TecnoCampus
Escola Superior Politècnica

Centre adscrit a la

upf. Universitat Pompeu Fabra *Barcelona*

Grado en Diseño y Producción de Videojuegos

Flocking sobre paradigma ECS

Memoria final

Manuel Duro Santos

Tutor: Dr. Enric Sesa Nogueras

Curso 2021-22

TecnoCampus Mataró-Maresme

Agradecimientos

A meus pais e á miña parella polo seu apoio durante o desenvolvemento dos dous traballos de fin de Grado.

A mi tutor, Enric Sesa, por guiarme durante el planteamiento y realización de este proyecto y proporcionar feedback y comentarios de gran valor.

Abstract

This project consists about the documentation, development and analysis of flocking models to examine their performance comparing implementations based on object-oriented paradigm against implementations based on ECS. The project focuses on the elemental definition of flocking and, afterwards, analyzes which paradigm fits better to this set of motor behaviors. The implementations have been coded using the videogame engines Unity and Bevy.

Resum

Aquest treball tracta sobre la documentació, desenvolupament i anàlisi de models de flocking per a examinar el seu rendiment al comparar implementacions realitzades sobre el paradigma orientat a objecte amb implementacions realitzades sobre ECS. El projecte s'enfoca en la definició elemental de flocking per a, posteriorment, valorar quin paradigma s'adequa més a aquest conjunt de comportaments motors. Les implementacions realitzades s'han programat utilitzant els motors de videojocs Unity i Bevy.

Resumen

Este trabajo trata sobre la documentación, desarrollo y análisis de modelos de flocking para examinar su rendimiento al comparar implementaciones realizadas sobre el paradigma orientado a objeto con implementaciones realizadas sobre ECS. El proyecto se enfoca en la definición elemental de flocking para, posteriormente, valorar qué paradigma se adecúa más a este conjunto de comportamientos motores. Las implementaciones realizadas se han programado utilizando los motores de videojuegos Unity y Bevy.

Índice

Índice de figuras

Glosario de términos

Bevy. Motor de videojuegos orientado íntegramente a una estructura del tipo ECS.

Boid. Bird-oid object, es un término acuñado por Craig Reynolds y que representa a cada una de las entidades que participan en una bandada.

Composición. Relación que se establece entre varias clases que implica que dentro de una clase hay otras clases como atributos.

ECS. Entity Component System, un patrón de arquitectura de software cuyo objetivo es conseguir una gran eficiencia gracias a un tratamiento y almacenamiento de los datos óptimo, permitiendo procesar un mayor número de entidades que en un modelo basado en la programación orientada al objeto.

Flocking. Es el tipo de comportamiento que se puede apreciar en las bandadas de pájaros. Para imitar dicho comportamiento, se han desarrollado modelos computacionales que, además, son generalizables para otro tipo de animales, como los bancos de peces (Reynolds, 1987).

Flockmate. Compañero de bandada de un boid.

FPS. Siglas de fotogramas por segundo (Frames Per Second).

POO. Programación Orientada a Objeto. Se trata de un paradigma de programación basada en el concepto de objeto, clase, herencia, polimorfismo, etc. Las siglas en inglés son OOP (Object Oriented Programming).

Steering behaviour. Conjunto de acciones que trata de determinar el camino de un personaje basándose en su dirección y posicionamiento.

1. Introducción

A medida que se ha producido un avance tecnológico significativo en el campo de la computación, el sector de los videojuegos ha ido mejorando sus técnicas de desarrollo hasta conseguir niveles de especialización muy elevados.

Concretamente, podemos hablar del flocking, un modelo computacional que trata de simular los comportamientos que se producen en las bandadas de pájaros o en los bancos de peces. Esta técnica tiene aplicación directa tanto en la gestión del movimiento de personajes, de elementos del mundo o incluso de partículas.

Por otra parte, los videojuegos tienen unos requerimientos técnicos que están estrechamente ligados a su condición de software en tiempo real y, además, a la necesidad de estar a la vanguardia tecnológica. Por este motivo, han surgido métodos que tratan de optimizar el rendimiento del hardware al máximo para poder ofrecer experiencias más espectaculares al jugador. Concretamente, es el caso del Entity Component System, un patrón de arquitectura de software cuyo objetivo es conseguir una gran eficiencia gracias a un tratamiento y almacenamiento de los datos óptimo, permitiendo procesar un mayor número de entidades que en un modelo basado en la programación orientada al objeto.

En este TFG se hace un acercamiento al modelo computacional de flocking a través de la filosofía del Entity Component System y la programación orientada a datos. Relacionar los conceptos de flocking y ECS tiene interés debido a que tienen en común que involucran un número de entidades muy elevado y, por tanto, el flocking puede verse tremendamente beneficiado por una arquitectura potencialmente óptima en lo que se refiere al tratamiento de los datos.

Para ello, se desarrollarán herramientas prácticas parametrizables para un determinado motor de videojuegos y, de este modo, comparar el rendimiento del flocking sobre un paradigma ECS con respecto a uno más tradicional.

2. Objetivos

Los principales objetivos de este TFG son:

- Analizar la adecuación del patrón de diseño ECS al modelo computacional de flocking.

- Desarrollar una herramienta práctica parametrizable de esta aproximación del flocking.

- Comparar herramientas de flocking utilizando una aproximación con ECS y otra bajo el paradigma de orientación a objeto tradicional utilizando diversas tecnologías.

3. Análisis de referentes

En este apartado se muestran ejemplos que sirven de inspiración para la realización de este proyecto, tanto de flocking como de ECS. Además, se mencionan herramientas similares al principal producto de este TFG que se pueden encontrar en las tiendas de los principales motores de videojuegos.

3.1. Aplicaciones de flocking

Las primeras aplicaciones de flocking en el mundo de la animación y los videojuegos se remontan a los inicios de la generación de gráficos 3D por ordenador. Es el caso de Stanley and Stella in: Breaking The Ice (Figura 1), un corto de 1987 en el que colaboró Craig Reynolds para demostrar las posibilidades del flocking en este campo.

Figura 1. Bandada de Stanley and Stella in: Breaking the Ice. Fuente: Stanley and Stella in: Breaking the Ice

Más adelante, en la película Batman Returns (1992) también se utiliza flocking para dotar de realismo al movimiento de una bandada de murciélagos creada digitalmente (Figura 2).

Figura 2. Escena de Batman Returns. Fuente: Batman Returns

En lo que se refiere al mundo de los videojuegos, hay propuestas que implementan versiones excesivamente reducidas de lo que podría ser un modelo de flocking, como en The Witcher 3 con una bandada de pájaros que gira en círculos y algunos de sus "boids" se desvían ligeramente de la trayectoria en común (Figura 3).

Figura 3. Bandada de pájaros en The Witcher 3. Fuente: The Witcher 3.

Por otra parte, existen videojuegos que utilizan el flocking de una forma mucho más metódica debido a las necesidades del propio gameplay. Es el caso de Subnautica (Figura 4), un juego de exploración submarina en el que algunos peces se desplazan en bancos, replicando un movimiento relativamente realista y orgánico.

Figura 4. Bancos de peces en Subnautica. Fuente: Subnautica.

3.2. Aplicación de ECS

Aquellos videojuegos que necesitan disponer de un gran número de entidades en pantallas son muy susceptibles de ser desarrollados bajo el paradigma ECS.

En juegos de estrategia como Total War (Figura 5), se pueden llegar a mostrar en pantalla miles de entidades con movimientos y comportamientos independientes. Utilizar una arquitectura óptima será clave para aprovechar los recursos al máximo.

Figura 5. Soldados en Total War Saga: Troy. Fuente: Total War Saga: Troy

En el año 2018, Unity empezó a trabajar en una demo técnica para mostrar las funcionalidades de DOTS. En esta demo, Megacity (Figura 6), Unity mostraba un mundo con cientos de miles de Game Objects con un alto nivel detalle y que se desplazaban de forma autónoma e individual. Es destacable que el rendimiento en esta demo es tan óptimo que permite que se ejecute en móviles.

Figura 6. Demo técnica de Unity, Megacity. Fuente: Unity

3.3. Herramientas similares

Los principales motores de videojuegos de la industria disponen de sus propias tiendas en las que la comunidad y las empresas pueden ofertar recursos y herramientas.

En el caso de Unity, en la Asset Store hay herramientas de uso genérico para utilizar flocking como Flocking Tool 2D/3D (Figura 7).

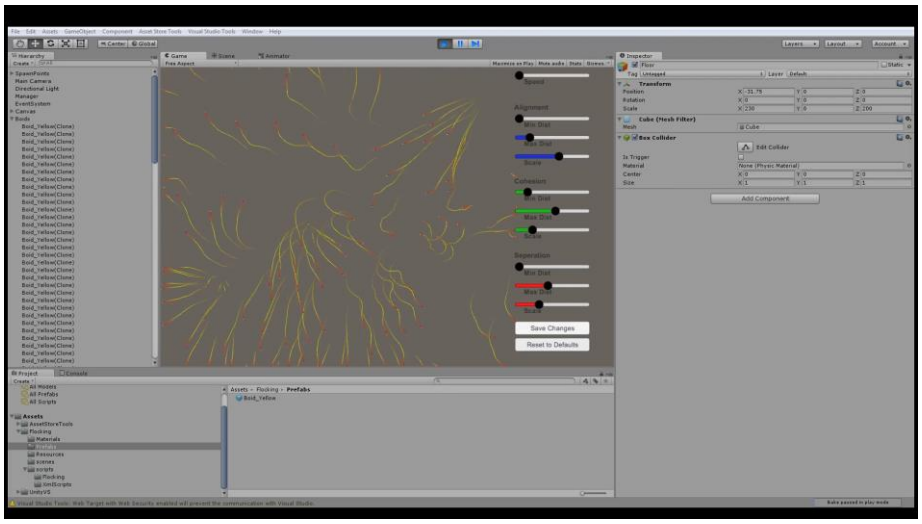


Figura 7.Flocking Tool 2D/3D. Fuente: Unity Asset Store

Por otra parte, en la tienda de Unity también hay herramientas especializadas en el flocking de un determinado tipo de ave, vendiendo el código, los modelos, las texturas y las animaciones (Figura 8).

Figura 8. Bird Flock Bundle. Fuente: Unity Asset Store

En lo que respecta a Unreal Engine, en la tienda también se pueden encontrar herramientas de pago para utilizar flocking. Flocking Behaviour System (Figura 9) implementa cinco *steering behaviours*: alineación, cohesión, separación, seguir un camino/ir a una meta y esquivar obstáculos.

Figura 9. Boids en Flocking Behaviour System. Fuente: Unreal Engine Marketplace

4. Marco teórico

En el marco teórico se analiza qué es el flocking y de dónde surge, los paradigmas de programación más destacables para la realización de este trabajo y, finalmente, la forma en que diversos motores de videojuegos implementan dichos paradigmas.

4.1. Comportamientos motores

Los comportamientos motores hacen referencia al conjunto de acciones que proporcionan a una entidad la habilidad de desplazarse en su mundo de una forma orgánica, reactiva, no planificada e improvisada (Reynolds, 1999). Este tipo de comportamientos se divide en una jerarquía de tres capas:

- Selección de acciones. Hace referencia a las estrategias, metas y planes que configuran el movimiento.
- Direccionamiento. Se trata de la búsqueda de caminos a través de la dirección y el posicionamiento.
- Locomoción. Es la capa más superficial, la cual hace referencia a la animación y articulación.

En este TFG se presta especial atención a los comportamientos direccionales (o *steering behaviours*), ya que son la base teórica sobre la que se construye el flocking (uno de los temas centrales del trabajo.

Los comportamientos direccionales definen al conjunto de acciones que trata de determinar el camino a seguir basándose en la dirección y el posicionamiento de la entidad. No se debe confundir con el *pathfinding*, ya que hace referencia a algoritmos que buscan el camino más corto en un grafo o en una red (como Dijkstra o A*), mientras que los *steering behaviours* hacen referencia a la filosofía del comportamiento a un alto nivel sin precisar los métodos para lograr dicho objetivo. A continuación, se listan algunos de los comportamientos direccionales más relevantes:

- Búsqueda. La entidad se direcciona hacia un objetivo estático.
- Huida. Es la acción inversa la búsqueda. La entidad se direcciona de forma opuesta a un objetivo estático.
- Persecución. Es similar la búsqueda, sólo que con objetivos móviles.
- Evasión. Es similar a la huida, sólo que con objetivos móviles.
- Evasión de obstáculos. Este comportamiento consiste en que la entidad tiene la habilidad de hacer ligeras modificaciones en su ruta para evitar colisionar contra obstáculos.
- Deambulación. Consiste en que la entidad hace modificaciones sutiles de carácter aleatorio en su trayectoria para simular que vaga sin rumbo.

Existen más comportamientos direccionales además de los mencionados. Es necesario mencionar en concreto a la separación, la alineación y la cohesión. Al combinar estos tres direccionamientos se obtiene la versión más básica de un comportamiento derivado: el flocking.

4.2. Flocking

El flocking (en español, comportamiento de bandada) es el tipo de comportamiento que se puede apreciar en las bandadas de pájaros. Para imitar dicho comportamiento, se han desarrollado modelos computacionales que, además, son generalizables para otro tipo de animales, como los bancos de peces (Reynolds, 1987).

Craig Reynolds acuñó el término “boid” ("bird-oid object") para denominar a cada una de las entidades que participan en la bandada. Por otra parte, los compañeros de bandada de un boid son denominados “flockmates”.

Los boids sólo pueden reaccionar a otros flockmates si se encuentran dentro de una distancia establecida (medida desde el centro del boid) y si tienen un ángulo similar, el cual determina su dirección de vuelo. Los flockmates que se encuentren fuera de su cono de visión, determinado por una distancia y un ángulo establecidos (Figura 10), se ignorarán.

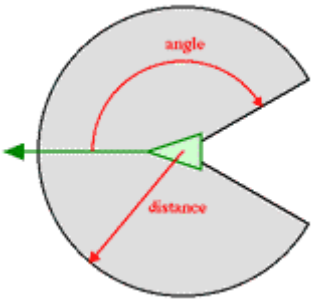


Figura 10. Área de vecindad. Fuente: Reynolds, 1987

Las principales reglas de interacción del movimiento colectivo se describen en el ámbito de la biología, destacando las fuerzas de atracción y repulsión que mantienen la distancia entre los individuos colindantes de, por ejemplo, un banco de peces (Breder, 1954). Este comportamiento se ha estudiado a través de modelos numéricos estocásticos que son capaces de simular patrones de movimiento en bandada de forma parametrizable y que, además, han permitido desarrollar un buen entendimiento de los mecanismos de comportamiento relacionados con los bancos de peces (Aoki, 1982).

El modelo de flocking es un comportamiento emergente derivado de la combinación de tres *steeering behaviours* más simples. Estos tres comportamientos básicos del flocking son la separación, la alineación y la cohesión; los cuales se detallan en los siguientes subapartados.

Por otra parte, cabe destacar que los modelos de flocking pueden incorporar otras características como esquivar obstáculos y direccionarse hacia un determinado objetivo. De este modo, se consigue un comportamiento más realista que se puede integrar fácilmente con las casuísticas típicas del mundo del videojuego o de la animación. Estos modelos se pueden implementar utilizando máquinas de estados (Figura 11), definiendo los diferentes posibles estados y las relaciones entre ellos.

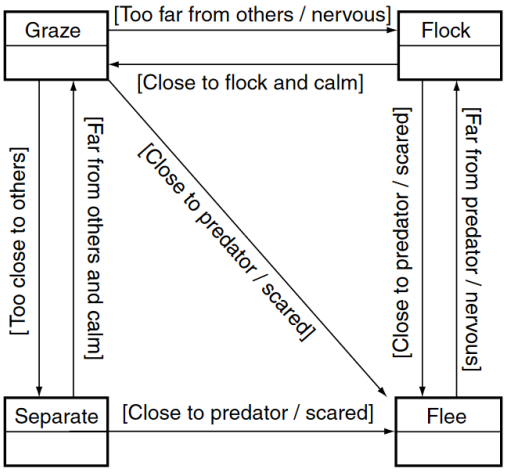


Figura 11. Máquina de estados de un modelo de flocking. Fuente: Millington, 2009

A continuación, se explican los tres *steering behaviours* que conforman el modelo de flocking más básico.

4.2.1. Separación

La separación tiene como objetivo evitar aglomeraciones con otros compañeros de bandada. El boid calcula la distancia con respecto a sus flockmates para trazar una trayectoria que lo aleja de ellos (Figura 12).

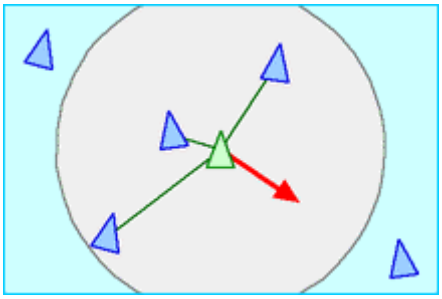


Figura 12. Separación en Flocking. Fuente: Reynolds, 1987

La implementación de la separación en pseudocódigo sería la siguiente (Millington, 2009):

```
class Separation
    # Variable que guarda la información de movimiento del personaje
    character

    # Lista de objetivos potenciales
    targets

    # Variable con el umbral para entrar en acción
    threshold

    # Constante del coeficiente del decay de la
    # ley de la fuerza del cuadrado inverso
    decayCoefficient

    # Variable con la aceleración máxima del personaje
    maxAcceleration


    def getSteering():
        # Variable con el resultado
        steering = new Steering ()

        # Iterar sobre cada objetivo
        for target in targets:
            # Comprobar si el objetivo está suficientemente cerca
            direction = target.position - character.position
            distance = direction.length()
            if distance < threshold:
                # Calcular la fuerza de repulsión
                strength = min(decayCoefficient / (distance *
distance), maxAcceleration)

                # Añadir la aceleración
                direction.normalize()
                steering.linear += strength * direction

        # En cuanto se acaba el loop, se devuelve el resultado
        return steering
```

Este algoritmo itera sobre una lista con todos los potenciales objetivos para determinar si están suficientemente cerca. Si es afirmativo, se calcula la fuerza y dirección que se debe aplicar. Una vez hayan terminado las iteraciones, se aplicará la suma de las fuerzas con respecto a los boids que están en su área de vecindad.

