido a que otros motores, como Unreal Engine, facilitan más las cosas en aspectos como puede ser la iluminación, convirtiéndose en la tendencia actual en la industria de los videojuegos para proyectos profesionales.

Aun así, es un motor que recomiendo para proyectos humildes puesto que considero que tiene una curva de aprendizaje bastante rápida y poco frustrante.

## C#

Para mí C# ha sido la gran sorpresa de este proyecto. Esto se debe en gran parte a que es un lenguaje de alto nivel que cuenta con aspectos particulares muy bien resueltos que han facilitado mucho la implementación del algoritmo WFC, como pueden ser el uso de eventos, la palabra reservada *yield* o el tratamiento de colecciones mediante *streams*.

Puedo decir, además, que no he echado en falta nada de otros lenguajes en este, pareciéndome personalmente mejor que otros lenguajes similares como puede ser Java en aspectos como el uso de eventos.

## Implementación

Respecto a la implementación del algoritmo WFC, me gustaría en primer lugar decir que ha sido una experiencia realmente satisfactoria gracias al aprendizaje que he podido obtener en el uso de Unity en aspectos que eran desconocidos para mí, como son los *Scriptable Objects* o los *Custom Editors*, que causaron problemas durante el desarrollo, principalmente por la serialización. Por otro lado, debido a las necesidades de la implementación, puesto que se quería interactuar con el algoritmo WFC de forma gráfica, también he tenido que profundizar en las particulares del lenguaje C#, como son los eventos y el uso de *yield*, adquiriendo también conocimientos en ese aspecto.

También puedo decir que la complejidad del proyecto me ha servido para poder obtener mayor capacidad de abstracción, dividiendo el grueso del proyecto en ramas bien diferenciadas y modularizando en cada una de ellas.

## Algoritmo WFC

El algoritmo WFC es un algoritmo no muy complejo de codificar y nos permite resolver un conjunto de restricciones en un determinado espacio de manera eficaz. Sin embargo, tiene un gran coste computacional debido principalmente al proceso de propagar restricciones a lo largo del espacio. Este factor es limitante para trabajar con espacios grandes, donde la probabilidad de encontrar una contradicción se incrementa significativamente. Esto provoca, a su vez, que el tiempo de encontrar un colapso para una función de onda crezca. A pesar de ello, a su favor debemos decir que el algoritmo WFC nos proporciona un entorno flexible para trabajar con infinidad de restricciones.

Como conclusión, debo decir que este TFG me ha dado la oportunidad de realizar un camino de auto aprendizaje en Unity y aspectos concretos de C# desconocidos para mí previamente.

Los objetivos que se propusieron para el proyecto han sido realizados de manera satisfactoria, sintiéndome orgulloso del trabajo realizado.

# Trabajo futuro

En la finalización de este proyecto se ha dejado una buena base sobre la que ampliar configuraciones en la que ejecutar el algoritmo WFC. Teniendo en cuenta esto, para futuro se propone lo siguiente:

- Generar nuevos tipos de configuraciones con el fin de ejecutar el algoritmo WFC para contemplar otros tipos de restricciones en entornos 2D.

