UPC: Facultad de Informática de Barcelona

Cálculo eficiente de puntos de paso por Navigation Mesh con tecnología ECS

Grau en Enginyeria Informàtica

Departamento de computación

Autor: Manuel Fernando Romani Ortiz
Director: Alejandro Ríos Jerez

27/03/2019

Resumen

En este proyecto se pretende implementar un algoritmo de cálculo de puntos de paso en Unity, orientado a mover grandes multitudes de personajes utilizando tecnologías basadas en el paralelismo. Con esto se intenta mejorar varias de las funciones que presenta Unity, así como ampliar la oferta de soluciones que ofrece el mismo de cara a desarrolladores de videojuegos y/o aplicaciones.

Resum

En aquest projecte es pretén implementar un algoritme de càlcul de punts de pas en Unity, orientat a moure grans multituds de personatges fent ús de tecnologies basades en el paral·lelisme. Amb això s'intenta millorar diverses de les funcions que presenta Unity, així com ampliar l'oferta de solucions que ofereix el mateix de cara a desenvolupadors de videojocs i / o aplicacions.

Abstract

The aim of this project is to implement an algorithm for calculating WayPoints in Unity, oriented to move large crowds of characters making use of technologies based on parallelism. This is intended to improve several of the functions presented by Unity, as well as expand the range of solutions offered by the software for developers of video games and/or applications.

Índice

1. Introducción y formulación del problema

1.1 Introducción

Hoy en día, la industria de los videojuegos es una de las que más dinero genera y más ha crecido en los últimos años, sin expectativas de cambio. Así pues, la competencia en este sector se ha visto incrementada enormemente. Es por eso, por lo que las herramientas de desarrollo de videojuegos y tecnologías utilizadas en esta industria están en constante evolución, de forma que todo el mundo quiere hacerse eco de las últimas novedades en este campo.

Un ejemplo de esto sería el motor de videojuegos multiplataforma Unity, sobre la que se llevará a cabo este proyecto. Con el fin de integrar nuevas tecnologías y variar su oferta de opciones, Unity, en sus versiones posteriores a las de 2017 ha integrado la tecnología **ECS** [1], **C# Jobs System** y **Burst compiler** que le permiten mejorar enormemente el rendimiento de cualquier aplicación que las utilice por separado o en combinación.

Entity **C**omponent **S**ystem consiste en una nueva forma de escribir código, cambiando el paradigma de programación orientada a objetos al de programación orientada a datos. **ECS** no piensa en objetos sino en entes (Entity) con sus distintos datos (Component) y sería un sistema (System) el que orquestraria el comportamiento de estos dependiendo de sus datos asociados. Esto facilita la creación de código reutilizable y de fácil contribución por parte de desarrolladores externos. Ligado a esta tecnología, se introduce **C# Jobs System**, que permite a los usuarios escribir código seguro, rápido y paralelizable pudiendo así exprimir al máximo la multitud de threads con los que cuentan los dispositivos actuales. Tanto **ECS** como **C# Jobs System** se pueden utilizar por separado, pero la forma de programar de **ECS** facilita la utilización de la segunda tecnología y es, trabajando en conjunto, cuando obtenemos un mayor resultado. Finalmente, **Burst compiler** genera código máquina altamente optimizado y, al igual que en el caso anterior, si se combina con **C# Jobs System** estas optimizaciones alcanzan su máximo nivel.

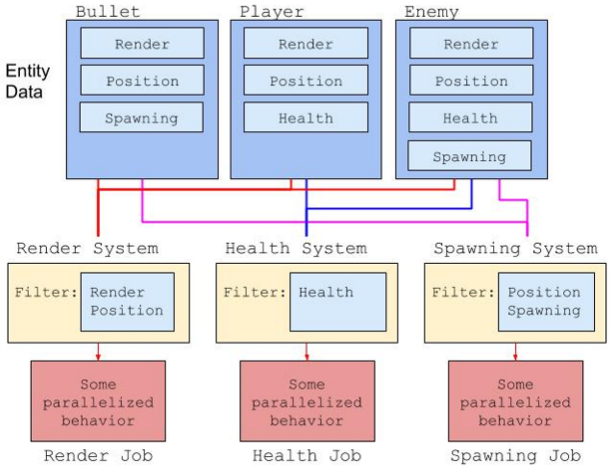


Figura 1. Diagrama de **ECS** integrado con **C# Jobs Systems** (Imagen extraida del articulo **[2]**)

En la **Figura 1** se puede observar cómo se estructura a alto nivel ECS junto con C# Jobs System. Cada entity cuenta con sus respectivos componentes que los definen, por ejemplo, un enemy cuenta con un render que renderiza su malla, una posición, su salud y el punto donde es invocado; para finalmente con los system definir su comportamiento a partir de estos components. En este caso se han definido tres, uno que renderiza todos los entities con el componente render, otro que controla la vida de los personajes que involucra a los entities con el componente Health y finalmente otro que invoca a los entities con el componente spawn y position. Para acabar, el trabajo que se realiza en cada system se paraleliza utilizando Jobs System de forma que el calculado total se ve reducido.

1.2 Formulación del problema

El mundo de los videojuegos sigue progresando por sí solo, y esto incluye cómo no, el motor de videojuegos Unity. Así pues, encontramos novedades como la implementación de nuevos modelos de iluminación tales como el Ray Tracing. Este modelo es conocido desde hace décadas, pero gracias a la inserción de nuevas tarjetas graficas en el mercado (con una unidad de cálculo específico para este proceso) permite que se pueda utilizar en juegos a tiempo real. En Unity por ejemplo, por lo que concierne a la IA (movimiento del personaje y comportamiento en general) se puede observar que, si se hiciera uso de las novedosas tecnologías anteriormente mencionadas, prácticamente la totalidad de tareas ya implementadas en Unity podrían ser reimplementadas de forma mucho más eficiente.

Por ejemplo, una de las muchas funciones típicas y básicas deseadas en todo videojuego es hacer que un personaje vaya de un punto de partida inicial a otro de destino siguiendo un camino mínimo por un escenario estático, y esto, ya está implementado en Unity. El problema reside, y es ahí donde nace este proyecto, en que la solución que ofrece Unity (se estudia con más detenimiento en **[Solución de Unity: NavMesh y NavMeshAgent]**) no se comporta, ni rinde de la forma más adecuada, cuando se aplica a una gran cantidad de personajes (un número mayor a mil). La idea pues, es crear un Asset que, trabajando con una alta cantidad de personajes en escena, permita que todos ellos puedan ir de un punto inicial a uno final comportándose de forma más o menos inteligente(organizándose entre ellos para no estorbarse unos a otros) y tengamos un rendimiento óptimo. Para ello, hay que solucionar los siguientes puntos clave que Unity no consigue, en primer lugar problemas de comportamiento indeseado de los personajes y en segundo lugar, que mejore el rendimiento.

El primer punto se pretende conseguir partiendo de la implementación propia del algoritmo de búsqueda de caminos mínimos A*. Este algoritmo sirve para encontrar el camino mínimo entre dos puntos en un grafo. La ventaja que nos da este algoritmo frente a otros de búsqueda, es que este utiliza un heurístico (función para obtener el nodo más probable a pertenecer al camino mínimo). Además, al ser un algoritmo de búsqueda global se puede garantizar que el camino es el más corto posible. Como añadido al A*, también se le piensan aplicar ciertas modificaciones a la ruta obtenida con cálculos sencillos y eficientes para simular el movimiento que tendrían un conjunto de personas o/y animales en grupo. El motivo de estas modificaciones, se deben a que este calcula el camino mínimo entre dos puntos pero, como vemos en la **Figura 2**, cuando tenemos más de un personaje a la vez, el mismo camino puede ser asignado a más de uno. Por lo tanto, que hay qué decidir que personaje toma ese camino y cual obtiene otro camino diferente y en base a qué se calcula este

camino distinto. Finalmente, el segundo punto, mejorar el rendimiento, se pretende conseguir haciendo uso de las tecnologías mencionadas en [**Introducción**].

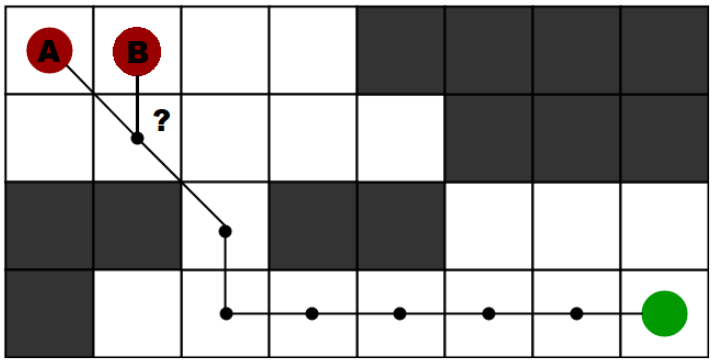


Figura 2. Ejemplo visual del uso de A* con más de un personaje.

2. Alcance

2.1 Objetivos del proyecto

Se pasa mencionar los objetivos principales del proyecto. Estos son:

- Implementar la obtención de una ruta a seguir para un personaje.
- Implementar algoritmos para simular un comportamiento más natural cuando hay multitud de personajes.
- Paralelizar el proceso para hacerlo mucho más eficiente.

Como se va a entrar en detalles más técnicos, aclarar para mayor entendimiento que, todo objeto en Unity, que se pueda visualizar, cuenta con una malla de triángulos que lo forman. Esto es así debido a que los triángulos son la unidad mínima en la que se pueden subdividir los polígonos en dicho motor. Así pues, teniendo en mente los objetivos principales mencionados antes, se enumeran de forma cronológica, el conjunto de sub-objetivos pensados para resolver el problema presentado. Cabe destacar que si los objetivos se cumplieran antes de los plazos establecidos se añadirán más objetivos complementarios.

- Crear un escenario prototipo con diferentes personajes, definiendo un punto de inicio similar para todos y un punto final común.
- Haciendo uso del **Navigation System,** mover los personajes y ver que se comportan de forma indeseada.
- A partir de la triangulación otorgada por la **NavMesh**, implementar el A* para obtener el camino de triangulos del punto inicial del personaje al final.

- Implementar el movimiento los personajes de forma que sigan el camino de triángulos obtenidos con el A*.

- A partir del movimiento implementado, añadir las siguientes funcionalidades:
    - Cuando un personaje vaya a pasar del triángulo A al B por una arista AB, asignarle un punto de la arista AB por el que pasar. Este punto a asignar se realiza siguiendo el método **[Dynamic WayPoints]**.
    - Si todos los puntos de una arista quedan ocupados, buscar un camino alternativo como se muestra en la **Figura 3**.

- Implementar el movimiento local de avatares basado en fuerzas. Para ello se implementarán los algoritmos de **seek** y **Obstacle avoidance**. El primero consiste en hacer que nuestro personaje tenga un destino objetivo (ya quedaría implementado en el punto anterior). El segundo consiste en que el personaje a través de un funnel de visión (simulación de la visión del personaje), detecte el obstáculo (en este caso otro personaje) más cercano e intenta evitarlo.

- Haciendo uso de la tecnología ECS, Jobs System y Burst Compiler, paralelizar el cálculo del movimiento de múltiples avatares en un entorno complejo.

- Publicar un Asset de Unity con esta técnica.

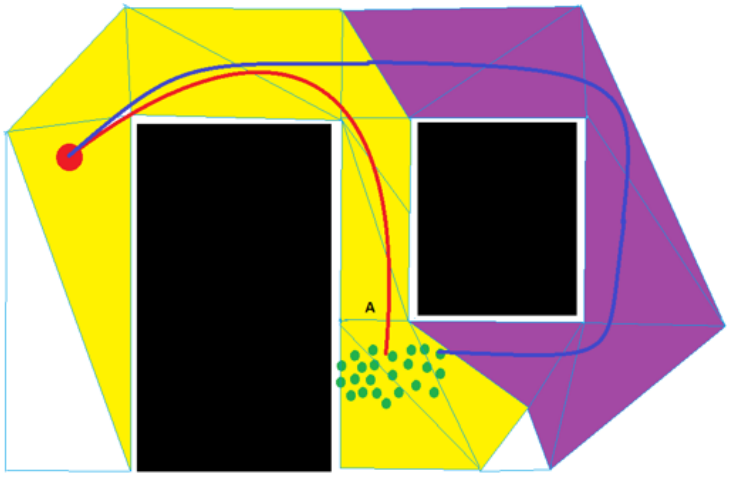


Figura 3. Imagen donde se se muestra cómo solucionar el problema de una arista sin puntos de paso disponibles. Siendo el punto rojo el destino establecido y los puntos verdes personajes, se puede observar como por la arista A no pueden pasar todos los personajes y los personajes que disponen de triángulos vecinos con aristas libres recalculan su trayecto.

2.2 Límite del proyecto

Una vez establecidos los objetivos principales del proyecto, se pasa a definir cuál es el alcance de este mismo. Posterior a esto, se echara un vistazo a los posibles obstáculos que pueden aparecer durante el transcurso del desarrollo del proyecto.

Como se ha mencionado en secciones anteriores, se pretende partir de cero (exceptuando la obtención de la malla de triángulos de la superficie) debido a que es imposible acceder a la implementación de Unity y modificarla. También esto se ha pensado así, ya que de esta forma, se dispondrá de un control y customización total sobre el proyecto que se va a desarrollar.

Aclarar también, que este no es un proyecto de simulación de comportamiento grupal únicamente. Es parte del proyecto que los personajes que compartan un mismo destino puedan gestionar sus rutas de forma que visualmente parezcan moverse de forma organizada. Sin embargo, esto se intentará resolver con algoritmos eficientes y soluciones ingeniosas, y no gastando demasiados recursos computacionales en ello. El motivo de esto es que se intenta crear una aplicación que pueda mover a miles de personajes sin pérdida de rendimiento, así que es primordial gestionar al máximo los recursos disponibles.

2.3 Posibles obstáculos

En los subapartados de a continuación, se expondrán los posibles obstáculos con los que nos podemos encontrar a la hora de desarrollar este proyecto y las soluciones o alternativas a estos. Cabe destacar que estos son los obstáculos que se estiman en una fase muy temprana del proyecto, por lo que probablemente surjan nuevos en el futuro.

2.3.1 Errores de programación

Como en todos los proyectos informáticos, el primer obstáculo con el que se encuentra todo desarrollador, son los errores de programación. Estos se pueden dividir en dos grupos que llamaremos, errores básicos y errores de alto nivel. Con el primero se hace referencia a fallos que imposibilitan la compilación de los diferentes scripts o produzcan un error que haga abortar la ejecución completa del programa. Por otra parte, cuando se habla de errores de alto nivel, se está hablando de los errores en código, que a pesar de que no interrumpen la ejecución entera del programa, si que alteran su esperado comportamiento de forma parcial o total.

La forma de solucionar los errores básicos, siendo esta la que se utiliza prácticamente siempre, es trackear, a partir de la información aportada por el compilador y varios chivatos, la localización de estos errores y corregirlos manualmente. Sin embargo, para los errores de alto nivel, al no tener información del compilador (ya que no hay errores de compilación), hace falta hacer un gran ejercicio de abstracción para entender porqué ocurre este comportamiento indeseado y poder reparar el error. Una forma de agilizar este proceso de entendimiento para los errores de alto nivel, es reproducir estos errores modificando parámetros de entrada del programa y/o mostrando los valores de todas las variables implicadas.

2.3.2 Encontrar un camino mínimo

En el apartado [**Algoritmos de búsqueda**] se hablará de que A* es un algoritmo de búsqueda de caminos mínimos, sin embargo, el camino mínimo, para este proyecto de momento, se calcula usando la distancia entre baricentros de triángulos. Como resultado, se obtiene el camino mínimo que tendría el personaje si este pasará por el baricentro de los triángulos forzosamente.

Esto ocasiona que en ciertas situaciones donde hay un gran obstáculo que sortear, el personaje no recorra el camino más corto realmente. De momento, se ha implementado una solución, que cuando se aplica el A*, calcula la distancia a otro triangulo teniendo en cuenta la posición del personaje y moviendolo virtualmente durante la ejecución del A*. Por el momento, para situaciones en las que el personaje o el destino están cercanos a las aristas de otros triángulos, obtenemos un cálculo más preciso del camino mínimo que si usaramos el baricentro.

2.3.3 Documentación disponible de ECS

Aun con los años que han pasado des del lanzamiento de **ECS**, sigue habiendo muy poca documentación por parte de Unity en comparación con el resto de funcionalidades implementadas que ya llevan más tiempo en la plataforma. En adición, si la documentación de Unity resulta bastante escasa, la que se puede encontrar por el resto de la red, lo es aún más.

Este problema podría dificultar enormemente la implementación de la tecnología **ECS**. Aun así, se ha visto que se puede implementar **ECS** de forma híbrida o pura. Obviamente, una implementación híbrida de **ECS** no sería tan efectiva como una de pura, pero podría ser una buena opción si el proyecto se queda parado por este punto.

3. Contexto

3.1 Areas de interes

Desarrollando el proyecto como se está haciendo, es decir, sobre un motor de videojuegos, queda claro que una de las principales áreas de interés de este proyecto es la de los videojuegos. Este proyecto podría ser utilizado en cualquier videojuego como de los que se habla en la sección [**Algoritmos de comportamiento grupal**].

Gracias a la versatilidad que ofrece Unity, si el proyecto se publicase en forma de Asset, este podría ser utilizado en cualquier plataforma. Esto nos abre las puertas a, por ejemplo, utilizar el proyecto en una aplicación de realidad virtual para mover a un gran cúmulo de gente de forma realista. Además, la gran eficiencia de la que dispondrá ayudará a disminuir el coste de la IA y poder centrar recursos en otras funcionalidades.

3.2 Partes implicadas

A continuación, en las siguientes secciones, se detallarán las diferentes personas que pueden estar tanto implicadas como interesadas en este trabajo.

3.2.1 Diseñador, programador, tester y jefe de proyecto

Para el caso de este proyecto, la primera persona interesada en este, será el que lo realiza. Para ello, los roles de diseñador, programador, tester y jefe de proyecto tendrán que ser asumidos por una misma persona.

3.2.2 Director y codirectora del proyecto

El director Alex Ríos, será la persona que supervise el proyecto para su ideal finalización. La codirectora Nuria Pelechano, al igual que el director, aportará, nuevas ideas y otros puntos de vista (más experto) en las reuniones que se vayan realizando a lo largo de la realización del proyecto.

3.2.3 Usuario final

El usuario final será cualquier persona que vaya, o esté interesada, en desarrollar un videojuego o aplicación usando el motor de Unity. Pensado especialmente para aquellas aplicaciones que requieran de muchos personajes en escena y/o requieran un movimiento en grupo más natural.

4. Estado del arte

Finalmente, una vez contextualizado y formulado el problema en este apartado se intentará hacer un estudio más en profundidad de la solución directa que ofrece Unity, así como porqué se ha decidido implementar el proyecto prácticamente desde cero (exceptuando la obtención de la malla). Además, se analizará de qué otras maneras Unity nos permite obtener los resultados similares a los que deseamos. Estas otras vías, tienden a ser proyectos independientes, por lo que veremos el ejemplo de uno en concreto. Por otra parte, el proyecto se estructura básicamente en usar un algoritmo de búsqueda, simular el comportamiento natural de un conjunto de personajes en grupo y resolver esto de forma eficiente. Así pues, se verán varias de las opciones de las que disponemos a día de hoy para cada uno de los puntos de la estructura del proyecto.

4.1 Solución de Unity: NavMesh y NavMeshAgent

Como era de esperar de un motor de videojuegos, Unity cuenta con un ***Navigation System.*** Este permite, que cualquiera que esté desarrollando un videojuego, pueda definir el espacio por el que se van a mover los personajes de la aplicación y el destino de los mismos de forma muy sencilla. En el artículo **[3]**, se explica de forma bastante detallada cada uno de los componentes parametrizables de este sistema, sin embargo, en esta sección nos centraremos en la **NavMesh** y **NavMeshAgent**, pues son las que están relacionadas directamente con el proyecto.

Así pues, como dato más importante, la **NavMesh**, contiene toda la triangularización de la superficie transitable. Gracias a las diferentes opciones que nos aporta Unity en el ***Navigation System***, se pueden definir qué objetos de la escena son obstáculos (por lo que estos quedarían fuera de la triangulación del NavMesh), que coste hay al caminar sobre la superficie de ciertos objetos (para decidir qué ruta debe tomar el personaje cuando hacemos uso de un **NavMeshAgent**), entre otras opciones. En la **Figura 5** se puede observar la malla de triángulos generada a partir del escenario de la **Figura 4**. Como se puede ver, no todas las figuras representadas son triángulos, aun así Unity las divide internamente en estos.

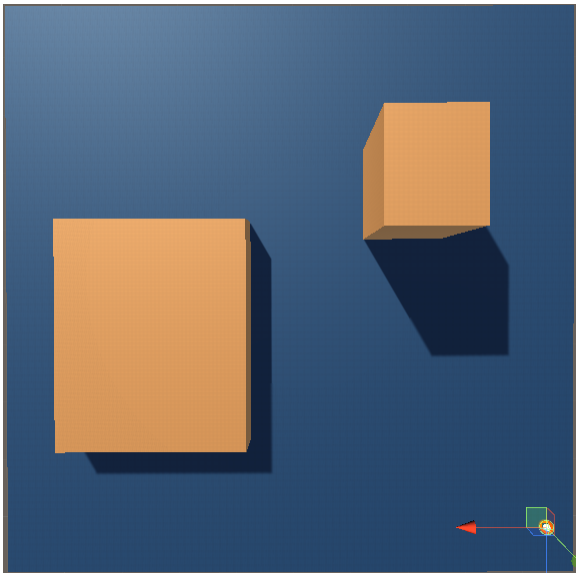


Figura 4. Escenario prototipo.

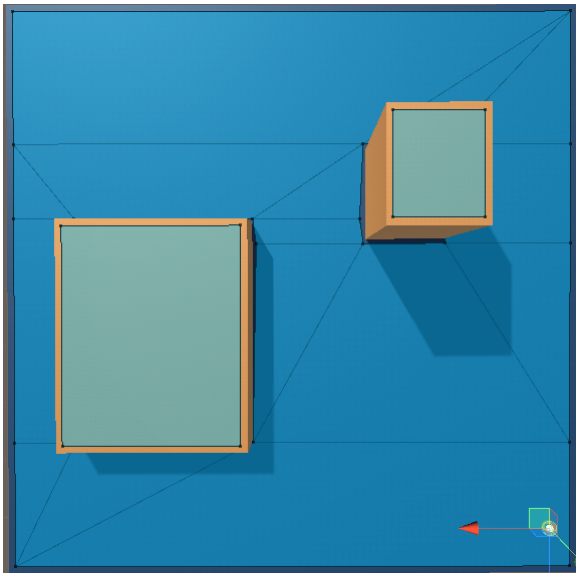


Figura 5. Escenario prototipo resaltando la NavMesh.

La **NavMeshAgent** por su parte, es un componente que se puede adherir a cualquier GameObject (Objeto básico en Unity). Al añadir este componente a nuestro personaje, podemos definir algunas características de este para que la **NavMesh** los tenga en cuenta. Características como la altura del personaje, su velocidad, su anchura, entre otras, además de poder establecer un destino para el personaje y que esté de forma automática se dirija a él recorriendo la **NavMesh**.

Esta solución nos permite parametrizar el movimiento de nuestro personaje de forma muy sencilla e intuitiva, pero cuando tenemos, por ejemplo, más de 10 personajes en escena, ocurre la situación de la **Figura 6**. Como se puede ver, todos los personajes se pelean por pasar por un mismo punto, cuando realmente hay más espacio transitable disponible. Esto ocurre, porque cuando el **NavMeshAgent** calcula el camino mínimo entre el personaje y el destino establecido, se olvida de los triángulos y mira solo puntos de la malla de triángulos. Por lo tanto, al estar todos los personajes situados más o menos en un mismo punto de inicio, casi todos los personajes tienen el mismo listado de puntos por los que pasar. El resultado es que todos los personajes acaban alineándose y peleandose por los puntos del camino mínimo. Además, esta solución está programada haciendo uso de la programación clásica de Unity.

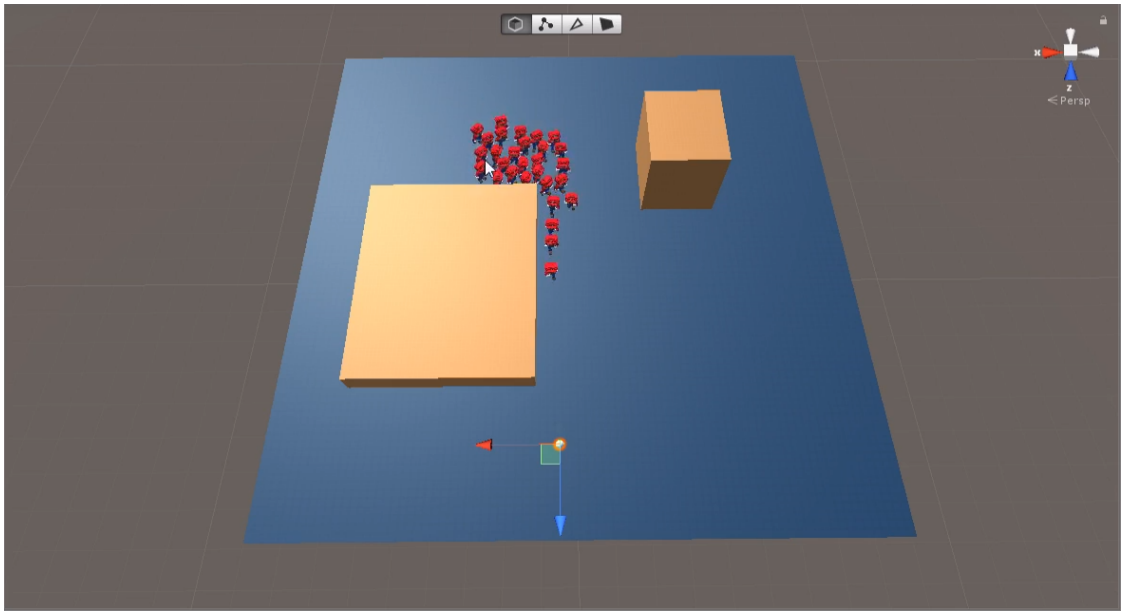


Figura 6. Escenario prototipo donde se puede ver cómo se aglomeran todos los personajes al pasar por un mismo punto y acaban alineándose. El punto objetivo es la circunferencia inferior.

Como se quiere encontrar una solución que replique los que hace el ***Navigation System*** pero con un comportamiento y rendimiento mejor para gran cantidad de personajes, se investigó qué se puede aprovechar de la solución ya existente. El problema es que no hay forma de acceder al algoritmo que utiliza el **NavMeshAgent** para el camino mínimo, ni el camino mínimo de triángulos (ya que solo utiliza puntos y sabiendo los triángulos se podrían distribuir de forma más homogénea los personajes). Por consiguiente, se ha optado por utilizar solo la triangulación que nos da la NavMesh como punto de partida y a partir de ahí, implementarlo todo desde cero y así tener un control total sobre el proyecto y poder añadir las nuevas tecnologías que mejoran la eficiencia de Unity [**Introducción**].

4.2 A* Pathfinding Project Pro

Dejando de lado la solución de Unity, nos topamos con que existe un proyecto llamado, A* Pathfinding Project Pro, muy parecido al **Navigation System,** pero en esta ocasión realizado de forma independiente por Aron Granberg **[4].**

Este Asset al contrario que el **Navigation System**, es de pago y permite cambiar la malla de triángulos (sobre la que se haría la búsqueda del camino mínimo) a malla de hexágonos o cuadrados. A parte de esto, en sus ultimas versiones, esta programado de tal forma que se aprovecha al máximo el multithreading.

El problema de esta solución, es prácticamente el mismo que el del **Navigation System**, y es que no está pensado para que sobre la malla circulen varios personajes a un mismo destino. Aun así, es una muy buena alternativa ya que se presenta como una versión mejorada de **Navigation System**.

