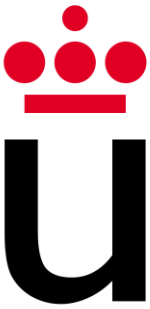


Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Grado en Diseño y desarrollo de Videojuegos

Curso académico 2022/2023

Trabajo de fin de grado

ADAPTACIÓN A UNITY DE UNA LIBRERÍA DE COMPORTAMIENTOS DE AGENTES INTELIGENTES PARA DISEÑO Y DEPURACIÓN EN TIEMPO REAL

Autor

Alejandro Orejudo Fraile

Tutores

Aarón Sújar Garrido
Carlos Garre Del olmo

Resumen

En el desarrollo de videojuegos, los personajes son uno de los elementos fundamentales por la capacidad que aporta a los diseñadores de narrar historias a través de ellos, y también de ofrecer desafíos a los jugadores adaptados a sus capacidades individuales.

Para crear buenos personajes es necesario tener un control preciso de cómo debe ser su comportamiento en cada situación, lo que requiere conocimientos en distintas técnicas como sistemas de toma de decisiones e inteligencia artificial. Por desgracia, muchos estudios no cuentan con recursos suficientes para poder utilizar herramientas profesionales e incluso estas herramientas pueden ser demasiado especializadas o no adaptarse a las necesidades del estudio. Además, especialmente en el caso de desarrolladores con poca experiencia o estudios independientes, aprender a usar estas herramientas puede requerir más tiempo del que pueden permitirse.

Teniendo en cuenta todo esto, en este trabajo se propone la creación de una herramienta para utilizar sistemas de comportamiento, específica para el motor de videojuegos Unity. Esta herramienta consistirá en una integración de una librería desarrollada en C# dentro del motor de videojuegos, a través de distintos *scripts* y elementos que faciliten su diseño y ejecución, tanto a través de código como utilizando un editor visual. Esta herramienta se centrará en ofrecer una interfaz amigable al usuario y una curva de aprendizaje especialmente planteada para estudiantes del grado de diseño y desarrollo de videojuegos, permitiendo crear tanto prototipos simples de personajes como sistemas complejos y manteniendo siempre como prioridad que los usuarios comprendan en todo momento su funcionamiento para que les ayude a entender cómo funcionan las distintas técnicas para crear sistemas inteligentes.

La librería original cuenta con bastantes funcionalidades, y está pensada para expandirse en un futuro, por lo que la herramienta se diseñará teniendo en cuenta esto. Será posible crear personajes funcionales utilizando solo partes específicas de la herramienta, de forma que los desarrolladores puedan aprender a usarla de forma práctica y en poco tiempo, y para apoyar esto se incluirán varias herramientas de depuración en tiempo real.

Palabras claves: Sistemas de comportamiento, agente inteligente, *NPC*, inteligencia artificial, assets, script, *Unity*, *visual scripting*, grafo, nodo, *Smart Object*.

Índice de contenido

Índice de figuras

1

Introducción

La inteligencia artificial en personajes es uno de los pilares fundamentales en el diseño de videojuegos. Aunque en general el objetivo de la inteligencia artificial es ayudar a resolver problemas que para las personas podría requerir una cantidad de tiempo enorme, o directamente resultar imposible, en videojuegos normalmente no es así. Si se creara un personaje con el sistema de inteligencia más sofisticado del momento, el resultado sería un reto completamente imposible de superar para el jugador y que no aportaría más que frustración a la experiencia.

En realidad, el objetivo de la IA en un videojuego es hacer que los personajes se adapten a cada jugador y le ofrezcan un reto a su medida. Existen ejemplos, como el caso de *Alien: Isolation* (Creative Assembly, 2014), en el que el principal enemigo y obstáculo del jugador, el Xenomorfo, usa un sistema de comportamiento basado en medir el estrés esperado en el jugador y modificar sus acciones en base a él [1]. También es cierto que en muchos casos no se pretende que los personajes supongan un reto de ningún tipo para el jugador, pero la inteligencia artificial también sirve para aportar variabilidad y sentido a los personajes y así enriquecer la jugabilidad.

La mayoría de los videojuegos del mercado utilizan sistemas de inteligencia artificial para diseñar comportamiento de personajes, sin importar si se trata de un producto triple A o desarrollado por un estudio independiente. Además, hoy en día el mundo del diseño y desarrollo de videojuegos ha evolucionado tanto que es posible crear un videojuego completamente funcional con sistemas complejos sin tener que escribir una sola línea de código, gracias a las herramientas de scripting visual. Estas permiten ahorrar tiempo de desarrollo a los programadores o incluso permitir que profesionales de otros campos distintos a la programación, como el arte o el diseño, puedan realizar el trabajo que les correspondería, lo que es especialmente interesante en equipos pequeños.

1.1 Objetivos

El objetivo del proyecto es integrar en el motor Unity [2] una librería de C# que permite crear sistemas de comportamiento en forma de grafos. Esta librería se ha desarrollado de forma simultánea en otro trabajo, titulado *"Contribuciones y refactorización de una librería de comportamiento de agentes inteligentes incorporando smart objects"*, y aunque se incluirá un resumen de éste en el apartado "Trabajo previo", es aconsejable leer antes dicho trabajo.

El proyecto se basa en una extensión de la *API* para aprovechar algunas características del motor, como la programación orientada a componentes, además de crear una herramienta que permita diseñar los sistemas de comportamiento de forma visual y depurar el resultado en tiempo real.

Hay que considerar que la librería de C# original está pensada para ampliarse con nuevas funcionalidades, por lo que se requiere que la herramienta se diseñe teniendo en cuenta en la adaptabilidad a dichas ampliaciones en vez de crear un sistema cerrado a la *API* actual. Por ejemplo, si se añaden nuevos tipos de sistemas de comportamiento, o nuevos nodos a los tipos existentes, es importante que la cantidad de código que deba cambiar a la herramienta de edición sea mínima siguiendo el principio *Open/Closed* [3], mantener el código abierto a extensiones, pero cerrado a cambios.

Hoy en día existen múltiples herramientas de "visual scripting" (ver fig. 1.1) en el mercado y aunque tienen grandes diferencias la mayoría emplean un lenguaje común que consiste en construir los elementos más simples como nodos y crear conexiones entre ellos para crear sistemas más complejos. Esto permite ampliar y personalizar la herramienta creando nuevos nodos y ofrece la flexibilidad de trabajar con elementos de bajo y alto nivel de abstracción dando gran libertad al diseñador. Al diseñar la herramienta del editor visual, se deben considerar estas características para que el usuario tenga un control de los elementos más simples dentro de los grafos de comportamiento, y pueda crear componentes más complejos que sean reutilizables.

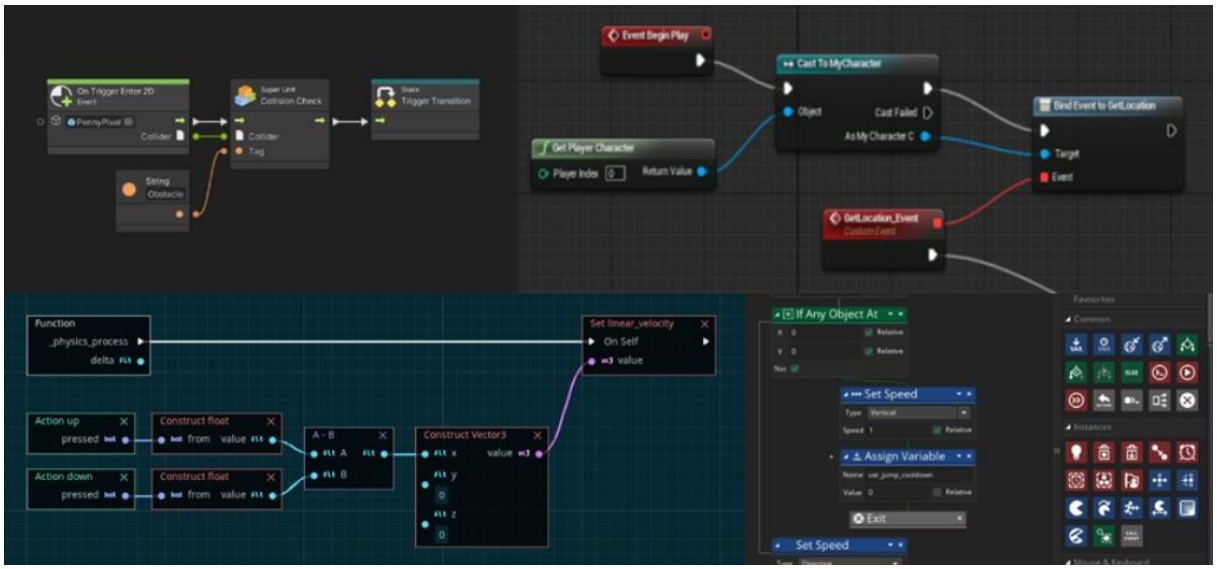


Fig. 1.1 Interfaces de herramientas para scripting visual en distintos motores de videojuegos [4][5][6][7].

A la hora de trabajar con la herramienta, los usuarios deben poder crear sistemas de comportamiento mediante scripts, o usando el editor visual, e incluso combinar ambas técnicas. Por ello, la representación de los sistemas de comportamiento en el editor debe ser fácil de relacionar y comparar con la forma de crearlos mediante código. Para facilitar esto, se creará una herramienta para generar código a partir de un sistema de comportamiento creado con el editor.

1.2 Metodología y herramientas

1.2.1 Planteamiento del trabajo a desarrollar

Antes de comenzar con el desarrollo del proyecto se realizó un análisis de las distintas herramientas disponibles en el mercado y de las características de la librería que se quiere adaptar a *Unity*. Una vez establecido se realizó una investigación sobre los mecanismos que ofrece el motor grafico para desarrollar herramientas visuales y ventanas dentro del editor, especialmente aquellas que facilitasen representar datos en forma de grafo.

1.2.2 Gestión del proyecto

El proyecto se ha dividido en distintos subproyectos (ver fig. 1.2) según la funcionalidad concreta que se quiere conseguir en cada uno de ellos. Para gestionar estos subproyectos se ha utilizado “Notion” [8], una herramienta gratuita destinada a la organización y a la productividad, que combina creación de bases de datos, wikis, notas, gestión de proyectos y

documentos. Es multiplataforma, funciona en la nube y gratuita, aunque cuenta con un plan de pago.

Los subproyectos se han dividido a su vez en tareas de forma que todas las tareas requieran aproximadamente la misma cantidad de tiempo para su desarrollo.

Aa Name	Tareas (ST)	Tareas
Integración de la API en Unity	1	15
Extensiones de Unity Toolkit	0	4
Acciones y percepciones específicas de unity	0	18
Interfaz del editor de grafos de comportamiento ABRIR	0	27
Depurador de Unity en tiempo real	0	7
Generador de código	0	8
Demos de Unity	0	6
Smart objects en unity	0	7

Fig. 1.2 División en subproyectos del desarrollo de la herramienta de Unity

El desarrollo del proyecto ha consistido en periodos o "sprints" de entre dos y tres semanas, y al comienzo de cada una se organizaba una reunión para establecer las tareas a realizar en el sprint siguiente y valorar el trabajo de ese sprint. El trabajo a realizar se dividía en tres partes:

- Cambios y correcciones a elementos ya desarrollados: Arreglar posibles fallos o bugs de los elementos desarrollados en sprints anteriores antes de comenzar nuevas tareas.
- Documentación y guías: Crear la documentación y las guías correspondientes a las funcionalidades que ya están cerradas.
- Nuevas funcionalidades: Siguiendo la planificación inicial, decidir que funcionalidades implementar en el siguiente sprint en base a una estimación del tiempo que va a ser necesario.

1.2.3 Desarrollo del código

El proyecto se ha desarrollado dentro de Unity [2], una aplicación que integra todos los aspectos del desarrollo de un videojuego: gráficos, sonido, comportamiento, físicas, etc. Para desarrollar el código del proyecto se ha utilizado Visual Studio [9], un entorno de desarrollo que permite editar y compilar código en distintos lenguajes y plataformas como C++ o .NET, además de ser el editor de código predeterminado para Unity.

2

Trabajo previo

Este proyecto consiste en la integración de una librería desarrollada en C# para crear sistemas de personajes en Unity, y se ha desarrollado en paralelo a la propia librería. Por tanto, es importante incluir un resumen de sus características, destacando los puntos que serán más relevantes para la herramienta desarrollada.

BehaviourAPI es una librería para crear árboles de comportamiento, máquinas de estados y sistemas de utilidad en forma de grafos dirigidos formados por nodos conectados entre sí (ver fig. 2.1). Cada nodo tiene una función distinta dependiendo del sistema concreto. Se incluyen varios tipos de grafos: árboles de comportamiento, máquinas de estados y sistemas de utilidad, añadiendo también las máquinas de estados de pila. Cada uno de ellos tiene sus propios nodos y controla su ejecución de forma distinta.

Las máquinas de estados son un mecanismo fundamental en agentes inteligentes y en casi cualquier disciplina dentro de la programación. Se basan en estados conectados entre sí mediante transiciones, y en cada instante hay un único estado activo. Son bastante útiles para crear sistemas de comportamiento por su facilidad de interpretación, aunque presentan limitaciones si se quieren crear sistemas complejos.

Los árboles de comportamiento se basan en una estructura de nodos en forma de árbol, en la que cada nodo construye su comportamiento en base a sus hijos, hasta llegar a los nodos hoja, que lo hacen a través de acciones. Cada vez que se actualiza el sistema, la ejecución sigue un camino por los nodos y cada nodo del camino devuelve un resultado a su nodo padre, hasta llegar a la raíz. Puede interpretarse también como una máquina de estados en las que las transiciones son automáticas y se basan en recorrer el árbol de forma trasversal, aunque esto varía con algunos nodos como los condicionales o paralelos.

Los sistemas de utilidad se basan en un conjunto de acciones y en cada actualización el sistema escoge una de ellas en base a un valor numérico llamado utilidad. La utilidad se obtiene en base a valores del entorno que se normalizan entre 0 y 1 y pueden modificarse y combinarse con otros valores a través de una jerarquía de nodos llamados factores.

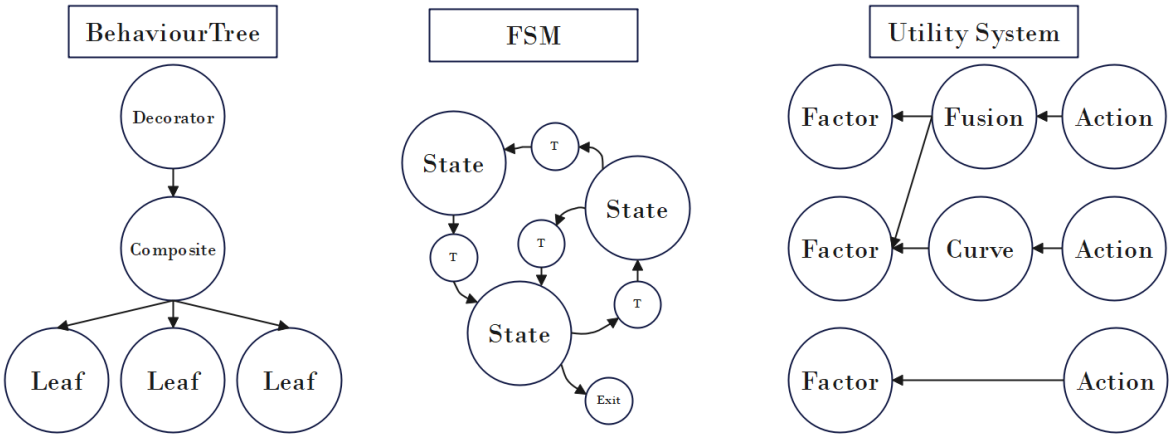


Fig. 2.1 Sistemas de comportamiento modelados como grafos dirigidos.

La ejecución de estos sistemas se basa en lanzar eventos que se propagan por los grafos y nodos y que modifican su estado interno o *Status*. Este estado interno es un enumerado que puede tomar los siguientes valores:

- *None*: El sistema no se está activo, porque no se ha comenzado la ejecución o porque se ha interrumpido de algún modo.
- *Running*: El sistema se está ejecutando actualmente pero no ha terminado.
- *Success*: La ejecución del sistema ha terminado con éxito.
- *Failure*: La ejecución del sistema ha terminado con fallo.
- *Paused*: La ejecución esta pausada.

Y los eventos que cambian dicho estado son los siguientes:

- *Start*: Se lanza al comienzo de la ejecución y pone el valor *Status* en Running.
- *Update*: Se lanza en cada iteración para actualizar la ejecución del sistema. En algunos casos, como en los árboles de comportamiento, puede poner el valor de *Status* a *Success* o *Failure*.
- *Stop*: Detiene la ejecución, sin importar si ha terminado o si está pausado. Pone el valor Status a *None*.
- *Pause*: Indica que la ejecución se ha pausado. Si el valor Status era *Running* lo cambia a *Paused*.

- *Unpause*: Se lanza para indicar que la ejecución se ha reanudado. Si el valor Status era Running
- *Finish:* Termina la ejecución del sistema con Success o Failure dependiendo del parámetro especificado. Aunque el programador puede usarlo directamente, está pensado para que lo haga el propio sistema de comportamiento, como en el caso de las transiciones de salida en máquinas de estados.

Los grafos de comportamiento interactúan con el entorno a través de *acciones* y *percepciones*. Las acciones son tareas que contienen algunos nodos y se ejecutan mientras estos nodos están activos y las *percepciones* son elementos que comprueban ciertas condiciones y sirven para controlar el flujo de ejecución.

Tanto las acciones como las percepciones tienen métodos que son lanzados desde el nodo que las contiene cuando cambia su estado de ejecución.

- *Start* (*Init* en percepciones): Inicializa la acción o percepción.
- *Stop* (*Reset* en percepciones): Resetea la acción o percepción.
- *Pause* / *Unpause*: Pausa / despausa y la acción o percepción.
- *Update* / *Check:* En las acciones este método devuelve un valor *Status* (*Running* si la acción no ha terminado o *Success* / *Failure* si ha terminado con éxito o fallo). En las percepciones devuelve *true* o *false* dependiendo de si se cumple o no la condición-

La *API* incluye varios tipos de acciones y percepciones, incluyendo la posibilidad al usuario de crear sus propios tipos. También incluye la posibilidad de crear sistemas jerárquicos a través de un tipo de acción concreto llamado *SubsystemAction* que contiene un sistema de comportamiento dentro de la propia acción y le propaga sus eventos de ejecución.

Otro elemento relevante de la API son las percepciones push, que permiten alterar la ejecución de un sistema desde fuera del mismo. Cada elemento de la clase *PerceptionPush* tiene una lista de elementos que implementan la interfaz *IPushActivable* y cuando se activa lanza el método *Fire* de cada elemento. El efecto que tiene en la ejecución depende de la implementación de la interfaz:

- En transiciones de una *FSM*, las activa sin importar si la percepción que tienen asignada se cumple o no.
- En nodos de un *BehaviourTree*, cambia su *Status* al valor especificado en el método *Fire*, que debe ser *Success* o *Failure*.

Otro elemento destacable es el *contexto de ejecución*. Se trata de una estructura de datos que se propaga a todos los elementos de un sistema de comportamiento y permite compartir información entre ellos.

Por último, la API incluye un sistema de *SmartObjects*, que son objetos con los que los agentes inteligentes pueden interactuar, con la particularidad de que es el objeto el que define como es esta interacción en lugar del agente. Además, estos objetos pueden cubrir necesidades especificadas en los agentes, como hambre, descanso, etc.

El funcionamiento de los *SmartObjects* se basa en un esquema petición-respuesta. Desde los sistemas de comportamiento, se usa un tipo especial de acción llamado *RequestAction*, que busca un *SmartObject* y le manda una petición en la que incluye el agente y otros datos como la necesidad que quiere cubrir. El objeto usa los datos recibidos y genera una *SmartInteraction*, un objeto que contiene la acción que debe realizar el agente para interactuar con él. La *RequestAction* recibe la interacción y propaga sus eventos (*Start*, *Update*, *Stop*, etc) hasta que la interacción termina. Si ha terminado con éxito, se aplican las capacidades del objeto en el agente.

3

Estudio de estado del arte

A continuación, veremos algunas de las herramientas que existen para crear sistemas de comportamiento en los principales motores de videojuegos.

3.1 Herramientas oficiales

Unity [2] y Unreal [10] son dos de los principales motores de videojuegos en la industria y ambos ofrecen una gran cantidad de extensiones creadas por la comunidad. Aunque es posible encontrar gran variedad de herramientas para crear agentes inteligentes en sus respectivos *Marketplaces*, también cuentan con soluciones oficiales integradas en el motor.

3.1.1 Unreal

Unreal proporciona una herramienta para crear árboles de comportamiento [11] a partir de su versión 4.26. Las principales características de esta herramienta es que la ejecución del sistema no se actualiza periódicamente, sino que reacciona a distintos eventos como cambios en variables.

Para compartir datos entre nodos, usa un sistema de pizarra o *blackboard*, en el cual se registran variables o *blackboard keys*. Esta estructura se guarda como un asset y puede asignarse a los distintos árboles de comportamiento. Para poder usarlos, se crea un personaje con un controlador de IA asignado. A través de este controlador, se activan los eventos del árbol usando el sistema de *blueprints* de Unreal.

3.1.2 Unity

Unity proporciona un sistema de máquinas de estados [12] integrado en *Bolt* (ver fig. 3.1), su sistema de visual scripting, que permite ejecutar scripts desde los estados de la máquina. La

herramienta proporciona dos tipos de estados: El primer tipo contiene un script visual que se ejecutará cuando el estado sea seleccionado, mientras que el segundo sirve para crear submáquinas de estados.

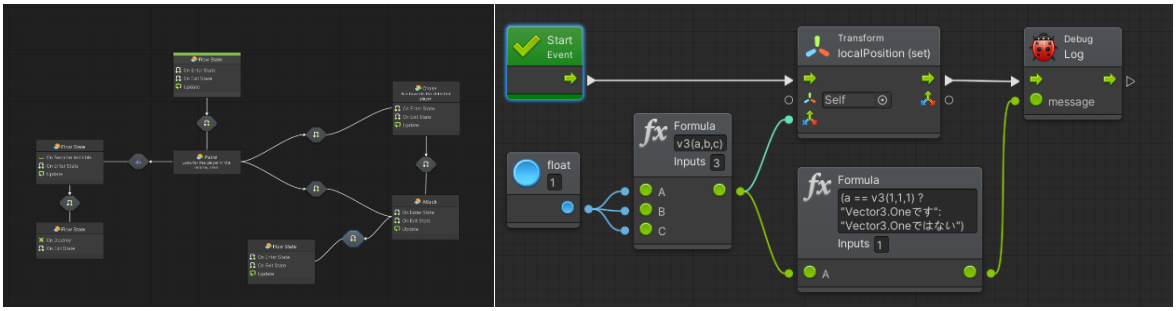


Fig. 3.1 Editor de máquinas de estados de Unity [12]

Una vez dentro de un estado, es posible utilizar los tres eventos que se proporcionan (cuando se entra en el estado, cada actualización dentro del estado y cuando se sale de él) para implementar el comportamiento utilizando el sistema de scripting visual.

3.2 Herramientas de terceros

3.2.1 PANDA BT

PandaBT [13] es una herramienta que permite crear árboles de comportamiento directamente con ficheros de texto que describen la estructura de los nodos. Esto hace que sean fáciles de modificar y compartir lo que lo hace una herramienta bastante buena para aprendizaje, además de ser robusto a cambios.

PandaBT permite a los usuarios crear acciones personalizadas y usarlas directamente escribiendo el nombre de los métodos en los ficheros de texto. Aunque es posible crear tipos de nodos, la librería no ofrece un soporte directo para ello.

Algunas funcionalidades como la utilización de breakpoints está limitada a la versión de pago.

3.2.2 NODE CANVAS

Node Canvas [14], es una herramienta para crear árboles de comportamiento, máquinas de estados y árboles de diálogos. Cuenta con un editor de grafos para crear cualquiera de los tres sistemas que permite la herramienta, además de un conversor a formato json.

La herramienta permite crear nuevos tipos de nodos y acciones, además de personalizar como se muestran en el editor. Estas acciones tienen acceso a una estructura de datos compartida

llamada pizarra o “blackboard”, que puede usarse para compartir datos entre nodos. Además, incluye múltiples acciones como mover al agente a una posición concreta o añadir una variable a una lista.

Distingue entre dos formas de crear los sistemas. Pueden asignarse directamente a componentes de objetos llamados “*GraphOwners*” lo que les permite afectar directamente a la escena en la que se encuentran. La alternativa es crearlos y guardarlos como assets del proyecto, de esta forma serán reutilizables en múltiples objetos, pero no tendrán acceso a variables de la escena.

Permite observar la ejecución de los sistemas creados en tiempo e incluso añadir y borrar nodos en tiempo de ejecución. Además, incluye soporte para una herramienta de scripting visual desarrollada por la misma compañía, llamada *Flow Canvas*, permitiendo usar scripts visuales como nodos de un árbol de comportamiento o estados de una máquina de estados.

3.2.3 Behaviour Designer

Behaviour Designer [15] es de una herramienta para crear árboles de comportamiento de forma visual y simple. La ejecución de estos sistemas se basa en tareas, que pueden ser tanto acciones como condiciones.

Permite extender la funcionalidad creando nuevos tipos de acciones o condiciones como clases propias. En lugar de usar un sistema de blackboard, define varios tipos de variables según el alcance que puedan tener (dinámicas, globales, compartidas, etc).

Algunas de las funcionalidades extra que incluye son depuración en tiempo real, y soporte para UNET, que era el sistema de red de Unity (actualmente remplazado por *Netcode for Objects* y *Netcode for entities*). La herramienta ha sido utilizada en múltiples proyectos profesionales y cuenta con extensiones propias y de terceros que añaden funcionalidades.

4

Diseño y desarrollo

Tal y como se ha mencionado anteriormente, este proyecto consiste en adaptar una librería de C# al entorno de Unity. Para ello, se crearán distintas herramientas basadas en el flujo de trabajo de Unity, de forma que resulten intuitivas y fáciles de usar a usuarios familiarizados con el motor de videojuegos.