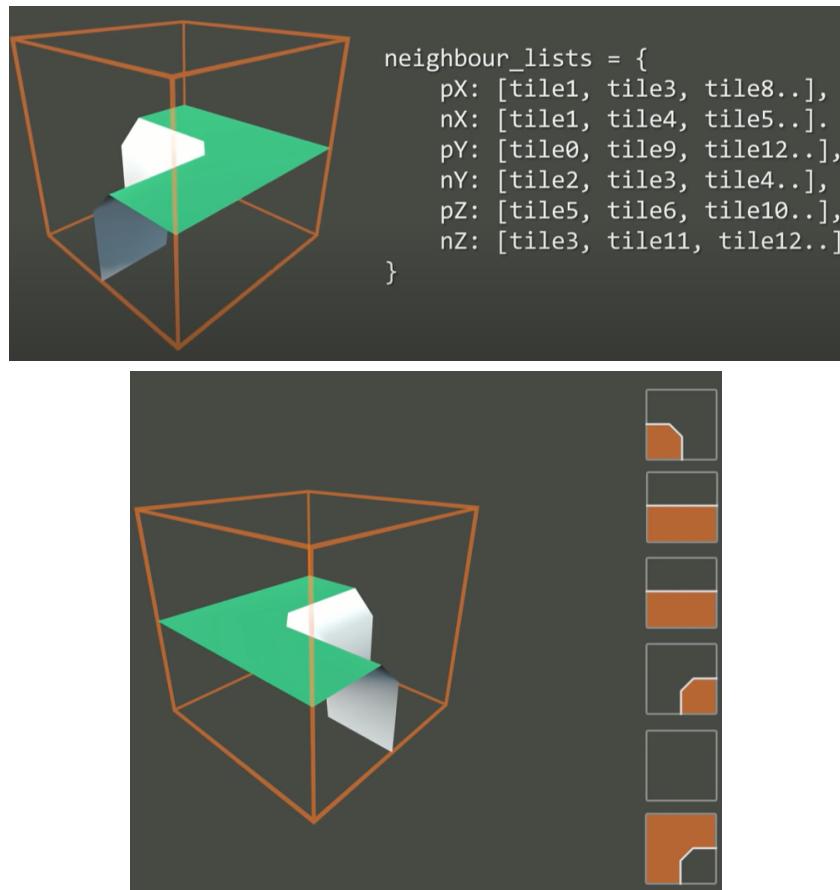
Esto es relativamente sencillo debido al nivel de abstracción con el que trabaja el algoritmo WFC, y que ha sido mostrado anteriormente a lo largo del documento.

Para estas nuevas configuraciones solo habrá que seguir los mismos pasos realizados para la generación de la configuración expuesta en este documento. Esto se traduce en los siguientes pasos:

1. Generar un script que herede de la clase abstracta *ModuleConstraint* e implemente la regla que se desee aplicar.
  2. Generar un script que herede de la clase abstracta *Module* y defina un nuevo tipo de módulo que se ajuste a la regla creada en el paso anterior.
  3. Crear un nuevo *Scriptable Object* que herede de la clase abstracta asociada a las configuraciones para el algoritmo WFC y que genere la configuración deseada para el tipo de módulo y restricción generados en los pasos anteriores.
  4. Realizar un *Custom Editor* para el anterior si se considera necesario.
- 
- Implementar la generación de entornos 3D.

Esta propuesta es más ambiciosa que la anterior, ya que implica una nueva línea de desarrollo para trabajar con modelos 3D. Se puede reutilizar la filosofía con la que se han trabajado los entornos en 2D con la salvedad, obviamente, de que al trabajar en entornos 3D hay más factores a tener en cuenta, como son el mayor número de vecinos, la rotación de los modelos 3D, las aristas de cada cara del modelo, simetría, etc.

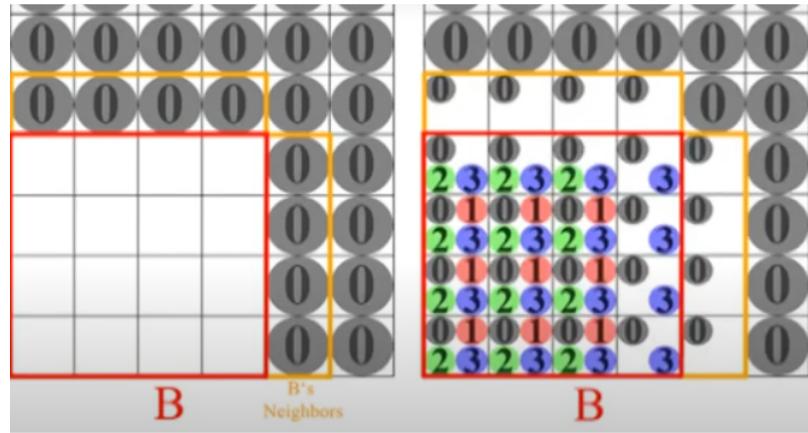
Una ilustración de lo comentado se puede ver en las siguientes imágenes.



En las imágenes anteriores podemos ver cómo los modelos 3D estarían contenidos dentro de cajas, siendo estas cajas los módulos con los que trabaja el algoritmo WFC. Podemos observar también un aspecto muy importante más allá del incremento del número de vecinos que va a tener ahora una celda de nuestro grid. La simetría y la rotación del modelo 3D juegan un aspecto clave, pues el modelo presentado en las imágenes podría ser adyacente a sí mismo bajo una determinada rotación en alguno de sus lados.