Cabe destacar que en este ejemplo se está utilizando un tipo de repulsión en concreto: la ley de la fuerza del cuadrado inverso. Existen otras alternativas como la repulsión lineal, por lo que en el desarrollo de este TFG se ambos tipos.

4.2.2. Alineación

La alineación consiste en que el boid mantiene la velocidad y orientación de sus compañeros de bandada. El boid analiza las velocidades (vectoriales) de sus flockmates para calcular la aceleración que necesita para adquirir dicha velocidad y dirección (Figura 13).

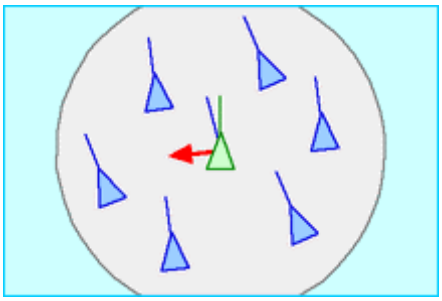


Figura 13. Alineación en Flocking. Fuente: Reynolds, 1987

La implementación de la alineación en pseudocódigo sería la siguiente (Millington, 2009):

```
class VelocityMatch:
    # Variables con información de movimiento del personaje y objetivo
    character
    target

    # Aceleración máxima del personaje
    maxAcceleration

    # Tiempo para alcanzar la velocidad del objetivo
    timeToTarget = 0.1

    def getSteering(target):
        # Variable con el resultado
        steering = new SteeringOutput()

        # Se calcula la aceleración para
        # alcanzar la velocidad objetivo
        steering.linear = target.velocity - character.velocity
        steering.linear /= timeToTarget

        # Comprobar si aceleración es demasiado grande
        if steering.linear.length() > maxAcceleration:
            steering.linear.normalize()
            steering.linear *= maxAcceleration

        # Retornar resultado
        steering.angular = 0
        return steering
```

Este algoritmo comienza restando las velocidades del objetivo y del boid para, a continuación, dividirlo entre la variable timeToTarget. Lo que se está calculando es la aceleración necesaria para que el boid adquiera la velocidad del objetivo. Se comprueba que la aceleración no supere la velocidad máxima y, finalmente, se retorna el resultado.

4.2.3. Cohesión

La cohesión consiste en moverse hacia la posición media de otros compañeros de bandada. El boid busca colocarse en el punto medio de sus flockmates para mantener la cohesión dentro del grupo (Figura 14).

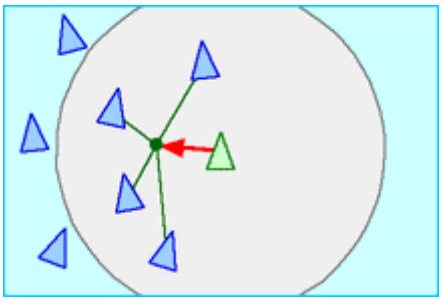


Figura 14. Cohesión en Flocking. Fuente: Reynolds, 1987

A continuación, se muestra una posible implementación de la cohesión en pseudocódigo basada en el artículo de Reynolds (Reynolds, 1999):

```
class Cohesion:
    # Variable con información del personaje
    character

    # Lista de objetivos (compañeros de bandada)
    mates

    # Los objetivos que pasen este umbral no se tendrán en cuenta
    cohesionThreshold

    def getSteering (character, mates):

        # Variable del centro de masas se inicializa a (0,0) o (0,0,0)
        centerOfMass = zero vector
        # Número de compañeros cercanos (acorde al umbral de cohesión)
        count = 0
```

```
# Iterar sobre los compañeros
forEach (mate in mates) {
    if (mate != character) {
        distanceToMate = (mate.position –
        character.position).magnitude
        # Sólo considerar compañeros cercanos
        if (distanceToMate <= cohesionThreshold) {
            centerOfMass = centerOfMass + mate.position
            count++
        }
    }
}

# Si no hay compañeros cercanos, finalizar
if (count == 0)
    return null;

# Calcular el centro de masas medio
centerOfMass = centerOfMass / count
surrogateTarget.position = centerOfMass
# Utilizar el steering behaviour "seek" para ir a la posición
SEEK(me, surrogateTarget)
```

Este algoritmo itera sobre todos los flockmates y sólo tiene en cuenta aquellos que están dentro de un determinado umbral. Una vez acaba la iteración, se utiliza el *steering behaviour* "seek" (o búsqueda) para que el boid se desplace hasta esa posición.

4.3. Paradigmas de programación

Un paradigma de programación es un planteamiento que define y establece una serie de reglas e ideas que se utilizan para el diseño y desarrollo de programas. A continuación, se describe uno los paradigmas más utilizados actualmente (la programación orientada a objetos) y se compara con un paradigma más reciente que está ganando popularidad en el sector de los videojuegos (el ECS).

4.3.1. Programación orientada a objetos

El paradigma de la programación orientada a objetos está basado en la idea principal del objeto, el cual puede contener atributos (datos) y funciones (métodos).

Además, en programación orientada a objetos tienen una gran importancia los conceptos de clase e instancia. Una clase es un modelo que define atributos y métodos de un determinado tipo de objeto, mientras que la instancia hace referencia a una entidad en particular de una determinada clase. Por ejemplo, una clase podría ser la definición formal de “perro”, mientras que la instancia es un perro en concreto.

Una gran parte de los lenguajes de programación basados en este paradigma permiten diseñar desde las relaciones naturales que se establecen entre los distintos objetos. De este modo, se plantean dos formas de relacionar objetos distintos: la composición y la herencia.

La herencia es la característica que permite que un objeto herede los atributos y funciones de un objeto padre. La herencia es una forma rápida, sencilla e intuitiva de reutilizar código; permite utilizar el polimorfismo (los objetos hijo pueden adoptar la forma del padre) y tienen versatilidad gracias a la sobreescritura de funciones. Sin embargo, la herencia tiene una serie de problemas asociados debido a que puede no ajustarse bien a las metodologías ágiles de desarrollo por su rigidez estructural (la cual va en contra de la filosofía de que cualquier característica puede cambiar en cualquier momento).

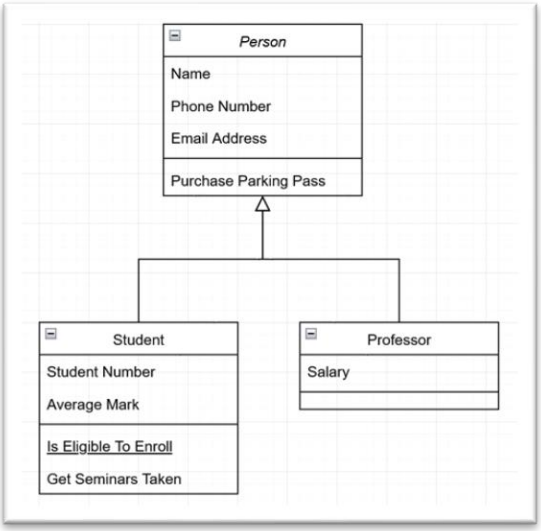


Figura 15. Herencia. Student y Professor heredan de la superclase Person. Fuente: elaboración propia.

La composición implica que dentro de un objeto hay otros objetos como atributos. De este modo, el objeto delega en otros objetos la responsabilidad de realizar determinadas funciones, desacoplándolo del objeto inicial y ofreciendo una mayor flexibilidad que en la herencia.

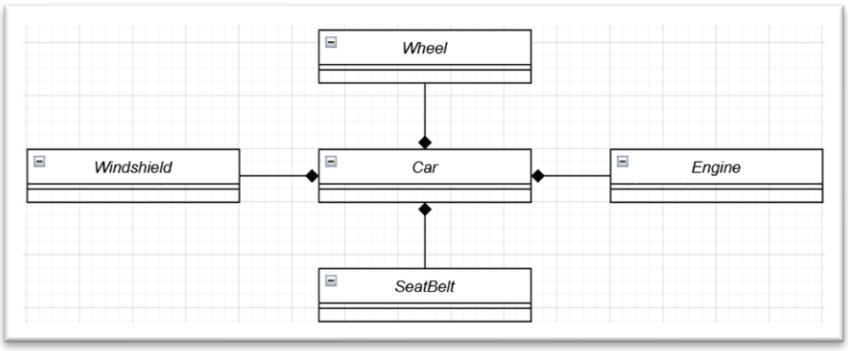


Figura 16. Composición. Car tiene atributos de la clase Engine, Wheel, Windshield y SeatBelt. Fuente: elaboración propia.

La herencia tiene ventajas sobre en la composición en lo que se refiere a la realización de un diseño de software muy intuitivo y que resulta fácil de programar, mientras que la composición implica una solución más compleja en la que generalmente se verán involucradas un mayor número de clases. No obstante, la composición permite una mayor adaptación al cambio, mantenimiento y testeo más fácil y, además, ampliar el diseño con nuevas funcionalidades es una tarea más sencilla si se parte de un modelo basado en composición.

4.3.2. Entity Component System

El ECS (Entity Component System) es un patrón de diseño de software basado en la premisa de “composición sobre herencia” con el objetivo de minimizar la redundancia de datos (Masiukiewicz, 2019). Se fundamenta en la programación orientada a datos (DOD), centrándose en lograr una gran eficiencia en la gestión de datos gracias al procesamiento en paralelo (Fabian, 2013).

La programación orientada a datos es una filosofía que pone el foco en los datos a la hora de diseñar software. Esto implica que se analiza con cautela cómo se guardan los datos en memoria, cómo se leen y cómo se ejecutan, resolviendo posibles errores de caché y mejorando la eficiencia general del programa. Por tanto, esta forma de diseñar software tiene muy en cuenta la forma en la que los datos se relacionan con el hardware.

La DOD acaba con el problema de jerarquías muy complejas de la herencia de la programación orientada a objetos, el código es muy modular y con pocas dependencias, los datos se tratan y se almacenan de una forma óptima y, además, facilita la paralelización y el testeo (Llopis, 2009).

El ECS es una arquitectura basada en esta filosofía y consta de tres elementos principales: entidad, componente y sistema (Martin, 2007):

- Entidad. Un objeto de propósito general que, generalmente, consta únicamente de un identificador.
- Componente. Los datos de un solo aspecto del objeto y de cómo se relaciona con el resto de objetos.
- Sistema. Contiene la lógica del programa y realiza las acciones necesarias en cada entidad que posee un determinado componente.

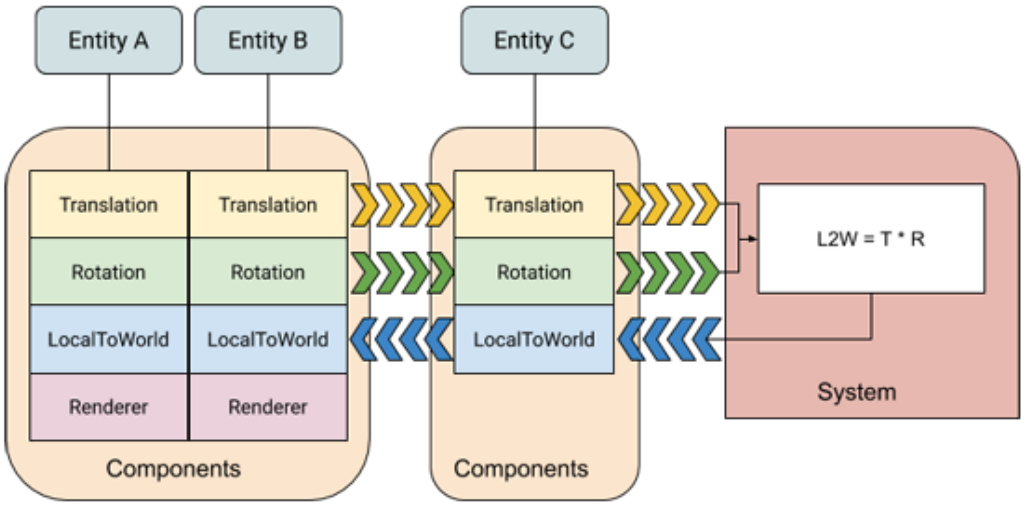


Figura 17. Diagrama de ECS. Fuente: Documentación de Unity

Cada combinación única de componentes que poseen las distintas entidades se denomina arquetipo (Figura 18). Los arquetipos determinan la forma en la que ECS almacena los componentes de una entidad en memoria.

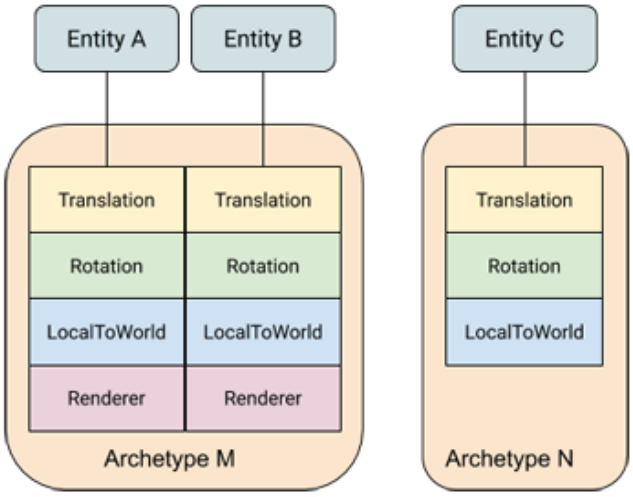


Figura 18. Arquetipos en ECS. Fuente: Documentación de Unity

Los datos de los arquetipos son almacenados en memory chunks (trozos de memoria). No obstante, cuando un chunk se llena, ECS se encargará de crear nuevos chunks para las nuevas entidades del arquetipo (Figura 19).

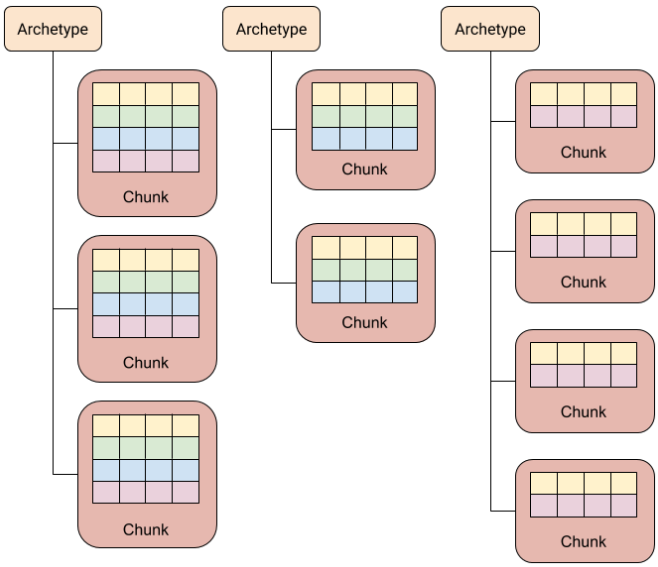


Figura 19. Memory chunks en ECS. Fuente: Documentación de Unity

De este modo, se establece una relación de uno a muchos entre arquetipos y chunks. Esto facilita encontrar entidades al tener que limitar la búsqueda tan sólo a buscar entre los arquetipos (un número muy inferior a la cantidad de entidades).

La importancia de ECS viene debido a es un sistema que trata de sacar el mayor rendimiento del hardware. Las variables que se declaran en un programa, pueden almacenarse en lugares distintos. Generalmente, la mayor parte de las variables se almacenan en la memoria RAM pero, con el objetivo de conseguir un mayor rendimiento, el procesador puede almacenar variables en su memoria caché, a las cuales puede acceder mucho más rápido que a la memoria RAM. No obstante, esto tiene un problema implícito, que se trata de los errores de caché.

Un error de caché es un fallo a la hora de leer o escribir un determinado dato almacenado en caché, el cual se puede deber a un error de lectura de instrucción en caché, de lectura de datos o de escritura de datos. Se debe a una incoherencia entre la información almacenada en caché con la almacenada en la memoria principal.

Las consecuencias de este tipo de errores implican que el hilo en ejecución tiene que esperar hasta obtener el dato correcto. ECS al ser una estructura orientada a datos trata de minimizar que este hecho ocurra, reduciendo tiempos de espera debido a errores de caché.

4.4. Motores de juegos y ECS

En este punto se hace un estudio de diversos motores de juegos y de su aproximación al ECS.

4.4.1. Unity

Unity es un motor de videojuegos creado por la empresa Unity Technologies. Se puede utilizar para crear juegos de dos o tres dimensiones (2D o 3D), además de otro tipo de simulaciones y experiencias interactivas. Se utiliza C# para programar en Unity.

La aproximación de Unity al ECS es a través de su tecnología DOTS, la cual permite aprovechar el paralelismo para procesar los datos y mejorar el rendimiento de los proyectos de Unity. Actualmente, Unity utiliza un sistema híbrido con DOTS en el que sigue manteniendo los clásicos MonoBehaviours. Es remarcable que DOTS (Unity's Data-Oriented Tech Stack) es un conjunto de tecnologías que siguen la filosofía del DOD. Los elementos que conforman esta tecnología son:

- ECS. Es el paradigma sobre el que se construye esta tecnología.
- C# Job System. Se trata de un paquete que permite generar código multihilo.
- Compilador Burst. Es un paquete que genera código nativo rápido y optimizado que tiene en cuenta la plataforma para la que se está compilando.