Se ha decidido dividir el proyecto en dos partes. La primera parte es un conjunto de extensiones y *scripts* para integrar el uso de sistemas de comportamiento y *smart objects* en *Unity*. Dentro de esta sección, se crearán varios componentes que permitan ahorrar trabajo al desarrollador a la hora de crear y ejecutar estos sistemas, así como distintas clases que simplifiquen su diseño, como acciones o percepciones específicas para usar los distintos componentes de *Unity*.

Esta primera parte del proyecto se desarrollará teniendo en cuenta que también servirán como base para la segunda parte. Esta consistirá en una herramienta para crear sistemas de comportamiento con un editor visual, inspirado en otras herramientas visuales propias de *Unity* como pueden ser el editor de shaders o "*shader graph"* o el animador, basado en máquinas de estados.

Esta herramienta permitirá crear y editar elementos de la API de C#, de la primera parte del proyecto y también elementos creados por el usuario, ya sean nodos, acciones, percepciones, etc. Además de crear sistemas de comportamiento, se incluirán funcionalidades para comprobar el flujo de ejecución de dichos sistemas en tiempo real, así como poder pasar de sistemas creados con el editor a sistemas creados por código de una forma simple, a través de un generador de código.

4.1 Unity Toolkit

Esta sección de la herramienta consiste en un conjunto de clases y scripts que facilitan la creación y el uso de sistemas de comportamientos en el entorno de *Unity*, además de aprovechar ciertas características del motor de videojuegos.

4.1.1 Diseño e implementación

Las distintas funcionalidades que incluye este *toolkit* se centran en el modelo de programación orientado a componentes. En *Unity*, cada objeto en el juego se divide en componentes o *scripts* independientes que se enfocan en una tarea en concreto. El componente *Transform* se encarga de controlar la posición, rotación y escala de objeto mientras que el componente *Rigidbody* controla las físicas, el componente *Collider* las colisiones, etc. Siguiendo este modelo, para que un personaje ejecute un sistema de comportamiento, es necesario crear un *script*, añadirlo al personaje y después crear y ejecutar el sistema de comportamiento desde él.

En *Unity*, todos los componentes son clases que heredan de una clase primaria llamada *MonoBehaviour*. La particularidad principal de esta clase es que funcionan en base a métodos que se invocan en respuesta a ciertos eventos, como *Start*, cuando comienza la ejecución del *script*, o *Update*, en cada frame mientras está activo [16]. Dado que la ejecución de los sistemas de comportamiento en la API también funciona en base a eventos, es fácil relacionar cada método de Unity con un evento del sistema (ver fig. 4.1).

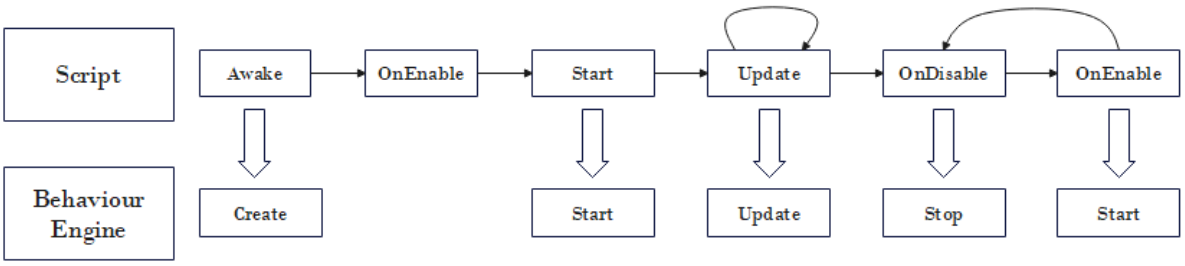


Fig. 4.1 Relación entre los eventos de Unity y los eventos de los sistemas de comportamiento

Para evitar al usuario la tarea de implementar este sistema de llamadas por cada script que desee crear, se incluye en el *toolkit* de *Unity* una clase llamada *BehaviourRunner*, que se encarga de hacerlo. De esta forma el usuario solo tiene que crear un script que herede de dicha clase y crear el sistema de comportamiento a través de su método *CreateGraph*.

Aunque la mayoría de los eventos de ejecución son fijos, esta clase también permite configurar que eventos lanzar cuando el componente es deshabilitado y habilitado, es decir, cuando se lanzan los métodos *OnEnable* y *OnDisable*, existiendo tres posibilidades.

- No se lanza ningún evento: este es el comportamiento por defecto, y hace que cuando el *script* este desactivado no se llame a *Update*, pero no se lanzará ningún evento al activar y desactivar el componente.
- Pausa: Además de bloquear las llamadas a *Update*, al desactivar y activar el *script* se lanzarán los métodos *Pause* y *Unpause* respectivamente.
- Reinicio: Cuando se desactive el *script*, la ejecución se parará completamente y al reactivarlo comenzará desde el principio. Se corresponde al diagrama anterior (ver fig. 4.1).

También es posible configurar para que ejecute el sistema en bucle, de forma que cuando termine su ejecución con éxito o fallo, se reinicie y comience desde el principio.

Otra de las funcionalidades añadidas para facilitar el uso de la *API* es el contexto de ejecución de *Unity* o *UnityExecutionContext*.

El contexto de ejecución es una funcionalidad de la *API* base que consiste en una estructura de datos compartida para todos los elementos de un sistema, que permite compartir datos entre ellos. Dentro del *toolkit* de *Unity*, se ha aprovechado esta funcionalidad para que las acciones y percepciones tengan acceso implícito al objeto que las ejecuta, y así facilitar el diseño de acciones específicas y su integración posterior en la herramienta del editor visual.

La clase *UnityExecutionContext* es una implementación de la clase *ExecutionContext* que se crea usando una referencia al *script* que va a ejecutar el sistema de comportamiento. Al crearse, guarda referencias a los componentes más comunes del objeto (*Transform*, *Rigidbody*, *Collider*, etc), actuando como una caché de componentes para las acciones y percepciones, y también permite acceder a cualquier otro componente usando el método *GetComponent*.

Para hacer uso del contexto de ejecución de *Unity* se han creado tipos específicos tanto de acciones como de percepciones, llamados *UnityAction* y *UnityPerception* respectivamente. Estas clases guardan una referencia al contexto de ejecución cuando se propaga de forma que es posible usarlo a la hora de implementar sus eventos. Además, para evitar tener que buscar las referencias a los componentes requeridos cada vez que se lanza un evento de ejecución, se ha añadido un método *OnSetContext*, que se lanza únicamente cuando la acción o percepción recibe el contexto, y así guardar las referencias necesarias en ese momento.

4.1.2 Lista de componentes

A continuación, se muestra la lista de componentes incluidos en la herramienta. El diseño de estos componentes se ha basado en el patrón *Strategy* [17], de forma que, por cada funcionalidad implementada, se han definido una serie de métodos en una interfaz para aplicarlos en componentes concretos según las necesidades de cada proyecto. Esto también permite que los usuarios puedan implementar sus propios componentes para funcionalidades que ya han sido consideradas, y así facilitar el desarrollo.

Componentes de movimiento: Estos componentes implementan la interfaz *IMovementComponent* y se usan en acciones que se basan en mover al personaje. Se incluyen dos implementaciones de este componente:

- *NavMeshMovementComponent*: Utiliza un *NavMeshAgent* para realizar el movimiento del personaje. Este componente calcula las rutas para moverse de un punto a otro mediante una malla de navegación [18], una estructura que representa las zonas del escenario en las que el agente puede situarse.
- *TransformMovementComponent*: Utiliza el componente transform del agente para realizar el movimiento.

Componentes de dialogo: Implementan la interfaz *ITalkComponent* y se usan para que el agente muestre un diálogo o mensaje.

- *TextTalkComponent*: Usa un componente *Text* para mostrar el dialogo deseado.
- *TMProTalkComponent*: El funcionamiento es idéntico al anterior, pero usa un componente *TextMeshPro* en lugar de Text.

Componentes de sonido: Implementa la interfaz *ISoundComponent* y permite usar acciones que consistan en reproducir sonidos.

- *BasicSoundComponent*: utiliza una referencia a un componente *AudioSource* para generar los sonidos.

Componentes de render: Implementa la interfaz *IRenderComponent* y se usa para renderizar sprites en la escena.

- *SpriteRenderComponent*: Usa un componente *SpriteRenderer* para mostrar el *sprite* directamente como un elemento del escenario.
- *GUIRenderComponent*: Usa un componente *Image* para mostrar el *sprite* en un elemento del *GUI*.

Es importante tener en cuenta que el usuario no está obligado a usar estos componentes para implementar sus propias acciones, ya que el objetivo es hacer la herramienta lo más flexible posible. Es posible usar cualquier componente del objeto a través del contexto de ejecución de *Unity*.

4.1.3 Lista de acciones

Para servir como guía al usuario a la hora de desarrollar sus propias acciones en *Unity*, se han añadido a la herramienta una serie de ejemplos, que además han servido para probar los componentes implementados.

Acciones de depuración: Estas acciones únicamente sirven para tareas de depuración y no tienen ningún efecto fuera del editor de *Unity*. Siempre devuelven *Success*.

- *DebugLogAction***:** Imprime un mensaje por consola. Al hacer clic en el mensaje dentro del editor de *Unity* se resaltará el objeto que contiene el sistema de comportamiento en la escena.
- *DebugBreakAction*: Pausa la ejecución de la aplicación. Útil para crear *breakpoints* dentro de un sistema de comportamiento.

Acciones de movimiento: Acciones que consisten en modificar la posición del agente. Para usar estas acciones es necesario que el objeto tenga asignado un componente que implemente la interfaz *IMovementComponent*.

- *WalkAction*: Mueve el personaje hasta una posición especifica, devolviendo *Success* cuando ha llegado a dicha posición.
- *PathingAction*: Mueve el personaje por una serie de puntos, devolviendo *Success* cuando los ha recorrido todos.
- *PatrolAction:* Mueve al personaje a una posición aleatoria alrededor del agente, pudiendo especificar la distancia máxima desde el punto actual.
- *MoveToMousePosAction:* Mueve al personaje a la posición actual del cursor. Para calcular la posición en el escenario, se lanza un rayo desde la cámara usando la función *ScreenPointToRay* y se establece como destino el punto de colisión. La acción permite especificar tanto la distancia máxima del rayo como las capas a las que afecta.
- *ChaseAction*: Al ejecutar esta acción, el agente se moverá hacia otro objeto especificado por la variable target, de tipo *Transform*. Si el agente logra alcanzar su objetivo en un tiempo especificado, la acción devolverá *Success*, y en caso contrario devolverá *Failure*. Tanto el tiempo máximo, como la distancia a la que tiene que estar el objeto del agente

para que se considere que lo ha atrapado son parámetros de la acción que pueden modificarse.

- *FleeAction*: Su funcionamiento es muy similar a la acción anterior, pero en este caso el agente debe alejarse del objetivo en lugar de acercarse. Si logra alejarse una distancia concreta antes de que pase un tiempo determinado, la acción devolverá *Success*, en caso contrario devolverá failure.

Acción de diálogo: Esta acción usa un componente que implemente la interfaz *ITalkComponent*. Como se muestre el diálogo en la escena depende del componente concreto que se use.

- *TalkAction*: Reproduce un diálogo especificado por una variable *text* de tipo *string*. La acción termina cuando el diálogo se ha terminado de reproducir y se ha esperado un tiempo concreto especificado por la variable *delay*.

Acciones de sprites: Estas acciones usan un componente que implemente la interfaz *IRenderComponent* para modificar como se muestra una imagen. Estas acciones cambian ciertas propiedades del componente que se usa para renderizar las imágenes, y permiten parametrizar el tiempo que duran estos cambios. Si el valor especificado para el tiempo es menor o igual a 0, el cambio será permanente. La acción siempre devolverá *Success* cuando el tiempo especificado haya pasado.

- *ChangeSpriteAction*: Cambia el Sprite que muestra el componente por otro especificado como parámetro en la acción.
- *FlipSpriteAction*: Voltea la imagen en el eje x, eje y o en ambos.
- *ChangeAlphaAction*: Cambia la opacidad de la imagen.
- *ChangeTintAction*: Cambia el color de la imagen.

Acciones de sonido: Sirven para reproducir un sonido o cambiar alguna propiedad relacionada con componentes de audio. Para usarse, el objeto debe tener un componente que implemente la interfaz *ISoundComponent*.

- *PlaySoundAction*: Reproduce un clip de audio concreto
- *SetVolumeAction*: Modifica el volumen del componente de audio del objeto.

Acción de espera (*DelayAction*): Esta acción bloquea la ejecución del sistema de comportamiento hasta que pasa una cantidad de tiempo especifica. Para calcular el tiempo, en

cada actualización de la acción se utiliza la propiedad *Time.deltaTime*, que especifica la cantidad de tiempo en segundos que ha pasado del frame anterior al actual.

4.1.4 Lista de percepciones

Percepción de tiempo (*UnityTimerPerception*): Funciona igual que la clase *TimerPerception* devolviendo false hasta que pasa una cantidad de tiempo específica, pero usa el escalado de tiempo de Unity lo que permite adaptarse si la aplicación se pausa o si la tasa de frames por segundo cambia.

Percepción de distancia (*DistancePerception*): Comprueba que la distancia a otro objeto especificado por una variable *Transform* es menor o igual a un valor definido en la percepción. Si la distancia del agente al objeto es mayor que dicho valor, la percepción devolverá *true*, en caso contrario devolverá *false*.

Percepción de detección (*IsLookingAtPerception*): Comprueba que un objeto concreto está en el rango de visión del agente. Para especificar este rango de visión se construye un volumen en forma de cono truncado usando tres variables: distancia mínima al objeto, distancia máxima al objeto y ángulo de apertura.

Percepción de contacto con el suelo (*IsGroundedPerception*): Esta percepción devuelve *true* si el personaje está en contacto con el suelo. Para poder usar esta percepción, el objeto del agente debe tener el componente *CharacterController*.

4.1.5 Lista de nodos específicos para Unity

Además de las clases explicadas anteriormente, se han añadido varios tipos de nodos específicos para usarse en *Unity*.

UnityCurveFactor: El nodo *UnityCurveFactor* es un tipo de curva de utilidad que permite editar la función que modifica la utilidad de su factor hijo de forma visual, aprovechando las *animation curves* [19] de Unity. Una *animation curve* es una estructura formada por una serie de puntos o *keyframes* que forman una función, con la particularidad de que es posible editar la curva directamente desde el inspector de Unity, añadiendo y moviendo los keyframes (ver fig. 4.2).

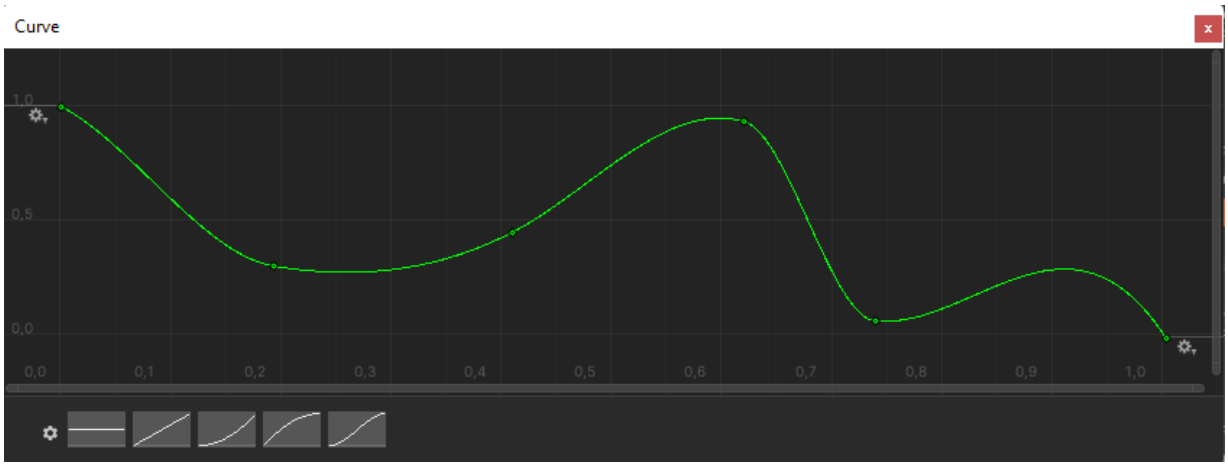


Fig. 4.2 Editor de curvas de animación en Unity

UnityTimerDecorator: La clase *UnityTimerDecorator* es una versión de *TimerDecorator* que usa el sistema de tiempo de Unity. Al igual que la clase *UnityTimerPerception*, permite adaptar el contador de tiempo a pausas en la aplicación o a cambios en la escala de tiempo.

4.1.6 Integración con Smart Objects

En este apartado se desarrollan las distintas clases implementadas para integrar el sistema de *Smart Objects* de la *API* de C# en *Unity*.

SmartAgent

Los agentes inteligentes son los elementos que permiten a los sistemas de comportamiento usar objetos inteligentes. Dentro de la herramienta, los agentes se han implementado a través de un componente llamado *SmartAgent*, que implementa la interfaz *ISmartAgent*. Implementar los agentes como *Scripts* de *Unity* tiene la ventaja implícita de que los objetos podrán acceder a cualquier otro componente del objeto directamente para crear sus interacciones. Por ejemplo, si un *SmartObject* quiere cambiar la posición del agente solo tiene que acceder al campo *transform.position* del *SmartAgent*.

Cada agente inteligente tiene una serie de necesidades, que deberá satisfacer usando los *SmartObjects* cuando complete interacciones con ellos. Para definir estas necesidades se ha creado la clase *SmartAgentSettings*, que hereda de *ScriptableObject* [20], lo que permite guardar la configuración de las necesidades del agente como un *asset* del proyecto y así poder usar la misma configuración en varios agentes.

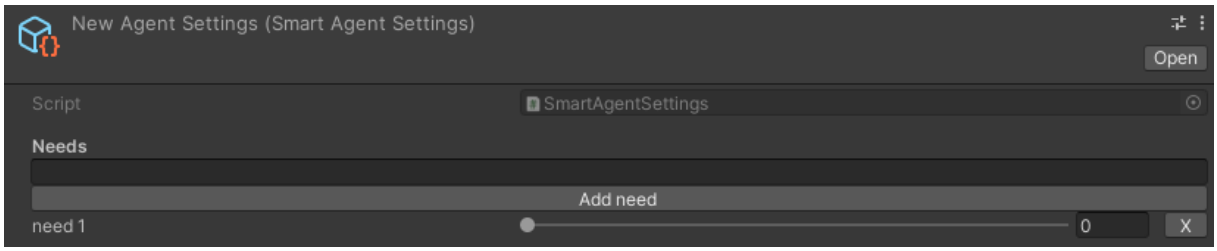


Fig. 4.3 Asset de configuración de agentes inteligentes

Internamente estas necesidades se modelan como un diccionario de pares clave-valor, en los que la clave es un *string* con el nombre de la necesidad y el valor es un *float* entre 0 y 1. Para poder configurar las necesidades más fácilmente, ya que *Unity* no permite serializar o representar diccionarios en el inspector, se ha diseñado un inspector personalizado (ver fig. 4.3) en el que el usuario especifica el nombre de la necesidad, pulsa en *"Add need"* para crearla y después asigna su valor inicial.

Smart Object

Para integrar los objetos inteligentes en *Unity* se ha creado el componente *SmartObject* que implementa la interfaz *ISmartObject*. La clase es abstracta y debe ser el usuario de la herramienta el que cree clases concretas heredando de ella e implementando los siguientes tres métodos, según sus requerimientos:

- *ValidateAgent*: Sirva para decidir si el agente que quiere interactuar con el objeto puede hacerlo. El método devolverá *true* si el agente es válido o *false* en caso contrario.
- *GetCapabilityValue*: Este método permite comprobar la capacidad del objeto de cubrir una necesidad concreta.
- *RequestInteraction*: Una vez que el agente ha sido validado, se solicitará una interacción, que consiste en una acción que al completarse aplica las capacidades del objeto al agente.

Además de la clase principal, se han creado dos subtipos de *SmartObject* que facilitan su implementación (ver fig. 4.4). Ambos usan un nuevo tipo de componente llamado *SmartInteractionProvider* que se encarga de generar interacciones estáticas, es decir, cuya acción y capacidades no varían en función de parámetros del objeto. Para implementar un *SmartInteractionProvider* solo hay que crear una clase que herede de ella e implementar el método *GetInteractionAction* para que devuelva la acción deseada. Las necesidades que cubre la interacción pueden especificarse directamente en el inspector.

Los subtipos de *SmartObject* son los siguientes:

- *SimpleSmartObject*: Proporciona siempre la misma interacción independientemente de los datos de la petición usando un *SmartInteractionProvider*.
- *CompoundSmartObject*: Proporciona una interacción por cada necesidad que pueda cubrir. Para ello define un diccionario de pares clave-valor en el que las claves son los nombres de las distintas necesidades y los valores son *SmartInteractionProviders*. Se debe definir cuál es la interacción que se utilizará en caso de que la necesidad especificada en la petición no coincida con ninguna que el objeto cubra.

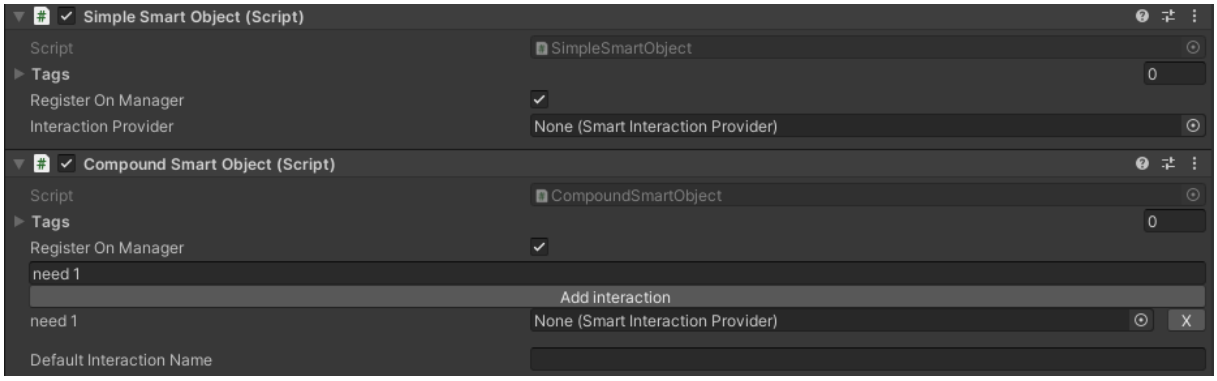


Fig. 4.4 Inspector de los componentes SimpleSmartObject y CompoundSmartObject

En ambos componentes, el método *ValidateAgent* devuelve siempre *true*, y para cambiar este comportamiento es necesario crear una subclase que sobrescriba el método.

Con el objetivo de facilitar la búsqueda y manejo de los *SmartObject* en la escena, se han creado dos clases que permiten gestionar un conjunto de *SmartObject* y realizar consultas sobre ellos:

- *SmartObjectManager*: Se trata de un componente *singleton*, es decir, único en el entorno de ejecución y accesible desde cualquier punto del código mediante una propiedad estática o instancia. La clase *SmartObject* usa esta clase para registrarse y eliminarse de una lista interna que almacena todos los objetos de la escena. Por defecto, los objetos se registran cuando son habilitados y borrados cuando son deshabilitados, en los eventos *OnEnable* y *OnDisable* respectivamente, pero también es posible hacerlo manualmente, por ejemplo, para eliminar el objeto de la lista mientras se esté usando.
- *SmartObjectLocator*: Este componente va ligado a cada uno de los agentes y busca los *SmartObjects* que estén a cierta distancia de él. El script permite configurar la distancia máxima a la que se buscarán, y también si la búsqueda se realizará periódicamente en el evento *Update* o solo cuando el usuario lance el método manualmente. En el primer caso, también se especifica el intervalo de tiempo para buscar los objetos, que por defecto es de un segundo, ya que esta operación puede ser costosa y afectar al rendimiento.

RequestAction

Las *RequestActions* son el mecanismo que usan los sistemas de comportamiento para acceder a los *SmartObjects*. El *toolkit* incluye un tipo especial de acciones llamado *UnityRequestAction* específico para el interactuar con los *SmartObjects* de *Unity*, además de ser compatible con la herramienta del editor visual.

Se incluyen varios tipos de *UnityRequestAction*:

- *TargetRequestAction:* Permite definir directamente el *SmartObject* que usará la acción.
- *RandomRequestAction:* Selecciona un objeto aleatorio de la escena.
- *NeedRequestAction:* Selecciona un objeto aleatorio que cubra una necesidad concreta.

Las dos últimas acciones requieren que exista un componente *SmartObjectManager* en la escena, ya que utilizan los objetos registrados en dicho componente.

4.2 Editor visual de grafos de comportamiento

Esta sección detalla la parte de la herramienta que permite diseñar los grafos de comportamiento usando un editor visual, así como todo el código necesario para integrar los sistemas creados con la herramienta con el resto de las funciones. Se ha dividido todos los scripts y assets en tres partes:

- *Framework*: Engloba todas las clases utilizadas para poder manipular los datos de los sistemas de comportamiento desde el inspector de Unity y desde el editor de grafos. Estas clases no están pensadas para ser usadas por los usuarios de la herramienta, sino únicamente para facilitar la implementación del resto de elementos de la API.
- *Runtime*: Clases y scripts destinadas al usuario de la API.
- *Editor*: Código y archivos necesarios para generar la ventana del editor y las funcionalidades que se incluyan. Incluir todos estos elementos en una carpeta llamada "*Editor*" permite que *Unity* excluya su contenido en la *build* final.