Con lo comentado anteriormente queda establecida la base sobre la generación procedural de entornos 3D.

Además, como trabajo futuro más importante, queda pendiente la mejora de rendimiento del algoritmo y reducir significativamente las contradicciones a medida que aumenta la complejidad del problema. Entre las posibles soluciones que se pueden explorar para mejorar este aspecto se encuentra la de seguir una estrategia de "divide y vencerás" con el fin de realizar propagaciones bajo un límite de distancia como si estuviésemos trabajando con unas dimensiones inferiores a las que realmente tenemos. La visualización de esta idea la podemos ver reflejada en la siguiente imagen.



Esta estrategia es la utilizada por Model Synthesis algorithm, consiguiendo una mejora significativa para entornos de trabajo grandes.

Pese al trabajo realizado, como se puede ver, siguen existiendo distintas líneas de ampliación para el proyecto que espero poder llegar a realizar en algún momento, siendo cada una de ellas relevantes e interesantes.

En cualquier caso, espero que mi experiencia con la implementación del algoritmo WFC haya servido de ayuda o inspiración a todos los lectores de este documento y animo a cualquier interesado a realizar su propia implementación del algoritmo WFC.

# Referencias

- [1] Donal, M. [Superpositions, Sudoku, the Wave Function Collapse algorithm](https://www.youtube.com/watch?v=2SuvO4Gi7uY). (31 de julio de 2020). URL: <https://www.youtube.com/watch?v=2SuvO4Gi7uY>
- [2] Kleineberg, M. [Infinite procedurally generated city with the Wave Function Collapse algorithm](https://marian42.de/article/wfc/). (6 de enero de 2019). URL: <https://marian42.de/article/wfc/>
- [3] Gumin, M. [Wave Function Collapse program for generating bitmaps](https://github.com/mxgmn/WaveFunctionCollapse). URL: <https://github.com/mxgmn/WaveFunctionCollapse>
- [4] Heaton, R. [The Wave Function Collapse algorithm explained very clearly](https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/). (17 de diciembre de 2018). URL: <https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/>
- [5] Unity Technologies. [Order of execution for event functions in Unity Scripting](https://docs.unity3d.com/Manual/ExecutionOrder.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/ExecutionOrder.html>
- [6] Unity Technologies. [Unity Website](https://unity.com/es). URL: <https://unity.com/es>
- [7] Unity Technologies. [Learn Unity](https://learn.unity.com/). URL: <https://learn.unity.com/>
- [8] Unity Technologies. [Unity Documentation](https://docs.unity3d.com/Manual/index.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/index.html>
- [9] Unity Technologies. [Introduction to components](https://docs.unity3d.com/Manual/Components.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/Components.html>
- [10] Unity Technologies. [Prefabs in Unity](https://docs.unity3d.com/Manual/Prefabs.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/Prefabs.html>
- [11] Unity Technologies. [Start method in Unity Scripting](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html>
- [12] Unity Technologies. [Update method in Unity Scripting](https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html>
- [13] Unity Technologies. [Transform component in Unity](https://docs.unity3d.com/Manual/class-Transform.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/class-Transform.html>
- [14] Unity Technologies. [Sprite Renderer component in Unity](https://docs.unity3d.com/Manual/class-SpriteRenderer.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/class-SpriteRenderer.html>
- [15] Unity Technologies. [Box Collider 2D component in Unity](https://docs.unity3d.com/Manual/class-BoxCollider2D.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/class-BoxCollider2D.html>
- [16] Unity Technologies. [Rigidbody 2D component in Unity](https://docs.unity3d.com/Manual/class-Rigidbody2D.html). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/class-Rigidbody2D.html>
- [17] Brackeys. [Scriptable Objects in Unity](https://www.youtube.com/watch?v=aPXvoWVabPY). (3 de enero de 2018). URL: <https://www.youtube.com/watch?v=aPXvoWVabPY>

