UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA
Departamento de Ingeniería del Software e Inteligencia Artificial

TESIS DOCTORAL

Aplicación de técnicas de aprendizaje automático supervisables por el diseñador al desarrollo de agentes inteligentes en videojuegos

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Ismael Sagredo Olivenza

Directores

Pedro Antonio González Calero
Marco Antonio Gómez Martín

Madrid, 2018

Aplicación de técnicas de aprendizaje automático supervisables por el diseñador al desarrollo de agentes inteligentes en videojuegos

TESIS DOCTORAL

Ismael Sagredo Olivenza

Dirigida por los Doctores

Pedro Antonio González Calero
Marco Antonio Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Abril 2017

Documento maquetado con TEXIS v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

Aplicación de técnicas de aprendizaje automático supervisables por el diseñador al desarrollo de agentes inteligentes en videojuegos

Memoria que presenta para optar al título de Doctor en Informática

Ismael Sagredo Olivenza

Dirigida por los Doctores

Pedro Antonio González Calero
Marco Antonio Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Abril 2017

A mis padres, a Beatriz y a mis abuelos.

Agradecimientos

En primer lugar, agradecer a mis directores de tesis, Marco Antonio y Pedro, por su apoyo, su colaboración, su dedicación y, sobre todo, su paciencia. Soy consciente de que ha sido un camino duro, tanto para mi como para ellos, pero siempre he tenido la seguridad de poder contar con su consejo y su comprensión y sin ellos esta tesis no habría sido posible. Un agradecimiento muy especial a mi pareja Beatriz Martín, por todas las horas que no he podido estar con ella y por su apoyo incondicional todos estos años. También a Pedro Pablo Gómez Martín, con el que he publicado varios artículos, sin cuyos consejos y revisiones no hubiera sido posible esta tesis.

Además, quiero hacer extensivo este agradecimiento a Gonzalo Flórez Puga y David Llansó, por su colaboración y guía en los primeros años de la tesis y por aclararme dudas cuando más oscuro estaba todo. A Antonio Sánchez, a quien no pocas veces pedí consejo, fruto de los cuales surgió la idea de los *Trained Query Node*. Por extensión a mis compañeros de departamento, entre los cuales se encuentran: Raquel, Fede, Guille, Manu, Samer y Virginia y a los demás miembros del Departamento de Ingeniería del Software e Inteligencia Artificial, por los consejos y los buenos momentos que hemos pasado juntos. A mis compañeros de batalla, que pronto pasarán por lo mismo: José Jorro, Marlon, Gineth y Rafa. Gracias por compartir sobremesa, ideas, sitios donde publicar y por el apoyo todo este tiempo. Sólo deciros: ¡muchos ánimos con lo que os queda! Más que compañeros, os considero amigos.

También quiero dar un agradecimiento muy especial a mis amigos del dream-team del desarrollo de videojuegos: Omar Pérez (que además nos echó un cable en los experimentos), David Rodríguez, Daniel Perivañez y Elena Fernández. Sin las amenas charlas en la cafetería y algún que otro consejo, el camino hubiera sido mucho más duro. ¡Gracias por todo! A los chicos del Máster de desarrollo de videojuegos, así como a los alumnos del grado de desarrollo de videojuegos, que se prestaron voluntarios para los experimentos de forma totalmente altruista y sin los que no hubiéramos podido realizarlos.

Y por último, quiero agradecer también el apoyo que me ha prestado mi familia desde siempre. Especialmente a mis padres, que han sufrido de cerca los ratos malos y los buenos. Su apoyo y comprensión han sido determinantes para llegar hasta aquí. Y para concluir, una mención especial a Samus Aran,

sin sus ronroneos, más de un deadline hubiera sido insoportable.

Índice

Índice de figuras

Índice de tablas

Resumen

La creación de los comportamientos de los personajes que interaccionan con el jugador en videojuegos es una tarea compleja que involucra a diferentes roles dentro de la industria del videojuego, con diferentes conocimientos y motivaciones, principalmente programadores, diseñadores y artistas. Dicho proceso ha intentado ser simplificado con multitud de modelos computacionales durante décadas, habiendo un consenso actual en usar árboles de comportamientos de forma generalizada, sobre todo en comportamientos complejos que requieren de cierta coordinación con otros NPCs o con el jugador.

Multitud de editores gráficos se han creado para simplificar la creación de estos comportamientos, permitiendo sistemas cada vez más potentes y robustos. Aun así, en la industria se sigue prefiriendo que la programación de los comportamientos la realicen los programadores. Esto sucede porque los propios programadores no se fían demasiado de los diseñadores, además de que muchos diseñadores se sienten abrumados cuando tienen que pensar usando el lenguaje de los árboles de comportamiento. Así pues, los diseñadores acaban diseñando el comportamiento y los programadores llevándolos a cabo.

En el presente trabajo, nos centraremos en resolver el problema de comunicación que existen entre los programadores y los diseñadores, a la hora de crear comportamientos, debido a esa relación entre ambos, que hace que sea necesario múltiples iteraciones hasta conseguir que el comportamiento sea desarrollado satisfactoriamente.

Este problema ha sido evaluado empíricamente en el presente trabajo, confirmando que los diseñadores sin conocimientos de programación tienen más dificultadas a la hora de crear comportamientos, incluso teniendo una herramienta gráfica para crearlos y restringiendo sólo el uso de esta herramienta para crear comportamientos de alto nivel.

Por lo tanto, se propone un modelo en el que los diseñadores puedan crear comportamientos sin necesidad de programar, usando para ello una serie de técnicas englobadas bajo el epígrafe de *Programación por Demostración*, que consiste en programar un sistema usando la propia interfaz del mismo, sin necesidad de escribir un programa en un lenguaje de programación, ni dibujar

un modelo gráfico que represente dicho programa, simplemente aportándole ejemplos de cómo resolver las diferentes situaciones que se plantean en el juego, tomando el control del propio personaje.

El sistema diseñado permite utilizar diferentes modelos para aprender los comportamientos. Se utiliza razonamiento basado en casos, árboles de decisión y redes de neuronas. Cada modelo tiene unas ventajas e inconvenientes que se discuten en el presente trabajo.

Además, la programación por demostración se integra dentro del formalismo de los árboles de comportamiento, creando un nodo especial para ello. Esta integración aporta mayor expresividad a los modelos, permitiendo dividir los comportamientos complejos en otros más sencillos, que pueden ser después seleccionados con un conjunto de nodos de árbol de comportamiento. Además, integra la programación por demostración dentro de una herramienta ampliamente utilizada por la industria del videojuego. Haciendo que sea mucho más accesible tanto para programadores como diseñadores.

Pensamos que esta es la forma más natural para que el diseñador pueda crear estos comportamientos, ya que los diseñadores están acostumbrados a jugar y, por tanto, cuando juegan saben si los comportamientos funcionan o no.

Sin embargo, este conocimiento es mucho más difícil de expresar con palabras en un documento de diseño, donde normalmente los diseñadores tienden a ser poco concretos. Si nos vamos al otro extremo y es el propio diseñador el que edita el árbol de comportamiento, nuestros estudios empíricos nos dicen que para aquellos diseñadores sin grandes conocimientos técnicos, no es una tarea que les resulte fácil. Así pues, para conseguir que el diseñador forme parte del proceso de creación del propio comportamiento, nuestra propuesta es que se use un enfoque alternativo: Que el diseñador entrene el comportamiento por demostración primero y sólo si no consigue obtener buenos resultados, entonces se aborde el problema de la forma tradicional, modificando el comportamiento entrenado usando árboles de comportamiento.

Los experimentos realizados demuestran que usar *programación por demostración*, dentro del modelo de árboles de comportamiento, tiene múltiples ventajas. Por ejemplo, se pueden conseguir modelos más precisos combinando los árboles de comportamiento y la programación por demostración, que solamente usando esta última. Así mismo, el sistema puede garantizar que el comportamiento final que se utilice en el juego esté libre de comportamientos emergentes, algo que normalmente no desean los diseñadores de videojuegos, gracias a que si el comportamiento no actúa con la precisión esperada, el sistema puede crear un árbol que tenga un comportamiento equivalente al que el diseñador entrenó, que será totalmente determinista.

Finalmente, presentamos una metodología que integra la creación de comportamientos por demostración dentro del ciclo de desarrollo iterativo. Esta

metodología identifica qué roles deben adoptar tanto los programadores como los diseñadores, dentro del proceso de desarrollo de los diferentes prototipos del comportamiento, moviendo al diseñador de la tradicional tarea de supervisor de los comportamientos creados por el programador, hacia una tarea mucho más involucrada en la creación del propio comportamiento, en colaboración con el programador.

Esta colaboración permite reducir los tiempos de desarrollo y minimizar los problemas de comunicación entre programador y diseñador, además de reducir la importancia de la definición textual del comportamiento y dándole mayor peso al propio prototipo, que se convierte en la propia definición del comportamiento.

Abstract

Creating the behavior for non-player characters in video games is a complex task that requires the collaboration among different roles in the game industry with different knowledge and motivations, mainly artists, programmers, and game designers. This process has tried to be simplified with a multitude of computational models during years. Nowadays, in the industry, there is a consensus on the use of behavior trees as a standard model, especially when the behavior is complex or some coordination among characters is needed.

A number of graphic editors have been created to simplify the behavior creation, allowing systems increasingly powerful and robust. Even so, in the industry the behaviors are created by programmers. This happens because the programmers do not trust the designers, furthermore, the designers feel overwhelmed when they must think in the language of behavior trees. For that reason, the designers typically only design the behavior while the programmers write the code.

Our work specifically concentrates on solving the communication problem among programmers and game designers in the creation of behaviors. Due to the complexity of their interaction, several iterations are needed to obtain a successful behavior.

This problem has been evaluated empirically in this work, confirming that the designers without technical knowledge have more troubles to create behaviors, even having a graphical tool and creating only high level behaviors.

For that reason, we propose a new model where designers can create behaviors without programming, thanks to a set of techniques named *Programming by Demonstration*. These techniques consist of programming a system using its interface, without coding or creating any diagrams, simply by giving examples of how to solve the different game situations, by taking the control of the character in the game.

Furthermore, programming by demonstration is integrated into the Behavior Trees framework, creating a special node. This integration brings more expressiveness to the models, allowing to split the complex behavior in other simpler, that can be merged using some nodes of the behavior tree. In

addition, by integrating programming by demonstration into a tool widely used in the game industry as behavior trees, makes our technique much more accessible both to programmers and designers.

The system allows different models to learn the behaviors: Case-Base Reasoning, Decision Trees and neural networks. Each model has some advantages and disadvantages that we discuss in this work.

Such authoring mechanism it is a natural way for designers to build the character behavior, because designers usually play games and when they play, they know if the behavior is working or not. However, this knowledge is more difficult to express in words, because the designers are usually not very concrete in their explanations.

In addition, creating a behavior tree is more complex, especially if the designer does not have technical knowledge. In this way, we intent to put the designers into the development process and not only the design or validation, using the model itself obtained by demonstration as a description of the behavior.

The experiments demonstrate that the use of *programming by demonstration* along with behavior trees has multiple advantages such as an increased expressivity of the learned models, as well as being able to guarantee that the behavior works successfully, avoiding emergent behaviors. This feature is very important for the designers because they want to have everything under control.

Finally, we present a methodology that integrates programming by demonstration into the iterative development cycle. This methodology tries to describe the roles and responsibilities of both programmers and designers in the creation of the behavior prototype. The designer is no longer responsible just for the supervision, but also creates behavior in collaboration with the programmer. This collaboration allows to reduce the development cost and solves the communication problems among programmers and designers, reducing the weight of the textual description and increasing the weight of the prototype, which becomes the behavior definition itself.

Capítulo 1

Introducción

1.1. Motivación

El desarrollo de un videojuego es una tarea compleja que requiere de la estrecha colaboración entre una gran cantidad de personas con diferentes aptitudes, con distintas sensibilidades, conocimientos y objetivos. Por un lado, están los diseñadores del juego, que son los encargados de idear cómo se va a jugar al mismo, es decir las reglas y las mecánicas que lo gobiernan. Por otro lado, tenemos a los programadores, que se encargan de crear la tecnología necesaria para llevar a cabo las ideas de los diseñadores y finalmente están los artistas, que se encargan de generar los recursos visuales y sonoros que el videojuego necesita para ser jugado satisfactoriamente, así como producir la retroalimentación necesaria para que el jugador pueda interactuar con el juego.

Existen más roles que entran en escena, como puede ser el departamento de marketing de la empresa, que intenta vender el juego a sus potenciales compradores y que influye en las decisiones del resto de integrantes del equipo; o los productores, que controlan el rumbo del proyecto. Pero en esta tesis vamos a centrarnos principalmente en la relación que existe entre dos de esos roles, el diseño y la programación, ya que son los que tienen mayor contacto e interrelación dentro del equipo de desarrollo y por tanto, los más propensos a sufrir conflictos. Además, son los que mayor peso tienen en el desarrollo de un videojuego junto con los artistas.

En concreto, nos centramos en uno de los aspectos en donde diseñadores y programadores colaboran estrechamente: el desarrollo de los comportamientos de los personajes no controlados por el jugador, lo que se suele denominar como NPC por sus siglas en inglés (Non Playable Character), es decir, personajes dentro del juego que no son manejados por el jugador. El comportamiento de un NPC está directamente relacionado con el concepto de juego que ha ideado el diseñador, por lo que no podemos desligarlo del diseño del juego, ya que forma parte de a qué se juega y del cómo se juega.

Así pues, los diseñadores suelen querer controlar el comportamiento de los NPCs estrechamente, para estar seguros de cómo van a actuar en las situaciones que ellos han ideado, como parte del diseño del propio juego. Por lo tanto, son los diseñadores los que crean y describen como debe comportarse un NPC y, por el contrario, son los programadores los que se encargan de hacerlo realidad, ya que crear comportamientos no es sencillo a nivel técnico y requiere de conocimientos de programación bastante avanzados, que los diseñadores no tienen por qué tener.

Los diseñadores suelen expresar los comportamientos en documentos con descripciones muy generales de los mismos. Por ejemplo, el enemigo básico (denominado por el equipo de desarrollo como *Generic NPCs*) de nombre Phil, del videojuego Fallout Van Buren[1], que estuvo en desarrollo por Black Isle Studios® y que fue cancelado, es descrito en su documento de diseño de la siguiente forma:

> La primera vez que él ve al protagonista, realizará un disparo de emergencia cerca de él y le dirá: "¡Pon tus malditas armas fuera y no intentes ninguna estupidez, gilipollas!". Phil responderá sin mucha hostilidad si la gente hace lo que pide. El Hombre Ahorcado, sin embargo, podría incitar a Phil a abrir fuego a discreción.
>
> Documento de diseño de Fallout Van Buren[2].

Como se puede apreciar, la descripción del comportamiento es totalmente imprecisa y no queda claro exactamente qué debe hacer el NPC. Por ejemplo, con la descripción, no sabemos si el NPC está quieto o patrullando. Tampoco sabemos el rango de visión, ni el de disparo, ni cuál es la reacción concreta si el jugador no obedece sus órdenes. Conseguir que el programador implemente correctamente este comportamiento seguramente requerirá varias iteraciones y muchas consultas al diseñador para clarificar y concretar conceptos.

Los programadores generalmente necesitan una descripción mucho más precisa de lo que debe hacer el NPC, por lo que tendrán que producirse varias reuniones para clarificar el comportamiento. Las primeras versiones de dicho comportamiento seguramente, si no está bien definido, no ofrezcan un comportamiento acorde con lo esperado por el diseñador y se necesitarán varias iteraciones de ajuste. Este ha sido un punto caliente tradicional en el desarrollo de videojuegos que no ha sido del todo resuelto en la actualidad, debido a que los diseñadores no disponían de las herramientas necesarias para poder crear ellos mismos los comportamientos, como sí que existen para, por ejemplo, diseñar un nivel gracias a los editores de niveles.

[1] `https://en.wikipedia.org/wiki/Van_Buren_(video_game)`

[2] `http://vignette3.wikia.nocookie.net/fallout/images/d/d5/11_Burham_Springs.pdf/revision/latest?cb=20120625153352`

Estas sucesivas interacciones para perfilar el comportamiento, implican un incremento en el tiempo de desarrollo, lo que conlleva un aumento de los costes de producción. En la Figura 1.1 podemos ver la estructura de costes de Gears of War, creado por Epic Games® y publicado por Microsoft® en 2006. Con este gráfico podemos hacernos una idea de la estructura de costes de un juego de los denominados AAA. Se consideran juegos AAA aquellos que tienen altos valores de producción y que son creados por estudios que tienen decenas de trabajadores, durante varios años. En el gráfico, podemos ver como el 45 % del coste de un juego es la programación y el diseño/arte del mismo. Si se quiere reducir costes de desarrollo, es importante atacar los costes relacionados con la programación y el diseño de juego, ya que son el grueso principal de los mismos.

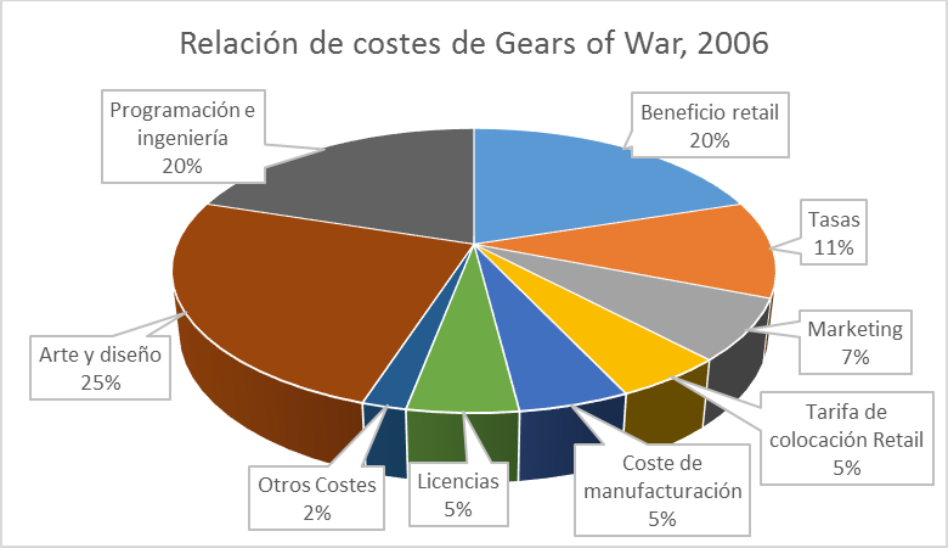


Figura 1.1: Estructura de costes de Gears of War[a]

[a]Fuente Forbes `https://www.forbes.com/2006/12/19/ps3-xbox360-costs-tech-cx_rr_game06_1219expensivegames.html`

Por consiguiente, en este panorama de desarrollo donde existen requisitos cambiantes, altos presupuestos y problemas de comunicación entre diseñadores y programadores; es importante que ambos roles puedan cooperar para reducir en lo posible los errores de interpretación al crear un comportamiento, minimizando las sucesivas fases de interacción que se necesitaban anteriormente para crearlos y haciendo que el diseñador, que es quien idea el comportamiento, forme parte del proceso de desarrollo del mismo, más allá del ajuste de parámetros que tradicionalmente realiza.

La industria ya ha intentado realizar movimientos en esa dirección, ya que con el paso del tiempo, ésta se ha ido moviendo hacia un modelo de ejecución dirigido por datos (Wirfs-Brock y Wilkerson, 1989) que se ha uti-

lizado en múltiples facetas en la industria del videojuego tanto para crear animaciones (Mizuguchi et al., 2001) como para crear otros aspectos como en la inteligencia artificial (IA) (Yannakakis, 2012), donde gran parte de los comportamientos son generados como datos, bien sean reglas, ficheros de descripción del comportamiento, etc.

Este modelo de ejecución dirigida por datos, ha ido evolucionando hacia la creación de editores gráficos que trabajan sobre dichos ficheros de datos y que están incorporados en la mayoría de los motores comerciales, que permiten programar de forma visual sin necesidad de saber programar a bajo nivel. Estos editores puede usar diferentes formalismos como los árboles de comportamiento o máquinas de estado, que describiremos más adelante. Dichos modelos conceptuales están a un nivel de abstracción mucho más alto que los lenguajes de script o directamente del código fuente del juego y tener editores visuales ayuda a su comprensión y uso, pero aun así y basándonos en los experimentos realizados que detallaremos durante el presente trabajo publicados en Sagredo-Olivenza et al. (2015b), se necesitan ciertos conocimientos previos de programación si se quieren hacer comportamientos algo más complejos.

Una posible solución para que los diseñadores no técnicos puedan crear comportamientos con mayor facilidad, es permitir que éstos puedan entrenarlos haciendo lo que mejor saben hacer, jugando con el NPC de forma que éste aprenda las acciones que debe realizar automáticamente. Esta técnica ya ha sido aplicada en múltiples campos anteriormente. Por ejemplo, en la literatura, se le suele denominar *Programación por Demostración* y ha sido explorada en numerosos artículos como en Lemaitre et al. (2015); Jaidee et al. (2013); Ontañón et al. (2010) entre otros, donde se intentaban aprender comportamientos en videojuegos usando programación por demostración en múltiples contextos como en la creación de comportamientos para juegos de estrategia o para coordinar diferentes agentes.

La programación por demostración, como veremos más adelante con mayor detalle, utiliza técnicas que infieren o deducen la acción que se debe realizar, teniendo en cuenta el estado del mundo actual y el estado del mundo en casos anteriores. Pero dicha inferencia no es exacta y los modelos pueden cometer fallos o generar comportamientos emergentes no esperados. Este es el principal problema que tiene esta técnica para ser utilizada en videojuegos, ya que los diseñadores suelen ser recelosos a la hora de perder el control de lo que el NPC va a hacer, porque, como explicamos previamente, esto puede implicar un cambio en la filosofía misma del juego y de lo que quiere transmitir el diseñador.

Pero, a pesar de estos problemas, esta técnica puede ayudar a crear comportamientos sin necesidad de saber programar, lo que dotaría a los diseñadores de una herramienta muy valiosa para crear comportamientos de forma autónoma. Por lo tanto, lo que perseguimos en el presente trabajo es con-

seguir acotar estos comportamientos no esperados, para poder utilizar esta técnica en un entorno de desarrollo real, sin producir rechazo en los diseñadores.

