Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Ampliación del Asset Game Artifical Intelligence para Unity incorporando Behavior Trees

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: Jorge Andreu Royo

**Tutores**: Ramón Pascual Mollá Vayá
Salvador España Boquera

2020-2021

# Agradecimientos

“A mi familia y amigos, por aguantar mis constantes preocupaciones durante la realización de este trabajo. A los profesores, por todo el conocimiento que nos han dado. A mis tutores, por su infinita paciencia y gracias los cuales este trabajo ha sido posible.”

# Resumen

En el presente proyecto se lleva a cabo la ampliación del llamado *Asset* GAIA (*Game Artificial Intelligence API),* incorporando *Behavior Trees* (Árboles de Comportamiento), como tecnología para construir inteligencias artificiales que dictarán el comportamiento de NPCs (*Non-Playable Characters*) en videojuegos de *Unity*. Además, se construye un módulo en *Unity* que incorpora este *asset* y el cual puede ser adquirido por cualquier desarrollador a través de la *Asset Store* de *Unity*.

**Palabras clave:** *Asset,* NPC, *Behavior Tree, Unity,* Inteligencia Artificial

# Resum

En el present projecte es du a terme la ampliació del nomenat *Asset* GAIA (*Game Artificial Intelligence API),* incorporant *Behavior Trees* (Arbres de Comportament), com a tecnologia per a la construcció d'intel·ligències artificials que dictaran el comportament de NPCs (*Non-Playable Characters*) en videojocs de *Unity*. A més, es construïx un mòdul en *Unity* que incorpora aquest *asset* i el cual pot ser adquirit per qualsevol desenvolupador a través de la *Asset Store* de *Unity*.

**Paraules clau:** *Asset,* NPC, *Behavior Tree, Unity,* Intel·ligència Artificial

# Abstract

In this project, the extension of the so-called Asset GAIA (Game Artificial Intelligence API) is carried out, incorporating Behavior Trees as a technology to build artificial intelligences that will dictate the behavior of NPCs (Non-Playable Characters) in video games developed in Unity. In addition, a module is built in Unity that incorporates this asset and which can be purchased by any developer through the Unity Asset Store.

**Keywords:** Asset, NPC, Behavior Tree, Unity, Artificial Intelligence

# Tabla de contenidos

# Índice de figuras, tablas e ilustraciones

# Índice de abreviaturas

- BT: Behavior Tree
- BTs: Behavior Trees
- FSM: Finite-State Machine
- IA: Inteligencia artificial
- IAs: Inteligencias artificiales
- API: Aplication Programming Interface
- ROS: Robotic Operating System
- TFG: Trabajo de Fin de Grado
- GAIA: Game Artificial Intelligence API
- XML: Extensible Markup Language

# 1. Introducción

En el presente capítulo se realiza una breve exposición de la tecnología sobre la que se ha trabajado, así como de los proyectos que han servido como punto de partida para la realización de este trabajo. En esta dirección, se procede a hablar de los aspectos que han motivado este trabajo, los objetivos del mismo y la metodología de trabajo seguida para su ejecución.

## 1.1 Motivación

La inteligencia artificial es una herramienta de uso masivamente extendido en el mundo de los videojuegos. Da igual lo insignificante que sea. Siempre que se desee incluir en un juego personajes con los que el jugador pueda interactuar, ya sea para hablar, comerciar, pelear, etc., será necesario determinar cómo se va a comportar este personaje. Existen multitud de herramientas a través de las cuales se puede programar dicho comportamiento, siendo los Behavior Trees (BT) [37] una de ellas.

El *Asset* de GAIA es un proyecto de fin de máster desarrollado por José Alapont [7] que ofrece al programador una interfaz sencilla para programar máquinas de estados finitos (FSM). Sin embargo, la tendencia actual en el campo de la inteligencia artificial (IA) en videojuegos favorece la aplicación de BTs frente a FSM como herramienta para la definición del comportamiento de los personajes.

## 1.2 Objetivos

Partiendo de la API procedente de trabajo realizado por José Alapont (Septiembre de 2014) y en base a su aplicación y uso en el desarrollo del trabajo de fin de grado de Nicolás Gil Soriano (Curso 2014-2015) [6], se derivan los objetivos de la realización de este trabajo de fin de grado.

En primer lugar, con la realización de este trabajo se pretende realizar un estudio en profundidad sobre los BTs, con el objetivo de conocer la situación actual de los mismos en el mercado, sus campos de aplicación y avances tecnológicos, así como las herramientas y librerías más utilizadas para trabajar con los mismos.

El objetivo de este estudio es primera y principalmente la obtención de la información necesaria para poder trabajar con BTs, y así poder llevar a cabo la

realización del segundo objetivo: la expansión de la API de José Alapont, sobre la cual se van a integrar los BTs como una herramienta más a disposición del programador para la implementación de inteligencias artificiales en sus videojuegos.

Por otro lado, con la expansión de la API se pretende alcanzar el desarrollo de un producto con capacidad de comercialización, siendo el objetivo la creación de un módulo que pueda ser adquirido a través de la Asset Store de Unity.

La creación de este módulo sirve tanto como una exploración de la Asset Store y las posibilidades que ofrece a programadores, como una motivación extra para la conclusión de este trabajo. Se pretende así también facilitar el acceso al API extendido a todo aquel que quiera ponerlo en práctica en sus proyectos, tanto desarrolladores con ánimo de lucro como en actividades docentes.

Es destacable que la comercialización del módulo no tiene pretensiones de lucro, dado que el precio del mismo sería de nada menos que 1 céntimo.

## 1.3 Metodología

La metodología seguida ha sido en primer lugar el estudio previo de la bibliografía pertinente sobre los BTs, así como de la API de José Alapont y de la implementación de la misma llevada a cabo por Nicolás Gil Soriano en su trabajo de fin de grado.

Posteriormente se ha procedido a la selección de una librería de BTs y a su implementación el el *Asset* de GAIA.

Una vez extendida la API, se ha generado un módulo con la misma y se ha registrado en la *Asset Store* de *Unity* como un producto que puede ser adquirido por cualquiera.

Se describe a continuación de manera extensa en cada uno de los respectivos apartados como se ha llevado a cabo cada uno de los pasos descritos.

# 2. Herramientas utilizadas

Durante la realización de este trabajo se han empleado diversas herramientas para el desarrollo del mismo, la redacción de la memoria y la realización de las pruebas necesarias.

- Unity [38]: Se de una plataforma de desarrollo de videojuegos en tiempo real en lenguage C#. Esta ha sido empleado para llevar a cabo las pruebas realizadas para este trabajo.
- Visual Studio: Entorno de programación empleado. Programación realizada en C#.
- Panda BT Free [15]: *Asset* y librería de BTs en *Unity*.
- Microsoft Word: Editor de texto empleado para la redacción y confección de esta memoria.
- Office 365: Plataforma Cloud utilizada para almacenar y compartir ficheros durante la realización de este trabajo.
- Asset Store de Unity [39]: Tienda online y plataforma de distribución de multitud de elementos, herramientas y servicios para utilizar en el desarrollo de videojuegos y aplicaciones en *Unity*.

# 3. Estudio de la tecnología empleada: Los Behavior Trees

## 3.1 ¿Qué son los Behavior Trees?

Los Behavior Trees (BTs) son modelos matemáticos sobre planes de ejecución, expresados como conjuntos de tareas cambiantes cuya ejecución es controlada por una estructura jerárquica en forma de árbol dirigido.

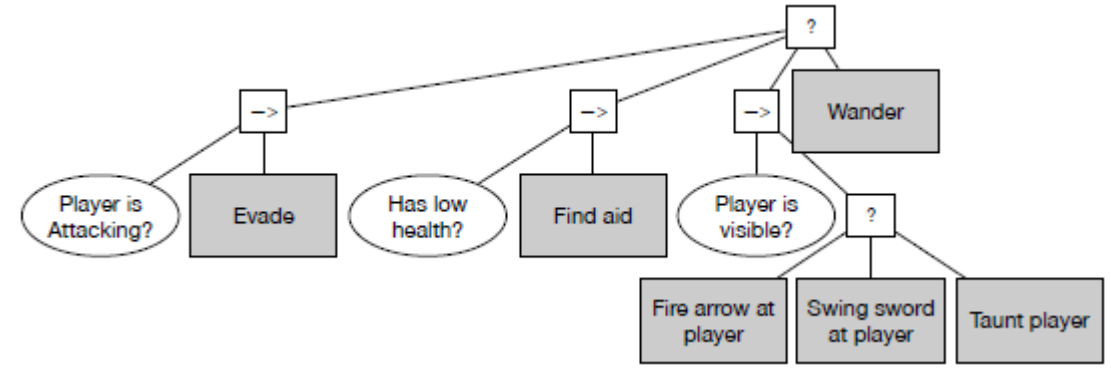


*Ilustración 1. Ejemplo de BT empleado en un shooter. Fuente: A Survey of Behavior Trees in Robotics and AI [1] (pág. 1)*

En un BT, como en cualquier árbol dirigido, existe un nodo raíz del cual parten otros nodos denominados nodos hijo. Estos nodos hijo tiene padre, siendo este su nodo antecesor. En el árbol existen nodos internos y nodos hoja. Estos nodos hoja no tienen hijos. En el caso de los BT, los nodos hoja son conocidos como nodos de ejecución y representan las tareas que puede realizar el BT. Los nodos intermedios se denominan nodos de control de flujo y expresan condiciones a partir de las cuales se producen cambios de estado en el BT.

Los BT se utilizan en muy diversos ámbitos, aunque tienen su mayor presencia en el desarrollo de inteligencias artificiales para videojuegos.

Se caracterizan frente a otras herramientas de desarrollo de inteligencias artificiales por su reactividad, entendida como la capacidad de adaptarse a los cambios en el entorno, y modularidad.

La modularidad de los BT proviene de la encapsulación de los estados en los nodos hoja del árbol (los cuadros grises en la ilustración 1) en vez de la dispersión del

estado actual por todo el árbol. Esto es debido a que todas las tareas presentan una interfaz común, pudiendo su estado ser *success, failure* o *running,* y es el estado de las taréas el que determina el estado del árbol. Los BTs se diferencian en este aspecto frente a las FSM en la simpleza con la que se determina el estado gracias a lo explicado, que resulta en una mayor agilidad a la hora de actualizar el estado del árbol.

La reactividad de los BTs procede de la facilidad para cambiar el estado de ejecución gracias a los "ticks" que emite el nodo raíz. Estos "ticks" son señales que provocan respuestas por parte de los nodos que las reciben, y dichas respuestas permiten conocer el estado actual del árbol y realizar cambios y actualizaciones con rapidez, en función de la frecuencia de los "ticks". Los "ticks" se lanzan desde el nodo raíz de forma automática y periódica, recorriendo el árbol en su enteridad y actualizando el estado del mismo con cada "tick".

En un BT, las tareas se componen de subtareas, y la ejecución de unas depende de las otras, en función de lo que dictaminen los nodos de control en base a la información obtenida del entorno, lo cual lleva a formar una estructura flexible y fácilmente entendible por el ser humano.

### 3.1.1 Los estados

Los diferentes estados en los que se puede encontrar o puede devolver un nodo y su significado se explica a continuación:

- Un estado *success* representa que la ejecución del nodo ha tenido éxito.
- Un estado *failure*, al contrario que el estado *success*, representa un nodo que ha fracasado en la ejecución del mismo. De nuevo, lo que se entiende por fracaso dependerá del tipo de nodo, de los nodos que dependen de él, de la tarea que realice el nodo, etc.
- Un estado *running* representa una acción o tarea no terminada. Un nodo puede estar en estado *running* en diversas situaciones. Por ejemplo, la ejecución de la tarea que tiene lugar en el propio nodo no ha terminado. También puede ser que el nodo esté en estado *running* si sus hijos están en estado *running*. Al igual que los estados explicados anteriormente, dependerá del tipo de nodo y de la tarea que ejecute.

## 3.2 Estructura de un Behavior Tree

Como se ha mencionado antes, los BTs se organizan en forma de árbol dirigido, teniendo un nodo raíz o padre del que parten los "ticks", una serie de nodos de control intermedios y finalmente unos nodos hoja, siendo estos últimos las acciones.

A continuación, se describen los diferentes tipos de nodos de control, cuyo entendimiento es absolutamente necesario para poder entender el comportamiento de un BT.

### 3.2.1 Nodos Secuencia

Los nodos secuencia se representan con el símbolo "→" y especifican un conjunto de tareas que deben de ejecutarse en un orden determinado. Para que una tarea pueda ejecutarse, la anterior debe haber concluido satisfactoriamente. Si todas las tareas de un nodo secuencia concluyen de forma satisfactoria, el nodo secuencia devolverá "éxito" en su ejecución. Si una de las tareas falla, la ejecución no continua y el nodo secuencia devuelve "fracaso". Mientras que el conjunto de tareas que forman la secuencia se esté ejecutando, el nodo secuencia devolverá "en ejecución" cuando reciba "ticks".

La imagen siguiente describe en pseudocódigo el funcionamiento de un nodo secuencia:

**Algorithm 1:** Pseudocode of a Sequence node with $N$ children

```
1 for i ← 1 to N do
2     childStatus ← Tick(child(i))
3     if childStatus = running then
4         return running
5     else if childStatus = failure then
6         return failure
7 return success
```

*Ilustración 2. Pseudocódigo de un nodo secuencia. Fuente: A Survey of Behavior Trees in Robotics and AI [1] (pág. 2)*

### 3.2.2 Nodos *Fallback* (o Nodos Selectores)

Los nodos fallback, también llamados nodos selectores, se representan con el símbolo "?" y constituyen un conjunto de alternativas que permiten logran un mismo

objetivo. Estos nodos ponen en ejecución a sus nodos hijo de uno en uno de izquierda a derecha, normalmente ordenados por prioridad de ejecución, y se mantienen en estado de ejecución hasta que uno de los hijos devuelve "éxito" o todos los hijos devuelven "fracaso".

El siguiente pseudocódigo representa el funcionamiento de un nodo *fallback:*

**Algorithm 2:** Pseudocode of a Fallback node with $N$ children

```
for i ← 1 to N do
    childStatus ← Tick(child(i))
    if childStatus = running then
        return running
    else if childStatus = success then
        return success
return failure
```

*Ilustración 3. Pseudocódigo de un nodo fallback. Fuente: A Survey of Behavior Trees in Robotics and AI [1] (pág. 3)*

Es en este tipo de nodos donde se suele aplicar técnicas de aprendizaje para predecir cual será el hijo del nodo *fallback* con mayores posibilidades de tener éxito en la ejecución, de forma que se mejore la eficiencia del BT.

### 3.2.3 Nodos Paralelos

Los nodos paralelos se identifican con un símbolo de dos flechas apuntando hacia la derecha, una encima de la otra, y son aquellos que ponen en ejecución a todos sus hijos de forma simultánea. Este tipo de nodos devuelven "éxito" en la ejecución si todos o M de los N hijos devuelven "éxito", siendo M un número árbitrario menor al total. En el caso de lejemplo de la ilustración 4, si más de M nodos hijo devuelven "fracaso", haciendo imposible el éxito de la ejecución, el nodo paralelo devuelve "fracaso". Mientras que no se cumpla ninguna de la condiciones descritas anteriormente, el nodo estará en estado de ejecución. Los nodos paralelos pueden no admitir que ninguno de sus hijos falle, y por tanto cuando uno de ellos falla se cancela la ejecución del resto de los nodos hijo. Lo mismo pasa si existe un número máximo de fallos permitidos.

La siguiente imagen describe en pseudocódigo el funcionamiento de un nodo paralelo:

**Algorithm 3:** Pseudocode of a parallel node with $N$ children and success threshold $M$

```
1 for i ← 1 to N do
2     childStatus(i) ← Tick(child(i))
3     if Σ_{i:childStatus.i/=success} 1 ≥ M then
4         return Success
5     else if Σ_{i:childStatus.i/=failure} 1 > N * M then
6         return failure
7 return running
```

*Ilustración 4. Pseudocódigo de un nodo paralelo. Fuente: A Survey of Behavior Trees in Robotics and AI [1] (pág. 3)*

Este tipo de nodos presenta la problemática típica del paralelismo (problemas con el acceso a recursos compartidos y la necesidad de esperar a que todos los hijos terminen de ejecutarse para considerarse que la ejecución del nodo paralelo ha terminado). Los métodos clásicos de gestión del paralelismo encuentran su versión en los BTs como los llamados “nodo paralelo sincronizado” y “nodo paralelo de exclusión mutua”

### 3.2.4 Nodos “Decoradores” [9]

Los nodos decoradores son un tipo de nodo que solo puede tener un hijo y puede tener varias funciones. Su nombre nace del concepto de elemento decorador, utilizados en programación para cambiar la funcionalidad de algo a lo que se aplica. Existen muchos tipos de nodos decoradores. Algunos de ellos son:

→ Nodo inversor: Es aquel nodo decorador cuya función es la de devolver es estado contrario al que reciban de su hijo.

→ Nodo *succeder*: Es un nodo que solo devuelve el estado “éxito” independientemente de lo que reciba de su hijo.

→ Nodo repetidor: Este nodo procesara su hijo cada vez que este devuelva una respuesta. Suelen estar programados para llevar a cabo este comportamiento un número determinado de veces antes de devolver una respuesta a su nodo padre.

→ Nodo repetidor hasta el fallo: Como su nombre indica, este nodo procesa a su hijo hasta que el mismo devuelva *failure*.

## 3.3 Ejemplo de un BT

A continuación, se ilustra el funcionamiento de un BT a través de un ejemplo visual.

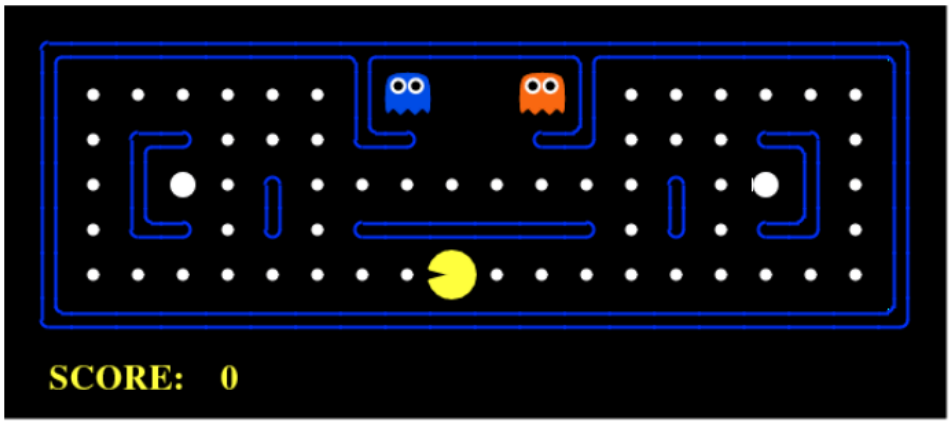


*Ilustración 5. Imagen del juego Pac-Man. Fuente: https://arxiv.org/pdf/1709.00084.pdf* [8]

En la imagen anterior se muestra el conocidísimo juego "Pac-Man". En este juego, el jugador, representado por la bola amarilla de abajo en la ilustración, debe comer todas las pequeñas bolas amarillas que hay repartidas por las calles del mapa sin ser cazado por los fantasmas, las criaturas de color azul y naranja con ojos saltones en la ilustración. El jugador también puede comer las bolas amarillas grandes para ganar momentáneamente poderes con los que puede comerse a los fantasmas y hacerlos desaparecer brevemente.

El comportamiento del jugador pude ser reemplazado por un BT, que regido por una serie de simples normas es capaz de completar satisfactoriamente una partida.

