



Universidad de Castilla-La Mancha
Escuela Superior de Ingeniería Informática

Trabajo Fin de Grado
Grado en Ingeniería Informática
Tecnología Específica de
Computación

Análisis comparativo entre la POO y la POD enfocado al desarrollo de videojuegos con Unity y Unity DOTS

Pablo Lario Gómez

Febrero, 2024



Trabajo Fin de Grado
Grado en Ingeniería Informática
Tecnología Específica de
Computación

Análisis comparativo entre la POO y la POD
enfocado al desarrollo de videojuegos con
Unity y Unity DOTS

Autor: Pablo Lario Gómez
Tutor: José Pascual Molina Massó
Co-Tutor: Arturo Simón García Jiménez

Febrero, 2024

*Dedicado a mi familia y a todos
aquellos que me animan
diariamente a dar lo mejor de mí
en cada proyecto que decido
emprender.*

Declaración de Autoría

Yo, PABLO LARIO GÓMEZ con DNI 49427234G, declaro que soy el único autor del trabajo fin de grado titulado *“Análisis comparativo entre la POO y la POD enfocado al desarrollo de videojuegos con Unity y Unity DOTS”* y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual y que todo el material no original contenido en dicho trabajo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 12 de Febrero de 2024

Fdo: Pablo Lario Gómez

Resumen

Durante los últimos años, la programación orientada a datos ha ido ganándole terreno a la popular y altamente utilizada programación orientada a objetos. Son muchos los programadores que han ido defendiendo las virtudes que este paradigma de programación llegando, en el caso de una empresa como Unity, a incorporar a su motor una serie de librerías para permitir a sus usuarios utilizar la programación orientada a datos para el desarrollo de videojuegos y aplicaciones de Realidad/Virtual Aumentada.

Este proyecto tiene la intención de discernir si la programación orientada a datos es capaz realmente de cumplir con lo prometido y ofrecer mejoras sustanciales a nivel de rendimiento frente a la programación orientada a objetos, la cual ha sido predominante dentro de la industria del desarrollo de videojuegos y, más en general, dentro del mundo de la informática. El objetivo, por tanto, es analizar diversos prototipos que, bajo las mismas condiciones, hayan sido desarrollados con ambos paradigmas, y enfrentar los resultados para poder discernir si existe realmente esa mejora de rendimiento.

Es por todo esto que a lo largo de este trabajo se presentarán dos prototipos distintos, cada uno de ellos con una serie de condiciones concretas, que después de su diseño e implementación han sido ejecutados para obtener datos referentes a su rendimiento, y que posteriormente dichos datos han sido analizados y comparados, con el objetivo de demostrar cuál de los dos paradigmas de programación evaluados ofrece una mejor alternativa a los usuarios del motor.

Agradecimientos

Quiero aquí mencionar sobre todo a mi madre, a mi padre y a mi hermana, los cuáles han sido un apoyo fundamental estos últimos años, así como a mis dos tutores, que han decidido embarcarse conmigo en esta aventura y me han ayudado en todo lo que he necesitado para la elaboración de este trabajo.

Índice general

Capítulo 1	Introducción	1
1.1	Introducción	1
1.2	Motivación	2
1.3	Objetivos	3
Capítulo 2	Estado del Arte	5
2.1	Programación orientada a objetos en Unity	5
2.1.1	¿Qué es la POO?	5
2.1.2	Partes de la POO	6
2.1.3	¿Cómo funciona la POO en Unity?	7
2.1.4	Problemática de la POO en Unity	9
2.2	Programación orientada a datos en Unity	10
2.2.1	¿Qué es la POD?	10
2.2.2	¿Qué es Unity DOTS y ECS?	11
Capítulo 3	Metodología	27
3.1	Qué es una metodología	27
3.2	Metodología aplicada	28
3.2.1	Estudio	29
3.2.2	Prototipo	29
3.2.3	Análisis	29
3.2.4	Evaluación	30
Capítulo 4	Diseño	31
4.1	Variables a analizar	31
4.1.1	FPS	31

4.1.2	Tiempos de creación y destrucción	32
4.2	Prototipos que desarrollar	32
4.2.1	Sobrecalentamiento inducido	33
4.2.2	Configuraciones de los diferentes prototipos	34
4.2.3	Los dos prototipos	35
4.2.4	Hardware utilizado para las pruebas	39
Capítulo 5	Implementación	41
5.1	Implementación del prototipo <i>Cubos</i>	41
5.1.1	Aspectos comunes a todas las configuraciones	41
5.1.2	Configuraciones de la POO	44
5.1.3	Configuraciones de la POD	51
5.2	Implementación del prototipo <i>Coruscant</i>	60
5.2.1	Configuración MONO	61
5.2.2	Configuración DOTS	63
Capítulo 6	Evaluación	65
6.1	Evaluación del prototipo <i>Cubos</i>	65
6.1.1	MSS vs DSS	66
6.1.2	MSC vs DSC	68
6.1.3	MCS vs DCS	69
6.1.4	MCC vs DCC	70
6.1.5	Otras comparativas interesantes	71
6.2	Evaluación del prototipo de <i>Coruscant</i>	74
6.2.1	Comparativa de los FPS de ambas configuraciones	75
6.2.2	Comparativa de los tiempos de creación de edificios	76
6.2.3	Comparativa de los tiempos de destrucción de edificios	79
6.2.4	Comparativa de los tiempos de creación de naves	80
6.2.5	Comparativa de los tiempos de destrucción de naves	81
Capítulo 7	Conclusiones y trabajo futuro	83
7.1	Conclusiones	83
7.2	Competencias adquiridas y específicas de computación	84
7.3	Trabajo futuro	85
Bibliografía		87

Índice de figuras

Figura 2.1 Captura de pantalla de un <i>GameObject</i> en Unity.....	8
Figura 2.2 Código en C# del componente Generator	9
Figura 2.3 Código en C# de un componente en Unity	13
Figura 2.4 Código en C# de un objeto y su correspondiente clase Baker	15
Figura 2.5 Captura de pantalla del vídeo de Brian Will sobre entidades, arquetipos y chunks [18]	16
Figura 2.6 Captura de pantalla del video de Brian Will sobre entidades, arquetipos y chunks [18]	17
Figura 2.7 Código en C# de la estructura de un sistema	18
Figura 2.8 Captura de la pestaña de <i>Systems</i> de una aplicación en Unity DOTS	19
Figura 2.9 Código en C# de un sistema de creación de edificios	20
Figura 2.10 Captura de pantalla del vídeo de Brian Will de un sistema que rota entidades de forma constante [22]	21
Figura 2.11 Captura de pantalla del código de un aspecto del vídeo de Brian Will [22]	22
Figura 2.12 Captura de pantalla del código de la consulta utilizando aspectos del vídeo de Brian Will [22]	22
Figura 2.13 Captura de pantalla de la definición de un job en C#	23
Figura 2.14 Captura de pantalla de la creación de un job en C#	24
Figura 2.15 Código C# de una excepción de seguridad al programar dos jobs.....	25
Figura 2.16 Código C# de una dependencia entre dos jobs	25
Figura 3.1 Esquema de las fases seguidas durante el desarrollo del presente trabajo	28
Figura 4.1 Código de sobrecalentamiento	33
Figura 4.2 Captura de pantalla del prototipo MSS con 4.000 cubos.	37
Figura 4.3 Escena de la película Star Wars – Episodio II: El Ataque de los Clones	37
Figura 4.4 Imagen del prototipo <i>Coruscant</i>	38
Figura 4.5 Captura de pantalla del eje X Y Z del prototipo <i>Coruscant</i>	39
Figura 5.1 Código para el cálculo de FPS cada frame	42
Figura 5.2 Método <i>Start()</i> de la clase <i>FpsCounter</i>	42
Figura 5.3 Implementación del método <i>DisplayFps()</i>	43
Figura 5.4 Implementación del método <i>SaveFpsData()</i>	44

Figura 5.5 Sección de declaración de variables y método <i>Awake()</i> de la clase <i>Generator</i>	45
Figura 5.6 Implementación del método <i>Update()</i> en la clase <i>Generator</i>	45
Figura 5.7 Implementación de los métodos <i>DestroyCubes()</i> y <i>CreateCubes()</i>	46
Figura 5.8 Lógica de los cubos en la configuración MSS	47
Figura 5.9 Configuración del movimiento del cubo con sobrecalentamiento	48
Figura 5.10 Código de la configuración MCS	48
Figura 5.11 Variables definidas en la configuración MCS.....	49
Figura 5.12 Código del job sin sobrecalentamiento	50
Figura 5.13 Método <i>Move()</i> con sobrecalentamiento inducido	51
Figura 5.14 Código del proceso de <i>baking</i> de <i>GameObject</i> a <i>Entidad</i> de un objeto <i>Generator</i>	52
Figura 5.15 Código de inicialización de un sistema <i>Generator</i> para la configuración DSS	53
Figura 5.16 Código de control de las configuraciones en la POD	54
Figura 5.17 Lógica de destrucción de cubos	55
Figura 5.18 Lógica de creación de cubos	56
Figura 5.19 Código de la configuración DSS para el movimiento de cubos	57
Figura 5.20 Código de la configuración DSC en el que se añaden el sobrecalentamiento	58
Figura 5.21 Programación del Job de movimiento de cubos	58
Figura 5.22 Código de implementación del Job de movimiento de cubos	59
Figura 5.23 Método <i>Execute</i> del job desarrollado para la configuración DCC.....	59
Figura 5.24 Captura de pantalla de una ejecución de la configuración MONO	60
Figura 5.25 Código de generación de edificios	61
Figura 5.26 Código de destrucción de edificios	62
Figura 5.27 Bucle de creación de naves	62
Figura 5.28 Código de movimiento de las naves	63
Figura 5.29 Código de obtención de la duración de la creación y destrucción de edificios.....	63
Figura 5.30 Código del movimiento de las naves en la configuración DOTS.....	64
Figura 6.1 Datos de FPS de las 8 configuraciones del prototipo Cubos	66
Figura 6.2 Resultado de realizar una operación <i>describe()</i> sobre el dataframe de fps	67
Figura 6.3 Gráfica de barras de las configuraciones MSS y DSS.....	68
Figura 6.4 Gráfica de barras de las configuraciones MSC y DSC.....	69
Figura 6.5 Gráfica comparativa de fps entre las configuraciones MCS y DCS.....	70
Figura 6.6 Gráfica comparativa de fps entre las configuraciones MCC y DCC	71
Figura 6.7 Gráfica comparativa de fps entre las configuraciones MSS y MCS	72
Figura 6.8 Gráfica comparativa de fps entre las configuraciones MSC y MCC.....	72
Figura 6.9 Gráfica comparativa de fps entre las configuraciones DSS y DCS	73
Figura 6.10 Gráfica comparativa de fps entre las configuraciones DSC y DCC	74
Figura 6.11 Datos de los FPS en la POO y la POD.....	75
Figura 6.12 Gráfico de barras de los FPS de MONO y DOTS	76

Figura 6.13 Datos de los tiempos de creación de edificios	77
Figura 6.14 Resultado de llamar al método <i>describe()</i> sobre el conjunto de datos	78
Figura 6.15 Gráfica de los tiempos de creación de edificios	78
Figura 6.16 Datos de los tiempos de destrucción de edificios	79
Figura 6.17 Datos de los tiempos de creación de naves	80
Figura 6.18 Gráfica de los tiempos de creación de naves	81
Figura 6.19 Datos de los tiempos de destrucción de naves	82

Capítulo 1

Introducción

1.1 Introducción

El desarrollo de videojuegos y, más en general, el desarrollo de aplicaciones informáticas requiere que el programador sea capaz de crear sistemas que deben ser optimizados para garantizar unos estándares mínimos de eficiencia y rendimiento. Esto es así en prácticamente todos los sectores que comprende la informática. Los videojuegos, además, cuentan con una serie de problemáticas que hacen que necesiten ser aplicaciones altamente eficientes en tiempo real, por lo que el rendimiento y la optimización son vitales para el éxito en el desarrollo de un videojuego.

Desde hace ya muchos años, la POO (Programación Orientada a Objetos) es el paradigma de programación usado en la mayoría de los sistemas y proyectos informáticos, y los videojuegos no iban a ser menos. De hecho, trabajar con Unity hasta hace poco tiempo significaba que el programador solo podía seguir este paradigma de programación, pero esto cambió con la introducción en el motor de Unity DOTS (Data Oriented Technology Stack), un conjunto de librerías que permiten a los desarrolladores utilizar la POD (Programación Orientada a Datos) para el desarrollo de videojuegos.

La POD y, en concreto, Unity DOTS, pretenden mejorar la eficiencia y el rendimiento de los sistemas informáticos que necesitan gestionar grandes cantidades de datos. De hecho, si se visita la página principal de Unity DOTS, se puede ver que la propia

empresa define Unity DOTS como un intento de <<Aplicar el diseño orientado a datos en la arquitectura de un videojuego para empoderar a los desarrolladores a escalar el procesamiento de una forma altamente eficiente>> [1].

Con la llegada al motor de esta librería, los desarrolladores, de acuerdo con Unity, tendrán la capacidad de crear juegos más ambiciosos, juegos que requieren un procesamiento en tiempo real de una gran cantidad de datos, algo que antes era mucho más costoso de realizar, llegando en muchos casos a ser imposible.

Además, Unity DOTS no solo está enfocado al desarrollo de videojuegos para ordenador. Es una buena opción para implementar complejos algoritmos, como algoritmos de generación procedural, pathfinding, inteligencia artificial dinámica, físicas complejas y visualización de datos científicos, entre otros.

También es recomendable para el desarrollo de juegos multijugadores competitivos, en los que es vital que haya rapidez de conexión y precisión en la transmisión de información. Por último, es interesante el uso de Unity DOTS para simulaciones a gran escala, como gemelos digitales de procesos de automatización de fábricas.

Para que los desarrolladores pudiesen llevar a cabo sus ideas antes de la introducción de esta librería, estos tenían que recurrir a crear herramientas personalizadas desde cero, programando a bajo nivel, sin poder disfrutar de las ventajas que ofrece utilizar un motor convencional y consolidado como es Unity. Unity DOTS pretende solventar esta problemática, aportando una serie de herramientas que faciliten a los desarrolladores la creación de sistemas complejos, pero altamente eficientes.

1.2 Motivación

La idea para desarrollar este trabajo surgió a partir de un vídeo publicado por el creador de contenido Tarodev y titulado <<How To Render 2 Million Objects At 120 FPS>> [2], en el que implementa diversos casos de uso utilizando diferentes técnicas, una de ellas siendo Unity DOTS.

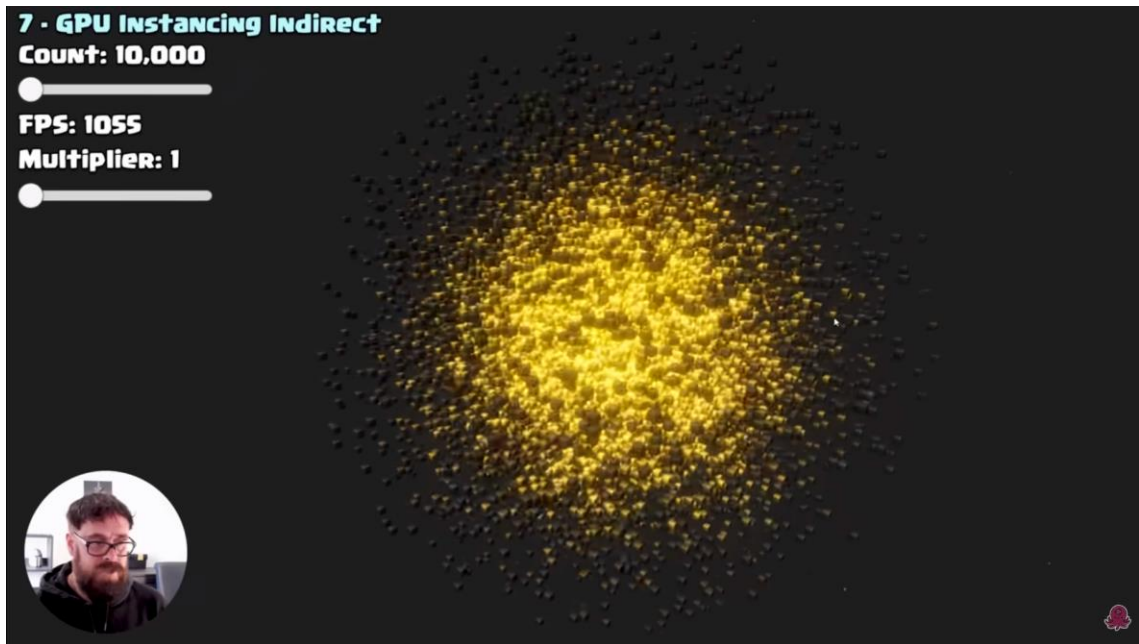


Figura 1.1 Captura de pantalla del vídeo de Tarodev

Aun así, este proyecto solo está inspirado en dicho video, y en ningún momento se ha concebido como una réplica o como una extensión del mismo. Es importante recalcar que Tarodev presenta diversas implementaciones, recurriendo a técnicas como el acceso directo a la GPU para renderizar entidades o al uso de Unity DOTS.

1.3 Objetivos

En este trabajo se pretende ahondar más en profundidad en las diferencias que supone el uso de la POO y la POD en Unity, y no en la comparación con otras posibles técnicas de optimización, como el uso directo de la GPU. Es en este punto cuando surgen algunas preguntas: ¿hasta qué punto es Unity DOTS una solución efectiva? ¿Son este conjunto de librerías realmente capaces de soportar una gran cantidad de jugadores, de enemigos o de entidades en pantalla, manteniendo un número razonable de FPS (Frames Per Second)? Pues son precisamente estas preguntas las que se pretenden responder con este trabajo.

Con este proyecto se aspira a desarrollar diversos casos de uso y realizar un análisis comparativo exhaustivo de los mismos, enfocándose en la comparación entre la POO y la POD. Inicialmente se presentarán prototipos simples que ofrezcan una primera impresión de las diferencias que pueden haber entre ambos paradigmas, y a continuación

se mostrarán prototipos más complejos, para ver hasta qué punto es la POD capaz de sacar una mejora de rendimiento.

Capítulo 2

Estado del Arte

En este capítulo se pretende explicar el funcionamiento tanto de la Programación Orientada a Objetos como de la Programación Orientada a Datos utilizando el motor de Unity, con el objetivo de entender más en profundidad las diferencias entre ambos paradigmas.

2.1 Programación orientada a objetos en Unity

2.1.1 ¿Qué es la POO?

Antes de pasar a ver cómo funciona la programación orientada a objetos en Unity es necesario que se defina claramente qué es la POO. La POO, de acuerdo con Geeks For Geeks en un artículo introductorio al tema, se refiere al *<<uso del concepto de objetos en la programación>>* [3]. Es decir, la POO aspira a implementar diversas ideas del mundo real como herencia, encapsulación, abstracción... al mundo de la programación.

De esta forma se pretende, por un lado, facilitar el entendimiento y el desarrollo de código, haciéndolo más cercano al lenguaje natural humano y, por otro lado, juntar los datos y las funciones que manipulan estos datos en el mismo lugar, de tal forma que cada objeto solo debe acceder y operar los datos que él mismo contiene, garantizando el acceso exclusivo a estos.

Es decir, la POO se concibe para facilitar el trabajo y el entendimiento por parte del programador. Pretende crear una serie de reglas que son cercanas a la forma que tenemos los seres humanos de entender el mundo, de tal forma que el código desarrollado siguiendo este paradigma sea accesible, escalable y modular. Estas propiedades, según muchos defensores de la POO, son cruciales a la hora del desarrollo de aplicaciones informáticas, pues permiten una rápida iteración de los productos y estos son más resilientes frente a cambios de requisitos, pues los cambios en una parte no deben afectar a otras partes dependientes de un mismo proyecto.

De acuerdo con Brian Will en su vídeo *Object-Oriented Programming is Bad*, <<los seres humanos preferimos pensar en términos de párrafos en vez de en oraciones individuales, y la programación orientada a objetos parece tener una respuesta a cómo podemos aplicar esta forma de pensar a la programación>> mediante la combinación de los datos y las funciones en una unidad lógica común como son los objetos [4].

Como se verá más tarde, la POO puede no ser la mejor solución para muchos de los problemas que nos encontramos en informática, pero centrémonos ahora en las distintas partes de las que está compuesta la POO.

2.1.2 Partes de la POO

2.1.2.1 Clases

Las clases son un tipo de dato que trata de aglutinar otros datos que tienen relación entre sí y las funciones que acceden y operan sobre estos datos. De acuerdo con el artículo de Geek for Geeks previamente mencionado, las clases <<representan un conjunto de propiedades o métodos que son comunes a todos los objetos de un tipo concreto>> [3], es decir, como si fuesen unos planos que todos los objetos de dicho tipo tienen que seguir.

2.1.2.2 Objetos

Los objetos son instancias concretas de una clase que existen en tiempo real. <<Un objeto tiene una identidad, un estado y un comportamiento>> [3]. Mientras que las clases son tan solo definiciones, los objetos son instancias reales de estas definiciones que al ser creados ocupan memoria. Además, los objetos pueden interactuar entre sí a través del envío y recepción de mensajes en los que solo es necesario conocer cómo son la entrada y la salida.

2.1.2.3 Abstracción

La abstracción consiste en mostrar al exterior de un objeto solo la información necesaria, sin exponer detalles acerca de cómo está implementada una determinada operación. De esta forma, un objeto que quiera acceder a otro sabrá lo que debe enviar y lo que espera recibir, pero no necesitará conocer cómo se obtendrá ese resultado.

2.1.2.4 Encapsulación

La encapsulación se puede definir como juntar los datos comunes en una unidad individual, que es el objeto, *<<es el mecanismo que une el código y los datos que manipula>>* [3].

Como se verá más adelante, esta es probablemente la mayor diferencia que exista entre la POO y la POD, pues la programación orientada a datos pretende juntar por un lado los datos y por otro las operaciones que acceden a estos datos, mientras que la programación orientada a objetos pretende juntar datos y operaciones que tengan relación entre sí.

2.1.2.5 Herencia

La herencia consiste en la adquisición de funcionalidad del padre hacia el hijo. Es decir, un objeto hijo puede implementar una funcionalidad que tiene el padre, pero esta funcionalidad puede ser modificada para ajustarse a las necesidades del hijo. Es decir, consistiría en compartir una definición, pero, si fuese necesario, en diferenciar la implementación.

2.1.2.6 Polimorfismo

El polimorfismo, de acuerdo con el artículo de Geeks For Geeks, se puede definir como *<<la habilidad de que un mensaje pueda mostrarse de varias formas distintas>>* [3]. Las personas, por ejemplo, no adoptamos un solo rol en nuestra vida, sino que podemos ser a la vez estudiantes, hijos y hermanos. Así mismo ocurriría con un objeto, que podría tener diversas características.

2.1.3 ¿Cómo funciona la POO en Unity?

Unity utiliza C# como lenguaje de *scripting*, un lenguaje fuertemente enfocado al desarrollo de aplicaciones siguiendo el paradigma de la programación orientada a

objetos, bastante similar a Java en muchos aspectos. Aun así, Unity recurre a un concepto propio para designar a todos aquellos objetos que se van a entender como tal dentro del entorno del motor, y es el concepto de *GameObject*.

Los *GameObjects* <<son objetos fundamentales en Unity que representan personajes, props, y el escenario. Estos no logran nada por sí mismos, pero funcionan como contenedores para Componentes, que implementan la verdadera funcionalidad>> [5]. En la figura 2.1 se puede ver como ejemplo de *GameObject* un objeto Generator.