Nuestro trabajo pretende conseguir que los diseñadores incrementen su grado de autonomía mediante el uso combinado de nuestro editor de comportamiento, *Behavior Bricks* y la programación por demostración, junto con una metodología de uso, complementándose entre sí ambas técnicas, para crear un entorno de trabajo ideal para diseñadores no técnicos, donde pueden idear y crear los comportamientos mientras juegan con el propio NPC, reduciendo la comunicación con el programador y así reduciendo el número de interacciones entre ambos. En definitiva, hacer mucho más versátiles a los diseñadores no técnicos, lo cual es ideal para las empresas con menos recursos y de paso, reducir los tiempos de desarrollo, algo muy necesario para los estudios de mayor tamaño, que soportan grandes costes de producción.

1.2. Objetivo

El objetivo de la presente tesis es ofrecer los medios para facilitar a los diseñadores la tarea de creación de comportamientos para los NPC de un videojuego, de forma mucho más autónoma de lo que hasta ahora habían sido capaces de hacer. Como mencionábamos anteriormente, la creación de comportamientos para los NPC es una tarea compleja que requiere la colaboración de diseñadores y programadores.

La solución que presentamos a este problema, y que ha supuesto el centro de esta investigación, es la utilización de un editor de comportamientos que hemos denominado *Behavior Bricks* que implementa el modelo de árboles de comportamiento con un especial énfasis en la reutilización de comportamientos, debido a que tanto las acciones primitivas, como los comportamientos son totalmente parametrizables, así como se han incorporado mecanismos para crear árboles de comportamiento con acciones genéricas, que pueden ser implementadas de diferentes formas en cada instancia concreta del NPC que las use. Y, por otro lado, para dar soporte a diseñadores que no tienen los conocimientos de programación suficientes como para usar árboles de comportamiento, hemos extendido este modelo con un nodo especial que denominamos *Trained Query Node*, que permite mediante una sesión de entrenamiento usando programación por demostración, recuperar una de las posibles implementaciones de un comportamiento a bajo nivel, permitiendo al diseñador implementar comportamientos sin programar.

El fin último es que el diseñador se involucre activamente en el proceso de desarrollo del comportamiento y no sólo en su diseño inicial y posterior ajuste. Para ello el sistema dispone de múltiples opciones para que el diseñador pueda adecuar los comportamientos entrenados a sus necesidades para asegurar que los comportamientos aprendidos pueden ser usados sin

temor en el videojuego, entre ellos, el sistema es capaz de mostrar un árbol de comportamiento, generado a partir de los datos de entrenamiento, para ayudar al diseñador y al programador a cooperar entre ambos, centrando la especificación del comportamiento en el propio modelo entrenado y no en las descripciones textuales poco precisas, que los diseñadores tradicionalmente usan.

Como paso previo a la consecución de este objetivo, en primer lugar, se ha realizado un estudio del proceso de desarrollo de videojuegos, así como de las diferentes herramientas existentes en la industria y la literatura, que van en la misma dirección, para conocer su estado actual, cuáles son sus puntos fuertes, así como las facetas que consideramos deben ser mejoradas. Este estudio está descrito en el Capítulo 2 del presente trabajo.

Partiendo de las conclusiones extraídas en dicho estudio, realizamos un estudio sobre la utilidad de *Behavior Bricks* como editor de comportamientos para diseñadores, sin utilizar programación por demostración que se muestra en el Capítulo 3, en el que constatamos la necesidad de ir más allá de un editor de comportamiento tradicional para que los diseñadores pudieran realmente crear comportamientos de forma autónoma, ya que demostramos la interrelación existente entre los conocimientos de programación y los resultados obtenidos usando *Behavior Bricks*.

Así pues, en los Capítulos 4 y 5, desarrollamos tanto el modelo como la metodología de uso del modelo en el que se integran árboles de comportamiento y programación por demostración usando *Behavior Bricks*, para crear árboles de comportamiento por parte de los diseñadores, sin necesidad de programar, con una serie de experimentos que demuestran la efectividad del modelo. Así mismo, se detallan las diferentes técnicas empleadas para crear el modelo de comportamiento y sus ventajas e inconvenientes.

Por último en el Capítulo 6 presentamos las conclusiones finales de la tesis, junto con un resumen de las posibles líneas de trabajo futuro.

1.3. Nuestras aportaciones

Las principales contribuciones de esta tesis están orientadas hacia la extensión del modelo de árboles de comportamiento mediante el nodo denominado *Trained Query Node*, que permite extender la funcionalidad de los árboles de comportamiento, permitiendo aplicar técnicas de aprendizaje automático en el mismo, lo que posibilita la creación de comportamientos sin necesidad de programar.

Parte de este trabajo está ligado a la creación de *Behavior Bricks* que está publicado en la *Asset Store* de *Unity*[3] y que es una de las herramientas gratuitas más utilizadas, de la que formé parte durante los dos primeros

[3] `http://bb.padaonegames.com/`

años del doctorado, junto con otros miembros del Grupo de Aplicaciones de Inteligencia Artificial (GAIA) de la Universidad Complutense de Madrid. Su modelo teórico se publicó en (Sagredo-Olivenza et al., 2014). Como parte de la contribución en *Behavior Bricks*, se creó un *middleware* para desarrollar extensiones del editor de *Unity* denominado *UHotDraw* del que se publicaron los artículos (Sagredo-Olivenza et al., 2015a, 2013).

Después de esta fase inicial, mientras se desarrolló la segunda iteración del editor por parte de otros compañeros del grupo de investigación, donde se perfeccionó el entorno de ejecutor y se dotó de mayor versatilidad al editor, se realizó una investigación sobre la utilidad de *Behavior Bricks* como herramienta para crear comportamientos para diseñadores que se publicó en (Sagredo-Olivenza et al., 2015b)[4] y en (Sagredo-Olivenza et al., 2015c) y de cuyas conclusiones surgió la necesidad de extender los árboles de comportamiento con programación por demostración, incluyendo ésta dentro de *Behavior Bricks*, mediante el nodo *Trained Query Node* y del que se publicaron artículos en (Sagredo-Olivenza et al., 2017a)[5]. También se publicó un artículo en (Sagredo-Olivenza et al., 2016) y en la revista IEEE Transactions on Computacional Inteligence and AI in Games en la edición especial AI-based and AI-assisted Game Design[6], de la cual, en el momento de presentar la tesis, estamos en fase de corrección de la primera revisión. Posteriormente extendimos el sistema para permitir utilizar diferentes sistemas de aprendizaje automático. Una de las técnicas introducidas fueron redes de neuronas. Se publicó un artículo en (Sagredo-Olivenza et al., 2017b)[7] donde se comparaban los resultados obtenidos con las redes de neuronas y otros modelos y cómo utilizar redes de neuronas y árboles de comportamiento, permite mejorar los resultados obtenidos al poder dividir los problemas complejos, en problemas más simples, más fáciles de entrenar por demostración.

[4]Publicado en el volumen 9353 de Lecture Notes in Computer Science (LNCS) e indexado como congreso Core C en `http://www.core.edu.au/`

[5]Congreso indexado como congreso Core B en `http://www.core.edu.au/`

[6]`http://www.graham-kendall.com/TCIAIG/special-issues/` con factor de impacto de 1 en 2015

[7]Congreso indexado como congreso Core B en `http://www.core.edu.au/`

Capítulo 2

Estado del Arte

2.1. Contexto general

El desarrollo de un videojuego es un proceso multidisciplinario que involucra a multitud de profesionales con diferentes conocimientos, diferentes anhelos, metas y formas de entender en sí mismo los videojuegos. Para llevar a buen puerto el proyecto, es muy importante que todos los miembros del equipo estén bien coordinados y que tengan en la cabeza la misma idea del juego. Y esto no es siempre posible ya que es muy común que roles diferentes entiendan el juego de forma diferente.

Por ejemplo, los diseñadores (muchas veces con poca formación técnica), pueden proponer características del juego difícilmente realizables o muy costosas. Por otro lado, los programadores pueden intentar interferir en las decisiones de diseño, en el mejor de los casos intentando desde su punto de vista mejorar el juego, pero adoptando un rol que teóricamente no le pertenece. El proceso de negociación de qué es indispensable para el juego y de qué no lo es, así como qué cantidad de ideas permean desde el equipo técnico hacia el de diseño, es un proceso complejo debido a que en multitud de ocasiones ninguna de las partes está dispuesta a ceder.

Ahí es vital la presencia del productor que es el encargado de mediar entre los diferentes equipos, teniendo en mente siempre los costes de desarrollo. Así pues, podemos decir que los principales roles dentro del desarrollo de un videojuego son los siguientes:

- **Grafistas**: son los encargados de crear los recursos artísticos que hay en el juego como: los personajes, los escenarios o las animaciones. Entre otras, sus principales tareas son el arte conceptual (son los bocetos iniciales que sirven para plasmar la idea y sobre los que se construirá el juego), los modelos 3D o sprites 2D del juego, tanto personajes como escenarios, las animaciones y la estética de los menús.
- **Diseñadores**: son los encargados de crear el concepto de juego y es-

cribir las reglas del mismo. Es decir, a qué se juega y cómo se juega. Es un rol creativo, pero también en cierta manera productivo ya que normalmente se encarga de coordinar a los equipos artístico y técnico (McGuire y Jenkins, 2008). Su cometido es idear un juego que sea atractivo de jugar y por tanto, tienen en cierta manera la iniciativa en las decisiones. Alguna de las cosas que son definidas por el diseñador siguiendo el modelo MDA (Mechanics, Dynamics, and Aesthetics) (Hunicke et al., 2004):

- **Las dinámicas de juego** que son aquellas necesidades e inquietudes humanas que motivan a las personas en general.
- **Las mecánicas** de juego que el usuario realiza para llevar a cabo las dinámicas, es decir las cosas que el jugador puede hacer en el juego y que son representadas normalmente con algoritmos o ecuaciones matemáticas por los programadores en la implementación.
- **La estética del juego**, desde un punto de vista general. La responsabilidad última de la estética es del equipo de arte del estudio, pero la idea o estilo artístico inicial del juego es decidido y sugerido inicialmente por los diseñadores que idean el juego.
- **La historia, los personajes y el comportamiento** de dichos personajes y cómo interaccionan con el juego y el jugador. Están fuertemente relacionadas con las mecánicas jugables y la estética.

- **Programadores**: son los encargados de hacer realidad en un software, todas las ideas que los diseñadores han realizado, con los assets o recursos que los artistas les proporcionan. Son los encargados, en definitiva, de lidiar con la tecnología subyacente del juego y deben decidir qué tecnología toman prestada de terceros o qué tecnología es desarrollada internamente por el estudio.

  Tradicionalmente, por ejemplo en los años 80, los estudios solían crear su propia tecnología. Sin embargo actualmente los estudios tienden a delegar en herramientas profesionales muchas de las tareas. Por ejemplos es muy extendido el uso de motores de videojuegos comerciales como son **Unreal Engine®**[1] o **Unity®**[2].

  Estos motores proporcionan gran parte de la tecnología básica necesaria para crear un juego y son en si mismos un compendio de librerías de los propios creadores del motor y de otros, integradas y engranadas para que funcionen correctamente. Esto se ha convertido en un estándar debido a la magnitud de los juegos actuales, que hace totalmente inviable crearlos desde cero para la mayoría de estudios.

[1] `https://www.unrealengine.com/`
[2] `https://unity3d.com/`

- Productores: Son los encargados de mediar entre las tres partes anteriores, velando por la viabilidad del proyecto. Ellos también se encargan de priorizar las tareas, de resolver conflictos en los diferentes departamentos de la empresa y mantener los costes dentro de los limites establecidos.

Luego existen roles menos importantes que los anteriores, pero que no podemos obviar, sobre todo en juegos de cierto tamaño como son:

- **Marketing**: Es el encargado de dar a conocer el juego al público. Normalmente su cometido se restringe a transmitir las bondades del producto a sus posibles compradores, pero en ocasiones tiene parte activa en el desarrollo, ya que el equipo de marketing hace un estudio de mercado con los datos recogidos, tanto de las métricas que envían los propios juegos, como mediante el estudio de la opinión en las redes sociales, con los que saca conclusiones y solicita modificaciones en el diseño del juego, para tratar de conseguir mayores ventas.

- **Localización**: Es el encargado de traducir el juego. Normalmente es un equipo externo, pero también a veces existe un equipo interno destinado a tal tarea, sobre todo cuando el estudio es muy grande. Dependiendo de la magnitud de la compañía, pueden ser un departamento muy grande que también puede influir en el desarrollo del juego ya que en ciertos países hay restricciones de contenido o expresiones que pueden no sonar bien o que no son traducibles.

- **Testing**: Son los encargados de probar el juego, es decir, de verificar si las indicaciones dadas por el diseñador se cumplen en el juego final, así como, medir la estabilidad del programa y la satisfacción del usuario final. Deben recibir las especificaciones esperadas de los diseñadores y también pueden aportar ideas y modificaciones al diseño si consideran que el juego es o no divertido, ya que suelen ser jugadores habituales de videojuegos. Al equipo de testing se le suele denominar habitualmente el equipo de QA de las siglas en inglés *Quality assurance*[3].

Todos estos roles se pueden agrupar en tres tipos de equipos de desarrollo en un videojuego:

- *El equipo técnico*, que estará compuesto por los programadores, los proveedores de herramientas internas, los que se encargan del *backend* del juego (servidores, estructura del flujo de desarrollo, etc) y en general de todo lo relacionado con la tecnología dentro del juego en sus diferentes vertientes.

[3]ISO 9000:2005, Cláusula 3.2.11

- *El equipo artístico*, que se encargará de crear el guión del juego, los modelos, las texturas, el sonido.
- *El equipo de gestión*, que realiza tareas transversales en el desarrollo como el equipo de producción, marketing, localización o garantía de la calidad (QA).

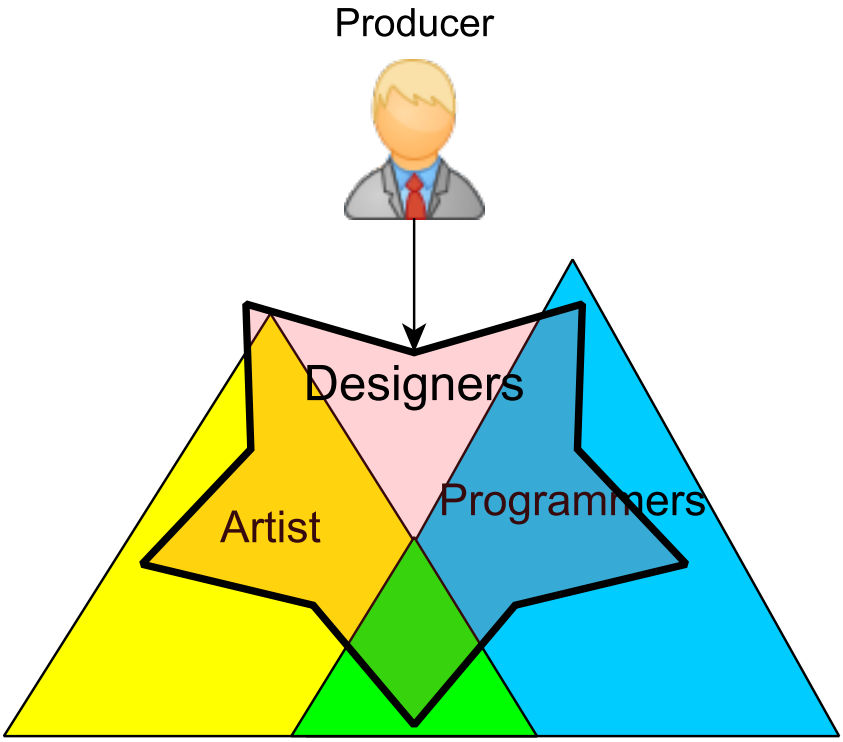


Figura 2.1: La relación entre programadores, diseñadores, artistas y el rol del productor como supervisor.

Podemos decir que el pegamento que une al equipo técnico con el artístico es el equipo de diseño del juego (Véase Figura 2.1). Tiene una gran parte artística, ya que idean el juego y sus reglas para que sea divertido o transmita la experiencia que el diseñado quiere transmitir, pero cierta parte técnica ya que sus decisiones implican crear código, herramientas de desarrollo, definir tablas, ecuaciones o descripciones de los comportamientos de los personajes del juego, que normalmente necesitan de un lenguaje formal para que no sean ambiguas.

Así pues, podemos decir que el equipo de diseño está a caballo entre lo artístico y lo técnico. Debido a esto, podemos encontrar una plétora de perfiles en los diseñadores profesionales, desde programadores reconvertidos a

diseñadores, a artistas, periodistas, físicos, matemáticos y un sinfín de perfiles más. Los requisitos para ser un buen diseñador, que por ejemplo formula *Jesse Schell* en su libro (Schell, 2014), son realmente difíciles de encontrar en una sola persona, ya que engloban conocimientos tan dispares como animación, arquitectura, cinematografía, programación, escritura creativa y un largo etc. Por lo que la solución suele ser tener un equipo de diseño multidisciplinar. Esto implica que las herramientas de desarrollo internas, por ejemplo el editor de niveles, el de comportamientos, los ajustes de juego, la introducción de diálogos, etc, no pueden en principio suponer que los diseñadores van a saber manejar tecnologías que quizás no conozcan, como pueden ser lenguajes de script, complejos ficheros de configuración en JSON u otros formatos.

Las herramientas que usen los diseñadores deben ser sencillas, fáciles de aprender y muy visuales. Por eso, la mayoría de herramientas profesionales para crear videojuegos incluyen editores gráficos para crear los escenarios, herramientas que permiten esculpir terrenos, editores de comportamientos o de programación visual, o en su defecto lenguajes de scripting más fáciles de usar que el código a bajo nivel que constituye normalmente el núcleo del juego, herramientas gráficas para relacionar las animaciones de un personaje, etc.

2.1.1. La inteligencia artificial y la creación de comportamientos

En videojuegos, la utilización de técnicas de inteligencia artificial (IA) tienen un largo recorrido, debido a que, en todo momento, se han intentado recrear agentes o personajes dentro de los mismos, que debían reaccionar con cierta autonomía e inteligencia dentro del juego. Gracias al aumento de la potencia de cálculo y de la mayor demanda de realismo general en los juegos, que se ha ido produciendo con el paso de los años, la inteligencia artificial ha ido ocupando un papel cada vez más importante en el desarrollo del videojuego.

Muchos videojuegos han sido alabados precisamente por su buen uso de la inteligencia artificial y su buena integración en la faceta jugable (Champandard, 2007c). Algunos ejemplos son: Sin City (Maxis®, 1989-2013), Half Life (Valve Corporation®, 1998), Total War (The Creative Assembly®,2000-2017), Los Sims (Maxis,2000-2015) donde se aplicó por primera vez objetos inteligentes que aportaban una enorme escalabilidad a los comportamientos de los personajes, Halo (Bungie®, 343-Industries®, 2001-2016) donde se implementaron por primera vez los árboles de comportamiento o Black and White (Lionhead Studios®, 2001-2003) donde se utilizaban técnicas de aprendizaje automático como redes de neuronas, siendo uno de los primeros en ponerlas en práctica junto con Creatures (Millennium Interactive®, 1996).

Figura 2.2: Captura de Black and White 2, uno de los primeros juegos en utilizar aprendizaje automático.

Otros utilizan técnicas de inteligencia artificial y más específicamente de *aprendizaje automático* para modelar los comportamientos de los NPCs y vehículos, por ejemplo en el juego Dirt Track Racing se utilizaron redes de neuronas para modelar el comportamiento de los coches (John E. Laird, 2005).

Así pues, aunque ha habido cierta controversia en utilizar o no estás técnicas en videojuegos, sí que se han utilizado en casos puntuales y con buenos resultados. Pero la utilización de estas técnicas normalmente siempre ha venido acompañada de una programación más controlada por los programadores en la mayor parte del comportamiento. Por ejemplo, en el videojuego Black & White (Figura 2.2) se utilizaban redes de neuronas y árboles de decisión para modelar el comportamiento de la criatura, pero las acciones básicas de ésta estaban programadas. En general, para construir un sistema fiable, se deben tener desde nuestro punto de vista las dos vertientes: la parte controlada y programada y la parte aprendida.

Sin embargo, existe una necesidad por parte de la industria de crear herramientas que ayuden a los diseñadores y programadores a construir los comportamientos de forma más sencilla, ya que se tiende a facilitar el desarrollo de videojuegos a un público más amplio. Es decir, de no limitar el uso de las herramientas solamente a los trabajadores más técnicos, sino también a otro tipo de roles que podríamos denominar más artísticos. De esta

idea surgen multitud de herramientas que tratan de simplificar la creación de comportamiento en la mayoría de motores comerciales. La creación de estas herramientas es algo que lleva siendo abordado, tanto por la academia como por la industria del videojuego, desde hace años y cuya evolución no ha parado en la actualidad.

Por un lado, en la industria del videojuego se han utilizado diferentes enfoques, tratando de reducir el tiempo de desarrollo y aumentar la accesibilidad para diseñadores y desarrolladores en general con menos habilidades de programación. De esta búsqueda se han ido perfeccionando una serie de herramientas y técnicas que se han estado implementando en los principales motores de videojuegos actuales. Éstos, como veremos, centran sus esfuerzos en crear modelos más sencillos de programación con un mayor componente visual. Pero no suelen plantear modelos donde los usuarios puedan crear IA usando técnicas de generación automática de comportamientos. Está búsqueda de herramientas para simplificar la creación de IA, se puede ver en numerosas investigaciones que han intentado hacer accesible la creación de prototipos a diseñadores, sin necesidad de programadores (Neil, 2012).

A día de hoy, hay herramientas de producción simples que permiten a los diseñadores crear videojuegos. El modelado visual ha demostrado, según ciertos diseñadores, que es un gran aliado para comunicar la idea que el diseñador tiene del juego o de una parte del mismo, como puede ser una mecánica, a otros miembros del equipo, algo que es crucial en el rol de diseñador (Almeida y da Silva, 2013). Por este motivo es imprescindible construir herramientas visuales que ayuden a estos diseñadores a expresar sus ideas a otros en todos los niveles de desarrollo.

A pesar de estos avances, como veremos en el caso de la creación de comportamientos, a veces estas herramientas no son totalmente usables por diseñadores sin cualificación técnica, por lo que se requieren de herramientas más específicas que ayuden a los diseñadores en su cometido.

Inicialmente la mayoría de videojuegos usaron el modelo de máquinas de estados finitos para implementar la lógica de los comportamientos en los NPCs. Este modelo era (y es) muy útil porque utiliza un lenguaje entendible por los diseñadores. Es relativamente fácil que un diseñador exprese el comportamiento en forma de una máquina de estados. Sin embargo con el tiempo se ha extendido el uso de árboles de comportamiento como el modelo más usado en la industria, sobre todo en los juegos más complejos, debido a que este modelo tiene ciertas ventajas sobre las máquinas de estado, por ejemplo que son mucho más escalables (Champandard, 2007b) o que a diferencia de las máquinas de estado son *Turing completos* (Vasudevan, 2015).