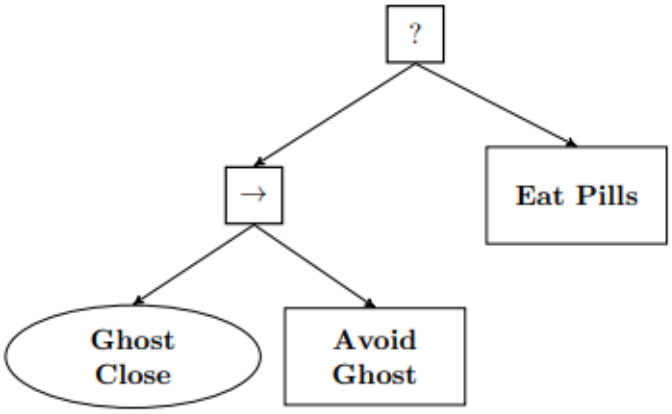


*Ilustración 6. BT del jugador en Pac-Man 1. Fuente: https://arxiv.org/pdf/1709.00084.pdf* [8]

El BT de la imagen anterior describiría el siguiente funcionamiento. En primer lugar, si hay fantasmas cerca, la figura que representa al jugador (dirigida por el BT) se alejará de los fantasmas. Si no hay fantasmas cerca procederá a comer bolitas amarillas.

El comportamiento descrito anteriormente se puede extraer del BT si se lee adecuadamente. Los nodos se ejecutan de izquierda a derecha a partir de la raíz, por lo que lo primero que hará el BT tras cada tick es entrar en el nodo secuencia. Dentro del nodo secuencia primero se ejecuta el nodo que está más a la izquierda y, como se trata de un nodo secuencia, si el nodo "Ghost close" determina que no hay ningún fantasma cerca este devolverá *failure* y por tanto el nodo secuencia devolverá *failure.* De ser así, se ejecutaría el siguiente nodo del árbol, que sería la acción de "Eat Pills". Dentro de la acción podrían determinarse otra serie de comportamientos, pero esto queda en manos del programador que diseñe la tarea. Si el nodo "Ghost close" devolviera *success*, indicando que hay un fantasma cerca, el nodo secuencia continuaría ejecutando su siguiente hijo, en este caso procediendo a ejecutar la acción de evitar al fantasma.

Dado que el nodo raíz es un nodo *fallback*, este devolverá *success* indicando que la ejecución del árbol ha tenido éxito, siempre que al menos uno de sus hijos devuelva *success.* Esto solo no pasaría en caso de que, para este ejemplo, el jugador fuera comido por un fantasma.

Con la llegada de cada nuevo tick desde el nodo raíz, se recorrerá el árbol entero y se actualizará el estado del mismo en función de los cambios observables en el entorno.

Este sencillo ejemplo facilita la comprensión de la forma en la que se leen los BTs, así como ilustra la capacidad de legibilidad de los mismos y cómo de atractivos son a la hora de representar comportamientos.

Una versión un poco más compleja de este BT sería la siguiente:

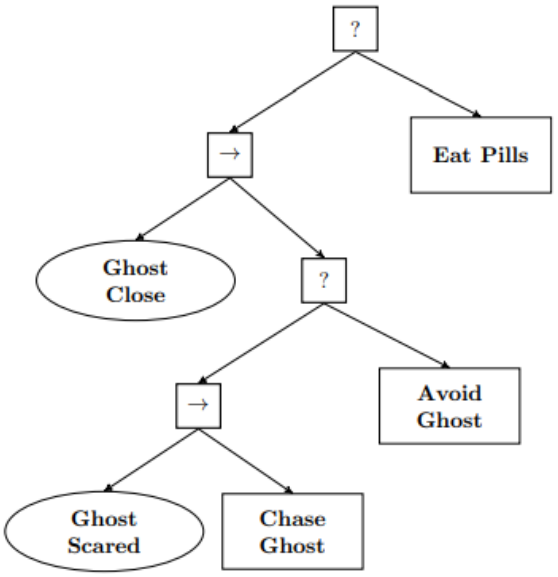


**Fig. 1.12:** BT for the Combative Behavior

*Ilustración 7. BT del jugador en Pac-Man. Fuente: https://arxiv.org/pdf/1709.00084.pdf [8]*

A través de esta ilustración se puede apreciar una simple aplicación de diseño incremental en el comportamiento del personaje del videojuego y expresa la facilidad con la que puede escalar un BT sin que se complique la legibilidad del mismo.

## 3.4 Evolución y usos de los Behavior Trees

Los BTs fueron concebidos inicialmente para ser empleados en videojuegos de diálogo como inteligencias artificiales sencillas de diseñar y entender. Se desconocen los autores originales de los mismos, aunque uno de los primeros registros de grandes avances en el desarrollo del campo es [2]. En [2] se describe el uso de un lenguage de planificación reactivo a través del cual se generan secuencias de eventos que se ramifican bajo ciertas condiciones. Estas secuencias se utilizan en el desarrollo de un videojuego de chat interactivo, donde el jugador experimenta una historia diferente cada vez dependiendo de sus acciones. Los BTs se hicieron muy populares en el campo de la IA de videojuegos tras su aplicación en el conocido videojuego “Halo 2” [4].

Las versiones iniciales de los BTs carecen de muchos de los elementos que son considerados básicos en los BTs de hoy en día. De hecho, los primeros BTs carecían de reactividad ante la falta de una forma de actualizar el estado mientras una tarea se está ejecutando [1].

Desde su concepción, se ha iterado sobre el concepto de BTs y se han agregado multitud de avances en su diseño. Los más destacables son la agregación del estado *running* [3] para las tareas, que junto con los "ticks" dota a los BTs de reactividad, la aplicación de pre y pos-condiciones en la estructura interna del BT, lo que elimina la necesidad de nodos condición, y por último los avances en *machine learning*, que dotan a los BTs de la capacidad de ajustar su funcionamiento a los resultados obtenidos en ejecuciones previas, así como supone una herramienta de diseño para BTs más eficientes, como se ha demostrado en [36].

Actualmente, los BTs se utilizan en muy diversos tipos de videojuegos, varios de ellos listados en [1], así como en el campo de la robótica (para el cual existe una extensa colección de librerías y herramientas recogidas bajo el nombre de "ROS" [33]), bots de chat (tanto para videojuegos como para otras aplicaciones [2]), conducción autónoma [34], tareas humanas secuenciables como conjuntos de acciones (incluso en el campo de la medicina [35]), y un largo etcétera. Es destacable también su uso como método de representación de flujos de tareas o *workflows*, los cuales tienen sus usos en muchos aspectos del diseño tecnológico.

Hoy en día el diseño de los BTs se sigue haciendo mayoritariamente de forma manual, gracias a la facilidad de interpretación de los mismos y a su capacidad de expresar tareas de forma sintetizada, aunque se han hecho grandes avances en la incorporación de herramientas denominadas "planificadores", cuya función es la descomposición y programación de tareas extensas que resultarían en BTs muy grandes para el diseño manual. El uso de planificadores se apoya a veces en herramientas de sistematización automática de BTs, aunque estos métodos todavía no son lo suficientemente maduros como para aplicarse exclusivamente en el diseño, siendo lo más común un diseño híbrido entre técnicas manuales y la aplicación de planificadores.

## 3.5 Behavior Trees frente a Finite State Machines (FSM)

Un BT se puede ver como un FSM con una estructura especial, según lo descrito en [4]. Es posible expresar el comportamiento descrito por un BT en forma de FSM y viceversa, pero mientras que ambos pueden ofrecer las mismas funcionalidades, la diferencia radica, como se describe en [1] y por las razones descritas a continuación, en la practicalidad de la implementación utilizando una tecnología u otra.

Mientras que las FSM transfieren el control de un agente ante cambios de estado de forma similar a las ya obsoletas sentencias "GOTO", la forma en la que funcionan los BT es mucho más versátil, siendo las transferencias de control en estos algo similar a llamadas a funciones que devuelven el valor de cada nodo.

Un ejemplo que muestra claramente la comodidad de utilizar BTs frente a FSMs esla ejecución de dos estados simultáneos. En un BT basta con utilizar un nodo paralelo y ejecutar dos tareas a la vez, mientras que en una FSM sería necesario crear una FSM entera por separado.

En cuanto a la iteración sobre el diseño y la realización de cambios, los BTs son mucho más fáciles de modificar, especialmente con la gran cantidad de herramientas para diseños visuales que existen. En muchos casos es tan sencillo como pinchar y arrastrar la tarea y desplazarla a la parte del nodo deseada. En una FSM habría que cambiar cada una de las transiciones individualmente. [5]

Por otro lado, los BT son mucho más sencillos de escalar, dado que cada subárbol tiene su propio nivel de abstracción, mientras que cada uno de los subárboles comparte una misma interfaz de ticks entrantes y devoluciones de estatus. Por el contrario, al escalar una FSM se genera una estructura monolítica extensa y compleja de

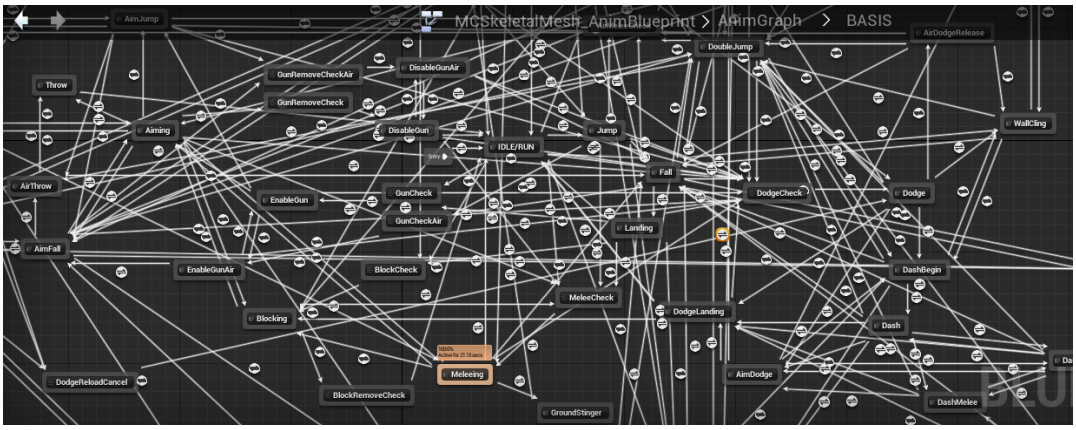


*Ilustración 8. FSM extenso con multitud de transiciones. Fuente: https://answers.unrealengine.com/storage/temp/23368-statemachinespaghetti.png*

depurar, dada la forma en la que se representan las posibles transiciones.

La siguiente frase de Alex. J. Champandard, editor jefe y fundador de AiGameDev.com, así como programador senior de inteligencia artificial en Rockstar Games, resume a la perfección la ventaja del escalado de un BT frente a una FSM:

"Sure you could build the very same behaviors with a finite state machine (FSM). But anyone who has worked with this kind of technology in industry knows how fragile such logic gets as it grows. A finely tuned hierarchical FSM before a game ships is often a temperamental work of art not to be messed with!" [8]

# 4. El Asset GAIA

El API diseñado por José Alapont es una herramienta muy completa que permite al programador diseñar sus propios autómatas de manera sencilla, rápida y, en gran medida, automatizada.

Se describe a continuación el funcionamiento general de la API, y posteriormente se explica detalladamente cada uno de los componentes que la forman.

## 4.1 Funcionamiento de la API

Se explica a continuación en líneas generales como se hace uso de la API de GAIA:

En primer lugar, se ha de cumplimentar, haciendo uso de las plantillas XML ofrecidas al programador, el comportamiento que deberá describir la FSM del tipo deseado. Para esto se han de definir los diferentes estados y las transiciones entre los mismos. Es necesario además cumplimentar un fichero, conocido como la clase “tags”, que actua como registro de punteros y es necesario para el funcionamiento de la API. Una vez se han definido las FSM en las plantillas y cumplimentado la clase “tags”, se debe indicar al controlador de la API dónde están ubicadas dichas plantillas. El controlador, al iniciar el juego, invocará un método del componente conocido como el *manager*. Dicho método hara uso del parseador, otro componente de la API, para construir las FSM a partir de las definiciones de las mismas descritas en las plantillas. Estas FSM serán guardadas en un diccionario dentro del *manager*, y podrán ser recuperadas y asignadas a objetos del juego accediendo a los métodos públicos de este componente de la API.

Un objeto del juego que quiera obtener una FSM deberá invocar un método del *manager*, “createMachine()”, indicando el nombre de la FSM deseada. Llegados a este punto, sólo quedaría programar el script del objeto del juego que ha solicitado la FSM con una serie de funciones necesarias para hacer uso de la misma, así como con las acciones y condiciones que serán utilizadas por la FSM para denotar el comportamiento definido para el objeto del juego en cuestión. Se entrará más en detalle y se dará ejemplos de estas funciones más adelante, en la sección 6.1.5

## 4.2 Componentes que forman el API

El API de José Alapont se compone de los siguientes elementos:

### 4.2.1 El *Parser* (FSM_Parser)

La sencillez del uso de este API proviene en gran parte del uso de este componente.

> Como su nombre indica, se trata de una herramienta a través de la cual se transforma la información presente en un fichero, siendo este fichero una plantilla en XML también creada por José Alapont, en un autómata finito.

El *parser* está programado de forma que el diseñador del autómata solo debe preocuparse por decidir qué tipo de FSM utilizar y por la definición de los diferentes estados, transiciones y acciones que determinarán el comportamiento del autómata.

Esta herramienta es invocada por otros componentes de la API a la hora de crear las FSM y, aunque su uso es recomendable, no es obligatorio.

La imagen siguiente procede del TFM de José Alapont y ilustra el funcionamiento del parser:

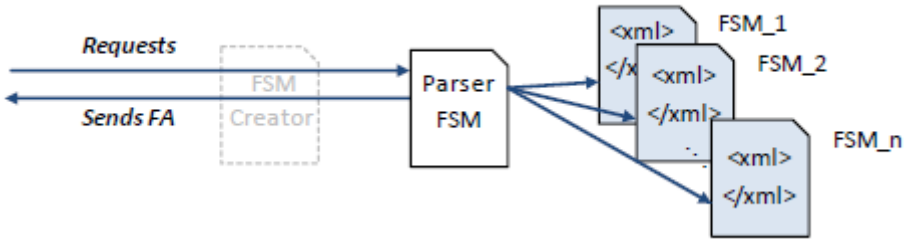


*Ilustración 9. Funcionamiento del Parser de la API. Fuente: TFM de José Alapont.*

### 4.2.2 El *Manager* (FSM_Manager)

El *manager* se inicializa a partir de un objeto del tipo “FSM_Parser” y es el componente que se encarga de recoger las FSM diseñadas por el programador del juego en los respectivos ficheros XML y asociarlas a los objetos del juego que requieran su uso. Es el *core* de la API.

Este componente utiliza un objeto de tipo diccionario que identifica las FSM existentes en base a su tipo e ID. Cuando una entidad del juego solicita una FSM, se

invoca un método al cual se le pasa el identificador de la entidad solicitante, así como la información necesaria para localizar la FSM dentro del diccionario del *manager*. El *manager* buscará en su diccionario el autómata solicitado y generará una FSM o un objeto de la clase "FSM_Machine" basado en el comportamiento del autómata solicitado.

El funcionamiento del manager es tal que permite que diferentes objetos del juego utilicen un mismo autómata, dado que para cada objeto se habrá instanciado una FSM diferente.

La imagen siguiente resume el funcionamiento del *manager:*

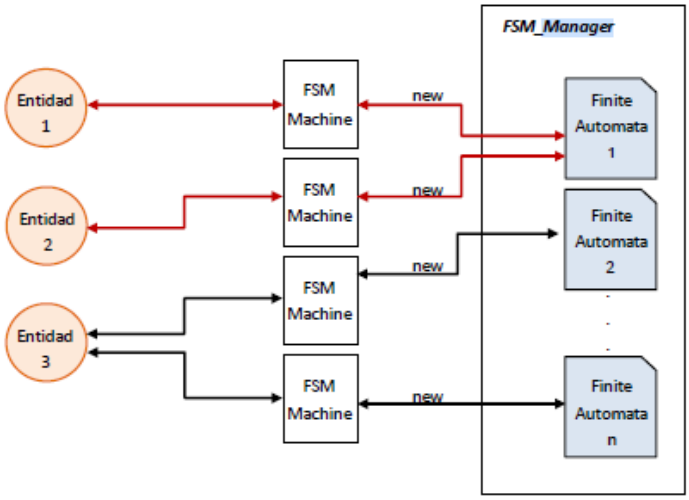


*Ilustración 10. Funcionamiento del FSM_Manager. Fuente: TFM de José Alapont.*

### 4.2.3 La máquina (FSM_Machine)

Se trata del objeto que recoge la lógica del autómata diseñado por el programador del videojuego a partir de los ficheros XML. Este objeto se encarga de recorrer la lógica del autómata en función de su tipo.

José Alapont diseñó en su TFM cuatro tipos diferentes de autómatas en base a las tendencias más comunes de la programación de IAs a través de FSM. Estos tipos son el autómata clásico, el autómata inercial, el autómata basado en pilas y el autómata basado en estados concurrentes. Dado que las FSM no son objeto de estudio en la realización de este trabajo, no se entrará en detalle en el funcionamiento de los autómatas.

### 4.2.4 El controlador (FSM_Controller)

Este componente ha sido diseñado por Nicolás Gil Soriano en su trabajo de fin de grado y es una adición muy útil a la API de José Alapont. Su propósito es la inicialización de los componentes de la API para su puesta en marcha, y se instancia dentro de un objeto en Unity.

El controlador recibe como parámetro los ficheros XML en los que se definen los autómatas e internamente crea un *parser*, un *manager* a partir del *parser* y genera los autómatas en el manager, listos para ser solicitados.

### 4.2.5 La clase *Tags*

La clase *Tags* recoge las etiquetas o *tokens* de los grafos que definen el tipo de autómata, así como de los estados, transiciones, eventos y acciones que forman parte del comportamiento definido por el programador.

Su uso no es obligatorio, aunque si recomendable, especialmente si se quiere hacer un uso simplista de la API.

La API hace uso de esta clase durante la fase de carga, llamando al método "StringToTag" definido en la misma. Este método devuelve los identificadores de los elementos referenciados al cargar los autómatas.

# 5. Estado del arte

En esta sección se lleva a cabo un análisis del estado del arte de la tecnología de Behavior Trees. En particular se analizan sus aplicaciones en el sector de los videojuegos y el de la robótica, donde esta tecnología tiene mayor uso y relevancia. Al final de la sección se presenta una tabla comparativa de las diferentes librerías estudiadas, tabla a partir de la cual se justifica la decisión adoptada sobre qué librería se incorporará en el proyecto de ampliación del Asset de GAIA.

## 5.1 El sector de los videojuegos

Como se ha comentado en apartados anteriores, los BTs son ampliamente empleados para la confección de inteligencias artificiales para videojuegos. Son una herramienta muy potente para describir con claridad comportamientos, y presentan claras ventajas frente al uso de FSMs.

Tras haber llevado a cabo un estudio exhaustivo de las tecnologías disponibles y abiertas al público, se recogen a continuación las librerías y herramientas de uso popular y de aplicación al desarrollo de BTs en lenguaje C# y en Unity con las que se ha trabajado.

### 5.1.1 Fluid Behavior Tree [12]

Se trata de una librería de BTs para C# muy completa, bien documentada y con actualizaciones recientes. Esta librería es una evolución de otra librería de BTs denominada Fluent Behavior Trees [14], la cual, aunque funcional, lleva sin recibir soporte y actualizaciones desde hace 4 años.

Mediante Fluid Behavior Trees el programador puede definir su BT de manera intuitiva y sencilla a través del código.

```
tree = new BehaviorTreeBuilder(gameObject)
    .Sequence()
        .Selector()
            .Sequence()
                .Condition("distance_far", tooFar)
                .Do(Move)
            .End()
            .Sequence()
                .Condition("distance_close", tooClose)
                .Do(MoveBack)
            .End()
            .Do("distance_ok", () =>
            {
                return TaskStatus.Success;
            })
        .End()
        .Sequence()
            .Condition("stopped", isStopped)
            .Do(Aim)
        .End()
        .Sequence()
            .Condition("stopped", isStopped)
            .Condition("close enough", shootingRange)
            .Do(Aim)
            .Do(Shoot)
        .End()
    .End()
.Build();
```

*Ilustración 11*. Construcción de un BT mediante Fluid Behavior Tree. Fuente propia.

El ejemplo de la ilustración anterior muestra un árbol de comportamiento diseñado para una de las pruebas realizadas con diferentes librerías de BTs.