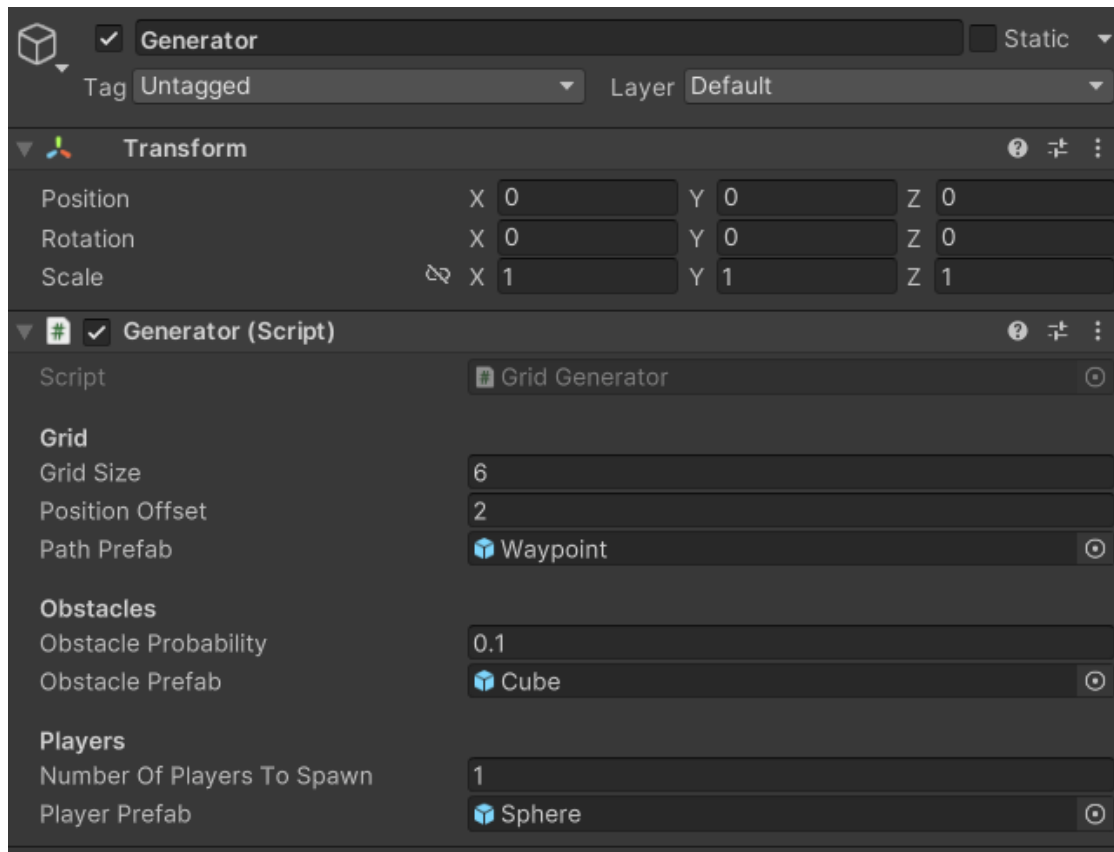
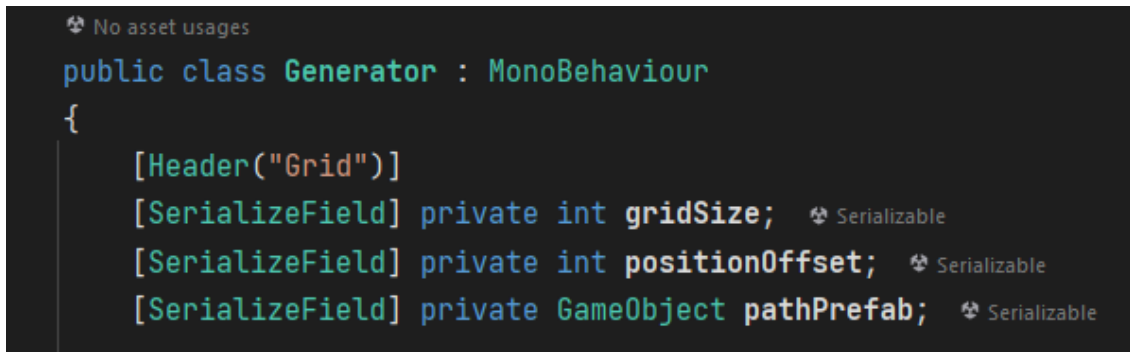


Figura 2.1 Captura de pantalla de un *GameObject* en Unity

Este objeto contiene dos componentes, un componente Transform, que se encarga de almacenar los datos y funciones relacionados con la posición, rotación y escala del objeto, y un componente Generator, que se encarga de crear una serie de puntos y obstáculos en una malla tridimensional.

Los componentes, como se ha mencionado anteriormente, les aportan una funcionalidad concreta a los objetos. Cada componente debe heredar de la clase MonoBehaviour para poder ser asignado a un objeto a través del inspector de Unity, como se puede ver en la figura 2.2.

A screenshot of a code editor showing the C# code for a class named 'Generator' which inherits from 'MonoBehaviour'. The code includes a header 'Grid' and three private fields: 'gridSize' (int), 'positionOffset' (int), and 'pathPrefab' (GameObject). Each field is decorated with the '[SerializeField]' attribute. To the right of each field name, there is a small icon of a box with a plus sign and the word 'Serializable'. At the top left of the code editor, there is a status bar that says 'No asset usages' with a small icon of a box with a plus sign.

```
No asset usages
public class Generator : MonoBehaviour
{
    [Header("Grid")]
    [SerializeField] private int gridSize;
    [SerializeField] private int positionOffset;
    [SerializeField] private GameObject pathPrefab;
```

Figura 2.2 Código en C# del componente Generator

Como cualquier clase normal, este componente Generator tiene variables y métodos. De hecho, al derivar de la clase MonoBehaviour, los componentes son capaces de implementar varios métodos que corresponden al ciclo de vida de los *GameObjects*. Los principales son:

- **Awake():** <<esta función siempre se llama antes de cualquier función Start y también justo después de que un prefab es instanciado>> [6]. Los prefabs son una plantilla de un objeto en Unity, que en tiempo de ejecución pueden ser instanciados o destruidos a conveniencia.
- **OnEnable():** función que se llama cada vez que un objeto pasa de estar inactivo a estar activo.
- **Start():** <<es llamado antes de la primera actualización de frame solo si la instancia del script está activada>> [6].
- **Update():** función que es llamada una sola vez por cada frame. Es la función donde se va a implementar la mayor cantidad de lógica de la aplicación.

2.1.4 Problemática de la POO en Unity

El uso de *GameObjects* en Unity, aun siendo muy conveniente para el programador, trae consigo una serie de problemas que suponen una reducción de rendimiento en los proyectos que los utilizan.

Uno de estos problemas es que los *GameObjects* son <<managed objects>>, lo que significa que no pueden ser utilizados en jobs (que como se verá más adelante permiten paralelizar el código en varios hilos), salvo en circunstancias muy concretas, ni pueden ser compilados a través del compilador *Burst* [7], es decir, no pueden aprovecharse de la programación concurrente, que explicaremos más adelante en este trabajo, ni tampoco pueden aprovechar la eficiente compilación que se realiza de C# a código máquina a través de *Burst*.

Además de esto los *GameObjects*, cuando son destruidos, tienen que ser recogidos por el recolector de basura del que dispone C#, lo cual es un proceso bastante costoso, sobre todo si se intentan destruir una gran cantidad de objetos a la vez. De acuerdo con la propia documentación oficial de Unity, *<<el recolector de basura es impredecible en el cómo libera y asigna memoria, lo que puede llevar a problemas de rendimiento>>*, como cuando la pantalla muestra en intervalos irregulares las imágenes que tiene que renderizar, *<<lo que ocurre cuando el recolector de basura tiene que pararse a liberar y asignar memoria>>* [8].

Es por todo esto por lo que el uso de *GameObjects* en Unity es en muchos casos de uso una solución ineficiente, dando lugar a degradaciones en el rendimiento de la aplicación. ¿Será Unity DOTS capaz de solventar estos problemas?

2.2 Programación orientada a datos en Unity

2.2.1 ¿Qué es la POD?

Yehonathan Sharvit, en un artículo titulado *<<Principles of Data-Oriented Programming>>*, describe la POD como *<<un paradigma de programación enfocado a simplificar el diseño y la implementación de los sistemas de software, donde la información está en el centro de los sistemas como el frontend o el backend de las aplicaciones>>* [9].

El principal objetivo de la programación orientada a datos es la separación del código y los datos. Como veíamos antes, esta premisa va en contraposición a lo que se pretendía conseguir en la programación orientada a objetos, cuyo propósito era el de juntar el código y los datos en unidades individuales, mientras que en la programación orientada a datos se pretende separar completamente los datos que tiene una aplicación de las operaciones que se realizan sobre estos datos.

En la POD, *<<los datos se consideran como ciudadanos de primera clase>>* [9]. De acuerdo con Yehonathan, existen cuatro principios a los que se adhiere la programación orientada a datos:

- **Principio 1:** separar el código (lógica y comportamiento) de los datos.
- **Principio 2:** representar los datos con estructuras de datos genéricas.
- **Principio 3:** tratar los datos como si fueran inmutables.

- **Principio 4:** separar el esquema de datos de su representación.

Como se puede observar, la POD es diametralmente opuesta a la POO, y de hecho pretende resolver los problemas que surgen del uso de la programación orientada a objetos. No es de extrañar que, en los últimos años, haya surgido un movimiento tanto dentro como fuera de la industria del videojuego que esté intentando crear herramientas para facilitar a los desarrolladores la implementación de sistemas siguiendo los principios y la filosofía de la programación orientada a datos. Muchos son los beneficios que promete.

2.2.2 ¿Qué es Unity DOTS y ECS?

Ahora bien, ¿cómo implementa Unity en su motor este paradigma de programación? Pues lo hace a través de un conjunto de librerías que se conocen como DOTS. Esta siglas hacen referencia a Data Oriented Technology Stack, y comprende, principalmente, las siguientes librerías:

- **Entities (ECS):** librería que *<<incluye una implementación orientada a datos de la arquitectura ECS (Entity Component System)>>* [10].
- **Entities Graphics:** librería que *<<provee de sistemas y componentes para renderizar entidades>>*. Es importante recalcar que esta librería no es un *<<render pipeline>>* (que podría definirse como una herramienta capaz de comunicarse con la GPU para dibujar objetos en pantalla), sino que es un *<<sistema que recolecta los datos necesarios para renderizar entidades y enviar esta información a la existente arquitectura de renderizado de Unity>>* [11].
- **Netcode for Entities:** esta librería *<<provee un framework para servidores autoritativos con predicción en cliente que puede ser utilizado para crear videojuegos multijugador>>* [12].
- **Physics:** librería que *<<provee de un sistema determinista y dinámico de rigidbodies, así como de un sistema espacial de consulta>>* [13] que permite a los desarrolladores replicar interacciones físicas utilizando ECS.
- **Job System:** el sistema de jobs de Unity *<<te permite escribir código multihilo de forma simple y segura, de tal forma que tus aplicaciones puedan utilizar todos los núcleos disponibles en la CPU y aumentar el rendimiento>>* [14].
- **Collections:** esta librería contiene *<<estructuras de datos unmanaged que pueden ser utilizadas en jobs y en código compilado por Burst>>* [15]. El uso de estas estructuras de datos es crucial para el desarrollo de aplicaciones con ECS, pues aumentan encarecidamente la eficiencia de los procesos complejos.
- **Burst:** el compilador Burst *<<traduce código escrito en IL/.NET a código nativo altamente optimizado utilizando LLVM (Low Level Virtual Machine)>>* [16]. Las

LLVM's son un conjunto de herramientas que permiten el desarrollo y creación de compiladores.

- **Mathematics:** esta es una librería de matemáticas de C# que <<provee de tipos de vectores y funciones matemáticas que tienen una sintaxis similar a lenguajes de Shaders como SIMD o HLSL>> [17]. El uso de esta librería es necesario pues, como hemos dicho con anterioridad, no es posible compilar código con Burst que tiene referencia a objetos o componentes *managed*, y las librerías Mathematics y Collections proveen de funciones que trabajan con datos *unmanaged*. Esto es importante, pues la diferencia entre componentes *managed* y *unmanaged* reside principalmente en que los primeros deberán ser recolectados por el recolector de basura de C#, mientras que los componentes *unmanaged* no. Este hecho supondrá diferencias de rendimiento que habrá que tener en cuenta.

A través del uso de todas estas librerías es cómo los desarrolladores son capaces de aplicar la POD. En las siguientes secciones se explicará de forma más profunda las partes más importantes que componen la POD en Unity.

2.2.2.1 Entidades

La primera palabra que compone las siglas ECS es la de Entidades. ¿Qué son las entidades? De acuerdo con Brian Will en un vídeo introductorio a Unity DOTS <<las entidades son simplemente cosas, que tienen asociadas un número entero como id y que pueden tener uno o varios componentes asociados, con un máximo de un componente por tipo>> [18]. Además, en contraposición a cómo funcionan los objetos, las entidades no tienen implícitamente el concepto de parentesco, es decir, una entidad no puede ser padre o hijo de otra. Eso es algo exclusivo de las clases y objetos.

Para poder crear entidades, primero es necesario crear un mundo, que básicamente son contenedores de entidades, y asociado a este mundo existe un EntityManager, que es una clase que provee Unity y a través de la cual se pueden crear y destruir entidades y añadir o quitar componentes de entidades [19]. Alguno de los métodos que provee la clase EntityManager son:

- **CreateEntity():** método que permite crear entidades en el mundo asociado.
- **DestroyEntity():** método que permite destruir entidades en el mundo asociado.
- **AddComponent():** método que permite añadir a una o varias entidades un componente concreto.
- **RemoveComponent():** método que permite eliminar a una o varias entidades un componente concreto.

2.2.2.2 Componentes

La segunda palabra que compone las siglas ECS es la de Componentes. Estos generalmente están pensados para ser solamente conjuntos de datos. No definen funciones de ningún tipo. ¿Cómo se puede crear un componente en Unity? Pues creando un *struct* y haciéndole implementar la interfaz *IComponentData*, tal y como se puede ver en la figura 2.3:

```
public struct BuildingsGeneratorComponent : IComponentData
{
    public Entity buildingEntity;
    public int initialBuildingsAmount;
    public float positionOffset;
}
```

Figura 2.3 Código en C# de un componente en Unity

Como se puede observar, este componente no define ningún método y solo contiene tres campos: una entidad edificio, el número inicial de edificios a construir y un número que servirá como offset para establecer la posición final de cada edificio.

Es importante recalcar, una vez más, que un componente no debe contener referencias a ningún tipo de dato *managed*, pues no se podrán aprovechar las características que nos ofrece el compilador Burst.

Los componentes son datos asociados a una entidad, están guardados en memoria en conjuntos denominados *chunks* y se integran con mucha facilidad con el sistema de jobs de C#. Ahora bien, ¿por qué los componentes son *unmanaged*, aparte de para poder compilarlos con Burst?

Pues de acuerdo con Brian Will, <<los componentes son datos *unmanaged* para que podamos conocer el tamaño exacto que ocupan en memoria>> [18]. Los chunks son realmente secciones preasignadas de memoria, y la mayor ventaja que se puede obtener por esto se observa cuando se procesa toda esta información, porque todos los datos similares entre si van a estar en posiciones de memoria contiguas, es decir, van a estar al lado. Esto implica que realizar consultas para acceder o modificar estos datos es un proceso realmente eficiente.

2.2.2.3 Baking

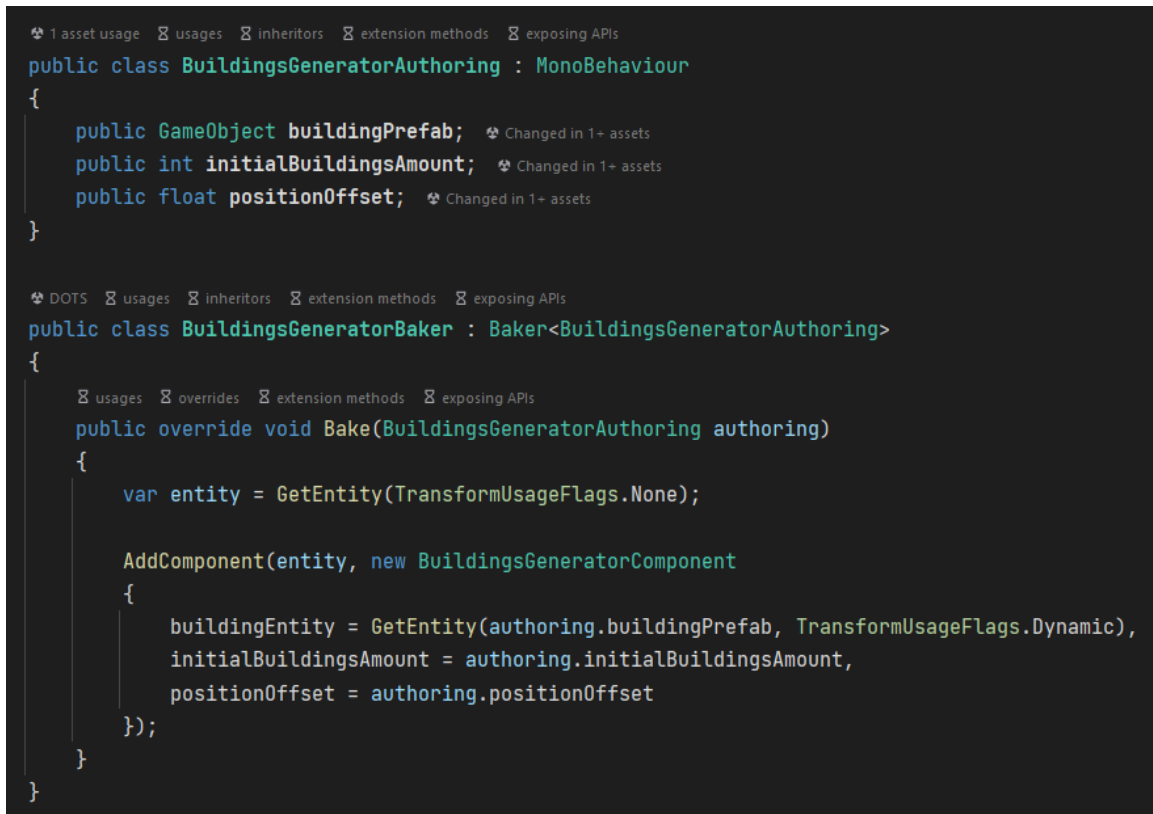
Para entender bien el proceso de *baking*, primero es necesario conocer qué es una escena en Unity. De acuerdo con la documentación oficial de Unity, <<las escenas contienen los entornos y menús del juego>> [24]. Es decir, son archivos que contienen información acerca de los obstáculos, el terreno y, en general, de todos los objetos que van a haber en el juego. Normalmente, un juego contendrá varias escenas para representar diferentes localizaciones.

Ahora bien, las escenas en Unity no pueden contener directamente entidades, por lo que es necesario cargar las entidades en lo que se conocen como subescenas, y para realizar esta carga se tiene que recurrir a un proceso que se conoce como *baking*.

El *baking*, de acuerdo con Brian Will en un vídeo titulado *Unity ECS Baking*, consiste en <<procesar cada subescena del juego con el objetivo de producir un conjunto de entidades serializadas>> [20]. Cuando una subescena es cargada, esta serializa (que consiste en transformar la información para procesarla posteriormente en un formato concreto) todas las entidades y no los *GameObjects*. Es decir, en el editor de Unity nosotros deberemos tratar con *GameObjects* que serán convertidos (*baked*) a entidades en tiempo de ejecución al cargar la subescena. De hecho, el proceso de *baking* consiste en cinco pasos [20]:

1. Un mundo de *baking* es creado para guardar todas las entidades creadas desde una subescena.
2. Por cada *GameObject* que haya en la subescena se añade una entidad a este nuevo mundo creado.
3. Por cada componente de cada *GameObject* que haya en la subescena que tenga asociado una clase Baker, se llama al método *Bake()* de esta clase.
4. Después de que todos los Baker se hayan llamado, los sistemas de inicialización del mundo de *baking* son llamados. El concepto de sistemas lo veremos más en detalle en partes posteriores de este trabajo. Por suerte, y a diferencia de las clases de *baking*, los sistemas sí que pueden ser compilados mediante *Burst* y procesar todas las entidades en masa.
5. Después de que todas las entidades hayan sido procesadas por un Baker, el último paso es serializar el mundo de *baking*, y el resultado de esta serialización es lo que se quedará cargado cuando la escena sea cargada en tiempo de ejecución.

Véase ahora un ejemplo de cómo se podría implementar este proceso:



```

1 asset usage  X usages  X inheritors  X extension methods  X exposing APIs
public class BuildingsGeneratorAuthoring : MonoBehaviour
{
    public GameObject buildingPrefab;  X Changed in 1+ assets
    public int initialBuildingsAmount;  X Changed in 1+ assets
    public float positionOffset;  X Changed in 1+ assets
}

DOTS  X usages  X inheritors  X extension methods  X exposing APIs
public class BuildingsGeneratorBaker : Baker<BuildingsGeneratorAuthoring>
{
    X usages  X overrides  X extension methods  X exposing APIs
    public override void Bake(BuildingsGeneratorAuthoring authoring)
    {
        var entity = GetEntity(TransformUsageFlags.None);

        AddComponent(entity, new BuildingsGeneratorComponent
        {
            buildingEntity = GetEntity(authoring.buildingPrefab, TransformUsageFlags.Dynamic),
            initialBuildingsAmount = authoring.initialBuildingsAmount,
            positionOffset = authoring.positionOffset
        });
    }
}

```

Figura 2.4 Código en C# de un objeto y su correspondiente clase Baker

Si en la figura 2.3 se podía ver una implementación de un componente *BuildingsGenerator*, en la figura 2.4 se puede observar la implementación del proceso de *baking* de dicho *GameObject*.

En este caso se tiene la clase *BuildingsGeneratorAuthoring*, que deriva de *MonoBehaviour*, lo que lo convierte en un componente diseñado para ser añadido a un *GameObject*. Debajo de esta clase se define otra clase llamada *BuildingsGeneratorBaker*, que deriva de *Baker*, al que se le pasa la clase anterior como parámetro genérico.

Asimismo, se sobreescribe el método *Bake* de esta clase, cuya función consiste en obtener la entidad asociada a dicho *GameObject*, que se hace a través del método *GetEntity*, y añadirle un componente *BuildingsGeneratorComponent*, asignando las tres variables que tiene este componente. De esta forma se consigue transformar un componente para *GameObjects* en un componente para entidades.

A continuación se verá cómo se guardan y estructuran realmente en memoria las entidades.

2.2.2.4 Arquetipos

De acuerdo con Brian Will, <<las entidades de un mundo se dividen en arquetipos, donde cada arquetipo guarda todas las entidades de que contienen una combinación especial de componentes. Dentro de cada arquetipo, las entidades y sus componentes se guardan en bloques de tamaño uniforme de memoria llamados chunks>> [18].

Es decir, un arquetipo A B C contendrá todos los chunks de entidades que tienen los componentes A, B y C. Asimismo, otro arquetipo A B contendrá todos los chunks de entidades que tienen solo los componentes A y B, como se puede ver en la figura 2.5.

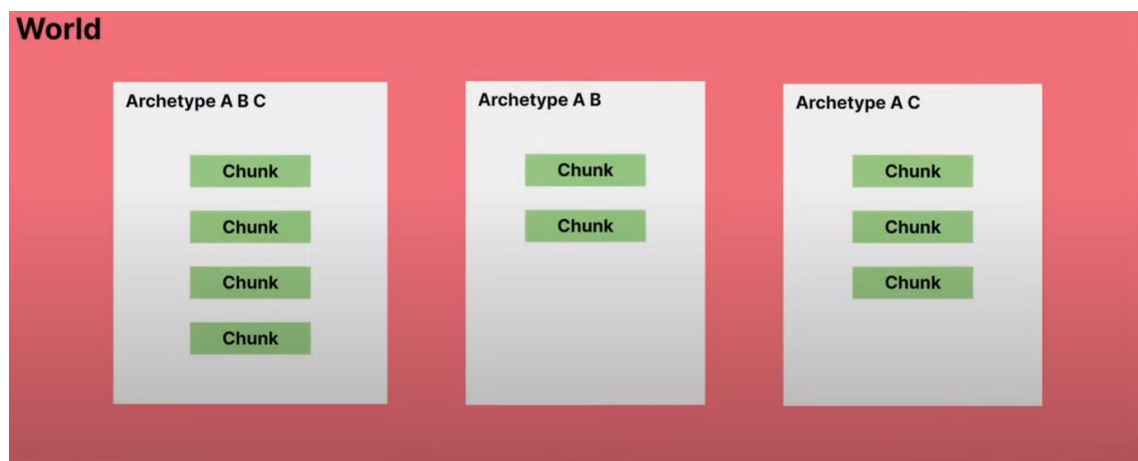


Figura 2.5 Captura de pantalla del vídeo de Brian Will sobre entidades, arquetipos y chunks [18]

Evidentemente, cuando se añaden o se eliminan componentes de una entidad, esta va a tener que ser movida de arquetipo, pues ya no contendrá el mismo conjunto de componentes que antes. Por suerte, de acuerdo con Brian Will, <<los usuarios no tienen que lidiar con este problema de forma manual, solo tienen que llamar a los métodos del EntityManager para crear y destruir entidades o para añadir y eliminar componentes de entidades>> [18].