4.4.2. Unreal Engine

Unreal Engine es un motor de videojuegos creado por la empresa Epic Games. Aunque fue creado para hacer shooters en él, este motor ha ido evolucionando hasta convertirse en un motor versátil que permite hacer cualquier tipo de videojuego. Se utiliza C++ para programar en Unreal Engine.

Unreal Engine no tiene una aproximación al ECS desarrollada por Epic Games pero existe un framework de terceros llamado Apparatus que utiliza este sistema. Tiene un precio de 199.99$.

La otra opción es que el propio usuario monte su propia arquitectura ECS desde cero.

4.4.2. Bevy

Bevy Engine es un motor de videojuegos de código abierto y software libre.

Se encuentra en fase beta, pero su propuesta es la de un motor enteramente basado en ECS. Se utiliza Rust como lenguaje de programación para desarrollar en Bevy.

4.5. Análisis algorítmico

En el ámbito de la Ingeniería Informática y la programación, la algorítmica es el eje central sobre el que orbita el desarrollo tecnológico. Un algoritmo es un conjunto de instrucciones que permiten hallar una determinada solución. Uno o más algoritmos conforman lo que se conoce como un programa informático. Por tanto, en un videojuego (el cual se trata de un tipo de programa informático) habrá múltiples algoritmos que se encarguen de resolver distintas casuísticas involucradas en el mismo.

Debido a que los ordenadores y consolas disponen de recursos limitados que dependen de los avances tecnológicos en el campo del hardware, cobra especial interés ser capaz de determinar el rendimiento y eficiencia del código desarrollado. Si bien existen técnicas que pueden permitir determinar si un algoritmo es mejor que otro (como medir el tiempo en el que se tarda en renderizar un fotograma), se requiere de un análisis teórico y formal que permita valorar la posible eficiencia de un algoritmo sin necesidad de ejecutarlo propiamente y de depender de las particularidades del procesador utilizado, el lenguaje de programación, el compilador, etc. De esta necesidad es de donde surge el análisis algorítmico.

El análisis algorítmico se encarga de estudiar a nivel teórico la complejidad de un determinado algoritmo para estimar los recursos necesarios para ejecutarlos conforme el problema va creciendo de tamaño (Knuth, 1997). Este tipo de análisis es de gran importancia debido a que se analizan los problemas de rendimiento que se pueden dar al utilizar un determinado algoritmo, así como la posibilidad de saber qué algoritmo se adecúa más para resolver un problema en específico.

Para poder analizar el coste de un algoritmo se utiliza la notación asintótica. En este tipo de notación se observa al algoritmo como una función que relaciona el coste de ejecución con diversos parámetros, tal y como podría ser el tamaño de la entrada. A continuación, sólo se mostrará la cota superior asintótica (o notación Big O), para

referirnos a la función que establece la cota superior de otra función cuando la entrada tiende a infinito.

Este tipo de notación se escribe utilizando el formato O(x), donde O hace referencia a que estamos hablando de una cota superior asintótica y x hace referencia al coste del algoritmo.

Este tipo de análisis permitirá poder valorar los modelos de flocking planteados para poder hacer propuestas de mejora y conseguir una mayor eficiencia y tasa de fotogramas.

A continuación, se muestran algunas de las complejidades temporales más destacables.

4.5.1. Crecimiento constante

En el caso de un algoritmo que fuera O(1), estaríamos hablando de un tiempo de ejecución constante y que no varía en función del tamaño de los datos de entrada. Es el mejor caso posible pero también es muy infrecuente.

Figura 20. O(1). Fuente: elaboración propia

En la Figura 20, el eje X se corresponde con el tamaño de la entrada y el eje Y con el coste. Se mantienen completamente constante independientemente del tamaño de la entrada. La línea estará más arriba o más abajo en la gráfica en función del tiempo de ejecución del algoritmo.

4.5.2. Crecimiento lineal

Para el siguiente caso, supongamos que queremos recorrer un array con un bucle. Una forma clásica para recorrerlo sería la siguiente:

```
for (int i = 0; array.length();i++){
    function(array[i]);
}
```

Cada operación (function(array[i])) tendrá un coste constante. Por tanto, en este algoritmo el coste crece de forma directamente proporcional al tamaño del array. Estamos ante un coste lineal O(n) (Figura 21).

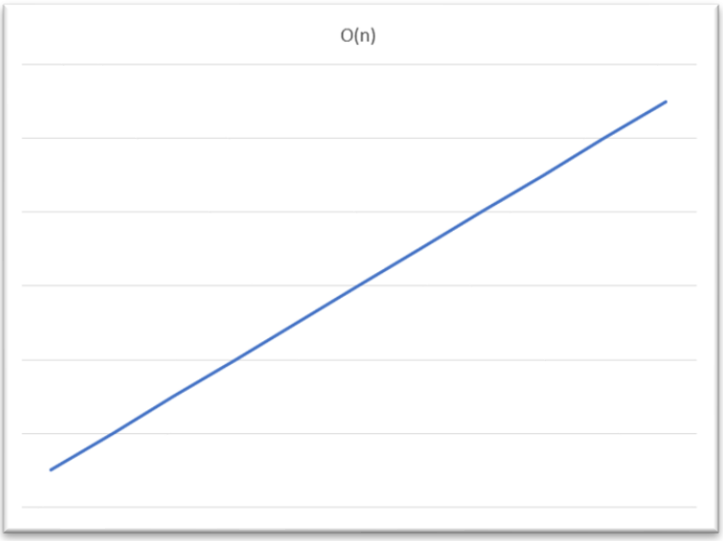


Figura 21. O(n). Fuente: elaboración propia.

La pendiente de la curva variará en función del tiempo de la complejidad de la instrucción que se repite a lo largo del tiempo.

4.5.3. Crecimiento logarítmico

Este caso implica un crecimiento de carácter logarítmico, en el que al principio el coste crece de forma muy abrupta pero que se acaba estabilizando a medida que tiende a infinito (Figura 22). Estamos ante un caso de O(log n).

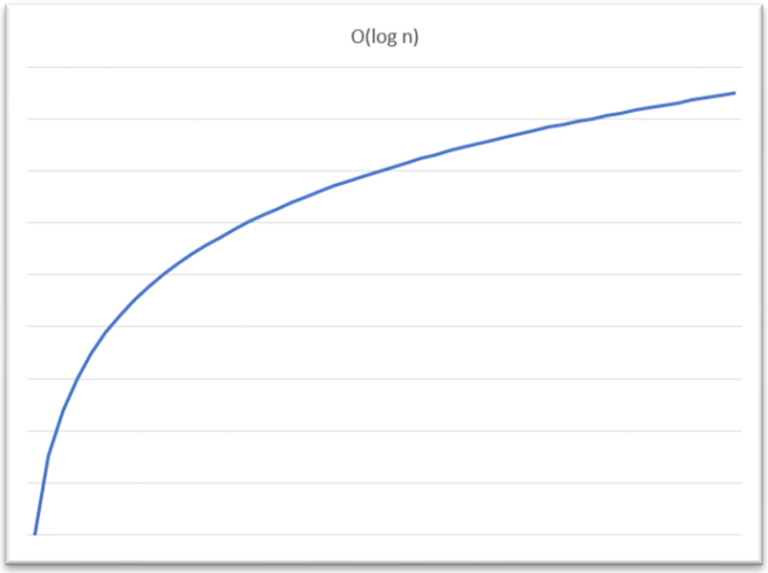


Figura 22. O(log n). Fuente: elaboración propia.

Es común ver este tipo de costes en algoritmos que utilizan árboles binarios o en la búsqueda binaria.

4.5.4. Crecimiento cuadrático

Tomando como referencia el caso presentado en el crecimiento lineal, supongamos que ahora queremos recorrer un array de dos dimensiones. La forma clásica para recorrerlo sería la siguiente:

```
for (int i = 0; array.length();i++){
    for (int j = 0; array.length();j++){
        function(array[i][j]);
    }
}
```

Al igual que en el caso anterior, cada operación (function(array[i][j])) tendrá un coste constante. A diferencia del caso del crecimiento lineal, ahora cada vez que el tamaño

del array aumenta, el número de operaciones realizadas aumenta de forma cuadrática al tratarse de una matriz. Se trata de un coste $O(n^2)$.

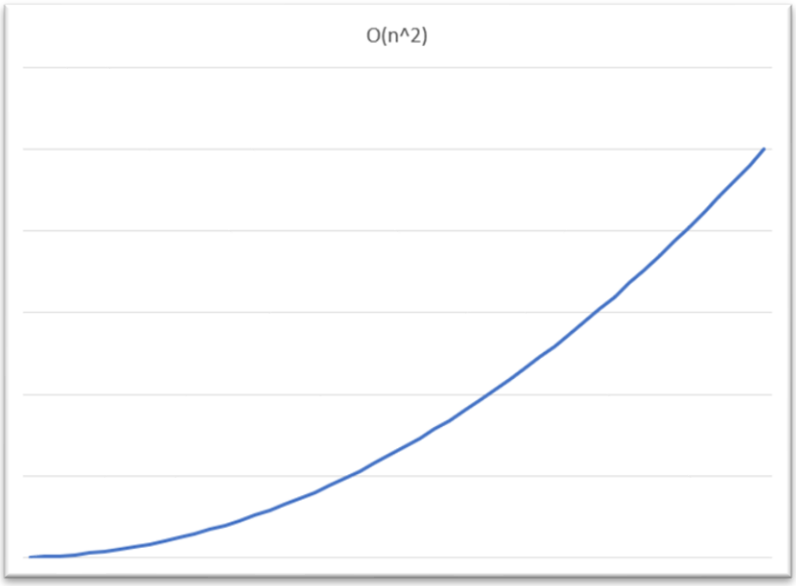


Figura 23. $O(n^2)$. Fuente: elaboración propia.

A medida que el tamaño del problema tiende a infinito, el crecimiento cada vez se vuelve más grande (Figura 23).

El crecimiento cuadrático es generalizable a un crecimiento polinómico en el que el coste aumenta a un ritmo muy elevado. Se engloban en esta categoría costes del estilo $O(n^3)$, $O(n^4)$, etc. Este tipo de crecimiento (lejos de ser óptimo) es viable algorítmicamente.

4.5.5. Crecimiento factorial

Este caso implica una explosión combinatoria que provoca que el algoritmo en cuestión sea completamente inviable a nivel computacional (Figura 24). Estamos ante un caso de $O(n!)$.

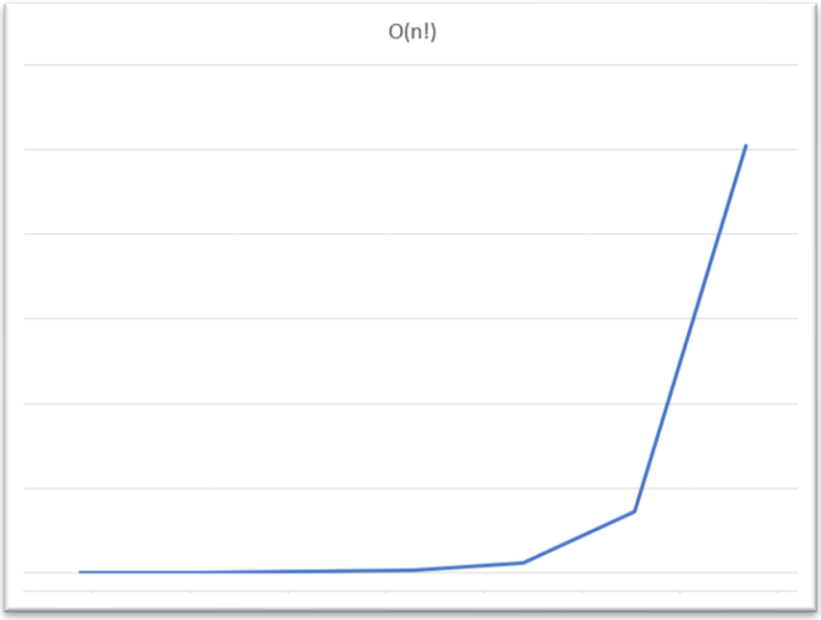


Figura 24. O(n!). Fuente: elaboración propia.

Los algoritmos con este tipo de crecimiento no son aptos para código de producción. Están enfocados más en un área académica de análisis algorítmico, tal y como puede ser el algoritmo de ordenación bogosort, orientado al ensayo y error y cuya utilidad real es prácticamente nula. Se considera intratable algorítmicamente, a pesar de ser resoluble.

5. Diseño metodológico y cronograma

Para realizar este proyecto se utiliza una metodología inspirada en “agile” debido a la naturaleza unipersonal del proyecto. Esta metodología consiste en plantear un desarrollo iterativo e incremental que permite evaluar de forma rápida el progreso del proyecto y que, además, permite una respuesta al cambio ágil.

Las principales premisas de esta filosofía (definidas en “The Manifesto for Agile Software Development”) son las siguientes:

- Individuos e interacciones por encima de procesos y herramientas.
- Software funcional por encima de documentación demasiado extensa.
- Colaboración con el cliente por encima de la negociación del contrato.
- Responder al cambio por encima de seguir un plan.

Concretamente, se utilizará una metodología similar al Scrum, planteando una serie de sprints de duración corta (dos semanas) para poder ir evaluando el desarrollo del proyecto y hacer cambios donde sea necesario.

El proyecto se divide en tres grandes hitos: el anteproyecto, la memoria intermedia y la memoria final. Debido a ello, la metodología se adaptará a estos hitos de forma que el proyecto se vaya desarrollando de forma iterativa e incremental.

El cronograma inicial del proyecto (atendiendo a las fases, objetivos y deadlines) quedaría de la siguiente manera:

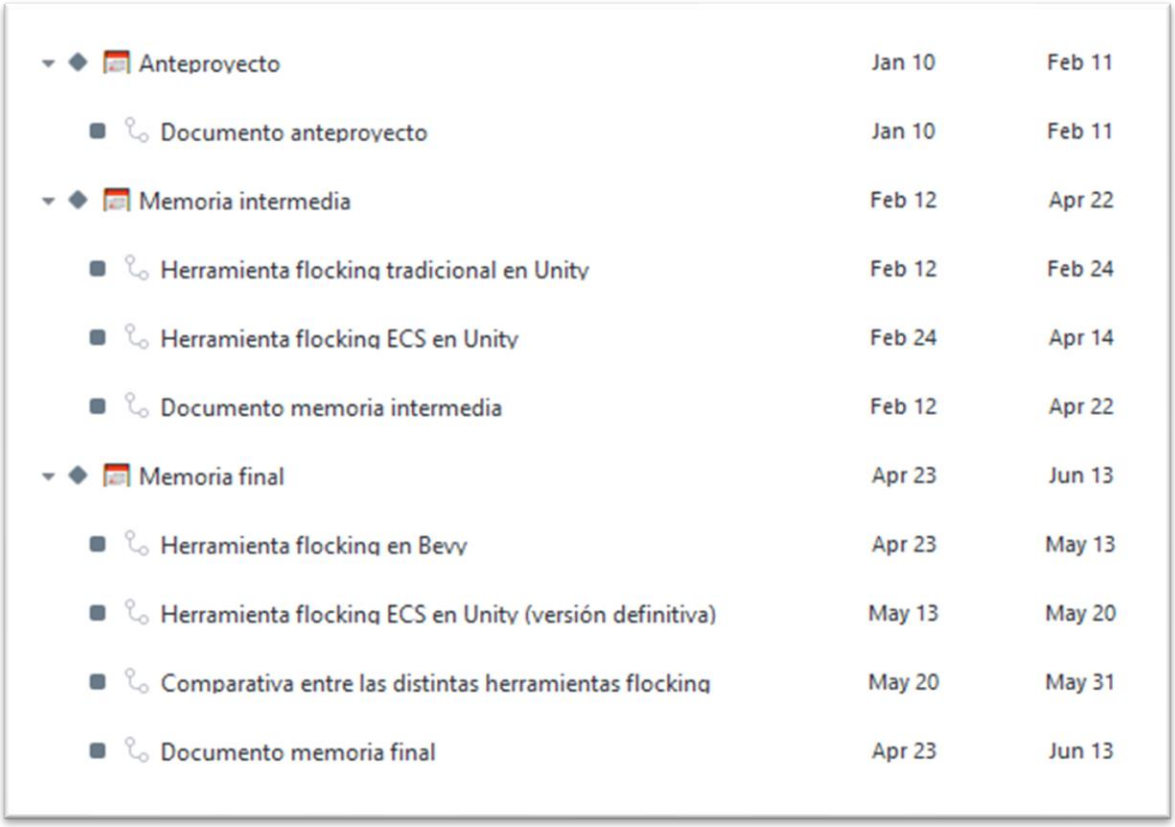


Figura 25. Fases, tareas y deadlines. Fuente: elaboración propia

Lo que en forma calendarizada adquiere la siguiente forma:

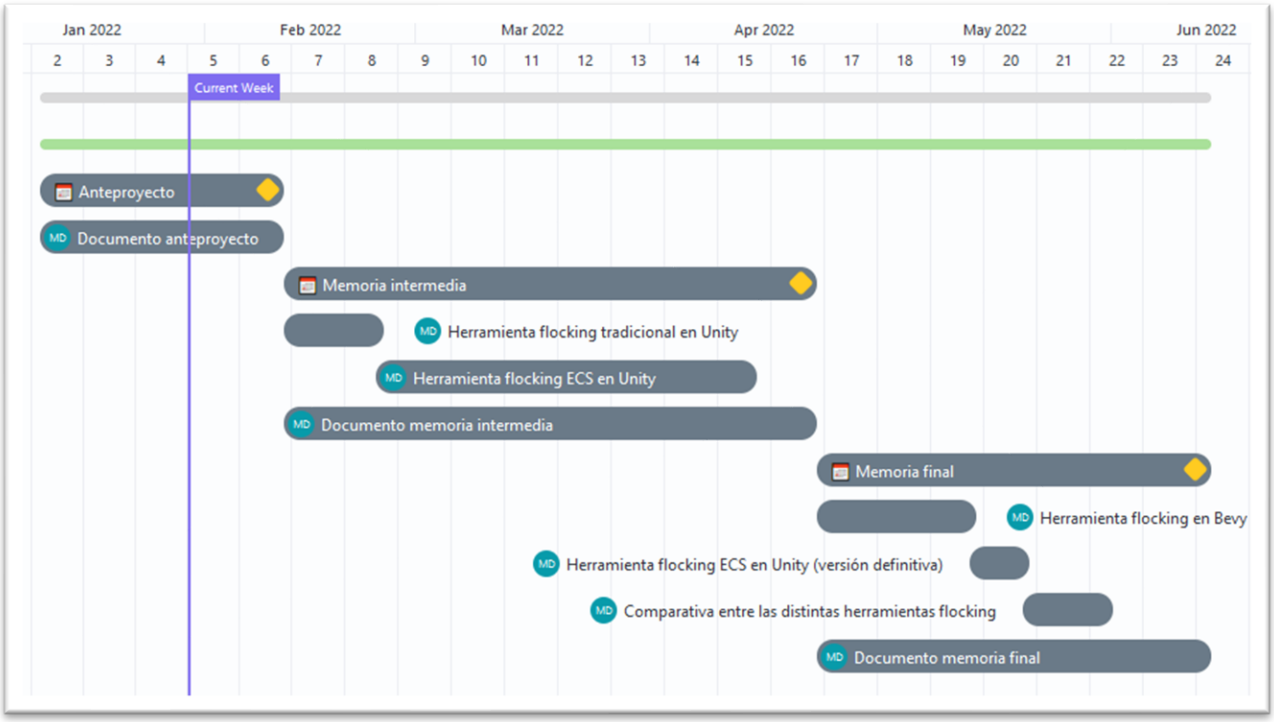


Figura 26. Calendario. Fuente: elaboración propia

6. Desarrollo del proyecto

En este proyecto se hace un análisis comparativo entre un modelo de flocking bajo un paradigma “tradicional” de orientación a objetos con otros modelos basados en ECS, por lo que el primer paso ha sido desarrollar las herramientas necesarias para ello. Se ha comenzado implementando cada steering behaviour por separado para, posteriormente, hacer un modelo en Unity bajo el paradigma de orientación a objetos, otro modelo en Unity utilizando DOTS y otro modelo en Bevy. Adicionalmente, se han planteado optimizaciones para obtener modelos más eficientes. En los siguientes apartados se detallan los requerimientos de cada modelo, la forma en la que se han implementado y las dificultades que han surgido en el proceso.

6.1. Flocking bajo paradigma de orientación a objetos en Unity

Este modelo de flocking se desarrolla en el entorno de Unity más habitual y, para su elaboración, sólo se tendrá en cuenta un espacio bidimensional (2D). Por otra parte, las aceleraciones calculadas en cada steering behaviour se aplicarán sobre un Rigidbody2D, con el objetivo de desarrollar una herramienta que permita tener en cuenta fuerzas alternativas a las calculadas por el modelo, tal y como puede ser la gravedad, el rozamiento o la colisión con otro objeto.

Por tanto, para este modelo inicial cada boid deberá tener como componente un Rigidbody2D sobre el que se aplicarán las aceleraciones calculadas (además del ineludible componente Tranform y un componente del estilo SpriteRenderer para ver la posición del boid en pantalla).

6.1.1. Steerings

Debido a la propia naturaleza de flocking, se desarrollan de forma separada cada uno de los steerings involucrados para, finalmente, combinarlos en un modelo de flocking. A continuación, se detalla la implementación de cada uno de ellos.

6.1.1.1. Búsqueda (Seek)

El steering de búsqueda ha sido desarrollado planteando un caso simple (Figura 27) en el que el boid (la flecha azul) tiene que perseguir a un objetivo en movimiento (la bola roja).

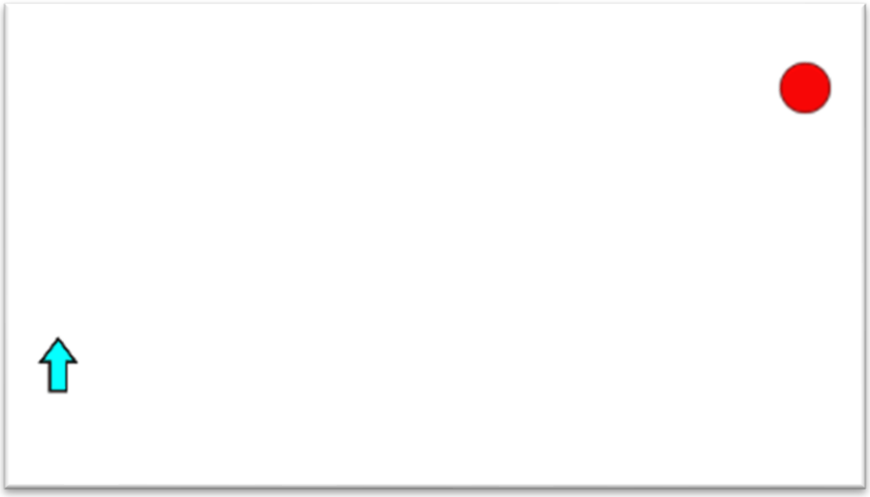


Figura 27. Captura de búsqueda. Fuente: elaboración propia.

Los parámetros de este script (Figura 28) consisten en el Rigidbody del objetivo, una aceleración máxima y, además, una velocidad máxima.

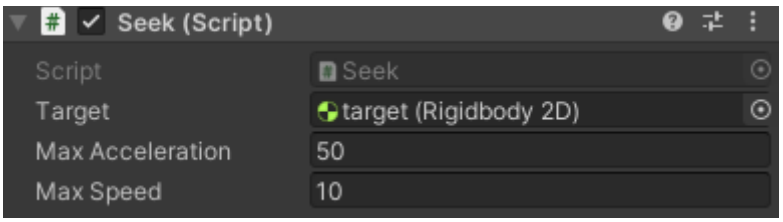


Figura 28. Parámetros del script de búsqueda. Fuente: elaboración propia.

6.1.1.2. Cohesión (Cohesion)

El steering de cohesión ha sido desarrollado planteando un caso (Figura 29) en el que el boid (la flecha azul) tiene que cohesionarse con los objetivos en movimiento (las bolas rojas) buscando el punto medio entre ambos.

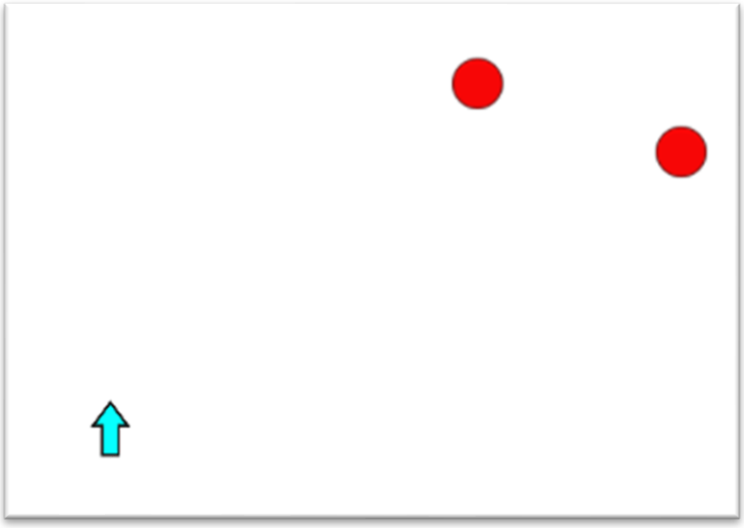


Figura 29. Captura de cohesión. Fuente: elaboración propia.

Los parámetros de este script (Figura 30) consisten en un array de Rigidbodies (los objetivos), el umbral sobre el que empieza a actuar la cohesión, una aceleración máxima y, además, una velocidad máxima.

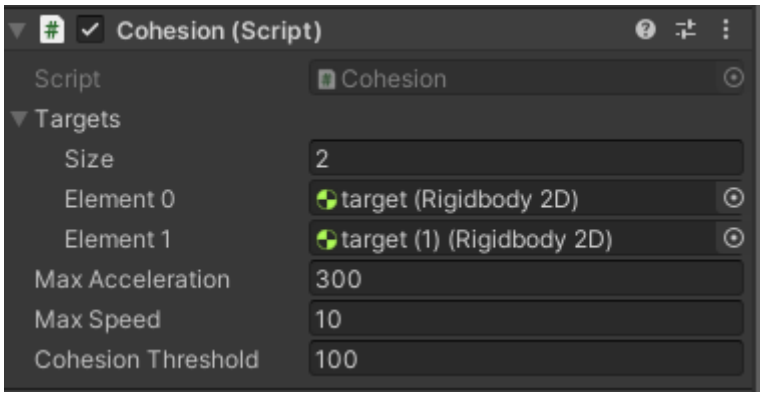


Figura 30. Parámetros del script de cohesión. Fuente: elaboración propia.

6.1.1.3. Alineación (Alignment or velocity matching)

El steering de alineación ha sido desarrollado planteando un caso (Figura 31) en el que el boid (la flecha azul) tiene que alinearse en velocidad y rotación con un objetivo (la flecha negra). Esta flecha negra persigue a la bola roja.

Figura 31. Captura de alineación. Fuente: elaboración propia.

Los parámetros de este script (Figura 32) consisten un Rigidbody (el objetivo), el tiempo para alcanzar la velocidad, una aceleración máxima y, además, una velocidad máxima.

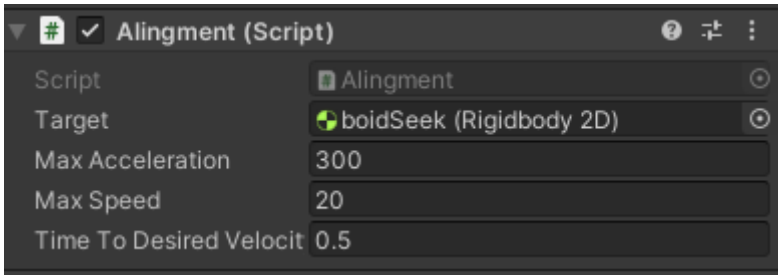


Figura 32. Parámetros del script de alineación. Fuente: elaboración propia.

6.1.1.4. Separación (Separation or repulsion)

El steering de separación ha sido desarrollado planteando un caso (Figura 33) en el que el boid (la flecha azul) tiene que separarse de dos objetivos (las bolas rojas).

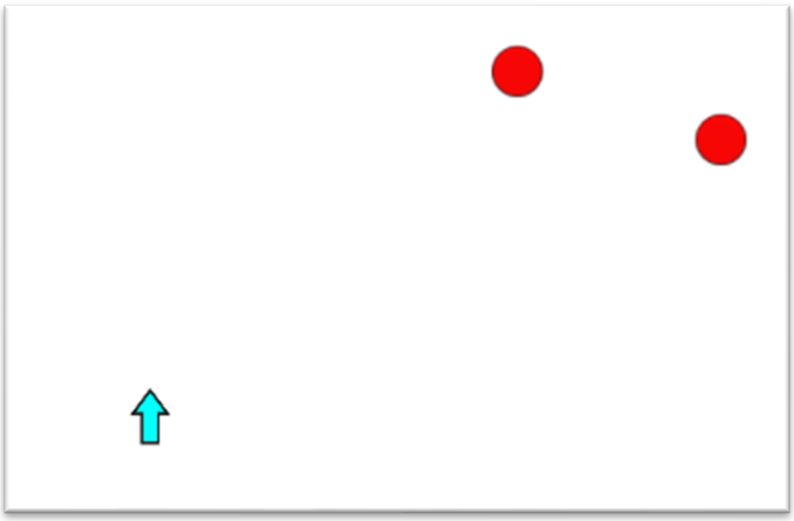


Figura 33. Captura de separación. Fuente: elaboración propia.

Los parámetros de este script (Figura 34) consisten un array de Rigidbodies (los objetivos), la distancia umbral sobre el que actúa la separación, una aceleración máxima y, además, una velocidad máxima.

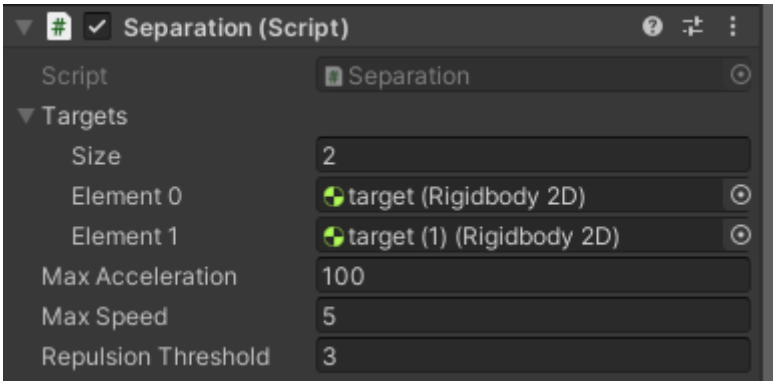


Figura 34. Parámetros del script de separación. Fuente: elaboración propia.

6.1.2. Modelo de flocking

El modelo de flocking se ha planteado desde la perspectiva más básica: incluir todos los steerings previamente desarrollados. Para combinar dichos steerings se ha utilizado un sistema de pesos en el que cada comportamiento calcula un determinado vector (aceleración) y, mediante la combinación de todos ellos, surge la dirección que debe tomar el boid.

En el primer modelo desarrollado cada boid tiene como componente su propio componente de flocking (Figura 35).

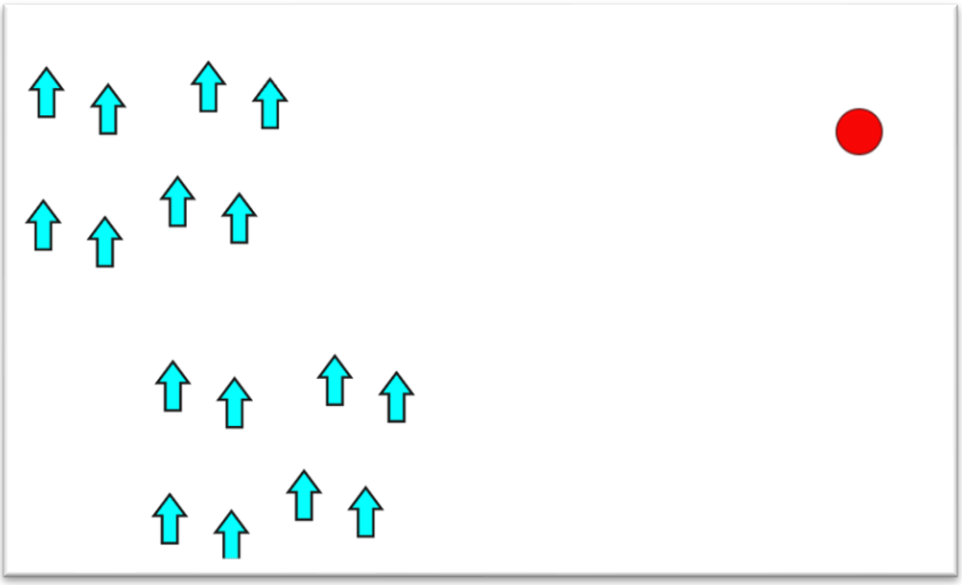


Figura 35. Captura del modelo básico de flocking. Fuente: elaboración propia.

Los parámetros del script incluyen todos los datos vistos hasta el momento (Figura 36), incluyendo también una etiqueta para poder identificar a los otros boids en tiempo de ejecución y, además, un ángulo que determina el área de vecindad. Adicionalmente también se añaden los pesos de cada steering.