Haciendo uso de la librería Fluid Behavior Tree, la creación del árbol se lleva a cabo con la definición de un nuevo objeto de tipo BehaviorTreeBuilder, sobre el cual se invocan una serie de métodos de forma concatenada y a través de los cuales se da forma a los nodos del árbol. Se genera así la estructura como la que se aprecia en la ilustración anterior, donde cada nodo comienza con un método que determina el tipo de nodo, como ".Sequence()", y el cual anida en las invocaciones ubicadas entre la invocación a este mismo método y la invocación al método "End()" todos los nodos que dependen del mismo.

Se trata por tanto de una estructura piramidal, donde el primer nodo o el nodo raíz es aquel que se define primero, y sus nodos hijo, así como los hijos de los hijos, se definen en el interior de este mismo.

Esta librería cuenta con todos los nodos estándar, además de ofrecer la infraestructura necesaria para definir nodos personalizados.

*Fluid Behavior Tree* cuenta además con un visualizador para poder ver de forma clara la ejecución del árbol en *runtime*. La ilustración siguiente muestra la representación visual del árbol de comportamiento definido en la ilustración 11, visto a través del visualizador de *Fluid Behavior Tree*.

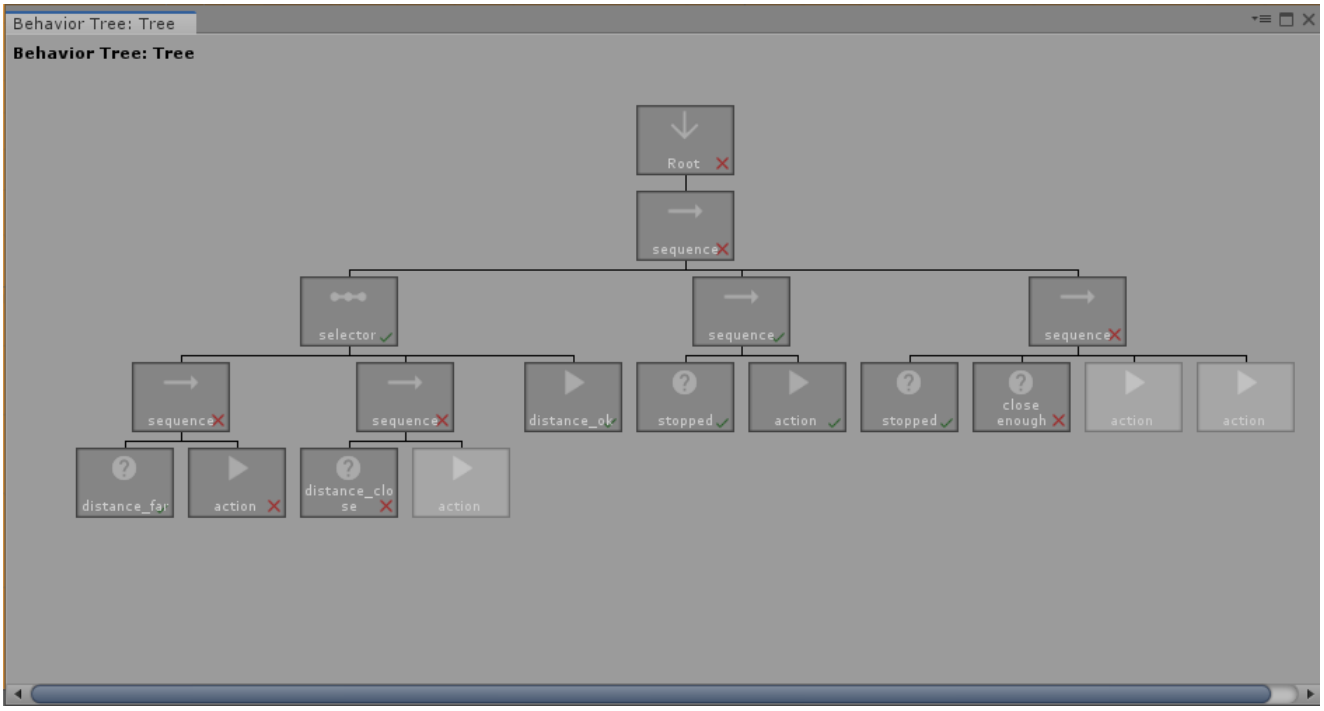


*Ilustración 12*. Árbol de comportamiento definido en la ilustración 11, visto a través del visualizador. Fuente propia.

En el visualizador podemos observar la estructura del árbol, así como los tipos de nodos y su estatus de retorno.

### 5.1.2 Panda BT Free [15] [26]

Esta librería se encuentra en la forma de un *asset* que se puede obtener de forma gratuita a través de la *Asset Store* de *Unity*. Está escrita en C# y ofrece una aproximación minimalista y ligera tanto a la ejecución como a la definición de BTs.

```
tree("Root")
    //Adjust position, aim and shoot, in that order
    sequence
        tree("Movement")
        tree("Aiming")
        tree("Shooting")

tree("Movement")
    //Check if position is too far, too close or just right and act acordingly
    fallback
        sequence
            tooFar
            Move

        sequence
            tooClose
            MoveBack

        isStopped

tree("Aiming")
    //If tank is not moving, Aim
    sequence
        isStopped
        Aim

tree("Shooting")
    //If we aimed, check if we are moving, check distance again, aim again and shoot
    sequence
        isStopped
        shootingRange
        Aim
        Shoot
```

*Ilustración 13.* Árbol de comportamiento definido mediante Panda BT. Fuente propia.

En la imagen anterior se observa la definición del mismo árbol de comportamiento que el representado en las imágenes 11 y 12, pero esta vez confeccionado haciendo uso de la biblioteca *Panda BT Free*.

En este caso, el árbol no se construye en código, sino que se recoge en un documento de texto que posteriormente se vincula a un componente en *Unity*. Se hablará más en detalle de la implementación de cada librería en el apartado de pruebas.

En este documento de texto, la definición de nodos comienza por la invocación al método “tree”, con el nombre del nodo como parámetro. El primer nodo definido será el nodo raíz. A continuación, se puede componer el resto del árbol de diferentes formas. Puede optarse por un formato monolítico, donde todo el árbol y sus nodos cuelgan de una misma invocación al método “tree”, o puede separase el árbol en subárboles.

En el caso de la imagen anterior, se puede observar cómo el nodo raíz es un nodo secuencia del cual cuelgan tres subárboles. Dichos subárboles son definidos a continuación con nuevas llamadas al método “tree”. Dentro de los nodos basta con

escribir el nombre de las tareas y condiciones que deberán ser programadas a posteriori en un script asociado al objeto en Unity.

Esta librería es particularmente sencilla de manejar debido al formato tan simple que se emplea en la definición del árbol y a que, al tratarse de un simple fichero de texto, no es necesario lidiar con el compilador durante el proceso de desarrollo si lo que se quiere es simplemente estructurar el árbol sin necesidad de trabajar en la implementación del mismo o de las tareas y condiciones de manera paralela.

Al igual que el *Fluid Behavior Tree*, *Panda BT Free* cuenta con un visualizador para poder seguir en *runtime* la ejecución del árbol. Una ventaja de *Panda BT Free* frente a *Fluid Behavior Tree* es que su visualizador no requiere que se lance a ejecución la escena para visualizar la estructura del árbol, mientras que el visualizador de *Fluid Behavior Tree* necesita del *runtime* para poder ver la estructura del árbol.


*Ilustración 14.* Componente de *Panda BT Free* en *Unity* con visualizador del árbol. Fuente propia.

### 5.1.3 Otras librerías / herramientas

Durante el proceso de escrutinio de las redes en búsqueda de herramientas y librerías de BTs se han hallado otra serie de opciones de las cuales se hablará mínimamente a continuación.

- **Unity_AI** [11]: Esta biblioteca de BTs, aunque funcional, es poco completa y algunos de sus componentes siguen en fase de desarrollo. La forma en la que se definen los árboles de comportamiento es muy incómoda, además de requerir del uso de un tipo particular para la definición de condiciones y acciones. Además, aunque incorpora un visualizador del árbol, este está construido sobre un componente obsoleto en Unity y por tanto no funciona. Esta librería lleva además 2 años sin recibir actualizaciones.
- **Aivo [13]**: Se trata de otra librería de BTs inspirada en la librería *Fluent Behavior Tree* que permite definir árboles de comportamiento a través de código. No se han realizado pruebas con esta librería, pero la misma carece de una buena documentación y de un visualizador, y la implementación del árbol es similar a la observada en Fluid Behavior Tree y Fluent Behavior Tree, donde se llevan a cabo llamadas a métodos de manera anidada desde el nodo padre hasta los nodos hijo.
- **Behavior Bricks** [18]: Esta librería de BTs se caracteriza por su editor visual, a través del cual se construyen los árboles haciendo uso de los diferentes tipos de nodos preprogramados. Este editor visual cuenta además con una interfaz para programar condiciones a partir de variables locales o de un "blackboard", un directorio que contiene variables e información a la que todos los nodos pueden acceder. Tras haber trabajado brevemente con este editor, se descartó su uso para este proyecto dadas las limitaciones encontradas en el funcionamiento del editor visual, así como la opacidad de los árboles generados y las limitaciones impuestas por los tipos de nodos con los que se puede trabajar.
- **NPBehave** [22]: Este *framework* de *Unity* está orientado a la programación de BTs basados en eventos a través del código. Aunque hace uso de la librería *BehaviorLibrary* [23], la cual no recibe actualizaciones desde 2013, NPBehave cuenta con una extensa documentación y el *framework* es mantenido activamente. Los árboles definidos con *NPBehave* no necesitan ser recorridos enteramente a cada tic, sino que el estado del mismo

evoluciona en base las necesidades de ejecución de la aplicación. De esta forma se obtiene un mejor rendimiento del árbol. No cuenta con un editor visual, pero si con un "blackboard" para compartición de datos entre nodos. El uso del "blackboard" no es obligatorio y existen alternativas al mismo. Los únicos estados de devolución posibles son éxito o fracaso; no soporta el estado *running.*

Por otro lado, existen otras herramientas de uso extendido para otros lenguajes y entornos de desarrollo que se destacan a continuación:

- **BehaviorTree.CPP** [16]

Se trata de una librería de BTs en C++ muy completa y con actualizaciones frecuentes. Cuenta con un editor gráfico y la posibilidad de importar o exportar árboles a través de ficheros XML. Cuenta además con una implementación en ROS para su uso en aplicaciones de robótica.

- **BehaviorTrees** [17]

Este proyecto agrupa una librería de BTs en C# junto con un editor gráfico para .Net Framework WinForms. La definición de árboles es muy sencilla y similar a la empleada en la librería de Fluid Behavior Tree en cuanto al anidamiento de invocaciones concatenadas a métodos para la construcción de los nodos hijo a partir del nodo raíz.

- **Py Trees** [19]

Esta librería de BTs en Python es una de las más utilizadas debido al amplio soporte que recibe. Se trata de una librería muy completa que cuenta con un editor visual, un "blackboard" para compartición de datos entre nodos y soporte para todos los nodos estándar. Cuenta además con extensiones para dar soporte a su uso en aplicaciones de robótica mediante ROS.

- **Gdx-ai** [20]:

Se trata de un *framework* de inteligencia artificial escrito en Java y que trabaja sobre libGDX, un *framework* de desarrollo de videojuegos multiplataforma en Java. Este proyecto agrupa muchas herramientas de inteligencia artificial utilizadas

comúnmente en la industria de los videojuegos y cuenta, entre otras muchas cosas, con una librería de BTs. Dicho *framework* recibe soporte relativamente limitado, aunque se constata en la descripción de Github que se planean ampliaciones.

- **BehaviorTree.js** [21]:

Esta librería es una implementación de la tecnología de BTs en JavaScript que cuenta con soporte para todos los nodos básicos, así como extensiones para dar soporte a nodos decoradores. Cuenta con facilidades para *debugging* y con un *parser* para importar árboles definidos en archivos JSON. La documentación disponible es extensa y ha recibido actualizaciones recientes.

## 5.2 El sector de la robótica

El sector de la robótica es otro donde los Behavior Trees se utilizan con regularidad. Muchas de las librerías de uso extendido en el sector de los videojuegos son también ampliamente utilizadas en el de la robótica, siempre que tengan soporte o extensiones para ROS. ROS [24], o *Robot Operating System*, es un *middleware*, o una colección de marcos de referencia para desarrollo de software para robots, que proporciona servicios como abstracción de hardware, control de dispositivos de nivel bajo, paso de mensajes entre procesos, etc. En otras palabras, ROS es un entorno en el cual se pueden desplegar las herramientas necesarias para el desarrollo y puesta en funcionamiento de aplicaciones de robótica.

Algunas de las librerías que han sido estudiadas y sobre las cuales se ha hablado en los apartados anteriores especifican explícitamente si tienen o no soporte de ROS. Esto se tendrá en cuenta como criterio para la valoración de las diferentes librerías en la tabla comparativa.

Las funcionalidades de cada una de las librerías son las mismas independientemente de si se usan para desarrollo de IAs para videojuegos o para robots. La diferencia radica en las funcionalidades extendidas que pueda tener la librería para hacer de interfaz con ROS.

## 5.3 Tabla Comparativa

A continuación, se presenta una tabla donde se comparan las librerías vistas durante el estudio del estado del arte. Los criterios utilizados en la tabla comparativa son:

- Lenguaje de programación en el que está escrita la librería.
- Soporte y mantenimiento / actualizaciones de la librería. Se considera que la librería recibe soporte si la fecha del último "commit" en *Github* es anterior a dos años.
- Se valora si la librería cuenta o no con un visualizador o un editor visual.
- Se valora si la librería cuenta con soporte o una extensión para ROS.

*Tabla 1*. Tabla comparativa de librerías de BTs. Fuente propia.

| | **Nombre** | **Lenguaje** | **Soporte** | **Visualizador/Editor** | **Extensión ROS** |
|---|---|---|---|---|---|
| **1** | Fluid Behavior Tree | C# | Sí | Visualizador | No |
| **2** | Panda BT Free | C# | Sí | Visualizador | No |
| **3** | Fluent Behavior Tree | C# | No | No | No |
| **4** | Aivo | C# | No | No | No |
| **5** | Unity AI | C# | No | No | No |
| **6** | Behavior Bricks | C# | Sí | Editor | No |
| **7** | NPBehave | C# | Sí | Visualizador | No |
| **8** | BehaviorTree.CPP | C++ | Sí | Editor/Visualizador | Sí |
| **9** | Behavior Trees | C# | No | Editor/Visualizador | No |
| **10** | Py Trees | Python | Sí | Editor/Visualizador | Sí |
| **11** | Gdx-ai | Java | Sí | No | No |
| **12** | BehaviorTree.js | JavaScript | Sí | No | No |

La información que deriva de la tabla es que, de entre las librerías de BTs en C#, el lenguaje de programación utilizado en *Unity* y en el cual se ha escrito el *Asset GAIA*, sólo cuatro de ellas cuentan con mantenimiento y actualizaciones recientes. De entre estas cuatro, una soporta ejecución orientada a eventos, por lo que no presenta un funcionamiento estándar. Mientras que esto aporta algunas facilidades, es preferible elegir una librería que sea familiar para la extensión de la API. Otra de las cuatro librerías en C# con mantenimiento reciente obliga a trabajar a través de un editor visual, lo cual es un factor limitante y dificulta la definición de BTs. Las otras dos librerías en

C# con mantenimiento reciente son *Fluid Behavior Tree* y *Panda BT Free*, las dos librerías sobre las cuales se han hecho las pruebas que a continuación se describen.

## 5.4 Pruebas realizadas

Antes de iniciar el proceso de desarrollo de la ampliación del *Asset* de GAIA se decidió llevar a cabo una serie de pruebas para determinar qué librería sería la más adecuada para el proyecto.

### 5.4.1 El banco de pruebas

Para esta prueba se utilizó el juego *Tanks*, desarrollado en *Unity* a partir de un tutorial [10] muy sencillo que sirvió a su vez de introducción al entorno de desarrollo *Unity*.

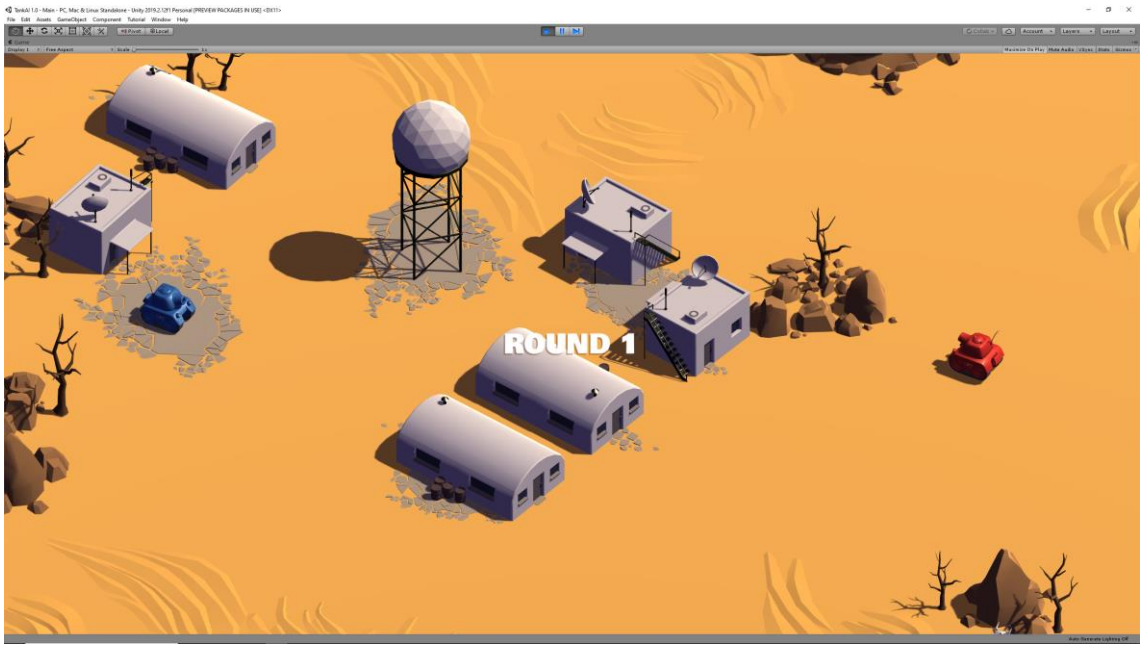


*Ilustración 15*. Juego "Tanks" ejecutado en *Unity*. Fuente propia.

*Tanks* es un juego muy sencillo, donde dos jugadores manejan cada uno un tanque mediante el uso de un único teclado. El objetivo del juego es acabar con el jugador rival a base de disparar proyectiles y esquivar.

Para poder probar las librerías de BTs se modificó el juego base, de forma que el segundo tanque fuera controlado por una inteligencia artificial.

Antes de utilizar ninguna librería de BTs se llevó a cabo una primera aproximación a la implementación de la IA mediante la redefinición del script que

controla las acciones del tanque, creando las nuevas tareas y condiciones que serán necesarias para que la IA presente el comportamiento deseado, además de añadir los elementos necesarios para permitir que la IA pueda desplazarse por el mapa.

En particular, se desea que el tanque controlado por la IA persiga al jugador a través del mapa, que se acerque a una distancia determinada y que cuando esté suficientemente cerca apunte y disparé hacia el tanque del jugador. Además, se desea que el tanque de la IA se aleje del tanque del jugador si este se acerca mucho, de modo que ambos tanques puedan continuar jugando, disparándose a una distancia justa.