Dentro de cada arquetipo se puede ver que existen diversos chunks. Brian Will comenta que un chunk tiene una capacidad de hasta 128 entidades, y en ellos se guarda un array por cada componente que tengan las entidades, como se puede ver en la figura 2.6.

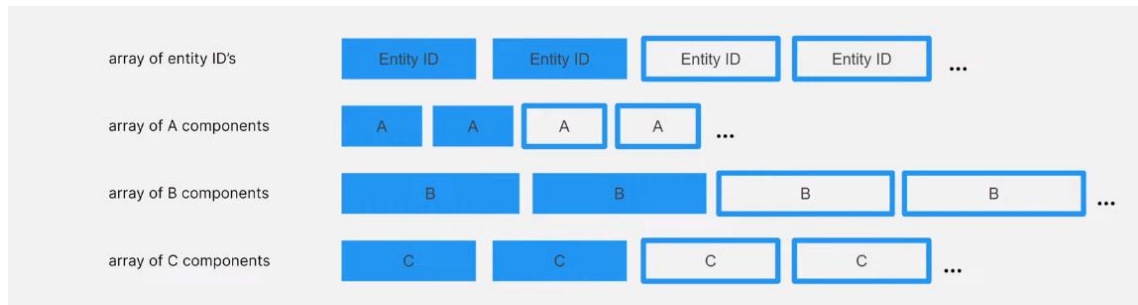


Figura 2.6 Captura de pantalla del video de Brian Will sobre entidades, arquetipos y chunks [18]

También es importante recalcar que, como se puede ver en la figura 2.6, algunos componentes pueden no ocupar en memoria tanto tamaño como el resto.

2.2.2.5 Sistemas

Los sistemas son una pieza fundamental de ECS, y representan la tercera letra de dichas siglas. En concreto, los sistemas son la lógica que accede y opera sobre los componentes para hacer que se sucedan los distintos eventos de un videojuego o aplicación. Estos corren automáticamente sobre los grupos de entidades que hemos explicado en el apartado anterior [21].

Los sistemas se ejecutan en el hilo principal, pero pueden ser programados por jobs para correr en diversos hilos. Como se puede ver en la figura 2.7 (que representa el código de la estructura de un sistema de ejemplo), los sistemas tienen que estar definidos como *public partial struct*, e implementan la interfaz *ISystem*.

```
[BurstCompile]
✱ DOTS
public partial struct SistemaEjemplo : ISystem
{
    [BurstCompile]
    public void OnCreate(ref SystemState state)
    {
        // ...
    }

    [BurstCompile]
    public void OnDestroy(ref SystemState state)
    {
        // ...
    }

    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        // ...
    }
}
```

Figura 2.7 Código en C# de la estructura de un sistema

Al implementar esta interfaz se pueden utilizar y sobrescribir tres métodos:

- **OnCreate():** método que se llama antes que el primer *Update*.
- **OnDestroy():** método que se llama cuando la instancia del sistema se elimina del mundo o el propio mundo es destruido.
- **OnUpdate():** se llama una vez por frame.

Asimismo, se puede observar cómo tanto la estructura *SistemaEjemplo* como los métodos que implementa tienen arriba el atributo *[BurstCompile]*. Este atributo es necesario si queremos que el código sea compilado a través de *Burst*. Esto lo veremos con más detalle en el apartado siguiente de este trabajo.

Cada sistema pertenece a un mundo, y normalmente solo los sistemas de ese mundo son los que acceden a las entidades de dicho mundo. El parámetro *SystemState* pasado por referencia nos da acceso a los sistemas del mundo y a su *EntityManager*, lo que nos permitirá realizar las consultas de acceso y modificación de componentes de forma eficiente.

De acuerdo con Brian Will, <<los sistemas de un mundo se organizan en una jerarquía de grupos de sistemas en los que cada grupo tiene como hijos otros sistemas y otros grupos de sistemas>> [21]. De hecho, si se ejecuta una aplicación hecha con Unity DOTS en el editor de Unity, en la pestaña *Systems* (figura 2.8) se puede ver la jerarquía de

sistemas y grupos de sistemas que existen en la aplicación. Por defecto se crean tres grupos principales:

- **Grupo de inicialización:** grupo de sistemas encargados de tareas de configuración e inicialización.
- **Grupo de simulación:** grupo de sistemas que se encargan de la lógica del juego.
- **Grupo de presentación:** grupo de sistemas encargados de renderizar todas las entidades en pantalla.

Systems	World	Namespace
▼ Initialization		
▼ Initialization System Group		
Begin Initialization Entity Command Buffer System	Editor World	Unity.Entities
Retain Blob Asset System	Editor World	Unity.Entities
World Update Allocator Reset System	Editor World	Unity.Entities
Live Conversion Editor System Group	Editor World	Unity.Scenes.Editor
Scene System Group	Editor World	Unity.Scenes
End Initialization Entity Command Buffer System	Editor World	Unity.Entities
▼ Update		
▼ Simulation System Group		
Begin Simulation Entity Command Buffer System	Editor World	Unity.Entities
Fixed Step Simulation System Group	Editor World	Unity.Entities
Variable Rate Simulation System Group	Editor World	Unity.Entities
Companion Game Object Update System	Editor World	Unity.Entities
Transform System Group	Editor World	Unity.Transforms
Companion Game Object Update Transform System	Editor World	Unity.Entities
Late Simulation System Group	Editor World	Unity.Entities
End Simulation Entity Command Buffer System	Editor World	Unity.Entities
▼ Pre Late Update		
▼ Presentation System Group		
Begin Presentation Entity Command Buffer System	Editor World	Unity.Entities
Hybrid Light Baking Data System	Editor World	Unity.Entities
Register Materials And Meshes System	Editor World	Unity.Rendering
Deformations In Presentation	Editor World	Unity.Rendering
Structural Change Presentation System Group	Editor World	Unity.Rendering
Update Presentation System Group	Editor World	Unity.Rendering
Entities Graphics System	Editor World	Unity.Rendering
Matrix Previous System	Editor World	Unity.Rendering

Figura 2.8 Captura de la pestaña de *Systems* de una aplicación en Unity DOTS

Para entender mejor cómo funcionan los sistemas, se puede ver en la figura 2.9 un ejemplo de un sistema en el que cada vez que un jugador presione la tecla Q, se crearán una cantidad cada vez mayor de edificios.

```

[BurstCompile]
DOTS
public partial struct SistemaEjemplo : ISystem
{
    private int _buildingsAmount;

    [BurstCompile]
    public void OnCreate(ref SystemState state)
    {
        state.RequireForUpdate<BuildingsGeneratorComponent>();
    }

    [BurstCompile]
    public void OnDestroy(ref SystemState state) { }

    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        if (!Input.GetKeyDown(KeyCode.Q))
            return;

        BuildingsGeneratorComponent spawnerComponent = SystemAPI.GetSingleton<BuildingsGeneratorComponent>();

        float positionOffset = spawnerComponent.positionOffset;

        float x = _buildingsAmount / 2f;

        for (float i = -x; i < x; i++)
        {
            for (float j = -x; j < x; j++)
            {
                Entity building = state.EntityManager.Instantiate(spawnerComponent.buildingEntity);

                state.EntityManager.SetComponentData(building, componentData: new LocalTransform
                {
                    Position = new float3(x * i, 0f, x * j) * positionOffset,
                    Scale = 1f,
                    Rotation = quaternion.identity
                });
            }
        }

        _buildingsAmount += 5;
    }
}

```

Figura 2.9 Código en C# de un sistema de creación de edificios

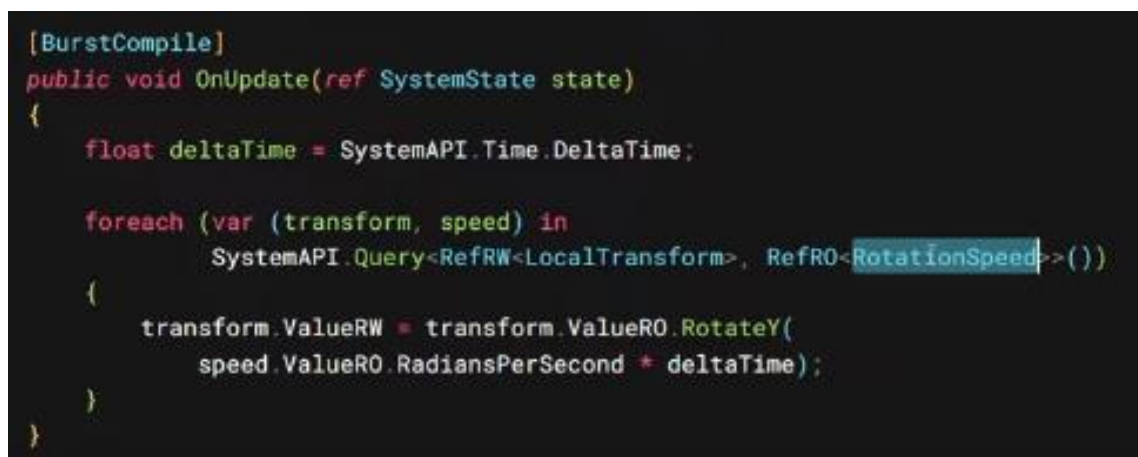
Inicialmente se puede ver como en el método *OnCreate()* se añade una llamada a *state.RequireForUpdate<BuildingsGeneratorComponent>()*. Con esta llamada lo que se pretende es decirle a Unity que este sistema no puede ser lanzado hasta que haya, al menos, una entidad en el mundo que tenga un componente *BuildingsGeneratorComponent*.

Dentro del método *OnUpdate()* primero se comprueba si se ha presionado la letra Q, en cuyo caso se procede a crear los edificios.

Primero se obtiene el componente *BuildingsGeneratorComponent*, que como solo va a existir una entidad con dicho componente en el mundo, podemos obtenerlo a través de *SystemAPI.GetSingleton<>()*. Esta es una forma muy conveniente de obtener este componente, pero hay que asegurarse de que solo exista una entidad con dicho componente. De lo contrario, la aplicación lanzará una excepción.

Después se puede observar un bucle encargado de crear los edificios. Primero se instancia la entidad del edificio a través del método *Instantiate()* de la clase *EntityManager*, y posteriormente se modifica su componente *LocalTransform* para establecerle una nueva posición de aparición.

Es importante recalcar que en este sistema no se está realizando ninguna consulta. Un ejemplo de una consulta en un sistema lo podemos ver en la figura 2.10.



```
[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    float deltaTime = SystemAPI.Time.DeltaTime;

    foreach (var (transform, speed) in
        SystemAPI.Query<RefRW<LocalTransform>, RefRO<RotationSpeed>>())
    {
        transform.ValueRW = transform.ValueRO.RotateY(
            speed.ValueRO.RadiansPerSecond * deltaTime);
    }
}
```

Figura 2.10 Captura de pantalla del vídeo de Brian Will de un sistema que rota entidades de forma constante [22]

En este caso, en el método *OnUpdate()* del sistema se realiza una consulta a través de *SystemAPI.Query()*. Esta consulta buscará entre todas las entidades que hay en la subescena aquellas que cumplan con los componentes que se especifican, que en este caso son los componentes *LocalTransform* y *RotationSpeed*.

Es interesante recalcar que, en esta consulta, la referencia que se obtiene del componente *LocalTransform*, está marcada como *RefRW*, mientras que la referencia que se obtiene del componente *RotationSpeed* está marcada como *RefRO*. Esto significa que la referencia al componente *LocalTransform* es una referencia de tipo *Read-Write*, lo que

supone que se puedan modificar los valores de dicho componente. En el caso del componente *RotationSpeed*, la referencia es de tipo *Read-Only*, por lo que solo se podrá acceder a los datos del componente, pero no se podrán modificar.

2.2.2.6 Aspectos

Los aspectos son agrupaciones de componentes que pretenden simplificar la definición de las consultas. Es decir, los aspectos son una forma más conveniente de agrupar componentes para facilitar la creación de consultas dentro de un sistema. Un ejemplo sería la figura 2.11, en el que se define un aspecto para el ejemplo anterior.

```
readonly partial struct VerticalMovementAspect : IAspect
{
    readonly RefRW<LocalTransform> m_Transform;
    readonly RefRO<RotationSpeed> m_Speed;

    public void Move(double elapsedTime)
    {
        m_Transform.ValueRW.Position.y = (float)math.sin(elapsedTime * m_Speed.ValueRO.RadiansPerSecond);
    }
}
```

Figura 2.11 Captura de pantalla del código de un aspecto del vídeo de Brian Will [22]

Este aspecto agrupa los componentes *LocalTransform* y *RotationSpeed*, y define un método *Move()*, que como se puede observar contiene el código que en el ejemplo de la figura 2.10 se establecía dentro del bucle *foreach*. Después de haber definido este aspecto, podemos simplificar la consulta, como se puede ver en la figura 2.12.

```
foreach (var movement in
    SystemAPI.Query<VerticalMovementAspect>())
{
    movement.Move(elapsedTime);
}
```

Figura 2.12 Captura de pantalla del código de la consulta utilizando aspectos del vídeo de Brian Will [22]

Como se puede ver, los sistemas son un engranaje fundamental de la programación orientada a datos en Unity DOTS, ya que permiten crear consultas altamente eficientes para acceder y modificar los datos almacenados en componentes.

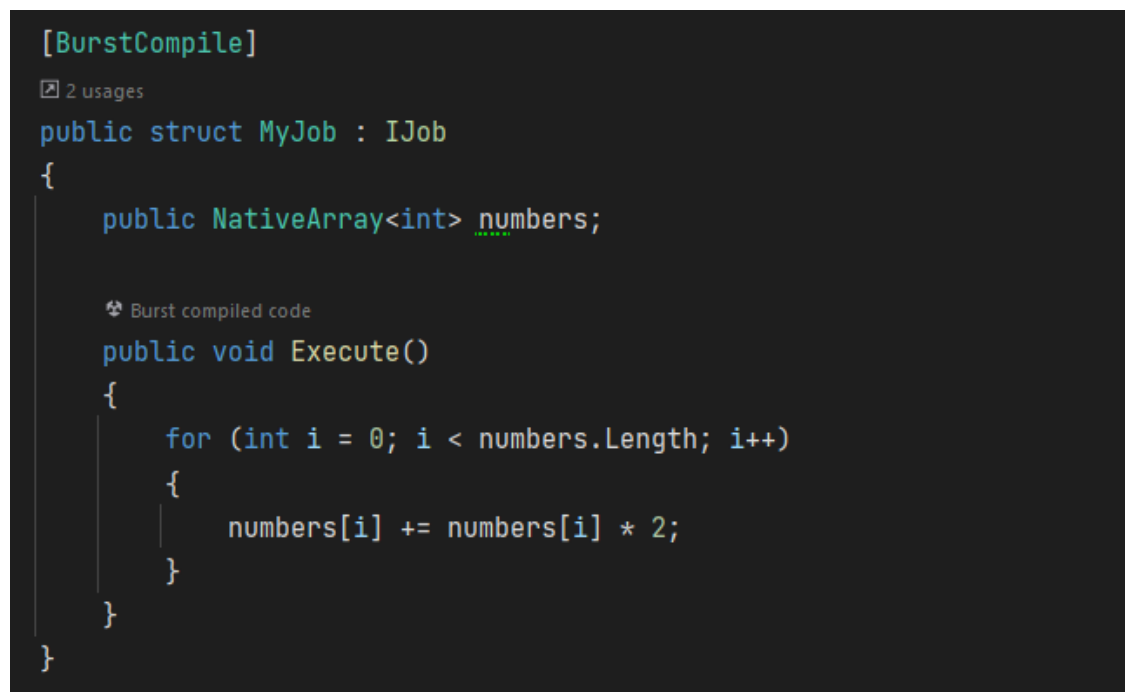
2.2.2.7 C# Job System

El Job System de C# es una librería que permite aprovechar las ventajas de la programación concurrente en el desarrollo de aplicaciones y videojuegos con Unity. Este no es exclusivo de Unity DOTS, sino que también puede ser utilizado en clases *MonoBehaviour*.

El Job System de C# incluye una serie de comprobaciones de seguridad para evitar *race conditions* (condición de secuencia) y otros problemas típicos de la programación concurrente, y está pensado para ser usado en cálculos y operaciones muy pesadas que necesitan ser realizadas en grandes cantidades de forma frecuente.

De acuerdo con Brian Will en su video *The Unity Job System* [23], Unity puede crear hilos adicionales al hilo principal, uno por cada núcleo que tenga la CPU del ordenador donde se ejecuta el juego. Los programadores pueden sacar ventaja de estos núcleos adicionales gracias al Job System de C#.

Ahora bien, ¿qué son los jobs? Pues <<un job es simplemente un grupo de trabajo que es ejecutado en uno de los hilos adicionales>> [23], y un ejemplo de cómo se podría crear un job se puede observar en la figura 2.13.



```

[BurstCompile]
2 usages
public struct MyJob : IJob
{
    public NativeArray<int> numbers;

    Burst compiled code
    public void Execute()
    {
        for (int i = 0; i < numbers.Length; i++)
        {
            numbers[i] += numbers[i] * 2;
        }
    }
}

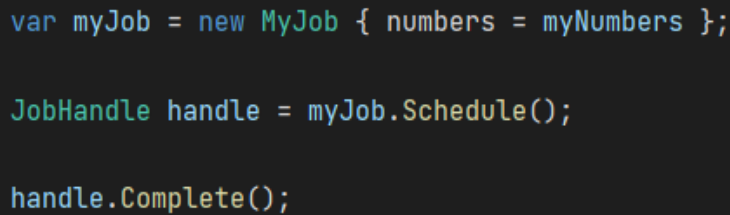
```

Figura 2.13 Captura de pantalla de la definición de un job en C#

Los jobs deben ser de tipo *struct* e implementar la interfaz *IJob*. En el ejemplo de la figura 2.13 se pretende crear un job que sume a los números de un array el doble de sí

misimos. En este caso estamos utilizando un *NativeArray* en vez de un array normal de C# debido a que queremos compilar el código utilizando el compilador *Burst*, y como veremos en la siguiente sección de este trabajo, este compilador solo puede compilar datos que son básicos o manejados.

Ahora bien, para poder ejecutar un job, necesitamos primero crear una instancia del mismo y programarlo. Esto se puede ver en detalle en la figura 2.14.

A screenshot of a code editor with a dark background and light-colored text. The code is written in C# and consists of three lines. The first line creates a new instance of a class named 'MyJob' and assigns it to a variable named 'myJob'. The 'numbers' property of 'myJob' is set to 'myNumbers'. The second line calls the 'Schedule()' method on 'myJob' and assigns the returned 'JobHandle' to a variable named 'handle'. The third line calls the 'Complete()' method on 'handle'.

```
var myJob = new MyJob { numbers = myNumbers };  
  
JobHandle handle = myJob.Schedule();  
  
handle.Complete();
```

Figura 2.14 Captura de pantalla de la creación de un job en C#

En este caso estamos primero creando una instancia de *MyJob* al que le pasamos el array de números que queremos modificar. Posteriormente llamamos al método *Schedule()* de este job. Este método <<se encarga de poner la instancia del job en la cola global de jobs para que, posteriormente, un hilo sin trabajo pueda acceder a la cola y encontrar este job para ejecutarlo>> [23].

Para finalizar, es necesario llamar al método *Complete()* sobre el *handle* devuelto por el job al programarlo. Es importante recalcar que un job solo podrá ser programado y completado desde el hilo principal. Un job no puede programar o completar otros jobs. Además, si se intenta acceder a los datos que tiene un job entre la llamada al método *Schedule()* y la llamada al método *Complete()*, Unity lanzará una excepción de comprobación de seguridad.

Tampoco se puede programar un job si un job anterior no ha sido completado (véase la figura 2.15).

```
// Ejemplo de excepción

var myFirstJob = new MyJob { numbers = myNumbersOne };
var mySecondJob = new MyJob { numbers = myNumbersTwo };

JobHandle handleOne = myFirstJob.Schedule();
JobHandle handleTwo = mySecondJob.Schedule();

handleOne.Complete();
handleTwo.Complete();
```

Figura 2.15 Código C# de una excepción de seguridad al programar dos jobs

Pero este problema lo podemos solventar a través del uso de dependencias entre jobs. Véase un ejemplo con la figura 2.16.

```
// Ejemplo de dependencias

var myFirstJob = new MyJob { numbers = myNumbersOne };
var mySecondJob = new MyJob { numbers = myNumbersTwo };

JobHandle handleOne = myFirstJob.Schedule();
JobHandle handleTwo = mySecondJob.Schedule(handleOne);

handleOne.Complete();
handleTwo.Complete();
```

Figura 2.16 Código C# de una dependencia entre dos jobs

Las dependencias entre jobs <<nos permiten establecer un orden de ejecución entre nuestros jobs cuando nos sea necesario>> [23]. Además, estas dependencias pueden a su vez tener sus propias dependencias, por lo que podremos crear cadenas de dependencias (que no tienen por qué ser lineales, además de que un job pueda tener más de una dependencia).

Otro punto importante que mencionar antes de pasar a la siguiente sección es que la paralelización de un código puede no mejorar su rendimiento. Esto es debido a que, a veces, un código no es tan complejo o costoso como para que tenga que ser dividido en hilos. En ciertas ocasiones las lecturas y escrituras que se producen en

memoria por la creación y destrucción de jobs puede ser más costoso que la propia lógica que se produce dentro del job. Es por este motivo por lo que más adelante en este trabajo se compararán diversos casos de uso en los que en unos se utilizarán jobs y en otros no, para ver realmente hasta qué punto podemos obtener una mejora de eficiencia o no.

2.2.2.8 Compilador Burst

Como se ha mencionado anteriormente, el compilador *Burst* es un compilador que <<traduce código IL/NET a código nativo altamente optimizado>> [16]. Este compilador está diseñado, además, para trabajar de forma conjunta eficientemente con el Job System de C# que hemos visto en la sección anterior.

Compilar nuestro código con *Burst* es tan sencillo como añadir el atributo *[BurstCompile]* sobre la definición de la estructura y de los métodos de nuestro job, como se puede ver en la figura 2.13.

Pero como ya se ha comentado en alguna ocasión a lo largo de este trabajo, compilar el código con *Burst* requiere de ciertas condiciones. Por ejemplo, una de ellas es que *Burst* solo permite los tipos primitivos, que son <<*bool, char, sbyte/byte, short/ushort, int/uint, long/ulong, float y double*>> [16], mientras que no permite tipos como *string* (que es un tipo manejado) o decimal. Tampoco permite clases, por eso es necesario el uso de estructuras.

Burst es una herramienta muy potente que podemos aprovechar a la hora de crear aplicaciones con Unity, pero nos fuerza a pensar nuestro código de forma diferente. Como veremos más tarde, para la obtención de algunas métricas ha sido necesaria la eliminación del atributo *BurstCompile* sobre las clases y métodos de algunas partes del código, puesto que era necesario acceder a librerías que no eran compatibles con *Burst*.

Capítulo 3

Metodología

<<Las decisiones que tomas sobre tu forma de trabajar impactan el producto final tanto o más que las decisiones que tomas sobre el propio producto>>

[Autor desconocido]

3.1 Qué es una metodología

De acuerdo con Maida y Pacienza en su tesis de licenciatura, *<<una metodología es un conjunto integrado de técnicas y métodos que permite abordar de forma homogénea y abierta cada una de las actividades del ciclo de vida de un proyecto de desarrollo. Es un proceso de software detallado y completo>>* [25].

Es decir, las metodologías ofrecen a los desarrolladores una serie de reglas que les permiten estandarizar el proceso de creación de productos software de forma que se aumente la eficiencia y se reduzcan los costes de desarrollo. Bajo esta definición, queda claro la vital importancia que supone el uso de una metodología a la hora de crear un buen producto de software.

3.2 Metodología aplicada

Ahora bien, debido a la característica comparativa y analítica de este trabajo se hace difícil atenerse a una de las metodologías que se pueden encontrar dentro del mundo de la ingeniería de software, como puede ser Lean UX, metodologías en cascada o metodologías ágiles, por poner algunos ejemplos. El desarrollo del presente trabajo siguiendo una de estas metodologías habría sido una desventaja que hubiera dañado el resultado del mismo.

Es por esto por lo que se ha preferido seguir una *Evaluación Comparativa* [26], una metodología utilizada principalmente en campos relacionados con las ciencias y las ciencias sociales pero que, para este caso en particular, nos resulta de gran ayuda a la hora de desarrollar el presente trabajo, y esto es debido a la comparación y posterior evaluación de las dos tecnologías estudiadas con el objetivo de entender cuál de ellas sería mejor utilizar en diferentes casos de uso, con tal de aportar información a la toma de decisiones de los desarrolladores de videojuegos.