4.3 Implementar una solución propia

Finalmente la última opción que se presenta sería implementar una solución propia. Esto es debido a la escasez de variedad en las opciones que Unity ofrece para solucionar el problema formulado (aparte del **Navigation System**). Si se quiere mover a un personaje en Unity sin utilizar el **Navigation System,** solo nos queda la implantación de toda la vida. De la misma forma para simular el comportamiento en grupo, no hay ninguna librería ni funcionalidad en Unity que lo ofrezca. Así pues, solo quedaría implementar una solución propia por parte del desarrollador. Cualquier solución que se intente implementar para este problema constará de tres elementos troncales sobre los que se basará: un algoritmo de búsqueda, un algoritmo para simular el comportamiento en grupo y hacerlo de forma muy eficiente (ya que esta solución estaría pensada especialmente para dirigir muchos personajes a la vez por un escenario estático). En los siguientes subapartados podremos ver varios ejemplos de cada con uno de sus elementos junto con sus puntos positivos y negativos.

4.3.1 Algoritmos de búsqueda

Desde antes de la existencia de los videojuegos contamos con infinidad de algoritmos de búsqueda. Los más conocidos son el BFS y el DFS, los cuales, dados un grafo con pesos pueden darnos el camino mínimo entre dos puntos. Aún así, usar estos algoritmos no es la mejor idea. Pongamos un ejemplo en el que representamos la superficie transitable de nuestro videojuego en forma de grafo para que BFS y DFS puedan trabajar con él. Aun si obteniendo el resultado que se espera, el problema aparece en la forma en cómo exploran estos algoritmos el grafo. Ambos, exploran nodos de forma innecesaria (nodos que nunca nos darían el camino mínimo) por lo que hay una pérdida de eficiencia computacional significativa.

Para paliar esto, existen muchos algoritmos de búsqueda heurísticos que intentan evitar ese gasto de cómputo innecesario. Por eso mismo en la implementación de este proyecto, se usará como algoritmos de búsqueda base el A*. En términos de memoria, este algoritmo requiere los mismos recursos que BFS y DFS (esto no nos importa demasiado porque el escenario no es infinito). Sin embargo, computacionalmente hablando, gracias a hacer uso de una función heurística, la complejidad de cómputo (en el mejor de los casos) puede llegar a ser lineal, siempre y cuando el heurístico fuese perfecto. Sin embargo, aun sin tener un heurístico perfecto, como en este proyecto estamos trabajando en espacios 3D y con distancias virtuales, A* es un algoritmo que funciona suficientemente bien para lo que se pretende implementar.

4.3.2 Algoritmos de comportamiento grupal

Los algoritmos de pathfinding grupal son muy utilizados en videojuegos de género RTS (Real time Strategy), por lo que vamos a ver como solucionan ellos el problema de tener varios personajes con un mismo destino asignado y el de las colisiones entre personajes [**Formulación del problema**].

En el artículo **[5]**, en el cual hablan de la experiencia de intentar desarrollar un videojuego RTS en URG4 (UnrealEngine4), proponen sencillas soluciones para los problemas mencionados anteriormente. En el caso de muchos personajes dirigiéndose a un mismo objetivo, ellos optan por usar una técnica llamada “transitive bumping”. Esta consiste en que, una vez haya llegado el primer personaje al destino, este le pasa la información de que ya ha llegado al destino al siguiente que colisione con él para que el segundo se quede quieto, y así sucesivamente. Esta solución, como bien dicen ellos, funciona especialmente bien con grupos fuertemente conectados, de lo contrario

acabaría sucediendo lo que ya nos ocurría antes, y es que se alineen todos los personajes. Por lo que concierne a las colisiones entre personajes, ellos recurren a que uno de los dos personajes pueda aplicar una fuerza sobre el otro y desambiguar la colisión empujando al otro. Cabe destacar incluso que en muchos de estos juegos para evitar este problema, las tropas enemigas pueden atravesarse entre ellas.

Estas soluciones están pensadas para aplicaciones con grupos de personajes ya predefinidos, pero nosotros queremos una solución algo más genérica por que se optara por otras vías mejor detalladas en el punto cinco de [**Objetivos del trabajo**].

4.3.3 Mejorar la eficiencia

Al igual que en el mundo de los gráficos, actualmente, el cálculo de toda la lógica de los videojuegos parece no tener margen de mejora a no ser que hablemos en términos de paralelismo. Vivimos en unos días en los que la velocidad de reloj de las CPU se han estancado sobre los 5Ghz y el paralelismo es la única opción para seguir mejorando el rendimiento.

Actualmente utilizar **ECS** y **Jobs System** es la forma más sencilla y segura de programar código paralelo para Unity. En el artículo **[2]**, se hace una prueba donde se compara código programado de forma clásica con la paralelizada con ECS y Jobs System. El resultado que se puede observar es que sin perder ratio de frames se puede pasar de 16.5 mil personajes en escena a más de 150 mil sin ningún tipo de problema en el mismo equipo de trabajo.

Finalmente, no se ha encontrado alguna otra manera de paralelizar código o mejorar la eficiencia en términos de computación para Unity, aparte de las mencionadas anteriormente. Una alternativa si trabajamos sin motor de videojuegos podría ser CUDA. Este nos permite programar en GPU y sacarle el máximo rendimiento al paralelismo (similar a **ECS**). Sin embargo, el problema es que como se ha dicho anteriormente (dejando de lado las opciones mencionadas) no parece haber forma fácil ni viable de integrarlo con Unity.

5. Metodologia y rigor

Debido al acotado tiempo del que se dispone para el desarrollo del proyecto, se ha optado, por sugerencia del director, a utilizar la metodología **Scrum [6][17]**. Esta permitirá llevar un control minucioso de la evolución del proyecto y que este vaya creciendo de forma iterativa e incremental. Gracias a esto, siempre dispondremos de una versión funcional del proyecto. En la siguiente subsección se podrá ver en qué consiste la metodología y cómo se adaptara a este proyecto.

5.1 Método de trabajo

Primeramente, aclarar que cada vez que se hable de sprint, se estará haciendo referencia a una unidad de tiempo (en este caso de dos semanas) en las que se realizará una o varias tareas asociadas al proyecto. En la metodología **Scrum** se pueden diferenciar tres roles distintos **Product Owner**, **SCRUM Master** y **SCRUM Team**. Estos roles consisten y estarán adaptados a nuestro proyecto por:

- **Product Owner**: Representa el consumidor, es decir, es quien decide qué características son más necesarias del proyecto, valida o rechaza los resultados, etc. Esta figura en este proyecto estará tomada por el director y la codirectora.

- **SCRUM Master**: Representa el jefe de proyecto, quien mantiene al día la documentación, ayuda a sortear obstáculos al **SCRUM Team**, dirige el rumbo de los sprints. Relativo al proyecto se trataría del mismo jefe de proyecto y el director.

- **SCRUM Team**: Representa el equipo que implementa los requerimientos de cada sprint. En nuestro proyecto estaríamos hablando del diseñador, programador y tester.

Una vez explicados los diferentes roles que conforman esta metodología, se pasa a explicar brevemente cada una de las fases de esta, observables en la **Figura 7**.

Figura 7. Metodologia **Scrum** (Imagen extraida de **[17]**).

Si partimos de la fase de Release planning, esta junto con la de Sprint planning corresponden con las reuniones que se irán realizando antes de empezar un nuevo sprint. El objetivo de estas fases es básicamente definir qué hacer y cómo hacerlo. Será al Product Owner quien decida qué es lo que se prioriza en el proyecto y junto con el resto de los roles se decidirá cómo hacerlo. Daily Scrum es el dia a dia de las implementaciones exigidas en el Sprint Planning. Esta fase tiene de duración lo que dura el sprint y es llevado por los roles de Scrum Master y Scrum Team. Cabe destacar que un sprint es como una unidad atómica de trabajo, por lo que los requerimientos dentro de esta no podrán cambiar. Finalmente, en las fases de Sprint Review y Retrospective se recibe el feedback del Product Owner y se revisan los problemas y/o dificultades encontradas. Estas últimas fases se realizarán en la misma reunión que las fases de Release planning y Sprint planning del sprint siguiente

5.2 Herramientas de seguimiento

Para llevar el seguimiento de la evolución del proyecto, se hará uso de Github. Es una de las herramientas más utilizadas a día de hoy y permite documentar cada uno de los cambios y

actualizaciones que se realicen sobre el repositorio en el que se está trabajando. Además, para la gestión de las tareas a realizar en cada uno de los sprints antes mencionados, se utilizará trello. Esta aplicación está diseñada para gestionar tareas de un proyecto, pudiendo asignarlas a diferentes conjuntos dependiendo del estado en el que se encuentren.

5.3 Método de validación

Gracias al método de trabajo ágil, el método de validación recaerá de forma independiente sobre cada uno de los sprints (Sprint Review y Retrospective). Por lo tanto, para la validación de un sprint en concreto se realizarán diferentes pruebas por el desarrollador de forma que se verifique el correcto comportamiento. Además, gracias al feedback otorgado por el director y/o codirectora, se facilita hallar fallos que el propio desarrollador, por el hecho de serlo, podría pasar por alto (un ejemplo podría ser probar el proyecto en un escenario propuesto por el director y desconocido por el desarrollador).

5.4 Desviaciones

En cuanto a la metodología de trabajo no han habido variaciones a lo largo del proyecto. Gracias a la flexibilidad y el constante flujo de feedback en el que se basa la metodología Scrum no ha habido necesidad de modificarla.

Resaltar la importancia de las constantes reuniones ya que gracias a ellas se ha podido tener en mente en todo momento la etapa del proyecto que se atravesaba. También, gracias a estas reuniones se ha podido tener varios puntos de vista sobre un mismo tema en concreto, lo cual ha ayudado muchísimo en el desarrollo del trabajo.

6. Planificación temporal

Teniendo claro cual es el proyecto que se intenta abarcar y posibles obstáculos que nos podamos encontrar, se pasa a definir el cómo se realizará este mediante una detallada planificación.

6.1 Datos generales de la planificación

La duración del proyecto se estima en unos cuatro meses aproximadamente. Este periodo va des del 4 de febrero de 2019 hasta el 17 de junio de 2019 (un par de semanas antes de la lectura del trabajo final de grado). El proyecto está pensado para tener una carga lectiva de 18 créditos, lo cual supone un total de entre 450 y 550 horas de trabajo. Dejando de lado los objetivos secundarios que se comentarán más adelante, el resto están pensados para ser conseguidos de forma satisfactoria en este periodo de tiempo.

6.2 Plan de acción

Debido a la metodología àgil Scrum y, como se ha descrito en el informe anterior, la carga lectiva será repartida en sprints de dos semanas de duración. Las tareas que entran dentro de esta repartición son las de **[Implementación]** y la de **[Publicación Asset Store y manual de usuario]**. La **[Toma de contacto]** no entraría dentro ya que se ha realizado antes de la primera reunión con el director y arrancar el proyecto. Por su parte, la **[Gestión de proyecto]** corresponde a la asignatura del mismo nombre, por lo que su gestión se realizará tal y como se menciona en la asignatura. Además, en este grupo también se ha añadido como tarea la realización entera de la documentación.

Seguidamente, se pasará a definir cada una de las tareas pensadas como plan inicial y serán presentadas de forma cronológica. Sus estimaciones temporales, así como los recursos necesarios para llevarlas a cabo, estarán especificados en las secciones **[Tiempo de dedicación]** y **[Recursos]** respectivamente.

6.2.1 Toma de contacto

Esta primera parte se ha realizado antes de la primera reunión de seguimiento (4 de febrero de 2019) con el director pero después de la primera reunión con él en el que se explicaba el proyecto por primera vez.

En esta tarea se ha realizado toda la instalación del software necesario para el desarrollo del proyecto, además de la contextualización para la mejor comprensión del proyecto. En general, hacerse con el uso de Unity buscando información sobre cómo funciona el **Navigation System** (código de tarea TC1) de Unity y entender en qué consiste la tecnología **ECS** (código de tarea TC2) del mismo.

6.2.2 Gestión de proyecto

En esta sección se encuentran las tareas propiamente asociadas a la asignatura de GEP. En estas, se acaba de perfilar el proyecto especificando su planificación y objetivos principales a conseguir, así como su viabilidad y sostenibilidad del mismo. Las tareas mencionadas son las siguientes:

- **GEP1**: Contexto y alcance, mencionando también el estado del arte.
- **GEP2**: Planificación temporal.
- **GEP3**: Presupuesto y sostenibilidad.
- **GEP4**: Entrega final.

Junto a estas tareas, también contamos con la **GEP5** la cual es la escritura de la memoria del TFG. Esta tarea, como se podrá ver en el [**Diagrama de Gantt**], se hará de forma paralela a todo el proyecto para poder explicar de forma mucho más detallada los algoritmos y soluciones implementadas.

6.2.3 Implementación

En la siguiente sección se pasa a definir las tareas relacionadas con la implementación clasificándolas en cuatro grupos. Como se podrá observar en el diagrama de Gantt (**Figura 9**), las tareas asociadas al **[Dynamic WayPoints]** y **[Local Movement]** son dependientes de las de **[A*]**, y a su vez **[ECS]** de todo el resto. En el caso de **[ECS]**, no sería necesaria esta dependencia, pero debido a la total inexperiencia con esta tecnología se optó por hacerlo de esta manera para asegurar un proyecto finalizado y funcional.

Si se consiguiera acabar el proyecto en el tiempo establecido, se podrían introducir nuevas funcionalidades (aún no especificadas por el director). Incluso se podría mejorar el escenario prototipo inicial a uno de mayor tamaño y más calidad. Además como añadido, se podrían hacer varios juegos de pruebas y analizar el comportamiento del proyecto en situaciones distintas a las ya probadas.

*A**

Para este proyecto se eligió el A* como algoritmo de búsqueda **[7]**, por lo que para llegar a implementarlo esta tarea se subdividió en las siguientes:

- **Escenario prototipo (código de tarea IA1)**: Implementación de un escenario prototipo sobre el que correr el algoritmo.

- **Generación de un grafo virtual (código de tarea IA2)**: A partir de la malla de triángulos de la escena, generar un grafo sobre el que se aplique el A*.

- **Implementación del A* (código de tarea IA3)**.

- **Pintar el camino mínimo obtenido (código de tarea IA4)**: Poder visualizar los triángulos relativos al camino mínimo para poder verificar el correcto funcionamiento del A*.

Dynamic WayPoints

En esta sección se encuentran las funciones que implementan el movimiento del personaje por el mapa. Para conseguirlo se tiene pensado usar la técnica llamada **Dynamic WayPoints** (Detallado en el apartado del mismo nombre en el artículo **[8]** y en **[Dynamic WayPoints]**). Como añadido se ha realizado una primera versión con WayPoints estáticos como una primera aproximación de los WayPoints dinámicos.

Las funciones a implementar asociadas a esta sección serán las siguientes:

- **“Static” WayPoints (código de tarea ID1):** implementar una técnica similar a la de **Dynamic WayPoints** pero sin la reasignación de puntos en un intervalo de tiempo y con un número fijo de WayPoints por arista.

- **Dynamic WayPoints (código de tarea ID2).**

- **Recalcular camino si hay atasco (código de tarea ID3):** Si hay una arista de paso está siendo muy transitada recalcular un camino por aristas libres.

Local Movement

La parte de implementación dedicada al **Local Movement** está relacionada con las funciones destinadas a simular un movimiento grupal más natural. Para conseguir esto hacen falta dos funciones (explicadas más detalladamente en **[9]**):

- **Seek (código de tarea IL1)**: Este consiste en hacer que el personaje vaya a un punto determinado (ya queda implementado de forma indirecta con la técnica de **Dynamic WayPoints**).

- **Obstacle avoidance (código de tarea IL2):** Técnica para evitar obstáculos observando los personajes cercanos.

Cabe destacar que **Obstacle avoidance** tiene varios parámetros que se pueden modificar por lo que se iterará varias veces sobre este punto para un mejor resultado. Incluso también cabe la posibilidad de probar otras técnicas si el tiempo lo permite.

ECS

Finalmente, como última tarea quedaría pasar de la programación clásica a usar **ECS**. Siendo más específicos en este grupo encontramos la tarea de documentarse acerca de ECS (código de tarea IE1) y la traducción como tal del proyecto (código de tarea IE2). Esta es la tarea de implementación a la que más tiempo se le piensa dedicar tal y como se puede observar en **Figura 8** por la inexperiencia de no haber trabajado antes con esta forma de programación. Este tiempo asignado se utiliza como forma de ayuda al problema explicado en **[Paralelizar con local movement]**.

Como apunte final, destacar que junto a cada una de las tareas de implementación, van asociadas las pertinentes de analisi y testeo aunque en el diagrama de Gantt no se especifica explícitamente.

6.2.4 Publicación Asset Store y manual de usuario

Una vez realizada toda la implementación dicha en el apartado anterior, y de haber funciones extra, también de estas, el proyecto está pensado para publicarlo en la Asset Store de Unity. Esta tarea no afectará al proyecto final en términos de mejoras funcionales, por lo que es independiente a la calidad del mismo si se publica o no. Sin embargo, es interesante publicarlo, ya que si se cumplen los plazos con las tareas mencionadas anteriormente y queda un buen producto, este es un punto que siempre suma.

Las tareas relacionadas con apartado consisten en el aprendizaje sobre cómo publicar un Asset en la Asset Store (código de tarea PA1), publicarlo y la creación de un manual de usuario para la persona que lo vaya a usar(código de tarea PA2).

6.2.5 Reuniones

En esta sección se engloban todas reuniones que se darán durante el transcurso del proyecto. Estas reuniones sirven como metodo de validacion de las implementaciones realizadas y como seguimiento del proyecto. Así pues, esta no crean dependencias entre los grupos de implementación pero dependiendo de la valoración del director, sí que podrían llegar a modificar los pasos a seguir de cara a los siguientes objetivos.

6.3 Tiempo de dedicación

A continuación se mostrará una tabla con los grupos de tareas a realizar y su carga de trabajo en horas estimada para cada uno de ellos. La tabla con el desglose a nivel de tarea junto con las dependencias y el código de cada tarea se encuentra en el [**Apéndice A]**.

Tarea		Tiempo (en horas)
Toma de contacto		10h
Gestión de proyecto	GEP	70h
	Documentación	60h
Implementación: A*		70h
Implementación Dynamic WayPoints		60h
Implementación: Local Movement		70h
Implementación: ECS		90h
Publicación Asset Store y manual de usuario		80h
Reuniones		10h
Total		520h

Figura 8. Tabla con los tiempos estimados de cada tarea.

Teniendo en cuenta la **Figura 8** y el diagrama de Gantt que se podrá observar en el apartado [**Diagrama de Gantt**], es difícil estimar cuál será la cantidad de trabajo dedicada de forma diaria. Más aún si hay varios roles implicados en el decurso del proyecto, aun así teniendo en cuenta el número de horas totales se estiman unas seis o siete horas de trabajo diario. Esto es orientativo ya que habrá días donde se podrá dedicar más o menos horas dependiendo de diferentes factores (reuniones, otras asignaturas, imprevistos, etc).

6.4 Dependencias

Como se puede ver en la tabla del [**Apéndice A**], el proyecto tiene varias partes secuenciales (la gran mayoría) y algunas de concurrentes.

Cabe destacar que los grupos de tareas de implementación son secuenciales entre sí. Esto es debido a la metodología **Scrum,** ya que en todo momento queremos un producto disponible que ofrecer y funcional. Así pues, al finalizar cada grupo de implementación nos aseguramos tener algo sólido sobre lo que seguir avanzando. Intragrupalmente, los grupos de implementación Dynamic WayPoints y Local Movement tienen tareas paralelizables porque están compuestas de funciones

independientes en términos de funcionalidad. Por otra parte, las tareas de implementación del A* y ECS por su propia naturaleza la única forma de realizarlas es de forma secuencial.

Finalmente, todas las tareas asociadas a reuniones y redactado de la memoria se irán haciendo en paralelo a la realización del proyecto como forma de seguimiento de este. Con esto se consigue que la documentación se realice a medida que se vayan finalizando las diferentes tareas de implementación.

6.5 Recursos

En los siguientes subapartados se mostrarán y especificarán los recursos que harán falta para el desarrollo del proyecto.

6.5.1 Humanos

Tal y como se explicó en [**Partes implicadas**], para este proyecto harán falta las labores de diseñador, programador, tester y jefe de proyecto, las cuales serán realizadas por la misma persona. A pesar de eso, se explicarán como si fueran llevadas por personas diferentes.

- **Diseñador**: La persona asociada a este puesto será la encargada del diseño los diferentes algoritmos y estructuras de datos que se utilizarán.

- **Programador**: Esta persona será la encargada de implementar todos los diseños algorítmicos entregados por el diseñador.

- **Tester**: Será el encargado de testear cada una de las tareas de implementación mostradas en el diagrama de Gantt.

- **Jefe de proyecto**: Corresponde con la persona encargada de supervisar y coordinar las tareas realizadas por los tres puestos anteriores para su correcta finalización.

6.5.2 Hardware

Para todas las tareas mencionadas en **[Plan de acción]** se hará uso del mismo ordenador de sobremesa. Este cuenta con las siguientes especificaciones generales:

- **CPU**: Intel Core i7-8700K 3.7Ghz

- **GPU**: Gigabyte GTX 1070Ti Gaming 8GB GDDR5

- **RAM**: Corsair Vengeance LPX DDR4 3000Mhz PC-24000 16GB 2x8GB CL15

- **Almacenamiento**:
    - SSD: Samsung 850 Evo SSD Series 500GB SATA3
    - HDD: Seagate BarraCuda 3.5" 1TB SATA3

Cabe destacar que para las reuniones de seguimiento con el director, y en ocasiones con la codirectora, se utilizara un portátil para mostrar en forma de video el trabajo realizado. Este es un portatil de gama baja con un procesador Intel Core i3, gráfica integrada, 1GB de ram y 250GB de almacenamiento HDD.

6.5.3 Software

Finalmente, se pasa a hacer un repaso por el conjunto de herramientas software del que se hará uso para este proyecto.

- **Windows 10 Pro** es el sistema operativo sobre el que se trabajará en la totalidad del trabajo.

- **Unity** en su versión 2018.3.3f1 como motor de videojuegos sobre el que correrá nuestra aplicación.

- **Visual Studio Community 2017** versión 15.6.7 como editor de código.

- **Trello** se utilizará como herramienta para poder organizar mejor las tareas a realizar a lo largo del proyecto.

- **Google Docs** se utilizará para la documentación tanto de la fita inicial como de la memoria final.

6.6 Diagrama de Gantt

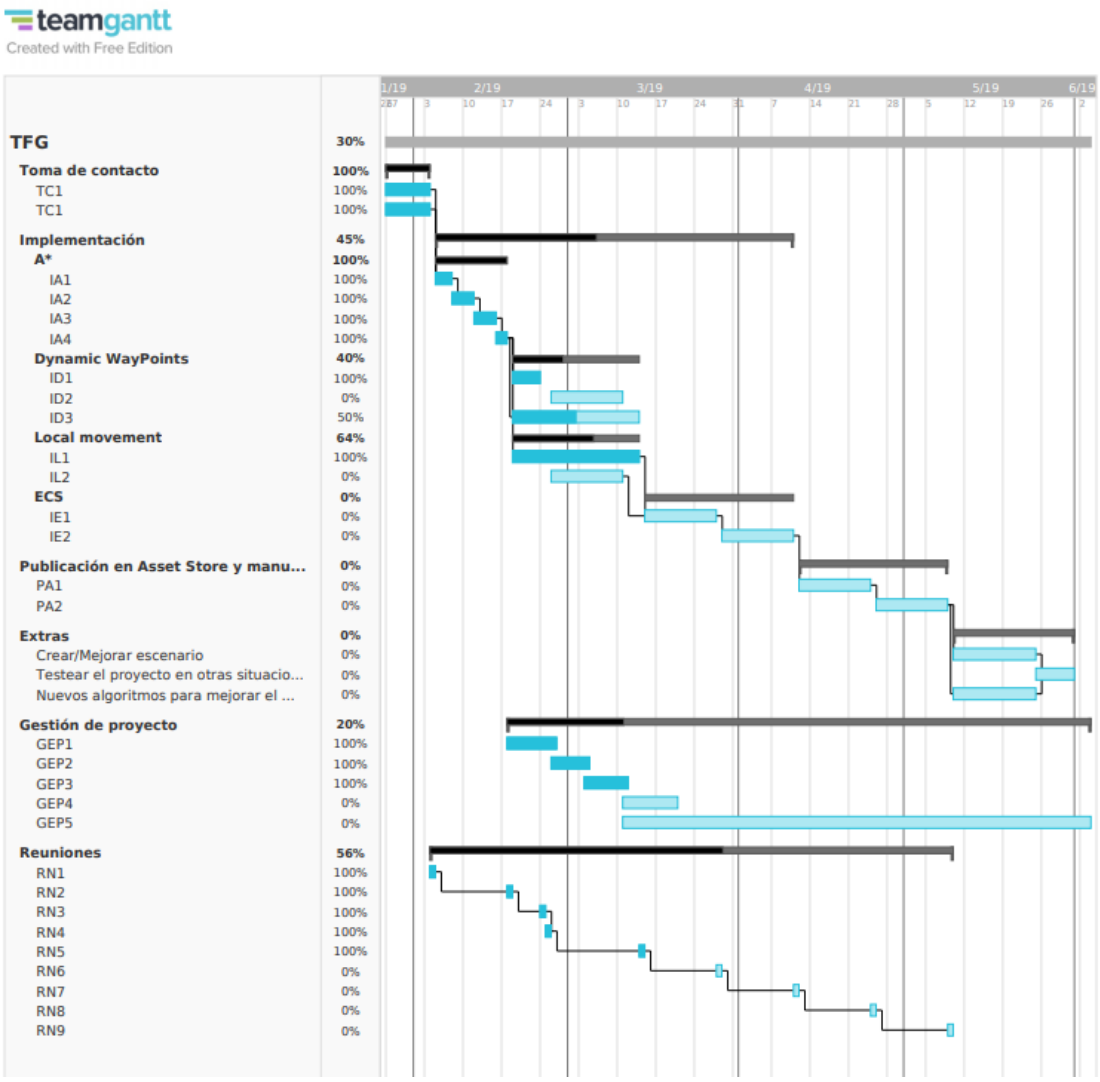


Figura 9. Diagrama de Gantt.

6.7 Valoración de alternativas

Como se explicará en las siguientes subsecciones las principales fuentes de problemas normalmente vendrán por los errores de código y por el desconocimiento de la tecnología **ECS**. Esto se ha intentado paliar haciendo una sobreestimación en el caso de la implementación de **ECS**.

También, si por algún motivo los Bugs o cualquier otra fuente de problemas fuera lo suficientemente grande como para detener el proyecto por un tiempo excesivo, se prescindirá de la realización del punto **[Publicación Asset Store y manual de usuario]**. Sera este punto el que se recorte y no otro, debido a que este no aporta nada en pro de solucionar los problemas planteados en la [**Formulación del problema**]. Además, desde el principio publicar el proyecto como Asset solo se ha planteado en el caso de llevar el proyecto con buen margen de tiempo y si se obtienen muy buenos resultados con la solución desarrollada.

6.7.1 Bugs

Como bien se explicaba en [**Errores de programación**], estos son los errores que provienen de cometer fallos en la programación del proyecto. Aun así, con el tiempo asignado a cada uno de las funciones de implementación debería ser más que suficiente. Si esto no fuera suficiente, se estima que se tendría que incrementar el trabajo del tester en 20h (5h por cada grupo de tareas de implementación).

6.7.2 Documentación ECS

Gracias a la escasa documentación en ECS para hacer según qué cosas, se ha optado por otorgarle más tiempo para cuando se use esta tecnología. Aun así, como otra alternativa se ha pensado en hacer uso de la versión híbrida de ECS la cual se puede combinar muy fácilmente con la programación tradicional. Llevar a cabo tanto la primera como la segunda opción se ha estimado en unas 10h extras de trabajo para el diseñador de algoritmos.