4.2.1 Modelo de serialización de grafos de comportamiento en Unity.

Uno de los principales problemas a la hora de crear una herramienta para modificar los sistemas de comportamiento de la *API* fuera de la ejecución del código, es adaptarla al modelo de

serialización de *Unity* [21]. Por defecto, los grafos de comportamiento no se pueden serializar debido a varios aspectos:

- Polimorfismo: Esta característica del lenguaje C# y de otros lenguajes orientados a objetos es usada en los sistemas de comportamiento para crear grafos, nodos, acciones y percepciones. Unity ofrece un sistema para serializar variables polimórficas utilizando el atributo *SerializedReference*, pero usarlo implicaría modificar el código existente de la API de C#, por lo que se requiere otra solución.
- Referencias cíclicas: Cada nodo tiene referencias a otros nodos para indicar sus conexiones. Referenciar objetos de la misma clase dentro de una instancia impide que ésta se pueda serializar directamente ya que podría provocar un desbordamiento de pila si algún nodo se referencia a sí mismo, además de almacenar información redundante.

El primer paso para integrar la *API* de sistemas de comportamiento es definir un modelo de clases que permita solventar estos problemas. Antes de llegar a la versión final se estudiaron varias opciones.

La primera alternativa fue guardar los datos en formato *JSON*. Este formato permite serializar prácticamente cualquier modelo de clases con la implementación adecuada y es la opción escogida por múltiples herramientas similares, pero por desgracia, supone varios problemas a la hora de trabajar en *Unity*. Para empezar, los datos se guardarían como un único string que habría que interpretar a la hora de modificar los datos. Esto implicaría aumentar mucho el coste de implementar el editor personalizado, ya que no sería posible usar la API del inspector de *Unity* directamente. Además, se requeriría demasiado tiempo de desarrollo en implementar un serializador a formato *json* que fuera lo suficientemente eficiente como para poder serializar y deserializar los datos mientras se modifican en el inspector. En definitiva, esta opción se descartó por el alto coste y complejidad en su desarrollo.

La siguiente alternativa fue representar cada elemento del sistema (es decir grafos, nodos, acciones y percepciones) como *ScriptableObjects* [20], un mecanismo de *Unity* para guardar datos sin que dependan de un componente concreto de un objeto de la escena, ya que se guardan como un archivo en los assets del proyecto. *Unity* serializa los ScriptableObject de una forma específica, generando un identificador único para cada instancia sin importar su tipo concreto. De esta forma se evita la replicación de datos y la gestión de referencias si se elimina el asset queda a cargo del propio motor, por lo que se resolvería tanto el problema de guardar las conexiones entre nodos como el de las referencias polimórficas.

Aunque esta solución sería correcta para un modelo más simple, presenta varios problemas para este caso concreto. Para empezar, no sería posible representar las jerarquías de elementos correctamente (acciones dentro de nodos, nodos dentro de grafos, etc), ya que *Unity* solo permite assets anidados en un nivel. Además, como los *ScriptableObjects* son archivos independientes de la escena, instanciar varios objetos de un mismo *prefab* que haga referencia a un sistema de comportamiento provocaría un error al ejecutarse, ya que ambos objetos harían referencia al mismo sistema.

Tras estudiar estas dos alternativas, se ha diseñado una solución que ofrece la flexibilidad de la serialización en formato *json* con la simplicidad de los *Smart Objects*. Esta solución consiste en crear una jerarquía de clases serializables que actúen como envoltorio de las clases de la *API* base. De esta forma la clase envoltorio será responsable de serializar los datos que las clases base no puedan, a través del atributo *SerializedReference*. A continuación, se detallan las diferentes clases:

***NodeData*:** Clase envoltorio para los nodos. Esta clase almacena los siguientes datos:

- Nombre del nodo: Sirve para identificar rápidamente al nodo en el editor y también para poder acceder a él desde código una vez que se ha creado.
- id: Identificador único para ese elemento del grafo, generado a partir de un *Guid* o *Global Unique Identifier* [22], un valor único de 128 bits. Es necesario ya que el nombre puede ser modificado por el usuario y, por tanto, repetirse.
- Nodo: Referencia a la instancia del nodo. Anotando la propiedad con el atributo *SerializedReference* se guardan todos los campos del objeto, así como su tipo, de forma que *Unity* puede guardar y recrear el objeto sin problemas.
- Listas de nodos padres e hijos: Para representar las conexiones entre nodos, en cada clase envoltorio se guardan dos listas con los identificadores de los nodos hijos y padres. Esto solucionan los problemas al serializar referencias a nodos, ya que se usa su identificador en lugar de la referencia completa.

***GraphData*:** Clase envoltorio para los grafos de comportamiento que contiene los siguientes campos:

- Nombre del grafo: Al igual que en los nodos, sirve para identificarlo en el editor y acceder a él en el código.
- Id: identificador único. Se usa a la hora de crear subgrafos.

- Grafo: Referencia a la instancia del grafo de comportamiento. Como en los nodos, usar el atributo *SerializedReference* permite serializar todos sus datos.
- Lista de nodos: Almacena todos los nodos del grafo como una lista de objetos de la clase *NodeData*.

***PushPerceptionData*:** Clase envoltorio para percepciones push. Almacena la lista de identificadores de los nodos a los que apunta.

***SystemData*:** Representa un sistema de comportamiento completo, formado por varios grafos y percepciones push. Almacena una lista de objetos de la clase *GraphData* para almacenar todos los grafos y otra lista de objetos de la clase *PushPerceptionData* para almacenar las percepciones push.

Esta implementación resuelve la serialización de grafos y nodos y crear, eliminar y modificar estos elementos de forma simple, pero aún no es posible guardar información de acciones y percepciones. Este problema es más complejo de resolver que el anterior, ya que no todos los nodos contienen acciones o percepciones. Sería posible añadir un campo acción y percepción a la clase NodeData, pero sería una solución poco flexible, ya que generaría mucha información irrelevante y no se podrían serializar nodos con varias acciones si fuese necesario.

La opción escogida ha sido crear una clase adicional llamada *ReferencedData* y cada instancia de esta clase almacena una propiedad de un nodo que no es serializable por defecto: concretamente acciones y percepciones. Cuando se genera una instancia de la clase *NodeData* a partir de un nodo, se recorren todos los campos públicos de dicho nodo y se genera una instancia de *ReferencedData* si el campo es una acción o una percepción. En dicha instancia se almacena el nombre del campo, su valor y un string con el nombre del tipo completo, que se usa para validación en caso de que se modifique el tipo. Se ha incluido en la clase *NodeData* una lista de objetos de la clase *ReferencedData*. Así, cada nodo solo almacena la información necesaria.

Aunque esto ya permite serializar acciones, falta añadir un mecanismo para serializar los métodos de dichas acciones. Para ello se ha creado una funcionalidad llamada método serializado o *SerializedMethod*, que permite almacenar un nombre de una función y de un componente y usar esos datos para generar un delegado usando reflexión. Para facilitar su uso, si no se especifica el nombre del componente el sistema usará el propio *Script* (*Runner*), ya que es el caso de uso más común.

Se han creado tipos especiales de acciones y percepciones para aprovechar este modelo de serialización:

- *CustomAction:* Clase que permite definir el método que se va a ejecutar en cada uno de los eventos del sistema (*Start*, *Update*, *Stop*, etc.) a partir de métodos serializados. Existe una variante para percepciones llamada *CustomPerception*.
- *SubgraphAction:* permite referenciar al subsistema que contiene la acción mediante el identificador del grafo.
- *AssetSubgraphAction:* Permite incluir un subsistema guardado en un asset.
- *CompoundActionWrapper:* Clase envoltorio para serializar una acción compuesta. Para las percepciones compuestas se usa *CompoundPerceptionWrapper*.

Una vez establecido el esquema para serializar los datos de un sistema de comportamiento hay que establecer como se genera el sistema ejecutable a partir de esos datos. Por un lado, se ha creado la interfaz *IBuildable*, implementada en aquellas acciones y percepciones que requieren datos externos para completar su estado interno. Esta interfaz obliga a implementar el método *Build* recibiendo por parámetro una estructura de datos llamada *BSBuildingInfo*, que contiene información como el componente (*Runner*) que va a ejecutar el sistema de comportamiento, diccionarios para buscar nodos y grafos usando su id, etc.

Cada elemento usa esta estructura para completar su información. Por ejemplo, las acciones personalizadas (*CustomActions*) construyen cada evento a partir de un objeto *SerializedMethod*, y cada uno de estos usa la referencia al componente guardado en la instancia de *BSBuildingInfo* para buscar el componente especificado, imprimiendo un error por consola. Por otro lado, las acciones con subgrafos (*SubgraphAction*) obtienen la referencia al subgrafo buscando en el diccionario de grafos el valor asociado a su variable *subgraphId*.

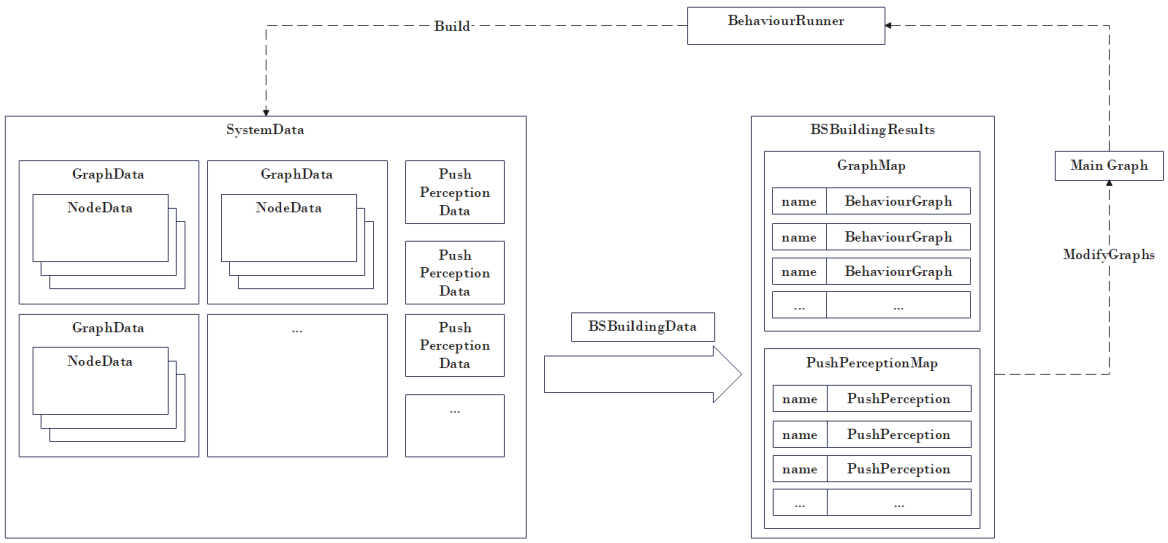


Fig. 4.5 Esquema de generación de un sistema de comportamiento ejecutable a partir de los datos serializados.

Para generar el sistema ejecutable, se ha incluido el método *BuildSystem* en la clase *SystemData*. Este método será invocado desde el componente que contiene el sistema de comportamiento, y lo propagará por todos los grafos, nodos, acciones etc. En lugar de devolver directamente el grafo principal, se devuelve una estructura de datos llamada *BSBuildingResults* (ver fig. 4.5) que permite acceder a todos los grafos y percepciones push usando su nombre, para poder modificarlos por código antes de comenzar a ejecutarlos.

4.2.2 Componentes

El componente *BehaviourRunner* permite crear un sistema de comportamiento implementando el método *CreateGraph*. A partir de dicho componente, se ha creado un subtipo llamado *DataBehaviourRunner*, en la que el sistema de comportamiento se obtiene a partir de una instancia de *SystemData*. Para dar flexibilidad al usuario de la *API* en cuanto a cómo manejar los sistemas creados con el editor, este componente tiene a su vez dos subtipos (ver fig. 4.6):

- *EditorBehaviourRunner*: Este componente almacena todos los datos del sistema de comportamiento directamente en el script. Puede usarse en *prefabs* e incluso crear variaciones de éstos modificando los datos.
- *AssetBehaviourRunner*: En lugar de almacenar el sistema de comportamiento dentro del script, contiene una referencia a un fichero donde se encuentra. De esta forma, varios componentes pueden usar el mismo sistema creando una copia en tiempo de ejecución, pero como esta guardado en los assets del proyecto no admite referencias a componentes de la escena.

Para el segundo tipo de componente, es necesario crear una forma de guardar un sistema de comportamiento dentro de un *asset*, por lo que se ha creado la clase *BehaviourSystem*. Esta clase hereda de *ScriptableObject* y contiene una instancia de *SystemData*.

Con la implementación actual sería necesario crear dos editores distintos, siendo uno para sistemas dentro de un *EditorBehaviourRunner* y el otro para sistemas dentro de un *BehaviourSystem*. La solución a este problema es crear una interfaz común llamada *IBehaviourSystem*, hacer que ambas clases la implementen, y que sea el código del editor el que la utilice para modificar los datos del sistema.

Por cómo funciona la serialización en el inspector de *Unity* es necesario utilizar la referencia al objeto que se usa para contener los datos serializados para aplicar los posibles cambios, usando un mecanismo llamado *SerializedObjects*. Por suerte, tanto los componentes que heredan de *MonoBehaviour* como los *ScriptableObjects* son considerados objetos de *Unity* y heredan de *UnityEngine.Object*.

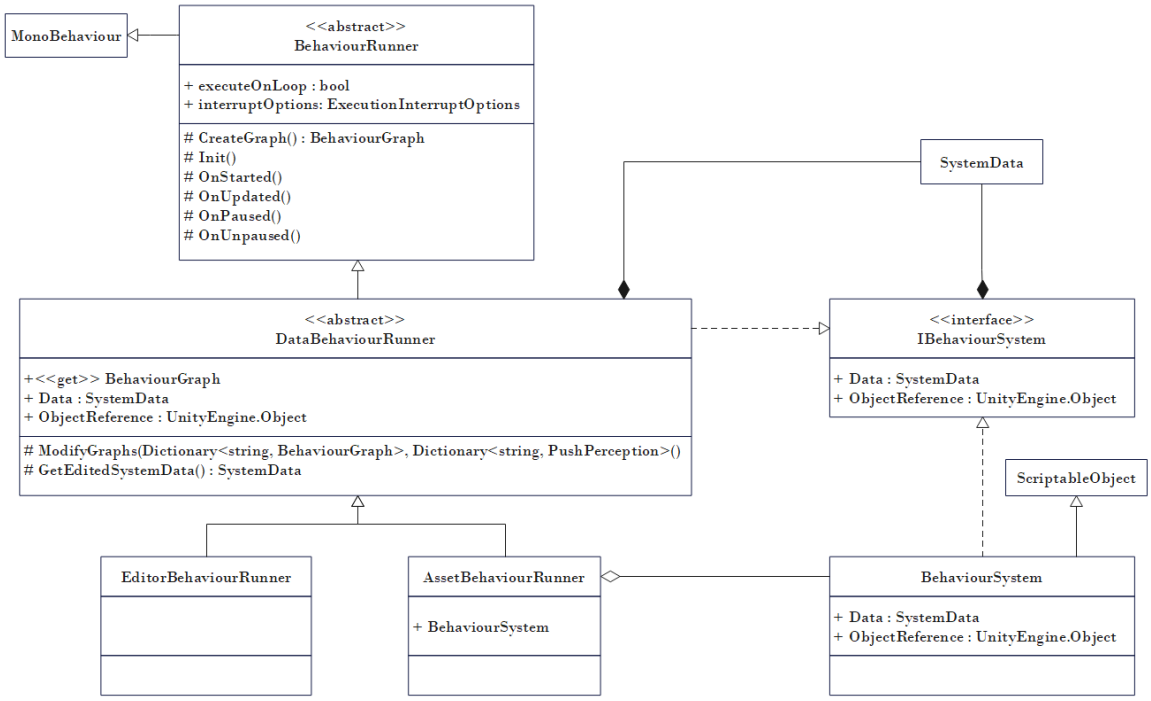


Fig. 4.6 Diagrama de clases de los scripts para el editor visual

Para integrar el editor visual con el sistema de *SmartObjects*, se ha creado un modelo de clases similar al anterior, pero basado en la clase *SmartInteractionProvider* (ver fig. 4.7).

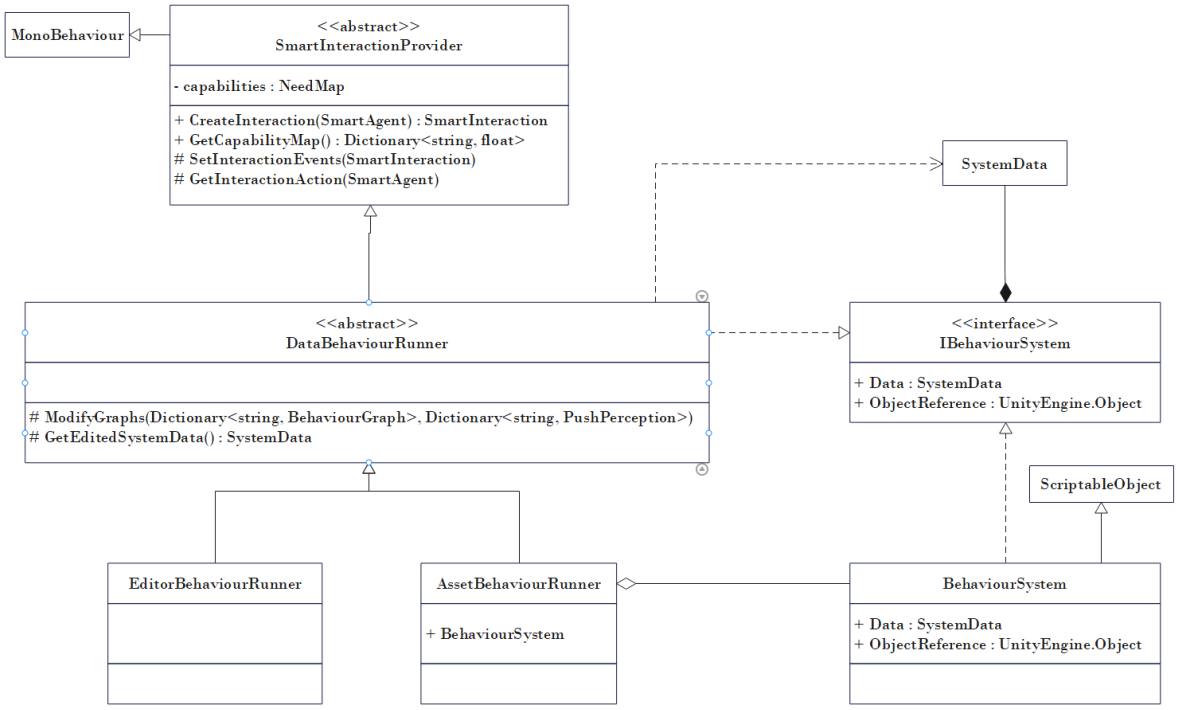


Fig. 4.7 Diagrama de clases de integración de Smart objects con el editor visual

4.2.3 Ventana del editor

La ventana del editor es el elemento principal de la herramienta para crear grafos de forma visual. Para crear tanto la ventana como sus componentes se ha usado el *UI* Toolkit [23], un conjunto de herramientas destinadas a crear extensiones para el editor de *Unity*, aunque también permite crear interfaces directamente para aplicaciones y juegos.

La forma de diseñar las interfaces usando esta herramienta es muy similar a la maquetación web o al diseño de aplicaciones móviles, ya que se basa en dos tipos de archivos: Los que tienen extensión *.uxml* sirven para definir el diseño o distribución de elementos mientras que los que tienen extensión .uss sirven para modificar propiedades del estilo como la posición, color, fuente del texto, etc. Ambos ficheros pueden modificarse directamente desde código o usando el editor de interfaces de Unity.

La ventana del editor se ha implementado mediante la clase *BehaviourSystemEditorWindow*, que hereda de la clase *EditorWindow* de *Unity*. Para poder establecer un sistema de comportamiento concreto a la ventana, se ha definido el método estático *Create*, que recibe como parámetro un objeto que implemente *IBehaviourSystem*. y se han modificado los inspectores de las clases que implementan dicha interfaz añadiendo un botón que ejecuta dicho método.

Para simplificar el manejo de la herramienta del editor, en lugar de crear una ventana por cada sistema de comportamiento, se genera una única ventana que actualiza su interfaz cada vez que se abre un nuevo sistema. Además, se añade la posibilidad de abrir la ventana sin definir ningún sistema (ver fig. 4.8), de forma que se ahorran recursos a la hora de trabajar con la herramienta y se reduce la posibilidad de fallos.

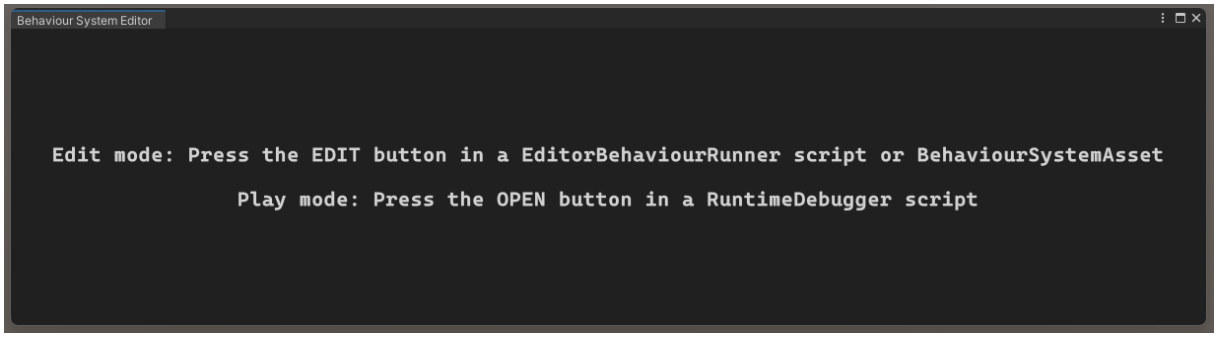


Fig. 4.8 Aspecto de la ventana del editor sin ningún sistema asociado.

Como los *scripts* trabajan directamente con una instancia de *SystemData*, que es serializable en el inspector, la interfaz que utiliza *Unity* para representarlos tiene bastantes limitaciones, como no poder especificar los tipos concretos de grafos, nodos, acciones, etc, además de ser propensa a errores. Se ha creado un inspector personalizado (ver fig. 4.9) para esta clase que en lugar de permitir editar valores muestre información útil como la cantidad de grafos, su tipo y la cantidad de nodos. Para hacer esto, se utilizan *PropertyDrawers* [24], clases que permite cambiar el inspector de las propiedades de un tipo concreto o aquellas que están anotadas con un atributo especificado.

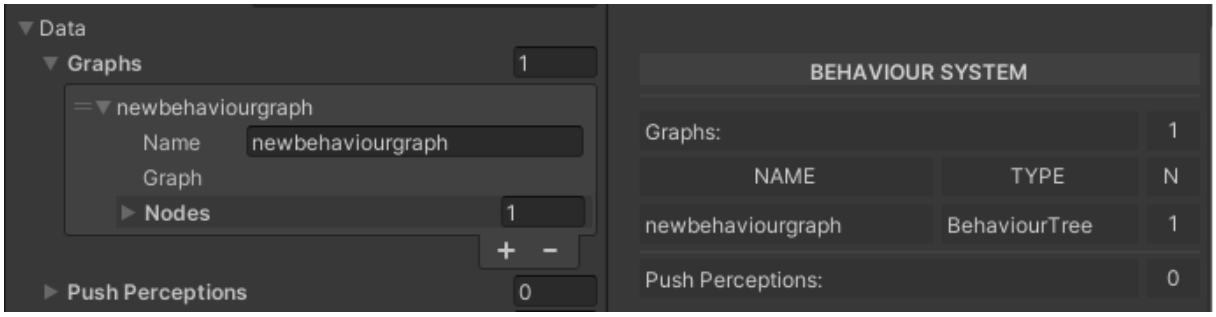


Fig. 4.9 Cambio de representación de los datos de un sistema de comportamiento en el inspector. A la izquierda la versión por defecto y a la derecha la versión modificada.

Al abrir la ventana del editor usando un sistema de comportamiento concreto se muestra la interfaz completa que se divide en tres partes (ver fig. 4.10):

- Barra de herramientas: Incluye controles para añadir y borrar grafos, cambiar el grafo seleccionado, generar código, etc.

- Inspector: Esta sección permite modificar propiedades y variables individuales tanto del grafo seleccionado como de los nodos de dicho grafo. También permite asignar acciones y percepciones a los nodos y añadir nodos a percepciones push.
- Editor de grafos: Muestra la representación visual del grafo de comportamiento. Permite añadir, borrar, mover y conectar nodos.

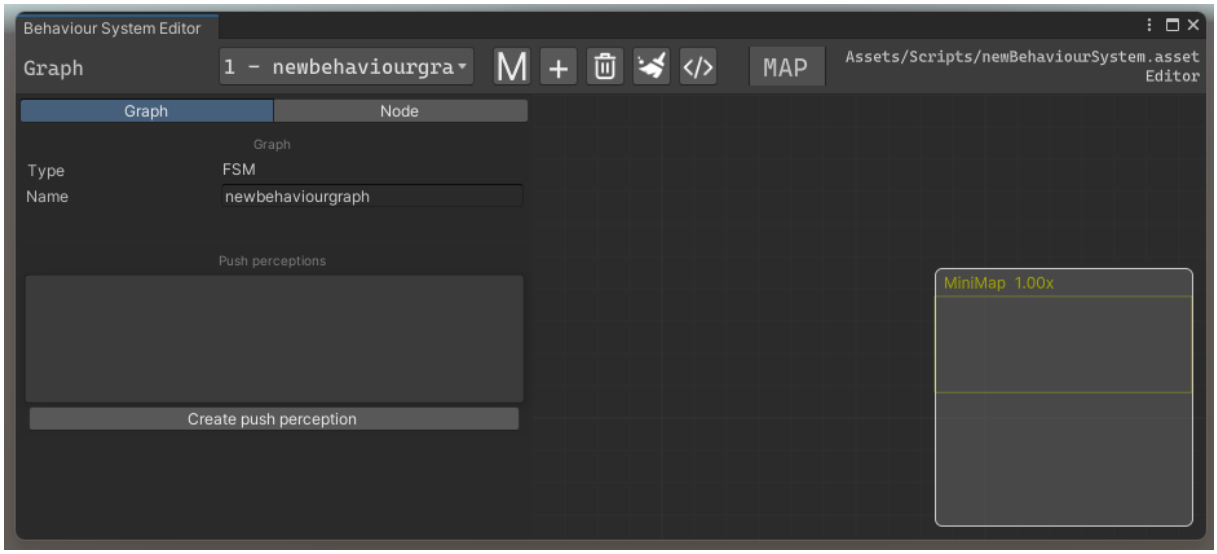


Fig. 4.10 Interfaz de la ventana del editor con un sistema asignado

Barra de herramientas

El primer elemento de la barra de herramientas es el selector de grafo, que consiste en un menú desplegable que muestra todos los grafos que contiene el sistema en orden y permite al usuario cambiar el grafo seleccionado. Al seleccionar una nueva opción se cargará la vista del grafo seleccionado en el editor, borrando la anterior, y se actualizará el inspector.