Para que la IA sepa por dónde puede desplazarse, se hace uso de un NavMesh, un objeto de Unity que se agrega a la escena y que representa la superficie sobre la que un agente puede desplazarse. El NavMesh está definido sobre una capa, en este caso la capa que representa el suelo, y para cada uno de los obstáculos presentes en esta capa debe definirse si el agente podrá caminar sobre ellos o no. Se debe también configurar

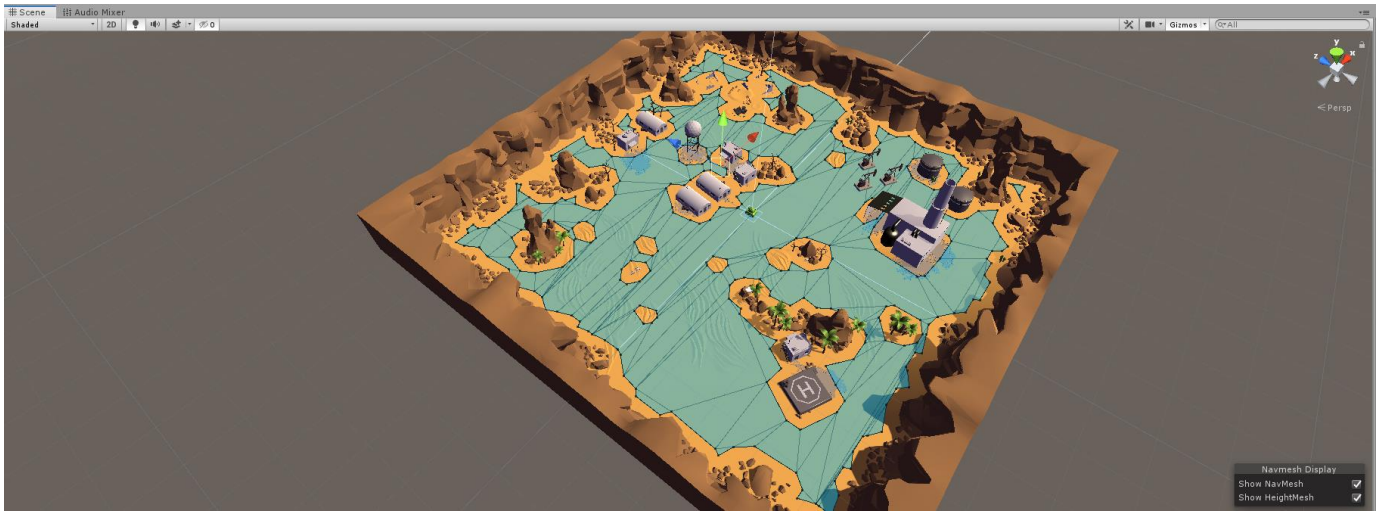


*Ilustración 16*. NavMesh generado para las pruebas sobre el juego "Tanks". Fuente propia.

las características que tendrá el agente, siendo estas su nombre, radio y altura del mismo, así como la altura de cada paso y la cuesta máxima que puede subir.

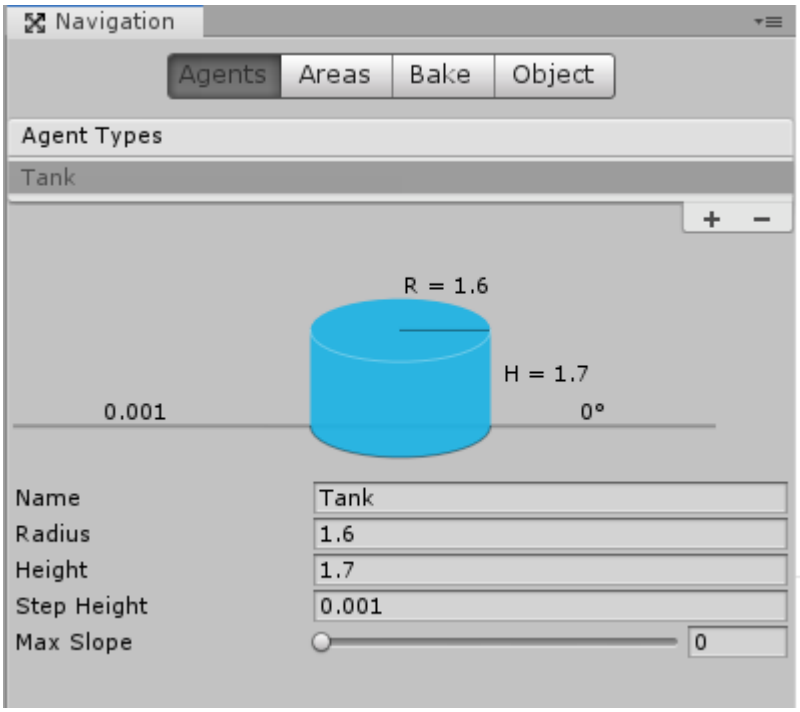


*Ilustración 17.* Configuración del agente del NavMesh. Fuente propia.

Una vez configurado todo, se genera el NavMesh mediante el botón "bake". A partir de este momento el NavMesh generado ya se puede visualizar como se ve en la ilustración 16. Faltaría añadir un componente "NavMesh Agent" al tanque y modificar el script de control del tanque para que se mueva haciendo uso del NavMesh.

En el componente NavMesh Agent se deben configurar aspectos como la velocidad del objeto, la aceleración, la distancia a la cual se parará cuando se dirija hacia una coordenada y el terreno o la capa del NavMesh por la que deberá desplazarse.

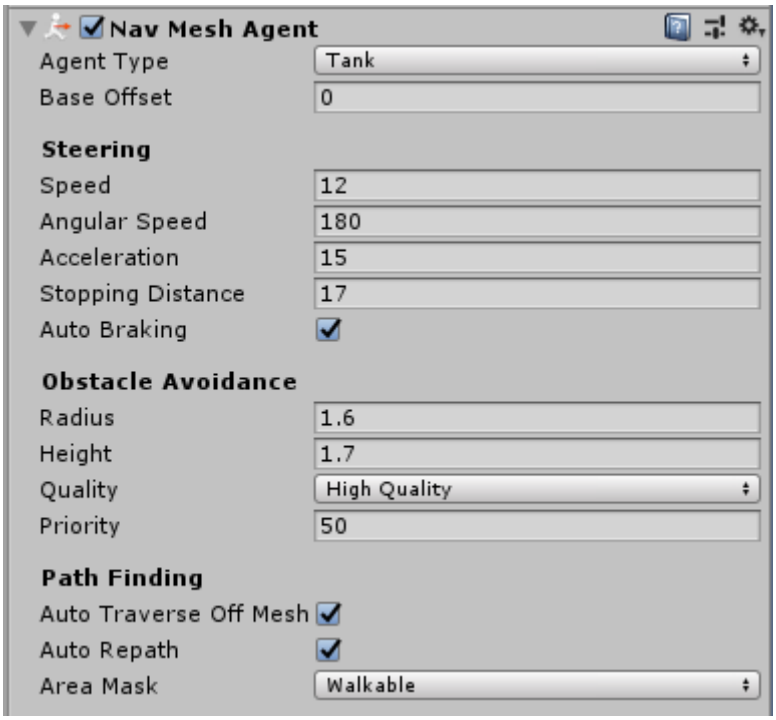


*Ilustración 18.* Componente NavMesh Agent asociado al objeto del tanque. Fuente propia.

Llegados a este punto, se procede a modificar el script que define el comportamiento del tanque.

En primer lugar, se define la acción de moverse hacia el tanque controlado por el jugador de la siguiente manera:

```
2 referencias
private void Move()
{
    Vector3 PlayerPosition = GameObject.Find("Tank(Clone)").transform.position;
    agent.SetDestination(PlayerPosition);
}
```

*Ilustración 19.* Acción de moverse mediante el uso del NavMesh. Fuente propia.

En esta función se genera un objeto que contiene las coordenadas en el espacio tridimensional del objeto que representa el tanque del jugador, determinado por el nombre del objeto del juego. Una vez obtenidas las coordenadas se da al agente la orden de desplazarse por el NavMesh hasta dicha coordenada, haciéndolo con la velocidad, aceleración y por la capa indicadas en la configuración del mismo.

Se define además una acción para apuntar al tanque del jugador:

```
private void Aim()
{
    var damping = 2;
    Vector3 PlayerPosition = GameObject.Find("Tank(Clone)").transform.position;
    var lookPos = PlayerPosition - transform.position;
    lookPos.y = 0;
    var rotation = Quaternion.LookRotation(lookPos);
    transform.rotation = Quaternion.Slerp(transform.rotation, rotation, Time.deltaTime * damping);
    Shoot();
}
```

*Ilustración 20*. Acción de la IA que apunta al tanque del jugador. Fuente propia.

Este método obtiene la posición del tanque del jugador, calcula la distancia entre la posición del jugador y la del agente, fija la posición en el eje y a 0 para permanecer apuntando a la misma altura, genera una magnitud de rotación necesaria para apuntar hacia el jugador en base a la diferencia entre las posiciones y, finalmente, aplica la rotación sobre el agente. Una vez apuntado, llama a la función de disparar.

```
private void Shoot()
{
    if (noShooting > 240 && noShootingAgain > 60)
    {
        m_TankShooting.FireAI();
        noShootingAgain = 0;
    }
}
```

*Ilustración 21*. Acción de disparar controlada por la IA del tanque. Fuente propia.

La acción de disparar es muy sencilla y se limita a una llamada condicional al método que controla la creación del objeto del proyectil definida en otro script que ha sido creado siguiendo el tutorial de programación del juego. Las condiciones impuestas son tales que la IA sólo pueda disparar como mucho una vez cada 60 fotogramas, con una condición adicional de no poder disparar durante los primeros 240 fotogramas de la ronda en los cuales el tanque aun no puede desplazarse dado que se está anunciando el comienzo de la ronda.

Por último, se define la acción para que el tanque se mueva hacia atrás siempre

que el jugador se acerque mucho a la IA:

Este método crea un vector de movimiento en la dirección frontal del tanque y lo aplica al cuerpo del objeto con signo negativo, de forma que se mueva hacia atrás. Este método no hace uso del NavMesh, sino que trabaja directamente sobre las

```
private void moveBack()
{
    // Create a vector in the direction the tank is facing with a magnitude based on the input, speed and the time between frames.
    Vector3 movement = transform.forward * m_Speed * Time.deltaTime;

    // Apply this movement to the rigidbody's position.
    m_Rigidbody.MovePosition(m_Rigidbody.position - movement);

}
```

*Ilustración 22*. Acción de desplazarse hacia atrás utilizada por la IA para evitar abusos por parte del jugador. Fuente propia.

coordenadas de posición del objeto, lo cual puede causar que dicho objeto acabe saliéndose de los límites de NavMesh. Para evitar que esto genere problemas, se permite al agente que navegue fuera de los límites del NavMesh para volver a entrar dentro del mismo.

Una vez definidas las acciones, queda definir cuándo se ejecutan y agregar algunas condiciones a las mismas.

```
void Update()
{
    Vector3 PlayerPosition = GameObject.Find("Tank(Clone)").transform.position;
    var distance = Vector3.Distance(m_Rigidbody.transform.position, PlayerPosition);
    noShooting++;
    noShootingAgain++;
    m_Rigidbody.isKinematic = true;
    m_Rigidbody.isKinematic = false;
    Move();
    if (previousPosition != m_Rigidbody.transform.position)
    {
        previousPosition = m_Rigidbody.transform.position;
    }
    else
    {
        Aim();
    }
    if (distance <= 5)
    {
        moveBack();
    }
}
```

*Ilustración 23*. Conjunto de métodos y actualizaciones de variables que se lleva a cabo cada fotograma. Fuente propia.

El método de la ilustración anterior se ejecuta cada fotograma. En este método se realizan varias cosa: En primer lugar, se obtiene la posición del tanque del jugador y se calcula la distancia entre dicho tanque y el tanque de la IA. Esta distancia se guarda como un valor en coma flotante. A continuación, se actualizan los contadores que controlan cuándo se puede disparar, se desactiva y reactiva la capacidad de movimiento del cuerpo del tanque de la IA para eliminar las posibles fuerzas aplicadas por impactos

de proyectiles y se recalcula el destino al cual debe moverse el agente. Posteriormente se ejecuta el método que controla el audio del tanque y, si se determina que el agente está parado, se apunta. Por último, se comprueba si el jugador está a 5 unidades de distancia o menos, y si es así se ejecuta el método de moverse hacia atrás.

Hasta aquí se han explicado los cambios realizados sobre el juego para soportar la IA. En esta versión del juego la IA presenta el comportamiento deseado, pero no hace uso de árboles de comportamiento.

A continuación, se detalla, para cada una de las pruebas realizadas con las librerías de BTs, el proceso de implementación y la experiencia con la librería. El objetivo de estas pruebas es replicar el comportamiento programado en la IA a través del uso de BTs.

### 5.4.2 Prueba con Fluid Behavior Tree

Para comenzar a utilizar esta librería es necesario modificar el archivo "manifest.json" del proyecto de Unity e incluir una serie de líneas que se pueden encontrar en su repositorio de Github. Mediante este cambio permitimos que el "Package Manager" de Unity pueda descargar e instalar la versión deseada de la librería directamente desde dentro del proyecto de Unity.

Una vez importada la librería, esta aparece como un "package" dentro del proyecto de Unity y ya está lista para utilizarse. Para utilizarla basta con añadir las siguientes líneas al inicio del script donde se quiera utilizar:

```
using CleverCrow.Fluid.BTs.Tasks;

using CleverCrow.Fluid.BTs.Trees;
```

A partir de este punto ya se puede proceder a definir el árbol, igual que en la ilustración 11. El árbol se define para esta prueba en el método "awake", de modo que se genere el árbol al crear el *GameObject* del tanque. El resto de las acciones deben modificarse para retornar un tipo "TaskStatus", que puede ser "Failure" o "Success". Esto se representa en el ejemplo siguiente:

```
private TaskStatus Move()
{
    if (noMoveAgain >= 30)
    {
        Vector3 PlayerPosition = GameObject.Find("Tank(Clone)").transform.position;
        agent.SetDestination(PlayerPosition);
        noMoveAgain = 0;
        return TaskStatus.Success;
    }
    else
    {
        return TaskStatus.Failure;
    }
}
```

*Ilustración 24.* Ejemplo de acción adaptada para utilizar *Fluid Behavior Tree*. Fuente propia.

Aparte de la modificación del valor de retorno, el resto de la tarea funciona de la misma forma que en la versión de la IA sin BT.

Deben definirse, sin embargo, una serie de condiciones para controlar la ejecución del árbol. Estas condiciones son:

- **tooFar**: Comprueba si la distancia al jugador es mayor o igual a un valor. Se utiliza para determinar cuándo debe ejecutarse la acción de moverse utilizado el NavMesh.
- **tooClose**: Comprueba si la distancia al jugador es menor o igual a un valor. Se emplea para determinar cuándo debe ejecutarse la acción de moverse hacia atrás.
- **isStopped**: Comprueba si el tanque está parado. Su función es determinar cuándo apuntar y disparar.
- **shootingRange**: Comprueba que la distancia al jugador es correcta (se encuentra entre el valor de moverse y el de ir hacia atrás). Se utiliza para determinar si se puede disparar.

Esta serie de condiciones son métodos que retornan un booleano.

Por último, queda añadir la línea “tree.Tick()” al método “Update()”, que se ejecuta cada fotograma, para enviar tics al árbol y recorrerlo cada fotograma.

Esta serie de modificaciones resulta en un comportamiento de la IA igual al de la versión sin BT. Ahora podemos lanzar la escena a ejecución y pausarla para ver, de entre las variables públicas del *game object* del tanque, el botón que nos permite

desplegar el visualizador de ejecución del árbol, tal y como se muestra en la ilustración 12. Con el visualizador abierto se puede quitar la pausa y comprobar el flujo de ejecución del árbol.

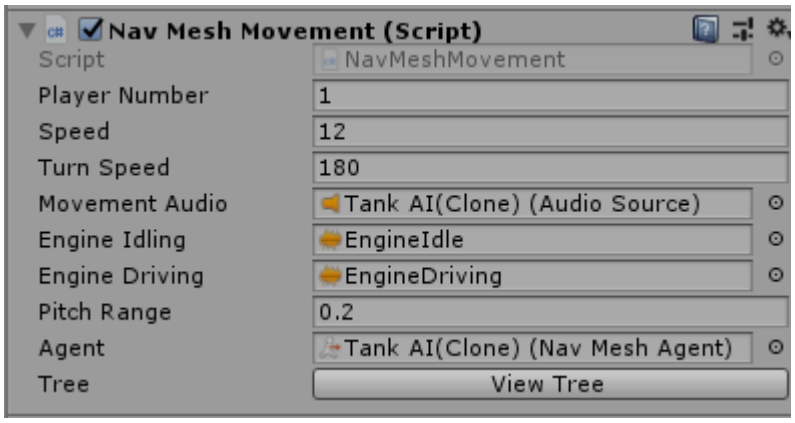


*Ilustración 25.* Script que controla el movimiento del tanque, donde está definido el árbol y en el que se puede apreciar el botón del visualizador. Fuente propia.

### 5.4.3 Prueba con Panda BT Free

Para comenzar a utilizar esta librería es necesario descargarla desde la *Asset Store* de *Unity* e instalarla en el proyecto. Una vez instalado aparecerá su carpeta entre los *assets* del proyecto y ya está lista para utilizarse. Igual que antes, lo primero es añadir una línea al principio del *script*, en este caso: using Panda;

A continuación, se deben redefinir las acciones como objetos de la clase "Task" que retornen un booleano, como en el siguiente ejemplo:

```
[Task]
0 referencias
bool Move()
{
    if (noMoveAgain >= 30)
    {
        Vector3 PlayerPosition = GameObject.Find("Tank(Clone)").transform.position;
        agent.SetDestination(PlayerPosition);
        noMoveAgain = 0;
        return true;
    }
    return false;
}
```

*Ilustración 26.* Ejemplo de acción definida para utilizarla con Panda BT Free. Fuente propia.

Las condiciones también deben definirse como objetos de tipo "Task", pero no deben modificar su valor de retorno.

Por último queda añadir la línea “tree.Tick();” al método “Update()”, al igual que con Fluid Behavior Tree, dado que debido a la naturaleza del juego se desea enviar tics de forma manual con cada fotograma. Por defecto, al añadir un nuevo componente de PandaBehavior a un objeto en *Unity* dicho componente estará configurado para lanzar ticks con cada “Update()” o fotograma. Esto es importante y algo a tener en cuenta dado que si se quiere cambiar a “ticks” manuales o de otro tipo se ha de cambiar dicho parámetro manualmente o mediante código.

Esto completaría la redefinición del script que controla el movimiento del tanque y faltaría definir el árbol en un fichero txt de acorde a como se muestra en la ilustración 13.

Una vez definido el árbol, queda agregar un componente “Panda Behavior” y vincular el fichero de texto que contiene la definición del árbol. Si el árbol está escrito correctamente y las acciones y condiciones programadas con el formato correcto en el *script* de movimiento, el visualizador del árbol debería aparecer como en la ilustración 14. Es necesario determinar en el componente que los ticks son manuales, dado que hemos optado por esta dirección para esta prueba.

De igual manera, con esta librería obtenemos un comportamiento de la IA idéntico al inicial, y podemos visualizar la ejecución del árbol en el componente del *GameObject* del tanque de la IA.

### 5.4.4 Conclusión de las pruebas

Es a través de estas pruebas y la experiencia de primera mano con las librerías que se decide optar por Panda BT Free como la librería a utilizar para la ampliación de la API, dada la simplicidad con la que definen los árboles haciendo uso de esta librería, la poca complejidad que añade a la definición de las acciones y condiciones y la facilidad con la que se adquiere e incorpora a un proyecto en Unity, permitiendo acceso rápido a la obtención de la librería o a actualizaciones que puedan darse.

El factor de soporte para ROS no es algo que importe para el desarrollo de este proyecto, dado que el *Asset* GAIA está orientado al desarrollo de IAs para videojuegos en *Unity* y ROS solo soporta librerías de BTs en C++ o Python, como la de [25].

# 6. Ampliación de la API

En esta sección se describe el trabajo realizado durante el proceso de ampliación del *Asset* GAIA. La ampliación de este API ha consistido en la modificación de diferentes componentes del mismo para incluir la gestión de BTs.

## 6.1 Puesta en escena

Antes de llevar a cabo ninguna modificación sobre el código de la API se decidió hacer uso de su API de FSM. Para ello se decidió construir una versión con FSM de la IA para el juego "Tanks", el banco de pruebas explicado anteriormente. Esta tarea se realiza con varios objetivos en mente. En primer lugar, se busca asegurarse de que el código que formará el punto de partida funciona correctamente. En segundo lugar, se pretende familiarizarse con el funcionamiento de la API y el uso de FSM. Por último, se hace con la finalidad de preparar una prueba que será empleada en las demostraciones posteriores a la ampliación.