Esta ha sido la hoja de ruta de la industria en los últimos años. Cada vez más herramientas y más accesibles a todos, incluso los principales motores de juego han implementado tiendas de extensiones, donde los desarrolladores

pueden publicar sus propias herramientas, como la Assets Store de Unity[4].

Sin embargo, por el lado de la academia y de las publicaciones científicas, se va justo en dirección contraria. Se busca sistemáticamente crear comportamientos mediante técnicas de aprendizaje automático o de programación por demostración, para intentar que no se necesite programar los comportamientos directamente, ni siquiera usando herramienta visuales, si no que se puedan inferir en base a datos almacenados por jugadores. Por otro lado, existe otra aproximación basada en técnicas de aprendizaje por refuerzo van Otterlo y Wiering (2012), algoritmos genéticos Mitchell (1998) u otras técnicas similares, que buscan crear comportamientos jugando miles de partidas y aprendiendo de estas de forma automática.

La premisa de nuestro enfoque es precisamente la de construir un marco de trabajo, donde aplicar métodos de aprendizaje automático sea fácil y que se integre con la dirección que está tomando la industria, que es la creación de comportamientos mediante editores visuales, bien sean árboles de comportamiento u otro tipo de formalismos. Nuestra aproximación parte de la base de que es el modelo gráfico, en nuestro caso los árboles de comportamiento, el eje del comportamiento del NPC y que el aprendizaje se aplica sólo a ciertas partes del mismo, aquellas que interese a los desarrolladores.

En los siguientes apartados se discutirán diferentes métodos y herramientas que la industria utiliza para crear comportamientos, mientras que en el apartado 2.5 se describirá el estado del arte de la academia sobre la programación por demostración y las diferentes aproximaciones, integrando este tipo de técnicas con árboles de comportamiento.

2.2. Motores de videojuegos

En la industria, sobre todo en los últimos años, se tiende a utilizar cada vez más motores comerciales de videojuegos, sobre todo desde la popularización de Unity como uno de los motores más usados. Según datos recogidos en *Statista*, que se pueden ver en la imagen 2.3, donde se muestra el resultado de una encuesta realizada en Reino Unido en 2014. *Unity* es el motor más usado por los desarrolladores británicos con el 62 % de los encuestados, frente a 12 % de *Unreal Engine*. El 47 % sigue usando motores propietarios internamente en alguno de sus juegos. En la encuesta, los participantes podían elegir más de una opción, ya que en un estudio se pueden estar usando más de un motor de desarrollo a la vez, por este motivo, los porcentajes superan el 100 %.

El éxito de los motores comerciales es debido a la cada vez mayor complejidad de los juegos a desarrollar, así como el abaratamiento de los motores y las necesidades de construir juegos para múltiples plataformas para amor-

[4] `https://www.assetstore.unity3d.com/`

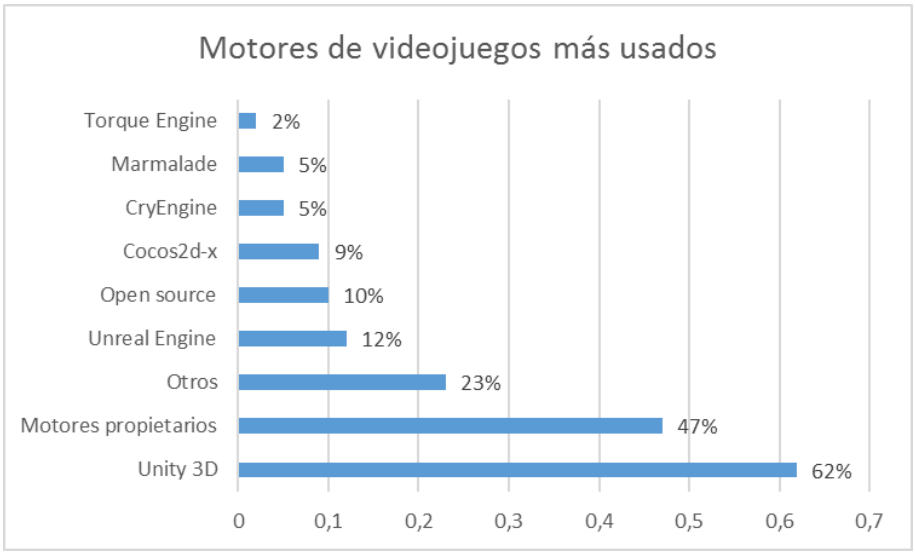


Figura 2.3: Gráfico que muestra el uso de los motores comerciales en Reino Unido en 2014 [a]

[a]Fuente Statista. Extraído de `https://goo.gl/f01gCg`

tizar costes, que con la llegada de los móviles se ha vuelto prácticamente una necesidad. Estos motores tienen como gran ventaja frente a las soluciones propietarias, que su código es utilizado por millones de usuarios, por lo tanto están fuertemente probados por muchos usuarios, que normalmente disponen de multitud de herramientas que simplifican la creación de los juegos, tanto propietarias como en sus tiendas de recursos, donde los estudios pueden publicar sus herramientas internas.

A cambio de disponer de todas estas herramientas , hay un coste a asumir al adquirir estos motores que varía dependiendo del motor elegido. Aunque existen algunos gratuitos o de código libre que no piden royalties, los motores más completos suelen ser motores comerciales. En cualquier caso, el coste de estos motores es muchísimo menor que hace unos años. Así pues y atendiendo al gráfico, podemos decir que los motores más usados actualmente son Unity, Unreal Engine, Cocos2d-x y CryEngine (CrytecK ®). Cocos2d-x es un motor muy centrado en móviles y que ha surgido relativamente hace poco tiempo, no disponiendo de un conjunto de herramientas potentes como el resto de motores más comerciales mencionados anteriormente.

2.2.1. Unity

Unity es uno de los motores de videojuegos más utilizados en la industria, creado por Unity Technologies. Es un motor multiplataforma que soporta una enorme cantidad de plataformas, desde móviles, a diferentes sistemas ope-

rativos de PC, pasando por casi todas las consolas del mercado. Su filosofía es, en palabras de uno de sus fundadores, David Helgason, *la democratización del desarrollo de videojuegos*[5], es decir, hacer accesible a muchos más usuarios, lo que antes sólo podían hacer unos pocos. Unity es creado en 2009 y desde entonces no ha parado de crecer y extenderse a múltiples plataformas. Inicialmente se concentró en juegos para móviles, donde ha dominado el mercado con claridad[6], como se puede ver en la Figura 2.4[7] es el motor más usado en móviles, seguido a gran distancia por Cocos2D-X.

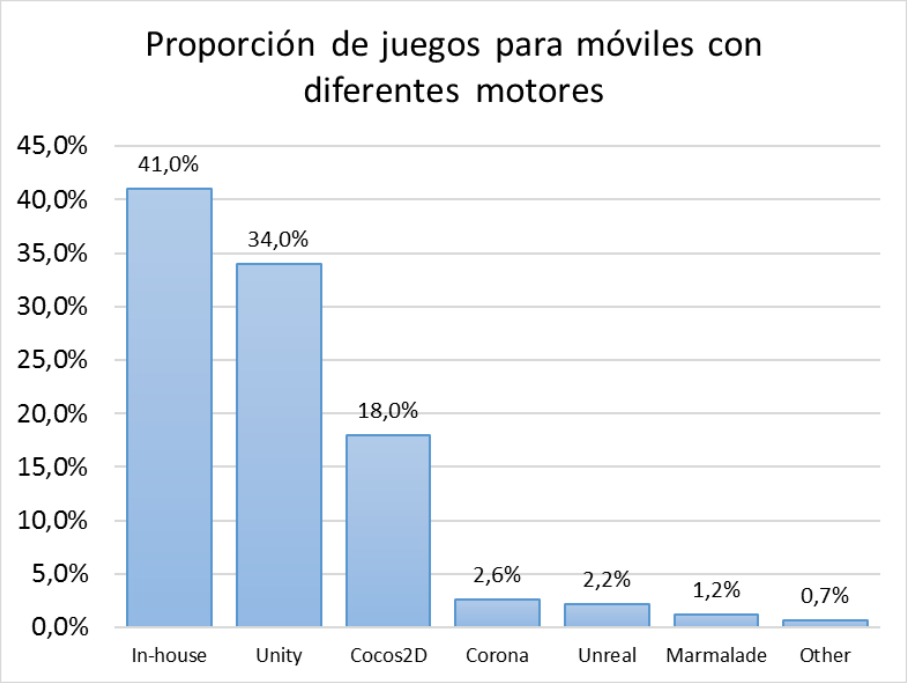


Figura 2.4: Gráfico que muestra el número de juegos gratuitos para móviles creados con Unity[a]

[a]Fuente DNA primer trimestre 2016: Extraido de `https://unity3d.com/es/public-relations`

Su éxito se basa principalmente en un editor intuitivo y fácil de usar, con un lenguaje de scripting en C# que es muy potente y flexible y que le hace más fácil de usar que motores donde se programa directamente en C++. Por otro lado, su política de precios también ha influido mucho en su popularización. El motor desde el principio se podía conseguir de forma gratuita e incluso se podía publicar con él juegos de forma gratuita. Inicialmente el motor en versión gratuita tenía ciertas restricciones, pero actualmente el mo-

[5]`https://goo.gl/9uFMD8`
[6]`https://unity3d.com/es/public-relations`
[7](fuente DNA, primer trimestre de 2016)

tor gratuito no tiene limitaciones a nivel de motor, tan solo en los servicios que ofrece. El coste del motor también era inicialmente bajo en comparación a otros y actualmente se puede conseguir una licencia profesional por una suscripción mensual. Su capacidad multiplataforma, su facilidad de uso y su precio han sido, por tanto, la clave para que el uso del motor se haya extendido tanto. Sobre todo, entre estudios pequeños que no pueden acceder a otros motores más costosos. A pesar de que Unity se asocia más a juegos de móviles o juegos independientes, cada vez más se utiliza en juegos de mayor presupuesto. Aunque en el mercado de juegos AAA (juegos de alto presupuesto con valores de producción muy altos) se suele utilizar más Unreal Engine.

Unity no dispone de ninguna herramienta nativa de creación de comportamientos, sin embargo, dispone de una tienda de recursos para el motor, donde no solamente se pueden descargar recursos gráficos o scripts que realizan tareas, si no que se pueden comprar extensiones del motor de todo tipo. Entre las más populares están herramientas de programación visual y edición de comportamientos como *PlayMaker*, *Node Canvas* o *Behavior Designer*. Hablaremos con más detenimiento sobre algunas de ellas más adelante.

2.2.2. Unreal Engine

Unreal Engine es un motor de juego de PC, consolas y móviles, creado por la compañía *Epic Games*®. Implementado inicialmente en el shooter en primera persona *Unreal*, en 1998. En las últimas décadas, este motor se ha convertido en uno de los más usados para juegos con grandes presupuestos, siendo la base de multitud de juegos como *Mass Effect* (Bioware®, 2007), *Gears of War* (Microsoft®, 2006) o *Bioshock* (2K Games®, 2007), entre muchísimos otros. La versión actual utiliza C++ como lenguaje de creación de comportamientos aunque en su momento disponía de su propio lenguaje de scripting denominado Unreal Script®.

Sin embargo, dicho lenguaje ha sido sustituido en las últimas versiones del motor por una herramienta de programación visual denominada *BluePrints*, que describiremos más adelante y también posee un editor de árboles de comportamiento propio. En cuanto a su portabilidad, la versión cuarta del motor es capaz de exportar juegos a plataformas móviles, a PC, Nintendo Switch, PlayStation4 y Xbox One. Está considerada como la herramienta estándar de desarrollo en muchas plataformas. Además incorpora, al igual que Unity, un motor físico (Physx ® de Nvidia Corp), un editor, tecnología de generación de gráficos en tiempo real muy avanzada, y búsqueda de caminos.

Inicialmente, el modelo de negocio de Unreal Engine era el de licenciar el motor comprándolo por una suma considerable de dinero para poder usarlo. Con la irrupción de Unity y su modelo de negocio de dar parte del motor gratis y monetizar los aspectos avanzados, Epic tuvo que cambiar el modelo de negocio de su motor y permitir que se usase libremente en su totalidad.

El modelo de negocio ha pasado a ser muy similar al de Unity, y el motor puede ser licenciado con licencia profesional con ciertas ventajas de soporte, pero es totalmente accesible para quien lo quiera usar sin pagar nada. La única condición para usar la versión gratuita es que si se publica un juego, parte de los beneficios del mismo deberán ir a parar a Epic, si el juego supera un cierto umbral de beneficios.

Para finalizar, al igual que Unity dispone de una tienda de recursos y extensiones que podemos utilizar[8].

2.2.3. CryEngine

CryEngine es un motor de juego creado por la empresa alemana Crytek, que fue originalmente un motor de demostración para la empresa Nvidia, pero que posteriormente se convirtió en un motor de juego debido a que demostró un gran potencial. Se utilizó por primera vez de forma comercial en el videojuego Far Cry, desarrollado por la misma empresa creadora del motor y que actualmente es propiedad de Ubisoft.

Después de la compra de Ubisoft®, Cryteck creó una versión mejorada del motor, que es la que se utiliza actualmente en su quinta versión. Al igual que Unreal, la irrupción de Unity obligó a Cryteck a cambiar su modelo de negocio. Hasta ese momento compartía modelo de negocio con Unreal Engine. Los usuarios del motor tenían que pagar una licencia de un coste elevado para su uso, no sólo para la publicación del juego. Sin embargo, actualmente es uno de los motores que menos restricciones de uso tiene.

Por ejemplo, no es necesario adquirir licencia para usarlo, incluso si el juego tiene éxito Cryteck no te pedirá royalties. La licencia premium sirve para aportar soporte sobre el motor y el juego, tutoriales y la posibilidad de publicar recursos en la tienda de recursos propia (similar a la de Unity y Unreal Engine)[9].

Al igual que Unreal, es un motor pensado para crear juegos de alto presupuesto, que integra un gran número de funcionalidades dentro del editor y que destaca especialmente por la fidelidad de los materiales y efectos gráficos que consigue mostrar. Usa Lua[10] como lenguaje de script sobre C++ y aunque no tiene un editor visual de programación, sí que dispone de un editor de árboles de comportamiento.

2.2.4. Otros

Cómo motores de propósito general, los tres mencionados anteriormente son los más usados y extendidos en la industria del videojuego. Aunque existen estudios (sobre todo los grandes estudios como Electronic Arts ®, o

[8] `https://www.unrealengine.com/marketplace/`
[9] `https://www.cryengine.com/marketplace`
[10] `https://www.lua.org/`

Ubisoft®) que disponen de motores propios como el Frostbite[11] o Snowdrop[12] adaptados a sus necesidades, en general la mayoría de estudios usan estos motores comerciales para generar los juegos ya que simplifican muchísimo la creación de los mismos. Otros estudios usan motores más sencillos pero que son totalmente gratuitos.

Normalmente estos motores no son tan versátiles como los motores comerciales, ya que no tienen tantas herramientas disponibles y no tienen la capacidad para generar código multiplataforma como tienen los principales motores nombrados. Algunos ejemplos de estos motores son:

- Cocos2D-X[13]: Es un motor multiplataforma orientado a móviles y PC, únicamente para crear juegos 2D. Recientemente, se ha creado un editor que ayuda a crear contenido con el motor, pero inicialmente lo único que proporcionaba el motor era una serie de librerías que simplificaban la creación de videojuegos 2D tanto en Android, como un iOS y Windows phone. Es muy usado debido a la proliferación de juegos en 2D para móviles y a que es un motor totalmente gratuito y en constante evolución, que da buen soporte multiplataforma en móviles con diferentes tecnologías subyacentes.

- Torque Engine[14]: es un motor gráfico código abierto con soporte para 3D y 2D. Al igual que Cocos 2D-X no dispone de las herramientas de los motores como Unreal o Unity y está más diseñado para crear juegos independientes y de pequeño presupuesto. Su gran atractivo es que es de código abierto y por tanto se puede extender y ampliar libremente.

- Source[15]: Es un motor de videojuegos desarrollado por la empresa Valve Corporation®, para consolas y PC, que se usó por primera vez con el videojuego *Counter Strike Source* en 2004. No es de código abierto pero es fácil de conseguir. Valve lo distribuye para crear MODs[16] de sus juegos con la simple compra de alguno de sus juegos. Al igual que el resto de motores de este apartado, no tiene un conjunto de herramientas equivalente a otros motores comerciales, como Unreal o Unity. A cambio de esto, Valve no pide ningún tipo de royalty sobre el motor y su uso.

[11] http://www.frostbite.com/
[12] http://blog.ubi.com/the-division-snowdrop-next-gen-engine/
[13] `http://cocos2d-x.org/`
[14] `https://www.garagegames.com/products/torque-3d`
[15] `https://developer.valvesoftware.com/wiki/Source`
[16] Un mod es una modificación de un juego realizada por uno o varios aficionados

2.3. Técnicas empleadas en la industria para crear comportamientos

2.3.1. Máquinas de estados finitos

Las máquinas de estados finitos (Bourg y Seemann, 2004) son un modelo computacional que realiza cómputos automáticamente sobre una entrada para producir una salida. Están fundamentadas en la teoría de autómatas y han sido ampliamente estudiadas y formalizadas. Para el desarrollo de videojuegos, donde históricamente han sido una de las técnicas más utilizadas para implementar comportamientos, se ven como un conjunto de estados en los que puede encontrarse un NPC y una serie de transiciones o condiciones de cambio de estado que hacen que el NPC cambie la acción que está realizando en un momento dado. Ese formalismo permite una representación visual muy simple e intuitiva en forma de grafo dirigido, donde los nodos son los estados y las transiciones son las aristas. Cada nodo o estado se describe mediante la acción o acciones primitivas que la entidad ejecutará, cuando se encuentre en ese estado.

Cada una de las aristas o transiciones son una descripción de la condición del mundo que dispara esa transición. La ventaja de las máquinas de estados frente a otros mecanismos de creación de comportamientos es su simplicidad, ya que no requieren conocimientos de programación para entenderlas y crearlas; su determinismo y rapidez de ejecución.

Además, es relativamente sencillo crear un sistema guiado por eventos, entendiendo las transiciones como eventos del sistema que se disparan y hacen que el estado cambie, minimizando el coste de evaluación de cada uno de ellos. Además, como tienen una representación visual simple, se pueden construir editores gráficos muy fácilmente.

El tradicional problema de las máquinas de estados estriba en que estas no escalan demasiado bien cuando el número de nodos es muy grande, ya que comienza a incrementar su número de transiciones rápidamente en mayor proporción que los nodos de la máquina (Champandard, 2007b) cuando la complejidad del problema comienza a crecer, por lo que rápidamente comienzan a ser difíciles de mantener, de seguir y controlar por parte del diseñador.

El modelo de las máquinas de estados permite definir estados especiales, considerados como estados finales. Aunque en juegos no es habitual su uso, Behavior Bricks permite indicarlos para dar por terminado el comportamiento. Como veremos, esto nos permitirá integrarlos fácilmente con la otra técnica que describimos en el apartado siguiente, los árboles de comportamiento.

2.3.2. Árboles de comportamiento

Los *Árboles de Comportamiento*, denominados en inglés *Behavior Trees* (BT), son un modelo matemático y formal que posee una representación gráfica en forma de árbol y que se usa en ciencias de la computación para ejecutar planes predefinidos de forma estática (Gonzalez-Perez et al., 2005).

Este modelo ha sido utilizado en multitud de dominios como por ejemplo en la implementación de sistemas de control (Colledanchise y Ögren, 2016), en robótica (Hu et al., 2015) o en videojuegos, entre otras muchas áreas. En concreto, en videojuegos, se suele utilizar para controlar el flujo de decisión en la creación de comportamientos de las entidades del juego y gracias a su fácil representación gráfica, es relativamente sencillo construir herramientas visuales que permitan su edición.

Esta técnica ha ganado popularidad en la última década como modelo estándar de creación de comportamientos para los NPCs en la industria del videojuego. Prueba de ello son los numerosos artículos publicados en la Games Development Conference (GDC) (Isla, 2008a) como en la serie de libros AI Game Programming Wisdom (Champandard, 2008), o en la utilización de juegos como Halo (Isla, 2005, 2008b) creado por Bungie® o Spore[17] (Electronic Arts®) o más recientemente en Saint's Row (The volation) (Aaron Canary, 2014) o The Division (Drew Rechner, 2016) (creado por Ubisoft®).

Los árboles de comportamiento son una mezcla entre las *máquinas de estados finitos jerárquicas* (o HFSM del la siglas en inglés: hierarchical finite state machine) (Rabin, 2002) ampliamente utilizados en la industria del videojuego para crear comportamientos y los Planificadores de Redes Jerárquicas de Tareas (o HTNP de las siglas en inglés Hierarchical Task Network Planners) (Kelly et al., 2007) y vienen a complementar la tradicional utilización de las máquinas de estados finitos, debido a que estas tienen un problema de escalabilidad cuando la complejidad del comportamiento crece demasiado y con un coste computacional más bajo que los planificadores de tareas jerárquicos. Además, permiten definir comportamientos dirigidos por objetivos en contraposición a las máquinas de estado que es dirigido por eventos, sin incurrir en la falta de control y supervisión de los planificadores (Champandard, 2005).

Las transiciones de las HFSM crecen mucho más deprisa que el número de estados, de forma que, si el número de estados es muy grande, el número de transiciones se convierte en inmanejable. Se han utilizado BTs en múltiples videojuegos actuales, siendo uno de los formalismos más comunes. Además, existen editores de BTs en motores de juegos tan populares como CryEngine, Unreal Engine o Unity.

Todos lo nodos de un BT cuelgan unos de otros en una relación jerár-

[17] `http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs`

quica, en cuya cima se sitúa un único nodo que denominamos nodo *raíz*. La ejecución de los árboles de comportamiento comienza en dicho nodo raíz y se propaga a sus nodos hijos y así sucesivamente. Hay dos tipos de nodo, por un lado, están los nodos intermedios que son los encargados de decidir el flujo de ejecución del comportamiento en función de los estados de terminación de sus nodos hijo, mientras que las hojas contienen los comportamientos básicos o primitivos, con las acciones y condiciones del comportamiento.