6.7.3 Paralelizar con local movement

Aun sin haber llegado a la parte de implementación del **ECS**, analizando cómo funciona el **Local Movement** es muy probable que haya problemas a la hora de integrarlo junto con **ECS**. **Obstacle avoidance** necesita saber de la posición de personajes cercanos a él, por lo que no se sabe cómo se podría realizar esto usando **ECS** y **C# Jobs System**. Hablando con los directores del TGF se ha llegado a un par de alternativas/soluciones a este problema. Una de ellas consistiría en construir una solución propia de **Local Movement** en **ECS** a partir de una de simulación de partículas (agua por ejemplo) en **ECS** ya que es el mismo problema. La otra sería optar por otro algoritmo de Local Movement que no sea el **Obstacle avoidance**.

Concluyendo, si no se pudiera implementar directamente **Obstacle avoidance** con **ECS** las dos soluciones supondría un sobrecargo de trabajo para el diseñador y el programador. Para cuantificar este sobrecargo de trabajo, nos pondremos en la peor de las situaciones. Este sería optar por la segunda solución propuesta anteriormente (cambiar **Obstacle avoidance** por otro algoritmo de similar performance). Esto supondría unas 10h de trabajo extra para el diseñador de encontrar un algoritmo compatible con **ECS**, unas 10h para el programador de implementación y unas 5h de pruebas para el tester. La suma de horas extra no es muy elevadas ya que se intentaría optar por un algoritmo similar pero muy sencillo.

6.8 Desviaciones y conclusiones

Inicialmente, todo el bloque de implementación se tenía pensado acabar a mediados de abril. Esto no ha sido posible, ya que tal y como se especuló al principio, el subbloque de implementación de Entity Component System (ECS) se ha alargado hasta el 23 de mayo. Esto se ha debido a varias limitaciones encontradas con las tecnologías relacionadas con ECS (**[Problemas encontrados]**), así pues, el diagrama de Gantt modificado, quedaría como se muestra en la **Figura 10.** Esto ya se penso que podia suceder, por lo que realizar la redacción de la memoria de forma paralela al proyecto ha sido todo un acierto. Como era de esperar, esta ampliación en el tiempo dedicado al bloque ECS ha comprometido el de Publicación en Asset Store, por lo que se ha decidido prescindir de este último.

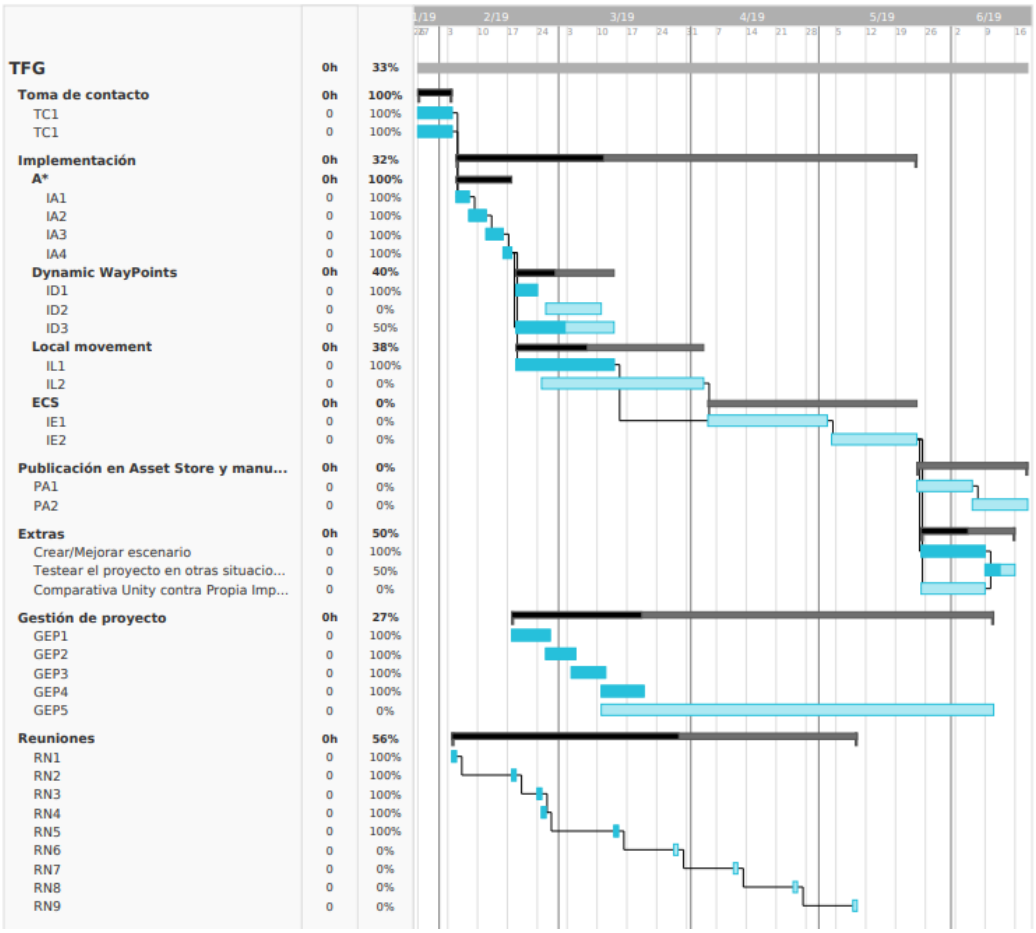


Figura 10. Diagrama de Gantt actualizado.

Esta modificación en el plan inicial reduce los gastos relacionados con la implementación debido a que, a pesar de que amplía las horas de trabajo dedicadas en ECS, los gastos en el bloque de la Publicación del proyecto como Asset eran mucho mayores. Destacar que aunque esto parezca un punto positivo, no hay que olvidar que este ahorro es a costa de suprimir uno de los objetivos, si bien no principales, planteados inicialmente.

Aún así, descartar la publicación del proyecto en la asset store ha permitido dejar respirar el proyecto. De haberlo intentado desarrollar, no se podría haber cerrado muchos puntos del proyecto con la correcteza con la que se ha hecho. Además, hubiera implicado publicar un proyecto claramente mejorable en algunos aspectos y con un manual de usuario redactado con muy poco tiempo.

7. Presupuesto del proyecto

En las siguientes secciones se estimara el presupuesto total sumando el coste de los recursos esmentados en su correspondiente sección, el gasto de los costes indirectos, los costes de contingencia y el de imprevistos. Todos estos costes se han aproximado incluyendo el IVA y otros impuestos.

7.1 Costes directos

Dentro de los costes directos podemos diferenciar los de recursos humanos. Así pues, se pasa a definir el presupuesto para cada uno de ellos.

7.1.1 Costes de recursos humanos

Todos los roles mencionados en recursos humanos serán llevados por una misma persona, aún así el precio por hora asignado a cada uno de ellos será el correspondiente con su rol. Estos datos se han obtenido de la página **[10]** la cual cuenta credenciales a nivel global. Las horas estimadas para cada uno de ellos se han calculado a partir del diagrama Gantt y las horas asignadas a cada tarea.

Rol	Precio(€/h)	Tiempo total(h)	Coste total(€)
Diseñador **[11]**	100	115	11,500
Programador **[12]**	30	110	3,300
Tester **[13]**	20	75	1,500
Jefe de proyecto **[14]**	100	220	22,000
Total	**-**	**520**	**38,300**

Figura 11. Presupuesto para los recursos humanos

A continuación la **Figura 12** proporciona la distribución de tiempo que cada rol gasta en las diferentes tareas del proyecto, así como su presupuesto.

Tarea	Duración (h)	Diseñador	Programador	Tester	Jefe de proyecto	Coste de la Tarea(€)
Toma de contacto	10	5	0	0	5	1,000
Gestión de Proyectos	130	0	0	0	130	13,000
Implementación: **A***	70	20	25	15	10	4,050
Implementación: **Dynamic WayPoints**	60	20	15	15	10	3,750
Implementación: **Local Movement**	70	20	20	20	10	4,000
Implementación: **ECS**	90	40	25	10	15	6,450
Publicación Asset	80	10	25	15	30	5,050

Store y manual de usuario						
Reuniones	10	0	0	0	10	1,000
Total	**520**	**115**	**110**	**75**	**220**	**38,300**

Figura 12. Tiempo en horas dedicado por cada rol a las distintas tareas y su coste.

7.2 Costes indirectos

Una vez finalizado con los costes directos se pasa a evaluar otros gastos asociados al proyecto de forma indirecta. En estos entrarían tanto los gastos de hardware como los de software, además de los gastos de electricidad y internet asociados al proyecto. Cabe destacar, que también se ha añadido el coste del desplazamiento, ya que el coste de este no entra dentro de los gastos propios del alumno en su dia a dia.

7.2.1 Costes hardware

A continuación en la **Figura 13**, podemos ver los gastos de Hardware relacionados con el proyecto mencionados en el plan de proyecto.

Producto	Precio(€)	Unidades	Vida Útil(años)	Amortización(€)
PC	1800	1	5	150
Portatil	300	1	3	25
Total	**2100**			**175**

Figura 13. Presupuestos asociados al hardware.

Teniendo en cuenta que la duración total del proyecto es casi de 5 meses debido a que se comenzó antes de tiempo, la fórmula que se ha utilizado para calcular la amortización ha sido la siguiente:

$$\forall p \in Producto \;\; Amortización_p = \frac{unidades_p * Precio_p * 5\ meses}{años_p * 12 meses}$$

7.2.2 Costes software

Por la parte de los costes software en este caso suman un total de 10 euros que es lo que costó la licencia de Windows 10 Pro. El resto del software del que se hará uso para este proyecto es totalmente gratuito.

7.2.3 Otros costes

Relativo a la electricidad consumida, la cantidad de kWh se ha obtenido multiplicando los W de consumo estimados por el total de 520h de trabajo. El consumo de W se ha estimado teniendo en cuenta que el ordenador dispone de una fuente de alimentación de 650 W. Así pues, para este proyecto con un rendimiento medio/bajo, este debería consumir alrededor de 400 W.

Producto/Servicio	Precio(€)	Unidades	Coste aproximado(€)
Internet(Fibra Òptica)	60/mes	4 meses	200
Electricidad	0,164632/kWh	208 kWh	34.24
Transporte: Metro**[15]**	2,20/viaje	2(ida y vuelta)*9 (reuniones)	39.6
Transporte: Tren**[16]**	5/viaje	2(ida y vuelta)*9 (reuniones)	90
Total	-	-	**363.84**

Figura 14. Tabla con los costes de electricidad, transporte e internet.

7.3 Costes de contingencia

Una vez mencionados todos los costes que demanda el proyecto, se estiman los costes de contingencia en pro de paliar posibles gastos imprevistos. Estos se han estimado con una probabilidad del 12% ya que a pesar de estar bastante fraccionado, el proyecto consta de tareas de hasta 40h de duración. A pesar de que para estas tareas (por ejemplo paralelizar con **ECS**) hay pensadas un plan B, para tareas como la publicación del Asset no lo hay y también contiene tareas de larga duración. Es debido a esta incertidumbre por lo que se ha optado por un porcentaje de contingencia superior al 10%.

	Porcentaje de contingencia	Coste(€)
Contingencia	12%	4639.66
Total	-	**4639.66**

Figura 15. Tabla con los costes de contingencia.

7.4 Costes de incidentes

En esta sección se cuantifican como se puede ver en la **Figura 16** los costes de los posibles incidentes. Destacar que, las soluciones como tal se especifican en el apartado de [**Valoración de alternativas**], así pues, en esta tabla solo se muestran los recursos necesarios para llevar a cabo esa solución.

Causa	Solución	Riesgo de ocurrencia	Impacto en coste(€)	Coste(€)
Bugs	20h extra de trabajo para el tester	20%	400	80
Falta de	10h extra de trabajo para el diseñador			

documentación sobre ECS		15%	1,000	150
Paralelizar local Movement	10h de trabajo extra para el diseñador, 10h de trabajo extra para el tester, 5h de trabajo extra para el tester	35%	1,400	490
Total		**-**		**720**

Figura 16. Tabla con los costes de los incidentes.

7.5 Presupuesto total

Finalmente se pasa a mostrar el presupuesto total en la **Figura 17** a partir de los datos calculados en las anteriores secciones.

Tipo de coste		**Coste estimado(€)**
Recursos humanos		38,300
Indirectos	Hardware	2100
	Software	10
	Otros	363.84
Costes de contingencia		4639.66
Costes de incidentes		720
Total		**46,133.5**

Figura 17. Tabla de presupuesto total

Como se puede observar en la **Figura 17** el mayor grueso del gasto viene por parte del equipo de desarrollo del proyecto. Esto es positivo ya que de forma global se ha obtenido un precio lo suficientemente competitivo a dia de hoy, pero como se explicará en la siguiente sección, en caso de desviaciones será el coste de recursos humanos el que más eleve el presupuesto del proyecto.

7.6 Control de gestión

Al igual que para la planificación, el principal problema para este proyecto puede venir de la desviación temporal debida a alguna tarea. Dejando el problema de poder paralelizar el local movement, el resto de desviaciones del proyecto requieren de más tiempo como solución a estas (bugs y implementar **ECS**). Por eso y como se ha visto en el **[Presupuesto total],** alargar la fase de implementación de trabajo de forma descontrolada podría llevar a un incremento considerable del total del presupuesto.

Por lo tanto, como medida para controlar esto, se ha optado por ir actualizando en tiempo real el tiempo dedicado a la realización de cada una de las tareas. De este modo se podrá llevar al día la cantidad de tiempo usado y del que todavía queda para el resto de tareas. Esto permitirá tomar mejores decisiones sobre cómo se reorganizan los recursos humanos de cara a descartar o ampliar objetivos. Aparte, una vez finalizado el proyecto se podrá restar el tiempo real utilizado con el estimado inicialmente y tener una visión real de la desviación obtenida a nivel de tarea.

Por lo que concierne al hardware y al software, suceda el imprevisto que suceda no hará falta la adquisición de nuevo material en ninguno de los sectores.

7.7 Desviaciones

Tal y como se ha mencionado en el apartado **[Presupuesto total]**, el mayor grueso de los costes viene dado por los gastos en recursos humanos. Es por eso que con las desviaciones que han habido en la planificación final, donde se ha visto modificado el presupuesto es en recursos humanos.

En la sección **[Desviaciones y conclusiones]**, se explica la modificación de planificación realizada, la cual consiste en cancelar la publicación del proyecto en la asset store de Unity debido al exceso de tiempo empleado en la paralelización con ECS. Esto se ha traducido en un ahorro de costes ya que las horas dedicadas a la publicación en su mayoría iban a ser realizadas por el Jefe de proyecto, el cual tiene el mayor sueldo que el resto del equipo. Cabe destacar que obviamente las horas extras realizadas para implementar el bloque de ECS se han tenido en cuenta como se muestra en la siguiente **Figura 18.**

Tarea	Duración (h)	Diseñador	Programador	Tester	Jefe de proyecto	Coste de la Tarea(€)
Implementación: **ECS**	90	40	25	10	15	6,450
Implementación: **ECS Actualizado**	160	70	40	25	25	11,200

Figura 18.Horas dedicadas a la tarea de implementación ECS actualizada.

Rol	Precio(€/h)	Tiempo total(h)	Coste total(€)
Diseñador **[11]**	100	135	13,500
Programador **[12]**	30	100	3,000
Tester **[13]**	20	75	1,500
Jefe de proyecto **[14]**	100	200	20,000
Total	**-**	**510**	**38,000**

Figura 19. Tabla de costes de recursos humanos actualizados.

Finalmente, haber conseguido realizar un proyecto que cumpla con las exigencias principales del usuario sin haber excedido el presupuesto inicial es todo un logro. Si se tratase de un proyecto con inversión real, habríamos conseguido ahorrar presupuesto.

8. Sostenibilidad

8.1 Ambiental

Los recursos necesarios utilizados en este proyecto con efectos en el medio ambiente son la electricidad e Internet. Esto provoca que la única forma de reducir el impacto de este en el medio ambiente, sea reduciendo su consumo. Cabe destacar, que dentro del impacto ambiental de la realización del proyecto también habría que englobar el del hardware utilizado. No obstante, todos ellos se piensan reciclar de forma adecuada una vez hayan llegado al fin de su vida útil.

Para ahorrar en consumo energético, dejando de lado soluciones difíciles de conseguir como reducir el tiempo de desarrollo del producto, una opción por la que se ha optado en este proyecto ha sido en limitar la potencia de algunos componentes del PC. La computadora sobre la que se está realizando está compuesta por piezas hardware de alta gama, por lo que la velocidad de la RAM o el clock de la tarjeta gráfica no hace falta que estén funcionando ni a un 30% de su capacidad para este proyecto.

Actualmente las soluciones existentes en el mercado requieren del mismo consumo que la nuestra ya que se está hablando de software. Sin embargo, nuestra solución intenta mejorar la eficiencia, por lo que con el mismo consumo, se aprovecharía mejor el hardware y se podría obtener una mejor performance.

8.2 Economica

En el apartado de **[Presupuesto del proyecto]**, se detallan y cuantifican todos los gastos asociados a este, incluyendo tanto los costes directos como indirectos. Si bien el presupuesto se ha estimado con las horas pensadas en la planificación del proyecto, este podría variar ligeramente debido a las desviaciones nombradas en la correspondiente sección.

Por otro lado, pensando en el usuario final, para este siempre será más barato la utilización de nuestro proyecto en el momento en el que no tiene que desarrollarlo por su cuenta haciendo uso de esta solución. Así pues, si el usuario quisiese una solución parecida a la nuestra en Unity tendría que seguir prácticamente los mismos pasos que estamos siguiendo nosotros con sus respectivos costes.

8.3 Social

Actualmente, el sector de los videojuegos está en constante crecimiento desde hace varios años. Sin embargo, a pesar de que Unity siga actualizándose, las empresas de desarrollo de videojuegos que usan Unity como motor de videojuegos optan por desarrollar o reimplementar sus soluciones. Esto es debido a que la mayoría de las que ofrece Unity son muy poco adaptables. Así pues, eso es exactamente lo que estamos haciendo para este proyecto, reimplementar varias funciones de Unity, por lo que como opinión, creo que esta es una experiencia muy valiosa a nivel personal.

Concretamente para este proyecto, se está intentando mejorar la solución actual que ofrece Unity ofreciendo en nuestro caso, la posibilidad de mover más personajes de forma más natural y más eficiente. Directamente no hay una necesidad de este producto, pero sí que es una solución que amplía la variedad en el mercado y ayuda a los desarrolladores a encontrar algo que se ajuste más a lo que buscan.

9. Leyes y regulaciones relativas al proyecto

Con respecto a la ley de protección de datos, este proyecto no hace uso de ningún tipo de dato personal. Esto es así tanto por parte del creador del propio proyecto como del usuario final.

Por otro lado, el proyecto es totalmente Open Source. El repositorio en el que se encuentra, es de libre acceso a cualquiera, por lo que no hay ningún tipo de preocupación respecto a temas de copyright.

Finalmente, si el proyecto se publicase en el Asset Store para su uso comercial, si que habría que hacer una revisión a este asunto ya que dejaría de ser un proyecto Open Source. Aun así, no debería de existir mayor problema ya que en la totalidad del proyecto solo se utiliza código material propio y en ningún caso de terceros. Todos los personajes utilizados en el proyecto han sido creados por el desarrollador del mismo al igual que el escenario con ProBuilder. En esta misma línea, todo el código generado, así como las clases creadas, han sido implementadas de la misma manera.

10. Justificación del proyecto

Se pasa a justificar en los siguientes subapartados la validez del proyecto como uno de la especialidad de computación. Esto implica hacer un repaso por las asignaturas de la especialidad que han ayudado a la hora de implementar el proyecto y qué competencias relacionadas con la computación desarrolla el mismo.

10. 1. Asignaturas relacionadas con Computación

Primeramente, se listaran el conjunto de asignaturas pertenecientes a la especialidad de computación que se cree han sido de utilidad a la hora de desarrollar este proyecto.

- **Algorismia**: Seguramente de las más importantes ya que el objetivo principal del proyecto es reimplementar algoritmos de movimiento y hacerlo de una forma muy eficiente. Por ello, la capacidad de análisis sobre la complejidad de ciertos algoritmos y estructuras de datos es de gran utilidad para el proyecto.

- **Inteligencia artificial**: A la par de importancia junto a **Algorismia**. En esta asignatura se estudia la base teórica de la inteligencia artificial, por lo que se aprende como funcionan algoritmos de búsqueda de mínimos y máximos globales. Dentro de estos destaca el A*, utilizado como pilar sobre el que se implementa el movimiento en este proyecto.

- **Gráficos**: Pese a trabajar en una plataforma como Unity, en la que el proceso de renderización queda prácticamente oculto para el desarrollador, esta asignatura ha sido de gran ayuda para agilizar el proceso de aprendizaje en dicha plataforma. Un ejemplo de esto se ve claro cuando se tiene que acceder a la Navigation Mesh, gracias a **Gráficos**, la extracción de dicha malla de triángulos y poder interpretarla ha sido relativamente sencilla. Esto es debido a que es devuelta como un conjunto de vértices e índices de triángulos justo de la misma manera en que lo hace openGL (API con la que se trabaja todo el curso de **Gráficos**).

- **Lenguajes de programación**: Gracias a esta asignatura se han facilitado el aprendizaje del lenguaje utilizado por Unity para el scripting, en este caso C#. También, en esta asignatura se estudiaron diferentes paradigmas existentes relativos a la programación lo cual ha ayudado a al estudio y entendimiento de **ECS**.

10.2. Competencias asociadas al proyecto

Partiendo del conocimiento adquirido en las asignaturas mencionadas anteriormente, se pasa a hacer un listado del conjunto de competencias técnicas del ámbito de la computación que desarrolla este proyecto.

- **CCO1.1: Avaluar la complexitat computacional d'un problema, conèixer estratègies algorísmiques que puguin dur a la seva resolució, i recomanar, desenvolupar i implementar la que garanteixi el millor rendiment d'acord amb els requisits establerts. [En profunditat]** Este es prácticamente el propósito principal del proyecto. Se ha visto un problema de cierta complejidad computacional el cual puede ser mejorado haciendo uso del paralelismo debido a sus características.

- **CCO1.3: Definir, avaluar i seleccionar plataformes de desenvolupament i producció hardware i software per al desenvolupament d'aplicacions i serveis informàtics de diversa complexitat. [Un poco]**

- **CCO2.1: Demostrar coneixement dels fonaments, dels paradigmes i de les tècniques pròpies dels sistemes intel·ligents, i analitzar, dissenyar i construir sistemes, serveis i aplicacions informàtiques que utilitzin aquestes tècniques en qualsevol àmbit d'aplicació. [Bastante]**

- **CCO2.2: Capacitat per a adquirir, obtenir, formalitzar i representar el coneixement humà d'una forma computable per a la resolució de problemes mitjançant un sistema informàtic en qualsevol àmbit d'aplicació, particularment en els que estan relacionats amb aspectes de computació, percepció i actuació en ambients o entorns intel·ligents. [Un poco]** Representado minimamente en la implementación de obstacle avoidance, el cual intenta simular el comportamiento humano de masas para resolver de forma local el problema de sortear un obstáculo.

- **CCO2.3: Desenvolupar i avaluar sistemes interactius i de presentació d'informació complexa, i la seva aplicació a la resolució de problemes de disseny d'interacció persona computador. [Un poco]** En su justa medida ya que se intentara crear una interfaz en Unity para probar el proyecto con diferentes parámetros.

- **CCO2.6: Dissenyar i implementar aplicacions gràfiques, de realitat virtual, de realitat augmentada i videojocs. [Bastante]** En gran parte, ya que el la totalidad del proyecto se va a realizar sobre una plataforma de desarrollo de videojuegos.

- **CCO3.1: Implementar codi crític seguint criteris de temps d'execució, eficiència i seguretat. [Bastante]** Esta también será una parte importante del proyecto, no tanto en el ámbito de la seguridad pero si en el de eficiencia y tiempo de ejecución ya que es uno de los objetivos principales.

- **CCO3.2: Programar considerant l'arquitectura hardware, tant en asemblador com en alt nivell. [Un poco]** Ligeramente ya que lo que se intenta en este proyecto es paralelizar el código, y aunque eso resulte una tarea más sencilla debido a **ECS** y Jobs System, hay que tener en cuenta el Hardware utilizado (en este caso la CPU).

10.3. Adecuación como proyecto de computación

Para el desarrollo de este proyecto se han utilizado algoritmos típicos y comúnmente utilizados en el ámbito de la computación. Estos son el A* para el movimiento por ejemplo o el Obstacle Avoidance como solución local a un problema global (que es el de evitar chocar con obstáculos).

También, se hace claro hincapié en obtener un alto rendimiento, por lo que hace falta un alto nivel de análisis. Esto es requerido a lo largo de todo el proyecto ya que no sirve de nada paralelizar el código si luego se hace un gasto computacional excesivo en cálculos innecesarios.

Por otra parte, en el proyecto se trata el paradigma de la programación orientada a datos cuando se intenta paralelizar el código con **ECS**. Otra vez, estamos frente a una de las cuestiones que suelen tratarse en el ámbito de la computación, el de los diferentes paradigmas y cómo se aplican estos dependiendo del lenguaje.

Teniendo en cuenta el conjunto de razones expuestas, no cabe ninguna duda de que el proyecto tiene cabida dentro del marco de la computación.

11. Implementación del movimiento

En el siguiente apartado se explicará en profundidad los algoritmos utilizados para implementar el movimiento individual de un personaje y cómo se ha extendido este para adecuar este movimiento a uno de más realista con varios personajes en escena. También se explicaran en sus correspondientes apartados las diferentes estructuras y clases utilizadas para la implementación de los mismos algoritmos. En concreto se han implementado los siguientes algoritmos/métodos para conseguir los objetivos principales:

- **Representación en forma de grafo del mapa**: para poder aplicar el A* sobre él y encontrar el camino mínimo entre dos puntos.

- **El algoritmo A*:** para poder desplazar al personaje por la ruta que se encuentre con este algoritmo. Con el A* solamente se encontrarán los triángulos que tiene que atravesar el personaje.

- **Static WayPoints:** Versión simple de Dynamic WayPoints donde se obtienen los puntos de paso del camino de triángulos.

- **Dynamic WayPoints:** Algoritmo donde se obtienen los puntos de paso del camino de triángulos.

- **Local Movement:** Algoritmo para distribuir de forma natural los personajes a lo largo de la arista que une dos triángulos.

- **Reimplementación de Dynamic WayPoints y Local Movement en ECS:** Adaptar estos algoritmos para poder utilizarlos con ECS, Jobs System y Burst Compiler.

Toda la fase de implementación sin paralelizar se ha realizado sobre un escenario muy simple creado con proBuilder **[18]** y personajes propios diseñados con MagicaVoxel **[19]**. Esta versión single core se aloja en el proyecto distinto al paralelo, el secuencial se encuentra en el proyecto TFGSinParalelizar mientras que el paralelo TFGConParalelizacion.

11.1 Implementaciones previas

Una vez visto cómo se comporta la solución de Unity se comienza la implementación del movimiento por cuenta propia. Tal y como se ha mencionado en apartados anteriores, para ello hará falta implementar el A*, el cálculo de los puntos de paso por los triángulos (Dynamic WayPoints) y el Local Movement. Por lo tanto, antes de poder implementar el A* hace falta representar la información en forma de grafo para poder aplicar el algoritmo sobre él y así obtener el camino a recorrer por el escenario.