De acuerdo con Vartiainen, <<la expresión "evaluación comparativa" se refiere a la investigación en la cual una evaluación y los resultados del proceso de evaluación se sitúan en un marco comparativo>> [25]. Es decir, el objetivo de esta metodología reside en investigar diversos objetos de estudio para posteriormente comparar sus resultados siguiendo los mismos criterios de evaluación para todos.

Se presupone que, después de una evaluación comparativa, el desarrollador habrá obtenido una pieza de información la cual le permitirá tomar mejores decisiones de cara al futuro. En el caso del presente trabajo, se ha decidido dividir el desarrollo del mismo en cuatro fases, tal y como se pueden ver en la figura 3.1.

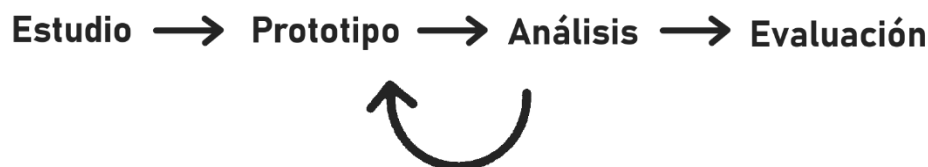


Figura 3.1 Esquema de las fases seguidas durante el desarrollo del presente trabajo

3.2.1 Estudio

La primera etapa del trabajo ha consistido en el estudio y aprendizaje de cómo funcionan los dos paradigmas de programación que se pretenden comparar. Para ello, se han seguido una serie de tutoriales explicativos sobre ambos paradigmas, así como la realización de diversos ejercicios de prueba y la lectura de la documentación oficial de Unity y Unity DOTS. El principal recurso de referencia utilizado ha sido el *roadmap* oficial que publicó Unity en un repositorio de GitHub [27], que es el itinerario recomendado para iniciarse en la POD con Unity DOTS.

3.2.2 Prototipo

Una vez se tenía una comprensión completa y suficiente sobre cómo funciona la POO y la POD en Unity, el siguiente paso ha consistido en la realización de un prototipo de prueba sencillo que permitiese mostrar, de forma rápida, si las promesas de rendimiento y mejora que promete Unity DOTS se cumplían.

Para realizar la comparación, era necesario que se desarrollase el mismo prototipo siguiendo ambos paradigmas. Esto es de vital importancia, pues pequeños cambios en la forma que se implementan algunas mecánicas pueden impactar severamente a los resultados obtenidos, invalidándolos.

3.2.3 Análisis

Una vez se tenían ambos prototipos listos, se procedió a la obtención de métricas (como el tiempo de creación/destrucción de entidades o los FPS a los que está llegando el prototipo) para realizar un primer análisis comparativo entre ambos paradigmas. Estos datos se guardan en un archivo json que posteriormente es procesado para la obtención de gráficas y métricas, las cuáles nos aportan información sobre el rendimiento del prototipo desarrollado en ambos paradigmas.

Como se puede observar en la figura 3.1, hay una flecha que parte de la fase de Análisis y llega a la fase de Prototipo. Esta flecha indica que, una vez terminado el análisis comparativo entre los dos prototipos desarrollados, se puede desarrollar otro prototipo diferente, con tal de obtener diferentes métricas y datos sobre rendimiento bajo distintas casuísticas.

3.2.4 Evaluación

Una vez se está conforme con los resultados obtenidos tras el análisis comparativo entre los distintos prototipos, se procede a la evaluación global entre ambos paradigmas, referenciando los resultados obtenidos en los diferentes análisis realizados, con el objetivo de obtener una respuesta a las preguntas iniciales planteadas en el inicio de este trabajo.

En los capítulos posteriores se procederá a explicar más profundamente cada una de estas fases, entrando en detalles sobre la implementación, el diseño de los prototipos y su posterior análisis.

Capítulo 4

Diseño

Llegados a este punto del presente trabajo se debe responder a la siguiente pregunta: ¿qué tipo de prototipos se deben diseñar, y cómo deben ser diseñados, para obtener datos fiables que permitan dar respuesta a las preguntas que fueron planteadas en un principio? A lo largo de esta cuarta sección se pretende dar respuesta a esta pregunta.

4.1 Variables a analizar

4.1.1 FPS

Primero de todo, ¿qué variables se pueden medir que ofrezcan resultados honestos sobre el rendimiento de los prototipos? La primera y más importante variable que se va a analizar son los FPS, cuyas siglas en inglés se refieren a *Frames Per Second* (imágenes por segundo). ¿Por qué esta variable es la más importante de todas las que se pretenden analizar?

Pues los FPS son una pieza fundamental en el procesado que realiza el cerebro de un jugador cuando este se encuentra jugando a un videojuego. Si un juego no es capaz de renderizar imágenes lo suficientemente rápido, los FPS disminuirán, lo que supondrá

que el videojuego se sienta torpe y poco reactivo, y esta situación es muy preocupante, pues un jugador no se sentirá inmerso en la experiencia de juego.

Además, los FPS son un estándar de rendimiento dentro de la industria del videojuego, precisamente por este motivo que se acaba de comentar. Establecer una comparativa de rendimiento utilizando los FPS como variable para la comparación es la forma más honesta y fácil de medir a la hora de realizar las pruebas sobre los prototipos, aunque no vaya a ser la única.

4.1.2 Tiempos de creación y destrucción

Otras variables que se pretenden también analizar son los tiempos de creación y destrucción de entidades y objetos, es decir, cuanto tiempo tarda en ejecutarse una parte de código concreta.

La obtención de estas dos variables introduce una complicación, pues para poder medir el tiempo entre dos sucesos, es necesario acceder a varias clases que nos ofrece Unity. El problema reside en que estas clases son *managed*, lo que supone que no se pueda utilizar el compilador Burst para compilar el código en el que se realicen llamadas a esas clases.

Es por esto por lo que los datos de creación y destrucción de objetos y entidades solo van a poder obtenerse con prototipos que no utilicen el compilador Burst, dando pie a obtener unos resultados que no serían del todo fieles a la realidad, pues en un videojuego real no se necesitaría medir ese tiempo y, por lo tanto, se podría hacer uso del compilador Burst.

4.2 Prototipos que desarrollar

Por todo esto, se ha decidido que se crearán dos prototipos diferentes, y para cada prototipo se va a desarrollar código que pueda ser compilado por Burst y utilizar jobs, y código que no pueda ser compilado por Burst y tampoco utilizar jobs.

De esta forma, en el caso de este último código se podrá incorporar la lógica necesaria para almacenar datos respecto a los tiempos de creación y destrucción de entidades y objetos, mientras que en el caso del primer código solo se podrá obtener una comparativa de los FPS de ambos prototipos.

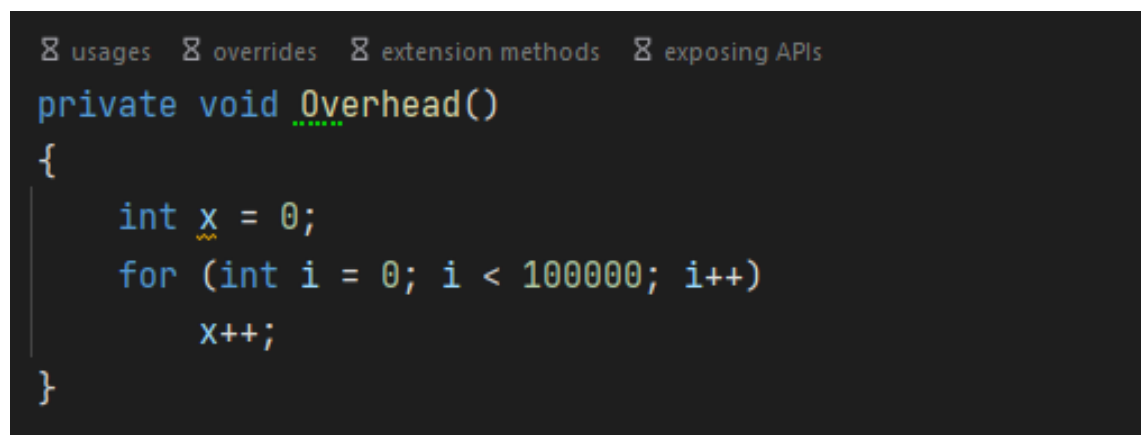
4.2.1 Sobrecalentamiento inducido

Por último, y como se ha comentado previamente en este trabajo en la sección 2.2.2.7 (C# Job System), existe una gran probabilidad de que no se obtenga una mejora de rendimiento tras paralelizar el código con jobs. Por esto, e inspirado por el vídeo de Ketra Games [28], se ha decidido introducir un *overhead* o sobrecalentamiento artificial al código que se pretende comparar.

El motivo para la introducción de este sobrecalentamiento es que el código utilizado para mover, rotar o escalar objetos es muy sencillo, y esta lógica no requiere apenas tiempo de procesamiento por parte del procesador. Debido a esto, si se paraleliza este código, las escrituras y lecturas en memoria que son necesarias hacer por cada job que quiera ser ejecutado pueden tardar más tiempo que la propia lógica que intentamos paralelizar.

Es decir, solo en el caso de que la lógica que se pretenda paralelizar tarde más que el tiempo de estas lecturas y escrituras el beneficio del uso de jobs será aparente. En el resto de los casos no es recomendable su uso.

Así, para hacer que el código tarde más y poder evaluar de forma fidedigna el uso de jobs en la POO y la POD, se ha considerado la introducción de un sobrecalentamiento inducido. Un ejemplo de un posible código de sobrecalentamiento se puede observar en la figura 4.1.

A screenshot of a code editor showing a C# method named `Overhead`. The method is private and void. It contains a loop that initializes an integer `x` to 0, then iterates from `i = 0` to `i < 100000`, incrementing `x` by 1 in each iteration. The code is written in a dark-themed editor with syntax highlighting. At the top of the editor, there are tabs for 'usages', 'overrides', 'extension methods', and 'exposing APIs'.

```
usages overrides extension methods exposing APIs
private void Overhead()
{
    int x = 0;
    for (int i = 0; i < 100000; i++)
        x++;
}
```

Figura 4.1 Código de sobrecalentamiento

El sobrecalentamiento consiste sencillamente en un bucle que realiza una operación más o menos sencilla un gran número de veces (en el caso de la figura 4.1, 100.000 veces). De esta forma, si se decide llamar a la función *Overhead* desde el código

de movimiento de entidades, este código se hará mucho más lento, convirtiendo este código en un código perfecto para ser paralelizado.

4.2.2 Configuraciones de los diferentes prototipos

Para garantizar un conjunto de pruebas exhaustivo, se ha considerado conveniente crear un prototipo con hasta ocho configuraciones diferentes que se pueden analizar, comparar y evaluar. Estas configuraciones son las siguientes:

- **MSS (Mono – Sin jobs – Sin sobrecalentamiento):** configuración desarrollada con la POO, sin paralelización de trabajo y sin sobrecalentamiento. Esta configuración sería la configuración estándar con la que se trabaja en Unity en la gran mayoría de casos de uso.
- **DSS (Dots – Sin jobs – Sin sobrecalentamiento):** configuración desarrollada con la POD, sin paralelización de trabajo y sin sobrecalentamiento.
- **MCS (Mono – Con jobs – Sin sobrecalentamiento):** configuración desarrollada con la POO, con paralelización de trabajo y sin sobrecalentamiento.
- **DCS (Dots – Con jobs – Sin sobrecalentamiento):** configuración desarrollada con la POD, con paralelización de trabajo y sin sobrecalentamiento.
- **MSC (Mono – Sin jobs – Con sobrecalentamiento):** configuración desarrollada con la POO, sin paralelización de trabajo y con sobrecalentamiento.
- **DSC (Dots – Sin jobs – Con sobrecalentamiento):** configuración desarrollada con la POD, sin paralelización de trabajo y con sobrecalentamiento.
- **MCC (Mono – Con jobs – Con sobrecalentamiento):** configuración desarrollada con la POO, con paralelización de trabajo y con sobrecalentamiento.
- **DCC (Dots – Con jobs – Con sobrecalentamiento):** configuración desarrollada con la POD, con paralelización de trabajo y con sobrecalentamiento.

Como se puede observar, estas configuraciones se han emparejado de tal manera que siempre se esté comparando la POO y la POD bajo las mismas circunstancias, lo cual es vital para la obtención de resultados fiables.

La primera pareja de configuraciones (MSS y DSS) se puede considerar como el caso por defecto, aquella que se implementaría en los videojuegos que no requieren gran capacidad de cómputo, por lo que no necesitan paralelizar la lógica del juego.

La segunda pareja (MCS y DCS) es útil para comparar si, en caso de que la lógica del juego no tarde mucho tiempo en ser ejecutada, se obtendría un mejor rendimiento tras la paralelización del código o no.

La tercera pareja (MSC y DSC) nos sirve para ver cuánta pérdida de rendimiento se produce frente a la primera pareja de configuraciones en caso el código a ejecutar si

que sea costoso (que podría ser el caso de la aplicación de un algoritmo de pathfinding, por ejemplo).

La cuarta y última pareja (MCC y DCC) nos es útil para ver si realmente, frente a un código que es costo de procesar, la paralelización del mismo nos produce un aumento significativo del rendimiento.

4.2.3 Los dos prototipos

Para poder llevar a cabo las pruebas pertinentes, se han decidido crear dos prototipos diferentes, cada uno de ellos con sus propias configuraciones, que explicaremos más adelante en este trabajo.

Como se ha comentado en secciones anteriores del presente documento, la programación orientada a datos impone una serie de restricciones que se deben tener en cuenta a la hora de diseñar los prototipos.

Una de estas limitaciones consiste en la imposibilidad de comunicar sistemas entre sí. Es decir, en la POO dos objetos pueden comunicarse entre sí accediendo a métodos públicos que se hayan definido, mientras que en la POD un sistema no puede comunicarse en caso de un evento concreto con otro sistema.

Se hace necesario por lo tanto una forma de que el código creado sea independiente entre sí pero que a su vez reaccione a un evento concreto. Este evento, por simplicidad, se ha decidido que sea la pulsación de la tecla Q.

Tanto en la versión desarrollada con POO como en la versión desarrollada con POD se va a realizar una comprobación tal que cuando se presione la letra Q, se procederá a reiniciar el estado del juego. Por ejemplo, inicialmente no hay ninguna entidad creada en escena. Cuando se pulse la letra Q, se crearán un número X de entidades, que ejecutarán la lógica pertinente de su comportamiento. Cuando se vuelva a presionar la letra Q, las entidades existentes serán destruidas y un grupo mayor de entidades será creado.

De esta forma, se pueden obtener datos referentes a los FPS del juego con un número cada vez mayor de entidades, lo que permite conocer el rendimiento de cada uno de los paradigmas con números de entidades diverso y en aumento.

4.2.3.1 Prototipo *Cubos*

Como punto de partida, se ha decidido desarrollar un prototipo en el que, inicialmente, se crean 1000 cubos y, cada vez que se pulsa la letra Q, se destruyen estos cubos y se vuelven a crear otros nuevos, esta vez aumentando en 1000 el número de cubos. Es decir, inicialmente se crearán 1.000 cubos, después 2.000, después 3.000, y así hasta 20.000 cubos, que es el número a partir del cual se ha observado que el rendimiento empieza a bajar mucho en todas las configuraciones (este efecto es en gran medida debido al hardware utilizado en las pruebas, que será explicado con más detalle en una sección posterior).

Estos cubos serán creados cada uno en una posición aleatoria en un rango que comprende una esfera de tamaño con radio igual a 1 metro. Ningún cubo podrá ser creado exactamente en el centro del mundo (0,0,0).

Una vez creados, los cubos seguirán la dirección que se les haya asignado al ser creados con una velocidad determinada de 10 m/s lo que, en conjunto, parecerá como una esfera formada por cubos que se hace cada vez más grande.

En cuatro de las configuraciones que se han desarrollado, al código del movimiento del cubo se le ha añadido un overhead, como hemos explicado en secciones anteriores. Este overhead es una forma de simular el cálculo de lo que podría ser un algoritmo completo de búsqueda de caminos, por ejemplo. Para mayor claridad se puede observar la figura 4.2, que representa la configuración MSS.

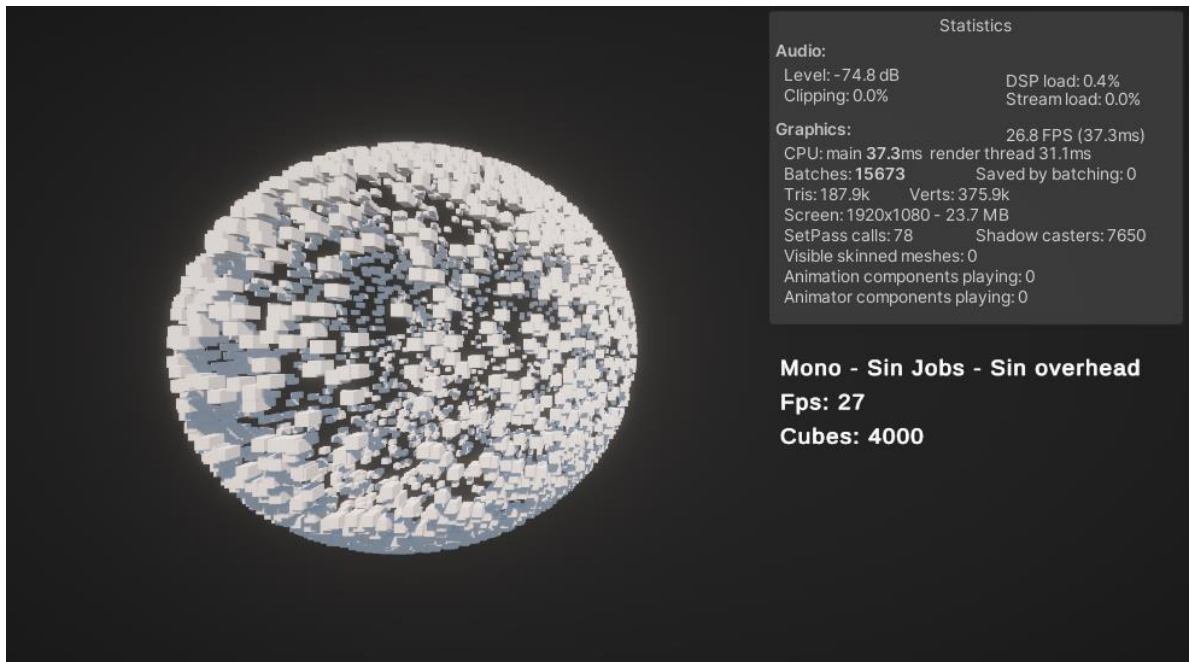


Figura 4.2 Captura de pantalla del prototipo MSS con 4.000 cubos.

4.2.3.2 Prototipo *Coruscant*

Después de haber desarrollado el primer prototipo, surgió la idea de crear un escenario que fuese más creíble de ver implementado en un videojuego real. Tras esto, y debido a la gran influencia que tiene Star Wars en la cultura moderna, se decidió replicar la capital de *Coruscant*, la cual puede verse en la figura 4.3.



Figura 4.3 Escena de la película Star Wars – Episodio II: El Ataque de los Clones

El universo de Star Wars es particularmente popular por la capacidad que tienen todo tipo de vehículos de volar. Esto hace que escenas como la que se puede ver en la figura 4.3 sean de especial interés para los espectadores. Esta escena muestra, en el

fondo, una gran cantidad de naves voladoras que se mueven a través de grandes edificios. Esta es precisamente la idea que se extrajo para realizar el prototipo.

Este prototipo contiene, por un lado, los edificios, y, por otro lado, las naves voladoras que se mueven entre ellos, tal y como se puede ver en la figura 4.4.

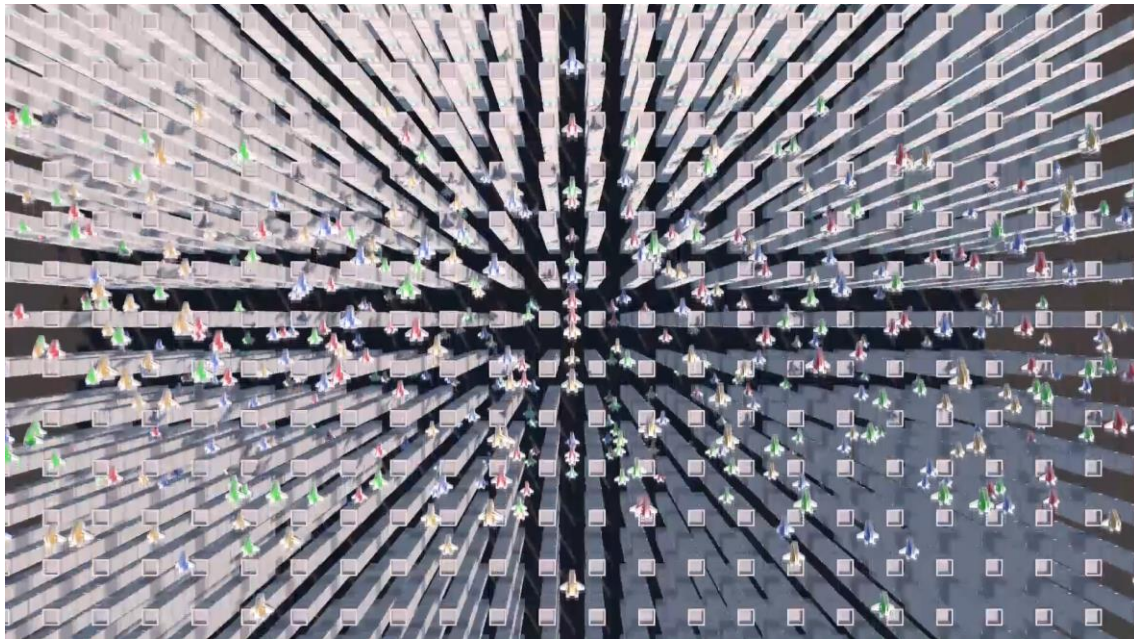


Figura 4.4 Imagen del prototipo *Coruscant*

Con este prototipo el objetivo es obtener, por un lado, una comparativa de FPS con un entorno más cercano a lo que podría ser un videojuego real, y, por otro lado, obtener los datos del tiempo que tardan en crearse y destruir números cada vez mayores de edificios y de naves voladoras, cuyos modelos son más complejos que los cubos utilizados en el prototipo anterior.

En este prototipo se crean inicialmente 100 naves y 100 edificios, después 225 naves y 225 edificios, después 400 naves y 400 edificios, y así hasta llegar a las 10.000 naves y los 10.000 edificios. Esta secuencia sigue un patrón concreto, que consiste en tener un número X al que cada iteración se le suma 5. Esta variable está iniciada a 10, y el valor del número de entidades y edificios cada iteración se consigue multiplicando X por sí mismo. ¿Por qué hacemos esto? Pues porque en este caso las naves y los edificios se están creando en una malla 2×2 .

Es decir, si queremos crear 100 edificios, la forma más sencilla de hacerlo es sacar la raíz cuadrada de este número e implementar dos bucles, cada uno de ellos que se encargue de representar cada uno de los ejes. Mientras que los edificios se

crearían en el plano X Z, las naves se crearían en el plano X Y, tal y como se puede ver en la figura 4.5.



Figura 4.5 Captura de pantalla del eje X Y Z del prototipo *Coruscant*

Es preciso recalcar que los modelos utilizados para el desarrollo de este prototipo se han obtenido de los paquetes publicados de forma gratuita por Kenney en su página web oficial [29].

4.2.4 Hardware utilizado para las pruebas

Para poder entender algunos de los resultados que se obtendrán al realizar las pruebas, es interesante conocer el hardware sobre el que se ejecutan dichas pruebas. Además, es crucial que todas las pruebas sean ejecutadas bajo las mismas condiciones, y esto no solo incluye al desarrollo de los propios prototipos, sino también a la máquina que ejecuta las pruebas.

En el caso del presente trabajo, todas y cada una de las pruebas han sido ejecutadas en un portátil MSI GF63 Thin 10SCXR, que tiene un procesador Intel Core i7-10750H de 2.60 GHz, 16GB de memoria RAM, una gráfica NVIDIA GTX 1650 y un sistema operativo Windows 11.