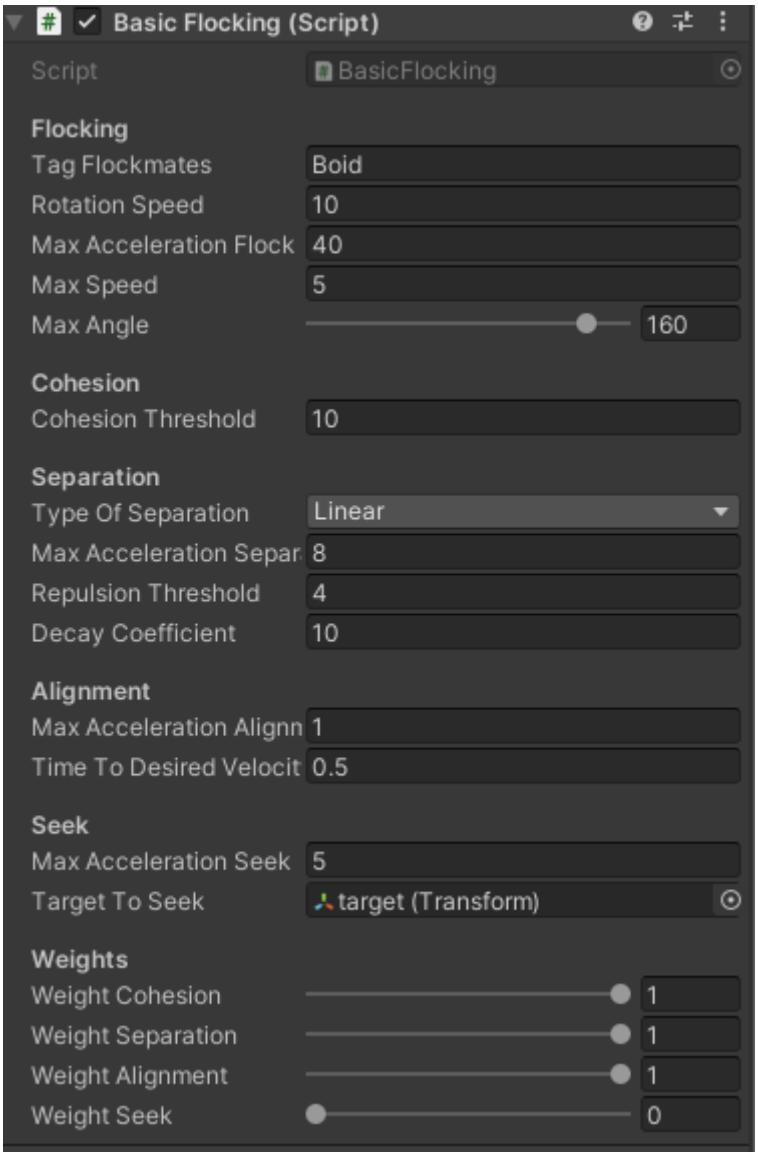


Figura 36. Parámetros del script de flocking básico. Fuente: elaboración propia.

También cabe destacar la inclusión de un parámetro para poder seleccionar el tipo de separación: lineal o cuadrática inversa.

```
if (typeOfSeparation == TypeOfSeparation.Linear)
    repulsionFactor = (repulsionThreshold - distanceToTarget)
    * maxAccelerationSeparation / repulsionThreshold;
else
    repulsionFactor = Mathf.Min(decayCoefficient / (distanceToTarget *
    distanceToTarget), maxAccelerationSeparation);
```

La diferencia entre ambas radica en que se calcula de forma distinta el factor de repulsión que permite posteriormente la fuerza de repulsión al tener en cuenta la

dirección hacia la que está el flockmate. En el caso de la repulsión lineal, se trata la fuerza de repulsión es inversamente proporcional a la distancia con el objeto que la provoca. En el caso de la repulsión cuadrática inversa, la fuerza de repulsión es inversamente proporcional al cuadrado de la distancia con el objeto que la provoca (como la repulsión eléctrica entre dos cargas del mismo signo).

No obstante, para la comparación se ha decidido sólo utilizar la repulsión lineal ya que no se trata de una cuestión realmente significativa en lo que respecta a las diferencias entre orientación a objetos y ECS.

Después de esta versión, se desarrolló otra en la que se instancian un número de boids de forma automática y, además, están todos ellos gestionados por un único script denominado Flocking Manager (Figura 37).

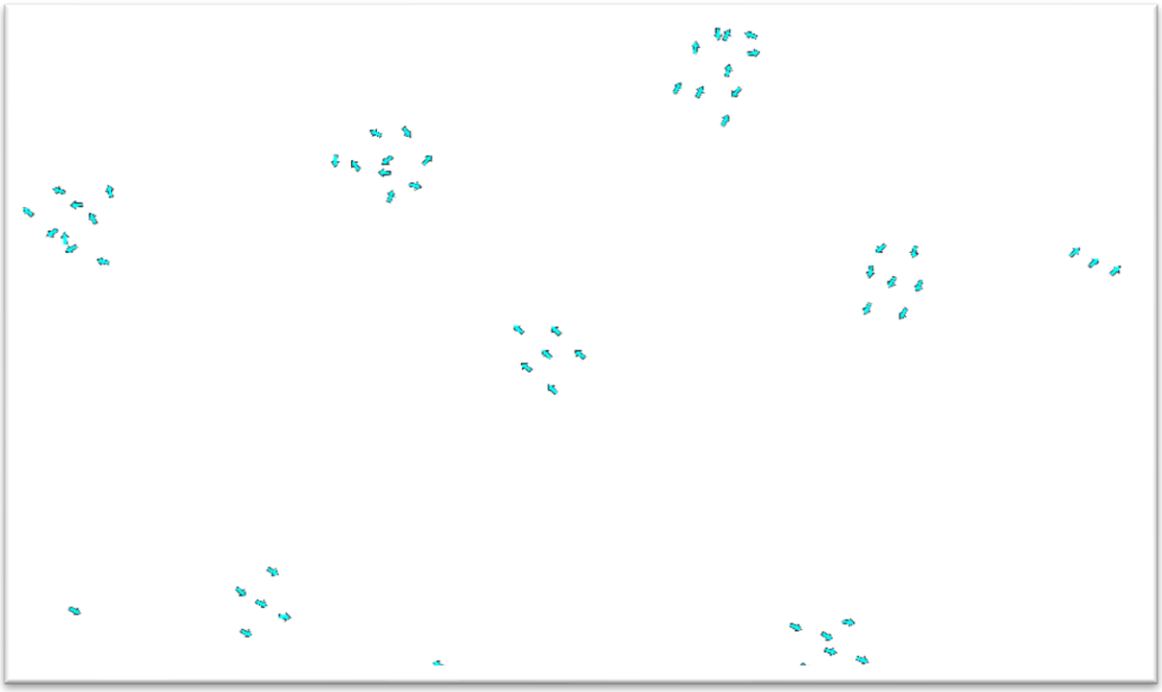


Figura 37. Captura del flocking manager. Fuente: elaboración propia.

Los parámetros del script incluyen el prefab del boid a instanciar (Figura 38). En cuanto al resto, el Flocking Manager presenta las mismas características y parámetros que la anterior versión.

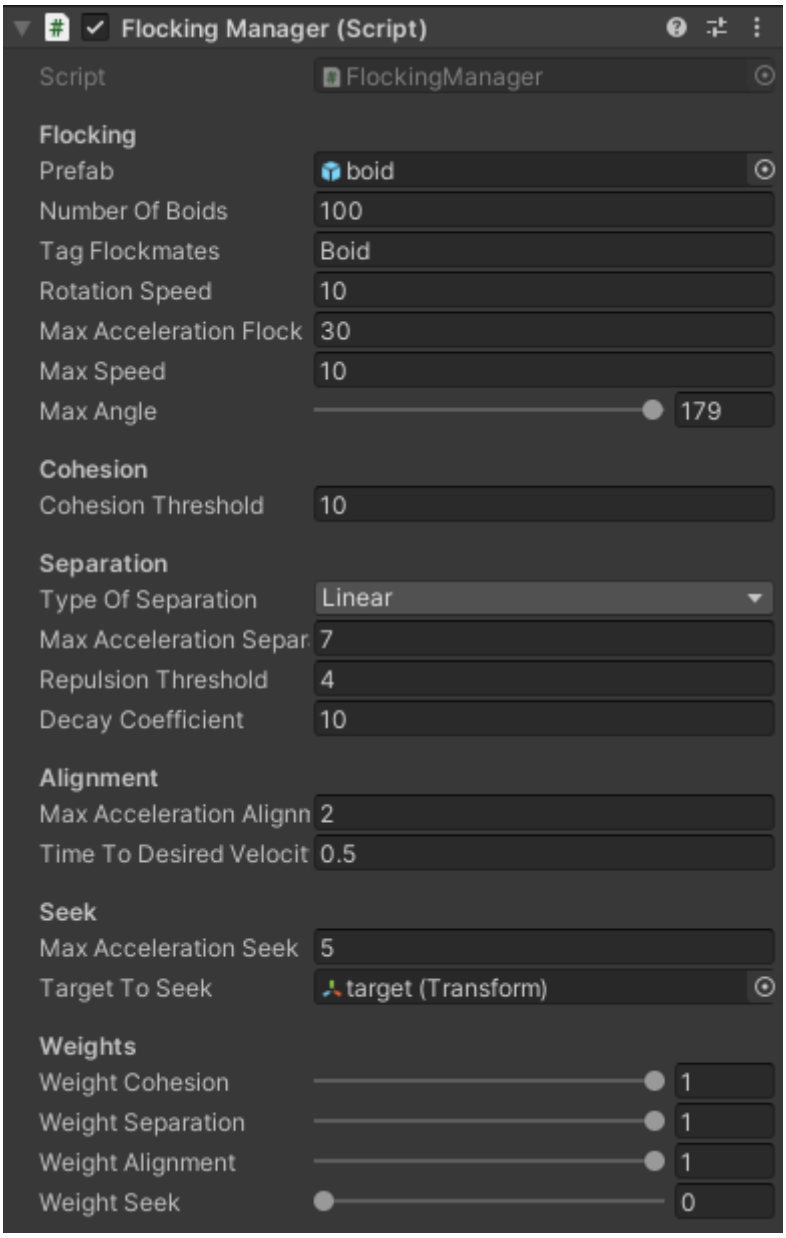


Figura 38. Parámetros del flocking manager. Fuente: elaboración propia.

6.1.3. Conclusiones iniciales

El modelo de flocking desarrollado es capaz de ejecutar correctamente los comportamientos esperados. No obstante, es apreciable un rendimiento muy bajo en cuanto se empieza a superar el centenar de boids en la escena. Probablemente sea

mejorable y optimizable el código escrito ya que se trata de una adaptación de los algoritmos planteados por Millington (Millington, 2009); aunque para esta comparativa inicial es un modelo completamente válido, ya que el objetivo principal es comparar las diferencias con respecto a un modelo construido sobre ECS.

En la siguiente gráfica (Figura 39), se muestra el rendimiento del modelo de flocking obtenido comparando el número de boids procesado (eje X) con la cantidad de fotogramas por segundo al que puede ir el programa (eje Y).

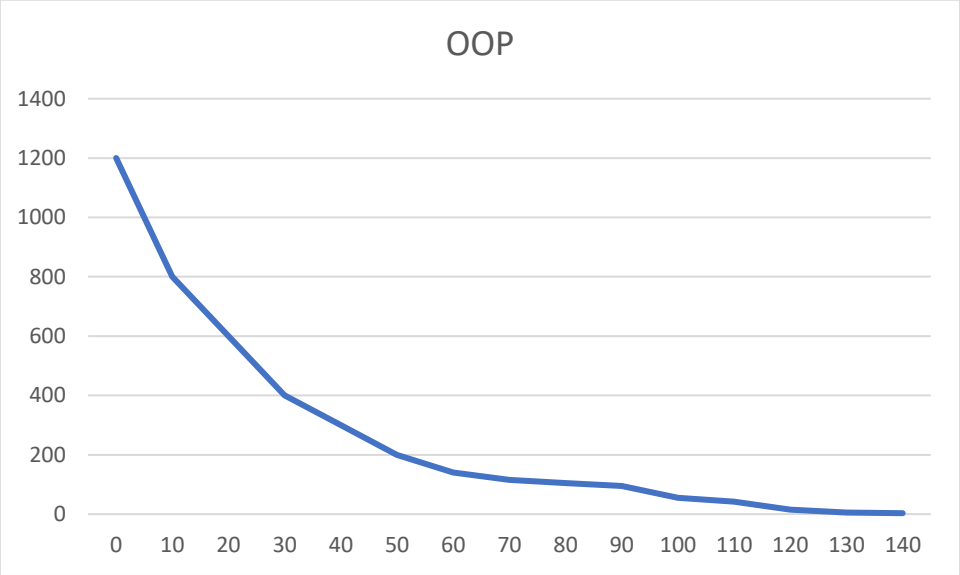


Figura 39. Rendimiento de flocking manager. Fuente: elaboración propia.

El rendimiento cae de forma muy abrupta debido a que el modelo utiliza varios bucles (con anidamientos), lo que implica una situación de crecimiento cuadrático.

Por otra parte, se debe destacar que es posible que la utilización de Rigidbodies esté afectando de forma negativa al rendimiento. Todos los steerings (y consecuentemente el modelo de flocking) pueden ser perfectamente implementados utilizando únicamente el componente Transform aplicando aceleración sobre el mismo y guardando la velocidad lineal en una variable. No obstante, se ha decidido utilizar Rigidbodies debido a que puede ser interesante tener un modelo de flocking que

reaccione a fuerzas externas como la gravedad o a las colisiones contra otros objetos de la escena.

6.2. Flocking bajo paradigma ECS en Unity

Para la implementación del modelo de flocking sobre el paradigma ECS en Unity se han utilizado una serie de paquetes que, en el momento de la realización de este proyecto, aún están en una fase preliminar. Estos paquetes son Burst (un compilador), Entities (una implementación de ECS para utilizar entidades, sistemas y componentes), Jobs (un sistema de jobs para paralelizar procesos) y Unity Physics (un componente de físicas para ECS). Todos ellos forman parte del ecosistema de DOTS.

Cabe destacar que DOTS se encuentra en proceso de desarrollo y está recibiendo actualizaciones de forma constante. En el momento de la realización de este trabajo, se utiliza la última versión disponible, la cual es compatible con la última versión estable de Unity.

Los paquetes que conforman DOTS son completamente susceptibles de ser modificados en el corto y medio plazo. Además de su inestabilidad, una evidencia de ello es la forma en la que se tratan los sistemas de ECS ya que, en versiones relativamente recientes, se utilizaban las clases ComponentSystem o JobComponentSystem. Con las últimas versiones, estas dos formas de implementar los sistemas han quedado obsoletas y actualmente se utiliza la clase SystemBase. No se trata de un cambio cosmético o de una modificación poco relevante, ya que un SystemBase tiene más limitaciones en cuanto a la escritura de código en su interior para forzar que el desarrollador utilice buenas prácticas y, de este modo, mejorar el rendimiento general del sistema.

En lo que se refiere a los componentes, en el paquete Entities se define la interface IComponentData, la cual sirve para crear structs en las que se almacenarán los datos de las entidades.

Finalmente, DOTS incluye una forma sencilla de convertir GameObjects en entidades gracias al script ConvertToEntity. Al añadirlo como componente de un GameObject, este se transformará en una entidad en tiempo de ejecución, convirtiendo los componentes de GameObject (como Transform) en componentes de la nueva entidad. La gran diferencia radica en que GameObject es una clase poco eficiente mientras que una entidad de Entities está optimizada para adaptarse a ECS.

6.2.1. Steerings

Debido a la propia naturaleza de flocking, se desarrollan de forma separada cada uno de los steerings involucrados para, finalmente, combinarlos en un modelo de flocking. A continuación, se detalla la implementación de cada uno de ellos.

6.2.1.1. Búsqueda (Seek)

El steering de búsqueda ha sido desarrollado de la misma forma que en el caso con GameObjects. La diferencia radica en el que el GameObject del boid ahora es un componente, tiene un componente con los datos relativos al steering además de los componentes de transform y física. Además, la lógica del steering se encuentra alojada en un sistema de la clase SystemBase (SeekSystem).

```
public struct BoidSeekComponent : IComponentData
{
    public float maxAcceleration;// = 20f;
    public float maxSpeed;// = 4f;

}
```

Figura 40. Componente de búsqueda. Fuente: elaboración propia.

6.2.1.2. Cohesión (Cohesion)

Igual que en el caso anterior, la diferencia con la versión previa es que el GameObject del boid ahora es un componente, tiene un componente con los datos relativos al steering además de los componentes de transform y física. Adicionalmente, la lógica

del steering se encuentra alojada en un sistema de la clase SystemBase (CohesionSystem).

```
public struct BoidCohesionComponent : IComponentData
{
    public float maxAcceleration;// = 20f;
    public float maxSpeed;// = 4f;
    public float cohesionThreshold;// =10f;

}
```

Figura 41. Componente de cohesión. Fuente: elaboración propia.

6.2.1.3. Alineación (Alignment or velocity matching)

Igual que antes, la diferencia con el caso anterior es que el GameObject del boid ahora es un componente, tiene un componente con los datos relativos al steering además de los componentes de transform y física. Además, la lógica del steering se encuentra alojada en un sistema de la clase SystemBase (AlignmentSystem).

```
public struct BoidAlignComponent : IComponentData
{
    public float maxAcceleration;// = 20f;
    public float maxSpeed;// = 4f;
    public float timeToDesiredVelocity;// = 0.5f;

}
```

Figura 42. Componente de alineación. Fuente: elaboración propia.

6.2.1.4. Separación (Separation or repulsion)

Como antes, la diferencia con el caso anterior es que el GameObject del boid ahora es un componente, tiene un componente con los datos relativos al steering además de los componentes de transform y física. Además, la lógica del steering se encuentra alojada en un sistema de la clase SystemBase (SeparationSystem).

```
public struct BoidSeparationComponent : IComponentData
{
    public float maxAcceleration;// = 20f;
    public float maxSpeed;// = 4f;
    public float repulsionThreshold;// = 10f;

}
```

Figura 43. Componente de separación. Fuente: elaboración propia.

6.2.2. Modelo de flocking

El modelo de flocking sobre ECS (Figura 44) funciona de la forma y permitiendo procesar un número mayor de boids que en el anterior caso.