Para ello se han creado la clase Node y Neighbor que representaran un nodo del grafo y el vecino a un nodo respectivamente. La clase Node cuenta con un identificador de nodo único, las coordenadas del triángulo que representa y una lista de Neighbors contienen los triángulos adyacentes. Además la clase Node cuenta con un método para obtener el baricentro del triángulo (utilizado en el algoritmo A*) y un método para saber si alguno de sus vecinos dispone de espacio para habilitar el paso a un personaje más este nodo al vecino. Por otra parte, la clase Neighbors, como atributos interesantes, contiene el mismo nodo que representa, los puntos de la arista contigua al nodo del que es vecino y un entero que representa la ocupación (número de personajes pasando de por la arista nombrada anteriormente). Como métodos, Neighbors, cuenta con varios para setear y devolver datos pero destaca getProyectedWayPoint que obtiene la proyección de un punto en el espacio sobre una la arista que contiene Neighbors (la que comparten un node con uno de sus neighbors). Esta función es la que se utilizará para decidir qué punto de paso toma un personaje de un triángulo a otro dado la arista que comparten. Cabe destacar, que esta función es la que se utilizará en la versión de **[Dynamic WayPoints]**, en la que los puntos se generan y se asignan de forma dinámica. Sin embargo, se realizó una primera aproximación en la que los puntos a asignar ya estaban precalculados. Esta primera versión está explicada en el apartado de **[Static WayPoints]**. Finalmente, el pseudocódigo de getProyectedWayPoint sería el que sigue:

```
Pre: Recibe una posición en un espacio 3D
Post: Retorna la proyección ortogonal del punto position sobre la
arista de Neighbors. En caso de no existir tal proyección, devuelve
el extremo de la arista más lejano al punto
getProyectedWayPoint(position)
res = []
extremo1Arista, extremo2Arista
si position se proyecta en la arista
      res = proyeccionOrtogonal
sino
      si position esta más cercano a extremo1Arista
            res = extremo2Arista
      sino
            res = extremo1Arista
```

Como se puede observar, cuando la proyección ortogonal queda fuera de la arista, se asigna el extremo de la misma más alejado a la posición del personaje. Pese a que eso pueda resultar anti intuitivo esto ayuda a repartir los personajes de forma natural a lo largo de la arista, gracias a que el cálculo de la proyección sobre la arista se recalcula cada pocos frames.

Una vez explicados como se van a representar los distintos elementos del grafo se detalla las funcionalidades del script GraphGenerator.cs. Este se ha utilizado para la generación del grafo el cual se ha decidido representar en un diccionario<int, Node>, donde el entero representa el identificador único de cada nodo y el Node el nodo en sí. **GraphGenerator** cuenta, como atributos de clase, el mismo grafo que vamos a crear y la triangulación del escenario a representar que obtendremos en la creadora. Esta triangulación nos la pasará el script del Astar.cs el cual se explicará en detalle en el apartado **[Movimiento a partir del A*].** A grandes rasgos, el algoritmo utilizado para la creación del grafo consiste en un recorrido sobre todos los nodos (triángulos del mapa) y obtener los adyacentes para cada uno de ellos. Este trabajo se ha realizado utilizando las funciones GenerateGraph, FindNeighbors y vertexShared. A continuación, se mostrará el pseudocódigo para las dos primeras funciones ya que la última función, vertexShared, es un recorrido por las coordenadas de dos triángulos y ver si dos coinciden para considerarlos vecinos.

```
Pre: Cierto
Post: El atributo Graph contiene todos los triángulos de la
NavigationMesh en forma de Nodos con los vecinos asignados
correctamente a cada uno de ellos
GenerateGraph()
indices = triangulizacion.Indices()
vertices = triangulizacion.vertices()

por cada indice i de indices
      List<Vector3> triangle = obtenerTriangulo(i, vertices)
      Nodo n = (i, triangle)
      Dictionary<int, Neighbors> neigh = FindNeighbors(i, triangle)

      si Graph contiene i entonces
            Graph[i].addNeighbors(neigh)
      sinó entonces
            Graph.add(i, n)
            Graph[i].addNeighbors(neigh)

      por cada par<int, Neighbors> (j, vecino) de neigh
            si Graph contiene j
                  Graph[j].addNeighbor(n)
            sinó
                  Graph.add(j, vecino)
                  Graph[j].addNeighbor(n)
```

Como se puede observar en el pseudocódigo, se aprovecha el hecho de que se descubre que "i" es vecino de "j" para crear la relación "j" es vecino de "i". De esta manera, los vecinos solo hace falta buscarlos de i en adelante i no hace falta buscarlos para un x<i ya que ya habría sido añadido. Realizando esta mejora o no, la complejidad del algoritmo es $O(n^2)$, siendo n el número de nodos, aun así para los órdenes de magnitud en los que se está trabajando si que se ha notado una pequeña mejoría.

```
Pre: Recibe index el cual es el índice a partir del que empezar a
buscar los vecinos de el triangle
Post: Retorna un diccionario<int, Neighbors> donde el entero
representa el identificador del nodo vecino y el Neighbors es el
nodo vecino al triangle de entrada
FindNeighbors(index, triangle)
neigh = []
indices = triangulizacion.Indices()
vertices = triangulizacion.vertices()
por cada indice i en indices a partir de index + 1
      triangle2 = obtenerTriangulo(i, vertices)
      puntosdeAdjacencia    =    vertexShared(i,    triangle,    index,
      triangle2)
      si puntosdeAdjacencia.length > 1
            Nodo n = (i, triangle2)
            neigh[i] = new Neighbors(i, n)
return neigh
```

Finalmente, a forma de resumen, la representación de las clases creadas para aplicar el A* sería la que se muestra en la **Figura 20.**

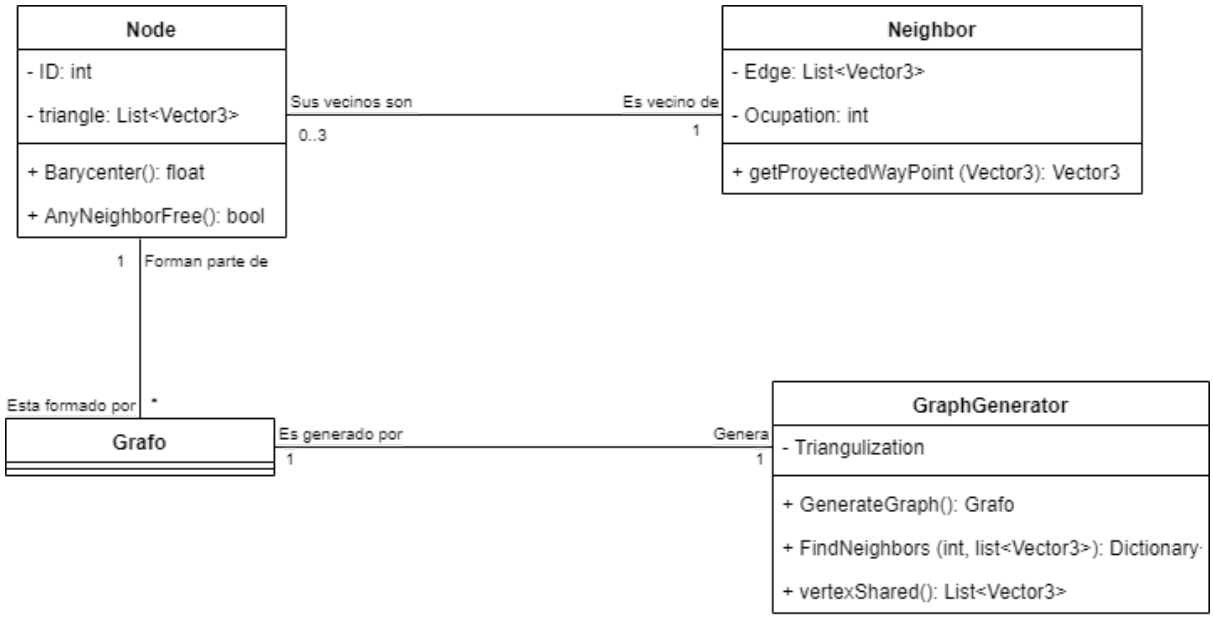


Figura 20. Diagrama UML con las clases necesarias para aplicar el A*.

Una vez finalizada la creación del grafo se decidió pintar todos los triángulos de este a modo de prueba para verificar la correcta representación del mapa. Esta muestra se puede ver en la **Figura 21**.

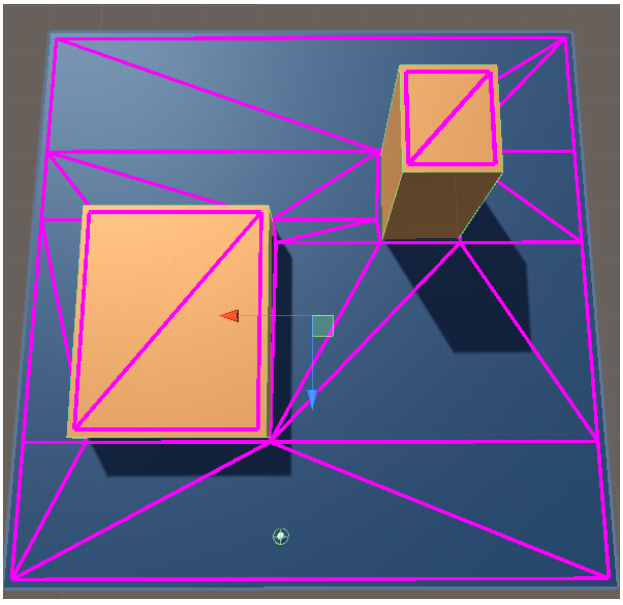


Figura 21. Dibujado de las aristas de todos los triángulos del mapa prototipo.

11.2 Movimiento a partir del A*

Una vez creadas todas las estructuras sobre las cuales se va a ejecutar el A*, se pasa a ver su implementación como tal, empezando por un repaso a los atributos de la clase y finalizando con los métodos más relevantes. Cabe destacar que tanto la implementación del Local movement como la del A* son independientes la una de la otra. Aún así, se decidió implementar este último algoritmo primero, ya que este es la base sobre la cual se implementará Static WayPoints y Dynamic WayPoints.

Toda la implementación del A* se ha realizado en un solo script **AStar.cs** con el único objetivo de que este nos retorne el camino a seguir por el personaje dado un punto inicial y final dentro de la malla de triángulos. Así pues, este script será usado por otro más genérico llamado **MarioMovement.cs**, el cual se explicara en los apartados **[Static WayPoints]** y **[Dynamic WayPoints]**, y será en este en el que el movimiento del personaje, frame a frame, sea implementado. El script se llama Mario en guiño al personaje de Nintendo ya que en esta versión single core los personajes que se mueven son Marios diseñados por el programador.

El script **AStar.cs** contiene como atributos más importantes el grafo que recibe de **GraphGenerator.cs** (en el formato Dictionary<int, Node>), el punto de inicio y final de la búsqueda del A*, la triangulación de la NavMesh y estructuras estrechamente relacionadas con el algoritmo A* que se explicarán a continuación. La mayoría de implementaciones de este algoritmo difieren en muy poco aspectos, aún así mencionar que la realizada para este proyecto se ha basado en el pseudocódigo de **[7]**. Teniendo esto en cuenta, las estructuras ligadas a este algoritmo, junto con su función y las estructuras de datos utilizados son las siguientes:

- **InvertedPath:** Consiste en un diccionario<int, Node> en el que la llave corresponde con el identificador de un nodo y el Node con el nodo anterior al de la llave. Como indica su nombre, si se recorre esta estructura se encontrara el camino buscado pero invertido. De

esta manera, partiendo del identificador del último nodo, podemos obtener el penúltimo nodo, y así hasta completar el recorrido.

- **OpenSet:** Consiste en un HashSet<int> el cual contiene contiene los nodos a los que tenemos acceso partiendo de los nodos visitados y aún no han sido tratados por el algoritmo.

- **ClosedSet:** Igual que en el caso anterior, esta implementado en forma de HashSet<int>. En esta estructura se guardarán lo identificados de los nodos ya tratados

- **gScore**: Consiste en un diccionario<int, float> donde el entero representa el identificador de un nodo y el float el coste de recorrer el grafo des del nodo inicial hasta el del identificador.

- **fScore**: Consiste en un diccionario<int, float> donde el entero representa el identificador de un nodo y el float el coste de recorrer el grafo des del nodo inicial hasta el final pasando por el nodo del identificador. El coste de ir de este nodo intermedio hasta el final se calcula haciendo uso de un heurístico como se explicará más adelante.

Para analizar la implementación realizada se dividirán los métodos utilizados en uno de principal y los secundarios. El principal és trianglePath que calcula el camino a recorrer y los secundarios destacables son distanceBetweenNodes,Intersect, heuristicCost y initialNode (análogo a LastNode). Para la fácil compresión del lector, se comenzará explicando el método trianglePath con su pseudocódigo:

```
Pre: Cierto
Post: Retorna una List<int> con los identificadores de cada uno de
los nodos s recorrer
trianglePath()

start = initialNode()
last = lastNode()

OpenSet.add(start)
gScore[start] = 0
fScore[start] = heuristicCost(Graph[start].barycenter)

mientras Openset no este vacio:
     #getMinimum obtiene el nodo con menos fScore del OpenSet (el
     que tiene más probabilidades para formar parte del camino
     mínimo)

     int current = getMinimum()
     si current es igual a last entonces
          reconstructPath(current)

     OpenSet.remove(current)
```

```
      ClosedSet.add(current)

      Dictionary<int, Neighbors> neigh = Graph[current].neighbors
      para cada uno de los neighbors n en neigh
            si n esta en ClosedSet entonces
                  pasamos al siguiente n

            #sino calculamos el potencial gScore de n
            float    potencialNewScore    =    gScore[current]    +
            distanceBetweenNodes(current, n)

            si n no estaba en OpenSet entonces
                  Openset.add(n.id)
            sinó si potencialNewScore >= gScore[n.id] entonces
                  este camino encontrado es más largo que otro que ya
                  conocemos por lo que pasamos al siguiente n

            #Si el algoritmo llega aquí es porque el camino de start
            a n es el mejor que conocemos
            invertedPath[n.id] = current
            gScore[n.id] = potencialNewScore
            fScore[n.id]          =          gScore[n.id]          +
            heuristicCost(Graph[n.id].barycenter)
#retornamos un camino vacío si no encontramos un camino entre start
y end
return new List<int>()
```

Como apunte final, es importante mencionar que las estructuras fScore y gScore son inicializadas en la creadora con un valor de infinito positivo para todos los nodos, de lo contrario el potencialNewScore siempre sería superior al gScore. La complejidad espacial y temporal del A* es exponencial en el peor de los casos dependiendo básicamente de la cantidad de nodos expandidos. Sin embargo, un buen heurístico es lo que limita la expansión de nodos innecesarios. Para nuestro caso, se ha utilizado un heurístico muy simple y bastante eficaz.

```
Pre: Recibe una posición de un espacio 3D.
Post: Retorna la distancia euclídea del punto recibido al punto de
final (el destino para el que se está calculando el camino)
heuristicCost(position)
return distancia(position, destinationPosition)
```

La estimación heurística se ha hecho a partir de la distancia euclídea, ya que en ninguno de los casos da una sobreestimación del camino real restante y es un cálculo muy rápido rápido de realizar. Es importante recordar que el heurístico no puede hacer una sobreestimación para ninguno de los nodos, porque de lo contrario se puede eliminar el camino mínimo del espacio de búsqueda.

Por otra parte, para el cálculo de la distancia entre dos nodos se han barajado un par de opciones. Estas dos, están representadas en la **Figura 22** y **23.** La primera calcula la distancia entre nodos por baricentros, es decir, dado que los nodos están representando triángulos, se calcula el baricentro de ellos y la distancia entre ellos es igual a la distancia euclídea entre los dos baricentros. Para el caso del segundo método, la distancia se calcula de arista a arista. Entrando más en detalle, esto quiere decir que si tenemos un camino formado los los triángulos {A B C} (en este orden), la distancia entre B y C es la distancia entre la arista entre los triángulos AB y BC.

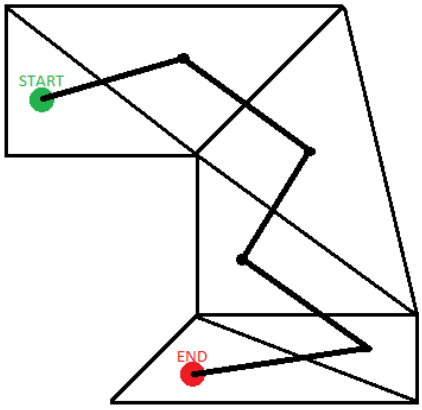


Figura 22. Distancia entre triangulos por baricentro.

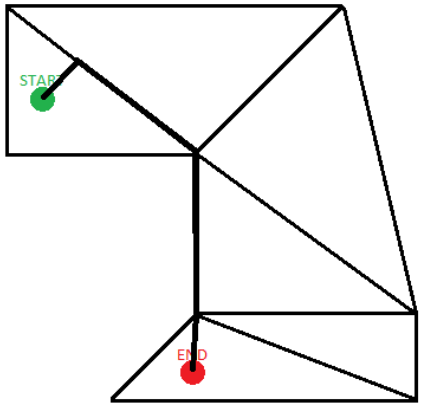


Figura 23. Distancia entre triangulos por arista.

```
Pre: Recibe el identificador del nodo actual a tratar y su nodo vecino en forma de Neighbor
Post: Retorna la distancia entre los baricentros del triángulo actual y su vecino
distanceBetweenNodes(current, neigh)
retorna distancia(Graph[current].barycenter,neigh.node.barycenter)
```

Finalmente, se utilizó la distancia por baricentros ya que era la más rápida de calcular y sencilla de implementar como se puede ver en el pseudocódigo anterior. Además tampoco es el objetivo del proyecto el cálculo de caminos mínimos y con la solución implementada se obtienen los caminos mínimos para la mayoría de casos.

Para acabar, se mostrará el pseudocódigo de la función InitialNode y Intersect respectivamente. El primer método no es demasiado interesante, pero se ha incluido porque es en este (initialNode y LastNode también) donde se utiliza la función Intersect, la cual sí que es destacable.

```
Pre: Cierto
Post: Retorna el identificador del nodo inicial
InitialNode()
```

```
#se crea un rayo que va desde la posición inicial del personaje hacia abajo
Ray r = new Ray(originPosition, down)
para cada nodo n en Graph
      Vector3 p1 = n.triangle[0]
      Vector3 p2 = n.triangle[1]
      Vector3 p3 = n.triangle[2]
      si Intersect(p1,p2,p3, r) retorna n.id
return -1
```

Como se puede observar, se trata simplemente de un recorrido lineal por todos los nodos en búsqueda del que haga intersección con un triángulo de la malla. No queda explícito en la precondición de la función, ya que no es parámetro de entrada, pero originPosition tiene que ser una posición que esté sobre alguno de los triángulos creados por la NavMesh.

```
Pre: Recibe p1, p2, p3 que constituyen los tres vértices de un triángulo del grafo y "r" un rayo con posición origen del personaje y direccion down(respecto el eje vertical en WorldSpace)
Post: Retorna cierto si el rayo intersecta con el triángulo recibido o falso en caso contrario.
Intersect(p1,p2,p3,r)
Vector3 e1 = p2 - p1
Vector3 e2 = p3 - p1

p = Vector3.Cross(r.direccion, e2);
det = Vector3.Dot(e1, p);

si det cercano a 0:
      entonces es paralelo al plano del triangulo por lo tanto
      retornamos falso
#Vector t = origen-p1
t = r.origen - p1
#Parametro u del algoritmo
u = Vector3.Dot(t, p) * invDet

si u no es cero
      entonces no hay intersección

#Parametro v del algoritmo
q = Vector3.Cross(t, e1)
v = Vector3.Dot(ray.direction, q) * invDet

si v es menor que cero o la suma de v+u es mayor que 1 entonces
      no hay intersección
si el dot product de e2 y q por la inversa de det es mayor que cero
      entonces si que hay intersección
```

```
#Si llegamos aquí no ha habido intersección
retorna falso
```

Esta implementación de la intersección rayo-triángulo está basada en el Möller–Trumbore intersection algorithm **[20]**. Se ha elegido esta ya que es la única implementación con un resultado correcto que no necesita el precálculo de la ecuación del plano que contiene el triángulo. Además, su uso es frecuente en la implementación de algoritmos de computación de gráficos como el famoso ray tracing.

11.3 Static WayPoints

Una vez generado el camino de triángulos que tienen que seguir los personajes con el A*, hace falta definir qué puntos de este camino de triángulos recorren cada uno de ellos. Esto significa decidir sobre qué punto de la arista compartida entre dos triángulos transita nuestro personaje. Este punto junto con el de Local movement son los más importantes ya que son los implementan el movimiento físico del personaje. Por lo tanto, no se puede asignar a todos los personajes un mismo punto de paso (ejemplo: el medio de la arista, uno de los extremos, etc.), porque sino se caería en el mismo error en el que incurre la solución de Unity.

Así pues, por parte del director de proyecto y la codirectora, se sugirió calcular la proyección ortogonal del personaje sobre la arista y que este sea su WayPoint (punto de paso). Esto es una muy buena solución ya que, gracias al local movement, los personajes tratarán de evitarse los unos a los otros y la proyección ortogonal sobre la arista de cada uno de los personajes debería de ser única (aproximadamente). En la **Figura 24** y **25** podemos ver un ejemplo de esto, dos personajes con una misma proyección gracias al Local Movement acabarán eligiendo un punto distinto.

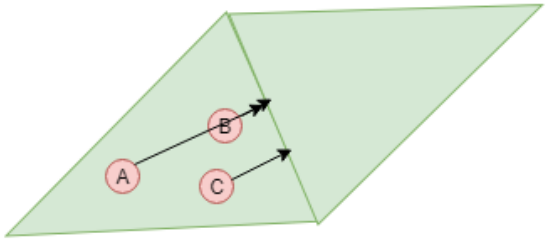


Figura 24. Los personajes A y B comparten el mismo WayPoint ya que tienen la misma proyección.

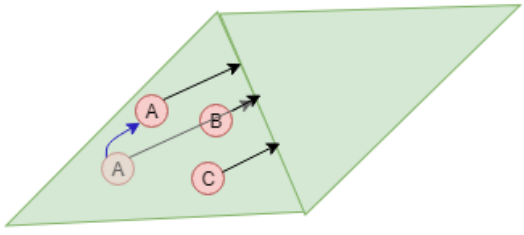


Figura 25. Los personajes A y B obtienen un WayPoint diferente de forma natural por el LocalMovement.

Como añadido, aún sin tener implementado el Local movement se debería obtener un resultado bastante bueno debido a que los personajes cuentan con el componente de Rigidbody de Unity el cual evita que los personajes se superpongan en el espacio. Sin embargo, antes de llevar a cabo íntegramente la implementación propuesta, se decidió probar un método presentado por el alumno el cual consiste en tener estos puntos de paso ya precalculados. Este método que llamaremos **Static WayPoints**, son la primera aproximación de movimiento de los personajes y será el que se describa junto a su pseudocódigo a lo largo de este apartado.

Para **Static WayPoints**, todas y cada una de las aristas que hay en el mapa se dividen en $\frac{distancia(extrem1, extrem2)}{2r}$ puntos, donde extrem1 y extrem2 son los extremos de la arista a subdividir y r el radio de los personajes. Esta subdivisión se crea en el mismo momento que se crean los neighbors de cada nodo, es decir, cuando se crea el grafo inicialmente y se guardan los puntos generados en una lista de vector3 llamada WayPoints. A partir de esto, cada vez que un personaje requiera un punto de paso, a este se le asignará el más cercano de la lista wayPoints. La gran ventaja que nos da este método, es que se puede controlar en todo momento qué WayPoints están siendo utilizados por otro personaje y por lo tanto, asignarle uno que esté libre al personaje que solicite un punto de paso. Gracias a esto, ya no haría falta el cálculo del local movement para que los personajes elijan wayPoints distintos. A primera vista, este método parece más sencillo de calcular ya que solo se requeriría de buscar el punto con la distancia euclidia más cercana al personaje, pero rápidamente se observan varios problemas con el uso de puntos estáticos.

Los principales problemas por los que finalmente se descartó son el claro overhead en memoria, cálculos más sencillos pero en gran cantidad y un resultado visual que no justificaba estos problemas. Para el caso de nuestro escenario, que solo cuenta con unos pocos polígonos, no hay problema, pero para mapas considerablemente grandes en los que haya cinco mil triángulos, mantener y calcular la lista de waypoints podría ser excesivo. Además, hay que tener en cuenta que el número de wayPoints que se crearan depende también del radio del personaje que queramos mover. Por lo tanto, en un escenario con muchos triángulos y muy grandes con respecto los personajes como una gran llanura y/o un desierto, esta solución podría sufrir bastante.

El segundo problema que aparece, es que a pesar de que asintóticamente el coste temporal de consultar un nuevo waypoint con el método estático es constante (no depende del número de personajes y/o del número de triángulos), en la práctica sí podría provocar una bajada de frames. Calcular la distancia entre un WayPoint estático y una posición es una operación muy rápida en Unity, sin embargo hay que tener en cuenta que esta se repetirá tantas veces como WayPoints hayamos creado para una arista. Suponiendo el mejor de los casos en el que solo hubiera dos puntos de paso sobre los que buscar para una arista, el número de operaciones seria practicamente el mismo que con el método dinámico. Por lo tanto, en la gran mayoría de casos, se harán menos cálculos con este último método. Finalmente, el resultado visual era muy parecido al de **Dynamic WayPoints** con **Local Movement** por lo que no valía la pena continuar con este método.

Explicado en qué consiste y porque se descartó Static WayPoints, se pasa a ver cómo se implementó en el proyecto. Este método está desarrollado de forma compartida entre el script MarioMovement.cs y Neighbors.cs. Esto es así, porque tal y como se mencionó en el apartado **[Implementaciones previas]**, el método que retorna el WayPoint dado un nodo y su vecino de paso

está implementado en la clase Neighbor. Para esta versión, en la que los puntos de paso son estáticos, getProyectectWayPoint se llamaba getClosestWayPoint y tenía el siguiente pseudocódigo:

```
Pre: Recibe una posición en un espacio 3D
Post: El WayPoint más cercano a la posición position y el entero
pointID contiene el índice del wayPoint más cercano.

getClosestWayPoint(position,pointID)
int index = -1
float minDist = InfinitoPositivo
para cada waypoint p en WayPoints
      si p esta libre entonces
            aux = distancia(position, p)
            si minDist es mayor que aux entonces
                  index = p.indice
                  minDist = aux
            p esta ocupado
pointID = index
retornar WayPoints[index]
```

Como se puede observar, se trata de un simple recorrido por todos los wayPoints en busca del más cercano. Así pues, ahora se pasa a presentar el pseudocódigo de las funciones más importantes del script MarioMovement.cs, GoToNextPoint y AsignNewPoint, para visualizar como se ha implementado **Static WayPoints**. En esta ocasión, no se hará un repaso por los atributos más importantes de la clase ya que son demasiados y la mayoría son variables de control. Aun así, destacar que el script MarioMovement.cs consta de una lista con los identificadores de los triángulos a recorrer, el índice actual de esta misma lista y una referencia a un script de consulta sobre el grafo del mapa. La lista de triángulos a recorrer se obtiene gracias al script AStar.cs y, a medida que se van atravesando los distintos triángulos del camino, se va actualizando el índice de esta lista. Por otro lado, este último script que utiliza PlayerMove.cs, se llama GraphData.cs y es donde hay guardado una referencia del grafo del mapa a la que todos los personajes pueden acceder ya sea para leer o para escribir. Finalmente, dejar claro que en esta representación, se omitirá código relacionado con otras tareas como las de animar el personaje o recalcular el camino en caso de mucho tráfico (este método descartado se explicara en su correspondiente sección **[Recálculo del camino mínimo]**).

```
Pre: Cierto
Post: Mueve el personaje en la dirección que pertoque.
GoToNextPoint()
      si estamos en el último triángulo y cerca del destino
            entonces
                  nos detenemos
      sinó si estamos en el último punto y lejos del destino
            entonces
                  #Al llegar al último triángulo, el NextPoint pasa a
                  ser el destino final
                  nos movemos al NextPoint
```

```
      sinó si estamos en el primer triángulo o cerca de nuestro
      NextPoint
            entonces
                  AsigNewPoint()
      sinó
            entonces
                  nos movemos al NextPoint
```

Cabe recalcar, para este punto de la implementación el movimiento al NextPoint está implementado con la función MoveTowards **[21]** de Unity que desplaza un gameObject en una dirección cierta distancia determinada por un parámetro de entrada. En el apartado de **[Local movement]** se verá cómo la función MoveTowards es substituida por el algoritmo de movimiento local.

```
Pre: Cierto.
Post: Asigna un NextPoint correspondiente a la arista entre al
triángulo actual al que se acaba de llegar y el siguiente.
AsignNewPoint()
si no estamos en el último triángulo
      entonces
            #Se mira si hay wayPoints libres
            bool    anyFreeWayPoint    =    GraphData.anyFreeWayPoint(
            indexPath,indexPath+1)
            si anyFreeWayPoint es cierto
                  entonces
                  si FirstTriangle == False
                        entonces
                              GraphData.liberateWayPoint(trianglePath[i
                              ndexPath-1], trianglePath[indexPath])
                              GraphData.liberateWayPoint(trianglePath[i
                              ndexPath], trianglePath[indexPath-1])
                  sinó
                        FirstTriangle = false
                  NextPoint = GraphData.getProyectedWayPoint(
                  trianglePath[indexPath], trianglePath[indexPath+1])
                  ++indexPath

sinó
      NextPoint = destinacionFinal.position
```

Se puede observar que el script GraphData se utiliza para consultar la disponibilidad de una arista (anyFreeWayPoint(indexPath,indexPath+1)) y para actualizar el estado de ésta en caso de liberar la ocupación del punto anterior. La función getClosestWayPoint de forma interna actualiza el estado del punto obtenido a ocupado. Como apunte final, es importante destacar que se han omitido varias variables de control para hacer el codigo mucho mas entendible. Aún así, el código completo de todo el método se encuentra en el script MarioMovement.cs.