Así pues, hay dos tipos de nodos hoja claramente diferenciados. Las acciones y las condiciones. Las acciones son aquellas que implican cambios en el entorno. Es decir, al ejecutar una acción normalmente algo sucede en el juego. Los nodos hoja como su propio nombre indica, no tiene descendientes. Sin embargo, las condiciones no influyen en el juego simplemente observan los que sucede. El estado de terminación de todos los nodos (tanto los intermedios como los nodos hoja) puede tomar uno de los siguientes estados:

- Éxito (success) se produce cuando la acción ha conseguido alcanzar su meta o la condición ha sido satisfecha. En los nodos intermedios, el criterio de éxito varía en función del tipo de nodo.
- Fracaso (failure) se produce cuando la acción no ha conseguido alcanzar su meta o la condición no ha sido satisfecha. En los nodos intermedios, el criterio de fracaso nuevamente varía en función del tipo de nodo.
- En proceso (running) se produce cuando la acción aún no ha sido completada.

El nodo padre del nodo ejecutado utiliza esta información de finalización para decidir el flujo de ejecución de sus hijos, para comprobar si ha terminado su ejecución y para devolver su propio estado de finalización a su nodo padre.

Aunque no existe un acuerdo sobre cuál debería ser el conjunto de nodos de decisión, existen diferentes propuestas en la literatura de las que podemos extraer un corpus de nodos razonable. Es muy normal encontrar nodos específicos de un equipo de desarrollo con ciertas características que permiten simplificar la edición de sus comportamientos. Para todos ellos, el orden en el que se definen los hijos es importante porque siempre se ejecutan de izquierda a derecha. A continuación, describimos los nodos de decisión o nodos intermedios más comunes en la literatura:

- Secuencia (Sequence): Los nodos secuencia ejecutan una lista de comportamientos hijos del nodo en el orden que se han definido en el árbol. Cuando el comportamiento que está ejecutándose actualmente termina con éxito (success), se ejecuta el siguiente hijo. Si termina con fallo (failure), el nodo secuencia también lo hará. Devolverá *éxito* si todos los hijos se han ejecutado con éxito.

- Selector (Selector): El nodo selector tiene una lista de comportamientos hijos del nodo y los ejecuta en orden, hasta encontrar uno que se ejecuta con éxito. Si no encuentra ningún hijo que se ejecute con éxito, el nodo termina devolviendo fallo. El orden de los hijos del selector proporciona el orden en el que el nodo evalúa los comportamientos.

- Selector con prioridad (Priority selector): Los nodos selector llevan incorporada una cierta prioridad en el orden de ejecución, pero no reevalúan nodos que ya han terminado. Es decir, una vez que uno de los hijos falla o tiene éxito, ese nodo no se vuelve a ejecutar hasta que se vuelva a dar ejecución al nodo selector. El selector con prioridad se diferencia del selector normal en que las acciones con más prioridad siempre intentan ejecutarse primero en cada iteración, por lo tanto, pueden interrumpir a un nodo que se esté ejecutando en ese momento.

- Paralelo (Parallel): El nodo paralelo ejecuta a la vez todos sus comportamientos hijo. Esta ejecución no debe necesariamente ser paralela a nivel de código, sino que puede realizarse en secuencia dentro del mismo turno de ejecución del nodo paralelo. Los nodos paralelos pueden tener como política de finalización la del nodo secuencia (acabar si todos los hijos lo hacen), o la de los selectores (acabar si uno de sus nodos hijos lo hace).

- Decoradores: Los decoradores son nodos especiales que sólo tienen un hijo y que permiten modificar el comportamiento o el resultado de ese hijo de alguna forma. Existen multitud de nodos decoradores en la literatura y se pueden crear nodos decoradores propios con funcionalidades específicas. Algunos ejemplos son:

  - Decorador de repetición (Repeat): Los nodos de repetición, como su propio nombre indican, repiten la ejecución del hijo un número determinado de veces. Pudiendo ser un número infinito de veces.

  - Decorador de guarda (Guard): Los nodos guarda evalúan una condición antes de ejecutar su hijo. Si la condición no es satisfecha, entonces el nodo guarda devuelve fallo. Si la condición se cumple, devuelve lo que devuelva su nodo hijo.

  - Decorador de negación (Not): Los decoradores de negación invierten el resultado de su nodo hijo. Si un nodo hijo devuelve en progreso (running) el nodo de negación no modificará el resultado. Si el hijo devuelve fallo el decorador lo cambiará a éxito y viceversa.

2.4. Herramientas de creación de comportamiento en la industria

Al igual que existen una serie de motores de juegos creados para facilitar la programación de videojuegos, también existen soluciones para implementar la inteligencia artificial que se usan como middleware en motores propios de los estudios e incluso en motores comerciales. Los principales middlewares de IA que se usan actualmente en videojuegos son **Autodesk Kynapse**, *BabelFlux NavPower*, *PathEngine* y *Havock AI* (Véase Figura 2.5[18]).

Todos estos middleware de IA que hemos mencionado, se centran principalmente en la optimización y búsqueda de caminos y la integración con motores de física y animación. Estos middleware se escapan de nuestro estudio ya que nos centramos en herramientas de creación de comportamientos y no en la optimización y búsqueda de caminos.

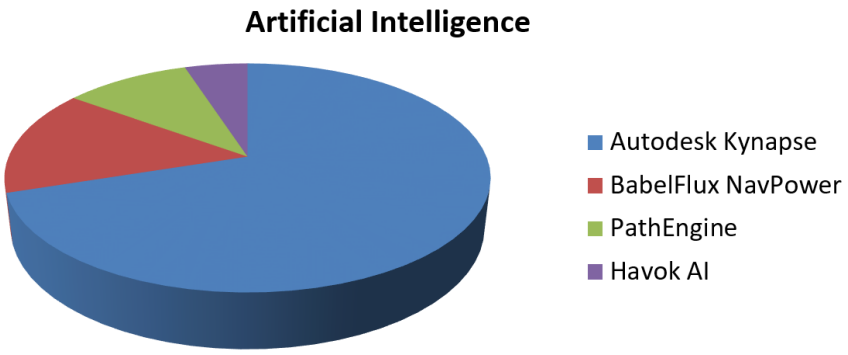


Figura 2.5: Middlewares de IA más usados en 2011[a]

[a]Fuente Game Engines and Middleware in the West, Mark DeLoura, CEDEC 2011

Por lo tanto, en cuanto a tecnologías utilizadas para crear o modelar comportamientos, se suelen usar soluciones propias en motores internos, o bien las soluciones que los motores comerciales aportan. Uno de los modelos más usados son los árboles de comportamiento que están nativamente soportados en *Unreal Engine*, *CryEngine* y aunque no lo están en Unity, sí que existen numerosas extensiones en la *asset store* de Unity que los implementan. Aunque la extensión más usada en Untiy de este tipo de tecnologías ha sido tradicionalmente *PlayMaker*, que no implementa árboles de comportamiento sino una máquina de estados finitos.

[18]Fuente: Game Engines and Middleware in the West, Mark DeLoura, CEDEC 2011

2.4.1. BluePrints

Blueprints[19] es un sistema de scripting visual implementado por Unreal Engine como herramienta para poder programar sin necesidad de tener conocimientos elevados de programación. Es una evolución de *Kismet*, su antiguo editor visual de comportamiento. Kismet a pesar que era una buena herramienta de programación visual, tenía algunas carencias importantes.

Por ejemplo, un programa en Kismet sólo podía aplicarse sobre un único nivel, y por lo tanto, toda la funcionalidad que quisiéramos utilizar en otro nivel, había de ser copiada a dicho nivel, perdiendo así uno de los aspectos más importantes en cualquier herramienta de programación moderna: la capacidad de reutilizar componentes. El nuevo sistema denominado Blueprints solventa estas carencias. Su modelo de ejecución se basa en el concepto de *nodo* como eje central del desarrollo. Un nodo es una caja negra que realiza una acción. Dicha acción puede ser una acción nativa del sistema o una acción creada por el usuario en C++. Un nodo recibe una serie de entradas y devuelve una serie de salidas que conectadas entre sí a otros nodos, van moviendo el flujo de datos y de ejecución de un nodo a otro. Estos nodos modifican el flujo de los datos introducidos inicialmente de forma que en algún punto del grafo, el programa llega a un resultado final. La combinación de estos nodos en un caso concreto pueden reutilizarse en otros Blueprints, algo que, como hemos dicho, no se podía hacer en el anterior Kismet.

Unreal posee multitud de nodos que permiten cambiar el material de un objeto, la velocidad, modificar la cámara, cargar un nivel, realizar operaciones matemáticas, pero permite añadir nuevos nodos mediante programación. También existen nodos que modifican el flujo de control del programa, por ejemplo: **Branch** que produce un salto si se cumple una condición, **forloop** que permite realizar una acción repetida para cada elemento de la matriz que se le pasa por parámetro, **whileloop** que repite una tarea hasta completar una condición o **sequence** que ejecuta una secuencia de acciones en orden. En general existen seis tipos de nodos identificados cada uno con un color diferente:

- Nodos no eliminables, marcados de color purpura. Al menos debe haber uno de estos nodos en la construcción del script y sirven para impedir que estos nodos puedan ser eliminados del diagrama.
- Indicadores de eventos, de color rojo. Los eventos se ejecutan simultáneamente.
- Nodos que operan sobre el la entrada del usuario, marcados de color azul, que son procedimientos, funciones y eventos que hacen algo sobre la entrada y luego pueden no devolver nada, o varias salidas, dependiendo del nodo concreto que se use.

[19] `https://docs.unrealengine.com/latest/INT/Engine/Blueprints/`

- Secuencias de control y macros, marcados de color gris.
- Nodos captadores de datos, denotados de color verde, que se encargan de meter datos procedentes de otras partes del juego dentro del grafo.
- Conversores de tipos, denotados de color cian, que tratan de convertir unos tipos en otros.

Por lo tanto, a nivel formal Blueprint es en esencia un lenguaje de flujo de datos (Harris, 2000). Los Blueprints no son esencialmente un editor de comportamientos, sino una herramienta de programación visual. Estos scripts controlan las diferentes facetas de jugabilidad del juego y son tremendamente versátiles. Por ejemplo, todo objeto capaz de interactuar con el mundo de juego puede ser una instancia de un Blueprint. Desde el propio personaje principal, hasta los enemigos, el HUD, armas, los *power ups* que el personaje puede recoger.

Se pueden crear Blueprints que hereden de casi cualquier clase del motor y añadan o sustituyan la funcionalidad de la misma, así como crear Blueprints que hereden de otros Blueprints. De este modo, disponemos de herencia de clases de una forma visual y potente. Por ejemplo, podríamos crear un Blueprint que defina el comportamiento y los componentes base de un enemigo, y crear otro que herede de éste para sobrescribir su comportamiento, exactamente igual que se podría hacer en un lenguaje de programación convencional. Esto hace al Blueprint una herramienta muy potente, pero también compleja de utilizar si el usuario que la maneja no tiene nociones de programación. Los usuarios sin conocimientos de programación podrán utilizarla, pero probablemente no conseguirán sacarle el máximo partido.

Así pues, los Blueprints, siendo una herramienta que simplifica enormemente el desarrollo de comportamientos, no sería una herramienta apta para todo tipo de diseñadores. En la Figura 2.6 podemos ver una captura de un Blueprint del motor Unreal Engine. En concreto del controlador de movimiento del modelo de personaje en tercera persona que trae por defecto Unreal.

En la figura se puede ver como ejemplo, cómo el primer nodo *obtener control de rotación* lee la información del nodo de rotación y se lo pasa al componente *BreakRotation*, que separa la rotación en ángulos y se conecta con *Make Rotation*, que produce la rotación en el eje z. A partir de esa rotación, el nodo *añadir entrada de movimiento* modifica la dirección del mundo del personaje.

Nuestra principal diferencia con respecto a los Blueprints es que su enfoque está pensado como herramienta de programación de propósito general y no tanto como una herramienta de autoría de IA. El propio Unreal Engine dispone de una herramienta de autoría de IA usando árboles de comportamiento que explicaremos en el apartado 2.4.5.

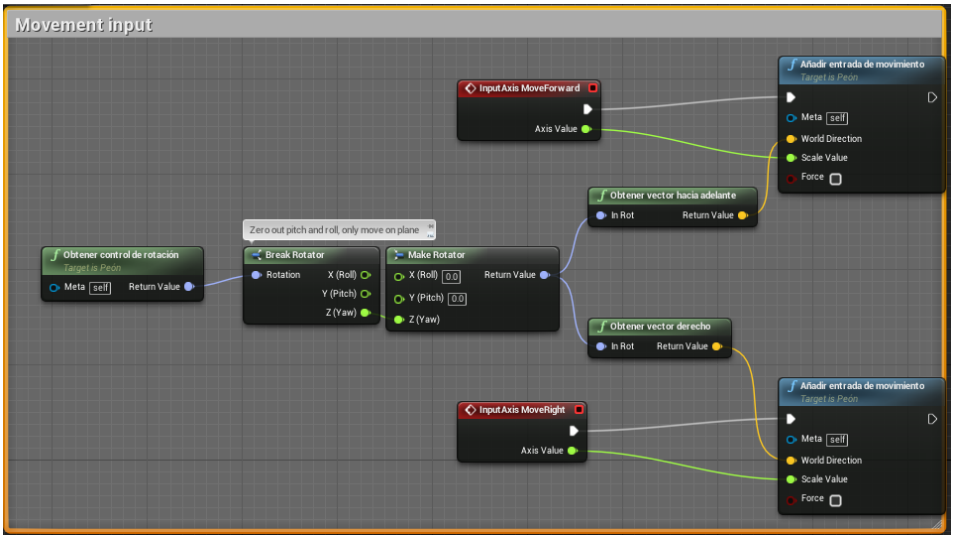


Figura 2.6: Captura del blueprint *input movement* del *Third Person Character Controller* de Unreal Engine

Nuestro enfoque está más orientado a crear una herramienta de creación de comportamientos más que una herramienta de programación de propósito general. Por lo tanto, Blueprints para crear scrips es más versátil que Behavior Bricks, sin embargo nuestra herramienta se adapta mejor para crear específicamente la inteligencia artificial de un personaje en un juego.

2.4.2. PlayMaker

Playmaker[20] es una de las extensiones de Unity más conocidas y descargadas. Es un editor de script visual altamente intuitivo (Kelley, 2016), que usa como modelo de ejecución las máquinas de estados finitos (Rabin, 2002). A pesar de que muchos autores afirman que las máquinas de estados son una tecnología superada por los árboles de comportamiento, como vimos en el apartado 2.3.1, las máquinas de estados son un modelo muy útil para diseñadores (Mohov, 2013) debido a que es un modelo que son capaces de entender con cierta facilidad.

Para definir una acción dentro de PlayMaker, los usuarios tienen la opción de definir sus propios scripts personalizados, o bien, de utilizar un conjunto de acciones básicas predefinidas que incluye PlayMaker. De esta manera, usuarios sin conocimientos de programación pueden crear comportamientos simples mediante un modelo como las máquinas de estados finitos o bloques de código previamente creados. PlayMaker ofrece una gran cantidad de acciones predefinidas.

[20] `http://hutonggames.com/`

En la mayoría de las ocasiones no será necesario crear nuevas acciones para implementar el comportamiento. A pesar de que es un modelo que los diseñadores entienden, el principal problema de las máquinas de estados es su escalabilidad. Crear comportamientos complejos con PlayMaker, implica un crecimiento exponencial en el número de transiciones entre acciones, por lo que no es una herramienta muy recomendable para crear comportamientos realmente complejos.

Una de las características más destacables de PlayMaker es su depurador en tiempo real. Se pueden ver los estados y transiciones resaltados a medida que se ejecutan, de forma que de un vistazo podemos darnos cuenta de cuáles de ellos no lo hacen. La depuración en una máquina de estados es importante ya que puede ser muy difícil detectar un error si la máquina de estados es muy compleja. Además de la depuración en tiempo de ejecución, PlayMaker viene equipado con un *comprobador de errores*, que es capaz de chequear la consistencia de la máquina de estado antes de ejecutarla, detectando multitud de errores de forma previa. Además, PlayMaker es muy extensible y existen multitud de plugins como *Animator Proxy* o *U GUI Proxy Full*, que extienden la funcionalidad de PlayMaker para su uso en animaciones o el diseño de la interfaz de usuario.

En la Figura 2.7 podemos ver el comportamiento de patrulla creado con Playmaker. Los estados en Playmaker pueden ejecutar una secuencia de acciones. Las transiciones entre estados se producen por eventos. Estos eventos se producen en un instante determinado de tiempo. En el ejemplo, el NPC patrulla por una zona hasta que detecta a un enemigo. Cuando el enemigo es detectado, el NPC intenta seguirlo para que quede dentro del rango de ataque. Cuando el enemigo está en el rango de ataque, ataca.

Los eventos son cadenas de texto que se pueden crear al vuelo y que posteriormente se pueden lanzar por código. Algunos eventos del sistema se pueden utilizar como eventos dentro de PlayMaker. Las acciones sin embargo, son código ejecutable que hereda de la clase *FSMAction*. Las acciones se añaden en los estados. Un estado puede ejecutar múltiples acciones en secuencia. Las acciones pueden estar paramétrizadas y leer de las variables de la pizarra su valor.

Nuestra herramienta de comportamiento, *Behavior Bricks* se diferencia de PlayMaker en que utiliza un formalismo, como los árboles de comportamiento, que es mucho más escalable que las máquinas de estados. También PlayMaker tiene menor capacidad de reutilización, que *Behavior Bricks*, ya que sus máquinas de estados no son parametrizables.

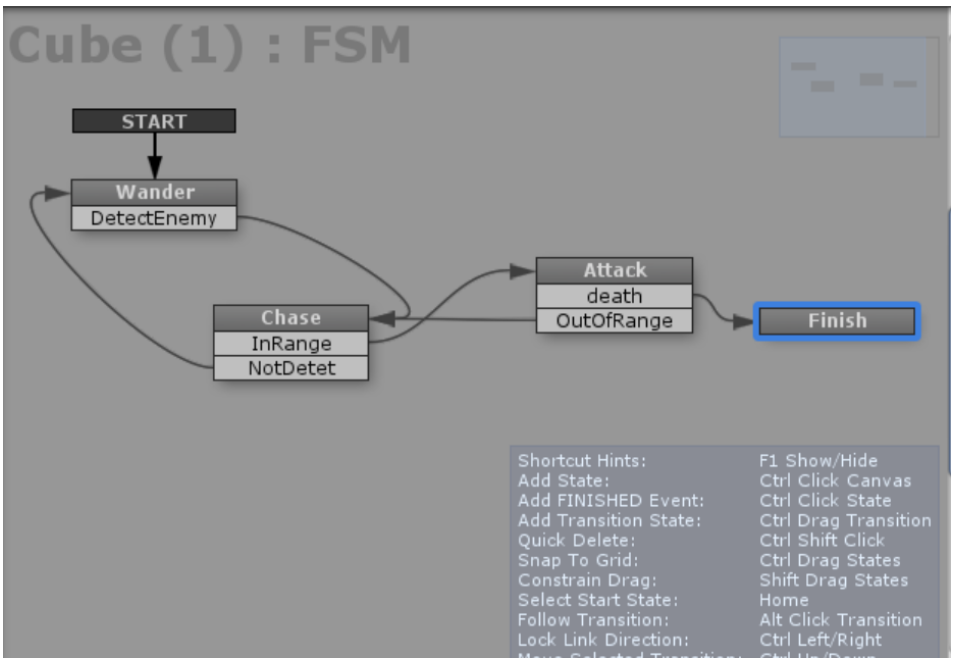


Figura 2.7: Captura de PlayMaker implementando un comportamiento de patrulla

2.4.3. Nodecanvas

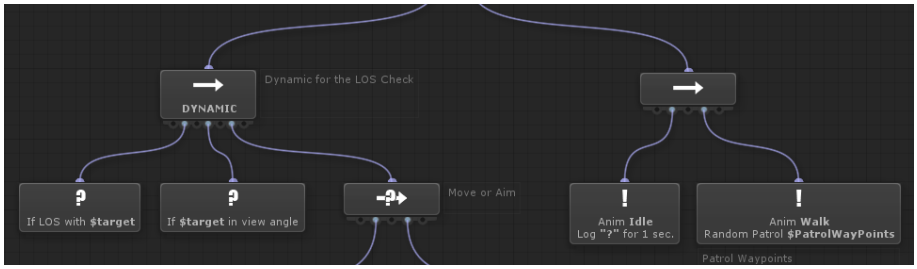
Existen varios editores gráficos para crear comportamientos en la Unity Store. *Nodecanvas*[21] en uno de ellos, pero existen algunos más. Hemos elegido hablar sobre este editor porque tiene ciertas similitudes con Behavior Bricks, nuestro editor de comportamiento que también está disponible en la *Asset Store* de Unity y del que hablaremos con más detenimiento en el apartado 3.4. Aunque otros editores como *Behavior Designer*[22] tienen mucha aceptación también entre los usuarios de la assets store.

Una de las principales características de Nodecanvas es que es capaz de crear tanto máquinas de estados como árboles de comportamiento con la misma herramienta. En la Figura 2.8 se muestran dos ejemplos de comportamientos creados con Nodecanvas.

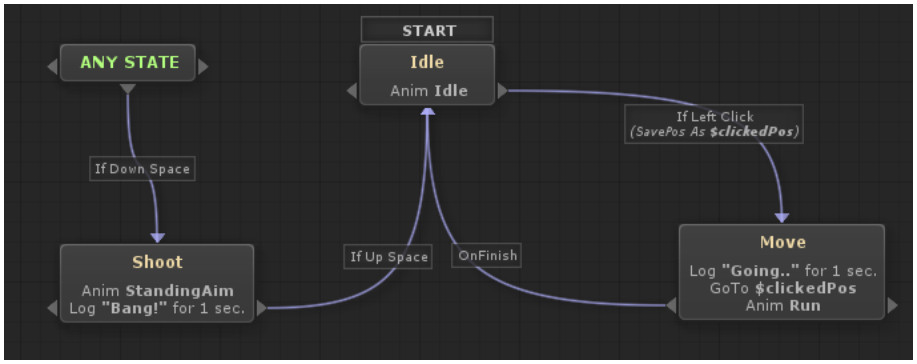
El comportamiento de la parte superior 2.8a muestra un BT que implementa un comportamiento de patrulla aleatorio en la rama derecha, que se ejecuta mientras no exista un objetivo disponible (establecido en la variable **target**). En caso de que exista un objetivo, entonces el comportamiento hace girar al personaje para que apunte al objetivo y lo dispara.