- [18] Unity Technologies. [Canvas in Unity](#). URL: <https://docs.unity3d.com/es/2019.4/Manual/UICanvas.html>
- [19] Press Start. [Understanding Orthographic Size in Unity](#). (14 de junio de 2018). URL: <https://www.youtube.com/watch?v=3xXlnSetHPM>
- [20] Brackeys. [Pause Menu in Unity](#). (20 de diciembre de 2017). URL: <https://www.youtube.com/watch?v=3xXlnSetHPM>
- [21] James Makes Games. [Pause in Unity without Timescale](#). (14 de mayo de 2021). URL: <https://www.youtube.com/watch?v=KPaEnLpu57s>
- [22] Unity Technologies. [Serialización de scripts en Unity](#). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/script-Serialization.html>
- [23] Unity Technologies. [Custom Editors in Unity](#). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/editor-CustomEditors.html>
- [24] Brackeys. [How to make a Custom Inspector in Unity](#). (9 de julio de 2017). URL: [https://www.youtube.com/watch?v=RInUu1\\_8aGw](https://www.youtube.com/watch?v=RInUu1_8aGw)
- [25] Unity Technologies. [Coroutines in Unity](#). (24 de junio de 2022). URL: <https://docs.unity3d.com/Manual/Coroutines.html>
- [26] Unity Technologies. [C# Coroutines in Unity! - Intermediate Scripting Tutorial](#). (2 de agosto de 2019). URL: <https://www.youtube.com/watch?v=5L9ksCs6MbE>
- [27] Vázquez Moreno, Juan Diego. [Introducción a Unity para videojuegos 2D](#). URL: <https://www.domestika.org/es/courses/716-introduccion-a-unity-para-videojuegos-2d>
- [28] Microsoft. [Documentación C#](#). URL: <https://docs.microsoft.com/es-ES/dotnet/csharp/>
- [29] Microsoft. [yield \(Referencia de C#\)](#). (23 de junio de 2022). URL: <https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/yield>
- [30] Microsoft. [Delegados \(Guía de programación de C#\)](#). (23 de junio de 2022). URL: <https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/delegates/>
- [31] Microsoft. [event \(Referencia de C#\)](#). (23 de junio de 2022). URL: <https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/event>
- [32] Microsoft. [IEnumerable Interfaz](#). URL: <https://docs.microsoft.com/es-es/dotnet/api/system.collections.ienumerable?view=net-6.0>
- [33] Microsoft. [System.Linq Espacio de nombres](#). URL: <https://docs.microsoft.com/es-es/dotnet/api/system.linq?view=net-6.0>
- [34] TutorialsTeacher. [Uso de System.Linq](#). URL: <https://www.tutorialsteacher.com/linq/sample-linq-queries>
- [35] Microsoft. [Métodos de extensión \(Guía de programación de C#\)](#). (23 de junio de 2022). URL: <https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>