11.4 Dynamic WayPoints

Viendo los problemas que planteaba la versión con puntos de paso estáticos, se decidió optar por la versión dinámica junto con la implementación de Local Movement. Antes de comenzar con la descripción de cómo se ha implementado este método, se recordará, brevemente ya que se ha ido explicando el concepto a lo largo de todo el proyecto, en qué consiste la implementación de puntos de paso dinámico.

La idea básica es calcular la proyección ortogonal de cada personaje sobre la arista entre el triángulo actual en el que está situado el personaje y el siguiente en su camino calculado con A*. Este punto será el que utilizará el personaje como punto de paso para llegar al siguiente triángulo. Una vez calculado, este se irá recalculando cada x frames (donde x es un random entre diez y quince) ya que junto con la implementación de Local Movement (explicado en más detalle **[Local movement]**), la proyección del personaje sobre la arista irá cambiando. Sin embargo, como todos los personajes tienen el el componente Rigidbody adherido, no se pueden solapar entre ellos por lo que de forma natural, sin uso de Local Movement, ya tendrán una proyección diferente. Aún así, el resultado final, debería ser mucho mejor con el método Local Movement implementado. Para el recálculo del punto de paso se utiliza un número aleatorio para añadirle naturalidad al movimiento. Si no se hiciera uso de la aleatoriedad, se vería que como para cada lapso de tiempo, todos los personajes giran su dirección a la vez, restándole realismo a la implementación. Ahora sí, para explicar como se ha implementado **Dynamic WayPoints** se utilizará pseudocódigo al igual que en todos los casos anteriores.

El método de puntos de paso dinámicos se ha implementado entre el script Neighbor.cs y MarioMovement.cs al igual que su análogo, puntos de paso estáticos. En la clase Neighbor se encuentra la función getProyectedWayPoint que es la misma que está explicada en el apartado **[Implementaciones previas]**. Esta clase, es la que se utiliza para consultar/calcular el punto de paso entre un nodo y su vecino. Por otro lado, para llevar al personaje a este punto calculado, actualizando su posición a cada frame, se utiliza el script MarioMovement.cs. Este script, parte del de mismo nombre que se utilizó para la implementación del método estático. Por este motivo, dejando de lado variables de control, los atributos y los métodos más relevantes son los mismos. Esto quiere decir, que al igual que el caso anterior, el script MarioMovement.cs cuenta, como atributos más importantes, con el camino de triángulos a recorrer en una lista, un entero que hace de índice actual de dicha lista y una referencia a la clase AStar. De la misma manera, se analizarán las funciones GoToNextPoint y AsignNewPoint respectivamente con el siguiente pseudocódigo:

```
Pre: Cierto
Post: Mueve el personaje en la dirección que pertoque.
GoToNextPoint()
      si estamos en el último triángulo y cerca del destino
            entonces
                  nos detenemos
      sinó si estamos en el último punto y lejos del destino
            entonces
                  #El NextPoint pasa a ser el destino final
                  nos movemos al NextPoint
```

```
      sinó si estamos en el primer triángulo o cerca de nuestro
      NextPoint
            entonces
                  AsigNewPoint()
      sinó si recalculate <= 0
            entonces
                  recalculate = Random.range(10,15)
                  recalculateWayPoint()
      sinó
            entonces
                  nos movemos al NextPoint
                  ++recalculate
```

Si hacemos una comparativa con su versión estática, tenemos que realizamos las mismas acciones para todos los casos, excepto para uno. Como se explicó anteriormente, cada x frames se iba a recalcular el punto de paso para reajustar la proyección del personaje con su posición real. Por eso mismo, donde antes siempre nos movíamos a un punto fijo, ahora este irá cambiando en función del valor de la variable recalculate como se puede ver en el pseudocódigo.

```
Pre: Cierto.
Post: Asigna un NextPoint correspondiente a la arista entre al
triángulo actual al que se acaba de llegar y el siguiente.
AsignNewPoint()
si no estamos en el último triángulo
      entonces
            FirstTriangle = false
            NextPoint = GraphData.getProyectedWayPoint(
            trianglePath[indexPath], trianglePath[indexPath+1])
            ++indexPath
sinó
      NextPoint = destinacionFinal.position
```

Para acabar, se puede observar que el código de AsignNewPoint se simplifica bastante, ya que solo hay que actualizar el índice de la lista de triángulos a recorrer y calcular el nuevo wayPoint. No hay que actualizar la ocupación de los puntos de paso ya que no es algo que nos importe (dos puntos no tendrán la misma proyección por debido al **Local Movement**).

11.5 Local movement

En este apartado se describe extensamente en qué consiste **Local Movement** y como se ha implementado en el proyecto. **Local Movement** ha sido el último de todos los algoritmos a implementar, sin contar la adaptación del código al modelo paralelo (**[ECS]**). Se podría haber implementado este algoritmo antes que Dynamic WayPoints pero no tenía demasiado sentido ya que no se tenían calculados qué puntos de paso tomar.

Primeramente, hay que destacar que a lo que se hace referencia cuando se habla de **Local Movement** es al conjunto de dos algoritmos. Se habla de ellos en conjunto, debido a que uno de ellos se ha implementado de forma indirecta y queda más claro para el lector hablar conjuntamente de los mismos. Estos dos son el **seek** y el **obstacle avoidance**.

El primero consiste en, dado un personaje y un destino, la implementación del movimiento de este recorrido. Esto ya se ha implementado indirectamente en las dos versiones de cálculo de puntos de paso. En nuestro caso, se ha implementado haciendo uso de la función MoveTowards(origin, target, step) de Unity que permite definir la traslación de un objeto de un punto inicial a uno de objetivo definiendo la cantidad de movimiento máximo por frame (step). También se podría haber modificado directamente el componente posición del personaje (componente que comparten todos los objetos de Unity), obteniendo el mismo resultado. Sin embargo, no hay que olvidar que también es necesario orientar la rotación del personaje a su destino. Si tomamos el vector forward de un personaje como aquel hacia el que está observando el frente del mismo, este vector debe estar apuntando siempre a su destino. Dado que el recálculo de los WayPoints se produce de forma relativamente frecuente (entre 10 y 15 frames), la orientación del personaje irá cambiando bastante antes de pisar el punto de paso objetivo. Por este motivo, para evitar el efecto de parpadeo en el vector forward del personaje, se ha optado por hacer una interpolación entre el vector forward actual y el que apunta el destino suavizando así la rotación del personaje.

El segundo algoritmo, **obstacle avoidance**, es algo más complejo y sobre lo que sí había que hacer una implementación de cero. Mencionar, que el algoritmo se ha implementado partiendo de la base explicada por Craig W. Reynolds en el artículo **[9]**, adaptando su desarrollo a las soluciones que nos ofrece Unity.

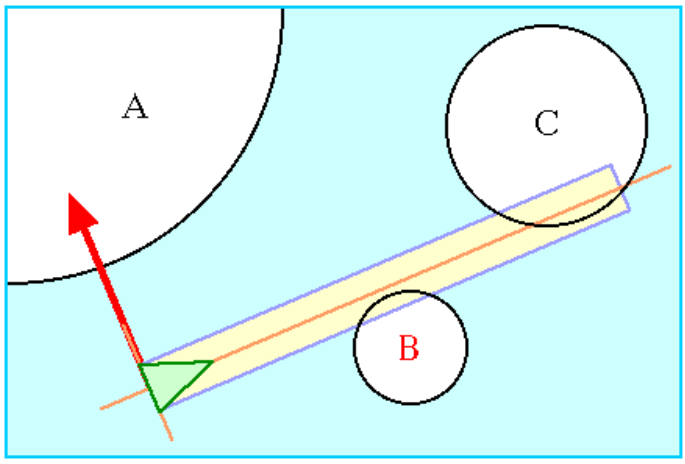


Figura 26. Obstacle Avoidance.

Obstacle avoidance, se basa en intentar rodear los obstáculos que se encuentran entre tu posición y tu destino, es decir, los obstáculos que aparezcan mientras que se ejecuta el **seek**. Para ello, el personaje, haciendo uso de un funnel de visión, detecta los personajes dentro de este y cambia su dirección en consecuencia. Como se puede observar en la **Figura 26**, el personaje (el triángulo verde), detecta dentro de su rango de visión los obstáculos “b” y “c”, tomando como prioridad a evitar el obstáculo “b” ya que es el más cercano. Si se observa el obstáculo “a”, este no se tomará en cuenta por el personaje para desviar su dirección, ya que para **obstacle avoidance** todo obstáculo

que se encuentre fuera del funnel de visión que creemos no es de interés. Como aclaración, el funnel de visión consiste en una región en el espacio donde el personaje puede detectar los obstáculos, su función básicamente es la de simular la visión del mismo.Para el proyecto se definirá como obstáculo para un personaje, el resto de personajes que hayan en escena.

Partiendo de esta definición para **obstacle avoidance,** se puede ver que hay dos elementos en la implementación que, más allá de definir cómo se van a trasladar a nuestro proyecto, son totalmente personalizables. Estos son la profundidad del funnel de visión y cuánto se modifica la dirección inicial hacia el WayPoint en relación con la nueva dirección de giro calculada. Por este motivo, la implementación del **obstacle avoidance**, se ha realizado teniendo en cuenta esto, y existe un parámetro para escalar a voluntad el funnel de visión y otro que se utiliza para interpolar la dirección inicial con la nueva calculada, estos se llaman scaleVisionW y directionW respectivamente.

El primero se utiliza únicamente cuando se va a instanciar un personaje teniendo por defecto el valor de uno, si incrementamos o decrementamos este valor, el funnel de visión del personaje será más o menos alargado, pudiendo ver más o menos lejos. No se permite cambiar parámetro este parámetro durante la vida de un personaje debido a que supondría provocar un movimiento demasiado aleatorio para los personajes; además de que el funnel de visión no está pensado para que vaya cambiando de tamaño durante la ejecución del algoritmo.

Para el segundo parámetro, directionW, se pensó inicialmente en que este contendría un float entre 0 a 1 (significando tener un valor de 0 no girar nada respecto al obstáculo de enfrente y un valor de 1 olvidarse de la dirección inicialmente calculada para rodear el obstáculo con el mayor ángulo de giro posible), que no debía cambiar durante toda la vida del personaje al igual que scaleVisionW. Sin embargo, debatiendo con los directores del proyecto se decidió que un uso más elegante y realista de directionW sería ir variando su valor dependiendo de la proximidad del obstáculo. Por lo tanto un obstáculo lejano a nosotros tendría un valor cercano a 0 y por el contrario si el obstáculo estuviera muy cerca directionW valdría practicamente 1. Para entender mejor la idea sobre la que se está hablando, se mostrarán las **Figuras 27** y **28**.

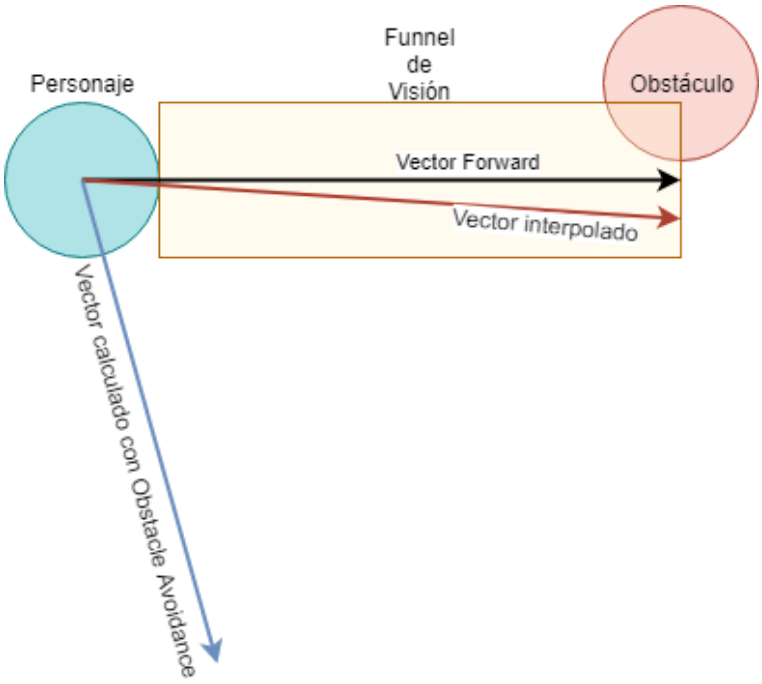


Figura 27. Obstacle avoidance con directionW variable. El vector que seguirá el personaje (Vector interpolado), casi no se ve afectado por el obstáculo ya que se encuentra lejos de él.

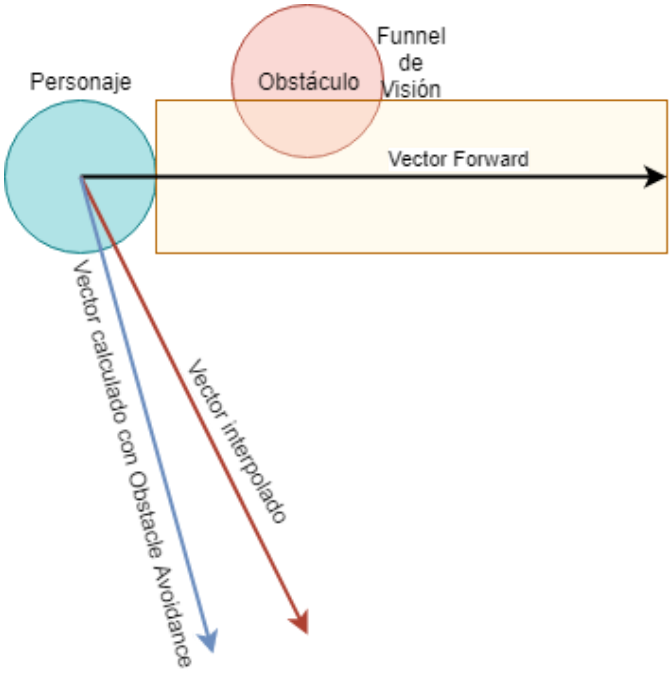


Figura 28. Obstacle avoidance con directionW variable. El vector que seguirá el personaje (Vector interpolado), está totalmente condicionado por el vector calculado por obstacle avoidance ya que el obstáculo se encuentra próximo al personaje.

Como se puede ver, la idea es bastante intuitiva y se corresponde bastante a la interacción real que podrían tener humanos entre ellos. Explicados estos dos parametros, scaleVisionW y directionW, que

si bien forman parte del algoritmo en sí, son algo más “personalizables” y su implementación queda más a ojo del programador, ahora se pasa a explicar como se ha implementado **obstacle avoidance**. Esto incluye la explicación de cómo se han integrado estructuras conceptuales como el funnel de visión para detectar a personajes en Unity, así como la parte troncal del algoritmo (cálculo de la nueva dirección de giro) y pequeños añadidos para mejorar visualmente la ejecución del mismo.

Toda la implementación de **obstacle avoidance** se ha realizado en el script MarioMovement.cs ya que su ejecución sólo aparece cuando nos queremos mover de nuestra posición a un punto de paso y siempre que haya un obstáculo en nuestro funnel de visión. Para desarrollar este método, se diferencian dos partes importantes, la primera es cómo se va a simular el funnel de visión y así detectar los obstáculos de forma eficiente y la otra es como se va a calcular el vector de giro.

Para el primero se ha optado por el uso de un BoxCollider, estos son componentes de Unity especialmente diseñados para detectar las colisiones. También se barajó la idea de utilizar la función overlapBox **[22]**, la cual nos permite detectar todos los colliders dentro de un cubo que se defina en el espacio y nos los devuelve en una lista de referencias a ellos. El problema de esta función, es que el cubo/rectángulo creado no se puede rotar junto al personaje debido a como Unity representa las coordenadas de este cubo. En contraposición usando un BoxCollider no hace falta preocuparse de este problema ya que al añadir este componente como hijo del personaje al que va adherido, las rotaciones que se le aplican al padre las recibe igualmente el hijo. Como añadido final, cabe destacar que tanto el SphereCollider como el BoxCollider son de los dos colliders donde es más eficiente el cálculo de colisiones debido a la sencillez de las figuras que representan. En la **Figura 29**, se puede ver un rectángulo delante del personaje, el cual corresponde con los BoxCollider de Unity. También se puede observar un CilinderCollider que sirve para simular la figura del personaje, el cual será el que utilice un BoxCollider para detectar un personaje.

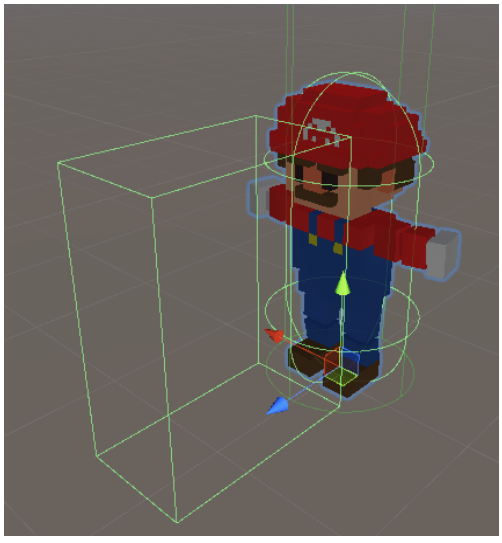


Figura 29. Personaje con un BoxCollider para la detección de otros personajes.

Antes de explicar cómo se realizan los cálculos del nuevo vector de giro, se necesita aclarar cómo un personaje obtiene la posición de otro a partir de los Colliders. Cada personaje, cuenta con dos colliders como se puede ver en la **Figura 29**, el primero un CilinderCollider que sirve para delimitar las dimensiones del propio personaje y el segundo un BoxCollider que hace la función de funnel, tal y

como se ha mencionado antes. De esta manera, los BoxCollider se utilizan para detectar los CilinderCollider del resto de personajes, y con esto su posición (ya que es la misma que la del collider). Aunque estas estructuras nos habilitan para detectar cuando un personaje entra en nuestro espectro de visión y cuando no, no nos da una lista de todos ellos o una forma explícita para obtener el más cercano. Por lo tanto, esto nos obliga a crear un script extra llamado FunelScript.cs, el cual se añadirá como componente a los BoxColliders de los personajes para mantener una lista de los personajes que están dentro de este collider. El Script en si es muy sencillo, contiene como atributo la lista de los colliders dentro del BoxCollider y esta lista se va actualizando siempre que el sistema llame a las funciones OnTriggerEnter (se llama cada vez que un collider entra en nuestro collider) y OnTriggerExit (se llama cada vez que un collider sale de nuestro collider). Por otra parte hace falta mencionar, que Unity utiliza los CilinderCollider junto con el componente Rigidbody para evitar que las mallas de los personajes se superpongan de forma automática sin necesidad de implementaciones externas.

Para el cálculo de la nueva dirección de giro, se toma en cuenta solamente el obstáculo más cercano al personaje como se ha mencionado. Esto es debido, a que se pretende aplicar un algoritmo de movimiento local y no global, teniendo como consecuencia sus pros y sus contras. El mayor contra frente al uso de un algoritmo de movimiento global, es la naturalidad del movimiento final. Esto es debido a que cualquier algoritmo global tiene en cuenta todos los personajes (o la mayor parte de ellos) en escena para calcular la mejor dirección de giro a tomar, mientras que uno de local solo unos pocos (los observables en el funnel, en concreto, el más cercano). Sin embargo, esto genera un contra para el algoritmo global inasumible si vamos a tratar grandes cantidades de personajes. Este es que el cómputo por frame de un algoritmo global crecerá de forma lineal con el número de personajes en escena. Por el contrario con obstacle avoidance por muchos personajes que tengamos en escena, como mucho siempre procesaremos los que hayan dentro del funnel de visión.

Explicado el porqué se utiliza solo se utiliza el personaje más cercano al que nos encontramos para el cálculo del vector de giro, se detalla cómo se realizan los cálculos de obstacle avoidance ayudándonos de la **Figura 30** para una mejor comprensión.

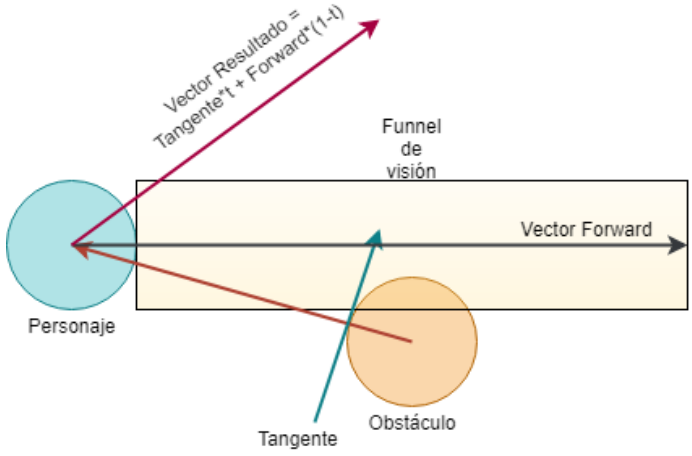


Figura 30 Cálculo de la nueva dirección para obstacle avoidance.

Ordenando de forma cronológica los pasos a seguir para conseguir el vector resultado, estos serían los siguientes:

- Se calcula el vector dirección del obstáculo al personaje.
- A partir de este vector dirección se hace el producto vectorial entre este y el vector up (0,1,0) y también con el vector down (0,-1,0). Como resultado de estas dos operaciones obtendremos dos tangentes que llamaremos tangente1 y tangente2.
- Se calcula cual de los dos crea un ángulo menor con el vector forward (si el obstáculo está a nuestra derecha, lo rodearemos por la izquierda y viceversa). Este nuevo vector será la tangente definitiva.
- Se interpolan linealmente el vector tangente y el forward haciendo uso del parámetro directionW (este tendrá un valor entre 0 y 1).
- La interpolación de los vectores tangente y forward es la que se utiliza como nuevo vector forward para calcular la nueva posición del personaje.

Finalmente solo queda ver, haciendo uso de pseudocódigo, como se ha integrado en el script MarioMovement.cs el cálculo del ángulo de giro. Como bien se ha dicho, **obstacle avoidance** solo se ejecuta cuando un personaje transita entre un WayPoint a otro y se encuentra un obstáculo (otro personaje), por lo tanto, esto significa que se tendrá que modificar la función *GoToNextPoint* de **Dynamic WayPoints** (ya que esta es la versión final sobre la que se está trabajando)

```
Pre: Cierto
Post: Mueve el personaje en la dirección que pertoque.
GoToNextPoint()
      si estamos en el último triángulo y cerca del destino
            entonces
                  nos detenemos
      sinó si estamos en el último punto y lejos del destino
            entonces
                  #El NextPoint pasa a ser el destino final
                  nos movemos al NextPoint
      sinó si estamos en el primer triángulo o cerca de nuestro
      NextPoint
            entonces
                  AsigNewPoint()
      sinó si recalculate <= 0
            entonces
                  recalculate = Random.range(10,15)
                  recalculateWayPoint()
      sinó
            entonces
                  #versión antigua
                  nos movemos al NextPoint (haciendo una simple
                  traslación del personaje)
                  #Local Movement
                  move()
```

```
                ++recalculate
```

Como se puede observar, para la situación en la que simplemente nos tenemos que mover hacia el NextPoint, se ha modificado la translación en línea recta del personaje por la función *move,* la cual es algo más compleja y incluye el **obstacle avoidance**. A continuación el pseudocódigo de dicha función, que incluyen tanto el **obstacle avoidance** como añadidos que mejoran un poco el resultado visual del movimiento.

```
Pre: Cierto
Post: Mueve el personaje hacia NextPoint. En caso de encontrarse
frente a un obstáculo aplica obstacle avoidance.
move()
      count = 0
      Mindistance = InfitoPositivo
      obstaculo = null
      hitColliders = FunnelScript.hitColliders
      para cada uno de los colliders c en hitColliders
            distancia = distancia(miPosición,c.posición)
            ++count
            si distancia < Mindistance entonces
                  minDistance = distancia
                  obstaculo = c
      #Si encontramos un obstaculo
      si Mindistance < InfitoPositivo entonces
            dot = Dot(transform.forward, obstaculo.forward)
            si dot > 0.8 entonces #el obstáculo tiene la misma
            dirección que nosotros
                  si count < 3 y el obstáculo está "muy cerca de
                  nosotros" entonces
                        frenamos
                  si count < 3
                        aceleramos sin sobrepasar SpeedMax
                  sinó #tenemos muchos personajes delante
                        frenamos
                  float step = Speed * Time.deltaTime
                  miPosición = moveTowards(miPosición, NextPoint,
                  step)
                  miRotación = interpolación entre la rotación
                  anterior y la de la nueva dirección calculada

            sinó entonces #se entiende que el obstáculo tiene una
            dirección contraria a la nuestra
                  directionW = 1-(minDistance/FunnelMaxDist)
                  directionW = min(1, max(0, directionW))
                  normal = Normalize(miPosción-obstaculo.position)
                  tangent1 = Cross(normal, up)
```

```
                tangent2 = Cross(normal, down)
                tangent = min(Angle(tangent1, forward),
                Angle(tangen2, forward))
                finalDir = tangent*directionW
                otherDir = obstaculo.finalDir
                si otherDir tiene el mismo signo que finalDir
                entonces
                    finaldir = -1*finalDir #ya que sino giraremos
                    en la dirección que lo hace el obstáculo
                finaldirection = forward*(1-directionW) + finalDir
                float step = Speed * Time.deltaTime
                miPosición = moveTowards(miPosición, finaldirection
                + miPosición, step)
                miRotación = interpolación entre la rotación
                anterior y la de la nueva dirección calculada
                recalculateWayPoint()
        sinó entonces #no tenemos obstáculos delante
            si Speed no supera MaxSpeed entonces
                Speed = Speed + Time.deltaTime
            finalDir = (0,0,0)
            float step = Speed * Time.deltaTime
            miPosición = moveTowards(miPosición, NextPoint, step)
            miRotación = interpolación entre la rotación anterior y
            la de la nueva dirección calculada
```

Dado lo extenso que es el pseudocódigo, aun omitiendo varias partes relacionadas con el movimiento del personaje (por ejemplo evitar que este se salga del mapa tras aplicar **obstacle avoidance**), este se ha dividido en tres secciones por colores como se ha podido ver. Tanto la sección roja como verde se corresponden con acciones a realizar cuando hemos detectado un personaje/obstáculo en el funnel de visión, mientras que la azul se aplica cuando no hay personajes enfrente del nuestro.