### 6.1.1 Extracción de la API

Para poder utilizar la API sobre el proyecto de *Unity* que contiene el juego "Tanks" es necesario generar un módulo o "package" de *Unity* que contenga el código fuente de la API. Para llevar esto a cabo disponemos de una versión no funcional del proyecto de Unity que conforma el juego *ToTheStars* desarrollado por Nicolás Gil Soriano para su TFG [6]. Basta con agrupar en una carpeta los archivos que forman parte de la API y exportar el módulo de *Unity* de la siguiente forma:

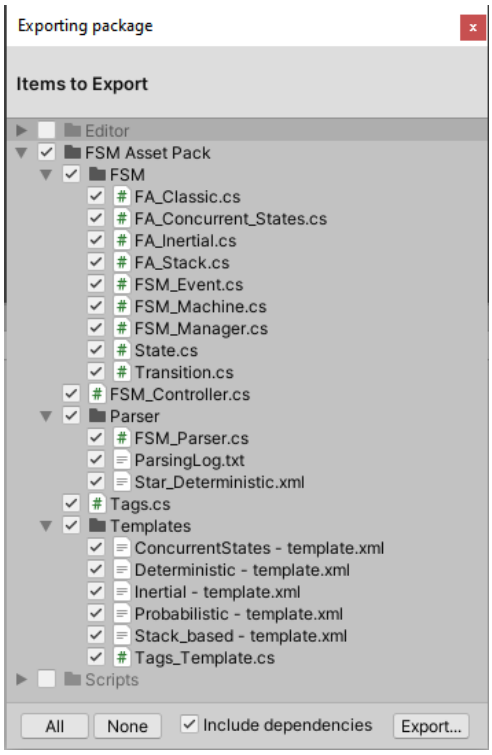


*Ilustración 27.* Exportación del módulo de *Unity* que contiene la API. Fuente Propia.

Una vez exportado el módulo, se procede a importarlo en el proyecto de "Tanks" y la preparación para su uso.

### 6.1.2 Importación y preparación de la API para su uso

Al exportar el módulo hemos agrupado todos los archivos dentro de una misma carpeta. Esto ha causado que la ruta de localización de dichos archivos haya cambiado y por tanto se produzcan algunos errores. Es por tanto necesario modificar la siguiente línea de código en el archivo "FSM_Parser.cs":

```
public void WriteLog(string logPath){
    try{
        file = new System.IO.StreamWriter (logPath);
    }catch(ArgumentException ar){

        logPath = System.IO.Directory.GetCurrentDirectory()+ "/Assets/FSM Asset Pack/Parser/ParsingLog.txt";
        file = new System.IO.StreamWriter (logPath);
    }
```

*Ilustración 28.* Modificación de la ruta del ParsingLog.txt. Fuente propia.

Con este cambio el *parser* puede encontrar de nuevo el directorio donde debe escribir el fichero de *logs*.

Una vez realizado este cambio ya es posible hacer uso de la API.

### 6.1.3 Puesta a punto de la API

En primer lugar, se ha de crear un "GameObject" de *Unity* que contendrá el script "FSM_Controller". Dicho "GameObject" se encargará de inicializar el API a la ejecución del juego, cargando en el *manager* de la API las diferentes FSM descritas en los ficheros XML que se desee. Llegados a este punto, se decide guardar el "GameObject" del controlador como un "prefab", o un objeto de *Unity* prefabricado, que será utilizado en la versión ampliada de la API. Dicho prefab siempre estará dentro de la escena del videojuego y por tanto no será necesario añadir y configurar el controlador manualmente cada vez que se inicie el juego.

El "GameObject" que contiene el controlador de la API queda de la siguiente forma dentro de *Unity*:

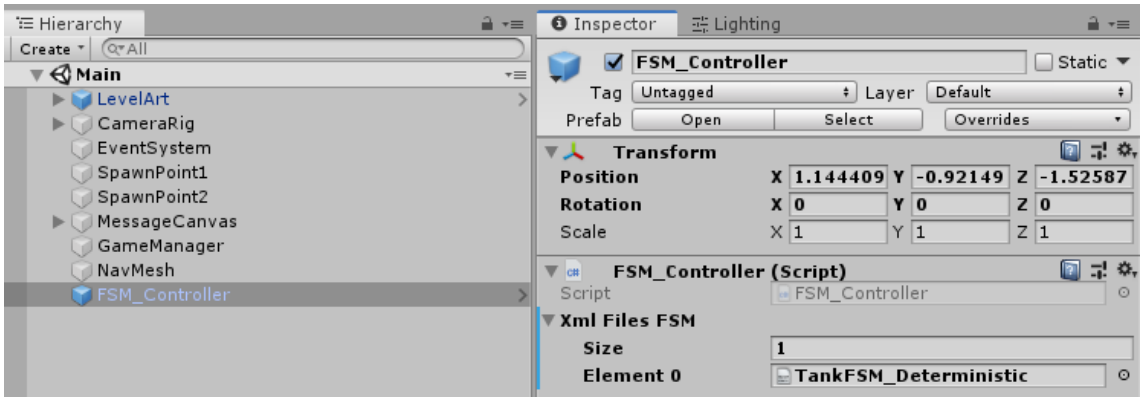


*Ilustración 29*. Controlador de la API dentro de su "GameObject" en *Unity*. Fuente propia.

El script del controlador contiene como parámetro público un array de "TextAssets". Es dentro de este array donde el programador de la IA debe vincular los ficheros que contengan la definición de las FSM. En el caso de la imagen anterior se puede apreciar el fichero que contiene la definición de la FSM empleada para el juego "Tanks" que se describe a continuación.

Una vez el controlador ha sido incluido en el proyecto, se puede proceder a la definición de la FSM.

### 6.1.4 Definición de la FSM para la prueba

Para definir la FSM basta con elegir el tipo de FSM que se desea utilizar y rellenar la plantilla adecuada en XML desarrollada por José Alapont en su TFM. Dicha plantilla viene incluida en el módulo que ha sido generado e importado anteriormente.

Dentro de esta plantilla es necesario determinar si la FSM va a ser de tipo probabilístico o no y darle un nombre:

```
<FSMtype Probabilistic="NO">CLASSIC</FSMtype>
<FSMid>TankAIDeterministic</FSMid>
```

*Ilustración 30*. Definición del tipo de FSM y su nombre en la plantilla XML. Fuente propia.

Es también necesario darle un nombre a la función "callback" de la FSM. Dicha función deberá ser, posteriormente, definida en el script que gestione el comportamiento del tanque y será la encargada de comprobar si se cumplen las condiciones necesarias para generar los eventos que implicarán cambios de estado en la FSM.

```
<Fsm>
  <Callback>BuscarEventos</Callback>
```

*Ilustración 31*. Declaración del nombre de la función "callback" de la FSM. Fuente propia.

A continuación, se definen los diferentes estados de la FSM:

```
<States>
  <State Initial="YES">
    <S_Name>MOVIENDO</S_Name>
    <S_Action>MOVERSE</S_Action>
    <S_inAction>MOVERSE</S_inAction>
    <S_outAction>DISPARAR</S_outAction>
    <S_Fsm></S_Fsm>
  </State>
```

*Ilustración 32*. Declaración de un estado en la plantilla XML. Fuente propia.

Cada estado puede ser inicial o no y dispone de:

- Un nombre identificador
- Una acción que se ejecuta mientras se está en el estado
- Una acción que se debe ejecutar al entrar en el estado.
- Otra acción que se debe ejecutar al salir del estado.
- De manera opcional, El nombre o ID de otra FSM que se desee cargar al entrar en el estado.

Si no se desea que se ejecuten acciones al entrar o salir del estado, basta con definir dicho parámetro como "NULL"

Por último, se deben definir las diferentes transiciones entre los estados, de la siguiente forma:

```
<Transitions>
    <Transition>
    <T_Name>ESPERANDO_A_MOVIENDO</T_Name>
    <T_Origin>ESPERANDO</T_Origin>
    <T_Destination>MOVIENDO</T_Destination>
    <T_Action>NULL</T_Action>
    <Events>
      <Event>
        <ID>ESPERADO</ID>
        <Type>BASIC</Type>
      </Event>
    </Events>
  </Transition>
```

*Ilustración 33*. Definición de una transición entre estados en la plantilla XML. Fuente propia.

Cada transición tiene:

- Un nombre
- Un estado de origen
- Un estado de destino
- Una acción a ejecutar al suceder la transición, que puede ser "NULL"
- Una serie de eventos que activen dicha transición. Los eventos tienen un nombre y pueden ser de tipo básico o apilables, utilizados en el tipo de FSM "Stack Based".

Una vez definidas las transiciones, queda completada la plantilla de la FSM.

A continuación, se debe cumplimentar la clase "Tags" con los *tags* para todos los estados, transiciones, eventos y acciones definidos en la plantilla XML.

El módulo de la API contiene también una plantilla para cumplimentar la clase "Tags" y su funcionamiento es el siguiente:

```
public static class Tags
{
    //FSM type TAGS (DO NOT change or delete)
    public const int CLASSIC            = 0;
    public const int INERTIAL           = 1;
    public const int STACK_BASED        = 2;
    public const int CONCURRENT_STATES  = 3;
    public const int UNKNOWN            = 1000; //UNKNOWN

    //State tags
    public const int MOVIENDO           = 0;
    public const int DISPARANDO         = 1;
    public const int APARTANDO          = 2;

    //Transition tags
    public const int MOVIENDO_A_APUNTANDO    = 0;
    public const int DISPARANDO_A_MOVIENDO   = 1;
    public const int DISPARANDO_A_APUNTANDO  = 2;
    public const int MOVIENDO_A_APARTANDO    = 3;
    public const int DISPARANDO_A_APARTANDO  = 4;
    public const int APARTANDO_A_DISPARANDO  = 5;

    //EVENT TAGS
    public const int DISTANCIA_OK    = 0;
    public const int DISTANCIA_NO_OK = 1;
    public const int DEMASIADO_CERCA = 2;

    //ACTION TAGS
    public const int MOVERSE     = 0;
    public const int DISPARAR    = 1;
    public const int APARTAR     = 2;
```

*Ilustración 34.* Definición de tags para la FSM del juego "Tanks" a partir de la plantilla suministrada en la API. Fuente propia.

Se deben definir constantes de *integers* para cada estado, transición, evento y acción. Estos son los tags. Los tags no deben repetirse para cada uno de los tipos y su valor pude ser cualquiera. Los tags mostrados en la ilustración anterior contienen tanto los tags que han sido añadidos para la prueba como los tags para los tipos de FSM, los cuales ya existen por defecto en la plantilla para el desarrollo de la clase “Tags”.

A continuación, se debe cumplimentar el método “StringToTag”, el cual devuelve el tag asociado a un *string* identificador:

```
public static int StringToTag (string word)
{
        //if(word.Equals("NULL")) return 49;// UnityEngine.Debug.Log("Soy null");
        switch (word[0]) {
        case 'A':
            if (word.Equals("APARTANDO")) return Tags.APARTANDO;
            if (word.Equals("APARTANDO_A_DISPARANDO")) return Tags.APARTANDO_A_DISPARANDO;
            if (word.Equals("APARTAR")) return Tags.APARTAR;
            break;
        case 'B':
            break;
        case 'C':
            if(word.Equals("CLASSIC"))           return Tags.CLASSIC;
            if(word.Equals("CONCURRENT_STATES")) return Tags.CONCURRENT_STATES; //DO NOT DELETE
            break;
```

*Ilustración 35*. Método "StringToTag" cumplimentado para la FSM del juego "Tanks". Fuente propia.

Basta con añadir una línea de código por tag declarado anteriormente.

Llegados a este punto sólo queda crear el *script* que hará uso de la FSM definida. Dado que ya se han realizado definiciones anteriores de IAs para este juego, ya se dispone de un script en el cual se encuentran definidas las acciones que puede ejecutar el tanque. Por tanto, sólo queda modificar dicho *script* para que funcione haciendo uso de la FSM a través de la API.

### 6.1.5 Preparación del *script*

El primer paso para poner en marcha la FSM sobre el tanque es crear la máquina de estados. Para ello se debe declarar en el script una variable que contenga al *manager*, que será creado por el controlador al iniciar el juego, y otra variable que contendrá la máquina. Esto queda de la siguiente forma:

```
private FSM_Machine FSM;                                    // Variable que contiene la FSM.
private FSM_Manager FSMmanager;                             // Variable que contiene la referencia al manager.
```

*Ilustración 36*. Variables de la máquina y del mánager en el script del tanque. Fuente propia.

Dichas variables serán inicializadas en la ejecución del método “Start” que se ejecuta al iniciar el juego:

```
void Start()
{
    FSMmanager = FSM_Controller.INSTANCE.m_managerFSM;
    FSM = FSMmanager.createMachine(this, Tags.CLASSIC, "TankAIDeterministic");
    FSMevents.Add(Tags.UNKNOWN);
}
```

*Ilustración 37*. Método "Start" del script del tanque. Fuente propia.

En el método anterior se instancia el *manager*, que ha sido creado por el controlador, y se utiliza para crear la FSM mediante la llamada a su método "createMachine". Dentro de este método hay que indicar el "GameObject" sobre el cual se instancia la máquina (en este caso "this"), el tipo de máquina y el nombre o id del autómata sobre el que se desea crear la máquina.

En este punto también se agrega a la lista de eventos, declarada al principio del *script*, un valor nulo para indicar que la cola está vacía.

A continuación, se debe agregar en el método "Update", que se ejecuta automáticamente cada fotograma, la llamada al a función "UpdateFSM" de la máquina. Esta función devuelve una lista de *integers* con los *tags* de las acciones que deben ejecutarse. Esta lista debe procesarse y por cada *tag* dentro de la misma realizar una llamada a una función que ejecute la acción indicada. En el caso del script del tanque esto queda así:

```
FSMactions = FSM.UpdateFSM();
for (int i = 0; i < FSMactions.Count; i++)
{
    if (i != Tags.UNKNOWN)
    {
        ExecuteAction(FSMactions[i]);
    }
}
```

*Ilustración 38*. Obtención de la lista de acciones y llamada al método de ejecución de acciones del script del tanque. Fuente propia.

El método "ExecuteActions" ha sido definido en el script de la siguiente manera:

```
public void ExecuteAction(int actionTag)
{
    switch (actionTag)
    {
        case Tags.APARTAR:
            Aim();
            moveBack();
            Shoot();
            break;

        case Tags.DISPARAR:
            Aim();
            Shoot();
            break;

        case Tags.MOVERSE:
            Move();
            break;
    }
}
```

*Ilustración 39*. Método "ExecuteActions" del script del tanque. Fuente propia.

Este método ejecuta varias de las acciones predefinidas (que son las mismas que las empleadas en las otras versiones de la IA del tanque anteriormente expuestas) según el tag recibido de la FSM. Es a través de la ejecución de estas acciones que el tanque expresa el comportamiento deseado por el programador de la IA.

Por último, queda definir el método "BuscarEventos", que es la función "callback" a la cual se le ha dado nombre en la plantilla XML de definición del autómata. Esta función es llamada cada vez que se ejecuta "UpdateFSM", función ejecutada tras cada fotograma como se muestra en la ilustración 38, y su función es comprobar si se dan las condiciones necesarias para que se produzcan eventos. En caso de que así sea, se agregan dentro de la lista de eventos los *tags* de los eventos que han sucedido. Si no suceden eventos se agrega un tag "NULL". Esta lista es la que es devuelta a la lista de acciones y que posteriormente es procesada como se muestra en la ilustración 38.

La función "BuscarEventos" para la IA del tanque se muestra a continuación:

```
public List<int> BuscarEventos()
{
    FSMevents.Clear();

    if (distance < 5)
    {
        FSMevents.Add(Tags.DEMASIADO_CERCA);
    }

    if (distance > 17)
    {
        FSMevents.Add(Tags.DISTANCIA_NO_OK);
    }

    if (distance <= 17 && distance >= 5)
    {
        FSMevents.Add(Tags.DISTANCIA_OK);
    }

    if (FSMevents.Count == 0)
    {
        FSMevents.Add(Tags.UNKNOWN);
    }

    return FSMevents;
}
```

*Ilustración 40.* Función "BuscarEventos" del script de la IA del tanque. Fuente propia.

Estos son todos los cambios que es necesario implementar sobre el script del tanque para la puesta en marcha de la FSM. El comportamiento mostrado por el tanque haciendo uso de esta FSM es prácticamente idéntico al demostrado en las anteriores pruebas.

Una vez finalizada la prueba con éxito y comprobado que la API funciona correctamente, se procede a la ampliación de la misma, partiendo del proyecto sobre el que se ha realizado esta prueba.

## 6.2 Introducción de Panda BT Free al proyecto

A continuación, se exponen los cambios llevados a cabo sobre los diferentes componentes del API de GAIA para hacer uso de la librería de Panda.

El punto de partida para la realización de los cambios que se describen a continuación es la prueba con FSM del apartado anterior.

### 6.2.1 Importación y licencia de Panda BT Free

Para poder proceder con la ampliación de GAIA es necesario importar Panda BT Free al proyecto sobre el cual hemos hecho la prueba descrita en el apartado

anterior. Esto se realiza de la misma manera que se ha explicado antes en el apartado 5.4.3.

Antes de comenzar a describir la ampliación llevada a cabo sobre cada uno de los componentes de la API, es necesario denotar algunas cosas sobre la licencia de Panda BT Free.

Según lo descrito en [15], Panda BT Free tiene una licencia de tipo "Extension Asset". Este tipo de licencias, según lo explicado en el apartado 2.3.2 del Apéndice 1 de los términos y servicios de *Unity* [27], requieren que cada usuario del *asset* adquiera una copia de la licencia, pudiendo ser la misma utilizada por la misma persona en hasta 2 ordenadores concurrentemente.

Debido a esta restricción, será necesario que cada usuario de la versión ampliada del *Asset* de GAIA que quiera hacer uso de las funcionalidades de *Behaviour Trees* adquiera, de manera gratuita, e importe desde la *Asset Store* de *Unity* la librería de Panda BT Free a su proyecto. De esta forma, el módulo generado con la versión ampliada de la API no contará con el *asset* de Panda BT Free, pero sí con las funcionalidades necesarias para hacer uso de dicha librería a través de la API.

Con la finalidad de permitir que el código de la ampliación de la API compile, aunque el *asset* de Panda BT Free no haya sido importado, se hace uso de la sentencia "#define" al principio del código de cada uno de los componentes del API. Dicha sentencia permite compilar de manera condicional diferentes secciones de código, en base a si se ha definido o no la sentencia. Las imágenes siguientes muestras un ejemplo de lo explicado:

```
//Uncoment the line below to enable the use of the BT functions.
//#define PANDA
```

*Ilustración 41*. Sentencia #define para la compilación condicional del código de la ampliación de la API. Dicha sentencia se debe descomentar para hacer uso de las funcionalidades de Panda. Fuente propia.

```
#if (PANDA)
...
#endif
```

*Ilustración 42*. Comprobación de la definición de la sentencia "#define". Si la sentencia no se ha definido, el código de la condición no se compila. Fuente propia.

### 6.2.2 Modificaciones realizadas sobre el controlador

En esta sección se describen las funcionalidades programadas en el componente de la API que se conoce como el controlador. Dicho componente ha sido explicado anteriormente en el apartado 4.1.4.

En primer lugar, se ha renombrado la clase del controlador de "FSM_Controller" a "GAIA_Controller", dado que dicho componente ahora pasa a poseer funciones múltiples y no solo de FSM.

El cambio realizado sobre este componente ha consistido en añadir un segundo array de objetos de tipo "TextAsset", dentro del cual se deberán colocar los archivos que contengan las definiciones de los BTs que queramos cargar en la API.

```
public TextAsset[] m_xmlFilesFSM; // ngs - xml files with the specification of the finite state machines

public TextAsset[] m_xmlFilesBT; // xml files with the specification of the behavior trees
```

*Ilustración 43.* Declaración de los arrays que contienen las definiciones de FSM y BTs en el controlador de la API. Fuente propia.