- [36] Microsoft. [Operadores y expresiones de acceso a miembros \(referencia de C#\)](#). (23 de junio de 2022). URL:  
<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/member-access-operators>
- [37] Microsoft. [var \(Referencia de C#\)](#). (23 de junio de 2022). URL:  
<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/var>
- [38] Microsoft. [Operadores de prueba de tipos y expresión de conversión \(referencia de C#\)](#). (23 de junio de 2022). URL:  
<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/type-testing-and-cast>
- [39] Microsoft. [virtual \(Referencia de C#\)](#). (23 de junio de 2022). URL:  
<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/virtual>
- [40] Microsoft. [override \(Referencia de C#\)](#). (23 de junio de 2022). URL:  
<https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/override>
- [41] Wikipedia. [Procedural generation](#). URL:  
[https://en.wikipedia.org/wiki/Procedural\\_generation](https://en.wikipedia.org/wiki/Procedural_generation)
- [42] Touti, R. [Perlin noise algorithm](#). URL:  
<https://rtouti.github.io/graphics/perlin-noise-algorithm#:~:text=Perlin%20noise%20is%20a%20popular,number%20of%20inputs%20it%20gets>
- [43] Galindo, S. [Unity - Procedural Terrain Object with Perlin Noise](#). (30 de octubre de 2018). URL: <https://www.youtube.com/watch?v=1FH0uspD2kc>
- [44] Sunny Valley Studio. [Procedural map using genetic algorithm](#). (3 de abril de 2020). URL: [https://www.youtube.com/watch?v=FgXf7yKwPH4&ab\\_channel=SunnyValleyStudio](https://www.youtube.com/watch?v=FgXf7yKwPH4&ab_channel=SunnyValleyStudio)
- [45] Dykeman, I. [Procedural worlds simple tiles](#). (12 de octubre de 2017). URL:  
<https://ijdykeman.github.io/ml/2017/10/12/wang-tile-procedural-generation.html>
- [46] Merrell, Paul. [Model synthesis](#). URL: <https://paulmerrell.org/model-synthesis/>
- [47] Merrell, Paul. [Example-Based Model Synthesis](#). URL:  
[https://paulmerrell.org//model\\_synthesis.pdf](https://paulmerrell.org//model_synthesis.pdf)
- [48] Wikipedia. [Sistemas-L](#). URL: <https://es.wikipedia.org/wiki/Sistema-L>
- [49] Pete P. [L-Systems Unity Tutorial](#). (5 de diciembre de 2018). URL:  
<https://www.youtube.com/watch?v=tUbTGWl-qus>
- [50] Wikipedia. [Autómatas celulares](#). URL:  
[https://es.wikipedia.org/wiki/Aut%C3%B3mata\\_celular#:~:text=Un%20aut%C3%B3mata%20celular%20\(A.C.\)%20es,interact%C3%BAen%20localmente%20unos%20con%20otros](https://es.wikipedia.org/wiki/Aut%C3%B3mata_celular#:~:text=Un%20aut%C3%B3mata%20celular%20(A.C.)%20es,interact%C3%BAen%20localmente%20unos%20con%20otros)
- [51] Academia de videojuegos. [Generación automática de niveles 2D en Unity - Autómata Celular Moore](#). (22 de junio de 2018). URL:  
<https://www.youtube.com/watch?v=30vnpyqAD5U>
- [52] Wikipedia. [Texture synthesis](#). URL: [https://en.wikipedia.org/wiki/Texture\\_synthesis](https://en.wikipedia.org/wiki/Texture_synthesis)

- [53] Wikipedia. [Azulejos Wang](https://es.wikipedia.org/wiki/Azulejos_Wang). URL: [https://es.wikipedia.org/wiki/Azulejos\\_Wang](https://es.wikipedia.org/wiki/Azulejos_Wang)
- [54] Gumin, M. [MarkovJunior](https://github.com/mxgmn/MarkovJunior). URL: <https://github.com/mxgmn/MarkovJunior>
- [55] Benard, S. [LDtk](https://ldtk.io/). URL: <https://ldtk.io/>
- [56] Study.com. [Constraint Satisfaction Problem](https://study.com/academy/lesson/constraint-satisfaction-problems-definition-examples.html). URL:  
<https://study.com/academy/lesson/constraint-satisfaction-problems-definition-examples.html>
- [57] Wikipedia. [El problema de las ocho reinas](https://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas#:~:text=El%20problema%20de%20las%20ocho,misma%20fila%2C%20columna%20o%20diagonal.). URL:  
[https://es.wikipedia.org/wiki/Problema\\_de\\_las\\_ocho\\_reinas#:~:text=El%20problema%20de%20las%20ocho,misma%20fila%2C%20columna%20o%20diagonal.](https://es.wikipedia.org/wiki/Problema_de_las_ocho_reinas#:~:text=El%20problema%20de%20las%20ocho,misma%20fila%2C%20columna%20o%20diagonal.)
- [58] Accenture. [Estudio de Accenture sobre el mercado mundial de videojuegos](https://www.accenture.com/us-en/insights/software-platforms/gaming-the-next-super-platform?c=acn_glb_thenewgamingexpbusinesswire_12160747&n=mrl_0421). (27 de abril de 2021). URL:  
[https://www.accenture.com/us-en/insights/software-platforms/gaming-the-next-super-platform?c=acn\\_glb\\_thenewgamingexpbusinesswire\\_12160747&n=mrl\\_0421](https://www.accenture.com/us-en/insights/software-platforms/gaming-the-next-super-platform?c=acn_glb_thenewgamingexpbusinesswire_12160747&n=mrl_0421)