La sección roja tiene que ver con esos añadidos que se mencionan antes de presentar el pseudocódigo. La idea tras esta sección es que no tiene sentido intentar rodear un obstáculo/personaje que lleva la misma dirección que tú. Aunque intuitivamente no sea algo descabellado, el problema reside en que el concepto de *adelantar a un obstáculo* requiere de realizar una maniobra demasiado compleja y el algoritmo está pensado para simplemente rodear un obstáculo. Así pues, en esta sección nos seguiremos moviendo rumbo nuestro punto de paso asignado en ese momento y modificaremos nuestra velocidad en base a la cantidad de personajes que tengamos delante y su proximidad. Por lo tanto, si tenemos pocos personajes pero el que tenemos delante está muy cerca nuestro personaje frenará (como lo haría una persona para no chocar con la de delante). Por otra parte, si hay pocos seguirá su velocidad como normalmente y finalmente si hay relativamente muchos delante de él, frenará. Cabe destacar que la acción de frenar nunca permitirá que la velocidad del personaje baje a 0 para evitar posibles deadlocks en ciertas situaciones.

Por su parte, la sección verde es la que se corresponde con **obstacle avoidance** propiamente. Para no entrar en repeticiones, en la sección se realizan las acciones mencionadas anteriormente en la explicación de la **Figura 30**, más un añadido extra. Este último, también relacionado con los añadidos comentados al inicio, consiste en cómo se calcula el atribulo finalDir. Este atributo representa el ángulo de giro que se ha calculado usando **obstacle avoidance** (es decir, la tangente) multiplicado por la ponderación de directionW. Este se ha añadido como atributo público de la clase MarioMovement.cs de modo que todos los personajes pueden acceder al finalDir de los demás. Su motivo de ser es que en un pequeño porcentaje de las situaciones, dos personajes pueden intentar rodearse eligiendo la misma dirección para hacerlo tal y como se ejemplifica en la **Figura 31.**

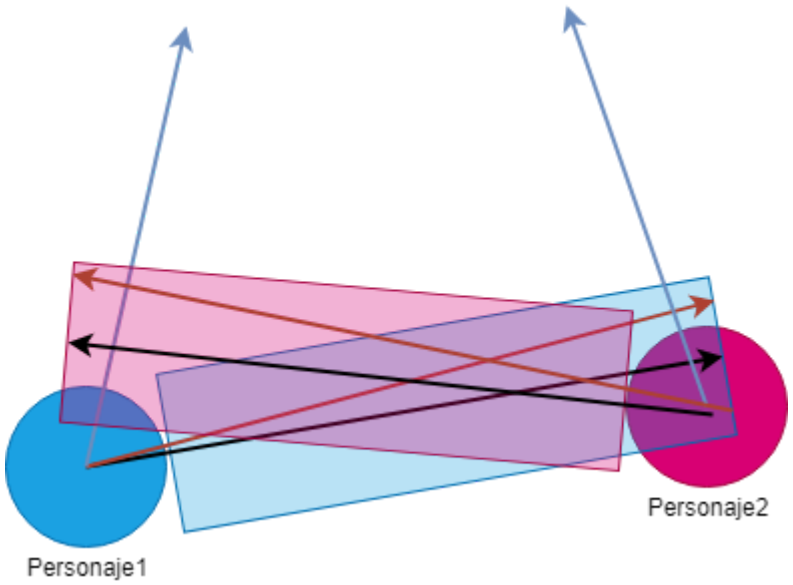


Figura 31. Ejemplo gráfico de dos personajes eligiendo una tangente de mismo signo para rodearse. Las flechas azules representan el vector de giro calculado por obstacle avoidance, las negras el vector forward y las naranjas la interpolación de ambos (aplicando el parámetro directionW).

Por lo tanto, dado que esta versión del código es secuencial, siempre que consultemos el valor de finalDir de el obstáculo que queremos rodear, este será o nulo (0,0,0) o ya habrá sido calculado por lo que en caso de tener el mismo signo de vector de giro que el nuestro, negaremos el nuestro. De esta manera, siempre que dos personajes con direcciones opuestas se vayan a rodear lo harán por el lado opuesto.

Finalmente la sección azul se corresponde con las instrucciones encargadas de mover el personaje normalmente, es decir, como antes de la implementación de la función move. Sin embargo, en esta ocasión se añaden un par de instrucciones más aparte de mover el personaje y rotar su posición. Estas son acelerar al personaje (sin superar la velocidad máxima determinada por el usuario), dado que tenemos vía libre por delante, y setear el valor de finalDir a (0,0,0) para indicar al resto de personajes que vayan a rodearnos que no tenemos una dirección de giro calculada.

11.6 Implementaciones descartadas: Recálculo del camino mínimo

Finalizadas todas las explicaciones de las implementaciones que se han realizado para el proyecto, se pasa a explicar una función que en un primer momento se pensó que sumaría al proyecto, pero que finalmente no ha sido así. Este es el caso del recálculo del camino mínimo, el cual consiste en, en vez de intentar atravesar una arista muy concurrida, recalcular el camino al destino evitando atravesar esta. El ejemplo gráfico de esto, se puede observar en la **Figura 3** de esta memoria.

Para este proyecto, el objetivo es mover miles de personajes a la vez de forma natural y eficiente. Por lo tanto, dada la gran cantidad de personajes que se van a intentar mover, es fácil que estos se queden atascados intentando pasar de un triángulo A a uno B por la falta de longitud de la arista. Buscando un símil, se trata de una situación muy parecida a la que se da en las entradas al metro en hora punta. Al igual que lo que se quiere implementar con el recálculo del camino mínimo, mucha gente al ver tal aglomeración de gente en una entrada del metro busca otra entrada por la que acceder. Así pues, se explicará como se ha implementado esta idea y los motivos por los cuales no ha sido incluida en la versión final presentada del proyecto. Intentando contextualizar temporalmente esta función, esta se realizó después de la implementación de **Static WayPoints** y se desarrolló a la par que **Dynamic WayPoints**.

Para implementar el recálculo del camino mínimo, se debe de poder detectar y representar cuán de ocupada esta una arista (para saber si recalcular el camino o no), además de tener en cuenta esta ocupación en el recálculo del camino para no obtener el mismo que ya teníamos. Para ello se añadió el atributo Occupation a la clase Neighbor que consiste en un entero que se irá incrementando en uno por cada personaje que pida un WayPoint a la arista de la clase. De la misma manera, cuando un personaje llegue a al WayPoint de una arista A y pase a calcular el siguiente de una B, entonces la ocupación de la arista A se decrementará en una unidad. Sabiendo ahora de forma aproximada la ocupación de una arista (ya que el atributo ocupación no describe exactamente cuantos personajes están pasando sobre la arista, sinó cuántos pasarán), en caso de recálculo del camino, hay que tener en cuenta esta ocupación. Para ello existe la clase GraphData.cs, mencionada y utilizada tanto en **[Static WayPoints]** y como en **[Dynamic WayPoints]**, donde hay guardado un grafo de todo el mapa y es al cual todos los personajes acceden para actualizar la ocupación de las aristas que van atravesando.

Presentada la idea detrás de la implementación de este método solo queda ver que clases de las presentadas (MarioMovement.cs, AStar.cs y clases para representar y generar el grafo) han sido involucradas en su desarrollo. La primera es MarioMovement.cs ya que es en la que se ha implementado el movimiento y donde se asignan los puntos de paso, por lo tanto, se necesita añadir la condición de que en caso de que la arista a la que le pidamos el punto de paso esté muy ocupada, calculemos un nuevo camino. Este recalculo tal y como se verá en el pseudocódigo, solo se realizará cuando vayamos a asignar un punto de paso de un nuevo triángulo (nos dirijamos al siguiente triángulo del camino) ya que si se realizara a cada frame o cada pocos frames, la cantidad de cálculo por frame sería demasiado alta. Para decidir si una arista está muy ocupada se utiliza el siguiente criterio:

$$Ocupación >= 2(\frac{LongitudArista}{2radioPersonaje})$$

Esto significa que se supondrá ocupada una arista si la ocupación (personajes que se dirigen a esta arista) es superior o igual al doble de la cantidad de personajes que se podrían poner en fila sobre la arista uno al lado de otro. Se eligió el doble y no justo esta cantidad debido a que como se ha

aclarado antes, la ocupación se actualiza cada vez que un personaje se le asigna un nuevo punto de paso al que dirigirse. Por lo tanto, si una ocupación tiene un valor de diez, significa que hay diez personajes dirigiéndose a esa arista, pero eso no significa que estén sobre esta necesariamente.

Este método se llegó a implementar de la misma manera tanto como para **[Static WayPoints]** como para **[Dynamic WayPoints]**, así pues será la versión utilizada en **[Dynamic WayPoints]** para exponer el pseudocódigo. La función que se modifica de la clase MarioMovement.cs para implementar este recálculo del camino mínimo es AsignNewPoint. Cabe destacar que solo se mostrará en el pseudocódigo relacionado con el método del recálculo o incluyendo las partes necesarias para entender el método.

```
Pre: Cierto.
Post: Asigna un NextPoint correspondiente a la arista entre al
triángulo actual al que se acaba de llegar y el siguiente.
AsignNewPoint()
si no estamos en el último triángulo entonces
      si GraphData.isTooBusy(trianglePath[indexPath],
      trianglePath[indexPath+1]) entonces
            por cada Neighbor neigh de Graph[currentTriangle]
                  anyFree = false
                  si     !GraphData.isTooBusy(trianglePath[indexPath],
                  neigh.id) entonces
                         anyFree = true
                  si anyFree entonces
                        si estamos en el primer triángulo entonces
                              GraphData.liberateOccupation(trianglePath
                              [indexPath - 1], trianglePath[indexPath])
                        FirsTriangle = true
                        trianglePath=script.recalculatePath(
                        currentTriangle,nextTriangle,destination)
                        indexPath = 0
                        currentTriangle = trianglePathID[indexPath]

      sino entonces
            FirstTriangle = false
            si no estamos en el primer triángulo entonces
                  GraphData.liberateOccupation(trianglePath[indexPath-
                  1],trianglePathID[indexPath])
            NextPoint = GraphData.getProyectedWayPoint(
            trianglePath[indexPath], trianglePath[indexPath+1])
            ++indexPath
sinó
      GraphData.liberateOccupation(trianglePath[indexPath    -    1],
      trianglePathID[indexPath])
      NextPoint = destinacionFinal.position
```

Como se puede ver, para incluir el método el Dynamic WayPoints simplemente antes de asignar un nuevo punto de paso, preguntamos si la arista a entre el triángulo actual y el siguiente está muy ocupada (siguiendo el criterio anteriormente nombrado). En caso de serlo, se busca por los otros vecinos del triángulo actual (como máximo un triángulo puede tener dos vecinos más) si también tienen las aristas ocupadas o no. Si si lo están, no hay instrucciones determinadas por lo que el personaje se quedara quieto sin hacer nada hasta que una arista quede libre (tal y como se quiere). Por el contrario, si alguna de las aristas, está libre, se recalcula el camino partiendo del triángulo actual hacia el destino original. Finalmente, si la arista no está ocupada, se calcula el punto de paso dinámico normalmente. Como aclaración, aunque se evidente, la función liberateOccupation de GraphData, disminuye la ocupación de la arista entre dos triángulos en los dos sentidos. Esto quiere decir que si tenemos un triángulo A con un vecino/neighbor B se decrementará la ocupación de la arista en AB y de la misma manera, hay que hacer el decremento en el otro sentido, el triángulo B que tendrá como vecino/neighbor A se le decrementa la ocupación en una unidad también. El método análogo que sería incrementar la ocupación de una arista, no existe ya que se ha implementado explícitamente en el método getProyectedWayPoint de GraphData.

La clase AStar.cs también se ve involucrada ya que es la encargada de recalcular el camino y en cierta manera tiene que tener en cuenta esta ocupación para calcular un camino diferente. Para ello, todo el código relacionado con el algoritmo A* se ha quedado intacto y solo se ha modificado la función DistanceBetweenNodes para que tenga en cuenta la ocupación de la arista contigua entre dos triángulos A y B. Para que este recálculo de camino tenga sentido, hace falta que la ocupación de las aristas esté debidamente actualizada por todos los personajes que se mueven por el mapa con nuestro método. Por ello, el recálculo se realiza a través de la clase GraphData.cs, la cual tal y como se ha mencionado anteriormente, contiene un grafo actualizado del mapa por todos los personajes (ya que todos acceden al mismo). De esta manera, la ocupación de una arista del grafo representado en la clase GraphData se corresponde con la ocupación teniendo en cuenta todos los personajes del mapa. Para ver cómo se penaliza este alto tráfico por una arista en la función DistanceBetweenNodes se presenta su siguiente pseudocódigo:

```
Pre: Recibe el identificador del nodo actual a tratar y su nodo
vecino en forma de Neighbor
Post: Retorna la distancia entre los baricentros del triángulo
actual y su vecino. Este valor se multiplica por la occupation de la
arista entre current y neigh en caso de Ocupaction >= 2(LongitudArista / 2radioPersonaje)
distanceBetweenNodes(current, neigh)
int penalización = 1
si neigh.isBusy()
     entonces
          penalización = neigh.ocupation
retorna    penalización    *    distancia(Graph[current].barycenter,
neigh.node.barycenter)
```

Observando el pseudocódigo se puede ver que en caso de sobrepasar el límite de lo que se ha decidido como una arista ocupada, entonces la distancia real entre dos triángulos se multiplica por la ocupación. La multiplicación sólo se realiza en este caso ya que el mapa entero esta siendo actualizado por todos los personajes y típicamente, aunque no se supere la ocupación máxima, las

aristas de los triángulos que vayamos a transitar tendrán una ocupación mayor a 0. Por lo tanto, realizar la multiplicación por la ocupación sin superar el límite, podría distorsionar el nuevo camino calculado más de la cuenta (evitar aristas que no queremos que sean evitadas). Finalmente, se decidió optar por la multiplicación por la ocupación ya que siempre va a ser un valor mayor a dos y es suficiente para hacer que el A* elija un camino alternativo sin pasar por la arista penalizada.

Para acabar, se van a presentar los motivos por los que se ha descartado este método. Estos motivos, se podrían resumir a uno prácticamente, y es que aunque el método realice justo los que se pretendía, el efecto visual no es el deseado. Para ejemplificar esto se presenta la **Figura 32**.

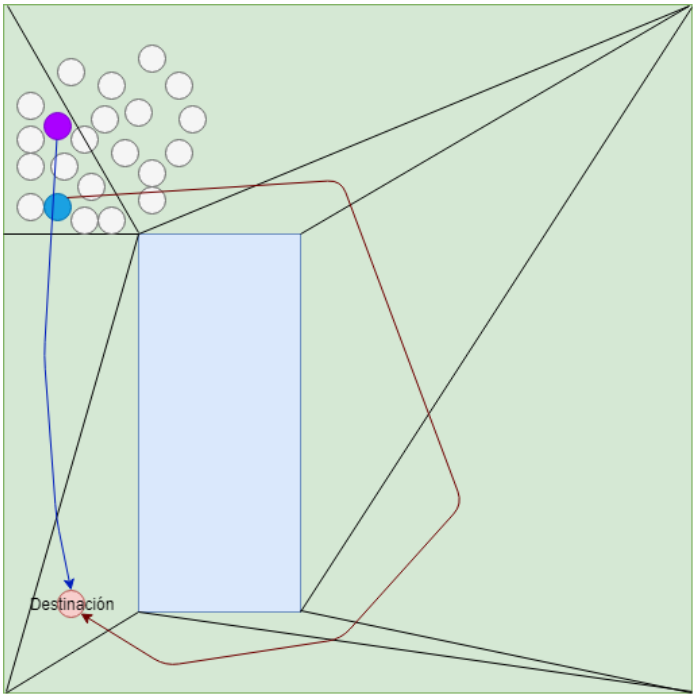


Figura 32. Problema visual con recalcular camino mínimo.

En la **Figura 32**, se puede observar un escenario hipotético donde se puede ver un conjunto de personajes agrupados en dos triángulos. Tal y como funciona nuestro A* explicado en **[Movimiento a partir del A*]**, todos los personajes seguirán el mismo camino que el personaje lila. A causa de esto, todos los personajes se aglomeran formando un cuello de botella en la arista que atraviesa el camino azul. El problema que aparece para nuestro método es que este recalculara el camino mínimo para los personajes en el mismo triángulo que el personaje lila y no para los personajes más alejados como los del triángulo superior. Por lo tanto, por lo general, solo se recalculara el camino cuando lleguemos al triángulo que genera cuello de botella. Esto se traduce en personajes como el azul, recalculando el camino e intentando tomar el de la flecha roja mientras la mayor cantidad de personajes intentarán tomar el camino azul (porque aún no han llegado al triángulo del conflicto o porque son de los que no han recalculado camino) y entrarán en conflicto. El efecto final, es que la gran mayoría de personajes que toma el camino azul acaba pasando primero y cuando se despeja un poco el camino en último lugar comienzan a recorrer el camino los personajes que toman el personaje rojo.

Así pues, se decidió descartar el método ya que dedicar más tiempo a encontrar una solución requería restarle al de implementar la versión paralela y esta era la parte más importante del proyecto. Además, recalcular el camino significa aplicar el A* entero el cual no es un algoritmo para nada gratuito (debido a su coste computacional) y menos para realizarlo cada x frames.

12. Paralelización con ECS

Como última fase del proyecto solo queda la paralelización del mismo. En este apartado se describirán cómo se ha abordado la problemática de trasladar un código secuencial a uno de paralelo utilizando las tecnologías ECS, Jobs System y burst compiler. Para ello esta fase de trabajo se dividió en dos partes, una primera de investigación dedicada a entender mejor cómo funcionan dichas tecnologías y cómo aplicarlas a nuestro proyecto y una segunda de implementación puramente, con sus consecuentes tareas de análisis, programación y testeo asociadas. Es importante destacar que se está paralelizando el cálculo de la IA (movimiento inteligente de los personajes), por lo tanto siempre que hablemos de paralelización esta es sobre la CPU y no sobre la GPU. Finalmente, también se describirán varios problemas encontrados a la hora de paralelizar el código, así como limitaciones que presentan estas tecnologías, provocando modificaciones de *performance* respecto a la versión secuencial.

12.1 Investigación

Antes de entrar en la implementación se dedicó varios sprints del proyecto a entender las tecnologías explicadas en **[Introducción]** (ECS, Jobs System y burst compiler). Para ello se realizaron varias búsquedas por la red, sobretodo en las páginas oficiales de unity y sus foros.

Tras una larga búsqueda sobre dichas tecnologías, se pudo tener una idea más clara de en qué consisten y cuál es el alcance de estas. Sobre ECS se cimentó el conocimiento adquirido al inicio del proyecto, ya que en esa etapa también se investigó sobre este, y se vió que consiste en (de forma muy resumida) manipular los distintos entities (personajes/objetos en ECS) con unos determinados components (datos **básicos**) a través de un System (script). Existía la opción de implementar ECS de forma Híbrida (que no todos los datos fueran components) pero debido a que se quería conseguir un rendimiento máximo se fue directamente a una versión puramente de ECS y ha sido la que se ha implementado finalmente. Así pues, con ECS se consigue estructurar los datos de forma mucho más eficiente ya que un System que se utilice para mover los personajes solo pedirà los datos relacionados con dicho sistema. Esto se traduce en que se realizarán menos cargas de datos innecesarios, los datos estarán menos fragmentados dado que están encapsulados en componentes más simples (**Figura 33**) y por lo tanto habrá menos fallos de cache.

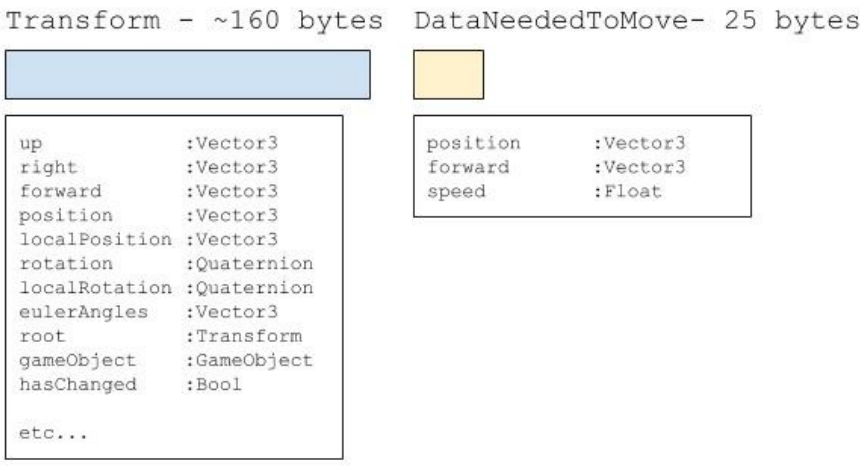


Figura 33. Memoria necesaria para mover a un personaje método clásico vs ECS (imagen extraída de **[2]**).

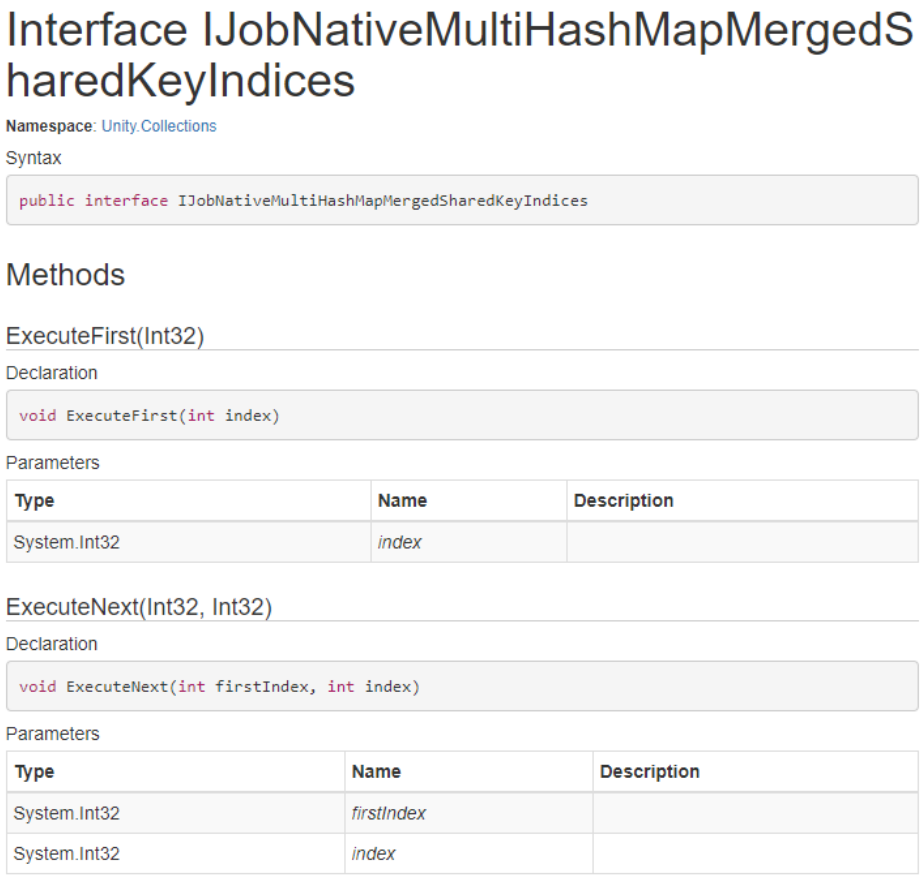
Sobre Jobs System se consiguió ver cómo llevaba a cabo esta paralelización. Resumiendo, ya que esto se verá más extensamente en el apartado de **[Implementación paralela]**, Jobs System te permite dividir el trabajo en Jobs (unidades atómicas de trabajo) que el main Thread de la CPU repartirá al resto de Threads para realizarlos en paralelo. A pesar de varias limitaciones que tiene, es bastante sencillo de utilizar y dispone de varias funciones para dividir el trabajo de un bucle sobre una lista en varios Jobs. Por último, sobre Burst Compiler no se hicieron grandes descubrimientos, más allá de los que ya se conocían, debido a que este simplemente es una opción para que el compilador optimice el código en paralelo.

A raíz de esta investigación se vieron varios problemas que se comentarán en la siguiente sección **[Problemas encontrados]**, pero hay uno que sí que se comentará en esta. Este hace referencia al problema de la gran falta de información sobre ECS en la red. Dejando de lado los simples proyectos en ECS que te presenta Unity y algunos otros muy parecidos creados por terceros que hay de ejemplo OpenSource, la mayoría de la documentación sobre ECS y Jobs System es muy pobre. Si que es cierto que existe su respectiva documentación **[23]** en la cual los elementos principales de ECS y algunos de Jobs System quedan brevemente explicados y en la mayoría de los casos es suficiente para poder utilizarlos. El problema es que cuando se quieren realizar cosas más complejas de las mostradas en sus ejemplos básicos, la documentación existente es insuficiente. Esto ocurre tanto para ECS como Jobs System y para demostrarlo se presenta un ejemplo de esto para cada uno.

Para ECS por ejemplo, sin comentar problemas que se explicarán más adelante, no hay lugar en internet donde se explique como realizar algo tan necesario y básico como el escalado de la malla de triángulos de un entity (los personajes en esta tecnología) de forma oficial. Todo lo que existe para solucionar este problema son códigos de terceros que en la mayoría de casos no es compatible con la versión de ECS que se utiliza debido a las múltiples versiones que existen y la nula compatibilidad entre ellas.

Para el caso de Jobs System si queremos consultar sobre elementos básicos de Jobs System como por ejemplo en qué consisten los JobHandle su documentación es muy pobre pero existe y es suficiente para comprender su funcionamiento **[24]**. Por otro lado, cuando se quiere paralelizar el trato de un

bucle sobre una lista se utiliza funciones como IJobParallelFor (divide las iteraciones de un bucle en chunks de 64), sin embargo cuando en vez de una lista queremos paralelizar el trato de un MultiHash hay que utilizar la función IJobNativeMultiHashMapMergedSharedKeyIndices. Dejando de lado lo críptico del nombre de la función, si nos dirigimos a la página oficial en busca de la documentación pertinente para saber cómo funciona dicha función nos encontramos la siguiente información **[25]**. En el enlace anterior se puede ver la documentación mostrada en la **Figura 34**.



Interface IJobNativeMultiHashMapMergedSharedKeyIndices

Namespace: Unity.Collections

Syntax

```
public interface IJobNativeMultiHashMapMergedSharedKeyIndices
```

Methods

ExecuteFirst(Int32)

Declaration

```
void ExecuteFirst(int index)
```

Parameters

Type	Name	Description
System.Int32	*index*	

ExecuteNext(Int32, Int32)

Declaration

```
void ExecuteNext(int firstIndex, int index)
```

Parameters

Type	Name	Description
System.Int32	*firstIndex*	
System.Int32	*index*	

Figura 34. Documentación sobre el método IJobNativeMultiHashMapMergedSharedKeyIndices.

Como se puede observar en la **Figura 34**, no hay ninguna descripción para ninguno de los métodos, ni siquiera una descripción global que explique como funciona el método. La única solución pues, es buscar en foros de Unity e intentar forjar una idea aproximada sobre el comportamiento de dicha función. Aún así, esta información resulta insuficiente ya que muchas veces ni entre los foreros se ponen de acuerdo sobre su funcionamiento.

Por este motivo, es decir, por la falta de información, y por el problema presentado en **[Paralelizar con local movement]** se decidió optar por desarrollar la versión paralela del proyecto partiendo de otro OpenSource de simulación de fluidos. Esta decisión se tomó conjuntamente con los directores de proyecto por dos motivos principales. El primero es poder aprender disponer de un ejemplo de implementación de ECS y Jobs System y no tener que depender tanto de la información disponible en internet, ya que se encontró que era realmente escasa. El segundo está relacionado con tomar como base un proyecto de simulación de fluidos. Dado que al trabajar en paralelo cada personaje es

tratado de forma independiente, en un bucle paralelo, no se sabía cómo acceder al resto de personajes desde este. Por lo tanto, teniendo en cuenta que cualquier proyecto de simulación de partículas cada partícula necesita saber el estado sus partículas cercanas para moverse, este nos sería de ejemplo clave para desarrollar el nuestro.