Capítulo 5

Implementación

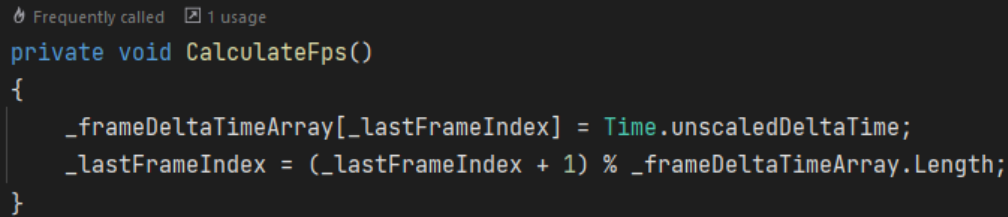
En esta sección del trabajo se pretende explicar cómo se ha llevado a cabo, después de la fase de diseño, la implementación de los dos prototipos y de sus diferentes configuraciones mencionadas en la sección anterior.

5.1 Implementación del prototipo *Cubos*

5.1.1 Aspectos comunes a todas las configuraciones

Para garantizar que la obtención de información sea independiente del tipo de configuración que se está intentando analizar, se ha decidido crear una clase llamada *FpsCounter* encargada de obtener la media de FPS que se han registrado en el transcurso de una iteración de la configuración.

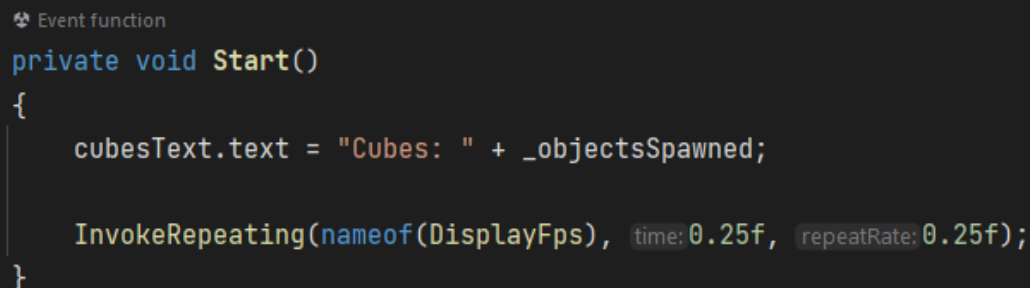
Es decir, en cada frame del juego se va a llamar a una función *CalculateFps()*, que se encarga de guardar en un array *_frameDeltaTimeArray* el tiempo *unscaledDeltaTime*, tal y como se puede ver en la figura 5.1



```
Frequently called 1 usage
private void CalculateFps()
{
    _frameDeltaTimeArray[_lastFrameIndex] = Time.unscaledDeltaTime;
    _lastFrameIndex = (_lastFrameIndex + 1) % _frameDeltaTimeArray.Length;
}
```

Figura 5.1 Código para el cálculo de FPS cada frame

Esta es una forma de evitar la influencia de picos extraordinarios de FPS, ya que el objetivo de este código es crear el array de FPS para después obtener la media de todos los FPS obtenidos durante un periodo determinado de tiempo. De hecho, cada 0.25 segundos se invoca la función *DisplayFps()*, tal y como se puede ver en la figura 5.2.

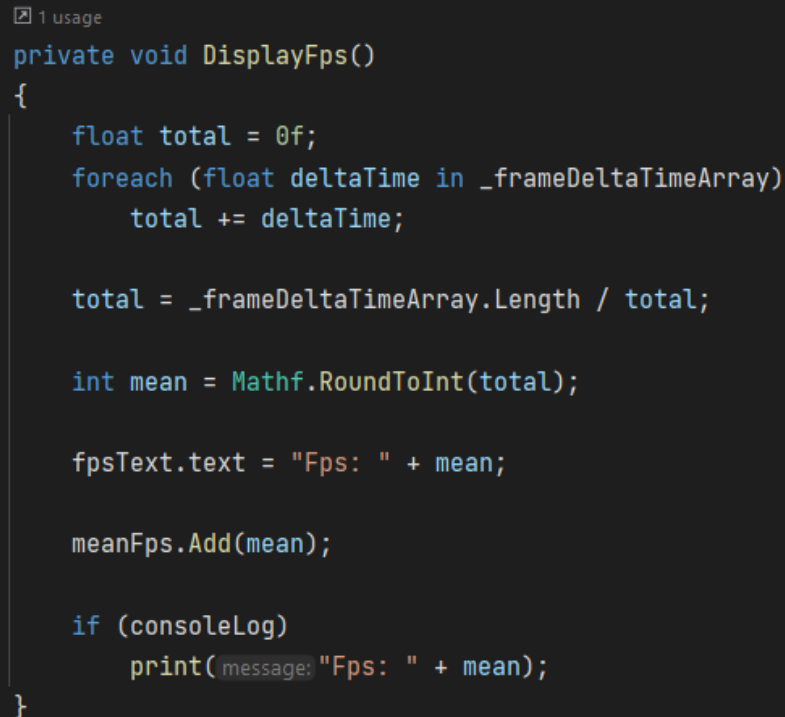


```
Event function
private void Start()
{
    cubesText.text = "Cubes: " + _objectsSpawned;

    InvokeRepeating(nameof(DisplayFps), time: 0.25f, repeatRate: 0.25f);
}
```

Figura 5.2 Método *Start()* de la clase *FpsCounter*

Esta función se encarga (véase la figura 5.3) de obtener la media de todos los FPS obtenidos hasta el momento de la llamada a la función, y una vez calculada esta media se muestra por pantalla y se guarda en una lista de fps.



```
1 usage
private void DisplayFps()
{
    float total = 0f;
    foreach (float deltaTime in _frameDeltaTimeArray)
        total += deltaTime;

    total = _frameDeltaTimeArray.Length / total;

    int mean = Mathf.RoundToInt(total);

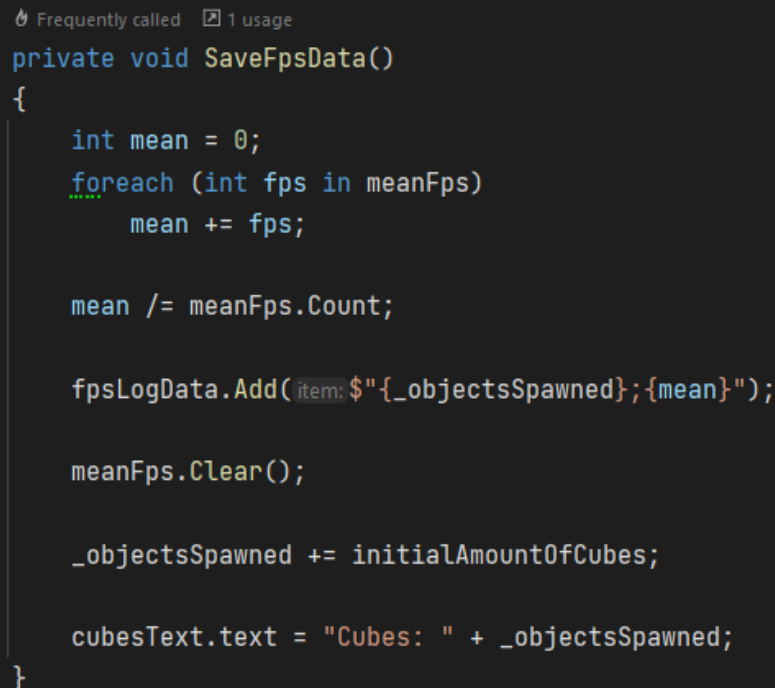
    fpsText.text = "Fps: " + mean;

    meanFps.Add(mean);

    if (consoleLog)
        print(message: "Fps: " + mean);
}
```

Figura 5.3 Implementación del método DisplayFps()

Cuando se presiona la letra Q, se procede a llamar a la función *SaveFpsData()*, que como se puede ver en la figura 5.4, obtiene la media de todos los fps que se han guardado para un determinado número de objetos en pantalla, que a su vez se guardan en otra lista que será utilizada para crear un archivo .csv para su posterior análisis.



```
Frequently called 1 usage
private void SaveFpsData()
{
    int mean = 0;
    foreach (int fps in meanFps)
        mean += fps;

    mean /= meanFps.Count;

    fpsLogData.Add(item: $"{_objectsSpawned};{mean}");

    meanFps.Clear();

    _objectsSpawned += initialAmountOfCubes;

    cubesText.text = "Cubes: " + _objectsSpawned;
}
```

Figura 5.4 Implementación del método SaveFpsData()

La clase *FpsCounter* permite obtener los fps asociados a cualquier configuración que queramos probar, ya que es un código independiente de la implementación de la creación de los cubos con DOTS o con MONO.

5.1.2 Configuraciones de la POO

5.1.2.1 Código para generar los cubos con la POO

Antes de pasar a explicar la implementación de cada una de las configuraciones que utilizan la programación orientada a objetos, es preciso comentar cómo funciona el código de creación de cubos.

Para crear cubos en las configuraciones de la POO se recurre a una clase *Generator*, tal y como se puede ver en la figura 5.5. Esta clase deriva de la clase *MonoBehaviour*, lo que significa que va a existir un objeto al que añadamos este componente.

```

1 asset usage
public class Generator : MonoBehaviour
{
    [SerializeField] private int initialAmountOfCubes; 1000
    [SerializeField] private CubeType type; MSS
    [SerializeField] private Transform cubesParent; Cubes Parent (Transform)
    [SerializeField] private GameObject[] cubePrefabs; Serializable

    private int _amountOfCubes;
    private GameObject _cubePrefab;

    Event function
    private void Awake()
    {
        _cubePrefab = cubePrefabs[(int)type];
    }
}

```

Figura 5.5 Sección de declaración de variables y método *Awake()* de la clase *Generator*

En el arranque de la configuración se llamará al método *Awake()*, que se encarga de obtener el tipo de cubo según la configuración que hayamos escogido desde el editor de Unity. Esto se hace mediante la variable *type*, que es una variable de tipo enum, cuyos posibles valores son MSS, MCS, MSC y MCC, es decir, cada una de las distintas configuraciones de la POO.

El método *Update()* definido en esta clase (véase la figura 5.6) se encarga de comprobar si se presiona la letra Q, en cuyo caso se van a destruir los cubos que haya en pantalla para crear la siguiente tanda de cubos.

```

Event function
private void Update()
{
    if (!Input.GetKeyDown(KeyCode.Q))
        return;

    DestroyCubes();

    CreateCubes();
}

```

Figura 5.6 Implementación del método *Update()* en la clase *Generator*

La implementación de cada uno de estos métodos es sencilla, y se pueden ver en la figura 5.7. Para destruir los cubos simplemente se accede al padre que los va a guardar y se destruyen todos los hijos que tenga definidos.

En el caso de la lógica para crear los cubos, inicialmente se aumenta en 1000 el número de cubos a crear y, a través de un bucle *for*, se calcula una posición aleatoria para cada cubo. Una vez obtenida esta posición, se instancia un cubo con el método *Instantiate()*.

```
Frequently called 1 usage
private void DestroyCubes()
{
    foreach (Transform cube in cubesParent)
    {
        Destroy(cube.gameObject);
    }
}

Frequently called 1 usage
private void CreateCubes()
{
    _amountOfCubes += initialAmountOfCubes;

    for (int i = 0; i < _amountOfCubes; i++)
    {
        Vector3 cubePosition;
        do {
            cubePosition = new Vector3(
                Random.Range(-0.1f, 0.1f), Random.Range(-0.1f, 0.1f), Random.Range(-0.1f, 0.1f));
        } while (cubePosition == Vector3.zero);

        Instantiate(_cubePrefab, cubePosition, Quaternion.identity, cubesParent);
    }
}
```

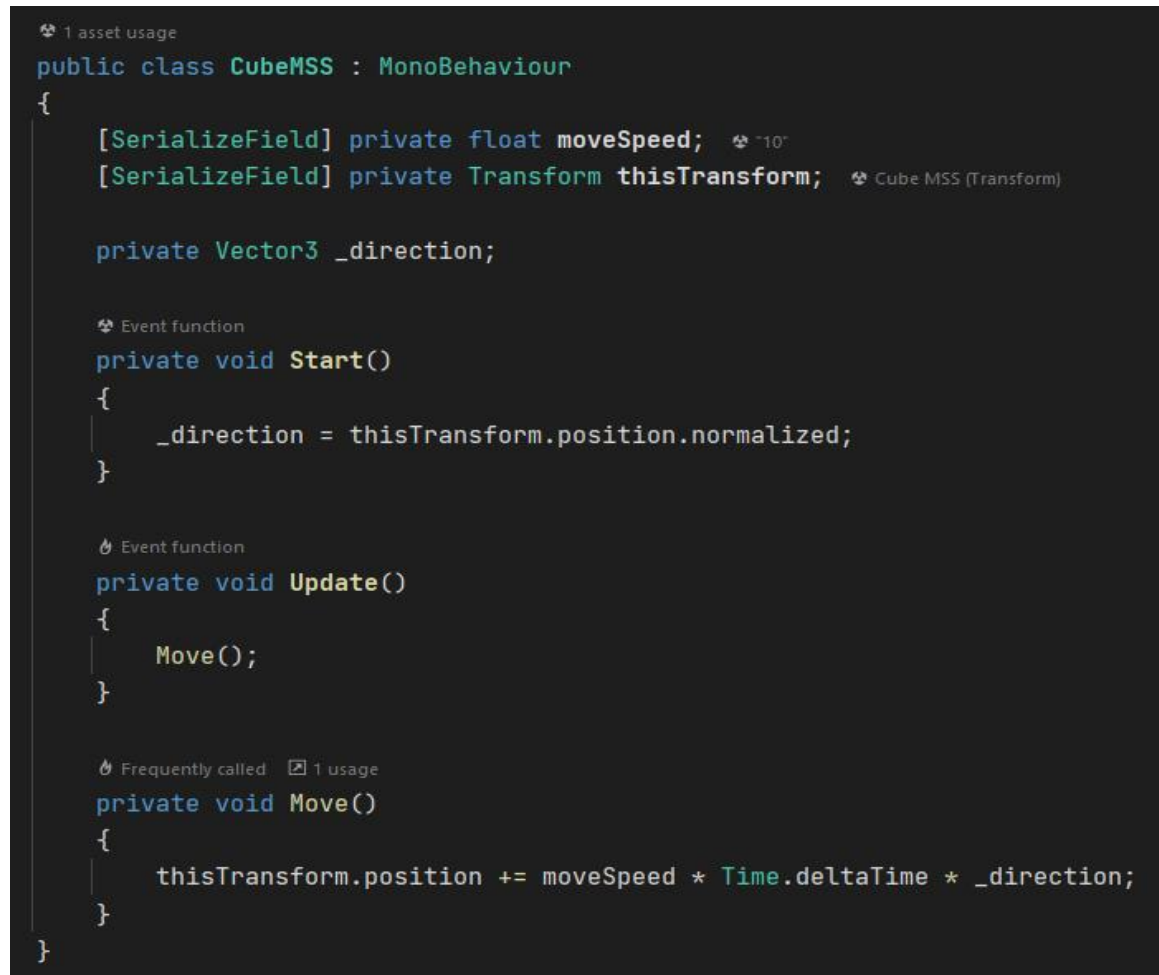
Figura 5.7 Implementación de los métodos *DestroyCubes()* y *CreateCubes()*

Es importante recalcar que el código de la clase *Generator* se puede utilizar para realizar las pruebas de las configuraciones de la POO (MSS, MSC, MCS y MCC), debido a que las diferencias entre estas configuraciones residen en el comportamiento de cada cubo, tal y como se puede ver a continuación.

5.1.2.2 Configuración MSS

La configuración MSS consiste en hacer que los cubos sean objetos que deriven de la clase *MonoBehaviour*, y crearlos a través de una clase *Generator*.

El componente que se añade a cada cubo en esta configuración se puede ver en la figura 5.8. Cada cubo dispone de una variable para guardar su velocidad de movimiento y de una referencia a su propio componente *Transform*, al que se accede para modificar la posición del cubo desde el método *Move()*.



```

1 asset usage
public class CubeMSS : MonoBehaviour
{
    [SerializeField] private float moveSpeed; 10
    [SerializeField] private Transform thisTransform; Cube MSS (Transform)

    private Vector3 _direction;

    Event function
    private void Start()
    {
        _direction = thisTransform.position.normalized;
    }

    Event function
    private void Update()
    {
        Move();
    }

    Frequently called 1 usage
    private void Move()
    {
        thisTransform.position += moveSpeed * Time.deltaTime * _direction;
    }
}

```

Figura 5.8 Lógica de los cubos en la configuración MSS

5.1.2.3 Configuración MSC

En el caso de la configuración MSC, el código del movimiento del cubo es muy similar, solo que en este caso al método *Move()* se le añade una línea de código para añadir el sobrecalentamiento inducido, explicado con anterioridad en este trabajo.

```
Frequently called 1 usage
private void Move()
{
    thisTransform.position += moveSpeed * Time.deltaTime * _direction;

    Overhead();
}

Frequently called 1 usage
private void Overhead()
{
    int x = 0;
    for (int i = 0; i < 100000; i++)
        x++;
}
```

Figura 5.9 Configuración del movimiento del cubo con sobrecalentamiento

5.1.2.4 Configuración MCS

Las configuraciones que recurren al uso de jobs para generar el código son un poco diferentes, ya que se hace necesario la programación de un job dentro del método *Update()*, tal y como se puede ver en la figura 5.10.

```
Event function
private void Update()
{
    CubeJob cubeJob = new CubeJob(
        _positionResult, _direction, thisTransform.position, Time.deltaTime, moveSpeed);

    _handle = cubeJob.Schedule();
}

Event function
private void LateUpdate()
{
    _handle.Complete();

    thisTransform.position = _positionResult[0];
}

Event function
private void OnDestroy()
{
    _positionResult.Dispose();
}
```

Figura 5.10 Código de la configuración MCS

Es importante recalcar en este caso que es necesario llamar al método *Dispose()* sobre *_positionResult*, que es una variable de tipo *NativeArray<>*. Este método se encarga de liberar la memoria utilizada por este *NativeArray* ya que el recolector de basura de C# no se encarga de hacerlo en este caso.

```
private Vector3 _direction;  
private NativeArray<Vector3> _positionResult;  
private JobHandle _handle;
```

Figura 5.11 Variables definidas en la configuración MCS

El código del job de movimiento del cubo se puede ver en la figura 5.12. Este job define un constructor al que se le pasan todos los datos necesarios para calcular el movimiento del cubo e implementa un método *Execute()*, desde el que se llama al método *Move()*, que se encarga de calcular la nueva posición del cubo y de guardar este resultado en la primera posición del *NativeArray* *_positionResult*, para que posteriormente podamos acceder a este valor cuando se complete el job.

```
Frequently called 1 usage
public CubeJob(NativeArray<Vector3> positionResult,
    Vector3 direction, Vector3 position, float deltaTime, float moveSpeed)
{
    _positionResult = positionResult;
    _direction = direction;
    _position = position;
    _deltaTime = deltaTime;
    _moveSpeed = moveSpeed;
}

Burst compiled code
public void Execute()
{
    Move();
}

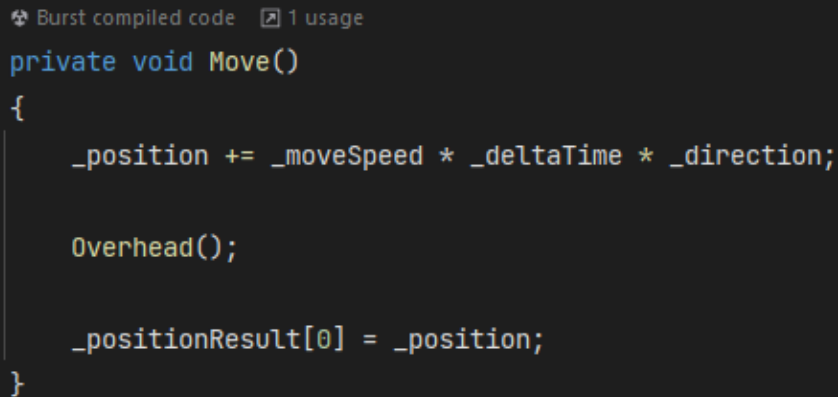
Burst compiled code 1 usage
private void Move()
{
    _position += _moveSpeed * _deltaTime * _direction;

    _positionResult[0] = _position;
}
```

Figura 5.12 Código del job sin sobrecalentamiento

5.1.2.5 Configuración MCC

Las configuraciones MCS y MCC se diferencian simplemente en el método *Move()* que definen, ya que en el caso de la configuración MCC se llamará al método *Overhead()* para añadir el sobrecalentamiento inducido al cálculo del movimiento del cubo, tal y como se puede ver en la figura 5.13.

A screenshot of a code editor showing a C++ method implementation. At the top, there are two icons: a lightning bolt and the text 'Burst compiled code', followed by a square icon and '1 usage'. The code is as follows:

```
private void Move()
{
    _position += _moveSpeed * _deltaTime * _direction;

    Overhead();

    _positionResult[0] = _position;
}
```

Figura 5.13 Método *Move()* con sobrecalentamiento inducido

5.1.3 Configuraciones de la POD

5.1.3.1 Código para generar los cubos con la POD

En estructura, el código necesario para crear un grupo de cubos utilizando el paradigma de la POD es muy similar en estructura al código que se ha explicado anteriormente utilizando el paradigma de la POO.

Inicialmente se necesita de un objeto *Generator* que se deberá convertir en una entidad a través del proceso de *baking*. Un ejemplo de esto se puede ver en la figura 5.14.

```

1 asset usage 2 usages More...
public class GeneratorAuthoring : MonoBehaviour
{
    public int initialAmountOfCubes; 1000
    public CubeType type; MCC
    public GameObject[] cubePrefabs; Serializable
}

DOTS
public class GeneratorBaker : Baker<GeneratorAuthoring>
{
    public override void Bake(GeneratorAuthoring authoring)
    {
        var entity = GetEntity(TransformUsageFlags.None);

        AddComponent(entity, new GeneratorComponent
        {
            initialAmountOfCubes = authoring.initialAmountOfCubes,
            cubePrefab = GetEntity(authoring.cubePrefabs[(int)authoring.type], TransformUsageFlags.Dynamic)
        });
    }
}

DOTS 17 usages
public struct GeneratorComponent : IComponentData
{
    public int initialAmountOfCubes;
    public Entity cubePrefab;
}

```

Figura 5.14 Código del proceso de *baking* de *GameObject* a *Entidad* de un objeto *Generator*

En este caso, y gracias a la comodidad que ofrece el baker, en el *GeneratorComponent* solo se necesita declarar el número inicial de cubos y la entidad del cubo que se va a generar. Este prefab se obtiene en base al tipo de configuración que se quiera probar, tal y como se hacía en el ejemplo de la POO, y se obtiene durante el proceso de *baking*.

Por lo tanto, este *GeneratorComponent* es independiente del tipo de configuración que se pretende probar. Por desgracia, no ocurre lo mismo con los sistemas de creación de cubos, ya que se hace necesario crear un sistema por cada tipo de configuración que se quiere probar.

Aun así, todos los sistemas de generación de cubos en la POD siguen exactamente la misma estructura de código, solo diferenciándose en el tipo de componentes a los que acceden.

Por lo tanto, mediante la explicación de uno de estos cuatro sistemas se estarían explicando realmente todos los sistemas de generación de cubos, pues son los mismos a excepción del tipo de componentes que utilizan.

Tal y como se puede ver en la figura 5.15, el *GeneratorSystem* implementa la interfaz *ISystem*. En el método *OnCreate*, tal y como se ha comentado en secciones anteriores, establece las dependencias que se tienen que cumplir para que este sistema sea ejecutado. Este paso es crucial, pues si no se tiene esto en cuenta se podrían estar ejecutando varios sistemas a la vez cuando varios de ellos deberían estar desactivados.