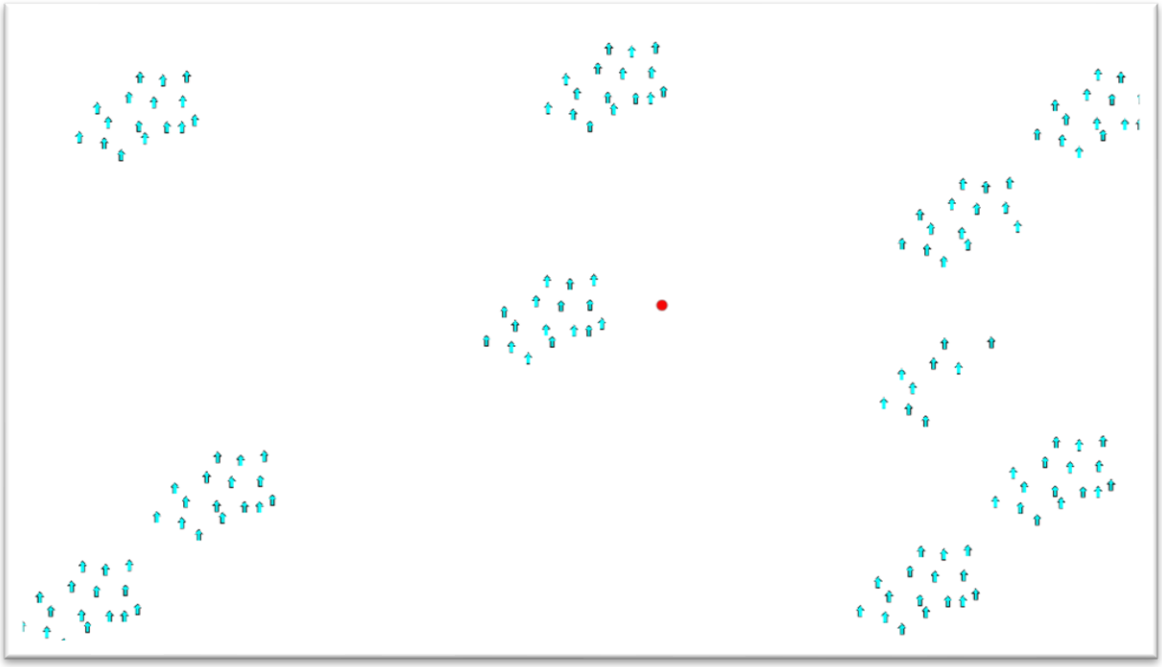


Figura 44. Captura de flocking con DOTS. Fuente: elaboración propia.

El componente creado para este sistema (Figura 45) incluye parámetros como un id, la posición y la velocidad. El motivo de incluir estos datos en este componente es debido a facilitar la forma en la que se acceden a determinados datos del boid en el sistema. Es posible que no sea la forma más óptima de solucionar esta casuística, pero en este punto del proyecto se han priorizado otro tipo de cuestiones relacionadas con la

comparativa entre paradigmas y no centrarse en exceso en la solución tecnológica específica que ofrece Unity con DOTS y el sistema de Jobs.

```
public struct BoidFlockingComponent : IComponentData
{
    public float id;
    public float maxAccelerationFlock;
    public float maxSpeed;
    public float maxAccelerationAlignment;
    public float maxAccelerationSeparation;
    public float maxAccelerationCohesion;
    public float timeToDesiredVelocity;
    public float cohesionThreshold;
    public float repulsionThreshold;

    public float weightAlignment;
    public float weightSeparation;
    public float weightCohesion;

    public float3 position;
    public float3 velocity;

}
```

Figura 45. Componente de flocking. Fuente: elaboración propia.

A la hora de desarrollar el sistema, se han encontrado diversas dificultades relacionadas con DOTS propiamente. Mientras que en sistemas más antiguos basados en JobComponentSystem o ComponentSystem permitían utilizar jobs anidados, con SystemBase no es posible realizar un "Entities.ForEach" dentro de otro "Entities.ForEach". La cuestión es que flocking implica (en principio) que se itere sobre cada boid contra todos los otros boids.

Debido a que se trata de una función de crecimiento cuadrático, SystemBase evita que se puedan producir este tipo de casos, lo que ha implicado que se haya tenido que incluir en el componente variables redundantes como position y velocity ya que,

debido a la imposibilidad de iterar de forma anidada sobre entidades, la solución a la que se ha llegado ha sido utilizar una tabla con todos los BoidFlockingComponent. De esta forma, se hace un "Entities.ForEach" para poder cambiar los valores de cada boid y, dentro de esa iteración, se itera sobre la tabla de componentes para poder calcular todo lo relacionado con otros flockmates.

6.2.3. Conclusiones iniciales

El rendimiento conforme el tamaño del problema crece es superior que el de la anterior versión basada en orientación a objetos (Figura 46), también debido a que los Jobs que se incluyen en el sistema permiten utilizar paralelismo de forma sencilla. Sin embargo, se esperaba un resultado mucho más eficiente del obtenido.

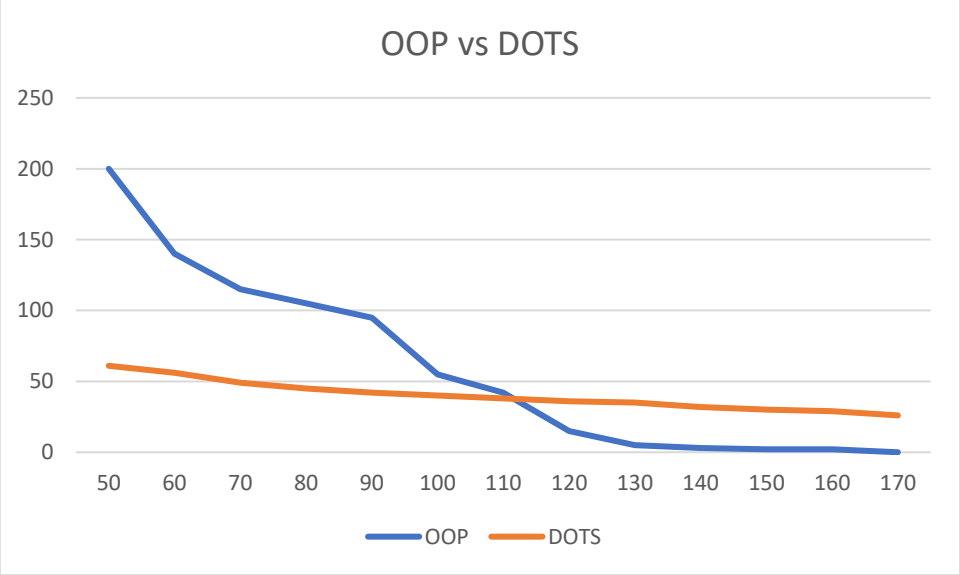


Figura 46. Gráfica comparativa entre los modelos iniciales de OOP y DOTS. Fuente: elaboración propia.

Cabe destacar que DOTS consume una gran cantidad de recursos sólo por estar ejecutando sus sistemas por lo que, por ese motivo, cuando el número de boids es bajo (eje X) la cantidad de fotogramas por segundos que se pueden procesar (eje Y) es notablemente inferior a la del modelo basado en orientación a objetos.

Con el paradigma orientado a objetos y 150 boids se consigue una cantidad de FPS completamente injugable: 2 FPS. Con ese mismo número de Boids, ECS en paralelo se permite ejecuciones a aproximadamente 30 FPS.

Probablemente sea posible mejorar el rendimiento de los algoritmos y hacer un uso más extensivo y eficiente del sistema de Jobs de Unity, pero **con esta implementación inicial, se ha podido validar que ECS demuestra potencial para ser aplicado en un modelo computacional de flocking.**

Son resaltables las diferencias entre los distintos modos de ejecución de Jobs. Estos modos son Run (ejecuta el job de forma inmediata en el hilo actual), Schedule (añade una instancia de IJobEntityBatch a la cola del job scheduler para ejecutarlo secuencialmente) y, finalmente, ScheduleParallel (añade una instancia de IJobEntityBatch a la cola del job scheduler para ejecutarlo de forma paralela).

Para probar estas diferencias, se ha planteado una escena en la que hay el mismo número de boids. Para la ejecución con Run se ha obtenido un Player Loop (el tiempo que se tarda en renderizar un frame) de aproximadamente 54 ms, con Schedule 54 ms y con ScheduleParallel 35 ms. Por tanto, no se aprecian grandes diferencias entre los modos de ejecución secuenciales, pero, por otra parte, la ejecución paralela que facilita el entorno de DOTS reduce el tiempo del Player Loop casi un 40% con respecto a las ejecuciones secuenciales.

6.3. Optimización

La forma académica propuesta por Millington para combinar steerings (Millington & Funge, 2009), implica que cada uno de los steerings se ejecuta de forma independiente y no interrelacionada. Esta aproximación permite que el código sea muy limpio y que se puedan utilizar patrones de diseño que utilicen interfaces que faciliten el desarrollo. Sin embargo, esta forma de abordar el problema implica un desaprovechamiento de los recursos en el plano más práctico.

En los modelos de flocking que se han desarrollado se utilizan 3 bucles distintos, uno para cada steering. De entrada, no sería un problema ya que cada bucle tendría un comportamiento lineal y su crecimiento sería del estilo O(n), por lo que sería equivalente a tener un único bucle (las notaciones asintóticas obvian las constantes por lo que O(n) es considerado equivalente a O(3n)). No obstante, los bucles utilizan varios cálculos que son comunes entre los steerings por lo que, en este caso, se incurriría en un desaprovechamiento de recursos. Estos cálculos tienen que ver con la obtención de la distancia y dirección con respecto a cada flockmate y determinar si un flockmate se encuentra en el área de vecindad o no.

```
foreach (Rigidbody2D boid in boidsRb)
{
    if (boid.Equals(rb)){continue;}

    directionToTarget = boid.position - rb.position;
    distanceToTarget = directionToTarget.magnitude;
    if (distanceToTarget <= Mathf.Max(cohesionThreshold, repulsionThreshold) && CheckNeighborhood(boid))
    {
        averageVelocity += boid.velocity;
        centerOfMass += boid.position;
        count++;
    }

    if (distanceToTarget <= repulsionThreshold)
    {
        if (typeOfSeparation == TypeOfSeparation.Linear)
            repulsionFactor = (repulsionThreshold - distanceToTarget) * maxAccelerationSeparation / repulsionThreshold;
        else
            repulsionFactor = Mathf.Min(decayCoefficient / (distanceToTarget * distanceToTarget), maxAccelerationSeparation);

        repulsionForce = -repulsionFactor * directionToTarget.normalized;
        resultVelocity += repulsionForce;
    }
}

if (count == 0)
    return;
averageVelocity /= count;
centerOfMass /= count;
```

Figura 47. Único bucle para los 3 steerings. Fuente: elaboración propia.

Tanto para el modelo sobre orientación a objetos como en el de ECS se ha aplicado esta optimización para conseguir un único bucle y ahorrar recursos. En ambos casos la mejora ha sido significativa, **mejorando en torno a un 50% los resultados previamente obtenidos.**

6.4. Flocking bajo paradigma ECS en Bevy

Para la implementación del modelo de flocking sobre el paradigma ECS en Bevy se ha utilizado la última versión de este motor, el cual aún se encuentra en fase beta.

Bevy no dispone de un entorno visual amigable para poder desarrollar. Se trata simplemente de una librería que proporciona utilidades para desarrollar videojuegos utilizando el lenguaje de programación Rust, por lo que se debe hacer todo desde código. Para poder probar cada cambio se debe hacer una compilación de todo el proyecto (aunque existen formas de agilizar el proceso).

Las dificultades a la hora de desarrollar el modelo de flocking en este motor no sólo implican el desconocimiento inicial de él y del lenguaje programación, sino que la documentación de Bevy está aún incompleta y su comunidad en Internet es muy pequeña.

El diseño de Bevy fuerza que se trabaje en ECS ya que el desarrollador debe indicar qué sistemas y componentes habrá en el proyecto. Al igual que ocurre con DOTS, las entidades presentan facilidades para realizar iteraciones sobre ellas e incorporar paralelismo de forma sencilla.

Por otra parte, Bevy no incorpora un sistema de “rigidbodies” en sí mismo, sino que existen librerías externas de Rust que permiten implementar sistemas de físicas. Debido a que no hay una opción estándar y que sea rápida de implementar en Bevy, se ha optado por simular las físicas de forma manual, teniendo un componente de velocidad lineal para cada una de las entidades para almacenar este dato. La actualización de las posiciones de cada entidad también se ha programado, la cual tiene en cuenta el valor de la velocidad para modificar la posición.

6.4.1. Modelo de flocking

Para el caso de Bevy, se ha optado ir añadiendo los steerings en un mismo proyecto de forma incremental.

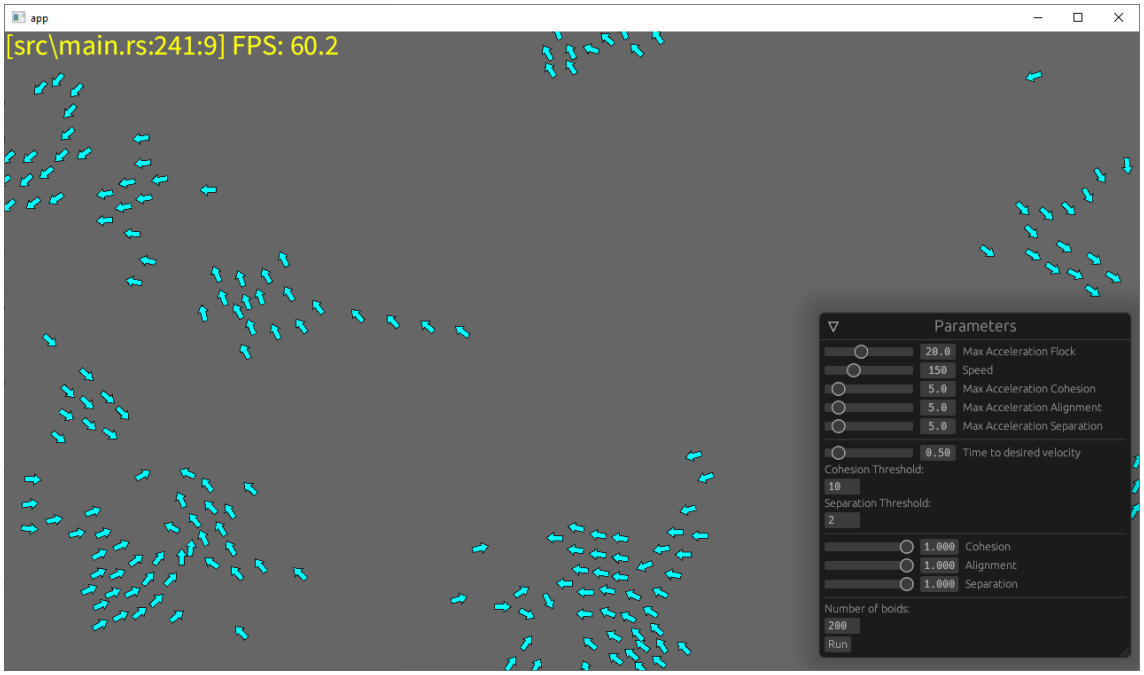


Figura 48.Captura de flocking en Bevy. Fuente: elaboración propia.

Debido a las limitaciones del motor, se ha añadido un panel de parámetros en pantalla para poder modificar los datos sin tener que recompilar el proyecto constantemente. Además, como no hay un entorno en el que poder navegar por el mundo libremente, se ha añadido la lógica de "screen wrap". Dicho de otro modo, cuando un boid sale de la escena por la derecha, volverá a aparecer por la izquierda y, así, la escena no se quedará vacía en ningún momento.

Por otra parte, también cabe destacar que este modelo se ha hecho directamente implementando un único bucle para los 3 steerings, permitiendo hacer comparativas más justas con las versiones optimizadas de los otros modelos.

Este modelo de flocking ha sido el que más complicaciones ha presentado a la hora de realizarse debido a la falta de experiencia tanto en Rust como en el propio motor. Además, el motor no es más que una librería para Rust que no tiene ningún tipo de ayuda visual, se hace todo desde el código. Este hecho implica que se debe tener un

conocimiento relativamente profundo de las funcionalidades del motor para saber qué se necesita en cada momento.

Con Bevy también se ha probado a implementar el programa de forma secuencial y con paralelismo. El rendimiento del programa en forma secuencial consigue mantener una tasa de fotogramas más alta que la versión de DOTS en paralelo, aunque a partir de cierto número de boids la eficiencia de esta versión cae de forma muy pronunciada (Figura 49). En lo que concierne a la versión en paralelo, se trata de la versión más óptima que se ha conseguido en este proyecto. Esto se implementa de forma fácil gracias a las funciones de Bevy para iterar sobre entidades “for_each”, en el caso del procesamiento secuencial, y “par_for_each”, en el caso del procesamiento en paralelo.