Así pues, investigando en busca de un proyecto que utilizara ECS/JobsSystem/BurstCompiler para la simulación de fluidos se encontró el siguiente artículo **How to implement a Fluid Simulation on the CPU with Unity (ECS/Job System) [26]**. En este artículo se adjunta el proyecto del que habla, el cual está desarrollado en Unity por Léonardo Montes y simula la interacción entre esferas a partir de varias características que se pueden definir del material. Dentro del proyecto se puede encontrar la versión *single-core* (sin paralelizar) de la simulación y la *multicore* (paralela) que utiliza las tres tecnologías de las que se quería hacer uso en un principio. Además de la gran ayuda que era disponer de un proyecto que contará justo con las características que se buscaban, lo bien explicado que estaba el artículo ayudo a entender cómo orientar la programación en ECS.

Tomando como base la simulación paralela del proyecto de Léonardo, se adaptaron y crearon las funciones necesarias para conseguir el mismo *performance* que para la versión sin paralelizar. Aún así, antes de explicar la implementación del mismo, se detallarán los problemas más relevantes que han alargado la fase de implementación de ECS (dentro de lo que se predijo). Se explican antes los problemas que la implementación, ya que así será más fácil entender después porque ciertas funcionalidades se han desarrollado como se ha hecho.

12.2 Problemas encontrados

En este subapartado se detallarán los diferentes problemas encontrados al llevar un código *single core* a uno que utilice *multithreading*. Para ello se abarcaran los problemas en cuatro bloques distintos (**[Compatibilidad entre versiones]**, **[Limitaciones de la paralelización en Jobs System]**, **[Limitaciones de ECS]** y **[Dificultad para depurar código en paralelo]**). Cabe destacar, que a pesar de que se esperaba cierta dificultad para testear el código paralelo y que Jobs System estuviera limitado, jamás se pensó que la compatibilidad entre las versiones de las tecnologías usadas y ECS fuera tan problemática.

12.2.1 Compatibilidad entre versiones

Este es el ejemplo de que siempre hay imprevistos en todos los proyectos, y es que nunca se pensó que la compatibilidad entre versiones llegara a ser un problema. Debido a que la versión paralela tomaba como base el proyecto simulación de partículas, era imprescindible primero de todo hacer funcionar este y luego importar los elementos que fueran de interés a otro proyecto o trabajar sobre este. Es en este punto, donde aparece el primer problema causado por la compatibilidad ya que hacer funcionar los diferentes scripts i elementos del proyecto original en otro, se convirtió en algo verdaderamente tedioso.

Al descargar el proyecto original y arrancarlo, este siempre funcionó normalmente. El problema es cuando se intentaba llevar los prefabs de los entities y los diferentes scripts necesarios a otro proyecto, que llamaremos proyecto B, estos no funcionaban, dando errores totalmente crípticos. Tras muchas búsquedas, se descubrió que el problema reside en la compatibilidad entre versiones de las

tecnologías. Así pues, en el proyecto B se habían instalado las últimas versiones de las tecnologías ECS/JobsSystem/BurstCompiler tal y como sugiere Unity, pero al no haber ningún tipo de retrocompatibilidad, ni warning sobre versiones, Unity lanza errores totalmente indescifrables. Por lo tanto, tras probar muchas combinaciones de versiones (ya que cada tecnología tiene la suya) se decidió usar exactamente las mismas que el proyecto original, a pesar de que estas no eran las más actuales.

Por otro lado, también se observó que entre versiones, de ECS por ejemplo, los nombres de las funciones podían cambiar totalmente, los tipos de datos que se pueden utilizar como componente de un entity también pueden cambiar y en general hay cierta inconsistencia entre algunas versiones de ECS. Como conclusión, se nota que se trata de una tecnología sobre la que Unity está trabajando a día de hoy y todavía les queda un amplio margen de mejora, quedando lejos una versión definitiva de estas novedosas tecnologías.

12.2.2 Limitaciones de la paralelización en Jobs System

Este apartado, junto con el siguiente **[Limitaciones de ECS]**, son los que más han lastrado el proyecto ya que han limitado fuertemente que es lo que se puede y no hacer usando estas tecnologías.

Poniendo el foco sobre Jobs System, este sufre del mismo problema que lo hace Unity en general. Al principio del proyecto se hablaba de que Unity, dentro del mundo de los motores de desarrollo de videojuegos, es uno de los más sencillos de utilizar. Esto es algo bueno, ya que se hace fácilmente usable para el usuario más inexperto pero presenta una gran contra. Para el usuario experto que desea modificar estas funciones predeterminadas, en pro de hacer más específico su funcionamiento, se le cierran todas las puertas. El motivo es que la mayoría de las funciones que ofrece Unity son como una caja negra a la que no se puede acceder para hacer modificaciones y esto es justamente lo que le sucede a la paralelización de Jobs System.

Si se pretende paralelizar por ejemplo, la suma de dos vectores de enteros entonces Jobs System nos hará la vida más fácil. Por el contrario, si lo que se pretende es paralelizar la creación de los *pathings* de todos los entities, entonces Jobs System se queda bastante corto. Esto no es debido a la complejidad de realizar tal acción, ya que simplemente se trataría de paralelizar un bucle que aplicara el A* para cada uno de los entities, sinó en lo que se permite hacer dentro de un bucle paralelo. Dado que las funciones de bucles paralelos están creadas para tener un alto rendimiento, los únicos datos que pueden tratar estas funciones, son los tipos básicos (enteros, floats, booleanos, etc.) y unas listas especiales que solo pueden contener tipos básicos dentro. Por lo tanto, aplicar algoritmos como el A* que requieren de estructuras complejas es totalmente inviable.

Así pues, la creación de todos los pathings al inicio se optó por hacerla de forma secuencial pese al elevado coste computacional que esto conlleva.

Finalmente, también hace falta destacar que las limitaciones no se quedan solo en que datos tipo de datos te deja utilizar, sino en cómo te deja utilizarlos. Tras la experiencia personal de haber trabajado con la API OpenMP para C/C++ y Cuda para los mismos lenguajes, las posibilidades con Jobs System son muy pocas. Mientras que OpenMP y Cuda ofrecen la posibilidad de acceder a variables globales des de secciones de código paralelo por todos los threads, en Jobs System esto ni existe. En Jobs System cuando creas un bucle paralelo, solo puedes modificar los datos que pasas como parametro a

esta función. Además, tampoco existe la posibilidad de crear instrucciones atómicas (solo un thread puede ejecutar una instrucción atómica a la vez), cuando en las APIs anteriores si. Por último, como ejemplo final de sus limitaciones se presenta la siguiente situación:

- Se está ejecutando un bucle en paralelo de 0 a N elementos en chunks de 64. Esto quiere decir que cada thread ejecutara un chunk con N/64 elementos.

- Estamos en el Thread i con i entre 0 y 64. Esto quiere decir que estamos tratando elementos entre i*N/64 y (i+1)*N/64.

- Este bucle está tratando una lista L de longitud N y contiene todas las posiciones de los personajes. Esta lista se ha declarado como una lista de escritura y lectura.

- Jobs System nos da un indice *index* para saber qué iteración se está tratando. Esto quiere decir que en la función trataremos el elemento *index* de la lista L.

- Con la lista L obtenemos la posición de *index*. Si se lee otra posición la lista L como la *index* + 64 y luego se escribe en la *index* Jobs System lanza un error.

Esto ocurre porque Jobs System no sabe si estas escribiendo en la posición index + 64, lo cual podría corromper en el valor del thread que trate el elemento index + 64. Por lo tanto, para evitar cualquier error en el que podría incurrir el programador sin darse cuenta, simplemente te quitan esa opción. Como conclusión, es una buena herramienta si se quieren hacer cosas básicas, pero en cuanto comienzas a ir un poco más allá, se queda muy corta en comparación con sus otras opciones.

12.2.3 Limitaciones de ECS

Si Jobs System solo nos permite trabajar con la paralelización de bucles muy simples con datos aún más simples (tipos básicos y listas de tipos básicos), para ECS ocurre algo similar con los tipos de dato que permite alojar a cada entity.

Como bien se ha ido recordando a lo largo del proyecto, ECS está formado por Entities, Components y Systems. Para entender la limitación que se va a presentar, se simplificará este modelo a entender que los Entities son cada uno de los personajes, los Componentes los datos que estos tienen y los System los scripts que mueven los personajes.

En el proyecto desarrollado, solo se ha implementado un system para mover los personajes, por lo tanto, todos los componentes creados para los entities son usados en este system. En el caso de tener otros system, entonces los entities tendrían más componentes que solo llamarian esos otros system. Esto tal y como se ha explicado en **[Investigación]**, mejora enormemente como se organizan los datos, disminuyendo los fallos de cache y facilitando el paralelismo. Sin embargo, y aquí es donde aparece la limitación, los datos que añadimos como componente a un entity no pueden ser de cualquier tipo.

Dejando de lado variantes, los datos que como componente se pueden adherir a un entity pueden ser **IdataComponent** (datos característicos y propios de un entity: posición, velocidad, fuerza, etc.) o **ISharedDataComponent** (datos compartidos entre varios entities: masa, densidad, viscosidad, etc.).

Como se puede ver, todos los ejemplos mostrados se pueden representar con enteros, floats o float3 (similar a vectores), el problema aparece cuando por ejemplo se quiere guardar el camino calculado con A* en un entity. Sin importar que queremos guardar de esa lista que contiene el camino mínimo, ECS directamente no permite tener listas de tamaño dinámico dentro de los componentes **IdataComponent** o **ISharedDataComponent**. Como se puede entender, esto limita muchísimo el desarrollo de este proyecto y cualquiera que se quiera implementar utilizando ECS, ya que los entities como datos solo pueden tener tipos básicos y clases de objetos que solo contengan tipos básicos. Además, algunos tipos como los booleanos, los vectores o incluso Strings tampoco se pueden utilizar en ECS, teniendo que recurrir a varios trucos a como utilizar enteros como booleanos y float3 como vectores.

De cara a superar esta limitación, se decidió crear una lista que contuviera el camino de cada entity, uno tras otro y guardarla dentro del script system dedicado a mover los personajes. De esta manera, cada entity solo tiene que guardar los índices iniciales y finales para indicar en qué segmento de esa lista se encuentra su camino. Caber recordar, que se decidió guardar los caminos para cada entity en esta lista gigante y no en una matriz (siendo más fácil de manejar), porque los métodos paralelos de Jobs System no aceptan matrices como argumentos de entrada.

Aún y estas limitaciones, se sabe que el personal de Unity está dedicando muchos recursos a mejorar ECS todo lo posible. Un ejemplo de esto es que varias versiones de ECS anteriores a la utilizada en el proyecto se intentó implementar un tipo de listas llamadas fixedArray, pero lo acabaron retirando por los fallos que daba. Otra muestra de los recursos que se están destinando a esta tecnología son los grandes y constantes cambios entre versiones, que pese a ser molestos para el usuario programador, a la larga derivará una una versión definitiva y más sólida de ECS.

12.2.4 Dificultad para depurar código en paralelo

Depurar código para encontrar bugs o intentar solucionarlos, siempre es una tarea bastante difícil y más cuando se está tratando un proyecto grande, ya que interacciones no deseadas pueden aparecer fruto de errores en partes de código insospechado. En el caso de la programación paralela, este factor se ve incrementado, debido a que la secuencialidad que asegura un código con programación clásica se ve desvanecida por el paralelismo. Esto produce que traquear un error, cuando se produce en una API que explota el paralelismo al máximo como Cuda o ompParallel, sea un arduo trabajo.

En jobs System, al contrario que en Cuda por ejemplo, los errores derivados del paralelismo como por ejemplo accesos no deseados desde varios threads a un mismo objeto, accesos a direcciones de memoria inválidos, etc. desaparecen totalmente. Esto es debido a que la tecnología está tan limitada, que el usuario es imposible que pueda cometer uno de los errores mencionados anteriormente porque directamente no se le deja realizar una acción que desencadene en eso. Sin embargo, al facilitar la programación en paralelo limitando las acciones posibles, los métodos de depuración tradicional se ven capados.

Así pues, acciones como escribir por consola strings o enteros, o utilizar los métodos de depuración de Unity para dibujar líneas o rayos, no se pueden utilizar dentro de los bucles paralelos de Jobs System. Esto se resume básicamente en que lo que ocurre dentro de cada sección de código paralelo, es como una caja negra de la cual resulta muy difícil extraer información.

Para intentar sortear estas dificultades, se han utilizado varias estrategias para extraer la información de interés de los bloques paralelos. Una de ellas consiste en guardar la información relevante dentro de los objetos de salida de la función y mostrarlos ya por pantalla utilizando el main thread (el que ejecuta el system y no trabaja en paralelo).

Finalmente, cabe destacar que si se ha encontrado un gran fallo de paralelismo imposible de solucionar. De forma casi aleatoria, a veces el programa crashea mostrando el mensaje "GetThreadContext failed" y sin ninguna explicación más Unity aborta. Investigando por la red se encontró que a varios usuarios de ECS les pasaba lo mismo, también de forma aleatoria, y se especula que seguramente se deba a un bug interno de ECS con windows ya que en UNIX no se han dado casos de este fallo.

12.3 Implementación paralela

Una vez presentadas las diferentes limitaciones y dificultades para la implementación que presentan las tecnologías ECS, Jobs System y Burst Compiler, se pasa a explicar como se ha llevado a cabo la implementación paralela. Dado que esta versión toma como base el proyecto de Leonardo Montes, primero se explicará cómo funciona este muy a alto nivel y seguidamente se presentarán los añadidos para conseguir la *performance* de la versión single core implementada.

12.3.1 Funcionamiento del proyecto base

El proyecto de Leonardo Montes, simula la interacción de varias esferas cayendo y chocando unas con otras a partir de sus propiedades físicas, además del cálculo de las distintas fuerzas que interaccionan. Dado que es un proyecto basado en ECS, este se explicará definiendo cómo se han adaptado los tres elementos característicos (entities, components y system) a su propuesta. Primeramente, se explicará cuales son los entities en este proyecto y que componentes presentan estos, para finalmente explicar el funcionamiento del system a alto nivel.

No es difícil deducir que para este proyecto los entities serán cada una de las esferas que se pretender mover. Cada una de ellas contará con varios componentes, los más relevantes son dos componentes de datos individuales o **IdataComponent** (el valor en cada entity no tiene porque ser el mismo) por ejemplo position y velocity; además de componentes de datos compartidos o **ISharedDataComponent** (cada entity que contiene este componente comparten los mismos valores) por ejemplo en este caso properties (masa, densidad, etc).

Finalmente, un system es el que mueve todas las esferas a partir de sus componentes. El system, dado que está implementado utilizando Jobs System, se ejecuta de forma que un thread (main Thread) es el que corre el script del system entero y este reparte los distintos bloques de trabajo entre el resto de thread para que lo vayan ejecutando en paralelo. Para entender mejor esta idea y ver como se ha llevado a cabo en el proyecto de Leonardo Montes, se explicará de forma cronológica en los siguientes puntos la ejecución del método OnUpdate (método que se ejecuta a cada frame) del system:

- Se cargan los datos individuales (position y velocity) de todos los entities y lo compartidos.

- Se copian de forma paralela los datos de los entities a estructuras temporales (native array) que sean soportadas por los métodos paralelos. Esta copia paralela se divide en chunks de 64 elementos ya que es lo recomendado por el propio Jobs System. Cuando se llama a ejecutar estos métodos se crea una dependencia llamada JobsHande que indica cuando ha acabado un método paralelo por completo y cuando no. Toda función paralela recibe una dependencia, en este caso como no se ha paralelizado antes se recibe la dependencia *inputs* del método OnUpdate.

- A partir de los datos característicos de los entities y sus datos individuales (position y velocity) en estructuras temporales, se calculan las diferentes fuerzas que interactúan para determinar las nuevas posiciones y velocidades de cada esfera. Esto, también se realiza de forma paralela en chunks de 64 elementos. Como método paralelo depende de que hayan finalizado las copias del punto anterior y este mismo crea nuevas dependencias.

- Se copian de forma paralela los datos de los entities en estructuras temporales a los componentes de los entities en sí. Este método depende de que hayan acabado de calcularse la nueva posición y la nueva velocidad del objeto. Esta copia paralela crea una dependencia se convertirá en el *inputs* del siguiente frame.

- Se repite el proceso des del primer punto. Esta vez, la copia de datos de entity a estructura temporal no se podrá realizar hasta que la posición y velocidad calculada en el frame anterior no se hayan guardado en los entities.

Como se puede observar, no se ha adentrado demasiado en cómo es un método paralelo en Jobs System ya que solo haría más compleja la explicación y se verán un par de ejemplos en **[Adaptación de la versión single core a ECS]**. Sin embargo, sí que se ha pasado por alto un elemento importante de este proyecto, y es como se obtiene el índice de esferas cercanas, motivo principal por el que se buscó un ejemplo en otro proyecto.

Cuando en el proyecto de Leonardo Montes se calculan las fuerzas aplicadas a una esfera, necesita saber qué esferas ocupan una posición circundante a la del cálculo para determinar correctamente la fuerza a aplicar en esta. Para ello el señor Montes hace uso de un HashMap y un GridHash de forma que a personajes cercanos les asignaremos el mismo Hash. La inicialización del HashMap ocurre una vez quedan cargadas todas las posiciones de los entities. En esa inicialización, para cada uno de los entities se les calcula un hash a partir de su posición y el radio del entity, agregando finalmente el hash y el índice de dicha esfera/entity en el HashMap. Así pues, cada vez que se necesite saber sobre qué esferas cercanas hay a la que se está tratando, basta con obtener el hash de la posición que queremos tratar y buscar en el HashMap.

12.3.2 Adaptación de la versión single core a ECS

Para la adaptación de la versión single core se decidió utilizar un nuevo escenario más grande ya que se pretende en un primer momento hacer pruebas con más de 1000 personajes simultáneos. Así pues, se creó un nuevo proyecto y en este fue en el que se implantó el nuevo escenario.

En primer lugar, antes de adaptar puramente la versión sin paralelizar, se comprobó que este método funcionara en el nuevo escenario para evitar bugs futuros lo antes posible, Esto fue un gran acierto dado que se encontró un bug que destrozaba la correcta creación del grafo representativo del escenario. Resumidamente, el bug afecta a la adjudicación de triángulos vecino debido a que los vértices que comparten dos triángulos adyacentes no tienen exactamente la misma posición.

Una vez funcionó la versión single core se testeo el proyecto de Leonardo Montes en este nuevo escenario antes de añadirle las funciones del proyecto sin paralelizar. Este no presentó mayor problema en cuanto a ejecutar correctamente el proyecto, dejando de lado la **[Compatibilidad entre versiones]**. Sin embargo, se encontró un problema que presentaba el proyecto de Montes, que si bien no entra dentro del *scope* del proyecto, hay que tener en cuenta. El defecto que presenta su proyecto es que si las esferas caen sobre el terreno demasiado rápido, estas lo atraviesan. Esto no afecta el performance de el proyecto que se intenta desarrollar pero si es algo a tener en cuenta, como por ejemplo, a la hora de invocar las esferas desde cierta altura.

Ahora sí, testeados los dos proyectos en el nuevo, junto al nuevo escenario, se pasó, partiendo del proyecto de Montes, a añadir las funcionalidades implementadas en el proyecto single core. Para ello y dado que se trata de una implementación ECS, primero se explicará en qué consisten los Entities y cuales son sus componentes, para finalmente explicar los añadidos al system del proyecto de Montes.

En cuanto a los entities respecto a su versión en el proyecto de Montes casi no han sufrido modificaciones, seguirán siendo el conjunto de esferas que se pretende mover. Esto se debe a que al no poder escalar ni animar la malla de un entity, no tenía mucho atractivo de cara al resultado final añadir una malla diferente a la de una esfera. Por otra parte, respecto a los componentes relativos a los entities sí que se han realizado varias modificaciones como se pueden observar en las **figuras 32** y **33**.

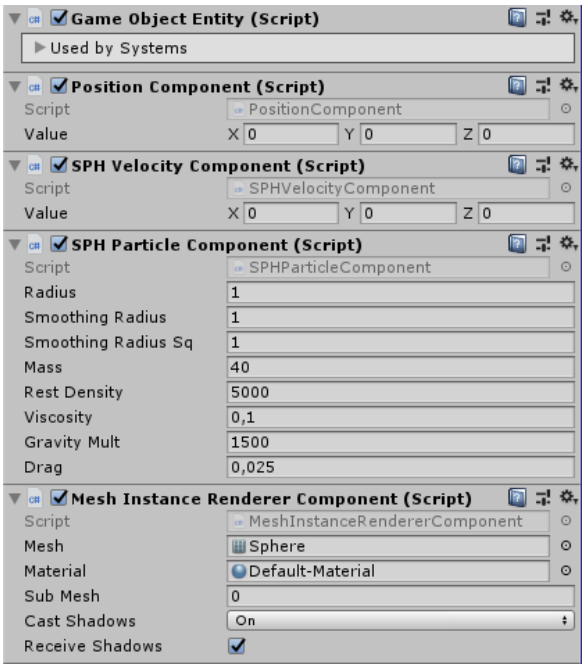


Figura 35. Componentes de los entities en el proyecto de Montes.

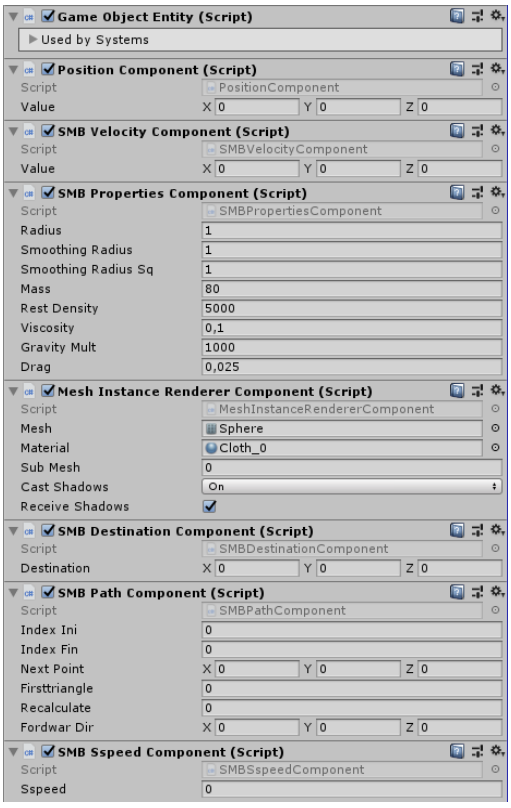


Figura 36. Componentes de los entities en la adaptación paralela propia.

En la **Figura 35** se pueden ver los componentes mencionados en **[Funcionamiento del proyecto base]** mientras que en la **Figura 36** se observan los *entity components* de la versión paralela propia desarrollada. Como se puede observar, se mantienen los tres primeros componentes debido a que se partirá del movimiento implementado por Montes. Este movimiento consiste en simular la colisión entre las esferas, el cual se mantiene para nuestra versión ya que en ECS no existe ningún tipo de elemento que haga esta simulación. En cuanto a los elementos necesarios para la adaptación del proyecto a ECS se añadieron tres componentes clave. El primero contiene la destinación de la esfera, este elemento se ha implementado utilizando un **IDataComponent** pero podría haberse hecho uso de un **IDataSharedComponent**. Se optó por el primer tipo de dato debido a ya que se quería realizar distintas pruebas con varios grupos de esferas yendo a distintos destinos. El segundo componente se trata de un **IDataComponent**, ya que los datos en cada esfera serán distintos, y contiene todos los elementos relativos a pathing que realizará la esfera. Finalmente, el último componente, también un **IDataComponent**, contiene la variable Sspeed con la que se calcula el movimiento en las nuevas funciones paralelas implementadas que se explicarán a continuación.

Una vez presentados cuales son los entities del proyecto y sus componentes que los definen se explica cómo se han adaptado las funciones del proyecto sin paralelizar. De forma general, las funciones implementadas en nuestro proyecto single core consisten en utilizar el A* para calcular los triángulos del camino para una esfera en este caso, calcular los puntos de paso de los diferentes personajes de forma dinámica y utilizar un algoritmo de movimiento local para simular el movimiento en grupo.

El primer punto no se ha podido realizar de forma paralela por los motivos expuestos en **[Limitaciones de la paralelización en Jobs System]**. Por lo tanto, el cálculo del camino mínimo para cada uno los personajes se ha realizado de forma secuencial como se explica en dicho apartado. Este cálculo se realiza una sola vez en la ejecución del programa ya que el escenario es totalmente estático (sin contar el resto de entities). También, debido a **[Limitaciones de ECS]**, los caminos resultantes no se han podido guardar como **IDataComponent** dentro de los entities por lo que se han tenido que guardar en una lista, que llamaremos Paths, aparte dentro del system. Esta lista no puede ser una matriz que corresponda índice con camino calculado por culpa de que las funciones paralelas no pueden recibir ningún tipo de matriz como parámetro. Debido a la simpleza del algoritmo no se mostrará ningún tipo de pseudocódigo ya que se trata de un bucle por todos los entities aplicando el A* para cada uno de ellos y guardando los caminos unos tras otros (lo que se guarda de estos caminos son las aristas por las que tiene que pasar la esfera). Justo después de la asignación de los pathings a la lista Paths, también se guardan los valores de los datos del SMBPathComponent que se puede observar en la **Figura 36.** Es en este componente donde se guardan qué posiciones de la lista Paths se corresponden con los caminos de un entity determinado.

Sin embargo, el calculo dinamico de los puntos de paso y el algoritmo de movimiento local sí que se han podido realizar de forma paralela. Esto es un logro ya que así sí que se podrá comparar de forma veraz la versión de Unity con la versión paralela implementada. En el apartado anterior, se ha explicado el modus operandis dentro de un frame en el proyecto de Montes. Así pues, se pasa a explicar las distintas funciones paralelas implementadas para el cálculo de los puntos de paso y el movimiento local. Cabe destacar que en cualquier muestra de pseudocódigo se omitierán partes irrelevantes como la copia de listas a native array (tipo de listas que soportan los métodos paralelos).

Entre otras cosas, el proyecto de Montes calculaba las fuerzas a aplicar y luego las guardaba en los entities. En esta adaptación, una vez calculadas las fuerzas se ejecutan tres funciones paralelas implementadas: computeNewPointJob, RecomputeNewPointJob y computePositionJob. La primera y la segunda función, computeNewPointJob y RecomputeNewPointJob respectivamente, estas se encargan del cálculo del siguiente punto de paso en función de si la esfera ya está muy próxima a su punto de paso calculado o el factor recalculate así lo pide. Se recuerda que el factor recalculate es un valor aleatorio entre 10 y 15 que determina en cuantos frames se recalculara el punto de paso sobre la misma arista objetivo pase lo que pase. Finalmente, la función computePositionJob es en la que se ha implementado el algoritmo de movimiento local en caso de encontrarse un personaje cercano o el movimiento simple en caso contrario. Con movimiento simple se hace referencia a moverse en linea recta hacia el próximo wayPoint calculado.

Estas funciones dependen la una de la otra justo en el orden en el que se han implementado. A continuación en el siguiente código se puede ver el ejemplo de cómo se invocan estas funciones paralelas

```
RecomputeNewPoint RecomputeNewPointJob = new RecomputeNewPoint
{
      particlesPosition = particlesPosition,
      waypoints = NwayPointspaths,
      #parametros de entrada...
};
      JobHandle RecomputeNewPointJobHandle = RecomputeNewPointJob.Schedule(
```

```
      particleCount, 64, computeNewPointJobHandle);
      JobHandle preparedToComputePositions = JobHandle.CombineDependencies(
      RecomputeNewPointJobHandle, particlesSspeedJobHandle);

ComputePosition computePositionJob = new ComputePosition
{
      #parametros de entrada...
};
      JobHandle comptePositionJobHandle = computePositionJob.Schedule(parti
      cleCount, 64, preparedToComputePositions);
```

En este ejemplo se puede ver como se llaman las funciones RecomputeNewPointJob y computePositionJob. Se puede apreciar que todas las funciones siguen el mismo patrón: se crea el Job asignando los parámetros de entrada y se esquematizan utilizando la función Schedule. La función Schedule permite designar cuantos elementos se van a tratar, de que tamaño serán los bloques que tratará cada thread (en este caso 64) y de que job depende el actual. Además, varias dependencias se pueden combinar utilizando el método CombineDependencies, ampliando así las posibilidades de uso.