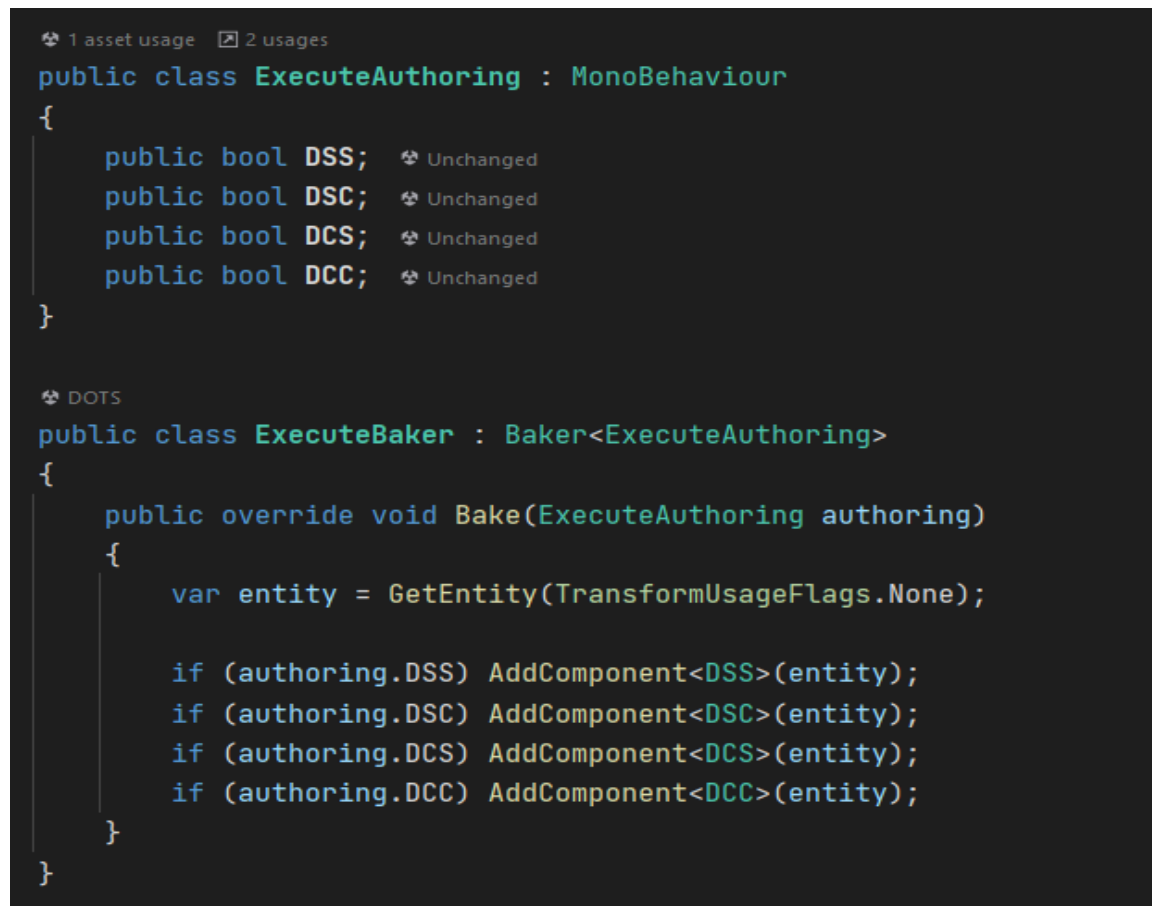
```
[BurstCompile]
⚙ DOTS
public partial struct DSS_GeneratorSystem : ISystem
{
    private uint _updateCounter;
    private int _amountOfCubes;

    [BurstCompile]
    public void OnCreate(ref SystemState state)
    {
        state.RequireForUpdate<BeginSimulationEntityCommandBufferSystem.Singleton>();
        state.RequireForUpdate<GeneratorComponent>();
        state.RequireForUpdate<DSS>();
    }
}
```

Figura 5.15 Código de inicialización de un sistema *Generator* para la configuración DSS

Para lograr este propósito, establecemos dos principales dependencias. Una de ellas es la dependencia al *GeneratorComponent*. Esta dependencia la van a establecer todos los sistemas de generación de cubos en la POD, pues necesitan obtener el número de cubos a crear y la entidad que se pretende instanciar.

La segunda principal dependencia es aquella que hace referencia al componente DSS. Este componente solo existirá en la escena si desde el editor de Unity hemos establecido que así sea. Esto se controla a través de la clase *ExecuteAuthoring*, la cual se puede ver en la figura 5.16.



```

1 asset usage 2 usages
public class ExecuteAuthoring : MonoBehaviour
{
    public bool DSS; 1 asset usage Unchanged
    public bool DSC; 1 asset usage Unchanged
    public bool DCS; 1 asset usage Unchanged
    public bool DCC; 1 asset usage Unchanged
}

DOTS
public class ExecuteBaker : Baker<ExecuteAuthoring>
{
    public override void Bake(ExecuteAuthoring authoring)
    {
        var entity = GetEntity(TransformUsageFlags.None);

        if (authoring.DSS) AddComponent<DSS>(entity);
        if (authoring.DSC) AddComponent<DSC>(entity);
        if (authoring.DCS) AddComponent<DCS>(entity);
        if (authoring.DCC) AddComponent<DCC>(entity);
    }
}

```

Figura 5.16 Código de control de las configuraciones en la POD

En el editor de Unity se debe añadir un *GameObject* a la subescena que se haya creado donde instanciar las entidades. A este *GameObject* se le añadirá el componente *ExecuteAuthoring*, y dependiendo de la variable booleana que hayamos marcado como true, se escogerá un componente u otro que añadir a la entidad generada por el proceso de *baking*.

Esto supone que, al haber establecido la dependencia en el *GeneratorSystem* hacia uno de estos cuatro componentes, solo en el caso de que se marque como true la variable booleana asociada al componente dependiente se va a ejecutar un sistema u otro.

Esta forma de controlar qué sistema se ejecuta en la POD es un poco más confusa que la forma de hacerlo en la POO, pero una vez se entiende bien es una forma eficaz y bastante útil a la hora de determinar qué sistemas se ejecutan en qué momento del flujo de eventos del juego. En un cierto momento del juego, si se produce un evento, se puede recurrir a eliminar o añadir un componente de una entidad, haciendo que todos los

sistemas que dependiesen de que este componente existiese en la subescena dejen de ejecutarse.

Por último, es necesario mostrar el código de destrucción y creación de cubos en sí. Esta lógica se puede observar en las figuras 5.17 y 5.18.

```
[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    if (!Input.GetKeyDown(KeyCode.Q))
        return;

    // Destroy the cubes

    var ecbSingleton = SystemAPI.GetSingleton<BeginSimulationEntityCommandBufferSystem.Singleton>();
    var ecb:EntityCommandBuffer = ecbSingleton.CreateCommandBuffer(state.WorldUnmanaged);

    foreach (var (cube:RefRO<DSS_CubeComponent>, entity) in
        SystemAPI.Query<RefRO<DSS_CubeComponent>>() // QueryEnumerable<RefRO<...>>
            .WithEntityAccess())
    {
        ecb.DestroyEntity(entity);
    }
}
```

Figura 5.17 Lógica de destrucción de cubos

Como en el caso de la POO, en la POD primero se realiza una comprobación para terminar si se presiona la letra Q o no. En caso de presionarla, lo primero que se necesita hacer es borrar los cubos que están presentes en la subescena.

Para lograr esto se puede recurrir a la creación de una consulta o *Query* (véase la figura 5.17). Esta consulta se encarga de buscar todas aquellas entidades que tengan un componente de tipo *DSS_CubeComponent*.

Esta consulta es una de las diferencias que existen entre configuraciones, pues la configuración DSS realizará la búsqueda de las entidades con el componente mencionado, pero la configuración DSC realizará la búsqueda de las entidades con el componente DSC.

Dentro del bucle se llamará entonces al método *DestroyEntity()* que provee el *CommandBuffer* que se ha obtenido previamente.

```

// Create the cubes

var generator = SystemAPI.GetSingleton<GeneratorComponent>();

_amountOfCubes += generator.initialAmountOfCubes;

var cubes:NativeArray<Entity> = state.EntityManager.Instantiate(
    generator.cubePrefab, _amountOfCubes, (AllocatorHandle) Allocator.Temp);

var random = Random.CreateFromIndex(_updateCounter++);

foreach (var cube:Entity in cubes)
{
    float3 cubePosition;
    do {
        cubePosition = random.NextFloat3(
            new float3(x:-0.1f, y:-0.1f, z:-0.1f), new float3(x:0.1f, y:0.1f, z:0.1f)); // float3
    }
    while (cubePosition.Equals(rhs:float3.zero));

    cubePosition = math.normalize(cubePosition);

    var transform:RefRW<LocalTransform> = SystemAPI.GetComponentRW<LocalTransform>(cube);
    transform.ValueRW.Position = cubePosition;

    var cubeComponent:RefRW<DSS_CubeComponent> = SystemAPI.GetComponentRW<DSS_CubeComponent>(cube);
    cubeComponent.ValueRW.direction = cubePosition;
}

```

Figura 5.18 Lógica de creación de cubos

En el caso del código de creación de los cubos en la POD, tal y como se puede ver en la figura 5.18, inicialmente se obtiene el *Singleton* del *GeneratorComponent*, algo que resulta muy conveniente, pero se hace necesario asegurarse de que en escena solo haya una entidad con un componente *GeneratorComponent* asociado. De lo contrario, la llamada *SystemAPI.GetSingleton<>()* lanzaría una excepción.

Seguidamente se actualiza el nuevo número de cubos a instanciar, y se instancian a partir del *EntityManager*. Después se inicializa una variable *random* en base a una semilla *updateCounter* que se va incrementando cada vez que el sistema se ejecute.

Finalmente se recorre el conjunto de cubos que hemos instanciado, en el que para cada cubo se calcula una posición aleatoria que se asigna al cubo y se actualiza posteriormente su dirección de movimiento.

5.1.3.2 Configuración DSS

La configuración DSS es la configuración más sencilla de implementar, pues solo consiste en mover los cubos que coincidan con los criterios de consulta en base a la dirección que lleven. Esto se puede observar en la figura 5.19.

```

[BurstCompile]
public partial struct DSS_MovementSystem : ISystem
{
    [BurstCompile]
    public void OnCreate(ref SystemState state)
    {
        state.RequireForUpdate<DSS>();
    }

    [BurstCompile]
    public void OnUpdate(ref SystemState state)
    {
        foreach (var (transform:RefRW<LocalTransform>, cube:RefRO<DSS_CubeComponent>) in
            SystemAPI.Query<RefRW<LocalTransform>, RefRO<DSS_CubeComponent>>())
        {
            transform.ValueRW.Position +=
                cube.ValueRO.direction * SystemAPI.Time.DeltaTime * cube.ValueRO.moveSpeed;
        }
    }
}

```

Figura 5.19 Código de la configuración DSS para el movimiento de cubos

En esta figura se puede observar que en el método *OnUpdate()* se realiza una consulta para buscar todas aquellas entidades que tengan un componente *LocalTransform* y un componente *DSS_CubeComponent*. Una vez se obtiene esta consulta se realiza el bucle de las entidades que coinciden con la búsqueda y se actualiza su posición.

5.1.3.3 Configuración DSC

En el caso de la configuración DSC, el código es similar al de la configuración DSS, solo que en este caso se le añaden el sobrecalentamiento inducido tras actualizar la posición del cubo, tal y como se puede ver en la figura 5.20.

```

[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    foreach (var (transform:RefRW<LocalTransform> , cube:RefRO<DSC_CubeComponent>) in
        SystemAPI.Query<RefRW<LocalTransform>, RefRO<DSC_CubeComponent>>())
    {
        transform.ValueRW.Position +=
            cube.ValueRO.direction * SystemAPI.Time.DeltaTime * cube.ValueRO.moveSpeed;

        // Overhead

        int x = 0;
        for (int i = 0; i < 100000; i++)
            x++;
    }
}

```

Figura 5.20 Código de la configuración DSC en el que se añaden el sobrecalentamiento

5.1.3.4 Configuración DCS

En el caso de las configuraciones con jobs, en el método *OnUpdate()* del sistema de movimiento de cubos se debe crear y programar el job, al que se le pasa el *DeltaTime* por parámetro del constructor (véase la figura 5.21).

```

[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    var job = new MovementJob
    {
        deltaTime = SystemAPI.Time.DeltaTime
    };
    job.Schedule();
}

```

Figura 5.21 Programación del Job de movimiento de cubos

Además, la implementación del job de movimiento de cubos se puede ver en la figura 5.22, al que se le pasan por parámetro dos variables, una de tipo *LocalTransform* (que será el componente sobre el que actualicemos la posición del cubo) y otra de tipo

DSS_CubeComponent (componente que nos dará la velocidad y dirección del cubo en concreto).

```
[BurstCompile]
DOTS 14 usages
public partial struct MovementJob : IJobEntity
{
    public float deltaTime;

    Burst compiled code 8 usages
    private void Execute(ref LocalTransform transform, in DCS_CubeComponent cube)
    {
        transform.Position += cube.direction * deltaTime * cube.moveSpeed;
    }
}
```

Figura 5.22 Código de implementación del Job de movimiento de cubos

5.1.3.5 Configuración DCC

La última configuración que se va a desarrollar siguiendo la programación orientada a datos es la DCC. Esta se diferencia de la anterior configuración en que en el método *Execute* del job añadimos el sobrecalentamiento inducido, tal y como se puede ver en la figura 5.23.

```
Burst compiled code 8 usages
private void Execute(ref LocalTransform transform, in DCC_CubeComponent cube)
{
    transform.Position += cube.direction * deltaTime * cube.moveSpeed;

    // Overhead

    int x = 0;
    for (int i = 0; i < 100000; i++)
        x++;
}
```

Figura 5.23 Método *Execute* del job desarrollado para la configuración DCC

Y tras explicar la implementación de esta última configuración se puede dar por sentado la explicación de la implementación del prototipo de *Cubos*. A continuación se detallará la implementación del prototipo *Coruscant*.

5.2 Implementación del prototipo *Coruscant*

Por comodidad, y como en el caso del prototipo *Cubos*, se ha decidido reutilizar el código de la clase *FpsCounter* explicado en la sección 5.1.1. Asimismo, se ha decidido seguir una implementación similar para la ejecución de eventos de este prototipo, de tal forma que podamos garantizar unas condiciones similares para evaluar de forma correcta las distintas configuraciones.

La principal diferencia a tener en cuenta entre el prototipo *Cubos* y el prototipo *Coruscant* es que en este último se pretenden analizar principalmente datos referentes a los tiempos de creación y destrucción de entidades, además de recoger también datos referentes a los FPS de ambas configuraciones.

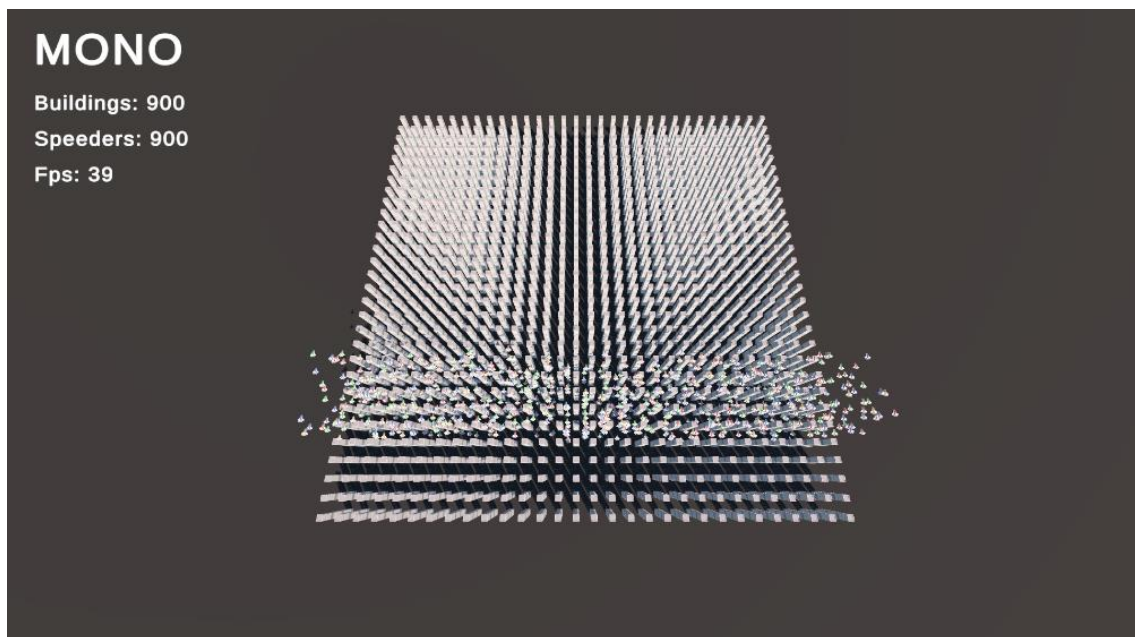


Figura 5.24 Captura de pantalla de una ejecución de la configuración MONO

Debido a los distintos objetivos que se quieren conseguir con el prototipo de *Coruscant*, se han decidido crear solo dos configuraciones, una configuración MONO y una configuración DOTS, y para cada una de estas configuraciones se medirán 5 variables:

- Tiempo que se tarda en crear X número de edificios (milisegundos).
- Tiempo que se tarda en crear X número de naves (milisegundos).
- Tiempo que se tarda en destruir X número de edificios (milisegundos).
- Tiempo que se tarda en destruir X número de naves (milisegundos).
- Media de imágenes por segundo durante una ejecución (fps).

5.2.1 Configuración MONO

La configuración MONO está compuesta principalmente por tres partes: el código para generar y destruir edificios, el código para generar y destruir naves y el código para mover las naves.

El primero de ellos está compuesto por dos métodos principales (véanse las figuras 5.25 y 5.26): un método de creación de edificios y otro de destrucción de edificios. Estos códigos son sencillos, y lo más interesante a comentar a este respecto es la forma que se ha escogido para obtener los tiempos que tardan dichos métodos en ser ejecutados.

```

Frequently called 1 usage
private void GenerateBuildings()
{
    DestroyBuildings();

    DateTime before = DateTime.Now;

    int buildingsSpawned = BuildingsAmount;

    float x = _buildingsAmount / 2f;

    for (float i = -x; i < x; i++)
    {
        for (float j = -x; j < x; j++)
        {
            Instantiate(buildingPrefab,
                position: new Vector3(x * i, 0f, x * j) * positionOffset,
                Quaternion.identity,
                _buildingsParent);
        }
    }

    _buildingsAmount += 5;

    DateTime after = DateTime.Now;
    TimeSpan duration = after.Subtract(before);
    CsvLoggerManager.Instance.LogData(logEntry: $"{duration.TotalMilliseconds};{buildingsSpawned}", LogType.MonoBuildingsCreation);
    Debug.Log(message: $"It took {duration.TotalMilliseconds} ms to create {buildingsSpawned} buildings.");
}

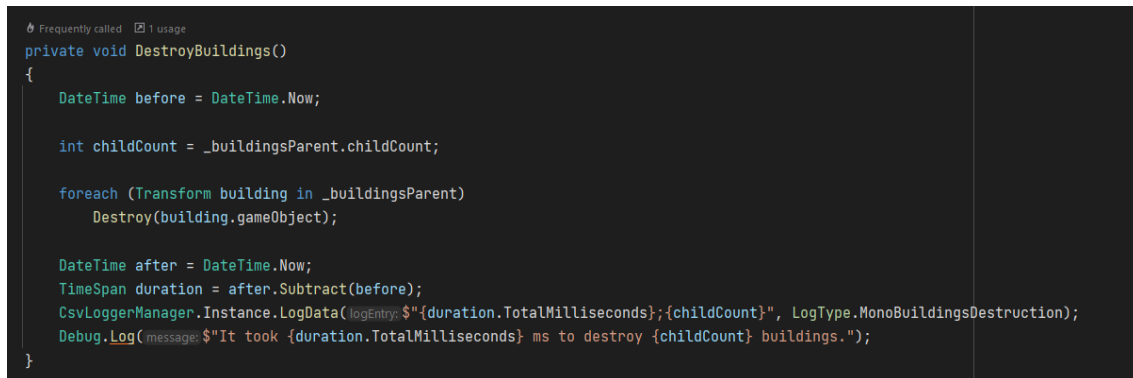
```

Figura 5.25 Código de generación de edificios

Para este propósito al inicio del método se obtiene el tiempo actual y se guarda en una variable *before* de tipo *DateTime*. Al terminar de ejecutarse el método se vuelve a guardar el tiempo actual en una variable *after*, y estas dos variables se restan para obtener la diferencia.

Posteriormente, tanto en el caso del código de construcción como en el de destrucción de edificios, se guardan (a través de un *CsvLoggerManager*) los datos del tiempo en milisegundos asociados a la cantidad de edificios que han sido creados o destruidos.

De esta forma, cuando se cierra la aplicación se cogerán todos los datos obtenidos de las ejecuciones de la configuración y se guardarán en un archivo .csv para su posterior análisis y evaluación.



```

Frequently called 1 usage
private void DestroyBuildings()
{
    DateTime before = DateTime.Now;

    int childCount = _buildingsParent.childCount;

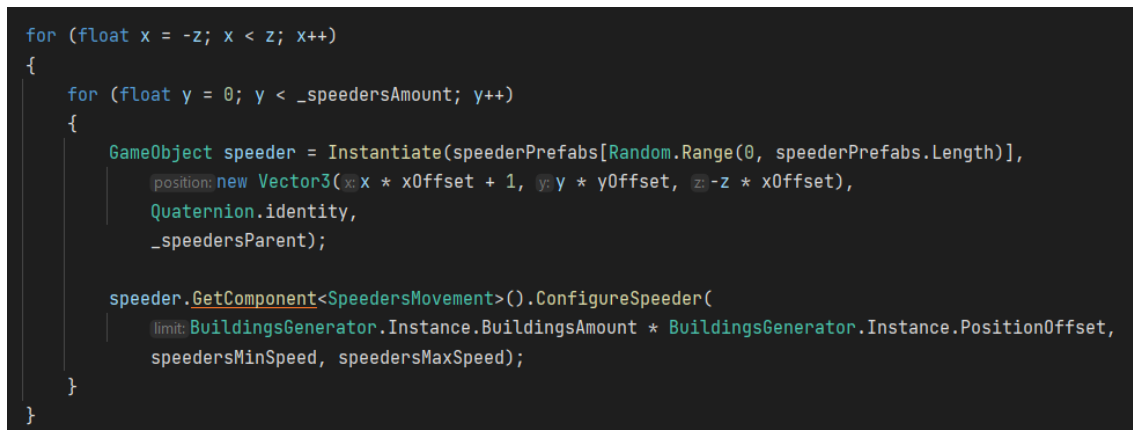
    foreach (Transform building in _buildingsParent)
        Destroy(building.gameObject);

    DateTime after = DateTime.Now;
    TimeSpan duration = after.Subtract(before);
    CsvLoggerManager.Instance.LogData(logEntry: $"{duration.TotalMilliseconds};{childCount}", LogType.MonoBuildingsDestruction);
    Debug.Log(message: $"It took {duration.TotalMilliseconds} ms to destroy {childCount} buildings.");
}

```

Figura 5.26 Código de destrucción de edificios

El código de creación y destrucción de naves es bastante similar al código de creación y destrucción de edificios, con una pequeña salvedad, y es que en el código de naves (tal y como se puede ver en la figura 5.27), dentro del bucle de creación de naves, se llama al método *ConfigureSpeeder()* sobre la nave que se crea para pasarle el límite de movimiento y las velocidades mínima y máxima.



```

for (float x = -z; x < z; x++)
{
    for (float y = 0; y < _speedersAmount; y++)
    {
        GameObject speeder = Instantiate(speederPrefabs[Random.Range(0, speederPrefabs.Length)],
            position: new Vector3(x * xOffset + 1, y * yOffset, z - z * xOffset),
            Quaternion.identity,
            _speedersParent);

        speeder.GetComponent<SpeedersMovement>().ConfigureSpeeder(
            limit: BuildingsGenerator.Instance.BuildingsAmount * BuildingsGenerator.Instance.PositionOffset,
            speedersMinSpeed, speedersMaxSpeed);
    }
}

```

Figura 5.27 Bucle de creación de naves

Asimismo, el código de movimiento de las naves es sencillo, ya que consiste principalmente en un método para calcular la velocidad y en un método para mover a la nave, tal y como se puede ver en la figura 5.28.


```

Frequently called 2 usages
public void ConfigureSpeeder(float minSpeedLimit, float maxSpeedLimit)
{
    _speed = Random.Range(minSpeedLimit, maxSpeedLimit);
}

Event function
private void Update()
{
    _thisTransform.position += _speed * Time.deltaTime * Vector3.forward;
}

```

Figura 5.28 Código de movimiento de las naves

5.2.2 Configuración DOTS

Para desarrollar el prototipo siguiendo la programación orientada a datos en Unity, se han creado una clase para los edificios (*BuildingAuthoring*), otra clase para el generador de edificios (*BuildingsGeneratorAuthoring*), otra clase para las naves (*SpeedersAuthoring*), otra para el generador de naves (*SpeedersGeneratorAuthoring*) y una quinta clase para un objeto de configuración (*ConfigAuthoring*).

Cada uno de estos archivos contiene una clase que deriva de *MonoBehaviour* (que será el componente que se añada a los objetos y prefabs desde el editor de Unity) y sus correspondientes *Baker* y *Component*. Este proceso se ha decidido no explicarse en detalle debido a su similitud con el prototipo *Cubos*.