De igual modo que con el array de ficheros FSM, se agrega, bajo la sentencia de compilación condicional anteriormente explicada, el código necesario para la comprobación del array y para pasarle al mánager el contenido de los diferentes ficheros donde se describen los BTs.

```
#if (PANDA)
        //Loads and parses all xml definitions of BTs
        if (m_xmlFilesBT != null)
        {
            for (int i = 0; i < m_xmlFilesBT.Length; i++)
            {
                m_manager.addBT(m_xmlFilesBT[i].text);
            }
        }
#endif
```

*Ilustración 44.* Código para cargar las definiciones de BTs en la API. Fuente propia.

Desde dentro de Unity, el *prefab* sobre el que hemos cargado el script del controlador queda de la siguiente forma:

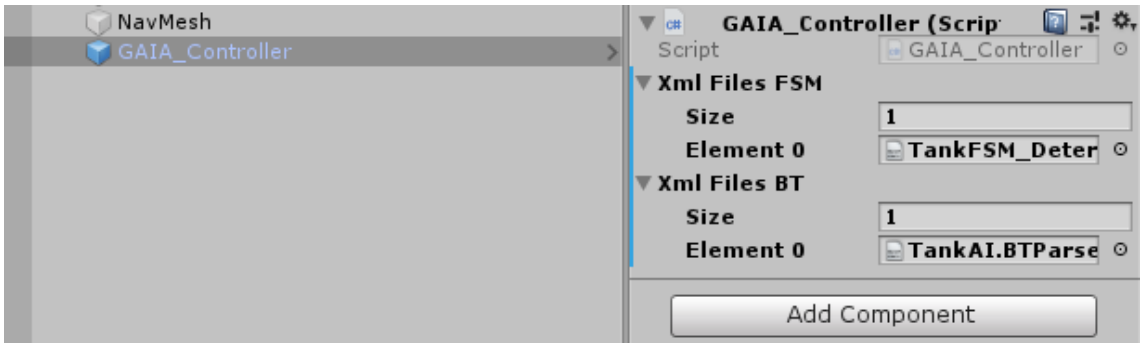


*Ilustración 45*. Vista desde Unity del prefab del controlador de GAIA. Fuente propia.

En la imagen anterior, "size" determina el tamaño del array y los elementos numerados a continuación son los archivos que contiene las definiciones de FSM y BTs respectivamente.

Dichos archivos son pasados al *manager*, cuyos cambios llevados a cabo en la ampliación se describen a continuación.

### 6.2.3 Modificaciones realizadas sobre el *manager*

De igual manera que en el controlador, se hace uso de la sentencia "#define" para la compilación condicional de las secciones de código incluidas en la ampliación de la API. Además, también se ha renombrado la clase del componente de "FSM _Manager" a "GAIA_Manager".

En primer lugar, se ha creado un nuevo diccionario, el cual contendrá los pares clave/valor que consistirán en el ID y el texto de la definición de los BTs. Dicho diccionario es de tipo <string, string> y se inicializa con la creación de un nuevo *manager*.

A continuación, se han creado las funciones para añadir y eliminar BTs de la API. Estas funciones son denominadas "addBT" y "deleteBT".

AddBT es la función invocada por el controlador al cargar cada uno de los archivos presentes en el array de BTs. Dicha función recibe el contenido del archivo de la definición del BT como parámetro y se encarga de invocar una función del componente *parser* (ParseBT) que procesa dicho contenido y devuelve un array de *strings*. Dicho array siempre presenta como su primer componente el identificador del BT cargado y la definición "parseada" del BT como su segundo componente.

Una vez se recibe la respuesta del *parser*, se carga en el diccionario el BT, siendo la clave el identificador y el valor la definición "parseada" del mismo.

```
public int addBT(string content)
{
    string[] parsedbt = new string[1];

    if (this.parser == null)
    {
        return -1;
    }
    else
    {
        if(content != null)
        { //Invoke parser with string
            parsedbt = parser.ParseBT(content);
        }
        if (parsedbt != null)
        {
            try
            {
                BT_dic.Add(parsedbt[0], parsedbt[1]);
                return 1;
            }
            catch (Exception e)
            {
                return -2;
            }
        }
        else return -3;
    }
}
```

*Ilustración 46*. Función "addBT" del *manager* de la API. Fuente propia.

La función "deleteBT" es muy sencilla y únicamente elimina una entrada del diccionario en base al identificador del BT que se desea eliminar.

```
public int deleteBT(string bt_id)
{
    try
    {
        BT_dic.Remove(bt_id);
        return 1;
    }
    catch (Exception e)
    {
        return -1;
    }
}
```

*Ilustración 47*. Función "deleteBT" del *manager* de la API. Fuente propia.

A continuación, se describen las funciones “createBT” y “changeBT”, las cuales deben de ser invocadas en los scripts de los “GameObjects” que quieran hacer uso de un BT a través de la API, así como la función “changeTickOn”.

```
public void createBT(GameObject character, string bt_id)
{
    try
    {
        PandaBehaviour component = character.AddComponent<PandaBehaviour>();
        component.Compile(BT_dic[bt_id]);
    }
    catch (Exception e)
    {
        Debug.Log("EXCEPTION: " + e);
    }
}
```

*Ilustración 48.* Función "createBT" del *manager* de la API. Fuente propia.

La función “createBT” de la ilustración anterior recibe como parámetros el “GameObject” sobre el cual se desea incorporar un comportamiento a través de uno de los BTs definidos y cargados en la API y el identificador de dicho BT.

La función añade un componente de tipo “PandaBehaviour” al “GameObject” e invoca sobre el mismo la función “Compile”. Dicha función es parte de la API de Panda BT Free y permite cargar la definición de un BT y ponerla en marcha. Para ello, accedemos al diccionario de BTs y extraemos dicha definición mediante la clave o identificador del BT obtenida por parámetros.

Al ejecutar la función “createBT” en el script de un “GameObject”, dicho “GameObject” comenzará a ejecutar el BT de manera inmediata y de forma automática, sin necesidad de incluir código para lanzar “ticks” manualmente en el script.

Por defecto, un componente de “PandaBehavior” está configurado para lanzar “ticks” al BT en cada “Update”, o lo que es lo mismo, tras cada fotograma. Si el programador lo desea, esta opción se puede cambiar haciendo uso de una de las siguientes opciones. En primer lugar se ha creado una función, denominada “changeTickOn”, que únicamente cambia este parámetro en el componente de “PandaBehavior”. Esta función se puede utilizar cuando ya haya sido creado el componente “PandaBehavior” en el “GameObject” sobre el que se ejecuta el BT. Si a la hora de crear el BT sobre un “GameObject” se quisiera definir una opción de

lanzamiento de “ticks” diferente, existe una segunda versión de la función “createBT” en la cual se puede especificar como tercer parámetro qué tipo de “ticks” se desea utilizar. La funcionalidad de esta función es por tanto la misma que la de las funciones “createBT” y “changeTickOn” combinadas. El código de las funciones “changeTickOn” y la versión de “createBT” que contempla la definición de en qué momento lanzar “ticks” pueden observarse en el anexo.

La función “changeBT”, es muy similar a “createBT”. En este caso, en vez de añadir un componente de tipo “PandaBehaviour” al “GameObject” pasado por parámetros, se obtiene el componente ya existente y se ejecuta la función de compilar un BT de la misma manera que en “createBT”.

Esta función permite cambiar de un BT a otro en tiempo de ejecución, y por tanto permite diseñar personajes que presentan diferentes comportamientos, pudiendo cambiar entre estos dependiendo de condiciones definidas en el script.

```
public void changeBT(GameObject character, string bt_id)
{
    try
    {
        PandaBehaviour component = character.GetComponent<PandaBehaviour>();
        component.Compile(BT_dic[bt_id]);
    }
    catch (Exception e)
    {
        Debug.Log("EXCEPTION: " + e);
    }
}
```

*Ilustración 49*. Función "changeBT" del *manager* de la API. Fuente propia.

### 6.2.4 Modificaciones realizadas sobre el *parser*

Al igual que en el controlador y el *manager*, la clase del parser ha sido renombrada para reflejar que pasa a poseer funcionalidades tanto de FSM como de BTs, y presenta la sentencia “#define” para compilar condicionalmente las funciones que a continuación se describen.

Dentro del *parser* se han escrito dos nuevas funciones: “ParseBT” y “addChildNodes”.

Además, se han incluido las declaraciones de variables necesarias para "parsear" los archivos de definición de BTs y para reflejar el resultado de la carga de todos los BTs en el archivo de "log".

También se ha llevado a cabo el siguiente cambio, donde tanto la dirección de escritura del archivo de "log" como la inicialización del componente encargado de generar el archivo de "log" en la dirección especificada se inicializan al declarar un nuevo *parser* en vez de al llamar a la función "WriteLog". Esto ha sido la solución encontrada a un problema de compilación surgido como resultado de la ampliación del *parser* con las nuevas funciones para "parsear" BTs.

```
public GAIA_Parser()
{
    LoadedMachines = LoadedSubMachines = LoadedBTs = numErrors = numErrorsBT = 0;
    string logPath = System.IO.Directory.GetCurrentDirectory() + "/Assets/GAIA/FSM Asset Pack/Parser/ParsingLog.txt";
    file = new System.IO.StreamWriter(logPath);
}
```

```
// Writes a parser log in a txt file ...
1 referencia
public void WriteLog(string logPath){
    //try{
        //file = new System.IO.StreamWriter (logPath);
    //}catch(ArgumentException ar){

        //logPath = System.IO.Directory.GetCurrentDirectory()+ "/Assets/GAIA/FSM Asset Pack/Parser/ParsingLog.txt";
        //file = new System.IO.StreamWriter (logPath);
    //}
```

*Ilustración 50.* Cambio realizado sobre la inicialización del *StreamWriter* en el *parser* de la API. Fuente propia.

### 6.2.4.1 La función "ParseBT"

"ParseBT" es una función que se invoca cada vez que se añade una definición de un BT a la API. Esta función es invocada por el *manager* y devuelve un array de strings, donde el primer valor es el identificador del BT y el segundo valor es la definición del BT "parseada" para que Panda BT Free pueda procesarla. El único argumento que recibe esta función es un "string" que contiene la definición del BT realizada por el programador. Dicha definición será un formato XML estandarizado en base a una plantilla proporcionada junto con la adquisición de la API y diseñada como parte de este TFG. Se hablará posteriormente sobre esta plantilla.

```
public string[] ParseBT(string content)
{
    string[] parsedbt = new string[2];
    parsedBTtext = null;

    numErrorsBT = 0;

    xDocBT = new XmlDocument();
    try
    {
        //xDocBT.Load(content);   // It is used to load XML either from a stream, TextReader, path/URL, or XmlReader
        xDocBT.LoadXml(content); //  It is used to load the XML contained within a string.
    }
    catch (FileNotFoundException e)
    {
        numErrorsBT++;
        LogLines += "\r\n>>ERROR. Cannot load file: '" + content + "'.";
    }
```

*Ilustración 51.* Parte 1 de la función "ParseBT" del *parser* de la API. Fuente propia.

Comencemos analizando la función “ParseBT” desde el principio.

En un primer momento, al invocar a la función, se inicializa el array de *string* que será devuelto como resultado de la ejecución de la función. Se inicializa además la variable que contendrá la versión “parseada” de la definición del BT. Además, se inicializa el contador de errores observados durante el “parseo” de un determinado BT a cero.

A continuación, se inicializa un lector de documentos XML, declarado al crear el *parser*, y se carga sobre este el *string* de la definición del BT obtenido por parámetros. En caso de no poder proceder con este paso se modifica el contador de errores y se añade una línea de texto al *string* del archivo de “log” que explica lo sucedido.

```
XmlNodeList Trees = null, treesList = null;
bool startParseFlagBT = true;

//Extract the trees and put them into a List "Trees"
Trees = xDocBT.GetElementsByTagName("Trees");
if (Trees != null)
{
    if (Trees.Count != 0)
        treesList = ((XmlElement)Trees[0]).GetElementsByTagName("Tree");
    else
    {
        numErrors++;
        LogLines += "\r\n>>ERROR. No <Tree> tag in BT with contents '" + content + "'.";
        startParseFlagBT = false;
    }
}
else
{
    numErrors++;
    LogLines += "\r\n>>ERROR. There must be a 'Trees' tag in BT with contents '" + content + "'.";
    startParseFlagBT = false;
}
```

*Ilustración 52.* Parte 2 de la función "ParseBT" del *parser* de la API. Fuente propia.

Esta segunda imagen presenta el código a continuación de la primera.

En esta imagen se aprecia la inicialización de dos listas de nodos XML: una para cargar el nodo “Trees”, utilizado en la plantilla para determinar la sección de definición de árboles, y otra para cargar individualmente cada uno de los árboles definidos en el BT. Ambas listas se inicializan vacías y se emplean a continuación.

Tras declarar las listas, se carga la lista “Trees” a través del lector de XML con los nodos de la sección del archivo XML donde se declaran los diferentes árboles del BT. Solo hay un nodo “Trees” en la plantilla, y de este mismo cuelgan individualmente cada uno de los árboles definidos bajo el “tag” “Tree” como se muestra en la siguiente imagen de la plantilla:

```
<Trees>
  <Tree Root="YES"><!--ROOT ATTRIBUTE DETERMINES WHETHER THE TREE IS THE ROOT OF THE BT OR NOT-->
    <Name><!--NAME OF THE TREE HERE--></Name>
    <Child_Nodes>
```

*Ilustración 53.* Sección de la plantilla XML de definición de BTs. Fuente propia.

Tras cargar la lista “Trees”, esta misma se recorre, si ha sido declarada en la plantilla y hay al menos un árbol definido, añadiendo a la lista “treesList” cada uno de los árboles declarados.

La imagen siguiente muestra la continuación del código de la función “ParseBT”

```
string BTid = "";
try
{
    //Extract the id of the BT
    if (xDocBT.GetElementsByTagName("BTid").Item(0).InnerText.Trim().Length != 0)
    {
        BTid = xDocBT.GetElementsByTagName("BTid").Item(0).InnerText;
    }
    else
    {
        LogLines += "\r\n>>ERROR. <BTid> tag in BT with contents '" + content + "' must be named.";
        BTid = "";
        numErrors++;
        startParseFlagBT = false;
    }
}
catch (Exception e)
{
    LogLines += "\r\n>>ERROR. No <BTid> tag in BT with contents '" + content + "'.";
    BTid = "";
    numErrors++;
    startParseFlagBT = false;
}
```

*Ilustración 54.* Parte 3 de la función "ParseBT" del *parser* de la API. Fuente propia.

En la imagen anterior se declara e inicializa un string que contendrá el identificador de la definición del BT. Para extraer dicho identificador de la plantilla XML se hace uso de lector XML y se busca el "tag" "BTid". Si este se encuentra, el string pasará a contener el valor del identificador. En caso contrario se darán una serie de errores de los cuales el programador será informado a través del archivo de "log".

En una de las ilustraciones anteriores se había declarado un booleano e inicializado este mismo a "True". Dicho booleano es utilizado para determinar si, llegados a este punto en el código, se procede con el "parseo" o no en base a si ha habido errores.

```
if (startParseFlagBT)
{
    LogLines += "\r\n \r\n================================================================================================";
    LogLines += "\r\n...LOADING A BT named '" + BTid + "'";

    //Analize trees
    for (int i = 0; i < treesList.Count; i++) {
        if(treesList.Item(i).Attributes.Item(0).Value == "YES")
        {
            parsedBTtext += "tree(" + '\u0022' + "Root" + '\u0022' + ")\r\n";
            treesAdded++;
        } else
        {
            try {
                if(treesList.Item(i).FirstChild.InnerText.Trim().Length != 0) {
                    parsedBTtext += "tree(" + '\u0022' + treesList.Item(i).FirstChild.InnerText + '\u0022' + ")\r\n";
                    treesAdded++;
                }
                else {
                    LogLines += "\r\n>>ERROR. COULD NOT ADD TREE " + treesAdded + ". NAME FIELD IS EMPTY";
                    numErrorsBT++;
                }

            }
            catch (Exception e)
            {
                LogLines += "\r\n>>ERROR. COULD NOT ADD TREE " + '\u0022' + treesList.Item(i).FirstChild.InnerText + '\u0022' + ". THERE IS A PROBLEM WITH ITS NAME";
                numErrorsBT++;
            }
        }
```

*Ilustración 55.* Parte 4 de la función "ParseBT" del *parser* de la API. Fuente propia.

Si no ha habido errores, se escriben una serie de líneas en el fichero de "log" indicando que ha comenzado el "parseo" de un BT y se procede a "parsear" los diferentes árboles de dicho BT. Para ello, se recorre la lista de nodos XML "treesList", inicializada anteriormente, y se determina si para cada árbol definido en el BT el atributo asociado es "YES". Dicho atributo indica si el árbol es la raíz del BT (el punto por el que entran los "ticks"). De ser así, se añade una línea de texto al archivo que representa la definición "parseada" del BT (la variable "parsedBTText"). Dicha línea de texto es un simple "string" que Panda BT Free reconoce y a través del cual se determina el nodo raíz del BT. En caso de no ser el nodo raíz, si el atributo "Root" no es igual a "YES", se procede a adquirir el nombre del árbol accediendo al primer hijo del nodo que representa el inicio del árbol y leyendo el texto del interior del nodo, siempre que este texto no esté vacío. En caso de estar vacío el nombre y no ser un árbol raíz se genera un error y se aumenta el contador de errores y escribe una línea en el fichero de "log". En caso de haber añadido cualquier árbol, sea raíz o no, se aumenta el contador de árboles añadidos.

```
if(treesList.Item(i).ChildNodes.Count > 1)
{
    try {
        addChildNodes(treesList.Item(i).ChildNodes.Item(1), 1);
    }
    catch (Exception e)
    {
        LogLines += "\r\n>>ERROR. COULD NOT ADD TREE " + '\u0022' + treesList.Item(i).FirstChild.InnerText + '\u0022' + " CHILD NODES";
        numErrorsBT++;
    }
}
LogLines += "\r\n>>ADDED TREE " + treesList.Item(i).FirstChild.InnerText + " AND ITS CHILDS.";
```

*Ilustración 56.* Parte 5 de la función "ParseBT" del *parser* de la API. Fuente propia.

La ilustración anterior es la continuación del código de la función "ParseBT" y en ella se muestra una condición que comprueba si, para cada árbol del BT, existe más de un nodo hijo. En caso de ser cierta dicha condición, esto implica que el árbol tiene nodos hijos, los cuales deben ser "parseados" de manera correcta para representar la estructura del árbol. Para ello se llama a la función "addChildNodes", encargada de "parsear" de manera recursiva todos los nodos hijo de cada árbol del BT y de la cual se hablará más adelante. En caso de no presentar más que un solo nodo hijo en el árbol sucede un error y se añade una línea al archivo de "log".

La última línea de código de la imagen añade al archivo de "log" una línea de texto en la que se determina que el árbol y sus hijos han sido "parseados".

```
}
LogLines += "\r\nTrees Added: " + treesAdded + " out of " + treesList.Count;

if(numErrorsBT == 0)
{
    LogLines += "\r\n\r\n(Parsing of BT named " + '\u0022' + BTid + '\u0022' + " is OK!)";
}
else
{
    LogLines += "\r\n\r\n(Parsing of BT named " + '\u0022' + BTid + '\u0022' + " is NOT OK!)";
}
LoadedBTs++;
```

*Ilustración 57*. Parte 6 de la función "ParseBT" del *parser* de la API. Fuente propia.

Una vez recorridos todos los árboles del BT se añade una línea al archivo de "log" que indica cuantos árboles del BT se han añadido correctamente. A continuación, según si ha habido errores durante el "parseo" o no se añade otra línea de texto al archivo de "log" y se aumenta el contador de BTs cargados.

```
else
{
    LogLines += "\r\n >>ERROR. Cannot PARSE xml file.";
    numErrorsBT++;
}

LogLines += "\r\n \r\nCritical errors parsing BTs: " + numErrorsBT + "\r\n";

file.WriteLine (LogLines);
LogLines = "";
treesAdded = 0;

parsedbt[0] = BTid;
parsedbt[1] = parsedBTtext;

return parsedbt;
```

*Ilustración 58*. Parte 7 de la función "ParseBT" del *parser* de la API. Fuente propia.

En caso de haber habido errores durante la fase inicial de extracción de los árboles y del ID y por tanto haber estado el booleano a "false" se añade una línea al "log" indicando lo sucedido y se aumenta el contador de errores.