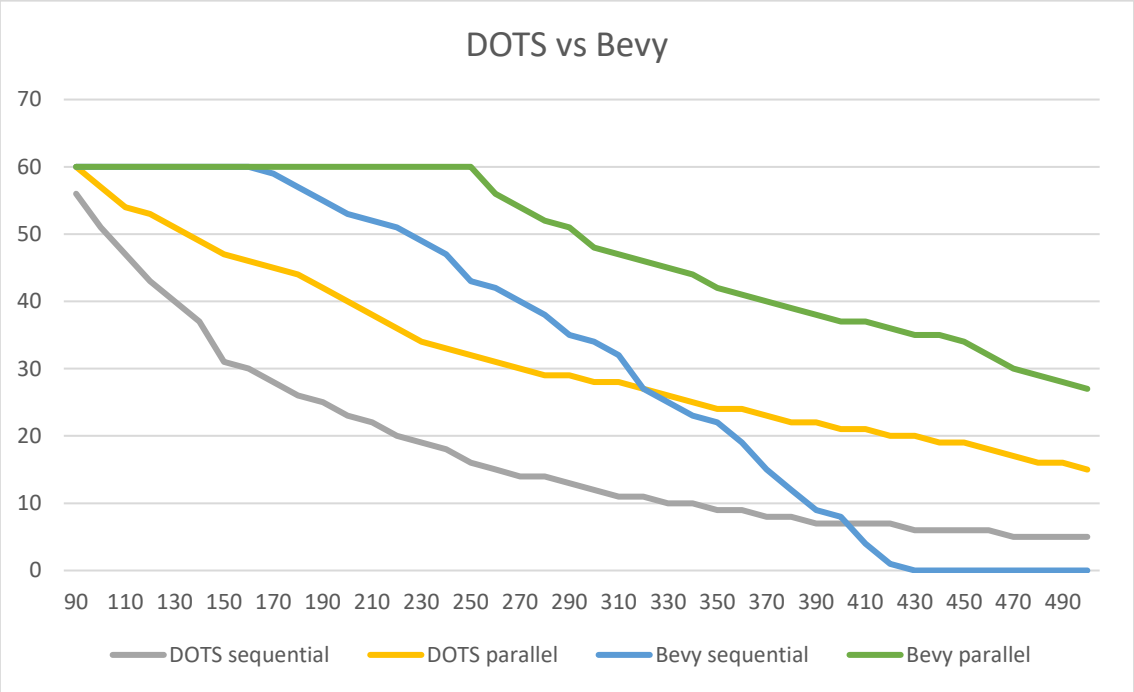


Figura 49. Gráfico comparando el rendimiento de DOTS con el de Bevy. Fuente: elaboración propia.

Cabe destacar que se han encontrado menos limitaciones a la hora de escribir el código que en DOTS y, además, el compilador casi siempre ofrece soluciones eficaces a los errores que pueda haber en el código, facilitando enormemente el proceso de desarrollo. En ese aspecto, DOTS es mucho menos claro que Bevy cuando hay errores o advertencias.

6.4.2. Conclusiones iniciales

El rendimiento de este modelo de flocking es notablemente superior a los vistos en Unity. Mientras los anteriores modelos no podían llegar a 200 boids con un rendimiento aceptable, Bevy mantiene los 60 FPS con ese número de entidades. A continuación, se plantean las hipótesis que explican este hecho.

En primer lugar, Bevy es un motor sumamente ligero. Mientras que cualquier proyecto de Unity (independientemente de su complejidad real) cuenta con una gran cantidad de sistemas que conforman el render pipeline y que pueden tener impacto sobre el rendimiento, Bevy permite que el desarrollador especifique de forma muy concreta qué se va utilizar y de qué forma. A la hora de implementar el modelo de flocking, se ha utilizado lo mínimo y esencial para que este pudiera funcionar.

Por otra parte, hay una diferencia sustancial entre los lenguajes de programación utilizados. Rust es un lenguaje que suele estar en la parte más alta de los rankings que comparan la eficiencia a la hora de resolver problemas de distintos lenguajes de programación (Debian, 2022), mientras que C# tiene resultados ligeramente inferiores.

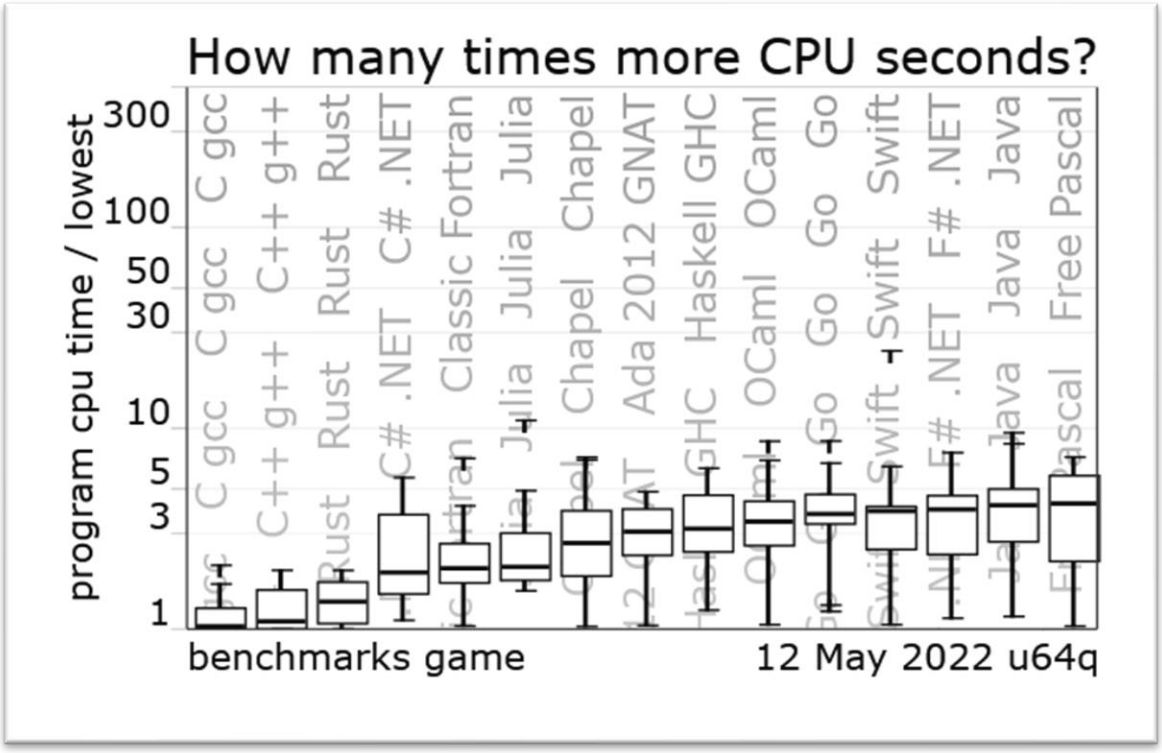


Figura 50. Eficiencia de lenguajes de programación. Fuente: Debian, 2022.

En la Figura 50 podemos ver a Rust en el top 3 de eficiencia, compitiendo casi al mismo nivel de C++ aunque con una mediana de tiempo de procesamiento ligeramente superior. Por otra parte, C# aparece en cuarto puesto (en su aplicación con el framework .NET) aunque con una variabilidad mucho mayor, tal y como se puede apreciar en su percentil.

Rust un lenguaje muy eficiente debido a la utilización de variables dinámicas pero que, contrariamente a C++, no es necesario que el programador gestione cuándo tiene que liberar la memoria. Tampoco utiliza un recolector de basura que afecte de forma negativa al rendimiento (como en C#), sino que usa un enfoque distinto en el que en el momento que el propietario de una variable sale del alcance del programa, se descartará el valor automáticamente. Este concepto de propiedad se establece en el momento de la compilación bajo una serie de reglas.

Finalmente, en lo que se refiere en la forma de trabajar con Bevy, no es (a día de hoy) un entorno agradable en el que poder desarrollar videojuegos comerciales serios. De momento su utilidad se centra en ser un lugar de experimentación que, con el tiempo, pueda acabar convirtiéndose en un motor completamente funcional y en el que puedan trabajar fácilmente gente del perfil artístico o de diseño.

7. Resultados del trabajo

Para poder evaluar las diferencias más significativas de los modelos de flocking desarrollados en este proyecto, se ha optado por una perspectiva empírica en la que se han hecho pruebas de rendimiento.

Para todas las pruebas se ha utilizado un ordenador portátil con un procesador Intel i7-7700HQ, 16GB de memoria RAM y una NVIDIA GeForce GTX 1050. El sistema operativo bajo el que se han ejecutado todas las pruebas ha sido Windows 10.

En las pruebas se ha relacionado el número de boids que se estaban ejecutando con la tasa de fotogramas. No se trata de una prueba exacta ya que los FPS varían a lo largo del tiempo y pueden depender de la ejecución en concreto que se esté realizando, pero permite ver las tendencias aproximadas de cada uno de los modelos a medida que el tamaño del problema crece.

Por otra parte, cabe destacar que en el caso de Bevy la tasa de fotogramas está limitada a 60 FPS debido a cómo se ha construido el programa, por lo que no se verán números por encima de este valor. En Unity no se ha aplicado ninguna limitación de fotogramas.

7.1. Comparativa entre el modelo en orientación a objetos y el modelo en ECS

Los datos obtenidos son los que se muestran en la Figura 51, siendo el eje Y la tasa de fotogramas y el eje X el número de boids. Se analizan los siguientes modelos:

- Programación orientada al objeto inicial en Unity (un bucle por cada steering).
- Programación orientada al objeto optimizado en Unity (un bucle para los 3 steerings).
- ECS secuencial en DOTS (un bucle para los 3 steerings).
- ECS paralelo en DOTS (un bucle para los 3 steerings).
- ECS secuencial en Bevy (un bucle para los 3 steerings).

- ECS paralelo en Bevy (un bucle para los 3 steerings).

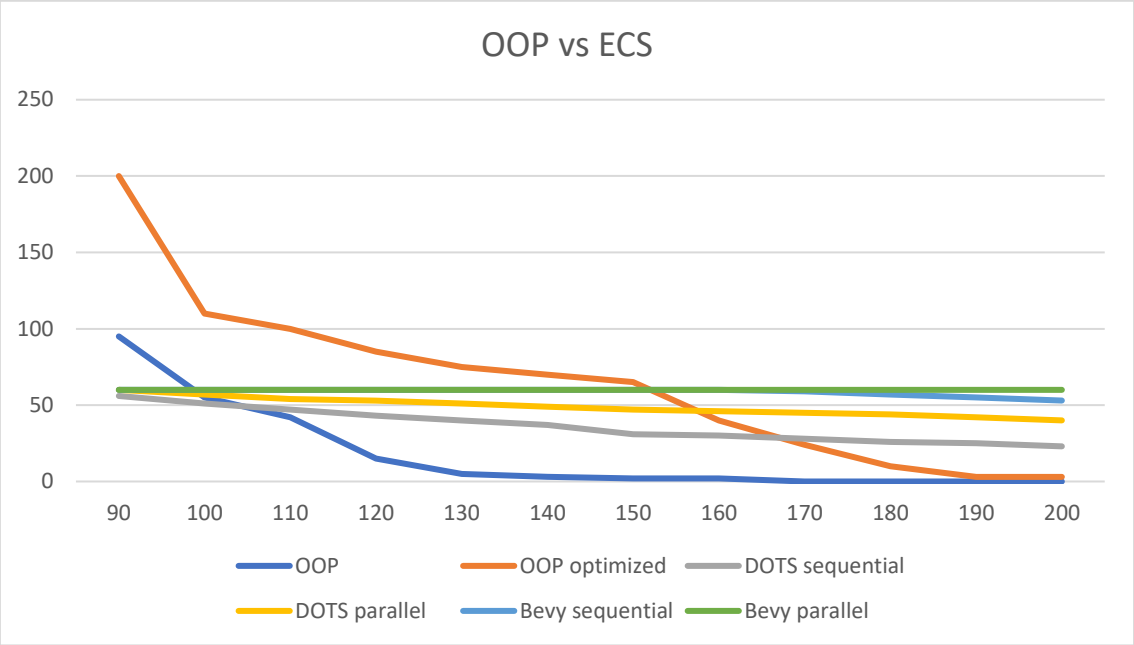


Figura 51. Gráfico comparando el rendimiento de cada versión. Fuente: elaboración propia.

La prueba se ha planteado a partir de los 90 boids, ya que hasta ese número todos los modelos rendían por encima de los 60 FPS.

Si nos centramos en las diferencias entre el modelo de orientación al objeto (OOP) y su respectiva versión optimizada (OOP optimized) se puede observar una diferencia significativa ya que a mientras que a partir de los 100 boids el modelo inicial cae por debajo de 60 FPS, la versión optimizada soporta hasta 150 boids manteniendo esa tasa de fotogramas.

En lo que respecta a las diferencias entre la versión optimizada del modelo de orientación a objeto y los modelos ECS, vemos que hasta los 150 boids la versión es tan funcional como cualquiera de las otras, teniendo una tasa de fotogramas que supera con creces a la de DOTS (en el caso de Bevy, como está limitado a 60 FPS, no sabemos cuál sería su rendimiento máximo).

Si se compara el modelo optimizado de orientación de datos con la versión de DOTS secuencial, hasta los 170 boids se obtiene un rendimiento superior en la versión tradicional. En este punto, se obtienen 24 FPS con el modelo OOP y 28 FPS con el modelo de DOTS secuencial.

No obstante, una de las ventajas de ECS es su facilidad para implementar paralelismo, por lo que se debe tener en cuenta en esta comparación ya que añadir procesamiento en paralelo no supone realmente un tiempo extra de desarrollo. En el caso de DOTS en paralelo, se consigue llegar a 200 boids con una tasa de 40 fotogramas, mientras que con el modelo tradicional se llega a 190 boids con tan solo 3 FPS.

Finalmente, si se compara el modelo de programación orientada al objeto optimizado con cualquiera de los modelos en ECS de Bevy se puede apreciar que Bevy llega a 200 boids manteniendo tasas de fotogramas altas independientemente de si se trata del modelo secuencial o paralelo.

7.2. Comparativa del modelo en ECS en Unity y el modelo en ECS en Bevy

Los datos obtenidos son los que se muestran en la Figura 52, siendo el eje Y la tasa de fotogramas y el eje X el número de boids.

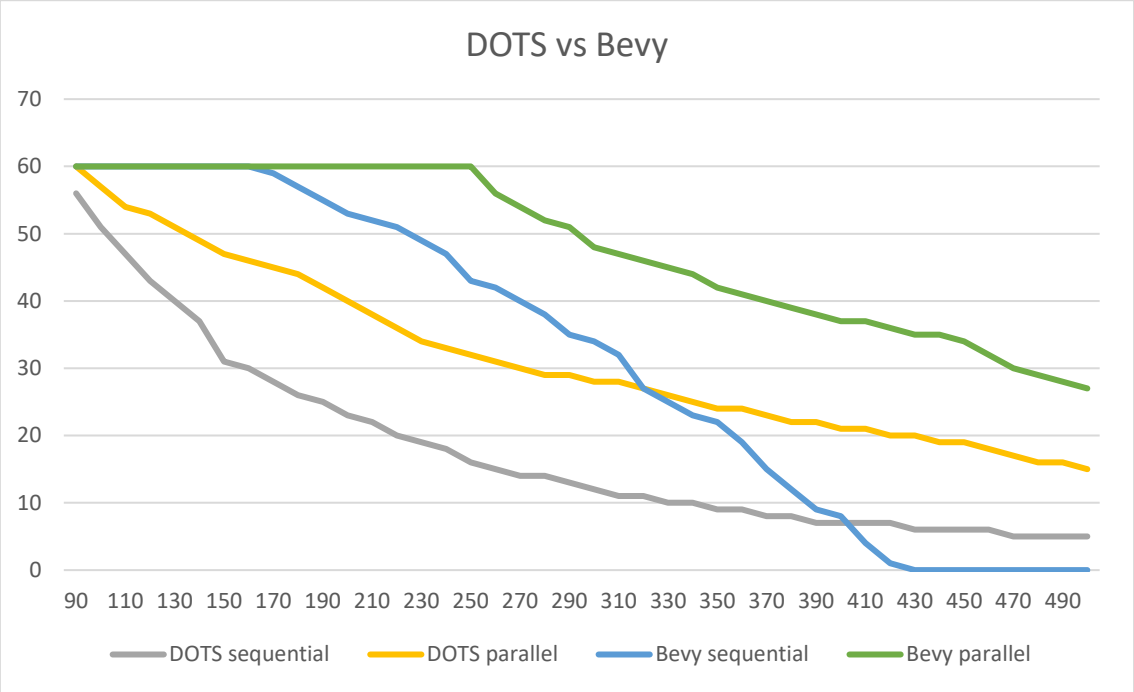


Figura 52. Gráfico comparando el rendimiento de DOTS con el de Bevy. Fuente: elaboración propia.

En este caso, se han tomado datos hasta obtener tasas de fotogramas inaceptables en todos los modelos, considerando ese límite en los 30 FPS.

El modelo con un mejor rendimiento es el de Bevy en paralelo, ofreciendo siempre un resultado igual o mejor que el del resto de modelos. Hasta los 250 boids se llega con una tasa de 60 FPS y, a partir de ese punto, va disminuyendo la tasa de fotogramas progresivamente hasta llegar a los 27 FPS con 500 boids.

Por otra parte, DOTS presenta un rendimiento inferior a Bevy tanto en secuencial como en paralelo. En los dos casos de DOTS, se empieza a obtener resultados por debajo de los 60 FPS desde los 100 boids. Con DOTS en secuencial a partir de 170 boids la tasa de fotogramas cae por debajo de 30 FPS, mientras que DOTS en paralelo llega a ese nivel de frames a partir de 280 boids.

Finalmente, llama la atención del deterioro de Bevy secuencial, ya que a pesar de presentar un rendimiento superior a DOTS hasta los 320 boids, a partir de ese punto su rendimiento cae en picado consiguiendo resultados peores que en DOTS secuencial y

paralelo. No se trata de algo especialmente relevante debido a que, precisamente hasta ese número de boids, Bevy secuencial consigue mantener tasas por encima de los 30 FPS. No obstante, llama la atención esa caída tan abrupta y que, a partir de 420 boids, el programa acaba crasheando, mientras que DOTS consigue funcionar con tasas muy bajas.

7.3. Propuesta de mejora

El mayor problema que presentan los modelos de flocking planteados en este trabajo tienen que ver con que son algoritmos del estilo $O(n^2)$, debido a un bucle anidado que genera este crecimiento de carácter cuadrático. Este bucle dentro de otro bucle se debe a que es necesario ver con cada uno de los boids su relación con el resto de los boids individualmente.