El botón 'M' sirve para cambiar cual es el grafo principal del sistema. Al hacerlo, se reordenará la lista de grafos del sistema de forma que el grafo seleccionado actualmente sea el primero y después se recargará la interfaz.

El botón '+' abre el panel para crear un nuevo grafo en el sistema (ver fig. 4.11). En el panel se muestran todos los tipos de grafos disponibles y un campo para introducir el nombre del grafo. Al pulsar el botón *"Create"* se creará una nueva instancia de tipo seleccionado encapsulado en una instancia la clase *GraphData* con el nombre especificado y se añadirá a la lista de grafos del sistema. Después se carga el nuevo grafo en la vista. Si no hay grafos en el sistema, el panel para crear grafos estará bloqueado y no podrá cerrarse.

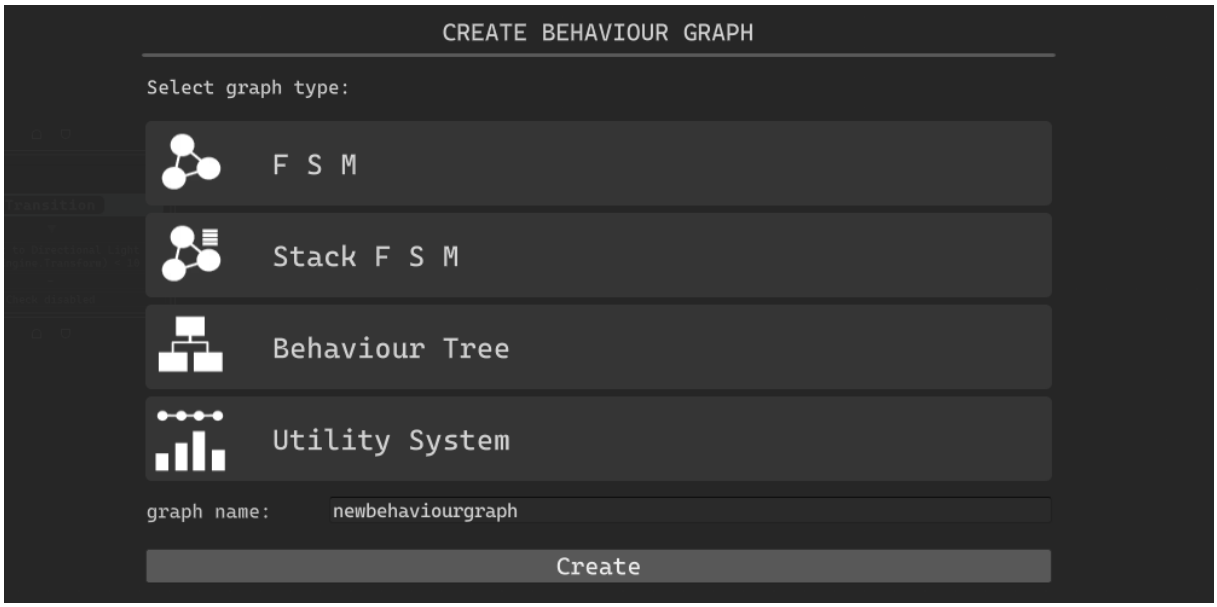


Fig. 4.11 Panel de creación de grafos

El botón con el icono de la papelera permite borrar el grafo seleccionado actualmente. Al hacerlo, se sacará el grafo de la lista de grafos del sistema y se cargará el último grafo en la interfaz.

El botón con el icono del cepillo sirve para borrar todos los nodos del grafo actual. El siguiente botón abre el panel del generador de código y el último permite activar y desactivar el minimapa del grafo.

Varias de estas acciones tienen un efecto drástico en los datos del sistema de comportamiento, por lo que se ha creado una ventana de confirmación para las acciones de borrar un grafo, eliminar sus nodos y cambiar el grafo principal, y de esta forma evitar posibles errores de los usuarios (ver fig. 4.12). Esta ventana se corresponde a la clase *AlertWindow* y funciona de forma similar a la ventana principal, añadiendo el uso del método *ShowModalUtility*, que fuerza al usuario a completar la interacción con la ventana antes de poder interactuar con el resto de los elementos.

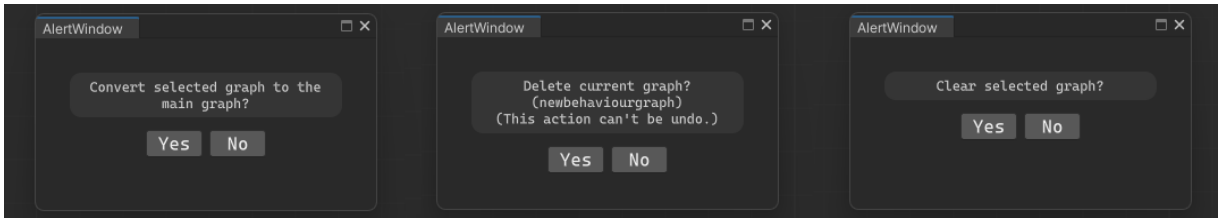


Fig. 4.12 Ventanas de confirmación

Inspector

El inspector es la parte de la interfaz que permite editar las propiedades serializables de nodos y grafos. Está dividido en dos pestañas, la primera muestra las propiedades del grafo actual y permite crear y borrar percepciones push, así como añadir y eliminar nodos objetivos, y la segunda muestra las propiedades del nodo seleccionado (ver fig. 4.13). El editor no soporta selección multiple, por lo que si se seleccionan varios nodos a la vez se mostrará un mensaje de aviso.

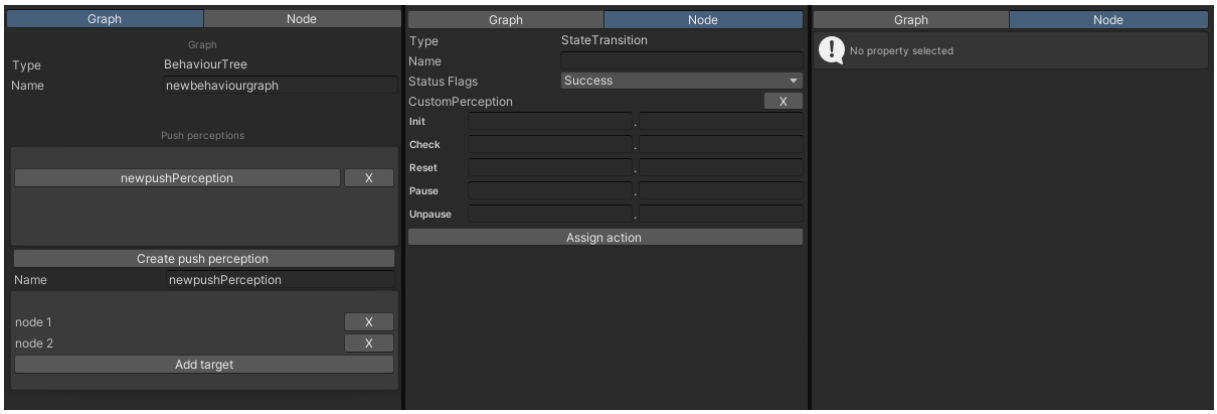


Fig. 4.13 Pestañas del inspector en la ventana del editor de grafos

Aunque con el modelo de datos utilizado es posible serializar las acciones y percepciones de los nodos a través del atributo *SerializedReference*, por defecto las propiedades anotadas solo permiten mostrar las variables del elemento actual, pero no permiten asignarles un elemento de un tipo específico. Para solucionar esto se han creado *PropertyDrawers* para acciones, percepciones y métodos serializados.

En el caso de acciones y percepciones se añade un botón que permite dar valor a la propiedad seleccionando un tipo concreto. Para ello se usa la clase *SearchWindow* que permite generar un menú con todos los tipos disponibles a partir de la jerarquía completa de acciones o percepciones. Esta funcionalidad también se usa a la hora de seleccionar un subgrafo en una acción de tipo *SugraphAction* o un nodo objetivo para una percepción push.

Metadatos del editor

Varias funcionalidades de la ventana del editor se basan en seleccionar un elemento de una jerarquía de tipos. Para definir estas jerarquías de tipos se ha creado la clase *APITypeMatadata*. Cada vez que se recompila la aplicación, se genera una instancia de esta clase que recorre todos los tipos de todos los ensamblados y genera una serie de estructuras con metadatos que representan distintas jerarquías de clases. También permite limitar que tipos están soportados

por la herramienta y cuales no, además de que implementarlo de esta forma hace que los tipos de nodos, acciones o percepciones creados por el usuario también puedan usarse en el editor.

Si el usuario crea muchas acciones o percepciones, encontrar el elemento deseado en el menú de búsqueda puede complicarse. Por ello, se ha creado un atributo *SelectionGroup* que permite definir a que grupo o grupos pertenece un elemento, creando un submenú pera cada grupo y así facilitar la búsqueda.

4.2.4 Editor de grafos

Para construir el editor de grafos se ha usado la API *GraphView*, un conjunto de clases basadas en el *UIToolkit* de *Unity*, que se usan en el propio motor para crear algunas herramientas basadas en grafos, como el editor de *shaders* o de efectos visuales. Aunque no está diseñada específicamente para crear herramientas de terceros y no cuenta apenas con documentación oficial, es suficientemente flexible para este proyecto.

Vista del grafo (*BehaviourGraph*)

Para crear la parte de la ventana del editor que muestra la representación del grafo se ha creado la clase *BehaviourGraphView*. Esta clase hereda de *GraphView*, un tipo concreto de elemento visual destinado a contener elementos como nodos y aristas.

Cada vez que el usuario selecciona un nuevo grafo, sus datos se envían a la vista del grafo para generar su representación, y para que el usuario pueda interactuar y modificar dicha representación es necesario definir una serie de controladores:

- *ContentZoomer*: Permite hacer zoom en la vista del grafo. Se especifica en el constructor el escalado máximo y mínimo del zoom.
- *RectangleSelector*: Permite crear un rectángulo arrastrando con el ratón para seleccionar todos los elementos que queden dentro.
- *SelectionDragger*: Permite mover los elementos seleccionados mediante drag and drop.
- *ContentDragger*: Permite mover la vista del grafo.

Estos controladores permiten modificar los nodos que ya existen, pero falta añadir un mecanismo para crearlos. La clase *GraphView* incluye por defecto un evento que se invoca al pulsar la tecla espacio o mediante el menú contextual al hacer click derecho, destinado a usarse para crear los nodos. Se ha implementado dicho evento para generar una ventana de búsqueda o *SearchWindow* en la que seleccionar el tipo de nodo concreto que se quiere crear (ver fig.

4.14). La jerarquía de nodos se obtiene de los metadatos del editor, especificando el tipo de grafo actual.

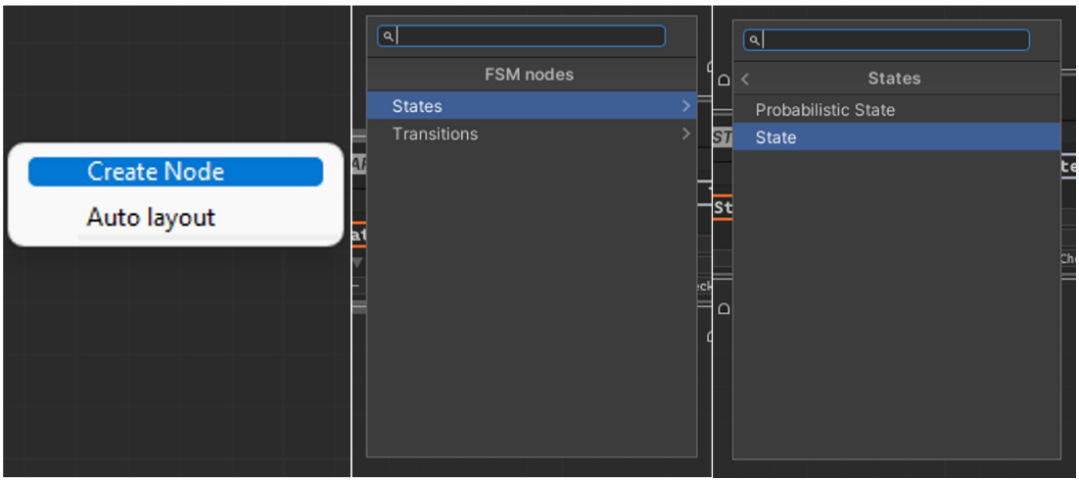


Fig. 4.14 Ventana de creación de nodos

Algoritmos de distribución de nodos (*Layout Handlers*)

En algunos casos, encontrar una distribución de los nodos de un grafo que facilite su comprensión puede ser difícil para los usuarios, especialmente si el grafo tiene muchos nodos. Además, mover los nodos uno a uno puede requerir bastante tiempo.

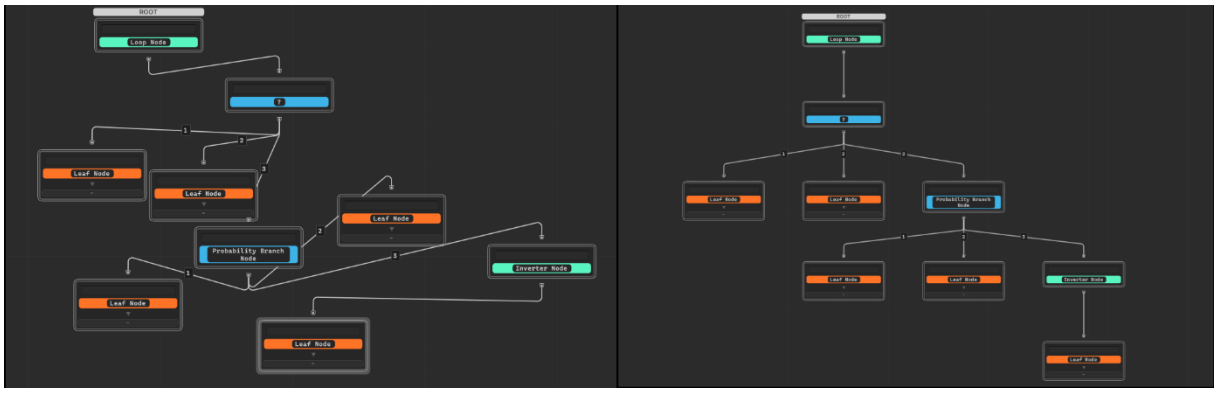


Fig. 4.15 Ejemplo de uso del algoritmo de distribución en un árbol de comportamiento

Para solucionar esto, se incluyen algoritmos de distribución de nodos que el usuario puede utilizar para ahorrar tiempo a la hora de colocar los nodos (ver fig. 4.15). Se han implementado tres variantes, cada uno en una clase independiente:

– *TreeLayoutHandler:* Coloca los nodos en forma de árbol, usando una simplificación del algoritmo de *Reingold-Tilford* [25] que elimina algunas restricciones como que todos los nodos de un mismo nivel estén separados por la misma distancia. Este algoritmo se ha usado en los árboles de comportamiento.

- *LayeredlayoutHandler:* Este algoritmo se ha diseñado para distribuir nodos en grafos dirigidos acíclicos y se ha usado en los sistemas de utilidad. Se basa en calcular la posición de forma que la coordenada x se corresponda a la "altura" del nodo y la coordenada y depende de la distribución de los nodos hijos.
- *CyclicLayoutHandler:* Se trata de una implementación del algoritmo de atracción-repulsión [26]. Este algoritmo se basa en que los nodos generan una fuerza de repulsión entre ellos, que aumenta cuanto más cerca estén, y los nodos que están conectados generan una fuerza de atracción entre ellos.

Los usuarios pueden usar estos algoritmos a través de la opción *"auto layout"* del menú contextual del grafo. Aunque ninguno de estos algoritmos tiene un resultado perfecto, generan un resultado fácilmente comprensible por el usuario.

Representación de nodos (*NodeView*)

Una vez definido el funcionamiento de la vista de grafos es necesario especificar como se representa cada nodo de forma individual. Para hacerlo se ha creado la clase *NodeView*, que utiliza los datos de un nodo concreto para generar una serie de elementos visuales distribuidos en capas (ver fig. 4.16). De la capa más externa a la más interna, los elementos son los siguientes:

- Puertos: Los puertos son elementos que permiten conectar unos nodos con otros, a través de aristas o conexiones.
- Borde de selección: Sirve para resaltar el nodo o nodos que están seleccionados.
- Borde de estado de ejecución: Sirve para mostrar el estado de ejecución en tiempo real.
- Contenido: Muestra la información del nodo, como el nombre o el tipo.

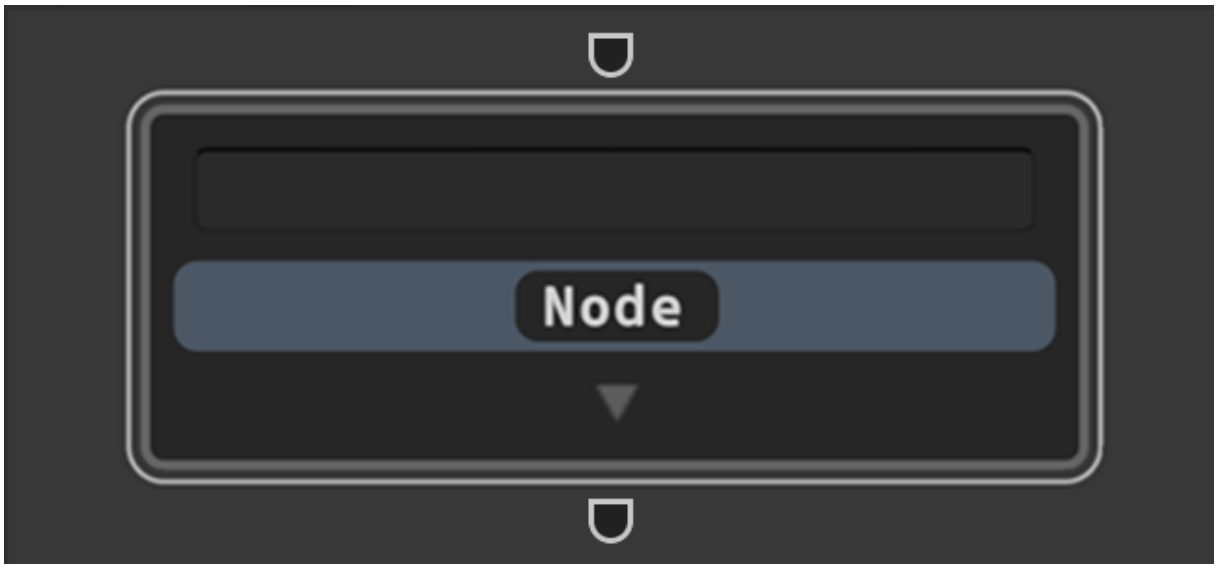


Fig. 4.16 Interfaz de los nodos

La clase *NodeView* reacciona a las acciones del usuario mediante eventos, que pueden ir desde mover el nodo, lo que cambia su valor de posición en los datos, hasta conectar dos puertos, lo que añade sus respectivos identificadores a las listas de nodos padres o hijos del nodo conectado.

Además, para hacer el editor aún más personalizable se ha creado la posibilidad de añadir elementos a la representación del nodo. Estos elementos se crean mediante la clase *ExtensionView*, y cada instancia de esta clase sirve para mostrar una propiedad concreta.

Algunos usos de esta función son mostrar información de las acciones y percepciones que contienen los nodos usando su método *ToString* para generar una etiqueta que muestre el texto en el nodo (ver fig. 4.17). Esto permite al usuario personalizar la representación de cada acción de una forma sencilla sin tener que modificar código del editor.

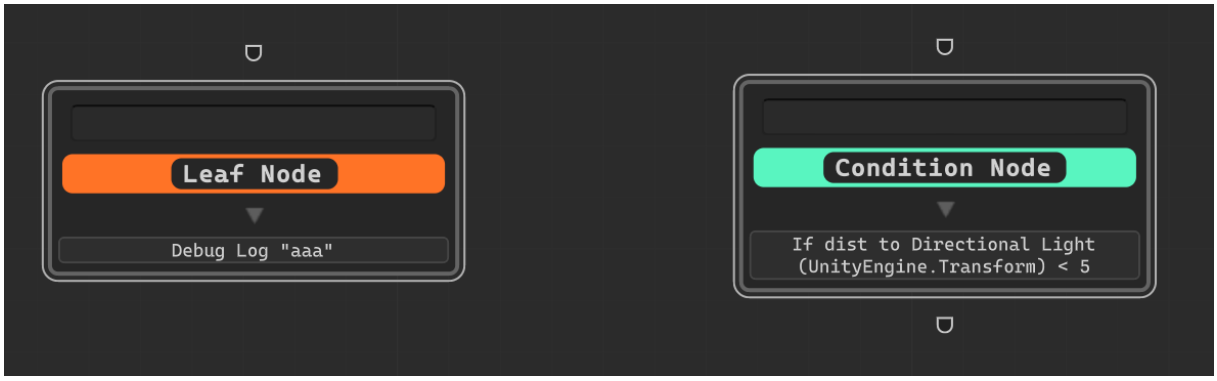


Fig. 4.17 Ejemplo de representación de acciones y percepciones en la vista de los nodos.

Otro elemento importante de la representación de los nodos son los puertos. Por defecto, los usuarios podrían conectar cualquier par de nodos, lo que provocaría errores a la hora de

construir el grafo en tiempo de ejecución. Para evitar esto se han introducido una serie de limitaciones para deshabilitar los puertos que no sean compatibles cuando se intenta crear una conexión, teniendo en cuenta ciertas características como que su dirección debe ser distinta, su tipo debe ser compatible, el número máximo de conexiones no debe sobrepasarse, etc.

Un posible problema a la hora de crear conexiones es que el índice de cada nodo hijo depende del orden en el que el usuario las cree. Al guardar un grafo y volver a abrirlo más tarde, es muy probable que el usuario no recuerde el orden de creación de las conexiones, por lo que se han modificado las aristas para que muestren su índice cuando hay más de una (ver fig. 4.18). Además, se da la posibilidad de ordenarlas en base a la posición en el eje x o y añadiendo una opción en el menú contextual del nodo, según el tipo de grafo.

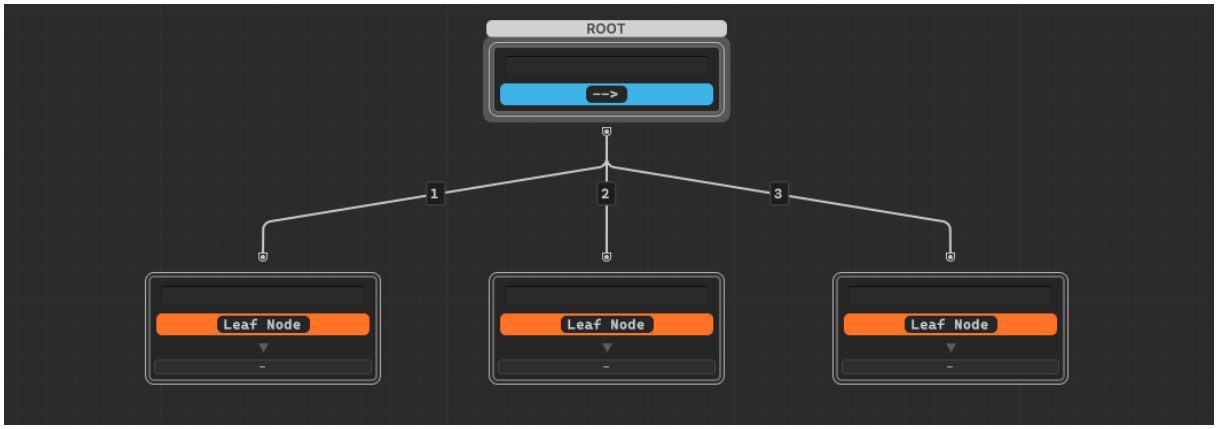


Fig. 4.18 Ejemplo de representación de un árbol de comportamiento en el que se muestra el orden de los hijos.

Decoradores de nodos (*NodeDrawers*)

Hasta ahora, la representación de los nodos es la misma independientemente del tipo de nodo concreto, lo cual puede ser confuso a la hora de diseñar los grafos. Los decoradores de nodos o *NodeDrawers* son clases que permiten modificar la representación de un nodo para un tipo concreto, y para todos los tipos que hereden de él (ver fig. 4.19). Las funcionalidades de esta clase son las siguientes:

- Representar los puertos: Las representaciones de cada tipo de grafo deben adaptarse a las características y limitaciones de sus conexiones. Por ejemplo, las aristas en un árbol de comportamiento van de arriba abajo mientras que las de las máquinas de estados pueden ir en cualquier dirección. Por ello, se ha añadido un método en la clase *NodeDrawer* que permite especificar como se representan los puertos y por tanto las conexiones.
- Representar el tipo de nodo: Por defecto todos los nodos tienen una etiqueta que muestra el nombre de su tipo, pero se ha añadido la posibilidad de establecer también un color asociado

a cada tipo, lo que permite entender más rápidamente la estructura del grafo. Además, en algunos nodos se ha remplazado el nombre del tipo por un icono, como en los nodos secuencia y selector.