La mayor diferencia que se puede encontrar entre el código de la POD de este prototipo frente al código de la POD del prototipo de *Cubos* reside en la imposibilidad de utilizar el compilador *Burst*, lo que va a provocar unos resultados un poco peores para esta configuración de lo que resultarían en realidad. Esto se debe al código que se necesita implementar para la obtención de la duración de los métodos de creación y destrucción de edificios y naves. Este código se puede ver en la figura 5.29.

```

DateTime after = DateTime.Now;
TimeSpan duration = after.Subtract(before);
CsvLoggerManager.Instance.LogData(
    logEntry: $"{duration.TotalMilliseconds};{_buildingsAmount * _buildingsAmount}", LogType.DotsBuildingsCreation);
Debug.Log(
    message: $"It took {duration.TotalMilliseconds} ms to create {_buildingsAmount * _buildingsAmount} buildings.");

```

Figura 5.29 Código de obtención de la duración de la creación y destrucción de edificios

Este código requiere del uso de la clase *DateTime* para la obtención de los tiempos, además del uso de la clase *CsvLoggerManager*, clase que accede a otras librerías que no pueden ser compiladas por el compilador *Burst*.

Por último, y ya para finalizar esta sección, es interesante explicar cómo funciona la implementación del movimiento de las naves. El método *OnUpdate()* del sistema que se encarga de esto es relativamente sencillo y se puede ver en la figura 5.30.

```
[BurstCompile]
public void OnUpdate(ref SystemState state)
{
    ConfigComponent configComponent = SystemAPI.GetSingleton<ConfigComponent>();

    var rand = new Random(seed: 123);

    foreach (var playerTransform :RefRW<LocalTransform> in
        SystemAPI.Query<RefRW<LocalTransform>>() // QueryEnumerable<RefRW<...>>
            .WithAll<SpeederComponent>())
    {
        var speed :float = rand.NextFloat(configComponent.speedersMinSpeed, configComponent.speedersMaxSpeed);

        var input :float3 = new float3(x: 0f, y: 0f, z: 1f) * SystemAPI.Time.DeltaTime * speed;

        var newPos :float3 = playerTransform.ValueR0.Position + input;

        playerTransform.ValueRW.Position = newPos;
    }
}
```

Figura 5.30 Código del movimiento de las naves en la configuración DOTS

Este sistema realiza una consulta buscando todas las naves que haya en la subescena y mueve a la nave en base a la velocidad calculada.

Capítulo 6

Evaluación

Una vez que se han diseñado e implementado cada uno de los prototipos, es hora de ejecutarlos y someterlos a juicio, y los resultados de dichas pruebas son los que se van a mostrar y explicar en esta sección.

6.1 Evaluación del prototipo *Cubos*

Para poder obtener resultados que aporten de verdad información útil es necesario que las diferentes evaluaciones que se pueden hacer de las configuraciones se hagan por parejas, en las que solo una de las 3 variables cambie entre ambas configuraciones a evaluar.

Una vez hemos ejecutado cada una de las configuraciones del prototipo *Cubos*, los resultados se han guardado en un archivo *Cubos.csv*, que contendrá una columna para cada una de las configuraciones del prototipo. Si abrimos este archivo en un *dataframe* de pandas, el contenido del mismo es el mostrado en la figura 6.1.

	CUBES	MSS	DSS	MCS	DCS	MSC	DSC	MCC	DCC
0	1000	102	177	79	123	60	111	85	112
1	2000	72	180	51	132	15	104	57	113
2	3000	58	176	39	134	11	112	43	110
3	4000	47	177	32	128	9	110	33	106
4	5000	40	167	26	119	6	109	28	100
5	6000	35	157	21	112	5	101	23	91
6	7000	31	149	19	111	5	93	20	93
7	8000	27	152	17	107	5	95	18	60
8	9000	23	139	14	102	4	93	16	35
9	10000	21	132	13	96	4	90	14	33
10	11000	19	122	11	38	4	87	12	32
11	12000	18	125	10	80	3	85	11	31
12	13000	16	125	9	37	3	82	10	30
13	14000	14	122	8	33	3	75	9	55
14	15000	13	118	8	30	3	63	9	31
15	16000	13	115	7	70	3	60	8	43
16	17000	11	110	7	32	2	28	8	24
17	18000	11	108	6	27	2	25	7	24
18	19000	10	102	6	70	2	24	7	24
19	20000	10	92	5	27	2	23	6	41

Figura 6.1 Datos de FPS de las 8 configuraciones del prototipo Cubos

A primera vista ya se pueden observar ciertas tendencias generales, pero veamos ahora comparaciones concretas entre configuraciones.

6.1.1 MSS vs DSS

Con esta comparación se pretende investigar si el caso de uso de un videojuego que no utilice ni jobs ni tenga la necesidad de realizar operaciones costosas a menudo (es decir, sin sobrecalentamiento) es mejor desarrollarlo siguiendo la POO o la POD. Para ello nos fijaremos en las 3 primeras columnas mostradas en la figura 6.1, es decir, las columnas del número de cubos y de fps de las configuraciones MSS y DSS.

Antes de mostrar la gráfica comparativa es interesante llamar al método *describe()* sobre el dataframe, cuyo resultado se puede ver en la figura 6.2.

	CUBES	MSS	DSS	MCS	DCS	MSC	DSC	MCC	DCC
count	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000
mean	10500.000000	29.550000	137.250000	19.400000	80.400000	7.550000	78.500000	21.200000	59.400000
std	5916.079783	24.169304	27.786545	18.661528	40.515624	12.787638	31.00679	20.132718	34.834043
min	1000.000000	10.000000	92.000000	5.000000	27.000000	2.000000	23.000000	6.000000	24.000000
25%	5750.000000	13.000000	117.250000	7.750000	36.000000	3.000000	62.250000	8.750000	31.000000
50%	10500.000000	20.000000	128.500000	12.000000	88.000000	4.000000	88.500000	13.000000	42.000000
75%	15250.000000	36.250000	159.500000	22.250000	113.750000	5.250000	101.750000	24.250000	94.750000
max	20000.000000	102.000000	180.000000	79.000000	134.000000	60.000000	112.000000	85.000000	113.000000

Figura 6.2 Resultado de realizar una operación *describe()* sobre el dataframe de fps

Antes si quiera de mirar las gráficas resaltan varios resultados que es preciso mencionar. La media de fps de la configuración MSS es de 29.55 fps, mientras que la media de fps de la configuración DSS es de 137.25 fps. Esta diferencia entre las medias es una diferencia muy grande que indica que la configuración de la POD es muy superior en términos de rendimiento que la configuración de la POO.

Este hecho se ve acentuado por el valor más pequeño que podemos encontrar en ambas columnas, indicado por el campo min, que en el caso de la POO es de 10 fps y en el caso de la POD es de 92 fps. Este valor es el valor que, como se puede ver en la figura 6.1, se obtiene al tener 20.000 cubos en pantalla moviéndose. Veamos ahora una gráfica comparativa entre ambas configuraciones (figura 6.3).

MSS vs DSS (Comparativa de FPS)

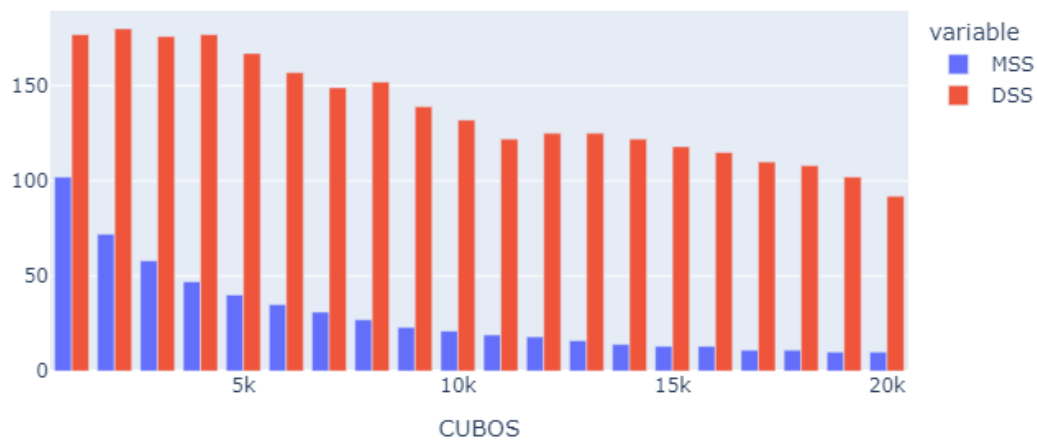


Figura 6.3 Gráfica de barras de las configuraciones MSS y DSS

La gráfica es clara, no es muy difícil sacar conclusiones. Para una configuración en la que un videojuego no implementa ni jobs ni algoritmos complejos y requiere de un gran número de entidades en pantalla, la POD y el uso de Unity DOTS es una solución mucho más eficiente que el uso de la POO con *GameObjects*.

6.1.2 MSC vs DSC

Ahora bien, ¿se conseguirá esa misma mejoría de rendimiento en el caso de añadir un sobrecalentamiento? Pues eso es precisamente lo que se pretende analizar comparando las configuraciones MSC y DSC.

Si nos fijamos en los resultados obtenidos tras llamar al método `describe()` sobre el dataframe, se puede observar que la media de la configuración DSC es 78.50 fps, mientras que la media de fps de la configuración MSC es 7.55 fps, es decir, existe una diferencia abismal entre las medias de cada una de las configuraciones. Veamos ahora una gráfica comparativa entre ambas configuraciones (figura 6.4).

MSC vs DSC (Comparativa de FPS)

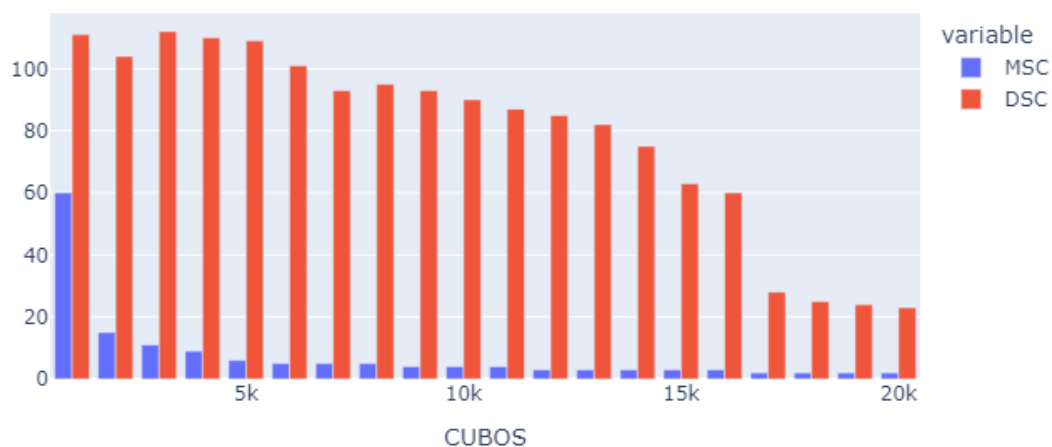


Figura 6.4 Gráfica de barras de las configuraciones MSC y DSC

Después de observar la gráfica queda aún más claro: en el caso de desarrollar un videojuego que requiera tener muchas entidades en pantalla con una lógica compleja y costosa (como podría ser un algoritmo de pathfinding), Unity DOTS y la programación orientada a datos son una mejor solución que la programación orientada a objetos que hay por defecto en Unity.

De hecho, se puede decir que el comportamiento de la configuración MSC es nefasto, no alcanzando si quiera los 20 fps con dos mil cubos en pantalla.

6.1.3 MCS vs DCS

Llegados a este punto puede estar surgiendo una pregunta: ¿cómo influirá la paralelización del código a través del Job System de C#? Eso es lo que se pretende analizar con las dos siguientes comparativas.

Con la comparativa de las configuraciones MCS y DCS se pretende observar si en el caso de paralelizar el código en diferentes hilos, la POO es superior en términos de rendimiento a la POD, o si por el contrario esta última es una mejor solución en dicho caso.

Volviendo de nuevo a la figura 6.2, se puede observar como la media de fps de la configuración DCS sigue siendo muy superior a la media de fps de la configuración MCS (80.4 fps frente a 19.4 fps). Este efecto se puede ver visualmente con la figura 6.5.

MCS vs DCS (Comparativa de FPS)

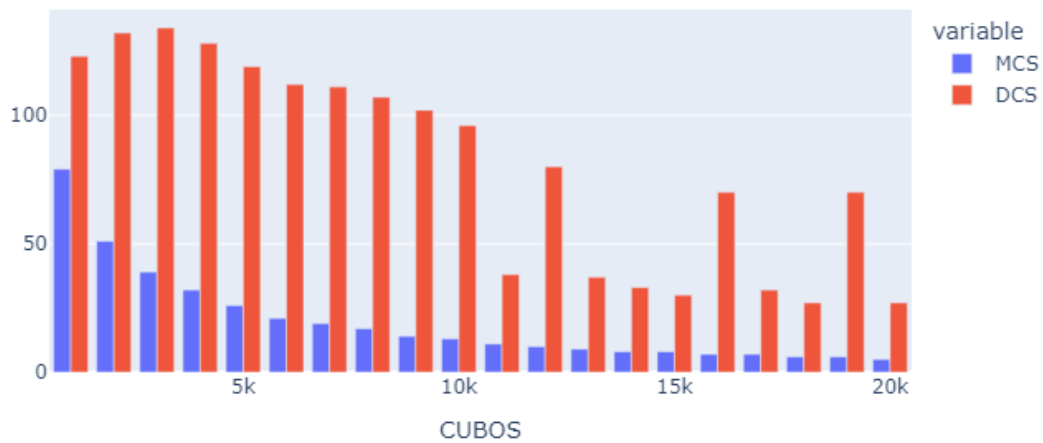


Figura 6.5 Gráfica comparativa de fps entre las configuraciones MCS y DCS

De nuevo se puede observar en la gráfica unos resultados bastante similares a los obtenidos en las comparaciones anteriores. Aun así, es interesante mencionar en este punto que, a partir de los 10.000 cubos, la configuración de la POD parece tener un comportamiento errático y poco previsible, alcanzando valores de fps bastante más bajos.

El motivo más probable por el que pueda estar sucediendo esto es el poco tiempo que lleva Unity DOTS publicado y, por consiguiente, su inestabilidad. Estas librerías aún están en una fase muy temprana de su desarrollo, aun cuando ya está publicada la versión 1.0 de todas ellas. De hecho, y como veremos en la última sección de este trabajo, al ser Unity DOTS un *framework* en desarrollo, aún quedan mucha funcionalidad pendiente de ser implementada, un hecho que todo desarrollador que quiera utilizar esta tecnología deberá tener en cuenta.

6.1.4 MCC vs DCC

La comparativa MCC vs DCC pretende analizar si, en el caso de un videojuego que requiere de un coste computacional alto que haya sido paralelizado con el Job System, sería mejor solución utilizar el paradigma de la POD o de la POO.

Volviendo a la figura 6.2, la configuración de la POO (MCC) obtiene una media de 21.2 fps, mientras que la configuración de la POD (DCC) obtiene una media de 59.4 fps.

De nuevo se observa una clara diferencia a nivel de rendimiento entre ambas configuraciones.

Este efecto se puede observar mejor a través de la gráfica mostrada en la figura 6.6, en la que se puede apreciar una clara diferencia de FPS entre ambas configuraciones. Aun así, de nuevo, se observa que el rendimiento de la POD es un poco errático, mientras que el rendimiento observado de la POO sigue una distribución logarítmica, mucho más predecible.

MCC vs DCC (Comparativa de FPS)

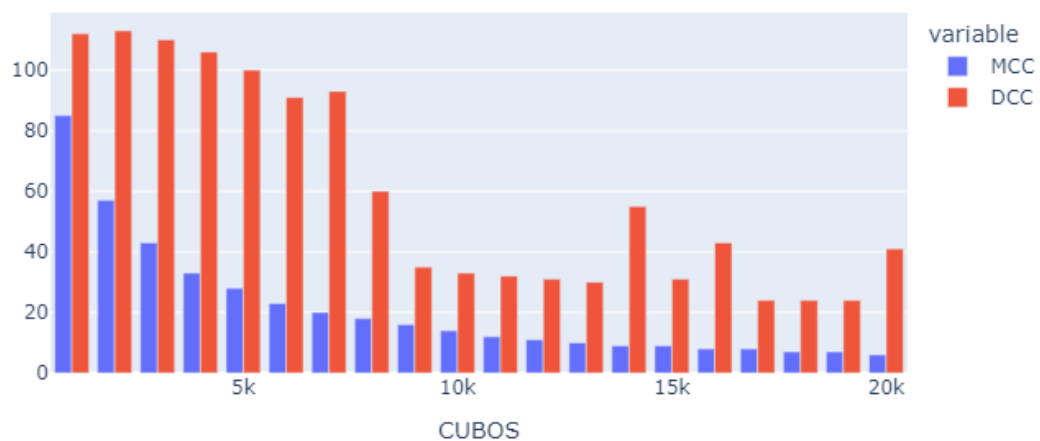


Figura 6.6 Gráfica comparativa de fps entre las configuraciones MCC y DCC

6.1.5 Otras comparativas interesantes

Además de comparar configuraciones desarrolladas con diferentes paradigmas, podemos evaluar el rendimiento de configuraciones del mismo paradigma, pero con una variable diferente en cada caso.

6.1.5.1 MSS vs MCS

Con la comparativa entre MSS y MCS se pretende observar si la paralelización de un código siguiendo el paradigma de la programación orientada a objetos mejora en el caso de no tener una lógica compleja que calcular cada frame del juego.

Tal y como se puede observar en la figura 6.7, la paralelización del código no solo no mejora el rendimiento, sino que empeora levemente. Esto, tal y como se ha comentado en secciones anteriores de este documento, se debe a que el coste de crear

en memoria un job y ejecutarlo es superior en tiempo al tiempo que requiere realizar la sencilla operación que se realiza en esta configuración.

MSS vs MCS (Comparativa de FPS)

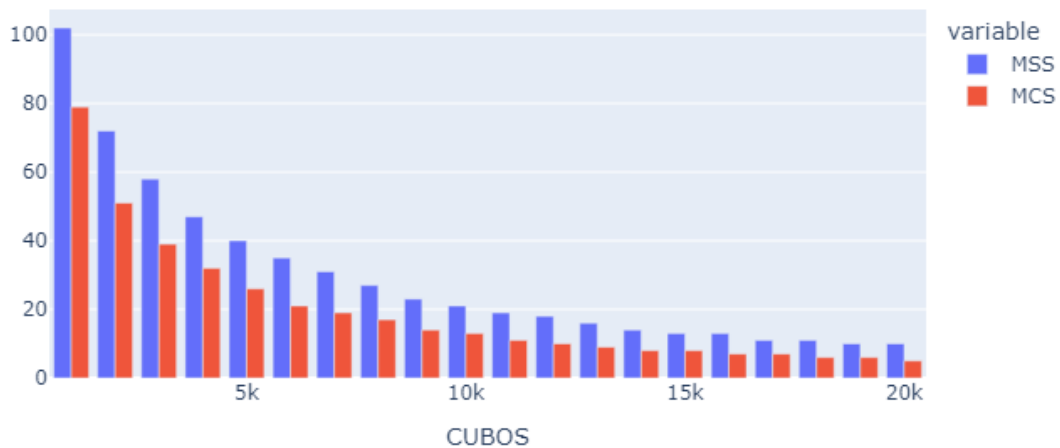


Figura 6.7 Gráfica comparativa de fps entre las configuraciones MSS y MCS

6.1.5.2 MSC vs MCC

Con la comparativa entre las configuraciones MSC y MCC se pretende analizar si mejorará el rendimiento la paralelización de un código costoso, lo que se puede ver en la figura 6.8, figura que muestra una gráfica de la comparativa de dichas configuraciones.

Figura 6.8 Gráfica comparativa de fps entre las configuraciones MSC y MCC

Es interesante mencionar como, aun teniendo un declive a nivel de rendimiento en ambas configuraciones bastante importante, la paralelización del código si mejora sustancialmente el rendimiento del prototipo, llegando a poder haber alrededor de 5.000 cubos en pantalla con una media de unos 30fps.

6.1.5.3 DSS vs DCS

En el caso de la POO (sección 6.1.6.1) se ha podido observar como la paralelización del código no mejoraba el rendimiento del prototipo. ¿Mejorará la paralelización del código el rendimiento en el caso de la POD? Para dar respuesta a esta pregunta se puede ver la figura 6.9, que muestra una comparación entre las configuraciones DSS y DCS.

DSS vs DCS (Comparativa de FPS)

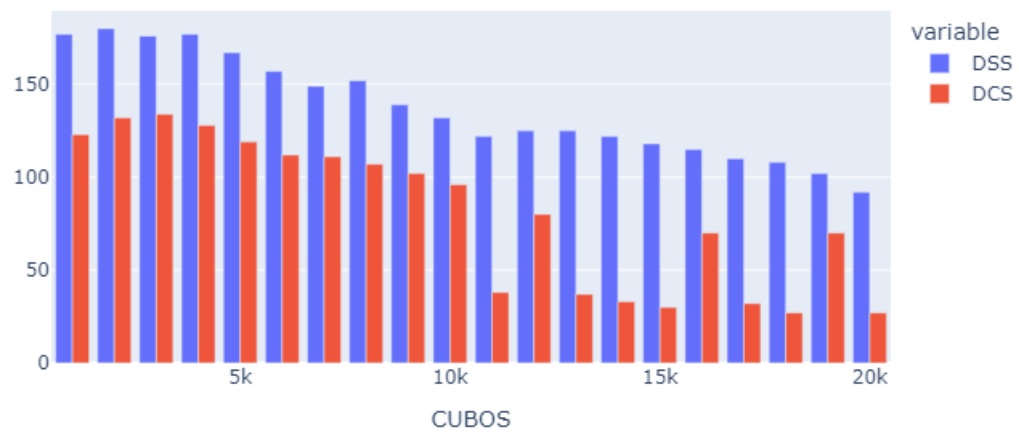


Figura 6.9 Gráfica comparativa de fps entre las configuraciones DSS y DCS

Ya a primera vista se observa cómo se obtienen unos resultados relativamente similares a los obtenidos en el caso de la POO, ya que la paralelización del código en este caso tampoco mejora el rendimiento del prototipo, siendo el deterioro incluso mayor que en el caso de la POO.

6.1.5.4 DSC vs DCC

Por último, es interesante analizar si la paralelización de un código desarrollado con la POD mejora el rendimiento en caso de que el código sea costoso de ejecutar. Esta comparativa se puede ver con la figura 6.10, que muestra una gráfica comparativa entre las configuraciones DSC y DCC.

DSC vs DCC (Comparativa de FPS)

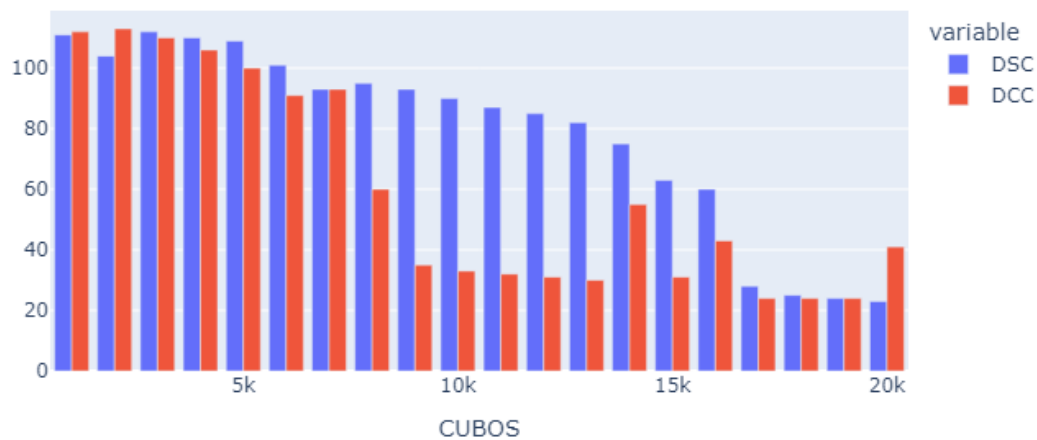


Figura 6.10 Gráfica comparativa de fps entre las configuraciones DSC y DCC

Esta gráfica es interesante, pues en el caso de desarrollar un código bajo la POD que tiene un gran coste computacional, la paralelización del código no parece mejorar el rendimiento del mismo, algo que no era para nada de esperar.