Finalmente, solo queda ver cómo se han implementado dichas funciones explicandolas a través de un pseudocódigo a bastante alto nivel. Sin embargo la función computeNewPointJob no se analizará ya que es bastante similar a la de RecomputeNewPointJob . Empezando por esta misma función su pseudocódigo es el siguiente:

```
[BurstCompile]
private struct RecomputeNewPoint : IJobParallelFor
      NativeArray indexPaths
      [ReadOnly] NativeArray particlesPosition
      [ReadOnly] NativeArray waypoints

         private static getProyectedWayPoint(position, start, end)
            { ... }
         public void Execute(int index)
            FirsTriangle = indexPaths[index].Firsttriangle
            NextPoint = indexPaths[index].NextPoint
            fordwarDir = indexPaths[index].fordwarDir
            indexPath = indexPaths[index].indexIni
            Length = indexPaths[index].indexFin
            recalculate = indexPaths[index].recalculate

            si recalcular es 0 y no estamos en el último triangulo
            entonces
                  recalculate = Random(10,15)
                  NextPoint =getProyectedWayPoint(particlesPosition[
                  index], waypoints[indexPath-1].start, waypoints[ind
                  exPath-1].end)
                  FirsTriangle = 0
```

```
sino entonces
      ++recalculate;
indexPaths[index] = new SMBPath { indexIni=indexPath,
indexFin=Length,Firsttriangle=FirsTriangle,  NextPoint=
 NextPoint,  recalculate  =  recalculate,  fordwarDir  =
 fordwarDir}
```

Viendo el pseudocódigo anterior se puede ver mucho más claro cómo funcionan los métodos paralelos. En la parte superior de la función se pone explícitamente entre llaves que el método utilice burst compiler (en versiones posteriores esto ya no hace falta) y dentro de la función se explicitan los datos de entrada, además de definir si estos son de lectura y escritura o solo lectura. El método execute es el que es puramente paralelo ya que mientras se está tratando el método execute para un índice con valor x, otro thread lo está ejecutando a la vez para otro entity con índice y. Así pues, recibido el indice del entity a tratar en la variable index, se extraen los datos de interés de las variables de entrada y se ejecutan los cálculos. En esta ocasión se manda a recalcular el punto de paso del entity en caso de haber pasado ciertos frames. Recalcar, que el método getProyectedWayPoint es exactamente el mismo que el explicado en **[Dynamic WayPoints]**. Finalmente los datos se guardan en la lista indexPaths que es la que se corresponde con el **IDataComponent** SMBPathComponent de la **Figura 36.**

Seguidamente se mostrará el pseudocódigo de la función computePositionJob pero muy a alto nivel ya que es bastante densa.

```
computePositionJob
(Se declaran los datos de entrada en este punto)
     Execute(int index)
          Se busca si hay un personaje delante del nuestro usando
          el hashmap calculado anteriormente
          si hay un personaje delante entonces
               si sigue nuestra dirección #dot product > 0.8
               entonces
                    dependiendo de la proximidad se desacelera
                    este personaje
               sinó entonces
                    se aplica el Local Movement implementado en la
                    versión single core
          sino entonces
               se calcula el movimiento en línea recta al waypoint
               destino

          se guardan los datos calculados en el índice
          correspondiente
```

Como se puede observar, en esta ocasión la explicación del funcionamiento de computePositionJob ha sido con un alto nivel de abstracción. Esto se debe a que, aparte de que queda mucho más simple la explicación, funciones como el Local Movement o el cálculo del movimiento en línea recta a un

objetivo ya han sido explicadas anteriormente. Como aclaración, la simulación del BoxCollider se realiza buscando en posiciones/celdas del HashMap delante de la dirección forward del personaje. Esto no otorga la misma precisión que la versión single core pero es suficiente para lo que se quiere conseguir.

13. Comparativa y resultados

Explicadas detalladamente todas las implementaciones realizadas para llevar a cabo el proyecto, incluyendo los distintos problemas encontrados para llevarlas a cabo, se concluye con la comparativa final entre los resultados obtenidos entre la versión paralela final implementada y la solución que ofrecía Unity inicialmente.

Primeramente se comenzará comentando el *performance* tanto visualmente (comportamiento de los personajes/esferas) como en temas de eficiencia de la solución de Unity y después se compararán estos aspectos con el proyecto final conseguido.

Para la versión de Unity, se invocaron 2000 gameObjects, esferas en este caso, en una zona concreta del escenario y se les marcó un mismo punto de destino a todas. Todas las esferas invocadas contaban con un Navigation Agent por lo que todas seguirán el algoritmo de cálculo de puntos de paso del Navigation System para llegar a su destino. En la **Figura 37**, se puede ver una captura de la ejecución de esta prueba.

Figura 37. Movimiento en grupo usando Navigation System.

Al ejecutar el programa sucede lo que se viene recalcando des del principio del proyecto, y es que dejando de lado el cúmulo de esferas en el punto inicial, al asignarle a todas las esferas los mismos puntos de paso como ruta al punto de destino, estas se acaban alineando de forma totalmente antinatural. Como se puede ver, la solución de Unity no sirve para simular multitudes. Además como añadido se puede observar que en la zona de origen las esferas se hacen un cúmulo, en vez de formarse medianamente entre ellas y salir más o menos a la vez como se verá para la versión propia implementada.

Para medir el rendimiento se hizo uso de la herramienta Profiler de Unity que entre otras funciones permite medir el tiempo empleado en cada frame para diferentes cálculos. En esta ocasión es de interés el tiempo de cálculo de las físicas en CPU debido a que es donde se calcula el movimiento de los objetos y es lo que se ha intentando mejorar usando ECS.

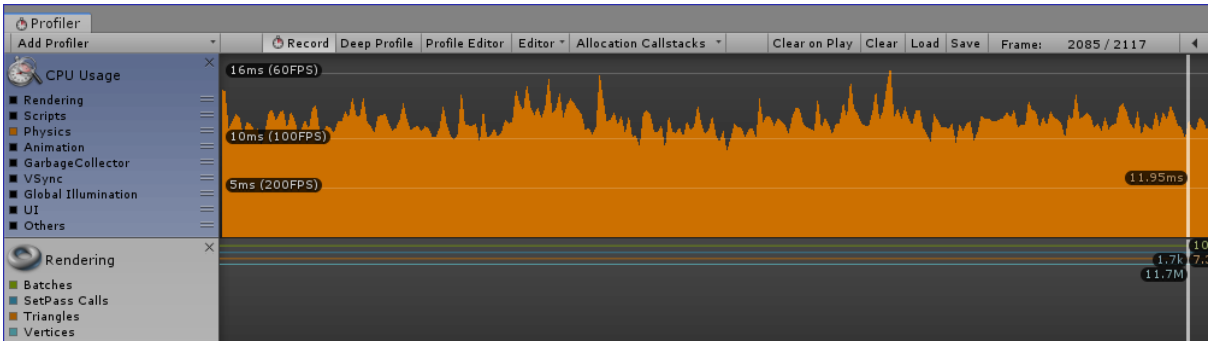


Figura 38. Gráfica del tiempo de ejecución por frame de las físicas en CPU (Navigation System).

Fijándonos en la **Figura 38**, se puede observar que por frame, Navigation System consigue mínimos de 5ms de en cálculo de físicas, 16ms de máximos y 10ms de media. Esto significa que potencialmente, si solo se tuviera en cuenta el cálculo de las físicas se podría conseguir un máximo de 100fps de media. Con 2000 gameObjects pueden no parecer unos números tan malos, sin embargo como se verá en la gráfica comparativa **Figura 41**, a medida que añades gameObjects el rendimiento cae rápidamente.

Para que la comparativa fuese justa, la ejecución utilizando el proyecto implementado en ECS se realizó con 2000 entities, además de con el mismo sitio inicial y de llegada. Visualmente, el resultado es el que se puede apreciar en la **Figura 39.**

Figura 39. Movimiento en grupo usando el proyecto implementado.

Como se puede ver, la simulación del movimiento de masas es claramente mejor. Esto se puede apreciar tanto en la forma que tienen las esferas de distribuirse a lo largo y ancho del recorrido a su destino. Además, no ocurre el atasco que sucedía en la zona de origen cuando se utilizaba Navigation System. Sin importar cuán grande sea la masa de personajes, estos se distribuyen de forma similar utilizando todo el ancho de las aristas de los triángulos que forman parte del camino mínimo, al contrario que la solución de Unity la cual tiene a alinear los personajes siempre.

En cuanto al rendimiento, quizá la parte más interesante, se obtienen los resultados visibles en la **Figura 40.**

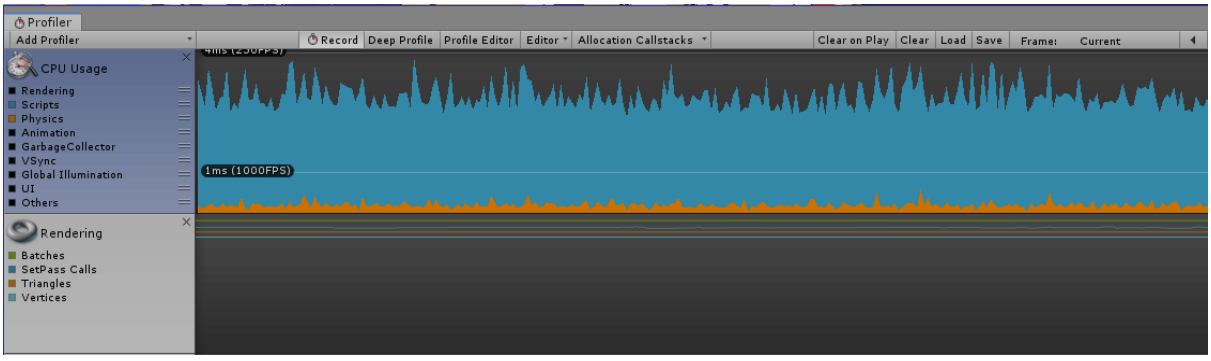


Figura 40. Gráfica del tiempo de ejecución por frame de las físicas en CPU (Proyecto en ECS implementado).

Viendo la gráfica, se puede ver que la media de los picos máximos se acerca mínimamente a la de los picos mínimos con la solución de Unity. En esta ocasión también se ha incluido el tiempo de cálculo de los scripts (franja azul) ya que el cálculo de físicas se realiza explícitamente vía scripts y no lo hace Unity. En concreto, la media de los picos máximos es de 4ms en el cálculo de físicas y de picos mínimos 1ms. Este tiempo de cálculo, crece muy lentamente hasta las 7000 entidades testeadas, sin embargo, existe otro cuello de botella que se empieza a notar a partir de las 5000 entidades y es el tiempo de rendering. Esto no forma parte del *scope* del proyecto, ya que lo que se intentaba mejorar era mejorar el tiempo de cálculo de físicas para grandes masas, pero es importante destacar que no se han realizado pruebas con más entities por este motivo. Aún así, aunque se pueda pensar que tener un buen rendimiento con una cantidad de personajes que ni el propio motor puede mover sea inútil, esto no es así. Si al tener 2000 personajes en escena el cálculo de físicas ya perjudica el tiempo de cálculo total de un frame, utilizar un método que convierte ese tiempo en negligible, abre las puertas a utilizar algoritmos más complejos y pesados para la IA sin perjudicar el tiempo de un frame.

A continuación se mostrará una gráfica con la evolución de los tiempos de cálculo de las físicas (y scripts en caso de ECS) por frame tanto en la solución de Unity como en la propia.

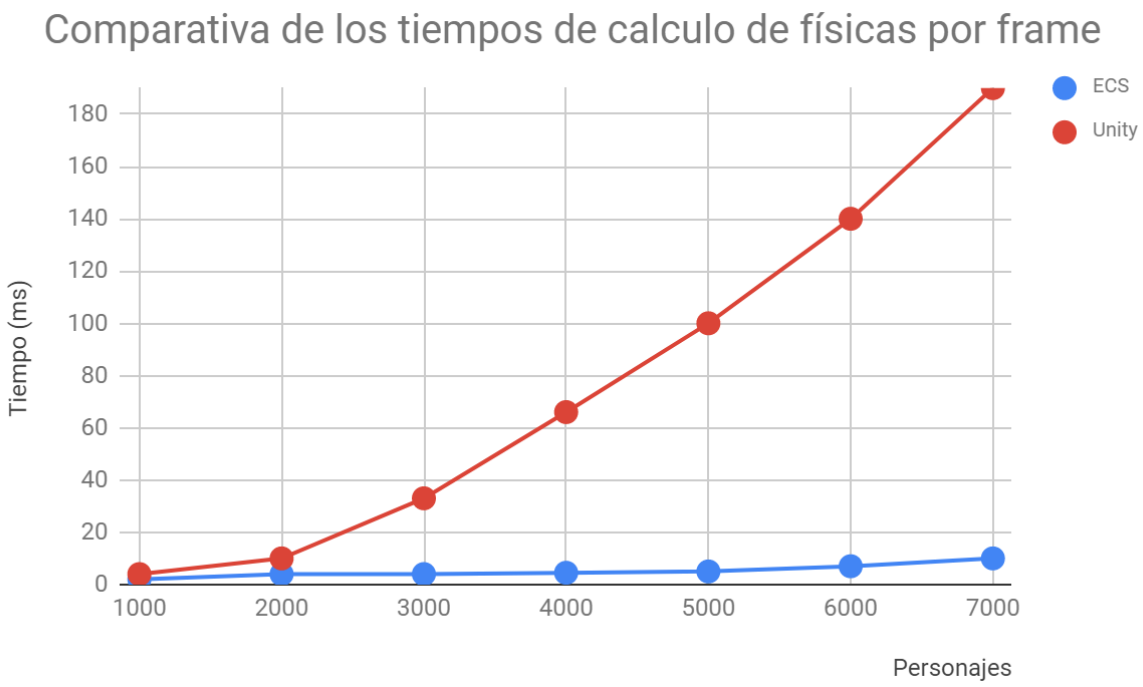


Figura 41. Gráfica comparativa entre Unity y el proyecto implementado con ECS.

Fijándonos en la línea que describe Unity se puede observar que esta presenta un crecimiento si bien no exponencial, es más que lineal del tiempo de cálculo de las físicas. En cuanto al proyecto desarrollado en ECS se puede ver que moviendo 7000 personajes en escena apenas tarda 10ms lo que nos da unos 100 potenciales frames por segundo.

Concluyendo, tanto en movimiento grupal, como en términos de eficiencia, la solución implementada es claramente mejor que la que ofrece Unity. Sin embargo, a día de hoy, Unity

presenta bastantes limitaciones en su tecnología ECS que siguen haciendo más atractiva y sencilla la solución de usar Navigation System. Aún así, este proyecto es el claro ejemplo de todo el potencial que estas tecnologías presentan de cara a un futuro cercano.

14. Futuros pasos

A continuación se comentan algunas mejoras que se podrían haber introducido en el proyecto, pero que por falta de tiempo o debido a su complejidad de implementación no se han llevado a cabo.

- Recalcular el camino en caso se ser arrastrado por la marea de personajes. Cuando se utiliza gran cantidad de personajes, suele ocurrir que algunos personajes se salen del triángulo del camino asignado. Esto produce que mientras la marea avanza hacia el destino, estos personajes fuera del camino intentan volver al triángulo en el que se quedaron mientras son arrastrados hacia delante. Como resultado, cuando la gran mayoría de personajes llegan al punto objetivo se ve observa como unos cuantos personajes van en contra dirección buscando su respectivo triángulo para luego volver al destino.

- Mejorar el recorrido de un personaje a través de un camino de triángulos. Dado las característica con las que se genera la malla de triángulos de Unity, la proyección ortogonal sobre la arista no es la mejor solución para calcular el punto de paso. Una mejor aproximación podría ser utilizar como proyección la interpolación entre el vector dirección que lleva el personaje más la media del vector director del triángulo en el que está este y el siguiente.

- Poder utilizar animaciones. Debido a las limitaciones de Unity con ECS, es imposible utilizar animaciones en Entities. Aún así, no se ha descartado que para futuras versiones no hagan este añadido. También se ha investigado la posibilidad de utilizar el GPU instancing **[27]** de Unity. Esta funcionalidad está basada en el instancing que permite renderizar uno o pocos objetos en múltiples sitios diferentes con pequeñas variaciones en su malla de primitivas.

- Paralelizar la creación de los pathings. Debido a las limitaciones de Jobs System, pasar un grafo tal y como se definió al principio del proyecto no es posible ni siquiera aplicando grandes modificaciones. Sin embargo, sí que se podría llegar a reimplementar el A* dentro de una función paralela para paralelizar el cálculo de pathings. Esto no se realizó ya que en el momento habían prioridades que afectaban directamente al *performance* final e implementar el A* utilizando tipos de datos más simples hubiera significado dedicarle demasiado tiempo de análisis.

- Publicación como Asset. Debido a la falta de tiempo no se ha podido llegar a publicar el proyecto como producto en la Asset Store. Esta publicación significa añadir oferta a las opciones que ofrece Unity para desarrollar videojuegos y/o aplicaciones.

15. Conclusiones

Como es común entre los estudiantes de ingeniería informática, desde siempre el campo de los videojuegos ha sido un campo sobre el cual me he sentido fuertemente atraído. A lo largo de la carrera se barajó la idea de desarrollar un videojuego como trabajo final de grado, sin embargo, gracias a la asignatura de videojuegos me pude quitar esa espina. Así pues, sin una idea clara en mente, el director de este proyecto me propuso uno muy interesante a mi parecer. Este interés aparece cuando vi que con lo aprendido estos años de Universidad se pueden mejorar las funcionalidades que ofrecen productos de grandes empresas como Unity. Además, el desarrollo de este proyecto significaba poder ampliar la oferta de posibilidades que ofrece Unity para la implementación de videojuegos.

A nivel personal creo que este proyecto me ha aportado muchas cosas. La primera y más importante, la importancia de saber adaptarse a cualquier cambio de planes y saber reformular los problemas según aparezcan. En el mundo laboral de hoy en día las empresas buscan gente que sea dinámica, creativa y sobretodo flexible, por lo que un proyecto de estas características que utiliza tecnologías poco conocidas, ha sido una gran experiencia que potencian estos tres aspectos.

En la era de la información que estamos viviendo, en el lapso de tiempo entre que se desarrolla un producto y finalmente se lanza, las exigencias del usuario pueden cambiar severamente. Para este trabajo, la metodología Scrum ha facilitado poder adaptar este feedback de forma natural al proyecto, pudiendo modificar este sin perder el rumbo del mismo. En concreto, se ha visto la importancia de contar con el feedback del usuario, en este caso representado por el Director y Codirectora, para reenfocar el partes del proyecto en varios sprints.

Así pues, gracias al trabajo duro y la ayuda de los directores, se ha conseguido satisfactoriamente los objetivos principales del proyecto, el cálculo de puntos de paso para muchos personajes de forma eficiente. Es una pena que por falta de tiempo el proyecto no haya podido ser publicado en la Asset Store, sin embargo no es algo que se descarte en un futuro, añadiendo los puntos comentados en **[Futuros pasos]**. Los dos puntos más interesantes a añadir sería el cálculo paralelo de los pathings y poder añadir animaciones. Con estas dos mejoras se conseguiría un Asset realmente interesante de simulación de multitudes, sobre todo en videojuegos del estilo Real Time Strategy (RTS).

Finalmente, me gustaría agradecer de todo corazón al director de proyecto por su apoyo e interés en el mismo des del primer hasta el último dia. Gracias a él, muchas de las dudas que me han aparecido en el proyecto se han podido solucionar de forma más llevadera. De la misma manera agradecer a la coDirectora, que a pesar de su poca disponibilidad, ha podido dar otro enfoque a muchos de los problemas que han aparecido en el proyecto. Para concluir, también me gustaría agradecer a las personas del ámbito personal que me han apoyado incondicionalmente en los años más duros de la carrera.

16. Referencias

[1] Article title: Unity - New way of CODING in Unity! ECS Tutorial
Website title: YouTube
URL: https://www.youtube.com/watch?v=_U9wRgQyy6s (accedido 28-02-2019)

[2] Article title: Get Started with the Unity* Entity Component System (ECS), C# Job System, and Burst Compiler
Website title: Software.intel.com
URL:https://software.intel.com/en-us/articles/get-started-with-the-unity-entity-component-system-ecs-c-sharp-job-system-and-burst-compiler (accedido 28-02-2019)

[3] Article title: [Unity3D + C#] PathFinding con NavMesh
Website title: Miguel Vedoya
URL:https://miguelvedoya.com/2017/04/14/unity3d-c-pathfinding-con-navmesh/ (accedido 28-02-2019)

[4] Article title: Arongranberg.com – Game Development with Unity 3D
Website title: Arongranberg.com
URL:https://arongranberg.com/

[5] Article title: Group Pathfinding & Movement in RTS Style Games
Website title: Gamasutra.com
URL:http://www.gamasutra.com/blogs/AndrewErridge/20180522/318413/Group_Pathfinding__Movement_in_RTS_Style_Games.php (accedido 28-02-2019)

[6] Article title: Qué es la metodología Scrum, todo lo que necesitas saber
Website title: Luís Gonçalves
URL:https://luis-goncalves.com/es/que-es-la-metodologia-scrum/ (accedido 28-02-2019)

[7] Article title: A* search algorithm
Website title: En.wikipedia.org
URL:https://en.wikipedia.org/wiki/A*_search_algorithm (accedido 28-02-2019)

[8] Article title: Clearance for diversity of agents' sizes in Navigation Meshes
Website title: Red3d.com
URL:https://upcommons.upc.edu/bitstream/handle/2117/27159/elsarticle-template-3-num-cag.pdf?sequence=1&isAllowed=y (accedido 02-03-2019)

[9] Article title: Steering Behaviors For Autonomous Characters
Website title: Red3d.com
URL: http://www.red3d.com/cwr/steer/gdc99/ (accedido 02-03-2019)

[10] Article title: About Us | Our Story | CompAnalys
Website title: Salary.com
URL: https://www.salary.com/about-us/ (accedido 17-04-2019)

[11] Article title: Hourly wage for Application Programmer/Analyst I | Salary.com
Website title: Salary.com
URL:https://www1.salary.com/application-programmer-analyst-i-hourly-wages.html
(accedido 17-04-2019)

[12] Article title: Hourly wage for Applications Programmer I | Salary.com
Website title: Salary.com
URL: https://www1.salary.com/Applications-Programmer-I-hourly-wages.html
(accedido 17-04-2019)

[13] Article title: Hourly wage for Software Tester I | Salary.com
Website title: Salary.com
URL: https://www1.salary.com/software-tester-i-hourly-wages.html (accedido 17-04-2019)

[14] Article title: Hourly wage for Engineer - Specialist/Project Lead | Salary.com
Website title: Salary.com
URL:https://www1.salary.com/engineer-specialist-project-lead-hourly-wages.html
(accedido 17-04-2019)

[15] Article title: Bitllet senzill | Transports Metropolitans de Barcelona
Website title: Tmb.cat
URL:https://www.tmb.cat/ca/tarifes-metro-bus-barcelona/senzills-i-integrats/bitllet-senzill
(accdedido 09-03-2019)

[16] Article title: Només rodalia
Website title: Rodalies de Catalunya
URL:http://rodalies.gencat.cat/ca/tarifes/servei_rodalia_barcelona/nomes_rodalies/
(accedido 09-03-2019)

[17] Author: Mahalakshmi, M; Sundararajan, DR M
Article title: Traditional SDLC Vs Scrum Methodology – A Comparative Study
Journal: The International Journal of Emerging Technology and Advanced Engineering, 192-195, Volume 3, Issue 6, June 2013
URL:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.413.2992&rep=rep1&type=pdf (accedido 18-04-2019)

[18] Article title: ProBuilder
Website title: Unity
URL:https://unity3d.com/es/unity/features/worldbuilding/probuilder (accedido 20-04-2019)

[19] Article title: MagicaVoxel
Website title: Ephtracy.github.io
URL:https://ephtracy.github.io/ (accedido 20-04-2019)

[20] Article title: Möller–Trumbore intersection algorithm
Website title: En.wikipedia.org
URL:https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorith (accedido 20-04-2019)

[21] Article title: Unity - Scripting API: Vector3.MoveTowards
Website title: Docs.unity3d.com
URL:https://docs.unity3d.com/ScriptReference/Vector3.MoveTowards.html (accedido 24-04-2019)

[22] Article title: Unity - Scripting API: Physics.OverlapBox
Website title: Docs.unity3d.com
URL:https://docs.unity3d.com/ScriptReference/Physics.OverlapBox.html (accedido 24-04-2019)

[23] Article title: Entity Component System | Package Manager UI website
Website title: Docs.unity3d.com
URL:https://docs.unity3d.com/Packages/com.unity.entities@0.0/manual/index.html (accedido 3-05-2019)

[24] Article title: Unity - Scripting API: JobHandle
Website title: Docs.unity3d.com
URL:https://docs.unity3d.com/ScriptReference/Unity.Jobs.JobHandle.html (accedido 3-05-2019)

[25] Article title: IJobNativeMultiHashMapMergedSharedKeyIndices | Package Manager UI website
Website title: Docs.unity3d.com
URL:https://docs.unity3d.com/Packages/com.unity.collections@0.0/api/Unity.Collections.IJobNativeMultiHashMapMergedSharedKeyIndices.html#Unity_Collections_IJobNativeMultiHashMapMergedSharedKeyIndices_ExecuteFirst_System_Int32_ (accedido 3-05-2019)

[26] Article title: How to implement a Fluid Simulation on the CPU with Unity (ECS/Job System)
Website title: Medium
URL:https://medium.com/@leomontes_60748/how-to-implement-a-fluid-simulation-on-the-cpu-with-unity-ecs-job-system-bf90a0f2724f (accedido 3-05-2019)

[27] Park H., Han J. (2008) Fast Rendering of Large Crowds Using GPU. In: Stevens S.M., Saldamarco S.J. (eds) Entertainment Computing - ICEC 2008. ICEC 2008. Lecture Notes in Computer Science, vol 5309. Springer, Berlin, Heidelberg
URL: https://link.springer.com/content/pdf/10.1007%2F978-3-540-89222-9_24.pdf (accedido 31-05-2019)

Apéndice A

Grupo de Tarea	Codigo	Nombre de la Tarea	Tiempo de la tarea (horas)	Tiempo del grupo de Tarea (horas)	Dependencias
Toma de contacto(TC)	TC1	Estudio de Navigation Mesh	5h	10	---
	TC2	Estudio de ECS	5h		
Gestión de proyecto(GEP)	GEP1	Contexto y alcance	20	70	GEP1<GEP2, GEP2<GEP3, GEP3<GEP4
	GEP2	Planificación	15		
	GEP3	Presupuesto y sostenibilidad	15		
	GEP4	Entrega final	20		
	GEP5	Documentación	60h	60h	GEP4<GEP5
Implementación: A*(IA)	IA1	Escenario prototipo	10	70	TC1<IA1, TC2<IA1, IA1<IA2, IA2<IA3, IA3<IA4
	IA2	Generación de un grafo virtual	25		
	IA3	Implementar A*	25		
	IA4	Pintar camino mínimo obtenido	10		
Implementación: Dynamic WayPoints(ID)	ID1	“Static” WayPoints	10	60	IA4<ID1, IA4<ID2, IA4<ID3,
	ID2	Dynamic WayPoints	20		
	ID3	Recalcular camino si hay atasco	30		
Implementación: Local Movement	IL1	Seek	40	70	IA4<IL1, IA4<IL2,
	IL2	Obstacle avoidance	30		
Implementación: ECS	IE1	Estudio de implementacione	45		IL1<IE1,

		s similares		90	IL2<IE1, IE1<IE2
	IE2	Traducción a ECS	45		
Publicación Asset Store y manual de usuario	PA1	Creación de manual de usuario	40	80	IE2<PA1, PA1<PA2
	PA2	Publicar en Asset Store	40		
Reuniones	RN1,RN2, RN3,RN4, RN5,RN6, RN7,RN8, RN9	Reuniones	10	10	RN1<RN2, RN2<RN3, RN3<RN4, RN4<RN5, RN5<RN6, RN6<RN7, RN7<RN8, RN8<RN9
Total				520h	