Abajo, en la Figura 2.8b, se utiliza una máquina de estados para cambiar

[21] `http://nodecanvas.com/`
[22] `http://www.opsive.com/assets/BehaviorDesigner/`



(a) Ejemplo de BT creado con Nodecanvas



(b) Ejemplo de FSM creado con Nodecanvas

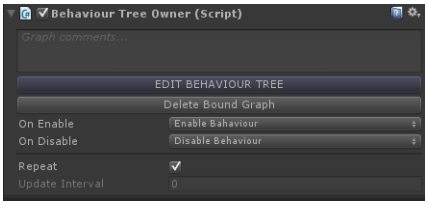
Figura 2.8: Ejemplo de BT y FSM creados con Nodecanvas.

entre el estado de reposo (*idle*) y el estado mover (*move*) que se activa al pulsar sobre el botón izquierdo del ratón. La máquina de estados está implementando un control de tipo "point and click", típico de juegos como *Diablo*. Finalmente, desde cualquier estado, si se pulsa la barra espaciadora, el personaje dispara.

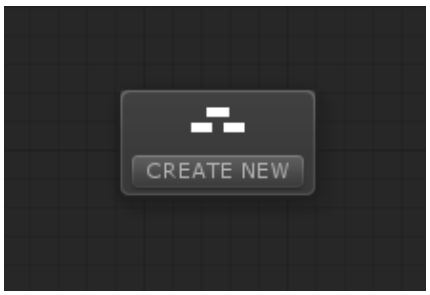
Como se puede ver, la máquina de estados, en este caso, controla la lógica del jugador y el árbol de comportamiento está implementando un enemigo.

No hay muchas herramientas que permitan crear máquinas de estado y árboles de comportamiento simultáneamente. Esta característica de Nodecanvas es similar a una de las principales características de *Behavior Bricks*, ya que ésta herramienta está diseñada para permitir usar árboles de comportamiento y máquinas de estados (Sagredo-Olivenza et al., 2014) simultáneamente en un comportamiento, aunque actualmente no están implementadas las máquinas de estados en *Behavior Bricks*, la abstracción que proporciona el sistema lo permitiría en un futuro.

Además, otra característica interesante de Nodecanvas es que se pueden crear sub-comportamientos y que dispone de una pizarra para compartir variables dentro del comportamiento. Así mismo, soporta depuración en tiempo de ejecución, lo cual es muy útil cuando el comportamiento no hace lo que esperamos, como en el caso de PlayMaker es algo muy importante en este



(a) El BT no es parametrizable



(b) El Nodo Subbehavior sólo permite crear un nuevo sub comportamiento no reutilizar uno ya existente

Figura 2.9: Ejemplos donde se muestra que los comportamientos hechos con Nodecanvas no son reutilizables.

tipo de editores.

Pero su principal problema es el de la reutilización de estos comportamientos. Los BTs deben estar instanciados en una escena como un GameObject de Unity y aunque puedes crear subcomportamientos, estos no están parametrizados, luego sólo se puede usar un subcomportamiento dentro de otro comportamiento, pero siempre van asociados ambos y no se pueden reutilizar estos subcomportamientos en otros comportamientos o incluso usarlos como el comportamiento de otro NPC. Esto se muestra en la Figura 2.9 donde se puede apreciar cómo el BT no es parametrizable más allá de los parámetros de funcionamiento del propio ejecutor, ni tampoco se puede asignar un BT ya creado a un subcomportamiento(véase figura 2.9b), simplemente se puede crear un subcomportamiento dentro de un BT para organizar el BT y mostrar diferentes capas de abstracción. En ningún caso para reutilizar este subcomportamiento en otro comportamiento.

La capacidad de reutilización de los BTs en *Behavior Bricks*, es una de las principales características de nuestro editor de comportamientos y es fundamental para crear sobre él los *Trained Query Node* que explicaremos en el apartado 4.2. Estos nodos aprenderán a seleccionar una subtarea de entre todas las posibles, en base a la información que le proporcione el entrenador. Como se verá más adelante, para *Behavior Bricks* una tarea puede ser una acción programada por código u otro comportamiento creado con el editor gráfico y ambos, para el sistema, son la misma cosa y los trata de forma indistinguible, por lo que puede usarse uno u otro de forma transparente. Se pueden crear comportamientos que luego son reutilizados en otros BTs y que implementan tareas de más alto nivel y posteriormente ser recuperadas por el *Trained Query Node*, en base a los ejemplos proporcionados por el diseñador, en tiempo de ejecución. Estos comportamientos de alto nivel pueden ser también reasignados en tiempo de diseño del BT proporcionando nuevas

implementaciones del mismo para cada NPC.

Hablaremos con más detalle de estas capacidades en el apartado 3.4. Por tanto, ambas tecnologías (BTs y programación por demostración) cohabitan perfectamente con nuestro modelo de árboles de comportamiento y no sería posible implementarlo sobre Nodecanvas.

2.4.4. Modular Behavior Trees en CryEngine

Modular Behavior Tree[23] es el modelo de árboles de comportamiento disponible en CryEngine 3.5, que es una evolución de Behavior Selection Tree[24], su anterior herramienta. Según la propia *Cryteck*, está especialmente diseñado para crear comportamientos a alto nivel sin necesidad de manejar conceptos de bajo nivel como: la gestión de memoria, los punteros, etc. La idea detrás de esta herramienta es la reutilización y la iteración rápida. Básicamente las mismas premisas que tiene *Behavior Bricks*. Su concepto de nodo es muy similar a nuestro concepto de Tarea. Nuevamente el paso de parámetros y la generalización de comportamientos es su punto flaco.

En la propia documentación el creador de Modular Behavior Tree, reconoce que debería tener un modelo de propósito general de variables asignables a variables de la pizarra del comportamiento, pero que por falta de tiempo, reutilizaron el sistema de variables del anterior modelo de BTs. Las variables se pueden definir desde un XML de forma estática y darle valores o leer de eventos del sistema, pero no pueden tomar o dejar valores en una pizarra global. En este ejemplo de XML se puede apreciar cómo las variables *TargetSpotted*, *ReceivedDamage* y *GroupMemberDied* leen de los eventos del sistema *OnEnemySeen*, *OnEnemyDamage* y *GroupMemberDied* respectivamente y después se usan en el nodo *WaitUntilTime*.

```
<BehaviorTree>
 <Timestamps>
  <Timestamp name="TargetSpotted" setOnEvent="OnEnemySeen"/>
  <Timestamp name="ReceivedDamage" setOnEvent="OnEnemyDamage"/>
  <Timestamp name="GroupMemberDied" setOnEvent="GroupMemberDied"/>
 </Timestamps>
 <Root>
  <Sequence>
   <WaitUntilTime since="ReceivedDamage" isMoreThan="5"
    orNeverBeenSet="1"/>
   <Selector>
    <IfTime since="GroupMemberDied" isLessThan="10">
     <MoveCautiouslyTowardsTarget />
    </IfTime>
     <MoveConfidentallyTowardsTarget />
   </Selector>
  </Sequence>
```

[23] http://docs.cryengine.com/display/SDKDOC4/Modular+Behavior+Tree
[24] http://docs.cryengine.com/display/SDKDOC2/Behavior+Selection+Tree+Editor

```
  </Root>
</BehaviorTree>
```

El desarrollo de Modular Behavior Tree en CryEngine está en proceso y en un estado prematuro ya que, a fecha de publicación del presente trabajo, aún no dispone de un editor integrado en la herramienta para generar el comportamiento, la depuración se realiza por una ventana de comandos del juego, así como su visualización y depuración.

2.4.5. Blueprints Behavior Trees en Unreal Engine

Los *Blueprints* son una herramienta de programación visual general. Pero no dejan de ser una herramienta de propósito general para simplificar la programación y no está especialmente bien adaptada para programar comportamientos complejos. Para construir comportamientos, *Unreal Engine* tiene una herramienta específicamente diseñada para crear la inteligencia artificial de los NPCs, que se basa en árboles de comportamiento. Los árboles de comportamiento en Unreal Engine difieren en algunas cosas de otras implementaciones en sistemas similares. Una de ellas es que los *Behavior Trees* están dirigidos por eventos[25].

Los eventos que pueden modificar las variables del mundo son escuchados de forma pasiva. Por lo tanto, se reduce el coste de evaluar las condiciones en cada frame. Esto también facilita a la depuración, ya que sólo cuando se produce un cambio relevante en los datos de entrada, se guardan para poder depurar sobre el mismo. Otro dato interesante y que diferencia los árboles de comportamiento en Unreal de otras implementaciones es que no disponen de nodos hoja condicionales. En su lugar, se han sustituido por decoradores (Véase Figura 2.10), similares a las guardas.

Su principal ventaja es que el árbol resultante es mucho más fácil de leer, ya que el propio nodo indica qué parte del árbol no se ejecutará si la condición no es satisfecha. Por otro lado, mejora la eficiencia de ejecución. Las guardas impiden que se ejecute parte del árbol siempre que no se cumpla la condición. Los nodos condicionales pueden impedir que se ejecute parte del árbol, pero para estar seguros de que esto se va a producir hay que interpretar que hace el nodo padre con dicha condición, ya que dependiendo de tipo de nodo padre, cuando un nodo condición falla, puede dejar de ejecutar el resto de nodos hijo (por ejemplo el nodo secuencia se comporta de esta forma) o no (por ejemplo el nodo selector). Otra modificación de los árboles de comportamiento generales es que no disponen de un nodo *parallel* tradicional.

Epic justifica la ausencia argumentando que los nodos *parallels* son muy confusos y que como pueden producir interrupciones de otros nodos en función de cómo acabe uno de ellos, son difíciles de depurar y crean situaciones

[25] `https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/index.html`

difíciles de entender.

Otro de los argumentos esgrimidos por Epic es que, en un sistema dirigido por eventos, los nodos *parallels* son difíciles de optimizar. En su lugar han creado diferentes nodos *parallel* más simplificados como, por ejemplo, el *Simple Parallel Node* que sólo dispone de dos ramas de ejecución. El nodo hace un símil de ejecución de "Mientras hago la acción A ejecuto la Acción B". La acción A siempre es una hoja, mientras que la B puede ser un subárbol o los *Services* que son decoradores que se pueden añadir a nodos compuestos (*Composite*, por ejemplo: selectores, secuencias o el anteriormente mencionado *Simple Parallel Node*) que permiten realizar una tarea en segundo plano que se repetirá cada cierto tiempo, especificado como parámetro del sistema, mientras se ejecuta la acción principal.

Por ejemplo, en un comportamiento de persecución, un decorador *Services* puede calcular cada cierto tiempo cual es el enemigo a perseguir mientras la acción perseguir se está ejecutando con el enemigo actual.

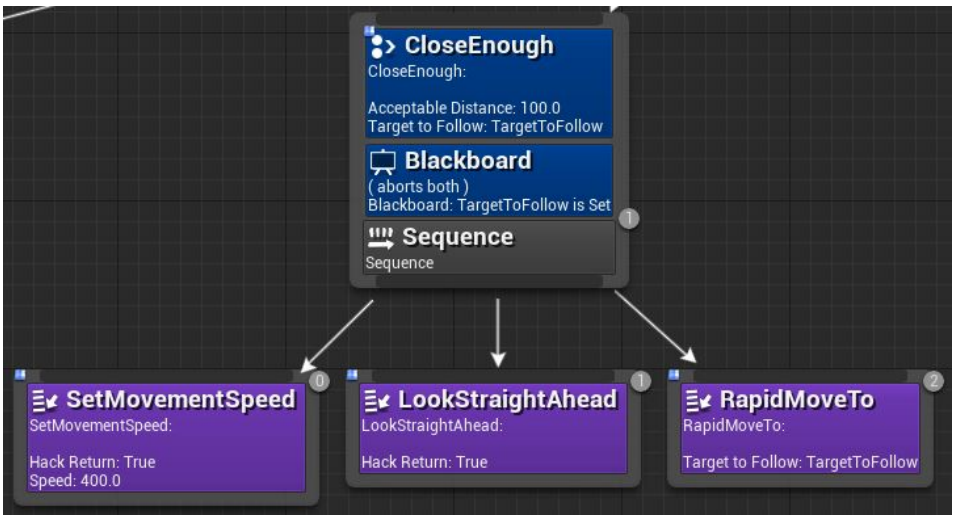


Figura 2.10: Los nodos condicionales en Unreal Engine son decoradores, similares a los nodos guarda

Los Behavior Trees en Unreal Engine son un componente especial que se añade a un NPC. La pizarra se puede añadir de forma separada al árbol de comportamiento como otro componente. Los nodos hoja son sólo tareas ejecutables como ya hemos comentado y se programan mediante Blueprints (o mediante programación convencional).

El IAController es el encargado de manejar las instancias de la pizarra y del árbol de comportamiento. Otra característica interesante es que se pueden crear no sólo tareas nuevas sino decoradores nuevos. En la Figura 2.11 se puede ver un árbol completo de comportamiento de un vigilante estático que está colocado en un punto del mapa.

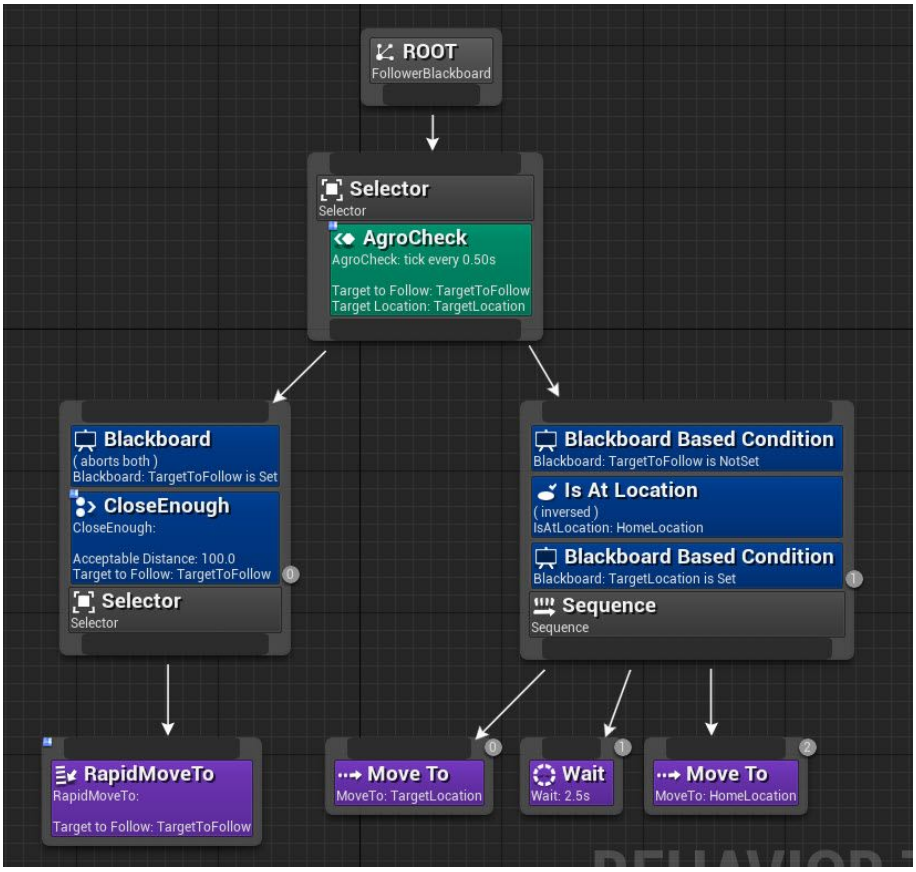


Figura 2.11: Ejemplo de un comportamiento creado con los árboles de comportamiento de Unreal Engine.

El comportamiento busca un personaje que no esté controlado por la IA. *AgroCheck* es un decorador *service* que está constantemente buscando un objetivo plausible. Si encuentra uno, se ejecuta la rama izquierda del árbol, lo que provoca que el NPC lo persiga. Cuando está a una determinada distancia la rama termina. Si el personaje al que sigue se esconde, este se mueve al último visto por el NPC y le espera un tiempo. Cuando ese tiempo finaliza, vuelve a su posición inicial (HomePosition).

La implementación de árboles de comportamiento de Unreal Engine nos parece magnífica, con un montón de buenas ideas. En muchas de ellas coincidimos cuando diseñamos nuestra herramienta. Por ejemplo, nosotros también detectamos que los nodos *parallel* eran muy complejos para los diseñadores, como explicaremos más adelanten en el apartado 3.5.2 y hemos dado especial relevancia a los nodos guarda, que no están incluidos en muchas implementaciones de árboles de comportamiento. Nos parece interesante también el hecho de eliminar los nodos condición de las hojas de los árboles.

Desde nuestro punto de vista, pensamos que aunque las razones que esgrime Epic Games para eliminarlas son correctas, creemos que en ese aspecto debemos dejar libertad al diseñador o programador que use *Behavior Bricks*, si prefiere usar decoradores guarda, nodos hoja condición o ambas cosas. Algunos usuarios pueden provenir de otras herramientas, en las que estén familiarizados con el uso de las condiciones en los nodo hoja y al no encontrarlas, pueden sentir cierto rechazo. También pensamos que si un comportamiento está siendo migrado desde otro sistema, disponer de este nodo tan común en la mayoría de herramientas de árboles de comportamiento, ayudará a trasladar dicho comportamiento a *Behavior Bricks* más fácilmente.

Nuestra herramienta de árboles de comportamiento, nuevamente se diferencia de la implementación realizada en Unreal Engine, en la reusabilidad de los comportamientos y su capacidad de parametrización, así como en la posibilidad de tener múltiples implementaciones de un mismo comportamiento, que son seleccionables tanto en modo de diseño como en tiempo de ejecución.

2.5. Tecnologías utilizadas en la literatura académica

Como se puede apreciar en el apartado anterior, la mayoría de herramientas para crear comportamientos en la industria, van encaminadas a crear comportamientos con editores visuales en torno a dos modelos de creación de comportamientos principalmente. El más usado son los árboles de comportamiento y aunque cada vez menos, el siguiente más común son las máquinas de estados. La mayoría de los tres motores comerciales más usados disponen de editores de este tipo, bien integrados en la herramienta, bien mediante extensiones en sus tiendas oficiales.

Todos estos sistemas están pensados para facilitar la tarea de creación de comportamientos sin programación. La mayoría de ellos, se venden como herramientas para no programadores, pero en el fondo todos estos modelos requieren, para ser usados, que los usuarios tengan ciertos conocimientos en lógica de la programación. De hecho, normalmente acaban siendo los propios programadores los que usan estas herramientas. Sin embargo, en la academia existe una tendencia actualmente en intentar construir agentes o NPCs que aprendan los comportamientos de forma automática. Este proceso permite crear agentes sin necesidad de programación, una vez está construida la herramienta de aprendizaje que se adapte al entorno específico. Normalmente esto es complejo, cada juego tiene su propia interfaz de cómo ejecutar las tareas de bajo nivel. Estas tareas al final deben ser programadas y si no hay una abstracción común, construir el sistema para que el NPC aprenda, puede ser más costoso que realizar los comportamientos de la forma tradicional.

Unir ambos mundos, por tanto, tiene como ventaja, que si usamos como abstracción del modelo de ejecución de un comportamiento los árboles

de comportamiento, entonces podremos comunicarnos con las primitivas de cada juego de la misma forma y por tanto, es mucho más fácil crear una herramienta de aprendizaje de comportamientos de carácter general, como la nuestra. A pesar de todo, se necesitará construir el entorno de entrenamiento y las acciones básicas de la forma tradicional.

En los sucesivos apartados veremos diferentes aproximaciones a la *Programación por Demostración* y otras técnicas empleadas en el mundo académico, similares a nuestra aproximación, como contexto general de las técnicas empleadas en nuestro trabajo.

2.5.1. Programación por demostración

Aprender observando como otras personas realizan una tarea es una forma natural y efectiva de aprender en el ser humano. Es algo que todos hacemos cuando somos pequeños, en la adolescencia y en muchas facetas de la vida. Imitamos, incluso sin darnos cuenta, el comportamiento de nuestros padres, de nuestros hermanos o de nuestros amigos y aprendemos de esa imitación. Desde el punto de vista de las ciencias de la computación, el aprendizaje por observación también es una vía intuitiva y muy prometedora para realizar aprendizaje automático.

Aprender por imitación es un mecanismo que las máquinas pueden llevar a cabo para aprender a realizar tareas de forma efectiva, siempre y cuando el instructor que esté enseñando a la máquina a hacer la tarea, sepa realizarla de forma correcta. Además, para muchos tipos de tareas, aprender observando cómo se realiza una tarea por un experto es una forma mucho más natural que intentar aprender mediante una serie de ejemplos estáticos proporcionados al sistema, debido a que realizándolo de esta forma, se permite modificar de forma fácil y ágil el tipo y la cantidad de información que utiliza el sistema para aprender y el entorno de aprendizaje para el entrenador normalmente le resultará más intuitivo y agradable que aportar al sistema ejemplos estáticos, donde además se pierde el contexto de donde se está obteniendo dicho ejemplo.

En el mundo académico existe una significativa cantidad de trabajo relacionado con la programación por demostración, aunque es un concepto que no está claramente definido y a veces es un poco ambiguo. En la literatura existen numerosas formas de denominar el mismo concepto. Podemos encontrar la misma idea bajo la etiqueta de *Program by demonstration* (PbD) (Ontañón et al., 2010) o *learning from observation* (Lenat, 1983) o *Programming By Example*(PbE), entre otras.

Hablaremos con mayor detalle sobre la programación por demostración en el Capítulo 4, pero para que sirva de aclaración en este momento, podemos definir la programación por demostración como un sub-campo del aprendizaje automático en el que un sistema es capaz de ser programado desde su

propia interfaz, sin necesidad de que el programador tenga conocimientos de programación (Halbert, 1984). Con esta técnica, un programa puede reproducir el comportamiento que los usuarios le han mostrado anteriormente, en lo que podemos denominar *fase de entrenamiento* o de *demostración*, de forma que el sistema es capaz de aprender a realizar la tarea y el usuario que se encarga de enseñar al sistema, no necesita expresar dicha tarea usando un lenguaje de programación, si no simplemente usando la interfaz del propio sistema.

Pero no solamente es capaz de recordar y repetir una secuencia de tareas ejemplificadas por el entrenador, ya que dependiendo de los algoritmos utilizados, el sistema es capaz de generalizar y de encontrar soluciones a otros ejemplos que no se han utilizado para el entrenamiento del sistema. Normalmente es un experto en realizar la tarea el que enseña al programa cómo debe hacerse dicha tarea, para que la información de la que parte el sistema esté lo más limpia posible de casos erróneos que dificulten el aprendizaje. Aunque ésto, en nuestro caso no tiene que ser necesariamente así ya que el diseñador, puede utilizar el propio entrenamiento como una forma de descubrir el comportamiento más adecuado.

En el contexto de la creación de juegos y la creación de comportamientos para NPCs, la idea clave de esta tecnología es que los diseñadores controlen al personaje del que quieren generar su comportamiento, para enseñar al sistema cómo debe comportarse ante diferentes situaciones. El sistema debe recolectar los datos necesarios de lo que está realizando el usuario en el juego, grabándolos en una base de casos o de ejemplos para que, posteriormente, pueda utilizar estos datos para construir un modelo del comportamiento de forma automática.