– Mostrar el nodo inicial del grafo: En árboles de comportamiento y máquinas de estados, es importante que el usuario sepa cuál es el punto de entrada de la ejecución, por lo que se ha añadido una etiqueta encima del nodo que indica cual es el nodo inicial y se ha añadido una opción en el menú contextual para establecer cualquier nodo como inicial.
– Información adicional: En algunos nodos, los contenedores de extensión se han usado para mostrar más información, además de las acciones y percepciones. Por ejemplo, en las transiciones se muestra cuando se comprobará la percepción en base al parámetro.

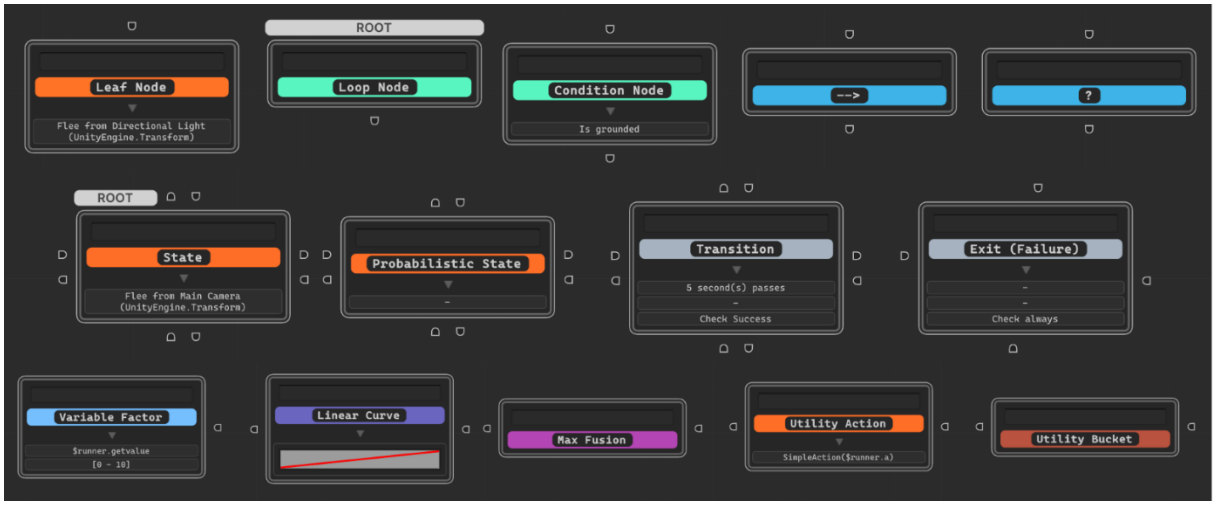


Fig. 4.19 Representaciones de los distintos tipos de nodos.

4.2.5 Depurador en tiempo real

El depurador en tiempo real consiste en varias herramientas que se usan en los sistemas de comportamiento cuando el editor de *Unity* está en *"PlayMode"*. La primera de esas herramientas consiste en modificar la ventana del editor para mostrar los grafos de comportamiento en ejecución (ver fig. 4.20). Para hacerlo se ha añadido un parámetro booleano al método que abre la ventana del editor de un sistema de comportamiento concreto, indicando si el modo es *Editor* o *Runtime*. La versión actual de la herramienta no soporta edición en tiempo real, por lo que en modo *Runtime* se bloquean todos los controles de edición excepto la posibilidad de mover nodos y cambiar el grafo seleccionado, para evitar posibles errores. En cambio, se añaden las siguientes características;

- Se muestra el estado de ejecución actual de los nodos cambiando el color del borde.

- En árboles de comportamiento, se muestra en las conexiones el valor devuelto por la rama concreta cuando ha terminado su ejecución.
- En máquinas de estados, se muestra el valor con el que ha terminado el último estado ejecutado.
- En sistemas de utilidad, cada nodo muestra su valor de utilidad actual.

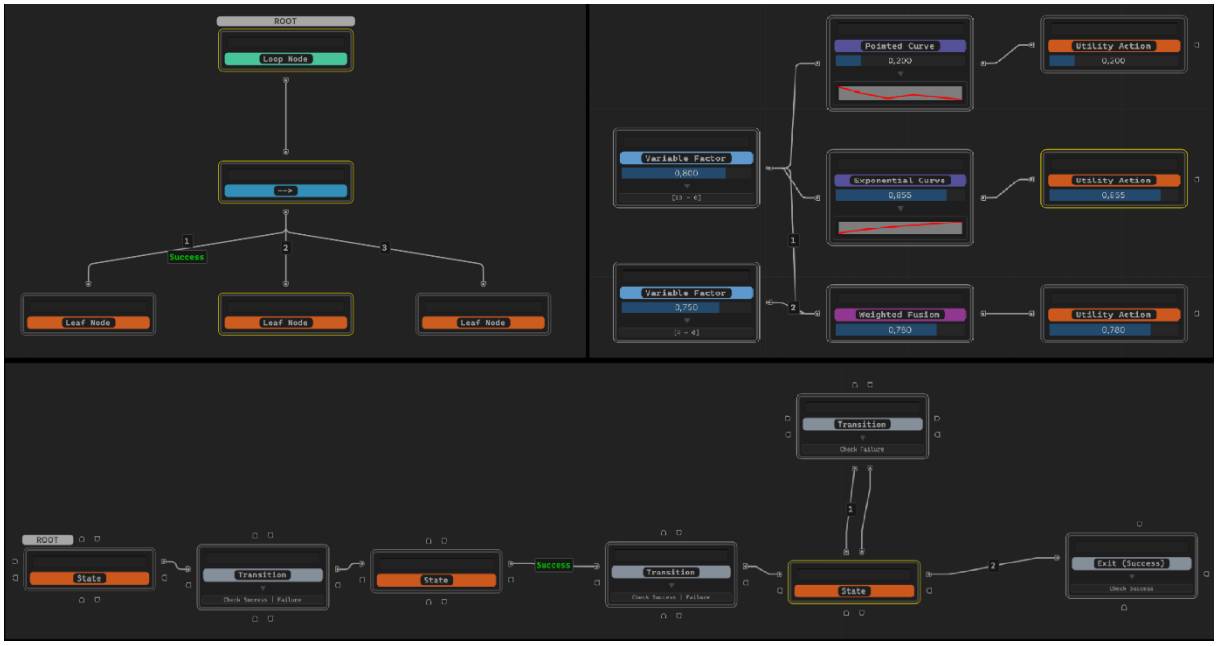


Fig. 4.20 Ventana del editor en modo depuración en tiempo real

Además de los cambios en la ventana del editor, se han creado varias clases que añaden otras funcionalidades.

BSRuntimeDebugger

El componente *BSRuntimeDebugger* permite abrir un sistema de comportamiento generado directamente por código en la ventana del editor, únicamente en modo *Runtime*. Internamente, la clase genera una instancia de *SystemData* y proporciona los métodos *RegisterGraph* y *UnregisterGraph* para añadir y eliminar grafos en el sistema.

BSRuntimeEventHandler

La clase *BSRuntimeEventHandler* se utiliza para gestionar los eventos en tiempo de ejecución de los nodos y grafos de un sistema de comportamiento (ver fig. 4.21). Tanto los componentes que ejecutan un sistema de comportamiento creado con el editor (*DataBehaviourRunner*) como el componente *BSRuntimeDebugger* utilizan internamente una instancia de esta clase.

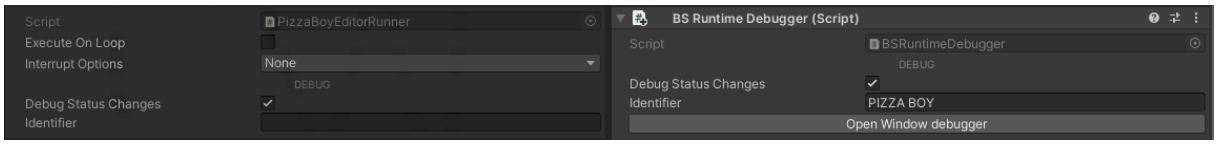


Fig. 4.21 Inspector de la clase BSRuntimeEventHandler

El objetivo de la clase es registrar el evento *StatusChanged* que se lanza en cada elemento que implementa la interfaz *IStatusHandler*. Para ello, se usa el método *RegisterEvents*, que recibe un *grafo* y registra un método que imprime el cambio de estado de ejecución de cada uno de sus nodos en la consola (ver fig. 4.22).

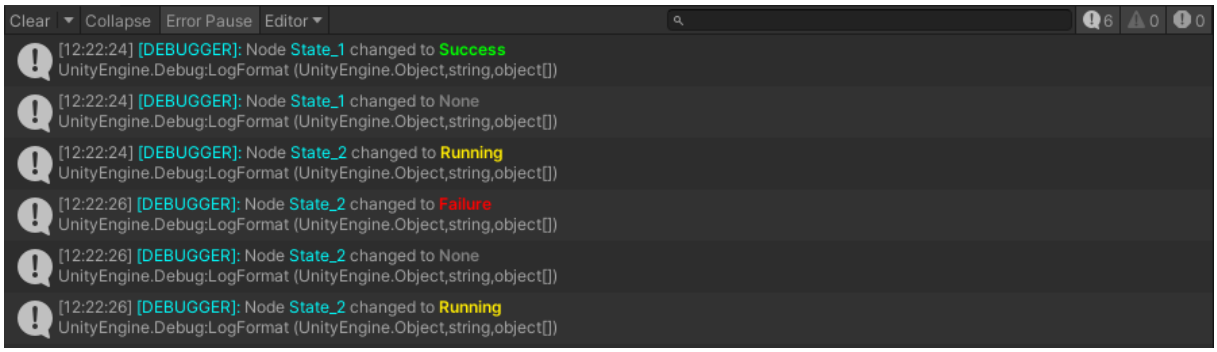


Fig. 4.22 Mensajes por consola del depurador en tiempo real

La clase tiene varios parámetros configurables:

- D*ebugStatusChanges: Flag* que activa y desactiva los mensajes por consola que imprimen los cambios de estado de ejecución.
- *Identifier:* Permite especificar el nombre que aparece en los mensajes por consola para distinguir los agentes de la escena. Si no se define un identificador, el nombre será "DEBUGGER".
- *Context:* Es una propiedad interna de la clase de tipo *GameObject* que referencia al objeto que está ejecutando el sistema. Este objeto se añade como parámetro en los mensajes por consola, lo que permite destacarlo en la jerarquía cuando el usuario hace click en el mensaje.

4.2.6 Generador de código

El generador de código es una funcionalidad incluida en la herramienta que permite convertir un sistema de comportamiento generado mediante el editor visual de grafos en un script.

Para poder convertir los datos de un sistema de comportamiento en código C#, se ha definido una estructura de clases basada en la API de *System.CodeDom* [27], la cual consiste en un conjunto de elementos que definen la estructura del código en un modelo en forma de árbol,

similar a la estructura de un fichero *html*, y después usan este modelo para generar código en distintos lenguajes.

La *API* completa solo está disponible en *.NET Framework* por lo que usarla en *Unity* requeriría configuraciones extra, así que en lugar de usarla directamente se ha creado una versión simplificada con las siguientes clases:

- *CodeMember*: Representa los miembros de una clase. Tiene dos subtipos, la clase *CodeFieldMember* representa las variables globales y la clase *CodeMethodMember* representa los métodos.
- *CodeStatements*: Representan las instrucciones dentro de los métodos. Se han creado varios subtipos para representar distintos tipos de instrucciones (asignación y creación de variables, comentarios, etc).
- *CodeExpressions*: Son los elementos principales de los que se componen el resto de los elementos. Una expresión puede ser desde una referencia a una variable hasta una llamada a un método, etc.

La herramienta incluye varias funcionalidades internas para evitar posibles errores de compilación en el código creado, como la generación de identificadores únicos para las variables. Todos estos aspectos se controlan desde la clase *CodeTemplate*, que es la clase principal del generador de código. Recibe como entrada el sistema de comportamiento y crea la estructura de expresiones para después generar el código de C# (ver fig. 4.23). Algunos aspectos del resultado final pueden personalizarse, como si se utiliza o no el nombre de tipo completo al declarar las variables o se incluye el nombre de cada nodo en el método que los crea.

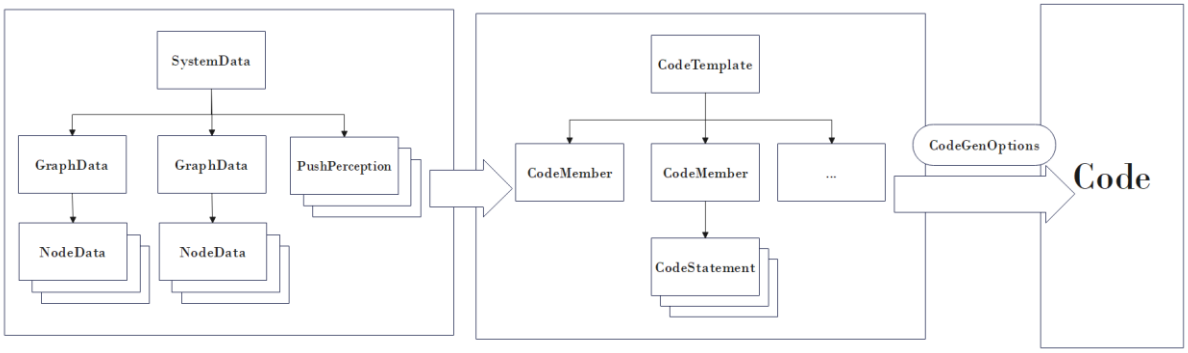


Fig. 4.21 Proceso de generación de código de un sistema de comportamiento

Como cada tipo de grafo de comportamiento usa sus propios métodos para crear los nodos, se ha creado una clase abstracta llamada *GraphCodeGenerator* y una implementación concreta de

dicha clase para cada grafo. Cuando el sistema recorra cada uno de los grafos de comportamiento, creará una instancia del generador de código de grafos correspondiente, que usará las clases definidas anteriormente para crear las estructuras de código.

Para integrar esta funcionalidad en la ventana del editor, se ha añadido un panel que se abre pulsando el botón "</>" de la barra de herramientas (ver fig. 4.24). Este panel permite configurar las opciones del generador de código y muestra una previsualización del código generado para que el usuario pueda ver el resultado sin tener que generar el script y esperar a que *Unity* recompile la aplicación.

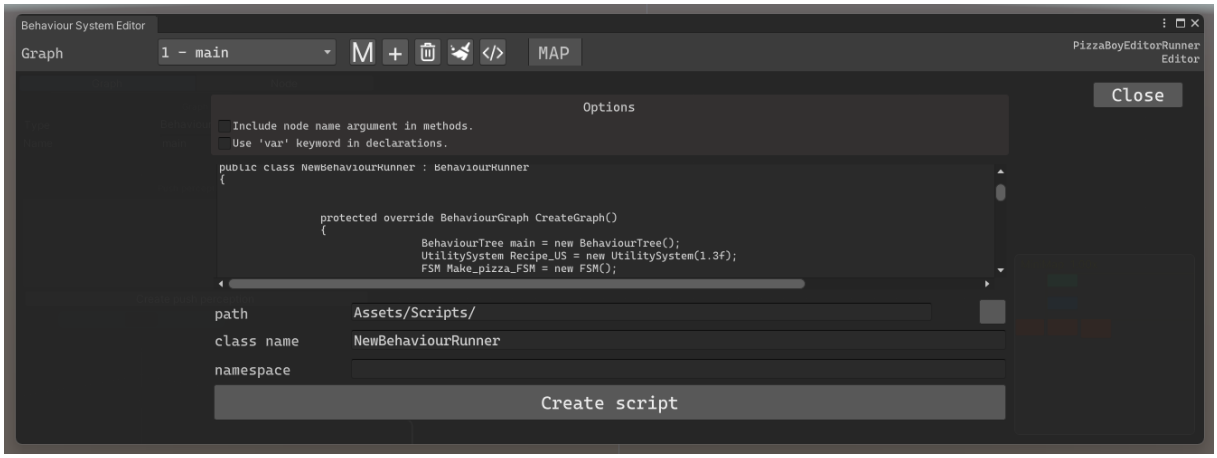


Fig. 4.22 Panel del generador de scripts

5

Pruebas

A lo largo del desarrollo del proyecto se han creado una serie de demos de prueba con varios propósitos. Por un lado, probar las distintas funcionalidades incluidas en la herramienta y tener un elemento de referencia para comprobar posibles errores después de hacer cambios importantes en elementos del código. Al ser escenas de *Unity*, su objetivo también es servir de ejemplo al usuario para conocer los elementos principales y el funcionamiento básico antes de desarrollar sus propios sistemas.

Por último, también han servido para comprobar la compatibilidad del editor visual con los sistemas generados por código y estudiar las posibles limitaciones con respecto a los scripts convencionales. Por esto, se han creado dos variantes de cada demo, una utilizando scripts para generar los sistemas de comportamiento y otra que usa componentes que lo generan usando el editor visual. Esto además servirá al usuario final para comparar ambas versiones y entender cómo se corresponden los distintos elementos.

Es importante mencionar que la versión previa de la librería de C# que este proyecto se encarga de integrar dentro de Unity ya contaba con escenas de prueba, y en lugar de crear las demos de cero, se han reutilizado dichas. De esta forma, aunque el código ha tenido que ser rehecho por completo, la mayoría de los modelos y assets se han podido aprovechar. Además, esto ha permitido tener una referencia del resultado deseado para cada escena de prueba.

A continuación, se desarrollan cada una de las demos, destacando como se han implementado, que elementos pone a prueba y cuáles son los objetivos de cada una.

5.1 Demo 1: Pescador

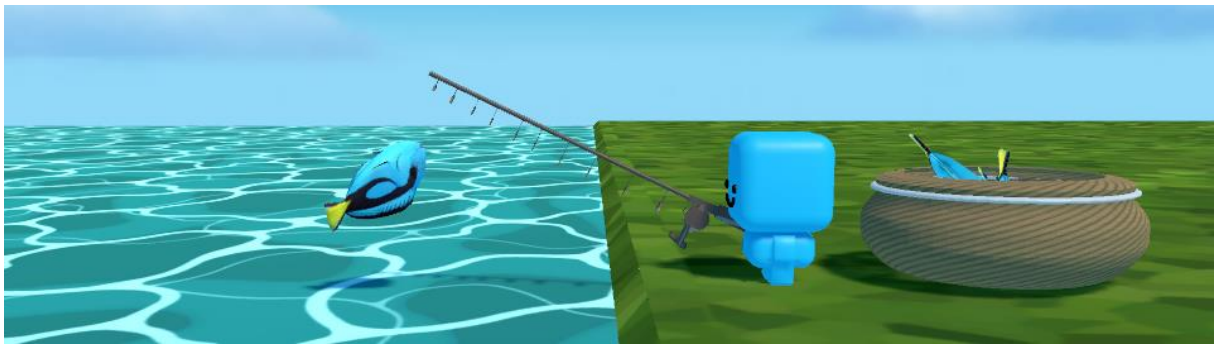


Fig. 5.1 Demo 1

En la escena de esta demo encontramos un personaje pescando en la orilla de un río (ver fig. 5.1). El personaje realiza una secuencia de tres pasos en bucle: Primero lanza el sedal al agua, después espera a que pique algo y por último recoge el objeto atrapado o lo devuelve al agua dependiendo de lo que sea.

El objetivo de esta demo es presentar un ejemplo de sistema de comportamiento que los usuarios pueden replicar fácilmente. Se trata de un árbol de comportamiento formado por un nodo bucle que ejecuta un nodo secuencia con varios hijos, patrón que puede usarse para crear enemigos o NPCs simples, o incluso para animar objetos.

El primer hijo de la secuencia es un nodo hoja que ejecuta una acción personalizada, lo que sirve como ejemplo para que el usuario conozca como crear acciones usando los propios métodos del script. El segundo es un nodo decorador “timer” que espera unos segundos y después ejecuta otro nodo hoja con una acción personalizada (ver anexo 2).

El ultimo es un nodo selector con dos ramas que muestra como dividir la ejecución de un árbol en base a una condición. La primera rama es un nodo condicional que comprueba si el objeto atrapado es un pez y si se cumple ejecuta su una acción para guardar el pez en una cesta. Si no se cumple se salta al siguiente nodo del selector.

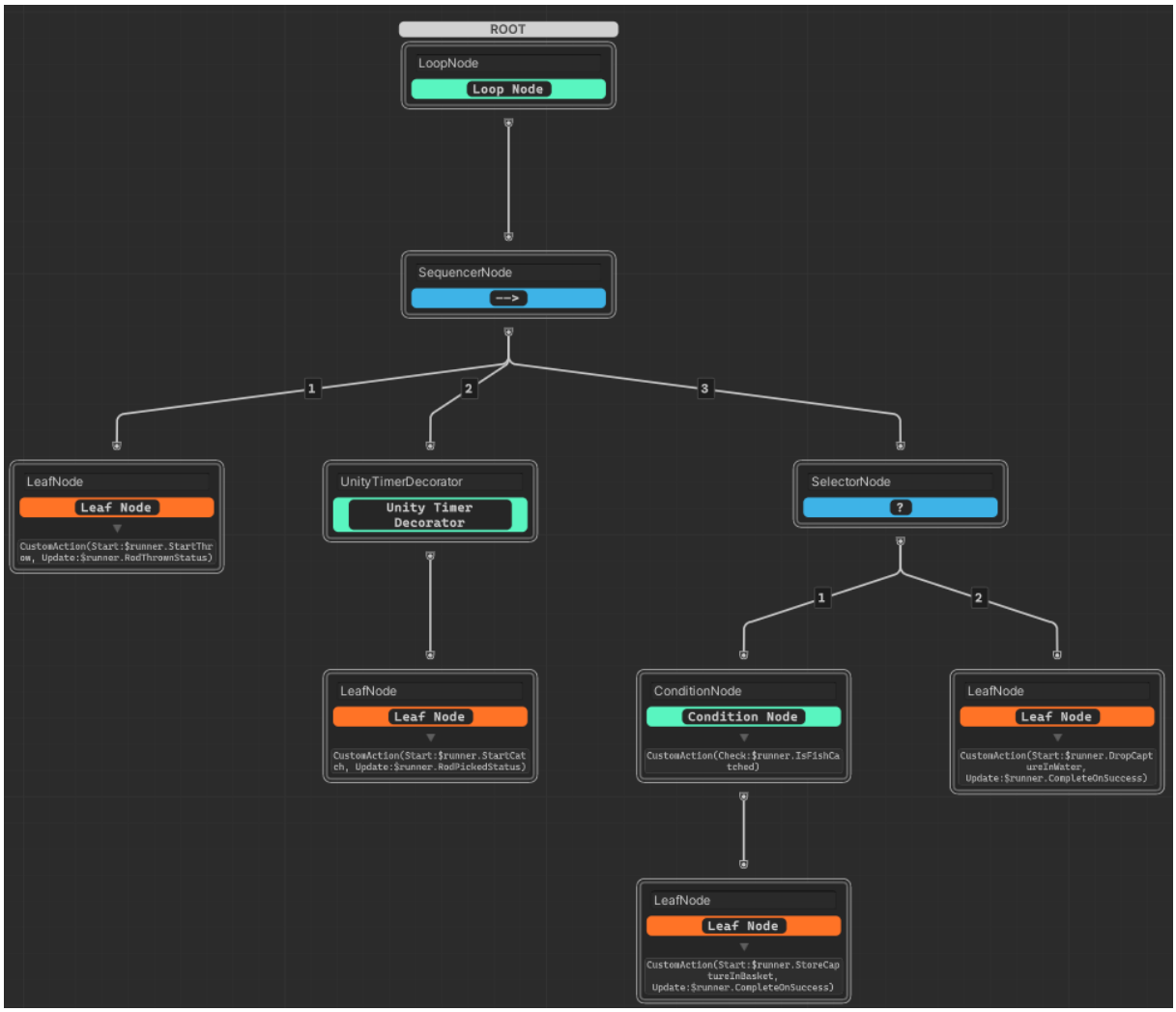


Fig. 5.2 Sistema creado con el editor visual para el personaje de la demo 1

La versión de la demo para el editor visual (ver fig. 5.2) tiene dos propósitos principales. El primero es mostrar cómo se crean acciones y percepciones personalizadas usando métodos del script, especificando el nombre de los métodos en los campos del inspector. De esta forma el usuario puede ver la correspondencia entre las acciones en la versión del editor visual y la versión de código. El segundo propósito es mostrar cómo se indica el flujo de ejecución en el depurador en tiempo real, especialmente como se indica el valor *Status* con el que ha terminado cada nodo.

5.2 Demo 2: El chico y la gallina

Fig. 5.3 Demo 2

Esta escena consiste en dos personajes, un muñeco y una gallina, que se encuentran en un bosque (ver fig. 5.3). El primer personaje es controlable por el usuario, que puede hacer clic en el escenario para moverlo y mientras que el segundo se mueve aleatoriamente hasta que personaje jugable entra en su rango de visión y comienza a perseguirlo, haciendo que el usuario deje de poder controlarlo y huya de él.

Esta demo se ha creado para explicar cómo se crean las máquinas de estados. Ambos personajes se han creado usando una máquina de estados con tres estados. Los dos primeros son *Idle* y *Move*, siendo el primero el estado inicial en el que el personaje no hace nada y el segundo el que usan para moverse. El tercer estado del personaje jugable es el estado *Running* y el de la gallina es el estado *Chasing*. Las transiciones hacia el estado *Idle* se lanzan automáticamente cuando los otros estados han terminado su ejecución.

También sirve como ejemplo de dos sistemas de comportamiento relacionados indirectamente mediante percepciones. Por un lado, el personaje jugable pasa del estado *Idle* o *Move* a *Running* cuando la gallina se acerca demasiado, y por otro, la gallina pasa al estado C*hasing* cuando el personaje jugable esté delante suya a cierta distancia. Ambas condiciones se comprueban usando las percepciones de las transiciones.

Otra característica de la herramienta que usa esta demo son las percepciones push. En el personaje, en lugar de que las transiciones comprueben continuamente si el usuario está haciendo clic, se crea una percepción push que lanza las transiciones hasta el estado *Move* ya sea desde el propio estado, creando una transición cíclica, o desde el estado *Idle*.