Por último, se añade otra línea al archivo de "log" indicando el número total de errores sucedidos durante el "parseo" del BT, se indica al componente encargado de escribir el archivo de "log" que agregue las líneas al archivo, se limpia la variable encargada de recoger las líneas a escribir en el archivo de log y el contador de árboles añadidos, se carga el array de strings creado inicialmente con las variables que representan el identificador del BT y el texto "parseado" del BT y se devuelve dicho array al manager.

### 6.2.4.2 La función “addChildNodes”

Como se ha comentado en la sección anterior, la función “addChildNodes” se encarga de “parsear” todos los nodos hijos de cada árbol del BT. Esta función recibe como parámetros el nodo a “parsear” y un *int* que representa el número de tabulaciones a añadir en el fichero de “parseo”

```
public void addChildNodes(XmlNode CN, int tabCount)
{
    string tab = "\t";
    string tabCollector = null;

    for (int i = 0; i < CN.ChildNodes.Count; i++)
    {
        for (int j = 0; j < tabCount; j++)
        {
            tabCollector += tab;
        }
        if(CN.ChildNodes.Item(i).ChildNodes.Item(0).InnerText == "tree")
        {
            try
            {
                parsedBTtext += tabCollector + "tree(" + '\u0022' + CN.ChildNodes.Item(i).ChildNodes.Item(1).InnerText + '\u0022' + ")\r\n";
            }
            catch (Exception e)
            {
                LogLines += "\r\n>>ERROR. CHILD NODE " + '\u0022' + CN.ChildNodes.Item(i).ChildNodes.Item(1).InnerText + '\u0022' + " COULD NOT BE ADDED.";
                numErrorsBT++;
            }
```

*Ilustración 59*. Parte 1 de la función "addChildNodes" del *parser* de la API. Fuente propia.

Al llamar a la función “addChildNodes” se inicializan dos strings: Uno que representa una tabulación en formato de texto y otro cuya finalidad es almacenar una cantidad de tabulaciones en formato de texto, de acorde a lo indicado en el parámetro “tabCount”. Las tabulaciones son necesarias para que Panda BT Free entienda la estructura del árbol, ya que los nodos hijo han de ser representados con una tabulación más frente a su padre.

A continuación, para cada uno de los hijos del nodo que representa la colección de nodos hijo del árbol, se procede a añadir al *string* “tabColector” la cantidad adecuada de tabulaciones y se comprueba si dicho nodo hijo es de tipo “tree”. De ser así, se añade una línea a la variable que recoge el BT “parseado” indicando que el hijo es un árbol y el nombre de dicho árbol. Dicha línea vendrá precedida por la cantidad de tabulaciones anteriormente estipulada.

En caso de que el hijo no sea un árbol se procede con lo mostrado en la siguiente imagen:

```
{
    if(CN.ChildNodes.Item(i).ChildNodes.Item(0).InnerText.Trim().Length != 0) {
        try
        {
            if (CN.ChildNodes.Item(i).ChildNodes.Item(1).InnerText.Trim().Length != 0) {
                parsedBTtext += tabCollector + CN.ChildNodes.Item(i).ChildNodes.Item(1).InnerText + "\r\n";
            }
            else
            {
                LogLines += "\r\n>>ERROR. CHILD NODE FROM THE TREE Nº " + treesAdded + " HAS ITS CONTENT EMPTY. PLEASE DEFINE THE CONTENTS OF THE NODE";
                numErrorsBT++;
            }
        }
        catch (Exception e)
        {
            LogLines += "\r\n>>ERROR. CHILD NODE " + '\u0022' + CN.ChildNodes.Item(i).ChildNodes.Item(1).InnerText + '\u0022' + " COULD NOT BE ADDED.";
            numErrorsBT++;
        }
    }
    else
    {
        LogLines += "\r\n>>ERROR. CHILD NODE FROM THE TREE Nº " + treesAdded + " HAS NO SPECIFIED TYPE. PLEASE SPECIFY EITHER A TREE OR A NODE";
        numErrorsBT++;
    }
}
```

*Ilustración 60*. Parte 2 de la función "addChildNodes" del *parser* de la API. Fuente propia.

En este caso, se comprueba que el contenido del nodo que indica el tipo del mismo no está vacío y de ser así se comprueba si el "nombre" del nodo o el contenido del mismo está vacío. Dicho "nombre" no debe ser un identificador, sino el tipo de nodo (secuencia, fallback, etc.) o el nombre de una condición o acción a ejecutar.

En caso de este "nombre" no estar vacío se añadirá a la variable de "parseo" una línea con dicho nombre, precedida por la cantidad de tabulaciones anteriormente preparada.

Si el tipo o el "nombre" del nodo estuvieran vacíos saltarían errores, añadiendo una línea al fichero de "log" y aumentando el contador de errores sucedidos.

```
    if(CN.ChildNodes.Item(i).ChildNodes.Count > 2)
    {
        try
        {
            if (CN.ChildNodes.Item(i).ChildNodes.Item(2).ChildNodes.Count > 0 && CN.ChildNodes.Item(i).ChildNodes.Item(2).Name == "Child_Nodes") {
                addChildNodes(CN.ChildNodes.Item(i).ChildNodes.Item(2), tabCount + 1);
            }
            else
            {
                LogLines += "\r\n>>ERROR. COULD NOT ADD SUBCHILDS OF THE TREE Nº " + treesAdded + ". PLEASE VERIFY THE DEFINITION OF THIS TREE IS CORRECT";
                numErrorsBT++;
            }

        }
        catch (Exception e)
        {
            LogLines += "\r\n>>ERROR. COULD NOT ADD NODE " + '\u0022' + CN.ChildNodes.Item(i).ChildNodes.Item(1).InnerText + '\u0022' + " CHILD NODES";
            numErrorsBT++;
        }
    }
    tabCollector = null;
}
parsedBTtext += "\r\n";
```

*Ilustración 61*. Parte 3 de la función "addChildNodes" del *parser* de la API. Fuente propia.

El código de la imagen anterior es la continuación de la función "addChildNodes" y en el se muestra una comprobación que indica si el nodo hijo que ha sido "parseado" presenta a su vez nodos hijo. De ser así, se comprueba que el nodo que recoge los nodos hijo no está vacío y que tiene el nombre esperado. Es caso afirmativo

se llama de nuevo a la función “addChildNodes”, pasando como parámetros el nodo que recoge los nodos hijo del hijo “parseado” y el contador de tabulaciones incrementado en 1.

En caso de que haya habido algún error se añadirá como siempre una línea al archivo de “log” indicando lo sucedido y se aumenta el contador de errores.

Por último, al finalizar el “parseo” de cada nodo hijo se vacía el parámetro que recoge las tabulaciones.

Al finalizar el “parseo” de todos los nodos hijo se añade una línea a la variable del “parseo” que representa un salto de línea. Dicho salto de línea indica que se ha llegado al final del árbol y que todos sus hijos han sido “parseados”. Con este salto de línea se separa además un árbol de los demás, de modo que Panda BT Free entienda los diferentes árboles, donde empiezan y donde acaban.

### 6.2.4.3 Fichero de “log”

Se presenta a continuación en esta sección una imagen del fichero de “log” generado tras haber “parseado” un BT.

```
================================================================================================
...LOADING A BT named 'BTDoom'
>>ADDED TREE Root AND ITS CHILDS.
>>ADDED TREE Movement AND ITS CHILDS.
>>ADDED TREE Shooting AND ITS CHILDS.
Trees Added: 3 out of 3

(Parsing of BT named "BTDoom" is OK!)

Critical errors parsing BTs: 0

LOG FILE    27/01/2021 4:53:43
------------------------------------

--------SUMMARY--------
Number of main machines: 1
Number of submachines: 0
TOTAL MACHINES ADDED: 1
NUMBER OF BTs ADDED: 2
```

*Ilustración 62*. Parte de un fichero de "log" generado por la API tras haber "parseado" una FSM y dos BTs. Fuente propia.

Este es un fragmento del fichero de “log” y en él se pueden apreciar las líneas de texto que informa al programador de que ha comenzado a “parsearse” el BT, las líneas que indican que se han “parseado” los diferentes árboles del BT, el resumen de

árboles “parseados” así como el resultado final del “parseo” y el número de errores encontrados.

Al final del documento se incluye la información sobre el número de FSM, sub-FSM y BTs añadidos a la API. Este último fragmento del fichero de log se escribe al ejecutar la función “WriteLog” del *parser* por parte del controlador, y sobre esta función se ha añadido la línea que indica el número de BTs añadidas a la API. El fichero de “log” se genera al ejecutar también esta función.

```
public void WriteLog(string logPath){
    //try{
        //file = new System.IO.StreamWriter (logPath);
    //}catch(ArgumentException ar){

        //logPath = System.IO.Directory.GetCurrentDirectory()+ "/Assets/GAIA/FSM Asset Pack/Parser/ParsingLog.txt";
        //file = new System.IO.StreamWriter (logPath);
    //}


    LogLines +="\r\n\r\n--------SUMMARY--------";
    LogLines +="\r\nNumber of main machines: "+(LoadedMachines-LoadedSubMachines);
    LogLines +="\r\nNumber of submachines: "+LoadedSubMachines;
    LogLines +="\r\nTOTAL MACHINES ADDED: "+LoadedMachines;
    LogLines += "\r\nNUMBER OF BTs ADDED: " + LoadedBTs;

    file.WriteLine("LOG FILE\t" + System.DateTime.Now + "\r\n---------------------------------------"+LogLines);
    //Closing log file
    file.Close ();
}
```

*Ilustración 63.* Función "WriteLog" del *parser* de la API. Fuente propia.

### 6.2.4.4 Plantilla XML para la definición de BTs

Se presenta a continuación una imagen de una parte de la plantilla XML diseñada para que los programadores puedan definir sus BTs de manera que el parser pueda entender y procesar la información de manera correcta.

```xml
<?xml version="1.0" encoding="utf-8" ?>

<BT>
  <BTid> <!--ID OF THE BT HERE--></BTid>
  <Bt>
    <Trees>
      <Tree Root="YES"><!--ROOT ATTRIBUTE DETERMINES WHETHER THE TREE IS THE ROOT OF THE BT OR NOT-->
        <Name><!--NAME OF THE TREE HERE--></Name>
        <Child_Nodes>
          <CN>
            <CN_Type><!--EITHER TREE OR NODE--></CN_Type>
            <CN_Name><!--CONTENTS MAY BE EITHER THE TYPE OF NODE OR THE NAME OF A CONDITION OR ACTION--></CN_Name>
            <!--SUBCHILDS ARE DEFINED IN THE ELEMENT BELOW, JUST LIKE CHILD NODES-->
            <Child_Nodes>
              <CN>
                <CN_Type>tree</CN_Type>
                <CN_Name>Movement</CN_Name>
              </CN>
              <CN>
                <CN_Type>tree</CN_Type>
                <CN_Name>Aiming</CN_Name>
              </CN>
            </Child_Nodes>
          </CN>
        </Child_Nodes>
      </Tree>
```

*Ilustración 64.* Parte 1 de la plantilla XML diseñada para programar BTs haciendo uso de la API. Fuente propia.

En la plantilla podemos observar el nodo "<BTid>", que contiene el identificador del BT, y el nodo "<Trees>", que contiene los diferentes árboles del BT.

Cada árbol está representado por un nodo "<Tree>". Dicho nodo contiene el atributo que indica si es la raíz del BT. Este nodo posee además posee al menos dos hijos: un nodo hijo que contiene el nombre del árbol y otro nodo hijo que contiene los nodos hijos del árbol.

Cada nodo hijo de un árbol tiene su identificador de tipo, pudiendo ser este tipo "tree" o "node", y a continuación un nodo en el cual se debe determinar o el nombre del árbol si se trata de un árbol o el contenido del nodo si se trata de otra cosa. Dicho contenido deberá ser el tipo de nodo (secuencia, fallback, etc.) o una acción o condición (representada por el nombre de la función a ejecutar).

Si un hijo tiene a su vez nodos hijo deberán agregar a partir del nodo "<Child_Nodes>". La profundidad del árbol puede ser infinita y el formato a seguir para declarar nodos hijo siempre es el mismo (tipo del nodo hijo, nombre o contenido y nodos hijo (si los tiene)).

```
<Tree Root="NO">
  <Name>Movement</Name>
  <Child_Nodes>
    <CN>
      <CN_Type>node</CN_Type>
      <CN_Name>fallback</CN_Name>
      <Child_Nodes>
        <CN>
          <CN_Type>node</CN_Type>
          <CN_Name>sequence</CN_Name>
          <!--SUBCHILDS CAN ALSO HAVE INFINITE CHILDS-->
          <Child_Nodes>
            <CN>
              <CN_Type>node</CN_Type>
              <CN_Name>tooFar</CN_Name>
            </CN>
            <CN>
              <CN_Type>node</CN_Type>
              <CN_Name>Move</CN_Name>
            </CN>
          </Child_Nodes>
        </CN>
```

*Ilustración 65.* Parte 2 de la plantilla XML diseñada para programar BTs haciendo uso de la API. Fuente propia.

En la imagen anterior se muestra otra parte de la plantilla, donde se define otro árbol del BT como ejemplo. En este caso el parámetro "Root" es negativo y los nodos hijo son de tipo "node", presentando ejemplos del contenido que se debe escribir en el nodo "<CN_Name>" para cada hijo.

## 6.3 Creación del *Asset* y subida a la *Asset Store* de *Unity*

En esta sección se describen los pasos seguidos para generar el *asset* de la API ampliada y subirlo a la *Asset Store* de *Unity*.

Una vez completado el proceso de ampliación de la API y tras realizar y comprobar su correcto funcionamiento a través de pruebas posteriores que se explican más adelante, se decide comenzar con el proceso de intentar subir el *Asset* de GAIA a la *Asset Store* de *Unity*. Para ello, el primer paso ha sido informarse de los requisitos y herramientas necesarias para la creación y subida del *asset*. Esto se explica en [28].

Tras registrarse como publicador de la *Asset Store* de *Unity*, es necesario descargar de manera gratuita la herramienta "Asset Store Tools" de *Unity* e importarla

en el proyecto de *Unity* que contiene la versión ampliada completa de la API. Esta herramienta permite seleccionar los archivos deseados y, una vez configurado el perfil del *asset* en el "Package Manager" de la cuenta de publicador en la *Asset Store* de *Unity*, subirlos como parte del paquete previamente configurado a la web para iniciar el proceso de revisión del *asset*.

Una vez los archivos del *asset* han sido subidos a la web y se han configurado el perfil y la página del *asset*, se puede presentar el paquete para revisión por parte del equipo de *Unity* para su aprobación. El perfil del *asset* generado tras este proceso y de manera previa a su aprobación se puede ver en el siguiente enlace: [29]

Tras el proceso de revisión del *asset* por parte del equipo de *Unity*, desafortunadamente este fue rechazado por no seguir algunos aspectos de los términos para la admisión de nuevos *assets.* Sin dejar clara la razón concreta, se pueden especular una multitud de causas por lo que esto ha sucedido.

En primer lugar, el API no cuenta con ningún tipo de documentación ni guía de uso. Esto es un plus muy importante para que el *asset* pueda ser entendido y utilizado por los usuarios, además de uno de los requisitos para superar el proceso de revisión. Por otro lado, no se ha acompañado el perfil del *asset* con material gráfico, ya sea capturas de pantalla, videos o audios explicativos o de alguna demostración de su funcionamiento. Es debido a estas carencias, además de otras razones como la falta de pruebas del funcionamiento de la API en diferentes versiones de *Unity*, así como en diferentes plataformas, que se cree que la petición de subida del *asset* a la *Asset Store* ha sido denegada.

Para buscar que una nueva petición sea aceptada, sería necesario llevar a cabo una serie de trabajos, de los cuales se hablará más adelante en la sección de trabajos futuros.

# 7. Pruebas posteriores

Una vez realizada la ampliación del *Asset* de GAIA se ha procedido a componer una serie de pruebas para comprobar su correcto funcionamiento, así como para servir de demostración en la presentación de este TFG.

Para ello, se han tenido que llevar a cabo una serie de modificaciones en la programación de varios componentes del entorno de pruebas "Tanks", de forma que este pueda soportar más de dos jugadores y que se puedan hacer aparecer NPCs con controladores diferentes. De esta forma, en las pruebas finales, el jugador se enfrenta a varios tanques controlados por IAs generadas mediante el *Asset* de GAIA. En una de las pruebas, uno de los tanques funciona haciendo uso de FSM, de igual forma que se ha descrito antes en la prueba con FSM del apartado 6.1, mientras que el otro tanque hace uso de las funcionalidades de BTs añadidas en la ampliación realizada en este TFG y su comportamiento es idéntico al del tanque que hace uso de FSM. Esto es hasta que pasan 10 segundos desde que se lanza a ejecución el juego. Tras estos 10 segundos se compila un BT diferente y este tanque pasa a presentar un comportamiento diferente (trata de acercarse al jugador y cuando llega a cierta distancia comienza a rotar sobre sí mismo rápidamente y a disparar sin parar). En las otras pruebas se han modificado los BTs y se han añadido acciones al script de los tanques controlados por las IAs para dar un "espectáculo" al tribunal. El objetivo de estas vistosas muestras es demostrar con que facilidad se pueden añadir y quitar elementos para construir o modificar comportamientos de IAs haciendo uso de la API, y en particular de las nuevas funcionalidades de BTs añadidas como resultado de la ampliación llevada a cabo.

## 7.1 Preparación de la prueba con BT

Se describe a continuación en este apartado la forma en la que se ha preparado el script del tanque que hace uso del Asset de GAIA para obtener un comportamiento a través del uso de BT. El objetivo de este apartado es ilustrar la sencillez del uso de la API a la hora de programar IAs con tecnología de BTs haciendo uso de la ampliación que se ha llevado a cabo en este TFG.

El punto de partida de este script es el mismo que el alcanzado tras la prueba con Panda BT Free descrita en el apartado 5.3.3, donde las acciones y condiciones han sido definidas como objetos de clase [Task] (requerido para que sean reconocidas por

Panda). Sin embargo, esta vez no es necesario configurar manualmente el "prefab" del tanque que va a hacer uso de IA mediante BT. Dado que va a ser el API el que incorpore el componente de Panda al "prefab" en tiempo de ejecución en base a lo programado en el script del tanque, podemos eliminar dicho componente del "prefab".

Para poder utilizar el API y la funcionalidad e BTs basta con añadir "using Panda" y "using GAIA" al principio del script.

Para que el tanque presente el comportamiento que deseamos haciendo uso de la API simplemente debemos, al igual que si usamos FSM, instanciar el *manager*, y a continuación llamar a la función "createBT" sobre ese manager, indicando el identificador del BT que queremos utilizar. La imagen siguiente muestra las líneas de código descritas:

```
manager = GAIA_Controller.INSTANCE.m_manager;
#if (PANDA)
manager.createBT(this.gameObject, BTFileName);
#endif
```

*Ilustración 66.* Creación del script del tanque que hace uso de BTs a través de la API. Fuente propia.

Ahora slo es necesario cargar el controlador con las definiciones de FSM y BTs en sus respectivos arrays. No es necesario añadir ninguna línea más de código. Desde el momento en que se ejecute "createBT" y el "GameObject" sea habilitado el BT comenzará a ejecutarse automáticamente, siempre que este haya sido definido correctamente, así como que las condiciones y acciones estén implementadas de manera correcta.

Como se ha mencionado anteriormente, se han preparado diferentes pruebas donde para cada una de ellas se han empleado definiciones diferentes de BTs, además de haber añadido varias acciones y realizado ciertas modificaciones sobre los scripts de los tanques, así como sobre el funcionamiento del entorno de pruebas.

Se hablará más en detalle de los cambios llevados a cabo y del esfuerzo dedicado a las pruebas durante su presentación en forma de vídeo ante el tribunal.

# 8. Trabajos futuros

En esta sección se describen posibles ampliaciones a realizar sobre la API para aumentar sus funcionalidades.

## 8.1 Ampliación a nuevas tecnologías

De la misma manera que se ha ampliado el API para hacer uso de la tecnología de BTs, es posible incorporar otras tecnologías para el desarrollo de IAs en *Unity*. Quizás la opción más interesante sería llevar a cabo una ampliación a redes neuronales, una tecnología de "Deep learning" [40].