Una vez que disponemos de los datos necesarios para crear el modelo, este puede generarse mediante múltiples técnicas, cada una tiene una serie de ventajas y desventajas y elegir una u otra no es algo trivial a priori. Por lo que disponer de varias alternativas nos parece un buen acercamiento. También es muy importante que lo aprendido pueda ser analizado por el diseñador o los programadores, para aportar mayor información a estos de qué modelo se está aprendiendo y darles así mayor fiabilidad hacia el mismo.

Dadas estas premisas, si investigamos en la academia, podemos ver que la programación por demostración se ha utilizado profusamente en la literatura, para que las máquinas puedan aprender a realizar tareas en multitud de contextos, por ejemplo, en Floyd et al. (2008) se usa programación por demostración para entrenar agentes que jugaban a la RoboCup (Kitano et al., 1997).

La Robocup es una iniciativa que surgió en 1997, en la que se busca crear agentes autónomos que jueguen en una liga de fútbol, bien sean agentes software o robots. En concreto, Floyd creó un agente que conseguía jugar en la liga de simulación (software) de la Robocup, usando razonamiento basado

en casos (Case-based Reasoning) (Aamodt y Plaza, 1994) en base a cómo jugaban otros agentes creados con otras técnicas diferentes. Para ello, calculaba la distancia entre dos casos con una distancia ponderada por pesos y usaba *k-Nearest Neighbors* (B. W. Silverman, 1989) para construir el modelo, usando algoritmos genéticos para calcular los pesos de la similitud. El *fitness* para evaluar a los individuos que calculaban los pesos en el algoritmo genético (es decir, la medida de calidad de las soluciones dadas por el algoritmo) utilizaba los valores de *precission y recall* (Powers, 2011) dentro de la ecuación de *fitness*, valorando el número de veces que la acción es correctamente elegida.

En general, en robótica, la programación por demostración se ha utilizado profusamente en multitud de trabajos, algunos de ellos resumidos en (Argall et al., 2009), que demuestran que la técnica funciona muy bien a la hora de aprender comportamientos para robots. También se ha utilizado en la creación de agentes que conducen vehículos de forma autónoma en (Pomerleau, 1989), donde una red de neuronas analiza la información recibida por una cámara, y en función de ésta, interpreta qué elementos hay en la calzada, modificando la conducción del vehículo en función de las entradas recibidas. Los datos observados son generados por un conductor experto que conduce un coche y graba todas sus acciones. La salida del sistema es un conjunto de valores que indican la velocidad, ángulo de giro del volante, marcha, etc.

Como demuestra la proliferación de artículos donde se aplica programación por demostración en robótica, en este dominio la técnica obtiene muy buenos resultados. Si tenemos en cuenta que los comportamientos para robots y los comportamientos en videojuegos, a alto nivel, son muy similares, simplemente difieren ambos en cómo se llevan a cabo las acciones básicas a bajo nivel, se puede intuir que el uso de esta técnica en videojuegos y en la creación de comportamientos para NPCs también tendría éxito.

Existen numerosos ejemplos de uso de esta técnica en videojuegos, incluso en videojuegos comerciales, por ejemplo, la programación por demostración se ha utilizado para intentar imitar el comportamiento humano en juegos como *Super Mario Bross* en (Floreano et al., 2008; Ortega et al., 2013), con diferentes aproximaciones basadas en redes de neuronas y algoritmos neuroevolutivos, midiendo los resultados haciendo un *Test de Turing* con usuarios, donde se mezclaban diferentes aproximaciones software, mezclados con partidas llevadas a cabo por humanos y se medía, si un grupo de usuarios era capaz o no de distinguir qué partidas eran jugadas por humanos y qué partidas eran jugadas por los agentes software. Aunque no consiguieron parecer tan humanos como los humanos reales, en muchas ocasiones los agentes software consiguieron engañar a los usuarios, haciéndose pasar por jugadores humanos.

También se realizaron trabajos similares de imitación de la conducción humana en juegos de carreras de coches como en Muñoz et al. (2013) donde

se usaba el simulador de carreras TORCS (The Open Racing Car Simulator) (Loiacono et al., 2010) para guardar la información de diferentes jugadores con parámetros como: la distancia a la línea de meta, las revoluciones por minuto del motor, la posición relativa del coche al centro de la carretera, la velocidad, etc.

El sistema, una vez aprendía el modelo, generaba una serie de salidas mediante las cuales se permitía controlar el coche, como por ejemplo: el ángulo de giro del volante, la presión en los pedales de aceleración y frenado, si se producía o no un cambio de marcha etc. Para entrenar el sistema, también usaban redes de neuronas, una técnica empelada también comúnmente en la industria, como por ejemplo en la serie Forza Motorsport en Drivatar[26], donde se guardan millones de partidas en la nube, que son procesadas por redes de neuronas y otras técnicas de aprendizaje automático para simular el comportamiento humano de los conductores del juego y aplicarlos a la IA de los oponentes.

También se ha aplicado la programación por demostración para conseguir una navegabilidad por el escenario más realista y más cercana a la que realizan los humanos en juegos como el Unreal Tournament (Atari 2006) (Karpov et al., 2013) y en construir bots con comportamiento humano, o en otro conocido first person shooter (FPS) como Quake 2 en (Moriarty y Gonzalez, 2009), donde también se utilizaron como modelo las redes de neuronas, con el algoritmo de entrenamiento Resilient Propagation publicado en (Riedmiller y Braun, 1993), una modificación del algoritmo de *backpropagation* del perceptrón multicapa.

Pero no solamente se ha buscado usar la programación por demostración para intentar imitar el comportamiento humano, también para modelar el comportamiento de un jugador o de un NPC. Por ejemplo, se ha utilizado para crear el comportamiento de un jugador en juegos de estrategia en tiempo real en (Ontañón y Ram, 2011), donde se aprendían planes por demostración en el juego de estrategia en tiempo real denominado Wargus (un clon del juego de estrategia Warcraft II) (Aha et al., 2005) usando Darmok (Weber et al., 2011) un planificador basado en casos online[27], o en (Dereszynski et al., 2011) donde se usaba programación por demostración para generar Hidden Markov Model (Rabiner, 1989) usando partidas *Protoss* vs *Terran* (dos razas diferentes en este videojuego de estrategia en tiempo real) donde se aprendían algunos comportamientos de ciertas unidades *Protoss*. Por ejemplo el *Reaver*, una unidad similar a un tanque que tiene una cantidad finita de bombas muy poderosas, que puede regenerar pero con un coste de recursos y de tiempo. La unidad es muy lenta pero tiene una potencia de fuego muy alta.

Una de las técnicas más usadas en este tipo de artículos es el razona-

[26] `https://www.microsoft.com/en-us/research/project/video-games-and-artificial-intelligence/`

[27] `https://sourceforge.net/projects/darmok2/`

miento basado en casos, aunque se han utilizado todo tipo de técnicas para modelar el comportamiento de los NPCs en la literatura, por ejemplo, podemos añadir a los articulos mencionados anteriormente, otros trabajos como en Rubin y Watson (2011), donde se modela un agente que juega al Texas Holdém Poker (es una versión del estándar juego de cartas poker, en su versión sin límite de apuestas) usando razonamiento basado en casos o en Jaidee et al. (2013) donde también sobre *Wargus* se utiliza esta técnica, pero esta vez empleada para la coordinación de múltiples agentes.

2.5.2. Combinando árboles de comportamiento con técnicas de aprendizaje automático

En todos los trabajos descritos en el apartado anterior, se utiliza la programación por demostración para crear un comportamiento global del agente o NPC que se está entrenando. Quizás el problema a resolver simplemente sea una parte del problema global debido a la complejidad del comportamiento en sí mismo (por ejemplo en el caso de starcraft), pero no se usan aproximaciones híbridas con otras técnicas de programación más convencional, más allá de que las acciones primitivas son acciones programadas en el motor del juego.

Sin embargo, nuestro enfoque es un enfoque híbrido, donde no sólo queremos modelar el comportamiento de un juego, si no que queremos construir una herramienta que sea capaz de modelar el comportamiento de multitud de juegos diferentes. Los ejemplos vistos en el apartado anterior están fuertemente condicionados al tipo de juego en el que se aplican. Además, queremos que nuestro sistema conviva con las herramientas de programación de comportamientos actuales. Así pues, para satisfacer ambos requisitos, usamos una aproximación híbrida como veremos más adelante donde usamos árboles de comportamiento, usados en la industria y programación por demostración.

En nuestra aproximación, la programación por demostración se realiza en algunos nodos concretos del árbol de comportamiento. Es decir, podemos programar partes del comportamiento mediante el árbol de comportamiento y otras partes mediante programación por demostración. En la literatura también se han realizado aproximaciones hibridas usando árboles de comportamiento y otras técnicas de aprendizaje automático. Por ejemplo, en Robertson y Watson (2015), donde se generan árboles de comportamiento con todas las acciones ejecutadas en una secuencia. Estas secuencias son seleccionadas por un nodo selector donde se guarda la descripción del ejemplo guardado.

Cuando llega un nuevo caso en modo ejecución, se comparan las condiciones con el nuevo caso y se selecciona la secuencia de acciones que tiene una pre-condición más similar al caso actual. El árbol inicial se crea con las observaciones realizadas y después se minimiza su tamaño iterativamente

usando *Gapped Local Alignment of Motifs* (GLAM2) para detectar patrones e ir reduciendo el árbol. En Lim et al. (2010) se generan BTs automáticamente para controlar el comportamiento del jugador en el juego de estrategia en tiempo real, DEFCON[28].

El sistema genera árboles de comportamiento de forma aleatoria y utiliza programación genética para hacerlos evolucionar. Esta aproximación es muy interesante, ya que permite crear árboles que pueden ser leídos por los diseñadores, pero perdemos la posibilidad de que el experto (en este caso el diseñador) nos facilite la tarea de crear un árbol de comportamiento inicial con cierto sentido del cual evolucionemos.

Los resultados obtenidos en el experimento son buenos, pero buscan maximizar resultados obtenidos, no como en nuestro caso, que buscamos ayudar al diseñador, de forma que los comportamientos que pueden ser buenos a nivel de efectividad, pueden no ser buenos a nivel jugable (Derek Neal, 2016). Además hay un pequeño handicap para utilizarlo en nuestro enfoque, y es que sería muy complicado codificar una función de *fitness* para cada comportamiento, para saber si el sistema juega bien o no conforme a los requisitos del diseñador.

De todas formas, es una aproximación muy interesante que queremos tener en cuenta para futuros trabajos, ya que es una idea de refinamiento de árboles una vez generados muy interesante. Otra aproximación a la generación de árboles de comportamiento mediante programación genética sobre el juego Super Mario Bross se describe en (Colledanchise et al., 2015), donde se utiliza la observación para generar las condiciones y las acciones que luego el sistema seleccionará mediante programación genética. La programación genética sirve para maximizar la función de *fitness* que determina si el BT generado cumple con la meta esperada.

En estos ejemplos podemos ver cómo en la mayor parte de los trabajos, se utilizan los árboles de comportamiento como formalismo sobre el que generar el comportamiento, en base a diferentes técnicas, la mayoría de ellas encaminadas a la programación genética. A modo de crítica constructiva y de comparación con nuestro trabajo, nuestra herramienta es capaz de generar árboles de comportamiento a partir de los datos de entrenamiento, pero nuestra aproximación de generación de árboles de comportamiento no se basa en programación genética, si no en la creación del modelo de comportamiento usando árboles de decisión.

Esta técnica empleada sobre los datos aportados por el experto, nos garantiza mayor imitabilidad que técnicas evolutivas, donde es más complicado conseguir comportamientos que el diseñador pueda acotar de alguna forma. Además, los árboles que genera nuestra técnica son muy simples y fáciles de entender, lo que permite al usuario editarlos fácilmente. Además, las técnicas expuestas en estos trabajos descritos en este apartado, no tienen en cuenta

[28] `http://www.introversion.co.uk/defcon`

árboles con tareas parametrizables, donde la evolución de comportamientos se complica debido al paso de parámetros y su semántica. Nuestra aproximación garantiza que las acciones que se seleccionan son semánticamente correctas para el nodo a aprender, debido a que son implementaciones de una acción conocida que es la que pretendemos aprender y por tanto conocemos su interfaz.

2.5.3. Otras técnicas de aprendizaje automático utilizadas en juegos

Fuera de la programación por demostración y los árboles de comportamiento, se han utilizado diferentes aproximaciones para modelar el comportamiento de NPCs. Muchas de ellas como ya hemos visto en el apartado anterior, tienen que ver con algoritmos genéticos y sistemas evolutivos, otras con aprendizaje por refuerzo, entre otras técnicas. El problema de estos algoritmos es nuevamente la confiabilidad. Estos sistemas pueden ser interesantes para crear comportamientos nuevos, es decir como sistemas que sugieran comportamientos al diseñador, más que para crear comportamiento automáticamente, debido precisamente a que el diseñador no verá con buenos ojos posibles comportamientos emergentes que se puedan crear con estas técnicas. Pero sí que nos parece como un posible e interesante campo a explorar para conseguir recomendaciones de comportamientos para el diseñador que den nuevas ideas al mismo.

Así pues, en este marco de trabajo, podemos ver muy por encima algunos trabajos interesantes que usan algoritmos neuroevolutivos como en (Avery et al., 2009) donde se usan mapas de influencia en el terreno para coordinar diferentes agentes de forma eficiente, o aplicados a juegos como EvoComander (Jallov et al., 2016).

Otra de las técnicas más utilizadas para generar comportamientos automáticamente es el aprendizaje por refuerzo (Dey y Child, 2013) que permite a los agentes aprender nuevos comportamientos sin la supervisión del diseñador. Algunos tipos de juegos donde se han obtenido buenos resultados usando esta técnica han sido en juegos de estrategia por turnos o juegos de tablero (Samuel, 1959), donde el aprendizaje por refuerzo clásico obtiene muy buenos resultados. Aunque también se ha utilizado con éxito en otro tipo de juego como en el Packman (Bonet y Stauffer, 1999).

Estas otras técnicas, más alejadas de nuestro enfoque, las dejamos aquí presentadas tibiamente, como punta de lanza para un posible trabajo futuro, donde el sistema permita generar comportamientos nuevos en base a los entrenados.

Capítulo 3

Uso de *árboles de comportamiento* por diseñadores

3.1. Introducción

Crear comportamientos para videojuegos es una tarea realmente compleja, sobre todo en los juegos actuales donde abundan personajes que acompañan al jugador durante el juego y que requieren de unos comportamientos realistas para que el jugador los sienta parte del juego, que cooperen con el o que ayuden a la inmersión dentro del mundo del juego, así como unos enemigos que ya no sólo tienen que ser inteligentes a nivel individual, sino que pueden coordinarse entre ellos para sorprender al jugador. Los jugadores esperan unas reacciones realistas en los NPCs y que ejecuten comportamientos que sigan patrones poco predecibles para evitar aburrir al jugador cuando estos aprendan el patrón de comportamiento del enemigo. Pero aún se complica más si cabe el proceso, si tenemos en cuenta que los diseñadores son los que idean esos comportamientos, pero en realidad son los programadores los que los implementan y tiene que haber un proceso de trasbase de información entre ambos que no es nada sencillo en la mayoría de las ocasiones.

Desde mucho tiempo atrás, y debido a múltiples factores como por ejemplo los prolongados tiempos de compilación de un proyecto como un videojuego, que puede estar compuesto por millones de líneas de código, o la facilidad de poder realizar cambios en caliente mientras se juega (es decir modificar los valores de ciertos parámetros mientras el programa está en ejecución) para poder probar y ajustar mejor el juego, en videojuegos se ha tendido a utilizar un paradigma de programación dirigida por datos (Stutz, 2006), el cual intenta que el flujo del programa se controle mediante los datos almacenados en el almacenamiento persistente y no mediante el código compilado.

A pesar de que éstos han sido los motivos principales de usar una arquitectura dirigida por datos, su uso ha ayudado a que los diseñadores tengan mucha más libertad para cambiar elementos del juego, ya que los lenguajes y formalismos utilizados en los ficheros de datos son mucho más legibles y entendibles por los diseñadores con menos conocimientos técnicos, que el código fuente. Así pues, muchos diseñadores han diseñado niveles editando ficheros de configuración en diferentes formatos. Con el tiempo, esta forma de proceder ha ido dejándose a un lado en favor de herramientas más maduras, como los editores de escenarios. Aun así, existen multitud de tareas que no pueden ser llevadas a cabo por los diseñadores usando herramientas y son responsabilidad de los programadores.

Esto no tendría por qué ser malo, si la comunicación entre ambos fuese perfecta o fluida. Cosa que no sucede siempre. No en vano, hay un precioso tiempo de comunicación que se pierde en la coordinación entre diseñador y programador para realizar las tareas, incluso cuando el diseñador tiene claro qué es lo que quiere conseguir, que no siempre es así (de hecho, normalmente, no lo es), se requieren sucesivas iteraciones para conseguir que ciertas características del juego (*features*) sean realizadas como el diseñador había ideado.

Estos problemas de comunicación no son algo nuevo, son un viejo problema en la ingeniería del software que tiene que ver con el problema de la captación de requisitos. Los clientes que solicitan un nuevo sistema normalmente no entienden la importancia de especificar correctamente y de forma precisa las características del sistema que solicitan, las entradas del mismo, así como las salidas (Wiegers y Beatty, 2013).

Si esto sucede con una aplicación comercial normal, en los videojuegos sucede aún más. En este contexto, es el diseñador el que asume normalmente el rol de cliente, salvo que sea un juego realizado por encargo donde el cliente sería externo, y es el programador el que hace de analista y debe interpretar lo que el diseñador desea, actuando en consecuencia. En el caso de los videojuegos, este problema se agudiza, debido a que el reto a abordar es mucho mayor y a que, en la mayoría de las ocasiones, ni siquiera el diseñador sabe cómo acabará el proyecto o si lo que está pidiendo en este momento, acabará finalmente desechado. Porque no hay una fórmula mágica para hacer un videojuego, aquella que nos diga los pasos para conseguir que el videojuego sea un super-ventas, que gusta a los usuarios. Es un proceso con multitud de incertidumbres y más incógnitas que certezas.

Hasta que el desarrollo no está muy avanzado, no está claro si el juego será divertido o no o si transmite la experiencia que se pretendía con él. Por este motivo, los requisitos en un videojuego son especialmente volátiles (Chandler, 2009), lo que requiere una aproximación de gestión más ágil del proyecto (Baskerville y Pries-Heje, 2004; Koutonen y Leppänen, 2013).

Por otro lado, existe una barrera comunicativa entre ambos roles, ya que

el diseñador puede no expresar totalmente lo que tiene en mente o simplemente al ver el resultado del comportamiento, se da cuenta que no es el que esperaba, no se ajusta a lo dicho o se habían pasado cosas por alto cuando se diseñó. Más aún si cabe sucede cuando, después de completar la característica solicitada por el diseñador, ésta es probada por los usuarios o el equipo de testeo y resulta que no es divertida, no se entiende o es demasiado compleja, lo que provoca una nueva iteración y ajuste y repetir todo el proceso de nuevo. Por eso, tradicionalmente, los programadores intentan darle todo el poder que puedan a los diseñadores, para que estos puedan construir partes del juego por sí mismos y que puedan ajustarlas y equilibrar el juego fácilmente, para que este acabe siendo divertido.

De estas y otras necesidades van surgen herramientas como los editores de niveles, los ficheros de configuración, la descripción de entidades en *blueprints* o arquetipos reutilizables, los lenguajes de scripting que simplifican la creación de comportamientos y una serie de sucesivos avances que se han ido realizando para hacer cada vez más autónomos a los diseñadores.

Paradójicamente, en la creación de comportamientos, estos avances hacia la autonomía de los diseñadores no se han llegado a producir con tanta intensidad. Esto es debido, por un lado, a la complejidad de la tarea en sí misma y por otro lado, por la reticencia de los programadores a ceder el control en esta tarea a los diseñadores, por miedo a crear comportamientos que no estén bien realizados o que sean poco eficientes.

En el Capítulo 2.3.2 ya describimos los árboles de comportamiento o como se les conoce comúnmente en su denominación anglosajona, Behavior Trees. Este formalismo es hoy en día uno de los más usados para crear comportamientos para NPCs. En el resto del presente capítulo, describiremos el motivo por el cuál este formalismo ha ganado importancia y cómo lo utilizamos en nuestra herramienta Behavior Bricks, para intentar acercarlo a los diseñadores sin conocimientos técnicos.

3.2. La popularización de los árboles de comportamiento en la industria y su uso como herramienta de diseño

Como ya hemos comentado anteriormente, paulatinamente desde las primeras implementaciones de árboles de comportamiento en videojuegos llevada a cabo en Halo 2® y Halo 3®, gran parte de la industria ha comenzado a utilizar este formalismo para sus comportamientos. A pesar de ello, principalmente sigue siendo una herramienta que suelen utilizar programadores y no tanto los diseñadores. Aunque existen ejemplos de uso de BTs por diseñadores, éstos suelen tener una formación técnica considerable y normalmente suelen ser diseñadores que fueron formados profesionalmente como progra-

madores.

Por ejemplo, en el videojuego Halo 3, la IA a alto nivel, la que coordinaba a los NPCs enemigos entre sí, la realizaban los diseñadores del juego (Isla, 2008a). Ciertamente, la mayoría de los diseñadores de Bungie®, la desarrolladora del videojuego, eran diseñadores técnicos, es decir, diseñadores con conocimientos de programación, lo cual ayudaba a llevar a cabo esta tarea. Sin embargo, en general esto no sucede así. En los estudios generalmente los programadores son los que implementan los comportamientos, y aunque existan editores visuales (creados como extensiones de motores comerciales y muchas veces como herramientas internas de los propios equipos de desarrollo) para que sean los diseñadores los que los implementen sin necesidad de conocer los entresijos de los lenguajes de programación, éstos siguen siendo usados principalmente por los programadores, para facilitarles la tarea de creación del comportamiento y para tener un modelo de ejecución propicio para implementarlos.

Sin embargo, existe una gran variedad de diseñadores que no provienen de una formación técnica en programación, sino más enfocada a lo artístico, que se ven abrumados por la complejidad de estas herramientas. En el fondo, muchas de estas herramientas, como vimos en el Capítulo 2.4, son herramientas de programación visual, que simplemente simplifican la sintaxis ocultándola mediante un editor gráfico, pero que requieren de una cierta comprensión de la lógica de la programación subyacente para poder dominarlas.