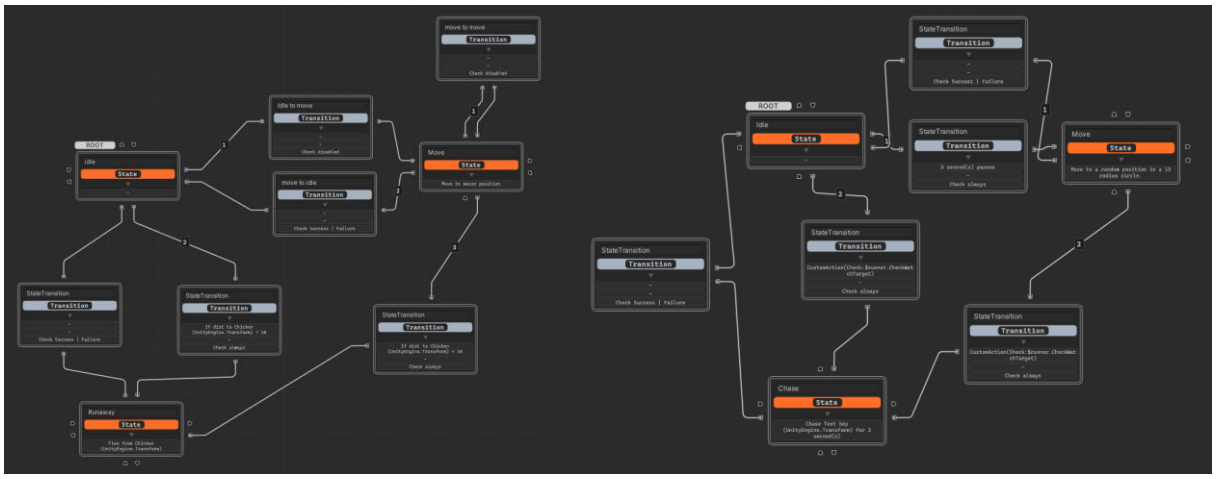


Fig. 5.4 Sistemas de comportamiento creados con el editor visual en la demo 2

También hay que destacar que esta demo ha servido para comprobar la importancia del orden de los nodos objetivos en una percepción push. En la demo, la percepción push activa las transiciones *Move-Move* e *Idle-Move* en ese orden, de forma que si la ejecución está en el estado idle la primera transición no se activa y la segunda sí. Si el orden fuera al revés, se activarían las dos transiciones una tras otra.

En cuanto a la implementación con el editor visual (ver fig. 5.4), esta se centra en cómo crear y editar las percepciones push y como configurar las transiciones para que se lancen automáticamente cuando el estado origen haya terminado de ejecutarse.

5.3 Demo 3: Vuelta en casa

Fig. 5.5 Demo 3

La tercera demo consiste en un personaje que debe entrar en una casa (ver fig. 5.5). El comportamiento del personaje se basa en dos condiciones: si la puerta está cerrada y si hay una llave. Si la puerta está abierta el personaje entra directamente en la casa sin importar si hay o

no llave, mientras que si está cerrada el personaje irá a coger la llave y luego volverá. En caso de que no haya llave el personaje hace explotar la puerta.

La idea de esta demo es crear un árbol de comportamiento más complejo que las demos anteriores en el que el flujo de ejecución dependa de condiciones del entorno. El árbol está formado por una secuencia de tres nodos (ver fig. 5.6), cuyo nodo central es un selector con otros tres hijos correspondientes a las tres posibilidades según la configuración de la escena.

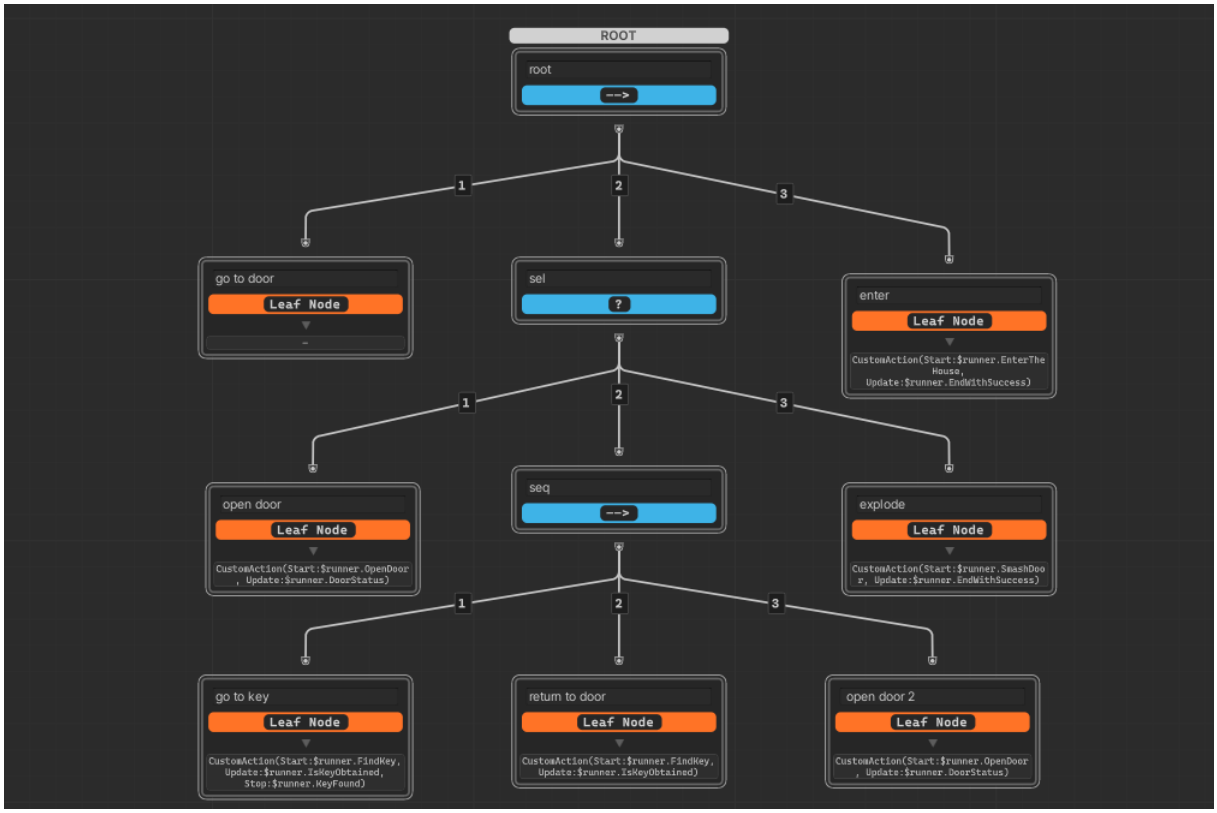


Fig. 5.6 Sistema de comportamiento creado con el editor visual para la demo 3.

Como el personaje es destruido y generado cada vez que se reinicia la escena, este está guardado en un prefab en lugar de en la escena directamente, lo que hace que no puedan usarse referencias a elementos de la escena en el sistema de comportamiento creado con el editor. En el script del personaje se muestra cómo resolver este problema, usando el método *ModifyGraphs* para añadir y modificar acciones y percepciones por código de un sistema creado con la herramienta visual.

5.4 Demo 4: Radar de tráfico

Fig. 5.7 Demo 4

Esta demo consiste en un radar de tráfico colocado en una carretera en la que van pasando coches (ver fig. 5.7). El radar alterna entre dos estados: funcionando y estropeado, y mientras está funcionando indicará si los coches que pasan superan el límite de velocidad o no, cambiando el color de una luz a verde o rojo. Cada cierto tiempo se genera un coche con su propio sistema de comportamiento que se mueve a una velocidad aleatoria. Los coches detectan cuando se estropea el radar y aumentan su velocidad hasta que se arregla.

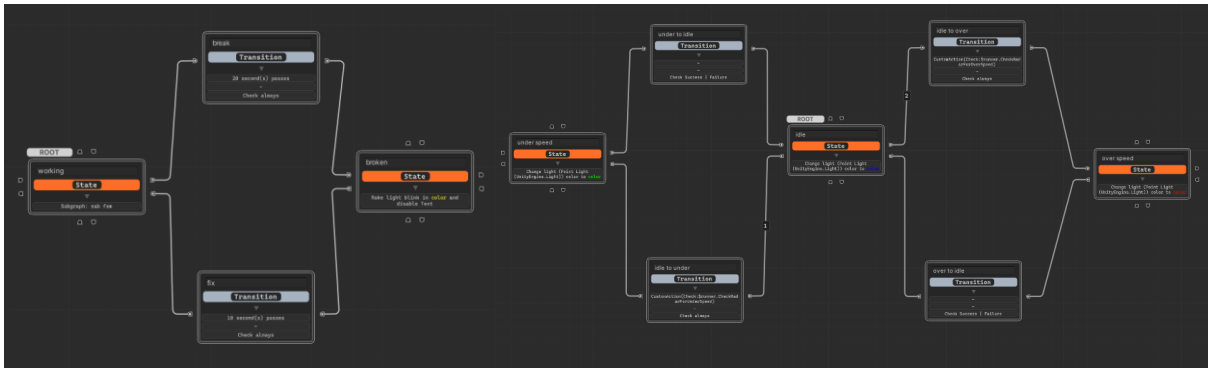


Fig. 5.8 Máquina y submáquina de estados del radar para la demo 4

El sistema de comportamiento del radar es un ejemplo de máquina de estados jerárquica (ver fig. 5.8) en la que la máquina principal se encarga de diferenciar entre estado funcionando y estropeado y la submáquina dentro del estado funcionando se encarga de cambiar el color de la luz usando los estados *Idle*, *OverSpeed* y *UnderSpeed*.

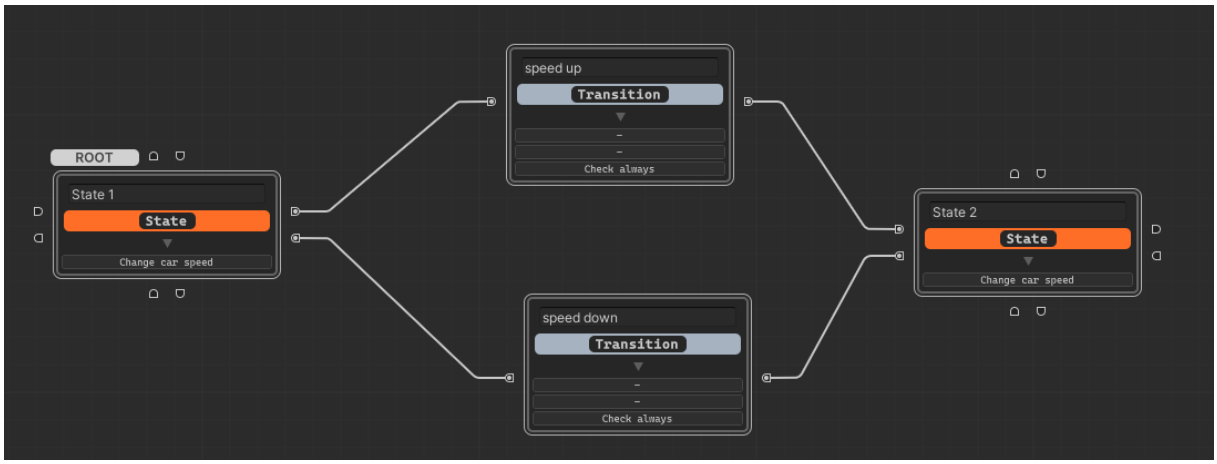


Fig. 5.9 Máquina de estados de los coches para la demo 4

Por otro lado, los coches funcionan con una máquina de estados con dos estados (ver fig. 5.9), uno de ellos para moverse a velocidad normal y el otro para cuando el radar esta estropeado y la velocidad aumenta. El objetivo de este sistema es explicar un caso de uso en el que el estado interno de un sistema de comportamiento afecta directamente a otro, ya que cuando la máquina de estados del radar pasa al estado estropeado, los coches pasan al estado con velocidad aumentada. Para hacer esto se usan las percepciones de tipo *ExecutionStatusPerception* que incluye la API de C#. Además, como este tipo de percepción no está soportado por la herramienta del editor, la versión del editor visual explica como incluirla por código.

Otra característica de esta implementación es que los coches usan un sistema de comportamiento guardado en un *asset*, ya que, aunque son varios prefabs distintos todos funcionan exactamente igual.

5.5 Demo 5: Casa vigilada

Fig. 5.10 Demo 5

La siguiente demo consiste en una variante de la segunda demo, en la que el personaje intenta entrar en la casa. En este caso hay otro personaje que se encarga de vigilar y el personaje original debe intentar entrar en la casa sin ser visto, en cuyo caso huira del personaje interrumpiendo su ruta (ver fig. 5.10).

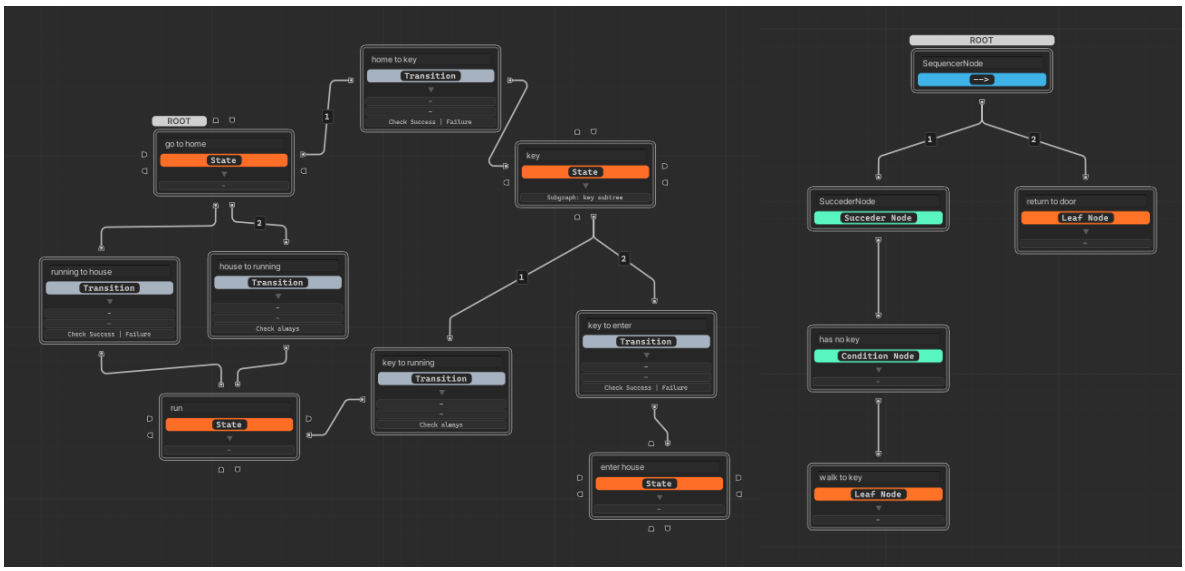


Fig. 5.11 Máquina de estados y subárbol de comportamiento para el personaje de la demo 5.

El personaje usa un sistema de comportamiento jerárquico formado por una máquina de estados y un subárbol de comportamiento (ver fig. 5.11). La máquina de estados tiene cuatro estados: *Go to* home, *find key*, *enter house* y *running*. La ejecución va avanzando por los tres primeros estados cuando sus respectivas acciones son completadas, pero si el personaje enemigo se

acerca demasiado, se salta al estado *running* hasta que el personaje se aleje y vuelva al estado inicial.

Esta demo sirve para ejemplificar que ocurre si el subsistema es interrumpido. El subárbol de comportamiento está dentro del estado *find key* y consiste en una secuencia de dos pasos. El primero es un nodo condición que comprueba si el personaje aún no tiene la llave y si se cumple lo mueve hasta ella. El segundo consiste en hacer que el personaje vuelva a la puerta. La secuencia y la condición permiten que, aunque el árbol sea interrumpido, el personaje no intente coger la llave si ya la tiene.

5.6 Demo 6: Pizzería

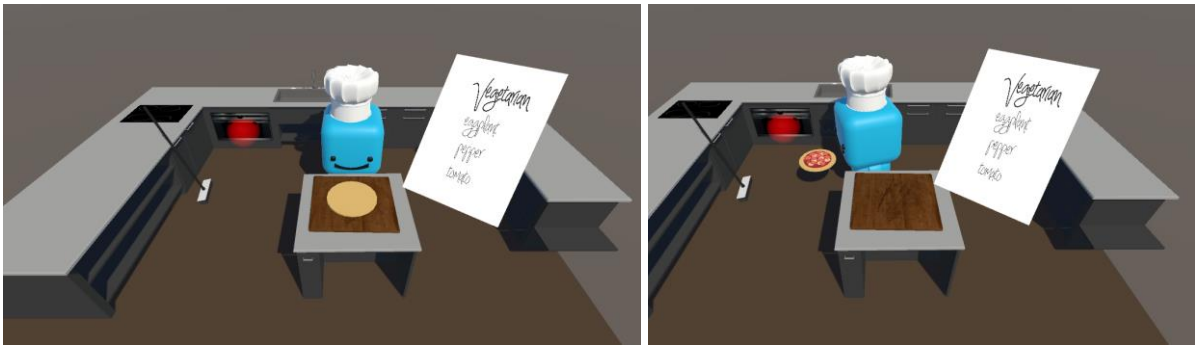


Fig. 5.12 Demo 6

Esta última demo consiste en un cocinero que recibe comandas de pizzas y las prepara siguiendo una serie de recetas (ver fig. 5.12). El sistema del personaje está compuesto por un árbol de comportamiento con una submáquina de estados y un subsistema de utilidad (ver fig. 5.13). El árbol consiste en una secuencia ejecutada de forma infinita y compuesta por tres nodos hoja.

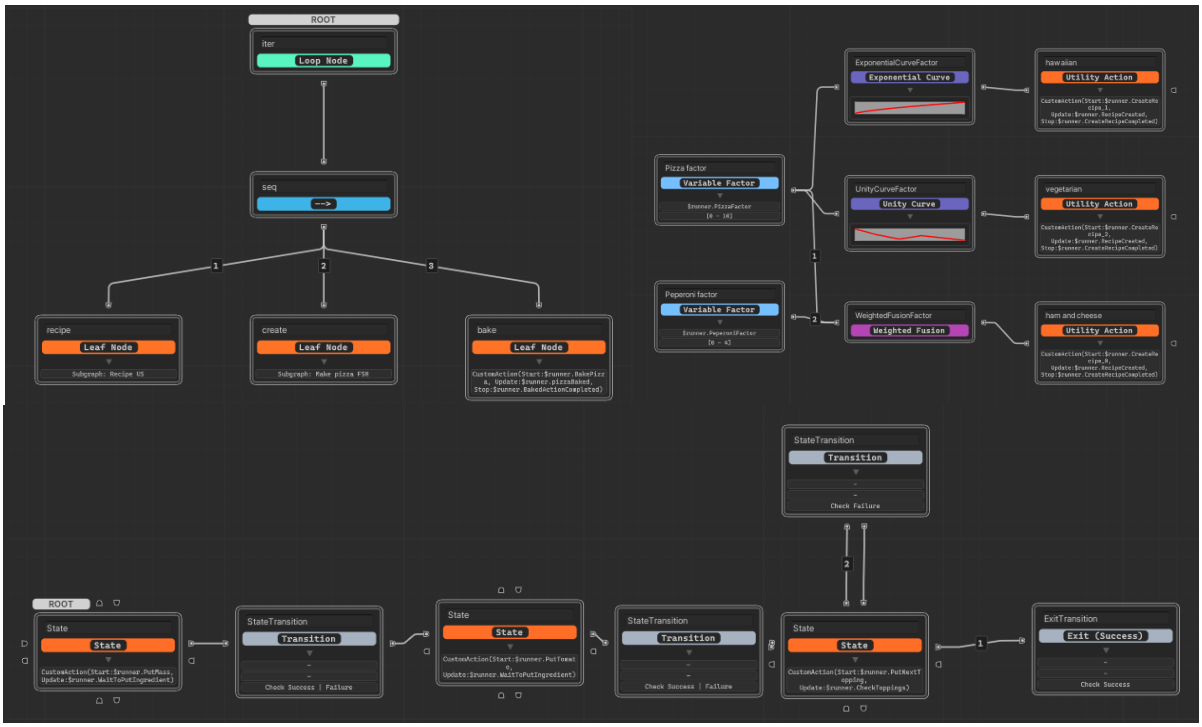


Fig. 5.13 Árbol de comportamiento, subsistema de utilidad y submáquina de e estados para el personaje de la demo 6.

El primer nodo hoja contiene el subsistema de utilidad y sirve para seleccionar la receta que el cocinero debe preparar. El sistema decide entre tres recetas: vegetal, hawaiana o jamón y queso, utilizando dos factores que calculan su utilidad en base a la cantidad de recetas que ha terminado. Para calcular la utilidad de cada acción se usan varios tipos de factores: fusión ponderada, curva lineal, curva exponencial, curva a trozos, etc.

Como los valores de los que depende la utilidad de los nodos solo cambian cuando se completa una receta, el código muestra cómo se puede optimizar el rendimiento desactivando la actualización de utilidad y calculándola solo cuando es necesario.

El segundo nodo contiene la submáquina de estados y cada estado sirve para colocar un ingrediente. Lo destacable de la implementación es que no se usa ninguna percepción para pasar de un estado a otro, sino que las transiciones se lanzan en función del resultado de la acción de su estado inicial, y además es un ejemplo de cómo usar las transiciones para separar la ejecución en función del resultado de una acción. Mientras que los dos primeros estados se encargan de crear la base de la pizza, el último se encarga de poner los ingredientes específicos de la receta. Si después de poner un ingrediente aún quedan más, la acción terminará con failure lo que activará una transición que reiniciará el estado para poner el siguiente, pero si es el último se lanzará una transición de salida para continuar el árbol principal.

El último nodo es simplemente una acción que hace que el personaje coja la pizza y la meta en el horno.

La versión del editor visual muestra como combinar varios grafos de comportamiento de distintos tipos. También sirve para mostrar como el depurador en tiempo real muestra los valores de utilidad de los nodos de un sistema de utilidad.

5.7 Demo 7: Sims

Fig. 5.14 Demo 7

La última demo muestra varios personajes interactúan con *Smart Object*. El escenario es una casa con múltiples objetos inteligentes distribuidos entre varias habitaciones y los personajes los usarán según su modelo de comportamiento (ver fig. 5.14).

Para dar variedad a la escena habrá dos tipos de personajes. El primero ejecuta una acción en bucle que interactúa con un *SmartObject* aleatorio de la escena y el segundo usa un sistema de utilidad para escoger la interacción en base a sus necesidades en cada momento. En cuanto a los objetos, se han implementado un total de 15 para la escena, que se pueden agrupar en varios tipos. El objetivo es mostrar al usuario diferentes formas de crear *SmartObjects* y que pueda entender fácilmente que objeto está usando cada personaje en cada momento, así como mostrar cómo se pueden crear *SmartObjects* enlazados, que es uno de los puntos a destacar de la herramienta.

El primer tipo corresponde a los *SmartObject* simples y engloba a la mayoría de los objetos desarrollados. La interacción con estos objetos siempre consiste en mover al personaje hasta una posición específica, normalmente delante del objeto y luego ejecutar una serie de acciones que pueden consistir en cambiar la pose del personaje, activar un sistema de partículas, etc. La

particularidad de estos objetos es que solo pueden ser usados por un personaje a la vez y para simplificar la implementación de todos los objetos se ha creado una clase común llamada *DirectSmartObject*.

Otro tipo de objetos se corresponde a aquellos que no cubren necesidades por sí mismos, sino que son usados por otros objetos. Dentro de este grupo, los asientos (*SeatSmartObject*) se gestionan de forma independiente al resto y tienen sus propias *RequestActions* para acceder a ellos.

El tercer grupo representa a todos los objetos que usan otros objetos para sus interacciones. Dentro de este grupo, algunos usan objetos específicos y otros seleccionan un objeto de un grupo en base a distancia, aleatoriedad, etc.

6

Conclusiones y trabajo futuro

6.1 Trabajo realizado

A lo largo del proyecto se han desarrollado distintas herramientas y funcionalidades que han permitido integrar una API de creación de sistema de comportamiento en Unity. Estas herramientas se han dividido en dos partes, siendo la primera un conjunto de scripts y clases que facilitan el desarrollo de los sistemas adaptándolos al modelo de programación de Unity, y la segunda un editor de grafos visual para poder crear los sistemas mediante una interfaz y sin necesidad de código.

En cuanto a la primera parte, nombrada *BehaviourAPI Unity Toolkit*, es importante tener en cuenta que se ha desarrollado de forma simultánea a la propia librería de C# que adapta. Esto ha supuesto una gran ventaja, ya que cada nueva funcionalidad en la *API* podía probarse directamente en Unity y así valorar que elementos podían incluirse para facilitar su uso.

Respecto a la herramienta de edición visual, las funcionalidades implementadas han consistido en una ventana del editor de *Unity* para modificar los datos de sistemas de comportamiento. La ventana incluía un inspector para modificar las variables de nodos y grafos, un editor de grafos para crear y conectar nodos de forma visual, un generador de scripts para crear código a partir del sistema editado, y un modo en tiempo real para depurar los sistemas creados.

La mayor parte de problemas durante el desarrollo han surgido a la hora de decidir el modelo de serialización para poder manipular los sistemas de comportamiento en el editor. Durante una parte del proyecto, se usó un modelo basado en *SmartObjects*, pero al probar a utilizarlo en prefabs se encontraron bastantes problemas, por lo que hubo que rehacer el modelo. Por suerte,

gran parte del código desarrollado para la interfaz del editor no requirió cambios y se encontró una solución relativamente simple.

Otro de los problemas en el desarrollo de esta herramienta fue el uso de la API de *GraphView* para crear el editor de grafos. Aunque la API ofrece todas las funcionalidades necesarias, ha sido realmente complicado modificar el diseño por defecto de algunos elementos además de que apenas hay documentación al respecto.

Por último, el desarrollo de las herramientas de depuración en tiempo real no ha supuesto muchos problemas y se han logrado cumplir los objetivos, creando una interfaz que ayude al usuario a crear sistemas de forma más eficiente y evitar errores.