Una posible solución a esta cuestión pasaría por redefinir el concepto de área de vecindad y, además, aplicar un tipo de separación distinto a los que se han planteado en los modelos.

En primer lugar, se dividiría el mundo en una rejilla (Figura 53). Cada cuadrante de la rejilla serviría para definir el área de vecindad de cada uno de los boids. Por tanto, sus vecinos serán aquellos que estén en su mismo cuadrante.

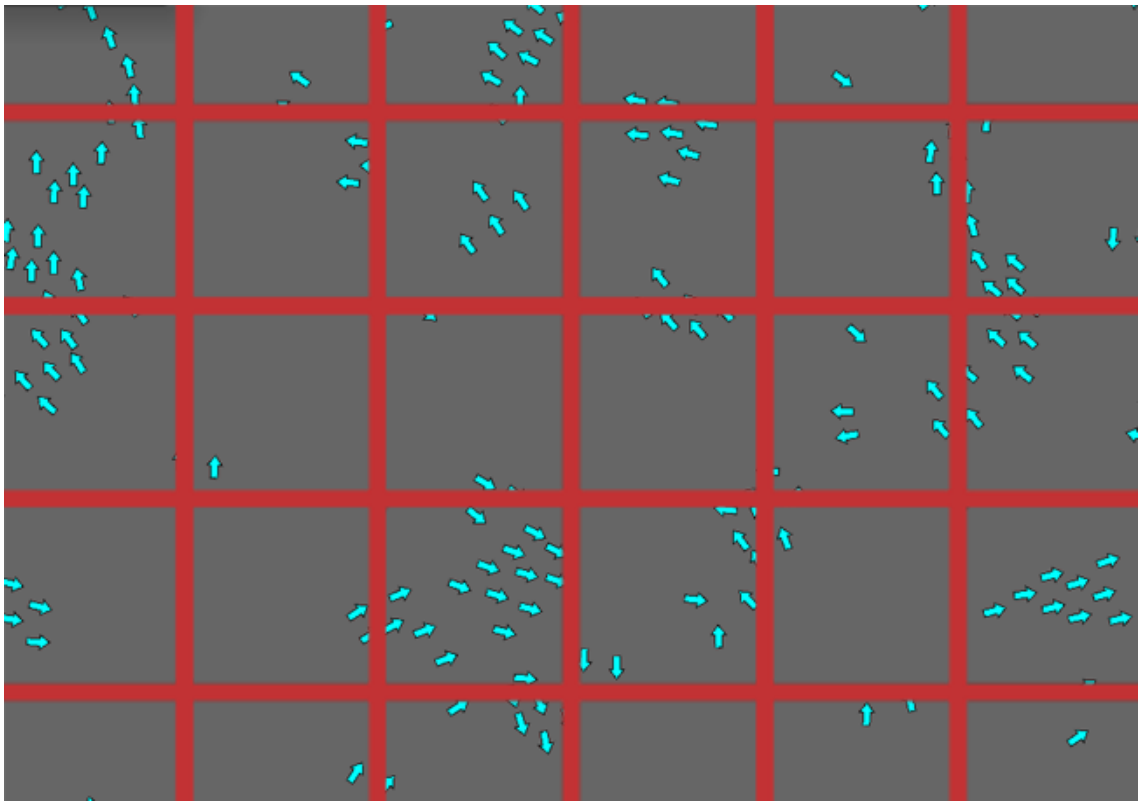


Figura 53. Rejilla de optimización. Fuente: elaboración propia.

El siguiente paso, sería crear una tabla de hash en la que las claves serían los identificadores de cuadrantes y los valores los identificadores de los boids.

El motivo de que hubiera un bucle anidado es que es necesario calcular el centro de masas y la velocidad media de los flockmates que se encontraban en el área de vecindad. Por tanto, existe la alternativa de guardar la suma de las posiciones y las velocidades de los boids vecinos en dos arrays, los cuales se vincularían con cada uno de cuadrantes para, posteriormente, poder dividir entre el número de boids del cuadrante (la cantidad de valores que hay para esa clave) y obtener de este modo el centro de masas medio y la velocidad media.

Así, gracias a la tabla de hash y los dos arrays, sólo quedaría hacer un único bucle en el que iterar por cada uno de los boids y utilizar los datos de estas estructuras sin necesidad de anidamientos. También habría que resolver la separación mediante un método distinto (como buscar una especie de inverso al centro de masas medio) pero

el rendimiento podría mejorar enormemente al pasar a ser un algoritmo del estilo O(n) (a pesar del proceso previo con las estructuras de datos planteadas).

Esta propuesta de mejora permitiría llegar a los límites de boids que se pueden procesar en cualquier modelo (programación orientada a objeto o ECS) pero implicaría ligeras modificaciones sobre el planteamiento inicial de flocking, aunque probablemente sean cambios difíciles de apreciar cuando se están viendo tantas entidades moviéndose en la escena.

8. Conclusiones

Los objetivos planteados inicialmente para este trabajo se han cumplido. Ha habido cierta variabilidad con respecto a la planificación temporal inicial, aunque en líneas generales se han respetado los tiempos establecidos.

Este trabajo planteaba una pregunta inicial de qué era mejor, si flocking sobre un paradigma de orientación a objetos o sobre un paradigma ECS. La respuesta no es de carácter booleano, ya que depende del tamaño del problema.

Si se pretende resolver una casuística en la que haya un número de boids relativamente bajo, el paradigma de orientación a objetos es completamente funcional y obtiene resultados que pueden ser incluso superiores a los del paradigma ECS. Por tanto, en este tipo de casos probablemente lo más adecuado sea utilizar el método tradicional ya que, además, el tiempo de desarrollo no se verá comprometido por tener que adaptarse a una forma distinta de programar.

Contrariamente, en cuanto el problema involucra un número de boids muy elevado, ECS se convierte en una alternativa muy interesante debido a que es capaz de soportar mejor este crecimiento del tamaño del problema. Por tanto, ECS se convierte en una alternativa muy a tener en cuenta cuando se quieren abarcar problemas de tamaños masivos.

Además, a esto se le suma la sencillez con la que se puede implementar el paralelismo en los sistemas preparados para ECS (como DOTS o Bevy), lo que permite que el rendimiento sea notablemente superior a ejecuciones de tipo secuencial. Cualquier videojuego requiere paralelismo para poder funcionar adecuadamente, pero en este caso se hace referencia a la capacidad de que en un algoritmo en concreto se puedan paralelizar ciertas secciones. Si no fuera gracias a estas facilidades de muy alto nivel, sería una tarea mucho más compleja debido a la necesidad de proteger las secciones críticas al estar escribiendo constantemente sobre variables que leerán otros hilos.

Con respecto al motor de videojuegos Bevy, ha sido un aprendizaje intenso y una experiencia satisfactoria. En un primer momento era difícil valorar qué resultados se obtendrían, pero finalmente, se ha podido dilucidar que se trata de un motor sumamente potente.

Posiblemente el factor más determinante para que Bevy disponga de esa potencia sea la utilización del lenguaje de programación Rust, cuyo rendimiento es superior al de C#. No obstante, Rust es un lenguaje con ciertas particularidades que provocan que sea complicado iniciarse en él. Aunque sea más fácil que C o C++, debido a que el programador no tiene que gestionar la liberación de memoria propiamente, sintácticamente tiene una curva de aprendizaje alta si nunca se ha trabajado con variables dinámicas. Para tratar de facilitar este aprendizaje Rust también dispone de tipado dinámico, el cual hace que sea más sencillo para aquellas personas que vengan de lenguajes como Python. Sin embargo, las variables cuentan con la particularidad de disponer de modificadores de escritura para que el compilador pueda realizar optimizaciones en función del uso de cada variable.

Bevy está aún lejos de ser un motor usable en el desarrollo de videojuegos profesionales. Para poder dar ese salto necesita, como mínimo, una interface de usuario en la que poder colocar las entidades de forma fácil y herramientas que faciliten el trabajo de los artistas para animar, iluminar, etc. Ahora mismo Bevy sólo se trata de una librería que permite hacer una compilación desde el terminal. No obstante, cabe destacar que el compilador proporciona información muy específica sobre las advertencias o errores en el código, facilitando enormemente el proceso de desarrollo.

Por tanto, en lo que a la forma de trabajar y productividad se refiere, Unity es un claro vencedor. Sin embargo, su ecosistema DOTS aún tiene bastante trabajo por delante debido a sus cambios constantes e inestabilidad general. En varias ocasiones Unity Technologies ha afirmado que DOTS es el futuro de Unity pero, a título personal, no se

puede evitar la sensación de que no están demasiado enfocados en su desarrollo y que se trata de un frente abierto más en su proyecto empresarial.

En cuanto a las herramientas desarrolladas en este trabajo, están orientadas al ámbito académico y probablemente necesitarían modificaciones para ser completamente usables en código de producción, aunque también es cierto que se han planteado modelos muy genéricos que no estaban pensaban solucionar ninguna casuística en particular, sino que su objetivo era analizar las diferencias entre el flocking desarrollado bajo un paradigma de orientación a objetos y un paradigma ECS.

A pesar de estos puntos de mejora, se valora de forma positiva el desarrollo del trabajo. El calendario establecido se ha seguido mayormente cumpliendo con los objetivos de desarrollo establecidos. Sin embargo, habría sido más adecuado dedicarle más tiempo a la memoria a lo largo de cada iteración, descargando el trabajo de documentación de los momentos previos a finalizar una determinada fase del proyecto.

Finalmente, la propuesta de optimización final no se ha llegado a implementar por falta de tiempo y debido a que esta cuestión ha surgido en los últimos compases del proyecto sin estar planeada inicialmente, pero se trata de una nueva línea a desarrollar que permitiría conseguir una optimización destacable a costa de modificar ciertas cuestiones del modelo de flocking.

9. Bibliografía

Aoki, I. (1982). A simulation study on the schooling mechanism in fish. *Nippon Suisan Gakkaishi*, 1081-1088.

Breeder, C. (1954). Equations descriptive of fish schools and other animal aggregations. *Ecology 35*, 361-370.

Debian. (2022). *Benchmarksgame*. Obtenido de https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

Fabian, R. (2013). *Data-Oriented Design.*

Knuth, D. (1997). *The Art of Computer Programming.* Addison Wesley Longman.

Llopis, N. (4 de 12 de 2009). *Games from within*. Obtenido de Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP): https://gamesfromwithin.com/data-oriented-design

Martin, A. (2007). *Entity Systems are the future of MMOG development - Part 2.* Obtenido de http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/

Masiukiewicz, D., Masiukiewicz, D., & Smolka, J. (2019). Research of an Entity-component-system architectural pattern designed with using of Data-oriented design technique. *Journal of Computer Sciences Institute*, 349-353.

Millington, J., & Funge, I. (2009). *Artificial Intelligence for Games.* CRC Press.

Reynolds, C. W. (1987). Flocks, Herds, and Schools: A Distributed Behavioral Model, in Computer Graphics. *SIGGRAPH Conference Proceedings*, (págs. 25-34).

Reynolds, C. W. (1999). Steering Behaviors For Autonomous Characters. *Game Developers Conference* (págs. 763-782). San Jose: Miller Freeman Game Group.

Unity 3D. (s.f.). *Documentación Unity 3D*. Obtenido de ECS: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/ecs_core.html

10. Anexos

Todo el contenido que se detalla en este apartado hace referencia a la carpeta de Google Drive compartida. Su dirección es la siguiente:

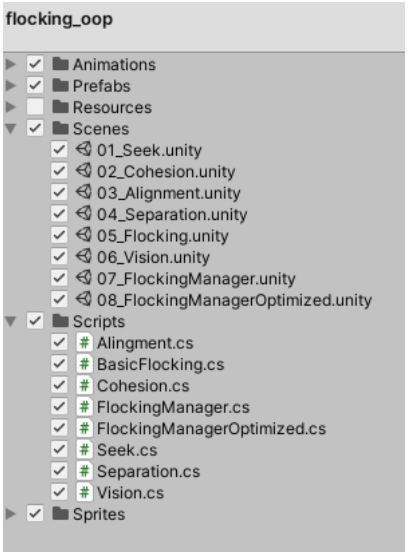
https://drive.google.com/drive/folders/1P-9ICjwZPvx1odwyPNifO0bQ9b5XKG0z

10.1. Código fuente

En la carpeta compartida en Google Drive se puede encontrar una carpeta llamada "CodigoFuente".

10.1.1. Flocking OOP en Unity

El código de esta herramienta se presenta en forma de paquete: flocking_oop.unitypackage. Tan sólo se requiere importar todo el contenido del paquete en un proyecto de Unity vacío. El contenido del paquete es el siguiente:



Lo más destacable son las carpetas de Scenes y Scripts. En la carpeta de Scene está cada una de las pruebas y funcionalidades que se han ido desarrollado de forma incremental. Los modelos que se utilizan para las comparativas en el trabajo son los de las escenas 07_FlockingManager y 08_FlockingManagerOptimized. Por otra parte, en la carpeta de Scripts está el código desarrollado.

El proyecto se ha realizado utilizando la versión de Unity 2019.4.18f1. Se recomienda utilizar esta versión, aunque debería ser compatible con otras versiones recientes.

10.1.2. Flocking ECS en Unity

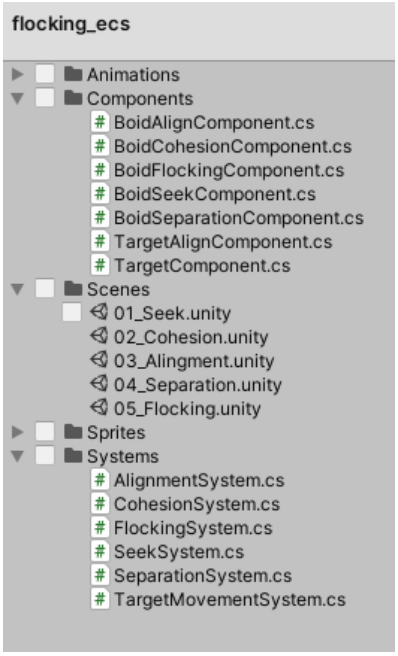
El código de esta herramienta se presenta en forma de paquete: flocking_ecs.unitypackage.

En primer lugar, se deben instalar las dependencias necesarias para que DOTS pueda funcionar. Se debe ir a la carpeta del proyecto y modificar el archivo manifest.json, el cual se encuentra en la carpeta Packages. En este archivo debemos añadir las siguientes dependencias:

"com.unity.dots.editor": "0.8.2-preview",
"com.unity.physics": "0.4.1-preview",
"com.unity.entities": "0.11.2-preview.1",
"com.unity.rendering.hybrid": "0.5.2-preview.4",

Al volver a Unity se deberían importar automáticamente estas librerías. En caso contrario, se deberá ir al Package Manager e instalarlas manualmente.

Posteriormente, se requiere importar todo el contenido del paquete en un proyecto de Unity vacío. El contenido del paquete es el siguiente:



Lo más destacable son las carpetas de Scenes, Components y Systems. En la carpeta de Scene está cada una de las pruebas y funcionalidades que se han ido desarrollado de forma incremental. El modelo que se utiliza para las comparativas está en la escena 05_Flocking. Por otra parte, en las carpetas de Componentes y Systems está el código desarrollado.

El proyecto se ha realizado utilizando la versión de Unity 2019.4.18f1. Se recomienda encarecidamente utilizar esta versión debido a la inestabilidad de DOTS y los paquetes que lo conforman.

10.1.3. Flocking ECS en Bevy

En este caso, se comparte la carpeta del proyecto. Esta en el Google Drive en un archivo comprimido llamado bevy_flocking_project.

El código del proyecto está en la carpeta src, en el archivo main.rs.

Si se quiere probar a hacer una build propia y probar el código directamente, se debe instalar Bevy. Se recomienda utilizar la versión 0.7. de Bevy. Cabe destacar que

también es necesario instalar Rust para poder trabajar con Bevy. Las instrucciones de instalación están en su documentación:

https://bevyengine.org/learn/book/getting-started/

Si se quiere compilar el código, será necesario abrir la carpeta del proyecto desde el terminal y escribir el comando “cargo run”. Esto generará permitirá ejecutar el proyecto y, además, generará una build en la carpeta target. Cabe destacar que la primera compilación tarda unos minutos, pero posteriormente, las compilaciones tardan sólo unos segundos.

Para poder probar el proyecto sin necesidad de instalar Rust y Bevy, se proporciona una build en un archivo comprimido llamado “bevy_flocking_build”. Para ejecutarlo sólo se debe descomprimir y ejecutar el archivo “flocking_project.exe”.

10.2. Instrucciones de uso

Para poder probar los modelos de Unity sólo se debe abrir la correspondiente escena y entrar en PlayMode.

En el caso de los modelos de OOP, en la escena hay un GameObject llamado FlockingManager en el que se pueden variar los parámetros de flocking.

En el caso del modelo de DOTS, en la escena ya hay un número fijo de boids, cada uno con su propio componente con sus datos parametrizables. Para modificar los parámetros de flocking se deben modificar los componentes de cada uno de los boids (se puede hacer a la vez seleccionándolos todos y modificando los datos desde el inspector).

En el caso de Bevy, la build proporcionada incorpora una serie de sliders para modificar los parámetros de flocking. Además, se facilita un botón de “Run” para poder probar fácilmente el programa con distintos números de boids.

10.3. Vídeo del resultado

El vídeo mostrando todos los modelos en funcionamiento se encuentra en la carpeta de Google Drive, en una carpeta llamada Video.