Así pues, los programadores no suelen confiar en que los resultados obtenidos por los diseñadores sean buenos, o incluso piensan que pueden incurrir en problemas de eficiencia, al no saber muy bien qué tareas de las incorporadas al árbol de comportamiento son más costosas y suelen acaparar totalmente la implementación de los comportamientos. Normalmente, lo máximo que solía hacer el diseñador es poder configurar ciertos parámetros que el programador deja visibles, para ajustar el comportamiento a sus necesidades, por ejemplo cosas como la velocidad de ataque, el tiemplo de reacción, el tiempo de recarga y un largo etcétera.

Sin embargo, existen formalismos más amigables que los Árboles de Comportamiento usados por Bungie para los diseñadores y a pesar de esto, son precisamente éstos últimos los que se están popularizando. El modelo tradicional usado para la creación de comportamientos, las máquinas de estados finitos, son mucho más amigables con diseñadores no técnicos en realidad que los BTs. De hecho, dicho modelo sigue usándose muy habitualmente en parte por este motivo y en parte por su fácil implementación. Por ejemplo, una de las extensiones más descargadas de la Unity Store es PlayMaker y como vimos en el apartado 2.4.2, usa precisamente máquinas de estados. No en vano, PlayMaker vende la herramienta como una herramienta de programación visual para aquellos usuarios que no saben programar, sin embargo, las herramientas que usan el formalismo de los Árboles de Comportamien-

to son mucho más específicas para crear comportamientos de personajes u objetos y no tanto para programar lógica de juego general.

Aun así, el formalismo de las FSMs sigue siendo útil para crear comportamientos en juegos más sencillos, por ejemplo, en juegos independientes, de bajo presupuesto o en juegos donde los NPCs tienen unos patrones de comportamiento por diseño simples (pensemos, por ejemplo, en juegos con comportamientos de NPCs cíclicos como los juegos de plataforma, donde se puede ver claramente cómo los NPCs pasan por una serie de estados diferenciados como: patrullar y atacar cuando el jugador es visto). Para este tipo de comportamientos, probablemente no merezca la pena utilizar BTs si los programadores y diseñadores se sienten cómodos con las máquinas de estados. Si tenemos un editor de FSMs intuitivo y fácil de usar, es posible que incluso lo diseñadores pudieran directamente construir los comportamientos.

Sin embargo, en juegos más complejos, con comportamientos más elaborados, se tiende a utilizar más los árboles de comportamiento, ya que las FSM en IAs muy complejas se vuelven ingobernables. Por ejemplo buena prueba es el propio Halo donde se utilizó por primera vez los BTs, un juego con una inteligencia artificial tradicionalmente bien valorada tanto por usuarios como por desarrolladores y académicos (Isla, 2008b,a, 2005), de una cierta complejidad, ya que maneja no sólo aliados del jugador, si no diferentes tipos de unidades con diferentes personalidades que además se comportan de forma diferente, dependiendo de si están o no rodeados por otras unidades.

Para que sirva de ejemplo, vamos a explicar algunos de los comportamientos que utilizan algunas unidades de Halo. Por ejemplo, en el videojuego existen unas unidades denominadas *Grunts*, unos pequeños seres del bando *Convenant*, enemigos del jugador, que son miedosos y si se ven solos, huyen. Sin embargo, si esta unidad está acompañada por otro tipo de unidades más poderosas, como los *Elites* (guerreros Covenant de mayor tamaño y habilidad), se vuelven mucho más agresivos y audaces. Otro ejemplo extraído del mismo juego muestra que, algunas unidades de Halo intentan flanquear a los jugadores mientras otras lo distraen, disparando desde zonas seguras o cubriéndose en los escudos de los *Jackals* (unidad de Halo que va equipada con un escudo de energía).

Así pues, el comportamiento en Halo de un NPC puede variar totalmente dependiendo de por quién esté rodeado el NPC y de multitud de otros factores, como el arma que lleven equipado el jugador, el NPC o el número y tipo de aliados que acompañen al jugador o al enemigo. Este tipo de comportamientos más complejos, que requieren coordinación entre varios NPCs, si se creasen usando FSMs implicaría un numero inmanejable de transiciones a implementar, ya que el número de condiciones para cambiar el comportamiento de los NPCs sería muy elevado. El grafo resultante sería difícilmente legible y el formalismo de los FSMs dejaría de tener sentido como herramienta para diseñadores y probablemente ni siquiera para programadores.

En este tipo de juegos como Halo y otros de su misma naturaleza, donde la IA es compleja y dependiente de tantas condiciones, es donde los BTs han adquirido especial relevancia. Tanto es así que la mayoría de motores de juegos profesionales disponen de editores de BTs o extensiones que los soportan. Por ejemplo en Unity podemos ver extensiones como: NodeCanvas[1] o Behavior Designer[2] que implementan este modelo o en Unreal Engine[3] y CryEngine[4] donde ambos incorporan editores de BTs integrados en el propio motor.

Algunas de estas herramientas ya fueron discutidas en el apartado 2.4 y como se comentó en dicha sección, en la mayoría de estos editores, las acciones básicas, es decir los nodos hoja de los árboles, no están correctamente abstraídas y se pierde capacidad de re-utilización, debido a que no disponen de un modelo de abstracción de los parámetros que se les proporciona y suelen compartir estos parámetros directamente de la pizarra del BT. Tampoco los propios BTs o FSM suelen ser parametrizables, lo que limita su re-utilización en diferentes comportamientos como sub-comportamientos. De esta forma, si en el BT donde se coloque la acción, la variable que está siendo usada por la acción no existe, esta no funcionará adecuadamente.

Además de estas limitaciones, existe una más y es que, aunque normalmente los diseñadores tienen claras las tareas que quieren que hagan los NPCs (al menos a priori), en realidad una acción a alto nivel puede ser implementada de muchas formas por diferentes NPCs, porque cada arquetipo de NPC puede tener ciertos matices sobre el comportamiento a alto nivel, descrito por el diseñador y esto es algo que la mayoría de editores de comportamiento no soportan. Es decir, si se usa una tarea como puede ser atacar, nadar, perseguir, etc, dependiendo del NPC, dicha tarea se implementará de diferente forma (unos atacarán de unas formas y otros de otras, unos perseguirán a pie y otros en un vehículo, etc.), pero no deja de ser la misma tarea a nivel conceptual.

A modo de ejemplo, en la Figura 3.1 podemos ver un árbol de comportamiento de un típico comportamiento de patrulla de un NPC con una estructura Wander, Chase, Attack que se podría reutilizar en múltiples comportamientos. Imaginemos que tenemos tres arquetipos de enemigos con estos comportamientos: el soldado normal equipado con una pistola, un ninja, que sólo puede atacar cuerpo a cuerpo con su espada y un kamikaze que lleva explosivos adosados a su cuerpo y que se hace explotar a sí mismo para atacar al enemigo.

Si fijamos nuestra atención por ejemplo en la acción "Attack" ésta, dependiendo de cada arquetipo de NPC donde sea asignado el BT, debería

[1] `http://nodecanvas.paradoxnotion.com/`

[2] `http://www.opsive.com/assets/BehaviorDesigner/`

[3] `https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/QuickStart/index.html`

[4] `http://docs.cryengine.com/display/SDKDOC4/Modular+Behavior+Tree`

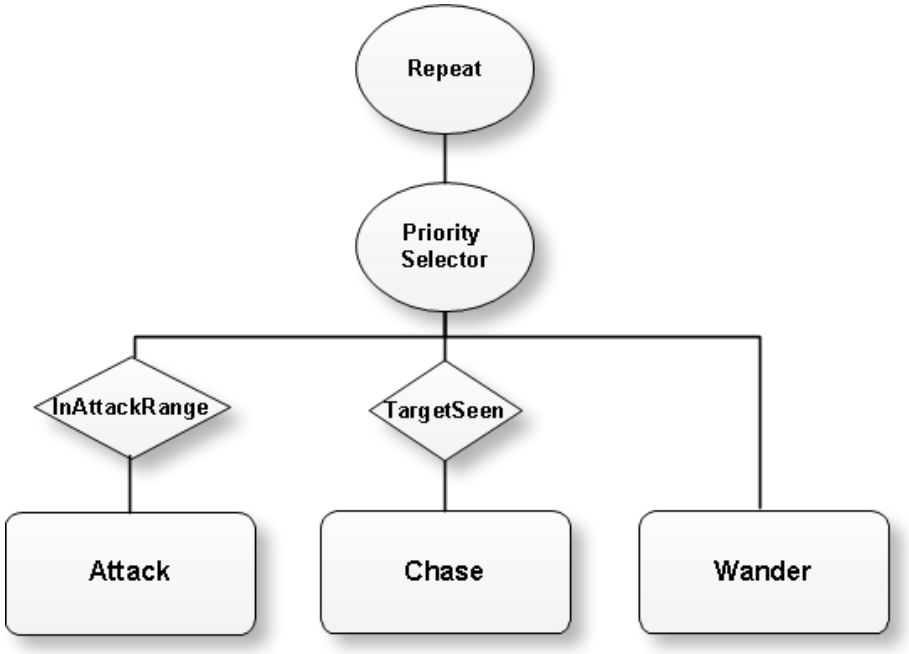


Figura 3.1: BT de ejemplo del típico comportamiento de patrulla de un vigilante

implementar un comportamiento diferente. Por ejemplo, en un soldado normal con un arma de largo alcance, el NPC disparará su arma cuando el objetivo esté a tiro. Sin embargo, en el comportamiento del ninja, éste intentará golpear repetidamente al objetivo con diferentes ataques de su espada y a su vez, intentará cubrirse del objetivo cuando éste le ataque. Finalmente, el Kamikaze intentará buscar un punto donde pueda maximizar el daño que provoque, en vez de acercarse al objetivo, ya que su ataque afecta a más de un NPC. Como podemos ver, la acción Attack no es la misma en los tres NPCs, sin embargo, a alto nivel, la acción sigue siendo la misma: atacar al enemigo o enemigos que acaba de detectar en su rango de ataque.

Para poder reutilizar la acción Attack en este contexto, o ésta es lo suficientemente versátil para poder adaptarse a todas estas circunstancias anteriormente descritas, lo que complica su implementación, o debería ser una acción más específica la que implemente la acción Attack en cada NPC. De esta forma, no podríamos reutilizar el comportamiento de patrulla directamente en los tres NPCs, habría que modificar la acción Attack en cada uno de ellos, añadiendo una acción más específica que fuese en consonancia a la implementación concreta de ese personaje. Una posible solución para poder reutilizar el BT sería que la acción Attack simplemente ordene con un mensaje a otro componente que ataque y sea ese otro componente el que realmente realice la acción de atacar. Ese componente estaría fuera del árbol de com-

portamiento, es decir, cada NPC debería llevar su propia implementación de atacar.

Lo mismo podríamos decir de la acción Wander. Dependiendo del tipo de NPC, la patrulla puede ser diferente. Imaginemos que un soldado patrulla moviéndose de forma aleatoria, otro siguiendo siempre una ruta a pie y otro siguiendo una ruta, pero subido en un vehículo. Las acciones a bajo nivel de esos tres tipos de patrulla son totalmente diferentes. El que patrulla de forma aleatoria, intentará buscar un punto aleatorio al que ir y cuando llegue, buscará otro punto aleatorio. Sin embargo, el que patrulla siguiendo una ruta, tendrá una lista de *waypoints* o puntos de ruta que seguir.

Finalmente, el que conduce un vehículo, también seguirá una ruta fija, pero surgen animaciones y tareas totalmente diferentes al dejar de hacer la patrulla (bajarse del coche para perseguir al intruso) o al volver a patrullar (buscar donde dejó el coche y subirse a él antes de continuar con la ruta). En la Figura 3.2 podemos ver las diferentes formas en las que las se pueden implementar las acciones Wander y Attack de alto nivel en el ejemplo mostrado.

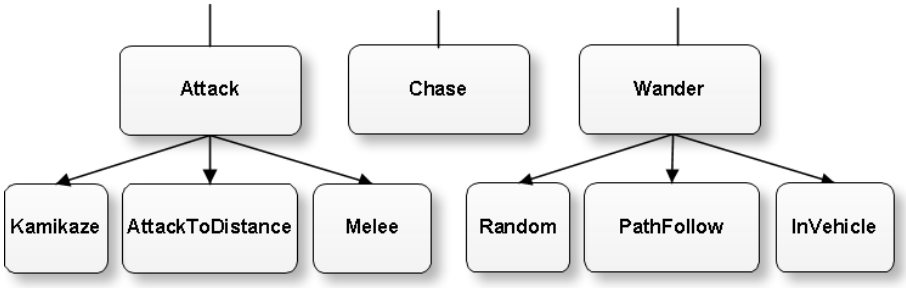


Figura 3.2: Posibles implementaciones de las acciones del comportamiento de patrulla genérico

Debido a estas dos características que los editores de comportamiento no presentan: la falta de parametrización de las acciones y subcomportamientos por un lado y la imposibilidad de sobrescribir comportamientos con acciones más específicas por otro, se ha desarrollado *Behavior Bricks*, un editor de comportamiento diseñado para poder ser totalmente reutilizable y parametrizable, tanto los árboles que se generan como las tareas que se implementan. Este editor lo describiremos con mayor detenimiento en el apartado que viene a continuación. Más adelante en el presente capítulo, demostraremos en base a una experimentación realizada, cómo los diseñadores y los programadores tienen diferencias objetivas a la hora de crear comportamientos. Aunque los primeros, bajo ciertas suposiciones y circunstancias son capaces de crear comportamientos de los que hemos denominado *comportamientos de alto nivel*, en general los programadores son más hábiles para hacerlo y sus resultados mejores. Y, por último, describiremos una metodología que

permite que ciertos diseñadores sean capaces utilizar BTs para crear ciertos comportamientos por sí mismos.

Como veremos más adelante, a pesar de que estas herramientas simplifican la creación de comportamientos, estas simplificaciones siguen siendo insuficientes para muchos diseñadores, ya que incluso con un editor gráfico, los diseñadores no técnicos puede tener problemas a la hora de crear comportamientos. A esto hay que añadir la falta de confianza de los programadores en el resultado que puedan conseguir, lo que en la práctica a muchos de ellos les impide utilizarlos en un entorno de desarrollo real. Para ellos hemos diseñado un sistema que permite a los diseñadores crear comportamiento por demostración sin necesidad de programar ni de crear gran parte del árbol de comportamiento, como veremos más adelante en el Capítulo 4.

3.3. Descripción del entorno de pruebas: Towot

La mayoría de los experimentos llevados a cabo en este trabajo, se han realizado sobre un videojuego desarrollado en el Grupo de Aplicaciones de Inteligencia Artificial (GAIA) de la Facultad de Informática de la Universidad Complutense de Madrid denominado Towot. El videojuego ha ido evolucionando a la vez que las investigaciones realizadas, cambiando su concepto jugable y su diseño durante las mismas. En este apartado, se describe el estado de Towot en el momento de realizar los experimentos asociados a este capítulo. Posteriormente describiremos el estado de Towot en los sucesivos experimentos para que se pueda tener un contexto del entorno de pruebas utilizado que permita entender mejor las mismas.

En el momento de realizar los experimentos de este capítulo, Towot era un juego de defensa de torre mezclado con acción en primera persona. Como buen juego de defensa de torre, en el juego había un núcleo, base o torre que hay que defender y una serie de oleadas de enemigos que provienen de diferentes caminos y que van hacia el núcleo. El jugador tenía dos vistas, una vista estratégica cenital donde podía colocar las torretas que defenderían las oleadas y unos robots autónomos que ayudaban a defender la base. Al instanciar una torreta o robot, se consumía una cantidad de energía de forma que no se podían construir todas las torretas o robots que se quisiese. Además, las torretas y los robots sólo se podían colocar en ciertas regiones del mapa, marcadas cómo tal para que fuesen fácilmente identificables y una vez se habían puesto todas las torretas que el jugador estimase conveniente, el juego podía comenzar.

En esta segunda fase, la cámara cambiaba a primera persona y el jugador podía disparar a los enemigos como un juego de disparos en primera persona tradicional, con la peculiaridad de que además de disparar, podía reparar aquellas torretas que estuvieran defectuosas, así como subirse a los robots que había instanciado, para manejarlos.

Así pues, el gameplay del juego consistía en defender de las diferentes oleadas a la base, usando las torretas para minimizar el número de enemigos que se acerca a la base, y los robots como NPCs secundarios que también ayudan a la defensa. La función del jugador mientras tanto es moverse libremente por el mapa ayudando a que las defensas no sean superadas.

Para realizar los experimentos, se restringió muchísimo el gameplay del juego. La fase de establecer torretas en vista cenital se desactivó, ya que lo que se pretendía era medir únicamente la calidad de los comportamientos de los enemigos creados por los diseñadores. Así que se simplificó todo el juego, dejando sólo en la escena al jugador y los generadores de enemigos. El jugador no podía instanciar torretas ni robots, simplemente luchar contra los enemigos que se acercaban a la base. Los generadores sólo generaban dos tipos de enemigos: el enemigo básico y el tanque, que eran los dos comportamientos que los usuarios debían implementar.

En los experimentos, lo que se pretendía era ver si el comportamiento de los enemigos se correspondía con la especificación dada por nosotros mismo, creado estos comportamientos con Behavior Bricks. Pero realmente lo interesante de la fase de pruebas de los comportamientos de los enemigos era saber si estos hacían lo que se supone que deberían de hacer. Así pues, lo realmente importante del experimento era que los diseñadores testearan si sus comportamientos eran correctos y se ajustaban a la especificación.

3.4. Behavior Bricks

Behavior Bricks es una herramienta de creación de BTs desarrollada sobre el popular motor de videojuegos Unity3D por el Grupo de Aplicaciones de Inteligencia Artificial (GAIA) de la Facultad de Informática de la Universidad Complutense de Madrid, cuya finalidad es el desarrollo de comportamientos inteligentes para videojuegos, usando como modelo base los árboles de comportamiento. El objetivo con el que se construyó y diseñó fue simplificar la tarea de creación de comportamientos a desarrolladores sin conocimientos técnicos y muy específicamente a los diseñadores de videojuegos.

La herramienta consta de dos partes, por un lado, un entorno de ejecución independiente del motor de Unity escrito en C# junto con un componente de Unity que lo utiliza y por otro lado, un editor gráfico de árboles de comportamiento, este sí, vinculado totalmente a Unity como una extensión del mismo, que está disponible en la *Asset Store* de Unity[5] y que podemos ver en la Figura 3.3. Para su construcción se diseño previamente un *middleware* de creación de editores gráficos para Unity denominado *UHotDraw* del que se publicaron los artículos (Sagredo-Olivenza et al., 2015a, 2013). Por clarificar conceptos, la Asset Store de Unity es la tienda oficial de extensiones y re-

[5] `http://bb.padaonegames.com/`

cursos de este popular motor de videojuegos y donde muchos desarrolladores comparten y venden sus herramientas o sus recursos.

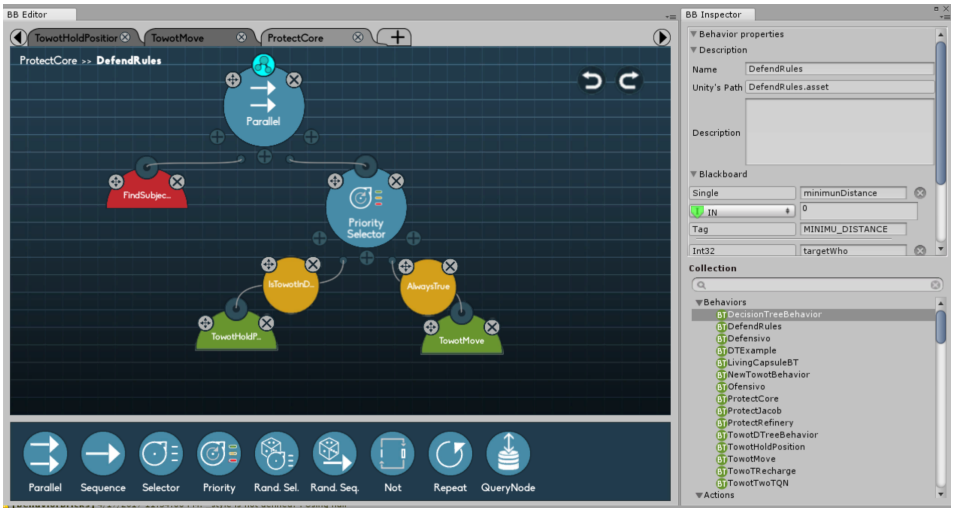


Figura 3.3: Captura de pantalla del editor de árboles de comportamiento de Behavior Bricks

La herramienta está diseñada con la reutilización en mente, ya que muchos comportamientos o subcomportamientos de videojuegos pueden reutilizarse en otros juegos u otros NPCs, simplificando la creación de los mismos y ahorrando trabajo a los usuarios. La herramienta permite la incorporación de tareas primitivas desarrolladas por los programadores que posteriormente pueden ser usadas en los árboles de comportamiento, construidos por los diseñadores a través de un editor visual integrado, lo que permite la colaboración entre ambos, ya que cada uno de los roles desempeña un papel diferenciado en la tarea de creación de comportamientos, minimizando así problemas de comunicación y permitiendo que ambos trabajen de forma conjunta. Los diseñadores pueden crear comportamientos a alto nivel con las piezas o "ladrillos" que proporcionan los programadores, mientras que estos últimos pueden seguir creando nuevas primitivas para construir otros comportamientos. Estos ladrillos pueden ser de varios tipos.

En Behavior Bricks se define como **primitiva**, *ladrillo* o *brick* a un nodo que es capaz de ejecutar, manipular o consultar de algún modo la información de entorno del juego. Son siempre nodos hoja del árbol de comportamiento y, por lo tanto, no tienen descendientes. Estas primitivas son parametrizables, es decir, pueden recibir cualquier número de parámetros de entrada y dependiendo de qué tipo de primitiva sea, estos pueden tener o no tener cualquier número de parámetros de salida. En Behavior Bricks hay dos tipos de primitivas claramente diferenciadas que explicaremos a continuación, **Tareas** y **Condiciones**.

3.4.1. Tareas

Al igual que las acciones en los BTs tradicionales, las tareas son primitivas que modifican o pueden modificar el entorno del juego. Pueden hacer que el personaje se mueva dentro de mundo virtual, pueden eliminar un enemigo, cambiar una animación, etc. Es decir, su ejecución implica cambios visibles para otras entidades del juego que puede ocasionar consecuencias en el comportamiento de esas otras entidades. Estas tareas pueden devolver, a parte de su estado de terminación, cualquier número de parámetros de salida. En Behavior Bricks, existen dos tipos de tareas desde el punto de vista de cuál fue el método de implementación de la tarea (cómo se creó): las acciones básicas y los subcomportamientos.