Hay que destacar también que, aunque no era una de las prioridades del proyecto, se han realizado pruebas de rendimiento ejecutando hasta 400 personajes con su propio sistema de comportamiento, tanto por código como por editor, y a pesar de que puede haber una pequeña latencia en la inicialización de los sistemas, apenas afecta al rendimiento medio, siendo mucho más relevante el renderizado de los modelos de dichos personajes.

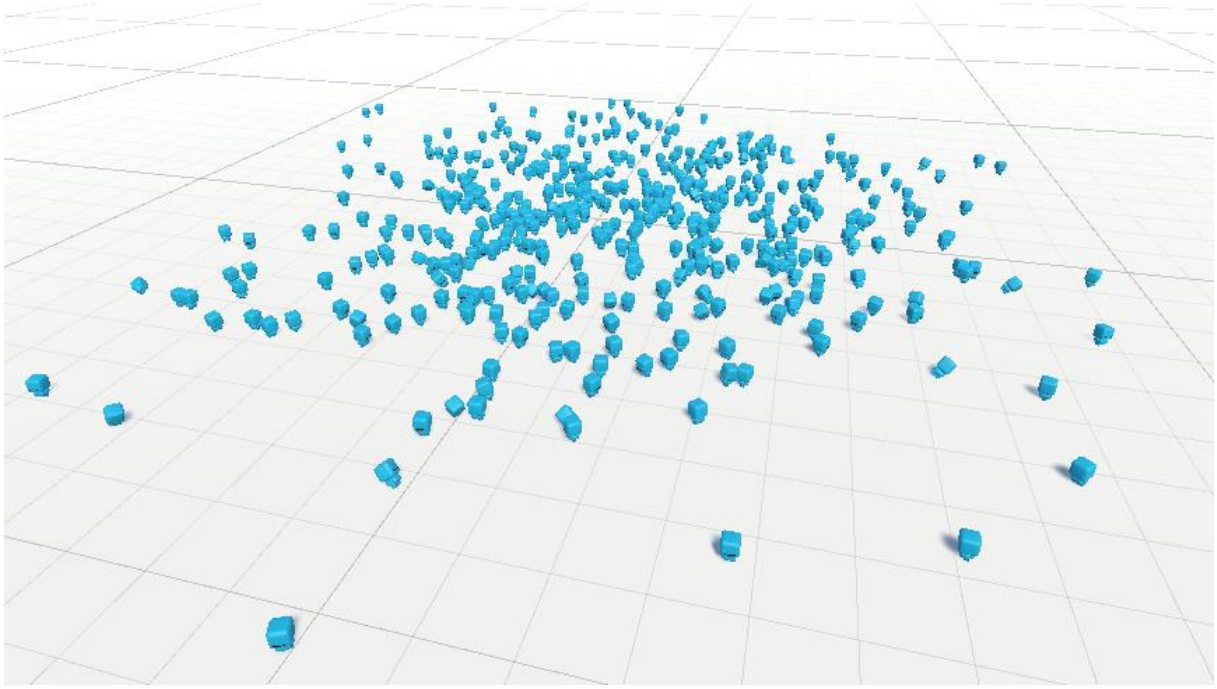


Fig. 6.1 Escena de prueba de rendimiento

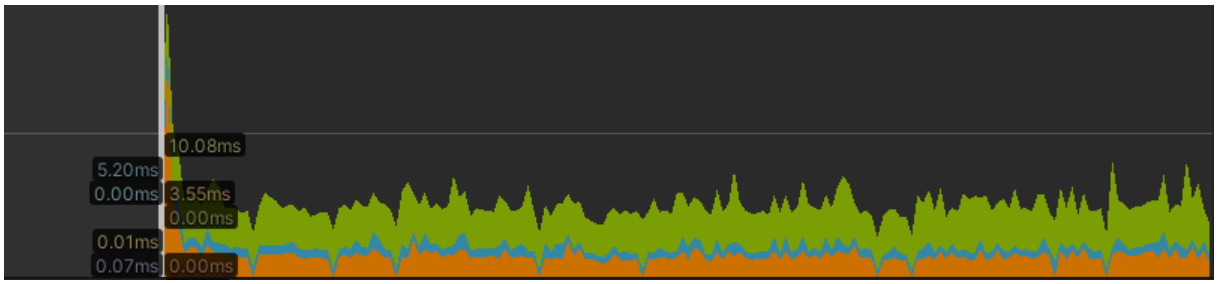


Fig. 6.2 Test de rendimiento en una escena con 400 personajes, cada uno con su sistema de comportamiento.

En la figura anterior se ve una captura del profiler de Unity, en la que el color verde representa el coste del render, el naranja el de las físicas y el azul de los scripts. Puede verse que, exceptuando en la inicialización, los sistemas de comportamiento apenas afectan al rendimiento. Por supuesto, es necesario estudiar cómo funciona la herramienta en casos más complejos y realizar pruebas con usuarios.

6.2 Trabajo futuro

Aunque la cantidad de funcionalidades desarrolladas para el proyecto ha sido bastante grande y la herramienta es completamente usable en su estado actual, es cierto que aún se podrán añadir muchas otras características interesantes. A continuación, se describen algunas ideas para posibles nuevas funcionalidades.

6.2.1 Blackboard

Un elemento muy común en otras herramientas similares, pero que no se ha incluido por complejidad y tiempo requerido, es un sistema de pizarra o blackboard. Una pizarra es un contenedor de variables en el que distintos elementos pueden leer y escribir, y puede ser accesible a distintos niveles, compartiéndose entre un grupo de nodos, un sistema completo o incluso varios sistemas.

6.2.2 Editor en tiempo real

Una limitación importante que tiene la herramienta desarrollada es que no permite modificar los grafos de comportamiento en tiempo real usando el editor, y tampoco permite actualizar su representación visual si se modifica usando código, por lo que sería interesante incluir esta funcionalidad en futuras versiones.

Esto sería especialmente útil en modelos de sistemas de comportamientos que se basan en añadir y eliminar grafos, como algunas variaciones de árboles de comportamiento.

6.2.3 Localización de la herramienta

Actualmente todo el código, comentarios y texto en las interfaces está en inglés, exceptuando los comentarios en los scripts de las demos, que están en español. En el caso del código el idioma no debería ser un problema, pero si sería interesante incluir una herramienta de localización para adaptar el texto de la ventana del editor. Esta tarea no debería ser especialmente compleja, ya que la interfaz es bastante intuitiva y no contiene muchos elementos con texto, pero puede facilitar enormemente el trabajo a los usuarios especialmente mientras no se hayan familiarizado del todo con la herramienta.

6.2.4 Catálogo de acciones y percepciones

Aunque hubiese sido interesante desarrollar un catálogo más completo de acciones y percepciones, esto no se ha hecho por dos motivos. El primero de ellos es el tiempo requerido, ya que se ha considerado que otros aspectos de la herramienta debían tener prioridad. El segundo es la simplicidad de la herramienta, ya que añadir más tipos de acciones y percepciones podría complicar el uso a la hora de buscar un elemento para un caso de uso concreto.

Por todo esto, podría ser una idea a considerar el crear nuevos grupos de acciones y percepciones, pero no incluirlos directamente en la propia herramienta, si no crear paquetes independientes agrupados por temática o funcionalidad que el usuario pueda importar a la herramienta según las necesidades.

6.3 Conclusiones

A nivel personal, aunque el proyecto ha sido todo un desafío debido a su extensión y al poco conocimiento de la materia que tenía antes de empezar el desarrollo, el resultado ha sido bastante satisfactorio. Además, la creación de herramientas para desarrolladores y la inteligencia artificial son dos de los campos dentro del desarrollo de videojuegos que más interesantes me resultan, y en los que me gustaría especializarme en un futuro.

Considero que haber realizado este proyecto me ha aportado las bases necesarias para trabajar en otras herramientas en el futuro, ya sea para *Unity* o para otros entornos. También me gustaría publicar la herramienta en la *asset store* de *Unity* de forma gratuita, para que cualquiera que lo desee pueda usarla en sus proyectos personales o profesionales.

7

Bibliografía y referencias

[1] J. velch (2020). Should the Monster Play Fair?: Reception of Artificial Intelligence in Alien: Isolation. [Online] Disponible en: https://gamestudies.org/2002/articles/jaroslav_svelch?fbclid=IwAR1ETeLfY5s85iCdLyiORqGHz9n0Fb1o922qXTes1z-_jh86nOHi5nmZBUo

[2] Unity Technologies. Unity. [online]. Disponible en: https://unity.com/es

[3] Sam Millington (2022, Nov 23). A solid guide to SOLID principles. [Online]. Disponible en: https://www.baeldung.com/solid-principles

[4] Unity Technologies. Unity visual scripting. [Online]. Disponible en: https://unity.com/es/features/unity-visual-scripting

[5] Epic Games. Blueprint visual scripting. [Online]. Disponible en: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/

[6] Godot Engine. Comenzando con visual scripting. [Online]. Disponible en: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/

[7] Yoyo Games. GML Visual Overview. [Online]. Disponible en: https://manual.yoyogames.com/Drag_And_Drop/Drag_And_Drop_Overview/DnD_Overview.htm

[8] Notion Labs. Your wiki, docs & proyects together. [Online]. Disponible en: https://www.notion.so/

[9] Microsoft. Visual Studio. [Online]. Disponible en: https://visualstudio.microsoft.com/es/

[10] Epic Games. Unreal. [Online]. Disponible en: https://www.unrealengine.com/es-ES

[11] Epic Games. BehaviourTrees. [Online]. Disponible en: https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/

[12] Unity Technologies. Flow states and super states. [Online]. Disponible en: https://docs.unity3d.com/bolt/1.4/manual/bolt-states.html

[13] Eric Beige (2016). Panda BT. [Online]. Disponible en: http://www.pandabehaviour.com/

[14] Paradox notion. Node canvas. [Online]. Disponible en: https://nodecanvas.paradoxnotion.com/

[15] Opsive. Behaviour Designer. [Online]. Disponible en: https://opsive.com/support/documentation/behavior-designer/overview/

[16] Unity Technologies. Order of execution for evento functions. [Online]. Disponible en: https://docs.unity3d.com/Manual/ExecutionOrder.html

[17] Refactoring Guru. Strategy. [Online] Disponible en: https://refactoring.guru/es/design-patterns/strategy

[18] Unity Technologies (2021). Construyendo un NavMesh. [Online]. Disponible en: https://docs.unity3d.com/es/2021.1/Manual/nav-BuildingNavMesh.html

[19] Unity Technologies (2023). Using animation curves. [Online]. Disponible en: https://docs.unity3d.com/Manual/animeditor-AnimationCurves.html

[20] Unity Technologies (2018, Oct). Scriptable Object. [Online]. Disponible en: https://docs.unity3d.com/2021.3/Documentation/Manual/class-ScriptableObject.html

[21] Unity Technologies. Script serialization. [Online]. Disponible en: https://docs.unity3d.com/Manual/script-Serialization.html

[22] Raymond Chen (2008, Jun). GUIDs are globally unique, but substrings of GUIDs aren’t. [Online]. Disponible en: https://devblogs.microsoft.com/oldnewthing/20080627-00/?p=21823

[23] Unity Technologies. Unity UI Toolkit. [Online]. Disponible en: https://unity.com/features/ui-toolkit

[24] Unity Technologies. Property drawer. [Online]. Disponible en: https://docs.unity3d.com/ScriptReference/PropertyDrawer.html

[25] R. Lim (2014, Abr). Algorythm for drawing tree. [Online]. Disponible en: https://rachel53461.wordpress.com/2014/04/20/algorithm-for-drawing-trees/

[26] A. Disney (2021, Feb). Force-directed graph layouts explained. [Online]. Disponible en: https://cambridge-intelligence.com/keylines-faq-force-directed-layouts/

[27] Microsoft (2021, Sept). Using the CodeDom. [Online]. Disponible en: https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/using-the-codedom

8

Anexos

8.1 Anexo 1: Guía de herramienta de sistemas de comportamiento en Unity

GUÍA DE UNITY TOOLKIT

Se trata de un conjunto de herramientas que facilitan la creación y el uso de sistemas de comportamiento en *Unity*. A continuación, se explican los principales elementos que contiene:

BehaviourRunner

Se trata de un *Script* preconfigurado para ejecutar un sistema de comportamiento. En lugar de crear una clase desde 0, es posible crear una que herede de *BehaviourRunner* y así solo es necesario sobrescribir el método *CreateGraph* sin tener que implementar cuando se lanzan los eventos del sistema de comportamiento, ya que el *script* se encarga de asociarlos a los eventos de Unity (*Start, Update, OnEnable*, etc).

El script tiene dos parámetros configurables:

- *ExecuteOnLoop:* Si se activa, el sistema se reiniciará cuando termine su ejecución.
- InterruptOptions: Decide que eventos se lanzan cuando se habilite y deshabilite el script. Por defecto no se lanzará ningún evento.

Importante*:* Los métodos correspondientes a los eventos de Unity, es decir *Awake, Start, Update, OnEnable* y *OnDisable* no deben implementarse en los *Scripts* que hereden de *BehaviourRunner*. En su lugar se deben sobrescribir los métodos de dicha clase (*Init, OnStarted, OnUpdated, OnUnpaused* y *OnPaused* respectivamente)

UnityExecutionContext

El contexto de ejecución es objeto compartido entre todos los nodos, acciones y percepciones de un sistema de comportamiento. Este *toolkit* incluye la clase *UnityExecutionContext* que permite usar esta funcionalidad para que las acciones y percepciones accedan al *GameObject* que contiene el sistema y a sus componentes.

La clase *BehaviourRunner* utiliza por defecto esta funcionalidad, pero si se usa otra clase, se debe crear una instancia de *UnityExecutionContext* pasando al constructor el propio script y después de crear el sistema usar el método *SetExecutionContext* sobre el grafo principal. Si el sistema usa subgrafos no es necesario repetir el proceso en estos.

```
BehaviourGraph graph = …
UnityExecutionContext context = new UnityExecutionContext(this);
graph.SetExecutionContext(context);
```

UnityActions y UnityPerceptions

Se han creado tipos específicos de acciones y percepciones para usar el contexto de ejecución de Unity. Para usarlas se crea una clase que herede de *UnityAction* o *UnityPerception*. Estas clases añaden dos características.

- Contienen una propiedad *context* para poder acceder al objeto de la escena y a su componente. Por ejemplo, es posible modificar la posición del agente usando *context.Transform*.
- Contiene el método *OnSetContext* que se ejecuta una sola vez cuando se propaga el contexto de ejecución por los nodos. Este método puede sobrescribirse en las subclases para guardar referencias a componentes del objeto y así evitar tener que usar el método *GetComponent* cada vez que se necesite ese componente.

```
public abstract class ExampleUnityAction : UnityAction
{
    private MyComponent _myComponent;

    protected override void OnSetContext()
    {
          _myComponent = context.GetComponent<MyComponent>();
    }

    public Status Update()
    {
          if (_myComponent.boolProperty)
          {
              _myComponent.ActionMethod();
              return Success;
          }
          return Running;
    }
}
```

GUÍA DE USO DE SMART OBJECTS EN UNITY

Los SmartObjects son elementos que pueden ser usados por personajes o agentes, siendo el propio objeto el que establece como se usa. Además, los *SmartObject* sirven para cubrir necesidades definidas en los agentes.

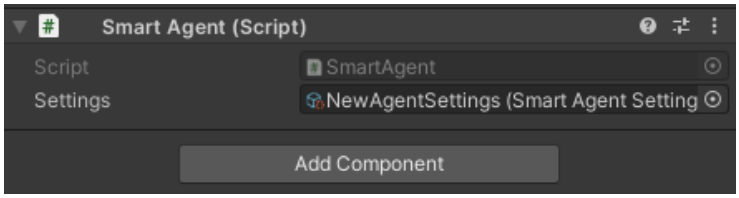
El uso de los *SmartObject* se basa en peticiones y respuestas. El sistema de comportamiento solicita al objeto su uso y este le responde con una interacción, que contiene la acción que el agente debe completar para usar correctamente el objeto.

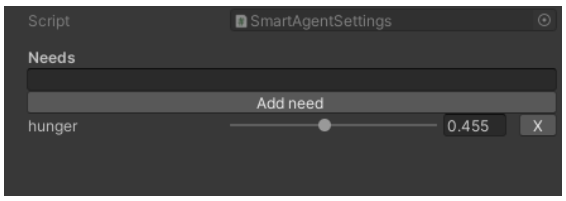
Los elementos principales del modelo de *SmartObjects* son el agente, los objetos y las acciones de petición.

Smart Agent

El agente es el elemento principal que permite interactuar con los *SmartObjects.* Para crear uno hay que añadir a un objeto el componente *SmartAgent*. Después se crea una configuración para el agente haciendo clic derecho sobre la vista del proyecto y seleccionado *Create* > *BehaviourAPI* > *SmartObjects* > *Smart Agent Settings*. Al hacer clic sobre el archivo generado se abrirá su inspector y podremos añadir y borrar necesidades, así como modificar su valor inicial.

Una vez creado el archivo de configuración se arrastra hasta el campo *Settings* del agente.





Smart Objects

Para añadir un nuevo tipo de objeto inteligente hay que crear un script que herede de *SmartObject* e implementar los siguientes métodos.

- *ValidateAgent*: decide si el agente que quiere interactuar con el objeto puede hacerlo. Puede usarse para comprobar si el agente tiene algún componente en concreto o cumple alguna condición.

```
public override bool ValidateAgent (SmartAgent agent)
{
    return true;
}
```

- *GetCapabilityValue:* Este método sirve para comprobar las capacidades del objeto para cubrir una necesidad concreta, recibiendo por parámetro el nombre de la necesidad y devolviendo la capacidad asociada en forma de valor numérico.

```
public override float GetCapabilityValue (string needName)
{
    return 0f;
}
```

- *RequestInteraction:* recibe como parámetros el agente y los datos de la petición y devuelve un objeto de la clase *SmartInteraction*. Para crear la interacción hay que crear antes la acción que se va a ejecutar y las capacidades que se aplican cuando se complete para pasarlas al constructor junto con el agente.

```
public override SmartInteraction RequestInteraction (SmartAgent agent, RequestData requestData)
{
    Action action = ...;
    Dictionary<string, float> capabilityMap = ...;
    SmartInteraction interaction = new SmartInteraction (action, agent, capabilityMap);
    return interaction;
}
```

RequestActions

Para que un sistema de comportamiento comunique agentes con *SmartObjects* es necesario usar *RequestActions*. Para crear una acción de este tipo, se crea una clase que hereda de *UnityRequestAction* e implementa los siguientes métodos:

- *GetRequestedSmartObject*: Devuelve el objeto al que se realiza la petición.
- *GetRequestData:* Devuelve los datos que se incluirán con la petición. En la mayoría de los casos no será necesario enviar ningún dato, y en otros, ese dato será solo el nombre de la necesidad que se quiere cubrir. Dependerá de cómo use los datos de la petición el *SmartObject* concreto.

```
protected override SmartObject GetRequestedSmartObject ()
{
      // return ...
}

protected override RequestData GetRequestData ()
{
      // return ...
}
```

Todas las *RequestActions* necesitar una referencia a un *SmartAgent* para funcionar. En la herramienta se incluyen dos tipos de *RequestActions*:

- *TargetRequestAction*: Se crear usando el agente, una referencia a un *SmartObject* y una instancia de *RequestData*. Cuando se ejecute la acción, se interactuará con el objeto establecido.
- *RandomRequestAction*: Se crea usando el agente y una instancia de *RequestData*. Cuando se ejecute la acción, se interactuará con un objeto aleatorio.

Para buscar los *SmartObjects* de la escena, se proporciona la clase *SmartObjectManager* que guarda una lista con todos los objetos, aunque es posible crear nuevas formas de gestionarlos.

Funcionamiento de una interacción con un SmartObject

1. El grafo de comportamiento entra en un nodo con una acción de tipo *RequestAction*.
2. Cuando se lanza el evento *Start* de la acción, esta busca un *SmartObject*. Si lo encuentra, le envía una referencia al agente y el resto de los datos de la petición (*RequestData*).
3. El *SmartObject* usa los datos recibidos para crear una instancia de *SmartInteraction* que contiene la acción que se va a ejecutar y se la envía a la *RequestAction*.
4. Si se ha recibido una interacción correctamente, se lanza el método *Start* de la acción que contiene.
5. A partir de este momento, los métodos lanzados en la *RequestAction* se propagan a la interacción. Hasta que termine o se pare. Si no se ha recibido una interacción se devolverá *Failure* directamente.

Ejemplo de uso de SmartObject

A continuación, se muestra los pasos a seguir para construir un *SmartObject* simple que mueva al personaje hasta una posición concreta y después imprima un mensaje por consola. Cuando la interacción termine el objeto cubrirá la necesidad de *"ejercicio"* del agente.

El primer paso es crear el agente. Se crea un objeto en la escena, añadiendo el componente *SmartAgent* y se crea un archivo de configuración de agentes añadiendo la necesidad ejercicio con valor inicial 0.

Después se crea una clase *ExampleSmartObject* que hereda de *SmartObject* y se implementan sus métodos. En este caso la acción de la interacción es una secuencia de dos acciones: *movementAction* y *logAction*.

```
public class ExampleSmartObject: SmartObject
{
    public Vector3 pos;

    public override SmartInteraction RequestInteraction (SmartAgent agent, RequestData
requestData)
    {
        BehaviourTree bt = new BehaviourTree ();

        Action movementAction = new WalkAction(_pos);
        LeafNode movementNode = bt.CreateLeafNode(movementAction);

        Action logAction = new DebugLogAction("El agente ha llegado al destino");
        LeafNode logNode = bt.CreateLeafNode(logAction);

        SequencerNode seq = bt.CreateComposite<SequencerNode>(false, movementNode, logNode);
        bt.SetRootNode(seq);

        Action action = new SubsystemAction(bt);

        Dictionary<string, float> capabilities = new Dictionary<string, float>();
        capabilities["ejercicio"] = 0.5f;
        return new SmartInteraction(action, agent, capabilities);
    }
```

```
    public override bool ValidateAgent(SmartAgent agent)
    {
        return true;
    }

    public override float GetCapabilityValue (string capabilityName)
    {
        if (capabilityName == "ejercicio") return 0.5f;
        else return 0f;
    }
}
```

El último paso será crear el sistema de comportamiento. Para simplificar el ejemplo, este sistema será un árbol de comportamiento con un único nodo hoja. La acción de este nodo será de tipo *TargetRequestAction*.

```
Public class ExampleSOBehaviourRunner: BehaviourRunner
{
    public SmartAgent agent; // Referencia al componente SmartAgent del objeto
    public ExampleSmartObject SmartObject; // Referencia al SmartObject creado anteriormente.

    protected override BehaviourGraph CreateGraph()
    {
          BehaviourTree bt = new BehaviourTree ();
          Action action = new TargetRequestAction (agent, SmartObject, new RequestData ());
          bt.CreateLeafNode(action);
          return bt;
    }
}
```

8.2 Anexo 2: Guía de uso del editor visual de grafos de comportamiento en Unity

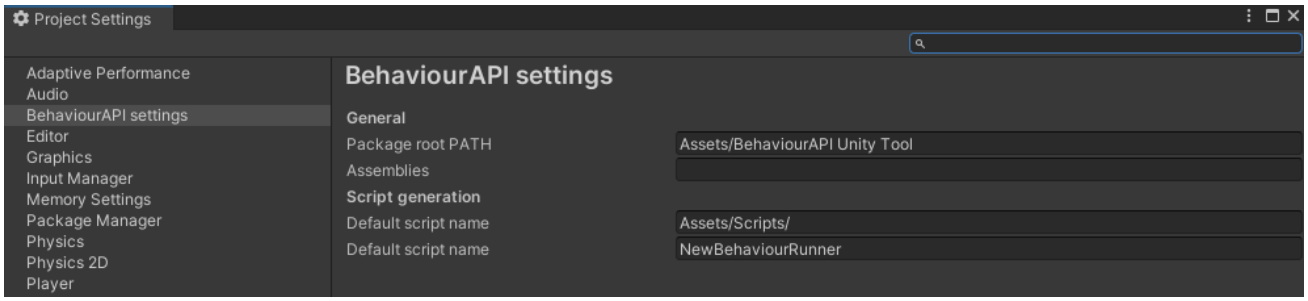
Como importar y configurar el paquete de Unity

El paquete "*Behaviour API Unity Tool*" incluye la API de creación de sistemas de comportamiento además de componentes para crear, ejecutar y depurar estos sistemas dentro de Unity.

Para usarlo se debe importar el paquete dentro de la carpeta "*Assets*". Si el paquete se importa en otra carpeta, se debe especificar la ruta en la configuración.

Configuración

Algunos elementos de la herramienta se pueden configurar en *Proyect Settings* > *BehaviourAPI Settings*.



1 Menú de configuración

- **Package root Path**: Indica el directorio raíz donde está el paquete de Unity.
- **Default script path**: El directorio por defecto donde se generan los scripts.
- **Default script name**: El nombre por defecto de los scripts generados.