Si bien las redes neuronales artificiales tienen sus inicios en los años 40 del siglo XX (Warren McCulloch and Walter Pitts [41]) no fue hasta la aparición del algoritmo de Backpropagation (Werbos 1975 [42]) que se hizo práctico entrenar capas ocultas. Con los desarrollos de Schmidhuber [43], Hinton [40] y muchos otros junto al avance en el procesamiento de cálculo (ley de Moore, aparición de GPUs,...) que el "Deep learning" ha experimentado el boom que conocemos en los últimos años. Hoy en día, esta tecnología tiene multitud de aplicaciones modernas, como, por ejemplo, en reconocimiento de voz o de imágenes. [30]

## 8.2 Incorporación de un editor gráfico

Una de las mayores barreras para el desarrollo de IAs con BTs es la abstracción de la estructura que finalmente presentará el árbol. Debido a las complejas redes de transiciones, condiciones y tareas, a veces es fácil perderse mientras se diseña un BT. Los editores gráficos facilitan esta tarea puesto que los BTs tienen una representación visual que permite realizar un diseño y analizarlo de una manera que no sería igual en modo texto. Diseñarlo visualmente en otro entorno para convertirlo a texto sería una barrera innecesaria que puede ser eliminada con la incorporación de un editor visual al API de GAIA.

Mientras que Panda BT Free ya cuenta con un visualizador para la ejecución del árbol, este no ayuda a la hora de diseñar los BTs, especialmente al tener que hacer uso de plantillas XML. Dicho visualizador solo es accesible una vez se ha cargado la definición del BT en el API, y, por tanto, mientras que este puede utilizarse para

comprobar que el árbol presenta la estructura que se había ideado, no ayuda durante la fase de diseño.

En esta dirección, sería interesante ampliar el *parser* de la API con un entorno gráfico que permita generar plantillas XML reconocibles por el *parser* a partir del esquema diseñado por el programador. Idealmente sería algo tan sencillo como arrastrar elementos sobre un lienzo, enlazar dichos elementos y poder dar nombres a los diferentes elementos colocados.

## 8.3 Documentación y guía de uso

Mientras que este trabajo de fin de grado explica el funcionamiento de la API, no hace sus veces ni de documentación del código ni de guía para usuarios. Es por ello por lo que, si se quiere subir el *Asset* de GAIA a la *Asset Store* de *Unity*, es necesario llevar a cabo dicho trabajo.

## 8.4 Optimización para ejecución de IAs sobre tarjetas gráficas

Debido al elevado coste de CPU de la ejecución de algunas IAs, algo que depende tanto de su diseño y programación como del entorno donde se ejecutan, es posible optimizar el rendimiento del sistema si se reduce la carga de la CPU y se deriva esta misma sobre la tarjeta gráfica.

Aunque esta técnica es muy dependiente de dónde se pueda producir más comúnmente un cuello de botella, es interesante facilitar al programador la posibilidad de decidir sobre que componente se ejecuten las IAs, con el objetivo de ofrecer fácil acceso a técnicas de optimización de sus proyectos.

Unity ofrece algunas opciones para el control de la ejecución sobre GPU y CPU, y sería interesante explorar alguna manera de incorporar dicha funcionalidad.

# 9. Viabilidad económica

En este apartado se lleva a cabo un breve análisis sobre el coste estimado del desarrollo de la ampliación de la API llevada a cabo en este trabajo de fin de grado. En base a los datos presentados, se trata de determinar si sería interesante para una empresa generar herramientas como esta API de manera interna o hacer uso de herramientas de terceras partes.

Según la web *PayScale* [31], un portal de trabajo donde personas de un mismo sector comparten información acerca de su sueldo, así como de su carrera profesional, habilidades más valoradas y otros datos de los cuales se extraen estadísticas y se presentan visualmente, el sueldo medio de un desarrollador de software en España es de 27,079€/año.

Si suponemos una jornada de 8 horas diarias y un calendario laboral para el año 2020 de 255 días laborables, según indica la web [32] teniendo en cuenta festivos nacionales en España, calculamos un salario medio de 13.27€/hora.

El tiempo dedicado específicamente a programación durante el desarrollo de este trabajo de fin de grado es difícil de estimar, debido a que *Unity* no ofrece funcionalidad alguna con este tipo de característica ni existen estadísticas semejantes. La fecha de creación del proyecto en *Unity* tampoco es una buena forma de medición debido a que, durante el desarrollo de este trabajo de fin de grado, se han utilizado numerosos proyectos diferentes. Por otro lado, ha habido periodos de trabajo constante, seguidos de largas pausas y tiempo dedicado a otras tareas. Es por ello por lo que la estimación de horas dedicadas a programar la ampliación de la API debe de ser muy inexacta y basada nada más que en la percepción temporal del alumno.

En base a lo dicho en el párrafo anterior y teniendo en cuenta que se comenzó a trabajar en el desarrollo de este trabajo de fin de grado a finales del mes de septiembre de 2020, el alumno estima que ha dedicado unas 150 horas a la programación y estudio del código de la API, así como a la búsqueda de recursos y formación personal necesaria para la tarea de programar la ampliación de la API. Se reitera que dicha estimación es altamente inexacta debido a la carencia de datos para determinar una cantidad de horas con exactitud.

Teniendo en cuenta las horas que se han estimado han sido dedicadas a la programación de la ampliación de la API, así como el salario medio por hora de un desarrollador de software en España, obtenemos un coste estimado de haber llevado a cabo la ampliación de alrededor de 2,000€.

Teniendo en cuenta el coste de utilizar herramientas de terceras partes presentes en el mercado, sería posible para una empresa valorar si el desarrollo interno de una herramienta similar supondría un coste asumible a pagar por tener el control total del desarrollo y no depender de terceros, o si por el contrario valdría la pena adquirir licencias como Panda BT Free, en este caso de manera gratuita, u otras opciones más extensas pero que acarreen un coste monetario, y estar ligado a actualizaciones y limitaciones presentes en dichas herramientas.

Hay que tener en cuenta además que al coste estimado de alrededor de 2,000€ por el desarrollo de la ampliación de la API habría que computar también el coste de haber llevado a cabo el desarrollo inicial de la API, el cual no podemos estimar. Por tanto, el coste total de haber desarrollado una herramienta similar de manera interna sería notablemente superior a los 2,000€ estimados.

# 10. Conclusiones

En este apartado se lleva a cabo un análisis de los resultados obtenidos tras la realización de este trabajo de fin de grado en base a los objetivos establecidos en un primer momento al comienzo del desarrollo del mismo.

Con la realización de este trabajo de fin de grado se ha obtenido un producto útil y funcional que, como se ha demostrado en las pruebas, es eficaz en su objetivo de dar facilidades al programador a la hora de elegir qué tecnología emplear en el desarrollo de IAs para su proyecto. Como resultado de esta ampliación, el API de GAIA cuenta ahora con una opción para desarrollar IAs que, a opinión del alumno y tras haber llevado a cabo las pruebas posteriores a la ampliación, es más cómoda, rápida e intuitiva que su contrapartida en FSM. Esto es debido a que definir un BT haciendo uso del API de GAIA exime al programador de la necesidad de tener que rellenar la clase “Tags”, que es necesaria para hacer uso de las funcionalidades de FSM de la API, además de requerir un menor número de líneas a escribir en la plantilla XML debido a que no es necesario especificar todas las transiciones y estados si se utiliza un BT. Sin embargo, mientras que la implementación de las funcionalidades de BTs pueden ser más atractivas por lo recientemente explicado, no quita que, por si el programador lo deseara, se siga pudiendo hacer uso de las funcionalidades de FSM ya existentes. Se puede afirmar por tanto que la ampliación a BTs ha dotado al API de GAIA una mayor destreza a la hora de generar IAs, requiriendo menos líneas de código y trabajo para ponerlas en marcha que si se utilizara FSM, además de un alto nivel de versatilidad, al poder utilizar ambos tipos de tecnología (BTs y FSM) de manera casi intercambiable dentro de un mismo módulo.

El camino hasta la obtención de la versión final ampliada de dicho producto, la API ampliada de GAIA, ha pasado por un largo camino de formación e investigación con el objetivo de conocer ampliamente y de primera mano la tecnología de behaviour trees. Ha sido necesario recopilar y realizar pruebas con las diferentes librerías de BTs que se han encontrado, de modo que se pueda juzgar con seguridad cuál es la mejor opción para la ampliación. En el transcurso de esta tarea se ha compilado una lista de las librerías de BTs gratuitas más populares y se han clasificado en base a sus características, funcionalidades, lenguajes de programación y soporte y actualizaciones.

Se puede afirmar que se ha obtenido un conocimiento extenso de las opciones gratuitas ofertadas en el mercado en cuanto a librerías para el desarrollo de IAs mediante BTs.

A la hora de desarrollar este trabajo de fin de grado, el alumno también se ha encontrado con otras barreras de conocimiento, en particular con la necesidad de aprender a utilizar *Unity* como entorno de programación. Familiarizarse con el mismo ha sido una tarea ardua que ha llevado horas de trastear, leer documentaciones, guías y tutoriales hasta alcanzar un punto de comodidad y destreza como para poder dedicarse de manera cómoda al desarrollo de la ampliación de la API.

La familiarización del alumno con *Unity* ha sido paralela con el estudio del código de la API. En una primera instancia el alumno fue presentado con un proyecto de *Unity* no funcional, dentro del cual se encontraban de forma separada los archivos que conformaban la versión inicial de la API al principio de la realización de este trabajo. Desde el momento en que se exporta el módulo contenedor de los archivos de la API hasta el momento en que se pone en marcha de manera satisfactoria la primera prueba con FSM sobre el entorno de pruebas de “Tanks” haciendo uso de la API, existe un periodo de prueba y error, incertidumbre y, francamente, desesperación ante la entonces concebida como monumental tarea de poner en marcha las tareas que finalmente han conformado el desarrollo de la ampliación de la API.

El proceso de desarrollo pasó por multitud de lecturas e interpretaciones del código para entender el funcionamiento de la API, por el desarrollo del juego que posteriormente serviría como terreno de pruebas, las numerosas pruebas individuales realizadas sobre las diferentes librerías para valorar cada una de ellas sobre el terreno de pruebas, y finalmente la incorporación de la API, la integración sobre el terreno de pruebas y la expansión del código de la misma para soportar las nuevas funcionalidades de BTs.

De principio a fin, la realización del estudio, investigación y desarrollo necesarios para llevar a cabo este trabajo han sido fascinantes, divertidos, estresantes y, finalmente, gratificantes.

En cuanto al objetivo propuesto de generar un módulo comercializable de la API para la *Asset Store* de *Unity*, conforme se ha comentado en el apartado 6.3, no ha sido posible debido a la carencia de una documentación completa, así como de material

de guía de usuario para facilitar el estudio y uso de la API al programador. Es por ello por lo que este objetivo no ha podido lograrse como parte de este trabajo de fin de grado, pero se propone alcanzarlo como parte de trabajos futuros.

Es destacable que la realización de este trabajo de fin de grado ha sido llevada a cabo de manera completamente individual en las condiciones de trabajo a distancia impuestas por las medidas restrictivas por la pandemia. Mientras que es algo asumible que un trabajo de fin de grado se realice por los propios medios del alumno, las condiciones herméticas y las circunstancias derivadas de haber trabajado desde casa desde hace ya casi un año no han ayudado en la superación de los momentos difíciles, y ciertamente han afectado al rendimiento de trabajo del alumno.

Por último, la realización de este trabajo supone el último escalón en el largo camino desde el inicio de la carrera hasta la obtención del título de Ingeniería Informática, y sirve como muestra de los conocimientos obtenidos, así como de la experiencia de trabajo adquirida por el alumno a lo largo de la carrera.

# 11. Referencias

[1] Matteo Iovina, Edvards Scukins, Jonathan Styrud, Petter Ögren and Christian Smith. (2020).

*A Survey of Behavior Trees in Robotics and AI.*

[2] Mateas, M. and Stern, A. (2002).

*A behavior language for story-based believable agents*. IEEE Intelligent Systems, 17(4):39–47.

[3] Champandard, A. J., Dunstan, P., and Dunstan, P. (2013).

*The Behavior Tree Starter Kit*. In Rabin, S., editor, Game AI Pro, pages 27–46. CRC Press, first edition.

[4] Isla, D. (2005).

*Handling Complexity in the Halo 2 AI*. Gamasutra. https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php?print=1.

[5] Overview: Behavior Designer. Opsive.

https://opsive.com/support/documentation/behavior-designer/

[6] Nicolás Gil Soriano. (2015).

*TO THE STARS. Desarrollo de un videojuego 2D con Unity. Empleando API de Gestión de máquinas de estados finitos.*

[7] José Alapont. (2014)

*API de gestión de Inteligencia Artificial basada en máquinas de estados finitos en C#.*

[8] Michele Colledanchise and Petter Ögren. (2020)

*Behavior Trees in Robotics and AI. An Introduction.*

https://arxiv.org/pdf/1709.00084.pdf

[9] Chris Simpson. (2014).

*Behavior tres for AI: How they work.*

Gamasutra. https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

[10] Unity Technologies. (11 de febrero de 2020).

*Tanks*. Learn Unity. https://learn.unity.com/project/tanks-tutorial

[11] Icremer. (19 de enero de 2019).

*Unity_AI.* Github. https://github.com/lcremer/Unity_AI

[12] ashblue. (4 de junio de 2020).

*fluid-behavior-tree.* Github. https://github.com/ashblue/fluid-behavior-tree

[13] mhjort. (23 de marzo de 2019).

*aivo.* Github. https://github.com/mhjort/aivo

[14] ashleydavis. (12 de septiembre de 2016).

*Fluent-Behaviour-Tree.* Github. https://github.com/ashleydavis/Fluent-Behaviour-Tree

[15] Eric Begue. (19 de febrero de 2019).

*Panda BT Free*. Unity Asset Store. https://assetstore.unity.com/packages/tools/ai/panda-bt-free-33057

[16] Michele Colledanchise. (8 de octubre de 2020).

*BehaviorTree.CPP*. Github. https://github.com/BehaviorTree/BehaviorTree.CPP

[17] EugenyN. (23 de mayo de 2019).

*BehaviorTrees.* Github. https://github.com/EugenyN/BehaviorTrees

[18] PadaOne Games. (3 de abril de 2019).

*Behavior Bricks*. Unity Asset Store. https://assetstore.unity.com/packages/tools/visual-scripting/behavior-bricks-74816

[19] splitered-reality. (6 de noviembre de 2020).

*Py_trees*. Github. https://github.com/splintered-reality/py_trees

[20] libgdx. (26 de julio de 2019).

*Gdx-ai*. Github. https://github.com/libgdx/gdx-ai

[21] Calamari. (20 de agosto de 2020).

*BehaviorTree.js*. Github. https://github.com/Calamari/BehaviorTree.js

[22] meniku. (25 de enero de 2020).

*NPBehave*. Github. https://github.com/meniku/NPBehave

[23] DeveloperUX. (6 de febrero de 2013).

BehaviorLibrary. Github. https://github.com/DeveloperUX/BehaviorLibrary

[24] Open Source Robotics Foundation. (11 de junio de 2020).

Wiki.ros Documentation. ROS.org. https://wiki.ros.org/

[25] Open Source Robotics Foundation. (3 de marzo de 2016).

Behavior_tree. ROS.org. http://wiki.ros.org/behavior_tree

[26] Eric Begue. (enero de 2016).

*Panda BT*. Panda BT. Behaviour Tree Scripting for Unity. http://www.pandabehaviour.com/

[27] Unity 3D. (31 de julio de 2020).

*Asset Store Terms of Service and EULA*. https://unity3d.com/legal/as_terms

[28] Unity 3D.

*Sell Assets*. https://unity3d.com/asset-store/sell-assets

[29] Unity Asset Store.

*GAIA, the AI API*. https://assetstore.unity.com/preview/189510/592911

[30] Larry Hardesty. (14 de abril de 2017).

*Explained: Neural Networks.* MIT News. *https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414*

[31] PayScale. (31 de enero de 2021).

*Average Software Developer Salary in Spain. PayScale*. https://www.payscale.com/research/ES/Job=Software_Developer/Salary

[32] Working Days. (24 de febrero de 2021).

*Working days in Spain. National Holidays.* https://www.dias-laborables.es/EN/dias_laborables_feriados_2020_Festivos%20nacionales.htm#

[33] Robot Operating System. ROS.

https://www.ros.org/

[34] Magnus Olsson. (2016).

*Behavior Trees for decision-making in Autonomous Driving*. http://www.diva-portal.org/smash/get/diva2:907048/FULLTEXT01.pdf

[35] Blake Hannaford, Randall Bly, Ian Humphreys, Mark Whipple. (26 de agosto de 2018).

*Behavior Trees as a Representation for Medical Procedures. https://arxiv.org/pdf/1808.08954.pdf*

[36] Aadesh Neupane, Michael Goodrich. (2019).

*Learning Swarm Behaviors using Grammatical Evolution and Behavior Trees*. https://www.ijcai.org/Proceedings/2019/0073.pdf

[37] Overview: Behavior Designer. Opsive.

*What is a Behavior Tree?* https://opsive.com/support/documentation/behavior-designer/what-is-a-behavior-tree/

[38] Unity. Unity Technologies.

Unity. https://unity.com/

[39] Unity Asset Store. Unity Technologies.

Asset Store. https://assetstore.unity.com/

[40] LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. (Mayo de 2015).

"Deep learning." Nature 521, 436-444(2015).

[41] Warren McCulloch and Walter Pitts. (1986).

*A Logical Calculus of the Ideas Immanent in Nervous Activity*. In: Palm G., Aertsen A. (eds) Brain Theory.

[42] Paul Werbos. (Enero de 1974).

*Beyond regression : new tools for prediction and analysis in the behavioral sciences.* Thesis (Ph. D.)--Harvard University. https://www.researchgate.net/publication/35657389_Beyond_regression_new_tools_for_prediction_and_analysis_in_the_behavioral_sciences

[43] Jürgen Schmidhuber. (Enero de 2015).

*Deep learning in neural networks: An overview*. Neural Networks, Volume 61, Pages 85-117, ISSN 0893-6080, https://doi.org/10.1016/j.neunet.2014.09.003

# Anexo

1. Función “changeTickOn” del *manager* de la API. Fuente propia.

```
// Change the order in which the tree is ticked.
// MUST specify when the BT will tick:
//    -->  updadeType:
//                      1: Tick on Update
//                      2: Tick manually
//                      3: Tick on FixedUpdate
//                      4: Tick on LateUpdate
0 referencias
public void changeTickOn(GameObject character, int updateType)
{
    try
    {
        PandaBehaviour component = character.GetComponent<PandaBehaviour>();
        switch (updateType)
        {
            case 1:
                component.tickOn = PandaBehaviour.UpdateOrder.Update;
                break;
            case 2:
                component.tickOn = PandaBehaviour.UpdateOrder.Manual;
                break;
            case 3:
                component.tickOn = PandaBehaviour.UpdateOrder.FixedUpdate;
                break;
            case 4:
                component.tickOn = PandaBehaviour.UpdateOrder.LateUpdate;
                break;
        }
    }
    catch (Exception e)
    {
        Debug.Log("EXCEPTION: " + e);
    }
}
```

2. Función “createBT” con declaración de “ticks” del *manager* de la API. Fuente propia.

```
public void createBT(GameObject character, string bt_id, int updateType)
{
    try
    {
        PandaBehaviour component = character.AddComponent<PandaBehaviour>();
        component.Compile(BT_dic[bt_id]);
        switch (updateType)
        {
            case 1:
                component.tickOn = PandaBehaviour.UpdateOrder.Update;
                break;
            case 2:
                component.tickOn = PandaBehaviour.UpdateOrder.Manual;
                break;
            case 3:
                component.tickOn = PandaBehaviour.UpdateOrder.FixedUpdate;
                break;
            case 4:
                component.tickOn = PandaBehaviour.UpdateOrder.LateUpdate;
                break;
        }
    }
    catch (Exception e)
    {
        Debug.Log("EXCEPTION: " + e);
    }
}
```