El motivo más plausible que puede explicar este fenómeno es el uso del compilador Burst en la configuración de la POD. El código de sobrecalentamiento, al ser simplemente un bucle for con una suma, puede estar siendo compilado de tal forma que no supone un coste considerable como si lo hacía en el caso del prototipo de la POO.

Otro motivo posible para explicar este efecto puede ser, de nuevo, el impredecible comportamiento que a veces tiene la POD en Unity debido a su actual estado en desarrollo, tal y como se ha mencionado en secciones anteriores.

6.2 Evaluación del prototipo de *Coruscant*

La evaluación del prototipo de *Coruscant*, como se ha comentado anteriormente, consistirá en la comparación de los tiempos de creación y destrucción de naves y edificios, así como la media de los FPS obtenidos en ambas configuraciones.

Estos datos, una vez obtenidos y ejecutada la configuración pertinente, se han decidido guardar en varios archivos .csv, cada uno de los cuales representa las diferentes configuraciones a comparar.

6.2.1 Comparativa de los FPS de ambas configuraciones

Veamos primero la comparativa entre los fps de ambos prototipos, para tener un primer acercamiento al rendimiento de ambos. Estos datos se pueden observar en la figura 6.11.

	Entidades	Mono Fps	Dots Fps
0	100	229	243
1	225	187	176
2	400	125	139
3	625	89	114
4	900	62	82
5	1225	47	63
6	1600	37	53
7	2025	28	45
8	2500	22	40
9	3025	18	36
10	3600	15	32
11	4225	12	28
12	4900	10	24
13	5625	8	22
14	6400	7	19
15	7225	7	17
16	8100	5	16
17	9025	5	16
18	10000	4	15

Figura 6.11 Datos de los FPS en la POO y la POD

Solo con ver estos datos ya se puede uno hacer a la idea de que, de nuevo, y como pasaba con las configuraciones del prototipo *Cubos*, la configuración desarrollada con el paradigma de la POD disfruta de un mayor rendimiento medido con FPS.

Si se mira la gráfica de barras mostrada en la figura 6.12, se podrá observar cómo, tanto en el caso del prototipo creado con MONO como el prototipo creado con DOTS siguen una distribución logarítmica, pues la propia cantidad de entidades en pantalla sigue esta distribución, pero en el caso de DOTS, este prototipo obtiene mejores resultados en casi todos los casos, es decir, independientemente del número de entidades en pantalla, el prototipo con DOTS obtiene un mejor rendimiento que el prototipo con MONO.

Comparativa de FPS

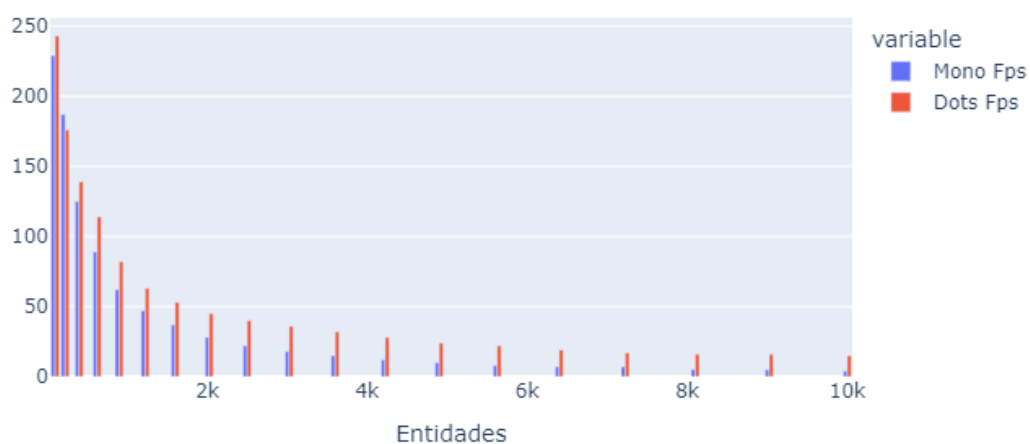


Figura 6.12 Gráfico de barras de los FPS de MONO y DOTS

6.2.2 Comparativa de los tiempos de creación de edificios

¿Mostrarán los datos de los tiempos de creación y destrucción el mismo comportamiento que tienen los datos de FPS, o existirán variaciones importantes? El conjunto de datos obtenido tras realizar las pruebas que se puede ver en la figura 6.13 da una primera respuesta a esta pregunta.

	Entidades	Tiempo Mono	Tiempo Dots
0	100	2.99	0.00
1	225	4.99	2.03
2	400	8.98	3.02
3	625	13.96	4.99
4	900	20.94	6.02
5	1225	30.92	8.98
6	1600	37.90	11.97
7	2025	49.60	16.01
8	2500	62.94	21.97
9	3025	78.50	27.95
10	3600	95.16	31.56
11	4225	112.79	39.43
12	4900	128.51	44.88
13	5625	148.36	55.95
14	6400	165.45	62.74
15	7225	219.41	68.82
16	8100	213.43	77.38
17	9025	253.91	88.25
18	10000	258.55	94.75

Figura 6.13 Datos de los tiempos de creación de edificios

De nuevo se puede observar como el tiempo que tarda MONO en crear un conjunto concreto de edificios aumenta de forma más rápida que el tiempo que tarda DOTS. Este hecho se ve más claro si se le echa un vistazo al resultado de llamar al método *describe()* sobre el conjunto de datos, tal y como se puede observar en la figura 6.14.

	Entidades	Tiempo Mono	Tiempo Dots
count	19.000000	19.000000	19.000000
mean	3775.000000	100.383684	35.089474
std	3170.206355	87.226106	31.221004
min	100.000000	2.990000	0.000000
25%	1062.500000	25.930000	7.500000
50%	3025.000000	78.500000	27.950000
75%	6012.500000	156.905000	59.345000
max	10000.000000	258.550000	94.750000

Figura 6.14 Resultado de llamar al método *describe()* sobre el conjunto de datos

Tanto la media como la desviación estándar son mucho más grandes para la configuración MONO que para la configuración DOTS, lo que indica unos tiempos mayores para MONO y también una mayor variabilidad en los mismos. Además, el valor más alto de MONO es de casi 2.8 veces mayor que el valor más alto de DOTS. Y este efecto se ve más directamente en la gráfico de barras que se muestra en la figura 6.15.

Comparativa del tiempo de creación de edificios

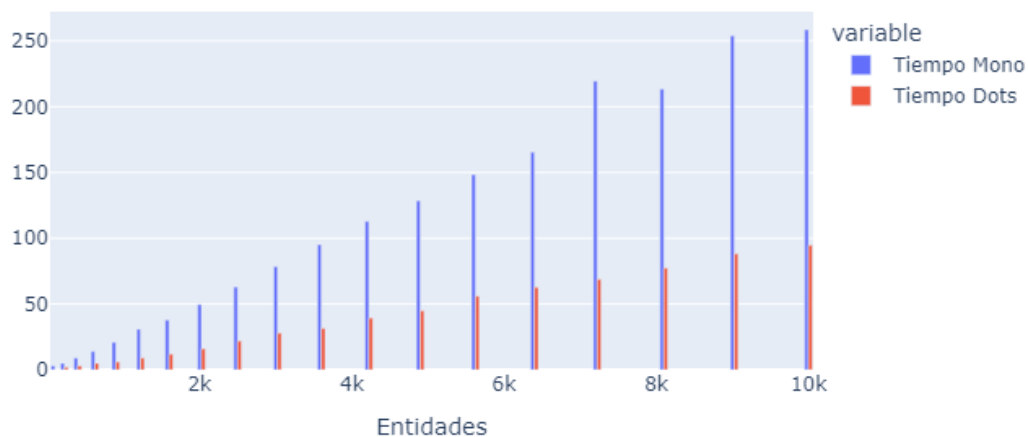


Figura 6.15 Gráfica de los tiempos de creación de edificios

En esta gráfica se puede ver claramente como el tiempo de creación de edificios del prototipo desarrollado con MONO aumenta más rápidamente que el tiempo de creación de edificios del prototipo desarrollado con DOTS.

6.2.3 Comparativa de los tiempos de destrucción de edificios

¿Se podrá observar la misma tendencia para los tiempos de destrucción que para los tiempos de creación? Pues para saber esto hay simplemente que echar un vistazo a la figura 6.16, que contiene los datos obtenidos tras la ejecución de ambas configuraciones.

	Entidades	Tiempo Mono	Tiempo Dots
0	0	0.0000	0.000743
1	100	0.9970	0.000050
2	225	0.9998	0.000051
3	400	0.9978	0.000069
4	625	0.9974	0.000156
5	900	19.9480	0.000147
6	1225	19.9490	0.000141
7	1600	25.7340	0.000179
8	2025	29.9230	0.000244
9	2500	32.9840	0.000231
10	3025	40.1110	0.000313
11	3600	53.5810	0.000332
12	4225	60.1680	0.001205
13	4900	7.0590	0.000526
14	5625	89.8090	0.000462
15	6400	109.6810	0.000530
16	7225	119.6610	0.000603
17	8100	13.0630	0.000656
18	9025	150.5430	0.000748

Figura 6.16 Datos de los tiempos de destrucción de edificios

Nada más ver la imagen, se puede dar uno cuenta de que los datos de la columna de la configuración de DOTS parecen ser erróneos, pues no parece creíble que la destrucción de miles de entidades suponga un coste de tiempo tan irrisorio, pero tal y como se ha comentado en secciones anteriores, debido a la forma que se guardan en memoria las entidades y sus compontes, el proceso de destrucción es increíblemente eficiente.

Además, hay que tener en cuenta que la programación orientada a datos no requiere del uso del recolector de basura de C#, por lo que la recolección de memoria se hace mucho más eficiente. Por todo esto se puede llegar a entender que estos números sean tan bajos, casi se puede asumir que para esta cantidad de entidades en pantalla el coste de destrucción de entidades es 0ms.

6.2.4 Comparativa de los tiempos de creación de naves

Si se analiza el tiempo de la creación de naves, se podrá observar que se produce un comportamiento similar al que se podía ver en el caso de la creación de edificios, y esto queda reflejado en la figura 6.17.

	Entidades	Tiempo Mono	Tiempo Dots
0	100	3.203	20.9020
1	225	55.118	0.9963
2	400	116.438	19.9440
3	625	189.494	29.9090
4	900	269.279	55.2890
5	1225	389.513	59.8610
6	1600	498.665	80.1650
7	2025	651.956	105.7990
8	2500	812.053	142.6660
9	3025	1012.196	176.9940
10	3600	1232.722	21.9400
11	4225	1378.984	261.6610
12	4900	1640.034	308.7910
13	5625	1892.482	359.0380
14	6400	2027.994	519.4940
15	7225	2573.419	478.7160
16	8100	2692.938	53.1730
17	9025	2956.826	60.5270
18	10000	3163.962	658.2360

Figura 6.17 Datos de los tiempos de creación de naves

En este caso se puede observar un comportamiento más errático que en el caso de la creación de edificios, un hecho que queda visiblemente demostrado por la figura 6.18.

Comparativa del tiempo de creación de naves

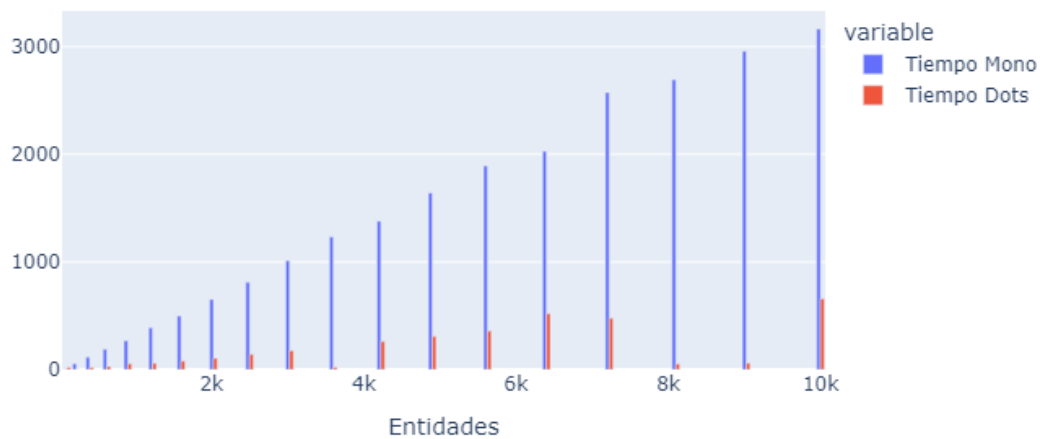


Figura 6.18 Gráfica de los tiempos de creación de naves

Aun con este comportamiento más errático, la tendencia sigue siendo clara, el tiempo que le lleva a la configuración de la POO es mucho mayor al tiempo que lleva a la configuración de la POD.

6.2.5 Comparativa de los tiempos de destrucción de naves

Por último, se analizarán los tiempos de destrucción de las naves bajo ambas configuraciones. Estos tiempos se pueden ver en la figura 6.19.

	Entidades	Tiempo Mono	Tiempo Dots
0	0	0.0000	0.002030
1	100	0.0000	0.000713
2	225	0.3238	0.000046
3	400	0.0000	0.000071
4	625	0.0000	0.000091
5	900	0.9977	0.000108
6	1225	0.9974	0.000155
7	1600	29.9120	0.000175
8	2025	30.2820	0.000215
9	2500	39.9040	0.000278
10	3025	46.5670	0.000319
11	3600	50.7390	0.000383
12	4225	6.0670	0.000467
13	4900	8.0160	0.000517
14	5625	7.9790	0.000572
15	6400	109.7060	0.000647
16	7225	109.7060	0.000721
17	8100	120.1450	0.000648
18	9025	130.7830	0.000751

Figura 6.19 Datos de los tiempos de destrucción de naves

De nuevo se puede observar unos datos muy similares a los obtenidos de la destrucción de los edificios, con una tendencia casi idéntica, aunque comprendan datos diferentes.

Capítulo 7

Conclusiones y trabajo futuro

7.1 Conclusiones

Tal y como se ha comentado a lo largo del presente trabajo, el objetivo del mismo ha sido comparar la programación orientada a objetos y la programación orientada a datos de tal forma que se pueda saber cuál de los dos paradigmas sería más interesante de usar y bajo qué circunstancias.

Como se ha podido observar a lo largo del capítulo anterior, en términos generales, la programación orientada a datos obtiene un mejor rendimiento que la programación orientada a objetos en prácticamente todos los casos de uso investigados y evaluados.

Tanto en el caso de incluir un sobrecalentamiento, como en el caso de utilizar el Job System, a nivel de rendimiento la programación orientada a datos es superior a la programación orientada a objetos. Aun así, es importante considerar algunos aspectos más.

Aunque DOTS sea por lo general mejor que MONO, su comportamiento es más errático e impredecible en ciertos casos. En un desarrollo, es importante poder prever

cómo se va a comportar tu videojuego, y desarrollar en DOTS puede añadir incertidumbre al desarrollo.

Por otro lado, la programación orientada a datos requiere de un cambio de paradigma y de mentalidad bastante grande que a muchos programadores les puede costar realizar. Esto supone que exista una barrera bastante alta para que un desarrollador sea capaz de trasladar sus conocimientos en MONO y aplicarlos en DOTS, pues ambos paradigmas no tienen demasiado que ver el uno con el otro.

Por último, es interesante mencionar que las librerías de DOTS fueron actualizadas a su versión 1.0 hace relativamente poco tiempo (Mayo, 2023). Esto supone que aún queda trabajo por hacer por parte de los desarrolladores de las mismas. Por lo tanto, aún quedan muchas características por implementar y pulir en Unity DOTS.

Para el desarrollador, esto implica que existan casos de uso que no se puedan desarrollar siguiendo Unity DOTS, o que estos casos de uso sean mucho más rentables y factibles de desarrollar siguiendo la POO. Aun así, un desarrollador de videojuegos puede utilizar DOTS en aquellas partes de su videojuego que considere que es rentable y recurrir a MONO para el resto de las partes de su videojuego. Esta es una gran ventaja a la hora del desarrollo, ya que se permite la comunicación entre sistemas desarrollados bajo diferentes paradigmas. Por otro lado, si el caso de uso que se pretende desarrollar no requiere de un gran procesamiento de entidades, el uso de la POD en Unity puede no ser la mejor solución.

El futuro de Unity DOTS es increíblemente prometedor, y con el desarrollo de unos sencillos casos de uso ya se ha podido observar una gran mejoría de rendimiento. Será interesante ver la evolución que tiene DOTS a lo largo de los próximos años, pues por lo menos parece que de momento va por el buen camino.

7.2 Competencias adquiridas y específicas de computación

En esta sección se pretende detallar las distintas competencias que se han adquirido durante el desarrollo del presente trabajo.

[CM3] Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar,

desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

[CM6] Capacidad para desarrollar y evaluar sistemas interactivos y de presentación de información compleja y su aplicación a la resolución de problemas de diseño de interacción persona computadora.

[CM7] Capacidad para conocer y desarrollar técnicas de aprendizaje computacional y diseñar e implementar aplicaciones y sistemas que las utilicen, incluyendo las dedicadas a extracción automática de información y conocimiento a partir de grandes volúmenes de datos.

7.3 Trabajo futuro

Durante el desarrollo del presente trabajo, han ido surgiendo diversas ideas de pruebas y prototipos a desarrollar que permitan una mejor comprensión del tema a tratar y que permitan llevar a Unity DOTS a su máximo a nivel de rendimiento.

1. Algoritmos de búsqueda de caminos.

El sobrecalentamiento inducido es una herramienta útil para realizar unas pruebas iniciales de rendimiento, pero la aplicación de un algoritmo complejo de búsqueda de caminos, como puede ser el algoritmo A* o algoritmos de fuerza bruta pueden ser muy interesantes para ver cómo se comportan en la programación orientada a datos.

2. Simulaciones a gran escala de ecosistemas.

Cualquier simulación que requiere de gran cantidad de entidades en pantalla y en movimiento es altamente probable que vea mejorado su rendimiento si su desarrollo se realiza con Unity DOTS. Simulaciones de bancos de peces, simulaciones de olas, simulaciones de fábricas, etc., son algunas de las posibles simulaciones que serían interesantes de desarrollar para realizar un análisis comparativo de dichas simulaciones desarrolladas con ambos paradigmas de programación.

3. Comparaciones de Unity DOTS con procesos lanzados a la GPU de forma directa.

Este trabajo ha demostrado que Unity DOTS es superior en rendimiento a la POO en Unity, ¿pero podrá Unity DOTS mantener esta ventaja en el caso de comparar su

eficiencia con el mismo prototipo desarrollado con un motor personalizado, que acceda directamente a la GPU?

Esta sería una comparación increíblemente interesante de realizar, pues la opción de desarrollar un motor de videojuegos personalizado y realizar pruebas directamente a la GPU ha sido una de las pocas formas que han tenido los desarrolladores de videojuegos de llevar a buen puerto ciertos casos de uso que usando un motor convencional no podían realizar.

Esta comparativa arrojaría luz a los desarrolladores sobre si es mejor realizar un motor propio o utilizar las librerías de Unity DOTS para una serie de casos de uso concretos, algo que el presente trabajo no es capaz de aportar.

4. Creación de un prototipo multijugador.

Uno de los casos de uso que la propia empresa de Unity utiliza para publicitar el gran rendimiento de Unity DOTS es el caso de los videojuegos multijugador. A este respecto sería de gran interés realizar un análisis comparativo de los tiempos de envío y recepción de paquetes en un videojuego multijugador (desarrollado con MONO y DOTS) entre cliente y servidor, así como de la pérdida de paquetes y de la precisión en los envíos.

Bibliografía

1. Unity Technologies. (s. f.). Unity DOTS. Recuperado de <https://unity.com/dots>
2. Tarodev. (2023, 26 de Marzo). How To Render 2 Million Objects At 120 FPS. Recuperado de https://www.youtube.com/watch?v=6mNj3M1il_c&pp=ygUMVGFyb2RldiBkb3Rz
3. Geeks For Geeks. (2023, 09 de Febrero). Introduction of Object-Oriented Programming. Recuperado de <https://www.geeksforgeeks.org/introduction-of-object-oriented-programming/>
4. Will, B. (2016, 18 de Enero). Object-Oriented Programming is Bad. Recuperado de <https://www.youtube.com/watch?v=QM1iUe6lofM>
5. Unity Technologies. (s. f.). GameObjects. Recuperado de <https://docs.unity3d.com/es/530/Manual/GameObjects.html>
6. Unity Technologies. (s. f.). Execution Order of Events. Recuperado de <https://docs.unity3d.com/es/530/Manual/ExecutionOrder.html>
7. Unity Technologies. (s. f.). Managed Components. Recuperado de <https://docs.unity3d.com/Packages/com.unity.entities@1.0/manual/components-managed.html#:~:text=Unlike%20unmanaged%20components%2C%20managed%20components,them%20in%20Burst%20compiled%20code>
8. Unity Technologies. (s. f.). Performance and Memory Overview. Recuperado de <https://docs.unity3d.com/Manual/performance-memory-overview.html>
9. Sharvit, Y. (2022, 22 de Junio). Principles of Data-Oriented Programming. Recuperado de <https://blog.klipse.tech/dop/2022/06/22/principles-of-dop.html>

10. Unity Technologies. (s. f.). Entities. Recuperado de <https://docs.unity3d.com/Packages/com.unity.entities@1.2/manual/index.html>
11. Unity Technologies. (s. f.). Entities Graphics. Recuperado de <https://docs.unity3d.com/Packages/com.unity.entities.graphics@1.0/manual/index.html>
12. Unity Technologies. (s. f.). Netcode for Entities. Recuperado de <https://docs.unity3d.com/Packages/com.unity.netcode@1.2/manual/index.html>
13. Unity Technologies. (s. f.). Physics. Recuperado de <https://docs.unity3d.com/Packages/com.unity.physics@1.0/manual/index.html>
14. Unity Technologies. (s. f.). Job System. Recuperado de <https://docs.unity3d.com/Manual/JobSystem.html>
15. Unity Technologies. (s. f.). Collections. Recuperado de <https://docs.unity3d.com/Packages/com.unity.collections@1.2/manual/index.html>
16. Unity Technologies. (s. f.). Burst Compiler. Recuperado de <https://docs.unity3d.com/Packages/com.unity.burst@0.2/manual/index.html>
17. Unity Technologies. (s. f.). Mathematics. Recuperado de <https://docs.unity3d.com/Packages/com.unity.mathematics@1.3/manual/index.html>
18. Will, B. (2022, 28 de Septiembre). Unity Entities package 1.0 - Entities and Components. Recuperado de <https://www.youtube.com/watch?v=jzCEzNoztzM>
19. Unity Technologies. (s. f.). EntityManager Class. Recuperado de <https://docs.unity3d.com/Packages/com.unity.entities@0.0/api/Unity.Entities.EntityManager.html>
20. Will, B. (2023, 21 de Octubre). Unity ECS Baking. Recuperado de <https://www.youtube.com/watch?v=r337nXZFYeA>
21. Will, B. (2022, 28 de Septiembre). Unity Entities package 1.0 - Systems. Recuperado de <https://www.youtube.com/watch?v=k07I-DpCcvE>
22. Will, B. (2023, 4 de Abril). HelloCube walkthrough (Unity EntityComponentSystem samples). Recuperado de <https://www.youtube.com/watch?v=32TLgtA9yUM>
23. Will, B. (2022, 28 de Septiembre). The Unity Job System. Recuperado de <https://www.youtube.com/watch?v=jdW66hA-Qu8>
24. Unity Technologies. (s. f.). Creating Scenes. Recuperado de <https://docs.unity3d.com/es/2018.4/Manual/CreatingScenes.html>

-
25. Maida, E. G., & Pacienza, J. (2015). Metodologías de desarrollo de software. Universidad Católica Argentina. Recuperado de <https://repositorio.uca.edu.ar/bitstream/123456789/522/1/metodologias-desarrollo-software.pdf>
26. Vartiainen, P. (2002, July). On the Principles of Comparative Evaluation. *Evaluation*, 8(3), 359-371. <https://doi.org/10.1177/135638902401462484>
27. Unity Technologies. (s. f.). EntityComponentSystemSamples. Recuperado de <https://github.com/Unity-Technologies/EntityComponentSystemSamples>
28. Ketra Games. (2023, 14 de Febrero). Improve Game Performance with Unity's Job System and the Burst Compiler! (Unity Tutorial). Recuperado de <https://www.youtube.com/watch?v=cJEmmvDQx5M>
29. Kenney. (s. f.). Recuperado de <https://www.kenney.nl/>