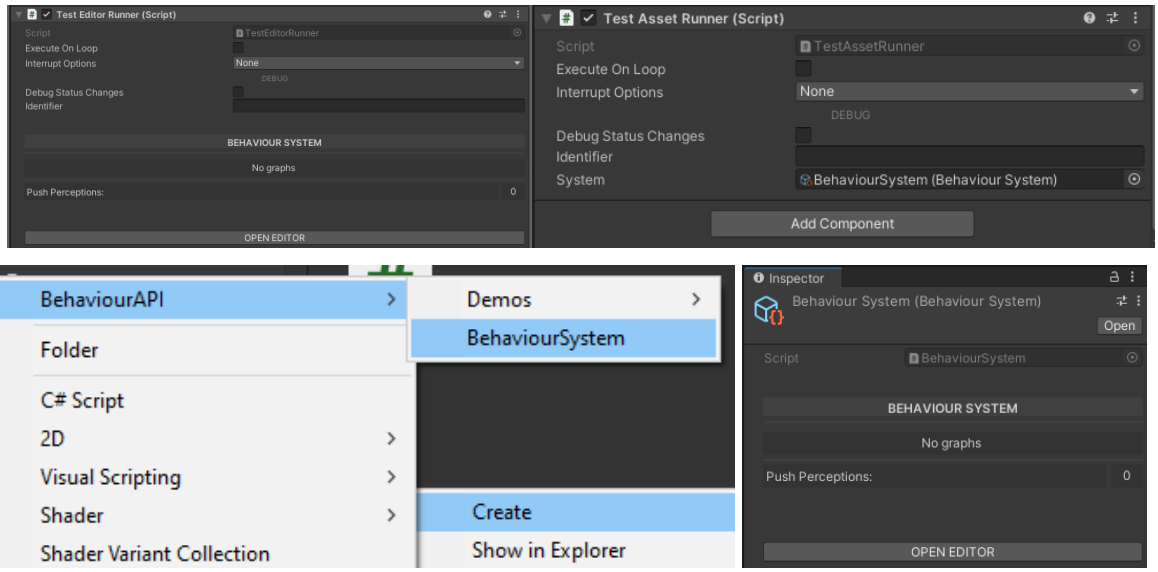
También es posible configurar los colores asociados a cada tipo de nodo en el editor.

Crear sistemas de comportamiento con el editor visual

Scripts principales

Se proporcionan dos formas de crear sistemas de comportamiento de forma visual.

- Vincular un sistema a un script de la escena: Para crear un sistema de comportamiento dentro de un *Script* usando la ventana del editor, el *Script* debe heredar de la clase *EditorBehaviourRunner*. Una vez creado se pulsa el botón *Open Editor* para abrir el editor visual.
- Vinculando el sistema a un asset del proyecto y referenciando el asset desde un script: Al hacer clic derecho sobre la carpeta de assets del proyecto y seleccionar *Create* > *BehaviourAPI* > *Behaviour*System se crea un archivo que contiene un sistema de comportamiento. Para editarlo se pulsa el botón *Open Editor* y para vincularlo a un *Script*, este debe heredar de la clase *AssetBehaviourRunner*.



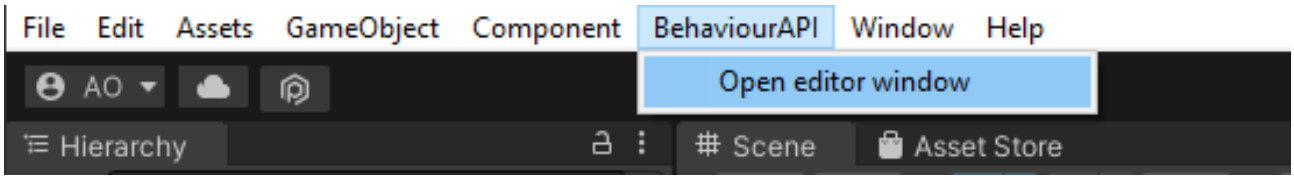
Modificar en el código el sistema creado

Es posible modificar el sistema de comportamiento creado en el método virtual *ModifyGraphs*, que recibe como parámetros diccionarios para buscar los grafos y percepciones push usando su nombre. Esto puede usarse para asignar acciones, percepciones, guardar referencias a nodos, etc.

```
protected override void ModifyGraphs (
          Dictionary<string, BehaviourGraph> graphMap,
          Dictionary<string, PushPerception> pushPerceptionMap)
{
   Node node = graphMap[“main tree”].FindNode<LeafNode>(“leaf 1);
   node.Action = new FunctionalAction(…);
}
```

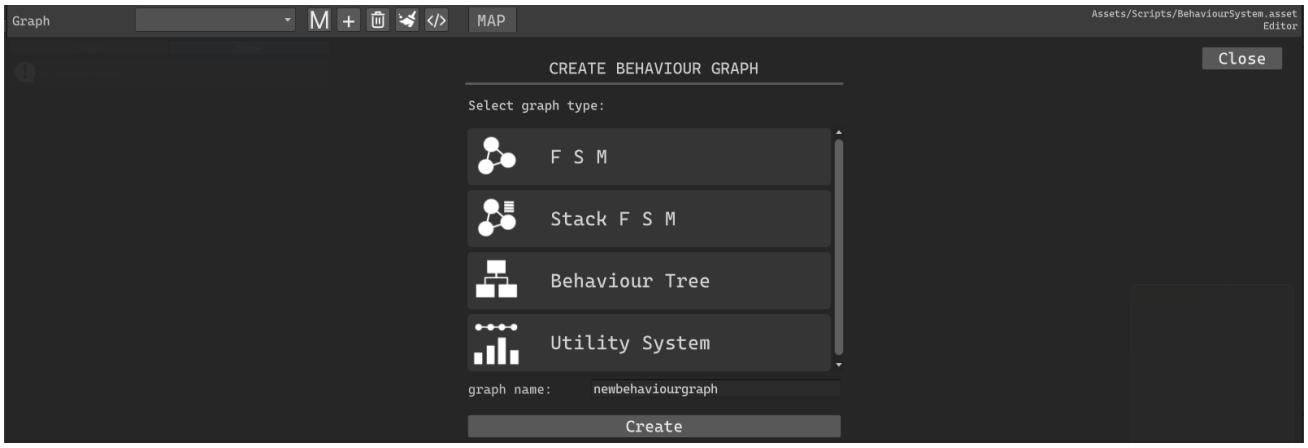
Guía para usar la ventana del editor de sistemas de comportamiento

La ventana del editor puede abrirse desde el menú *BehaviourAPI* > *Open Editor Window*. Para editar un sistema de comportamiento se debe pulsar el botón *Edit* en un componente de tipo `EditorBehaviourRunner` o en un asset de tipo `BehaviourSystem`.

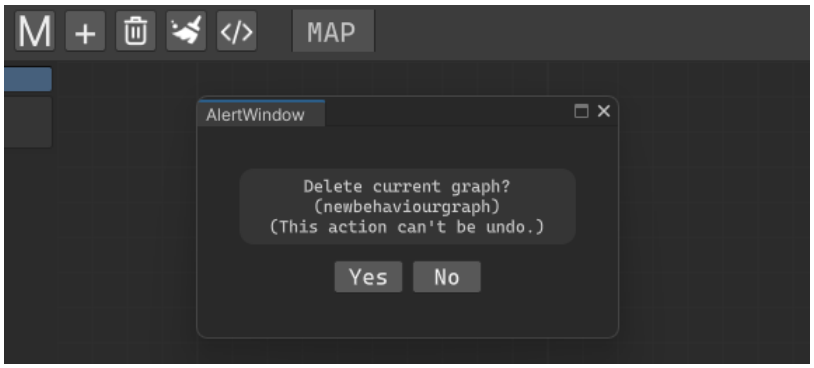


Añadir y borrar grafos

Si se abre un sistema de comportamiento sin grafos, se mostrará el panel de creación de grafos. Para crear uno hay que seleccionar el tipo y pulsar el botón "*Create"*. Es posible añadir nuevos grafos en cualquier momento pulsando el botón *"+"* de la barra de herramientas.

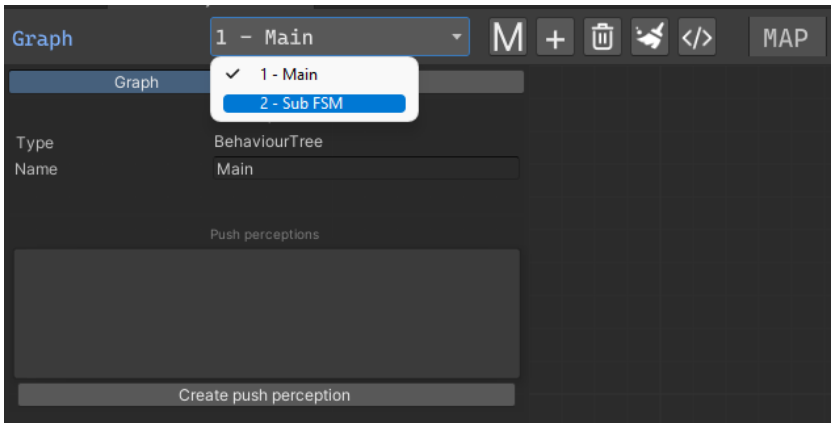


Para borrar el grafo actual se pulsa el **botón con el icono de la papelera** en la barra de herramientas. Aparecerá una ventana de confirmación antes de borrarse.



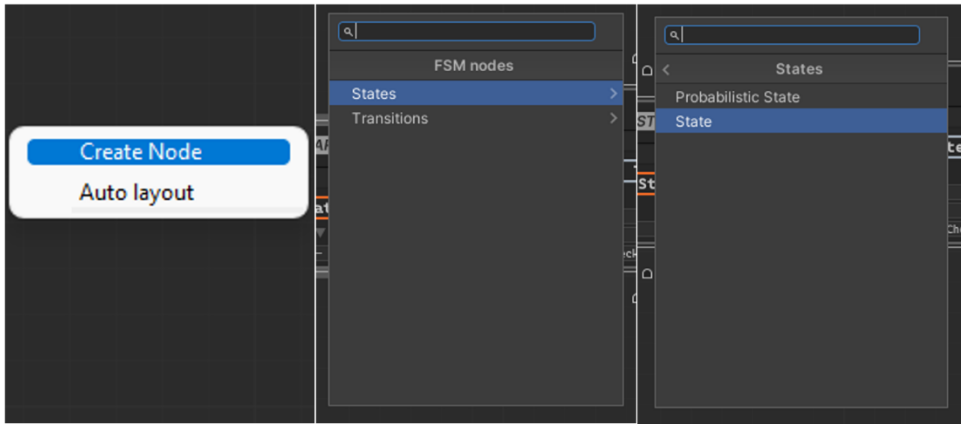
Seleccionar y editar grafo

Para cambiar el grafo seleccionado se usa el menú de selección de grafo de la barra de herramientas. Una vez seleccionado se cargará su representación y se pueden editar sus variables en el inspector.



Añadir y borrar nodos

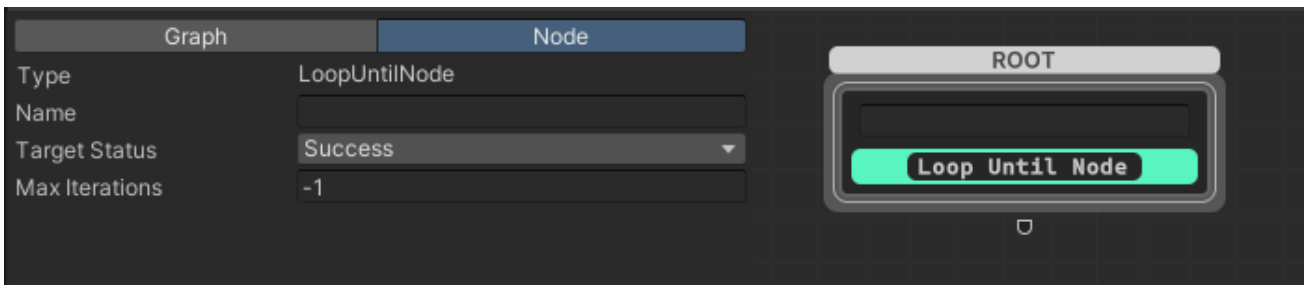
Para crear un nuevo nodo hay que hacer clic derecho para abrir el menú contextual y seleccionar *"Create Node"* o bien pulsar la tecla espacio. Al hacerlo se abre un menú que permite seleccionar un tipo de nodo de los disponibles para el grafo seleccionado.



Para borrar un nodo hay que hacer clic sobre él y pulsar la tecla suprimir.

Editar nodos

Hacer clic sobre uno de los nodos del editor abre el **editor de nodos**, que permite modificar el nombre del nodo y sus variables.

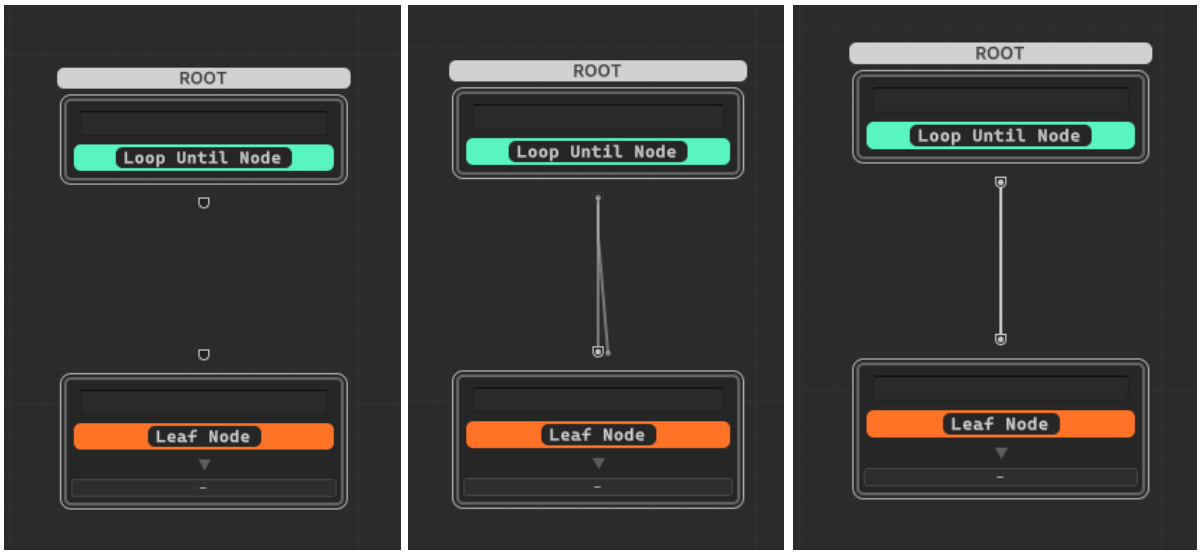


2 Editor de nodos

Conectar nodos

Los nodos pueden tener uno o varios "puertos" de entrada y de salida dependiendo de su tipo. Al hacer clic sobre uno de los puertos de un nodo y arrastrar hacia un puerto del tipo contrario en otro nodo se crea una conexión entre ellos, en la que el nodo con el puerto de salida es el "padre" y el otro nodo es el "hijo".

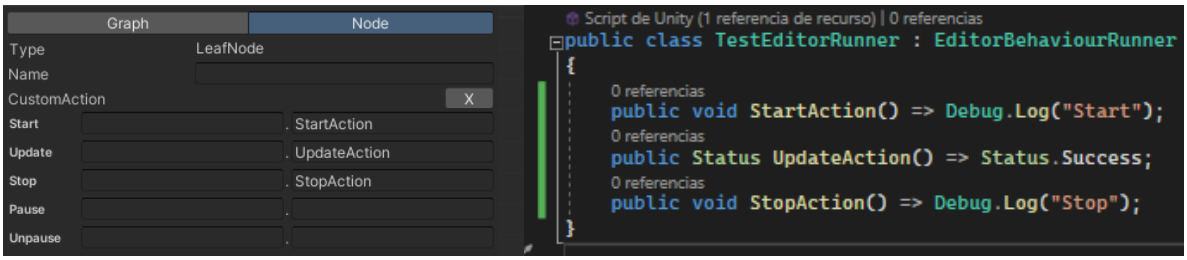
El tipo de puerto se distingue en que los puertos de entrada tienen el lado curvo hacia dentro y los de salida hacia fuera.



Para borrar una conexión hay que hacer clic sobre la arista y pulsar la tecla suprimir.

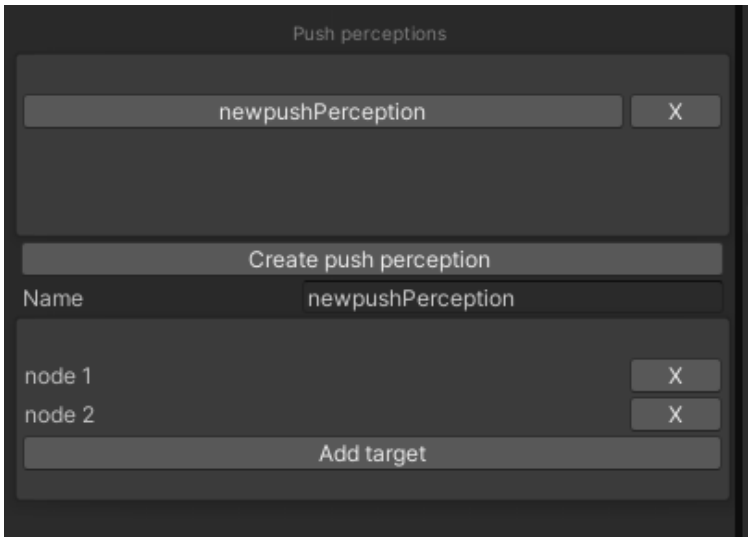
Asignar acciones y percepciones

En los nodos que pueden tener acciones o percepciones asignadas aparece un botón "Assign action", que permite abrir un menú para elegir el tipo de acción. Las acciones de tipo *FunctionalAction* y percepciones de tipo *ConditionPerception* no están soportadas directamente, pero es posible simularlas usando *CustomActions* y *CustomPerceptions*. Para ello hay que definirse define en cada evento el nombre del componente y del método que se va a usar. Estos métodos no deben tener parámetros y su tipo devuelto debe coincidir con el tipo devuelto del evento.



Crear percepciones push

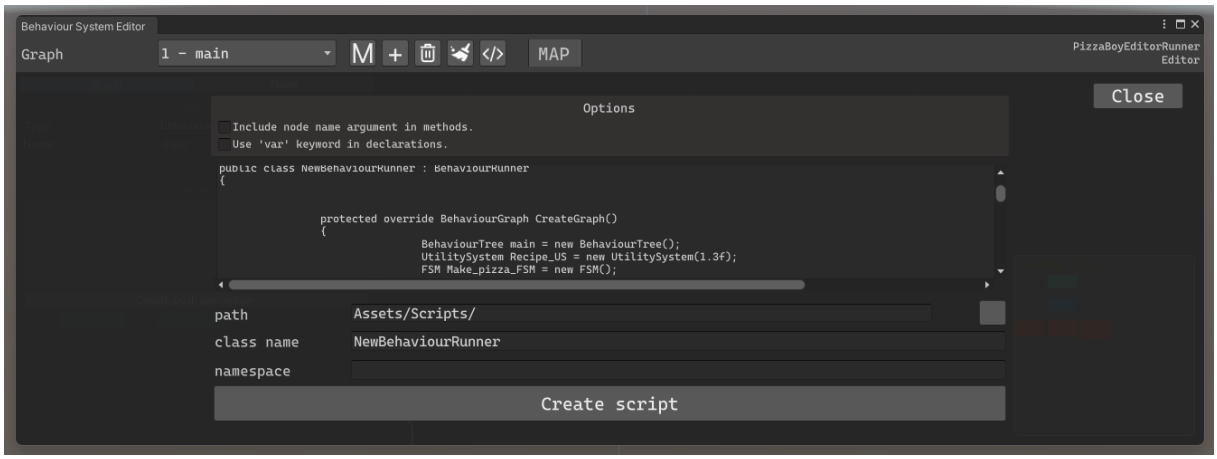
Para crear una percepción push se pulsa el botón *"Create push perception"* en el inspector de grafos. Una vez creada al hacer clic sobre una percepción push se abrirá su editor donde podemos cambiar el nombre y añadir y borrar nodos.



Generar un script a partir de un sistema de comportamiento

Al pulsar el botón con el icono "</>" en la barra de herramientas se abrirá un panel para generar un script a partir de los datos del sistema de comportamiento que se está editando. En esta ventana se puede especificar el nombre del script y la carpeta donde se va a guardar, además de otras opciones como el espacio de nombres.

Importante: Si se especifica un nombre y una ruta que ya existe, se sobrescribirá el archivo.

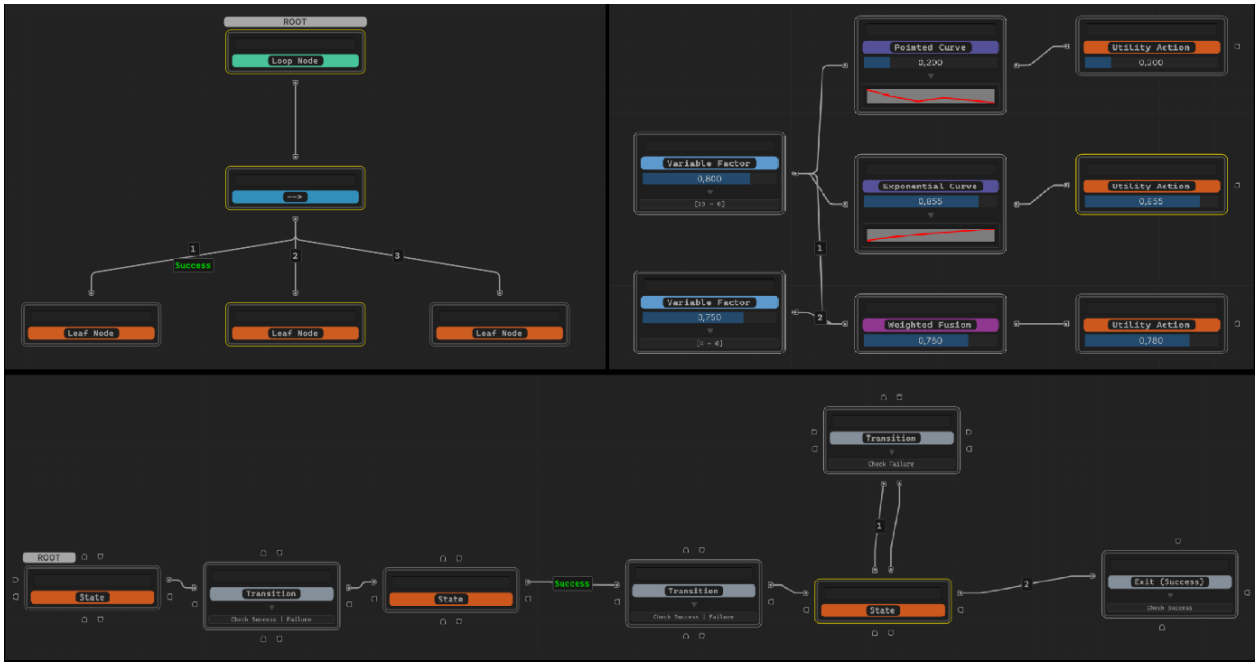


Se recomienda que el código generado se revise siempre antes de usarlo, ya que es posible que tenga errores de compilación, especialmente si se incluye nodos o acciones creadas por el usuario.

Usar el depurador en tiempo real

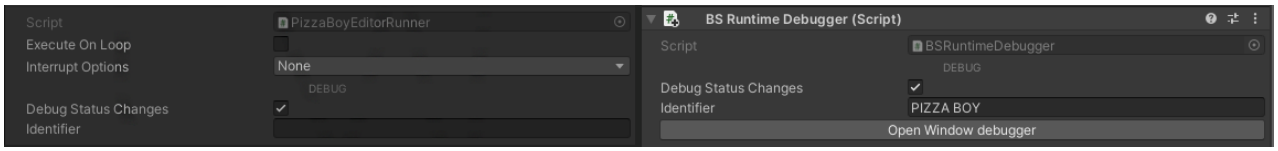
Es posible abrir la ventana del editor de sistemas de comportamiento en modo *"Debug"*. En este modo, todos los controles de edición están desactivados excepto la posibilidad de mover nodos, pero se ofrecen varias funcionalidades:

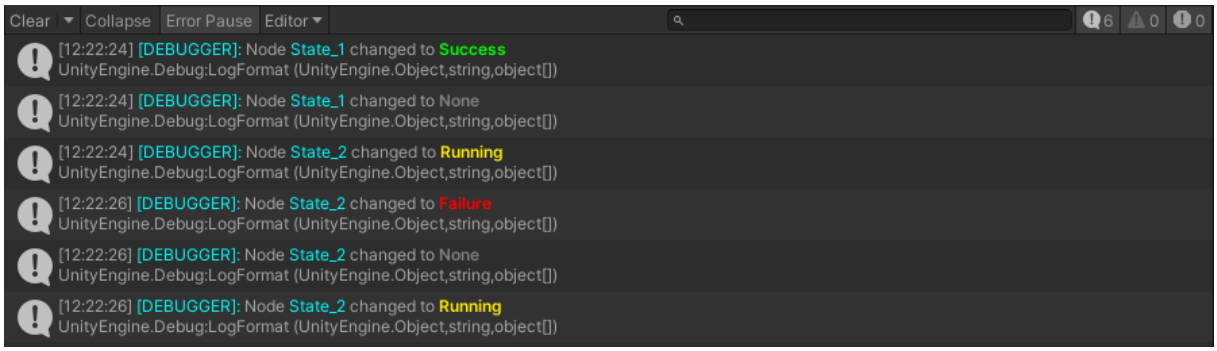
- Ver en tiempo real los nodos activos de cada grafo.
- En árboles de comportamiento, se muestra en las conexiones el valor devuelto por la rama concreta cuando ha terminado su ejecución.
- En máquinas de estados, se muestra el valor con el que ha terminado el último estado ejecutado.
- En sistemas de utilidad, cada nodo muestra su valor de utilidad actual.



Otra funcionalidad incluida en estos *Scripts* es el mostrar por consola los cambios de estado de ejecución de los nodos. Para ello hay que activar el flag *"Debug status changes"*.

También se puede especificar el identificador que aparece en los mensajes con la variable "Identifier".





Usar el depurador con sistemas generados por código

Para que el depurador tenga acceso a los grafos generados por código debemos incluir el componente *BSRuntimeDebugger*. Para registrar un grafo en el depurador en tiempo real se usa el método *RegisterGraph*, al que se puede pasar también el nombre del grafo para identificarlo en el editor.

```
public class ExampleRunner: BehaviourRunner
{
    Public BSRuntimeDebugger _bsRuntimeDebugger;

    protected override BehaviourGraph CreateGraph()
    {
          var fsm = new FSM();
          var subBt = new BehaviourTree();
          …
          _bsRuntimeDebugger.RegisterGraph(fsm, “Main FSM”);
          _bsRuntimeDebugger.RegisterGraph(subBt, “sub BT”);
          return fsm;
    }
}
```