- Acciones básicas: Son las acciones típicas de los árboles de comportamiento que se describen en la literatura, a las que hemos añadido parametrización. Están desarrolladas directamente en el código nativo del motor (en la actual implementación Unity y C#) y tienen acceso al API del motor de juego y al resto de componentes y entidades. Por tanto, son tareas que, para ser construidas, se requieren fuertes conocimientos técnicos y de programación.

- Subcomportamientos: estas tareas son implementadas mediante un árbol de comportamiento. Es decir, cualquier comportamiento creado con el editor, puede ser instanciado como subcomportamiento en otro BT y es tratado en éste como si fuese una acción básica. Al igual que éstas últimas, los subcomportamientos poseen una lista de parámetros de entrada y de salida, que sirven para comunicarse con el resto de nodos del comportamiento que las usa. La única diferencia es que unas (las acciones) están generados directamente por el código nativo o de scripting del motor y las otras son árboles de comportamiento complejos creados con el editor. El nivel de abstracción de estos subcomportamientos teóricamente debería ser mayor que el de las acciones básicas y pueden ser implementados por desarrolladores con menores conocimientos técnicos gracias a que disponemos del editor visual.

3.4.2. Condiciones

Las condiciones son otro tipo de primitiva dentro de Behavior Bricks. Se diferencian de las tareas en que simplemente chequean el estado del juego sin cambiarlo y evalúan si se cumplen una serie de premisas para devolver éxito o fallo. No devuelven ningún tipo de parámetro de salida, ya que no pueden modificar el entorno, pero pueden tener cualquier número de parámetros de entrada.

Estas primitivas normalmente se usan en los nodos hoja pero también pueden usarse en los nodos guarda, que mencionamos en el apartado 2.3.2.

Estos nodos guarda sirven para proteger la ejecución de una tarea o bien para evaluar el cumplimiento de una de las ramas de un *priority selector*. Dicho priority selector antes de ejecutar el subnodo hijo, evalúa la guarda y decide si ejecuta el nodo o no en función de su resultado.

Normalmente, el último nodo es la acción del hijo por defecto, por lo tanto, no lleva ninguna guarda asociada y se presupone que si ninguna de las anteriores ha sido activada, la última acción es la que se debe ejecutar. Por cuestiones de implementación en Behavior Bricks el nodo por defecto también lleva una guarda, pero es una guarda que evalúa la condición **AlwaysTrue** que siempre devuelve éxito.

3.4.3. La capacidad de abstracción de Behavior Bricks

La posibilidad de parametrizar las primitivas en Behavior Bricks, dota a nuestra implementación de los árboles de comportamiento de una potente capacidad de generalización y abstracción, lo que permite que un estudio de videojuegos pueda, con el tiempo, construirse una *biblioteca de primitivas* reutilizables entre diferentes proyectos y diferentes NPCs. Para permitir la comunicación entre las diferentes primitivas, los parámetros de entrada y salida de las mismas leen y escriben de un espacio de memoria compartido, dentro del comportamiento que las ejecuta, al que denominamos la **pizarra** del comportamiento. Estos parámetros se deben asociar a las variables de la pizarra en el diseño del comportamiento, permitiendo interrelacionar unas primitivas con otras. A la información almacenada en estas pizarras la denominamos **contexto del comportamiento**.

La extensión que proponemos en este trabajo surge de forma natural: si los BTs ya construidos pueden colocarse en otros BTs más generales como nodos hoja, podemos ver esos BTs como definiciones de tareas primitivas implementadas con árboles en lugar de directamente en código, lo que permite reutilizarlas.

Cuando diseñamos Behavior Bricks, una de las cosas que perseguíamos era poder reutilizar comportamientos genéricos en diferentes NPCs, buscando la flexibilidad que otorga el paso de mensajes en las arquitecturas por componentes. La utilidad de los mensajes en una arquitectura por componentes es debido a que no tienen un componente dueño del mismo. Sólo una entidad desde la que parten y una o varias entidades a las que se envían. Las entidades que reciben el mensaje se lo comunican a sus componentes para que lo procesen. Estos mensajes pueden ser descartados por los componentes si no están interesados en ellos y sólo son procesados aquellos mensajes que son interesantes para el componente.

Con esto en mente, en Llansó García (2014) se incide en la importancia de crear una ontología de mensajes debido a que facilita la reutilización de los componentes. Cuando se envía un mensaje de tipo Atacar, a una entidad

es posible que dicha entidad no atienda al mensaje atacar en sí mismo, pero si a un mensaje que hereda del mismo y que semánticamente sería similar, aunque tenga un modo de proceder diferente, por ejemplo el mensaje *atacar cuerpo a cuerpo* podría atender también el mensaje Atacar.

Otro ejemplo es el mensaje *MoveToPosition* que ordena a los componentes que una entidad se mueva hacia una posición. Un componente puede escuchar dicho mensaje, pero también puede escuchar un mensaje más concreto denominado *WalkToPosition* o *RunToPosition*. Ambos mensajes son válidos porque heredan de MoveToPosition. Si un NPC tiene el componente **Run** que acepta el mensaje *RunToPosition* cuando le llegue el mensaje *MoveToPosition* desplazará la entidad hasta la posición indicada, pero lo hará corriendo.

Sin embargo, otra entidad que tenga el componente **Walk** y acepte el mensaje *WalkToPosition* lo hará andando. Ambos atienden por tanto su mensaje y todos los mensajes padre, sucesivamente hasta la raíz de la jerarquía. Imaginemos que la entidad tiene ambos componentes, entonces ante el mensaje *MoveToPosition* cada uno ejecutará sus mensajes, pero probablemente sería un error semántico. En ese caso, debería mandarse un mensaje más especifico cuando se le ordene moverse o algún componente previamente ha debido desactivar uno de los dos componentes.

Esta idea se puede trasladar a la ejecución de comportamientos gracias al diseño arquitectónico de Behavior Bricks. Podemos asumir que una tarea es una especie de mensaje que tiene diferentes implementaciones. Cual sea la implementación real instanciada en el comportamiento en principio no es importante, salvo que ésta entre en conflicto con la definición del comportamiento.

Por ejemplo, volviendo al ejemplo del mensaje *MoveToPosition*, imaginemos ahora que *MoveTo* es una tarea en Behavior Bricks que tiene tres implementaciones. La propia tarea *MoveTo* que mueve el comportamiento de una forma genérica (imaginemos una velocidad 4 m/s) sin embargo hay otras dos tareas *WalkTo* and *RunTo* que implementan la tarea *MoveTo*. *WalkTo* establece la animación *Walk* en el personaje y mueve el objeto a una velocidad de 2 m/s y la tarea *RunTo* establece la animación *Run* y mueve el personaje a 8 m/s. La tarea *WalkTo* se usa en un zombi que se mueve despacio y la tarea *RunTo* la utiliza un enemigo mucho más rápido. El comportamiento a alto nivel es el mismo, cuando ve al jugador el personaje se mueve hasta su posición para atacarlo. Por tanto, en la definición del comportamiento debemos poner la tarea de más alto nivel, *MoveTo* pero la tarea específica en cada NPC debería ser diferente para cada tipo de enemigo.

Behavior Bricks soporta este mecanismo que permite crear comportamientos con tareas genéricas que tienen diferentes implementaciones según el NPC al que se le asigne, lo cual hace a los comportamientos mucho más reutilizables en diferentes NPCs o incluso en diferentes juegos. Esto se con-

sigue gracias a que las tareas se declaran con un atributo de C# y cualquier clase puede implementar ese nombre de tarea. Simplemente, el usuario deberá añadir como componente de la entidad que utiliza el comportamiento, la implementación concreta que dicho NPC necesita. En el siguiente ejemplo podemos ver dos implementaciones de *MoveTo*: WalkTo y RunTo por código para ejemplificar el mecanismo.

```
[Action("MoveTo")]
public class WalkTo : GOAction {
  [InParam("Position", typeof(Vector3))]
  ...
} // class

[Action("MoveTo")]
public class RunTo : GOAction {
  [InParam("Position", typeof(Vector3))]
  ...
} // class
```

Como veremos más adelante con mayor detalle, esta característica es esencial a la hora de implementar los *Trained Query Node* que explicaremos en el capítulo 4, pero que a modo resumen, aprende a seleccionan la mejor tarea de las posibles implementaciones de una tarea genérica, en base a la información aportada por el entrenador del nodo. En el nodo se debe establecer una tarea que tenga diferentes implementaciones, por ejemplo MoveTo en el ejemplo anterior, y mediante ejemplos, indicarle al nodo cuando debe ejecutar RunTo y cuando debe ejecutar WalkTo.

Para terminar, mencionamos otra característica a destacar en *Behavior Bricks* y es que las pizarras donde se comparten los parámetros de entrada y salida son privadas al comportamiento. Estas pizarras guardan también los parámetros de entrada y de salida del comportamiento de forma que las primitivas pueden leer de estos parámetros de entrada y modificar los parámetros de salida del comportamiento libremente al igual que se pueden modificar los parámetros internos ocultos. Cierta información que se almacena en la pizarra proviene de la percepción del NPC. Esta percepción se implementa como una acción que se introduce en el BT de más alto nivel o bien, desde un componente externo, se modifican los parámetros de entrada del BT de más alto nivel. Además, los parámetros de entrada del BT de más alto nivel son configurables directamente desde el editor de Unity, con lo cual, los diseñadores pueden cambiar el comportamiento del mismo, modificando sus parámetros. Los parámetros de salida del comportamiento de más alto nivel en principio no se asignan a ninguna pizarra por lo que se pierden. Lo normal es que el comportamiento de más alto nivel no tenga parámetros de salida ya que son digamos "procedimientos" que no necesitan informar de nada a otros comportamientos.

Con esta abstracción, *Behavior Bricks* permite reutilizar los comportamientos haciéndolos más independientes de los datos que se le proporcionan,

de forma similar a como se hace en los lenguajes de programación cuando se llama a una función o un método. Por lo tanto, para definir una tarea debemos determinar su tipo, darle un nombre que debe ser único en la colección de tareas y opcionalmente, la lista de parámetros que necesita del exterior (parámetros de entrada) y la información que proporciona al exterior en forma de parámetros de salida.

3.5. Metodología de uso de árboles de comportamiento

Como mencionamos en el Capítulo 1, los diseñadores deben poder crear comportamientos de forma independiente de los programadores, para reducir la comunicación entre ambos y poder trabajar en paralelo. Dado que éstos no tienen por qué tener conocimientos técnicos avanzados, los diseñadores necesitan herramientas para simplificar esta tarea, que de otro modo sólo estaría accesible para aquellos diseñadores que tuvieran experiencia previa en programación. Estas herramientas deben intentar utilizar un lenguaje cercano al que usan los diseñadores, para que les sea más fácil su uso. Algunas de las herramientas utilizadas en la industria como mencionamos en el apartado 2.4 utilizan formalismos más próximos a los programadores que a los diseñadores. Nuestra propia experimentación, que detallaremos más adelante en este capítulo, publicada en (Sagredo-Olivenza et al., 2015b) demuestra que, aunque en determinadas circunstancias que describiremos a continuación los diseñadores son capaces de crear comportamientos, los programadores son más hábiles y generan mejores resultados en general que los diseñadores.

Así pues, tenemos que fijarnos en cómo normalmente los diseñadores expresan los comportamientos de sus NPCs, para crear herramientas que puedan entender mejor. Los diseñadores suelen utilizar normalmente descripciones basadas en máquinas de estados (Champandard, 2007a), reglas o directamente en lenguaje natural (Rasmussen, 2016). Por lo tanto, formalismos como máquinas de estados o sistemas de reglas son percibidos por los diseñadores como modelos más amigables. Sin embargo, los BTs son más complicados para ellos, debido a que manejan conceptos de más bajo nivel y más cercanos al scripting. Por ejemplo, la diferencia entre los nodos secuencia y los nodos paralelos no es clara, la propia Epic Games® en su implementación de árboles de comportamiento para su motor Unreal ha prescindido de los nodos parallel sustituyendolos por versiones más sencillas (Véase Capítulo 2.4.5) de este tipo de nodo, o acceder a tareas de muy bajo nivel que requieren una secuencia concreta de pasos, son conceptos muy alejados de lo que un diseñador tiene en mente cuando crea un comportamiento.

La idea del comportamiento en la mente del diseñador es mucho más difusa y a alto nivel y se basa más en las reglas que la gobiernan que en los mecanismos propios de la programación para conseguirlos. Sin embargo, estos

conceptos de más bajo nivel pueden ser ocultados por los programadores, construyendo tareas primitivas más complejas, que realicen tareas de más alto nivel, bien programando una acción que las implemente o bien creando sub-comportamientos con el editor, que luego utilicen en el comportamiento final.

Por su definición intrínseca (cualquier sub-árbol es un árbol[6]), los BTs pueden ejecutar sub-comportamientos fácilmente. Estos sub-comportamientos pueden ser utilizados por los diseñadores en otros comportamientos como una simple tarea primitiva en *Behavior Bricks*, proporcionándoles un nombre y una lista de parámetros de entrada y salida que describen la *interfaz* de dicho comportamiento. Este mecanismo puede tener diferentes capas de abstracción en un comportamiento complejo, siendo estas capas, como cualquier otro lenguaje de programación, una abstracción de la complejidad del comportamiento subyacente, lo que va a favorecer al diseñador a la hora de construir con estas piezas, comportamientos más complejos y de más alto nivel. En un enfoque descendente, podemos definir dos niveles diferentes de detalles a grandes rasgos: los comportamientos *a alto nivel* y los comportamientos *a bajo nivel*.

Los comportamientos de bajo nivel pueden definirse como aquellos que pueden reutilizarse en otros comportamientos y que están más cercanos a las acciones que el NPC debe realizar que a una IA estratégica. Estos comportamientos gestionan conceptos más cercanos a los scripts tradicionales como: selección de objetivos, seguimiento de una ruta, búsqueda de una cobertura, etc. Los comportamientos de alto nivel pueden definirse como aquellos comportamientos que describen el funcionamiento general del NPC a grandes rasgos. Estos comportamientos pueden explicarse usando unas pocas palabras o una simple máquina de estado. Esta forma de definir comportamientos entra en el terreno de lo que el diseñador conoce y suele usar y en el lenguaje en el que ellos se expresan. En la Figura 3.4 podemos ver un ejemplo de esta división.

Behavior Bricks permite crear BTs de bajo nivel que se usen en BTs de más alto nivel, gracias a su capacidad de abstraer comportamientos y poder ejecutarlos como acciones. La parte indicada en la Figura 3.4 como comportamiento de bajo nivel, puede ser un subcomportamiento que podemos denominar *Wander* y que simplemente es usado por el diseñador en el BT de alto nivel como una acción más. De esta forma, podemos delimitar de forma clara las tareas en ambos roles (programador y diseñador) usando *Behavior Bricks*.

Los programadores en esta metodología crearán los comportamientos de bajo nivel y las acciones primitivas como hemos descrito anteriormente, es decir las tareas básicas del juego. Por el contrario, los diseñadores no técnicos

[6]Weisstein, Eric W. "Subtree" From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/Subtree.html`

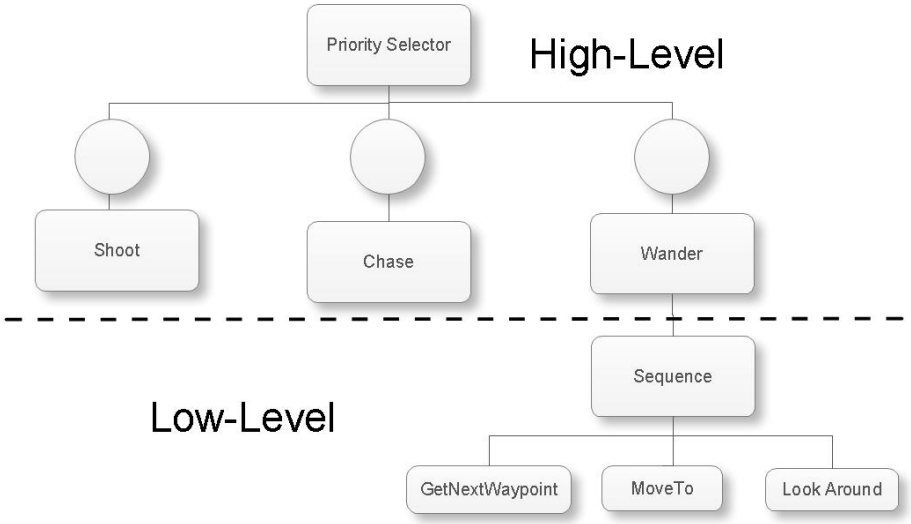


Figura 3.4: Ejemplo que describe la diferencia entre los comportamientos a bajo y a alto nivel

crearán los comportamientos de alto nivel, utilizando dichas tareas como primitivas de sus comportamientos.

Behavior Bricks sirve, por tanto, como nexo de unión entre ambos roles a la hora de crear un comportamiento. Los diseñadores al implementar los comportamientos de más alto nivel también testean los de más bajo nivel, anticipando posibles errores en los mismos, antes de terminar el comportamiento. Además, ambos roles tienen bien diferenciadas sus tareas y la cooperación se establece en la creación del propio comportamiento y no sobre un documento de diseño.

Mientras ambos colaboran para crear el comportamiento, el diseñador puede experimentar por sí mismo si los comportamientos ideados en la fase creativa del diseño realmente están funcionando o si no están claros y deben ser modificados, esto permite al diseñador ser más ágil a la hora de ajustar el comportamiento para que el juego sea divertido. A continuación vamos a delimitar más claramente las tareas que realizarán tanto los programadores como los diseñadores y cómo estos van cooperando en un proceso iterativo de desarrollo.

3.5.1. Behavior Bricks desde el punto de vista de los programadores

En este apartado, describiremos cómo los programadores pueden interactuar con *Behavior Bricks*. La responsabilidad de los programadores en *Behavior Bricks* es doble. Por un lado, deben crear las primitivas de bajo nivel utilizando el lenguaje de scripting del motor subyacente (en nuestro

caso Unity en este momento); Y, por otro lado, deben implementar con el editor de *Behavior Bricks* aquellos comportamientos de más bajo nivel.

Para definir una tarea primitiva, basta con establecer su nombre (que debe ser único en la jerarquía) y opcionalmente una colección de parámetros de entrada y salida. Los parámetros de entrada leen sus valores desde una pizarra o se establecen como constantes por el usuario manualmente. Esta pizarra se almacena en el ejecutor de comportamiento y es compartida por todas las tareas que se ejecutan en este comportamiento. Y finalmente, los parámetros de salida escriben sus valores en la misma pizarra. De esta forma, las acciones procesan la información y van modificando la misma en función del flujo de ejecución.

Para tener una mejor idea de cómo implementar una tarea, vamos a mostrar un pequeño ejemplo: imaginemos que el programador quiere crear una acción para esperar un número variable de iteraciones de actualización. Para implementar este comportamiento en *Behavior Bricks*, éste necesita saber qué número de ticks debe esperar (vamos a denominar al parámetro por ejemplo *Count*) y el nombre que tendrá la tarea (En este caso *WaitNUpdates*). El código que implementa la acción sería el siguiente:

```
[Action("WaitNUpdates")]
public class WaitNUpdates : BasePrimitiveAction
{
 [InParam("Count", typeof(int))]
 int remainder;

 public override void OnStart()
 {
 }
 public override TaskStatus OnUpdate()
 {
  if (remainder <= 0)
   return TaskStatus.SUCCESS;
  remainder--;
  return TaskStatus.RUNNING;
 } // OnUpdate
} // class
```

Como se puede ver en el código de ejemplo, el nombre de la primitiva (acción en este caso) se define utilizando un atributo C#:

```
[Action("WaitNUpdates")]
```

Y los parámetros de entrada se definen también utilizando un atributo de C# de la siguiente forma:

```
[InParam("Count", typeof(int))]
```

Se puede apreciar que pueden existir diferentes clases que implementen la tarea *WaitNUpdates*. Cada una puede implementarla de una forma diferente. En el editor de comportamientos podremos colocar la acción WaitNUpdates,

pero luego al instanciar el BT en un NPC concreto, en tiempo de ejecución, esa tarea puede ser implementada por la clase WaitNUpdates, o por un BT, o por otra acción con otro nombre de clase. Los métodos disponibles para implementar, como **OnUpdate** u **OnStart**, provienen de extender de la clase *BasePrimitiveAction* o bien de *GOAction*, si prefiere que la tarea sea un componente de Unity que luego pueda ser asignado a la entidad que lo utiliza.

La secuencia de llamadas a los métodos de una acción se puede resumir de la siguiente forma. Cuando la acción comienza a ejecutarse, el intérprete llama al método **OnStart** y en cada iteración llama al método **OnUpdate**. Si la acción es interrumpida por otra (por ejemplo, en un selector de prioridad, mientras se ejecuta una rama, otra más prioritaria interrumpe a ésta y toma la ejecución), el sistema llama al método **OnAbort**, que no aparece en el ejemplo, para informar a la acción de que ha sido interrumpida, por si tiene que liberar algún recursos. El método **OnUpdate** devuelve el estado de ejecución de la acción que recordemos puede ser *Running*, *Success* o *Failure* dependiendo de si aún no se ha completado la tarea, no se ha podido completar o si se ha completado con éxito. Con esa información de terminación, el nodo intermedio padre de la tarea decidirá qué hacer, volver a repetir la acción, ejecutar la siguiente, etc. Dependerá del tipo de nodo intermedio.

Crear condiciones se realiza de forma similar. El siguiente ejemplo se muestra como se crea una condición. La unica diferencia a nivel de programación entre acción y condición es en los métodos que tenemos disponibles para implementar. En la condición se sustituye el método **OnUpdate** por el método **Check** y este sólo puede devolver *true* o *false* que se transforman en success o failure. Pero en ningún caso *Running* ya que no son acciones latentes (Cordone, 2011).

```
[Condition("Basic/CheckMouseButton")]
public class CheckMouseButton : ConditionBase
{
 [InParam("button")]
 public MouseButton button = MouseButton.left;

 public override bool Check()
 {
  return Input.GetMouseButton((int)button);
 }
}
```

El programador también puede usar el editor de *Behavior Bricks* para crear los comportamientos que hemos denominado en la metodología como *comportamientos de bajo nivel*. Estos comportamientos son, en sí mismos, árboles creados con el editor y que están almacenados en el sistema. A modo de ejemplo, en la Figura 3.5 podemos ver un comportamiento de bajo nivel utilizado en el Towot en su segunda fase de desarrollo, del cual ya hemos introducido en este capítulo, pero que detallaremos más pormenorizadamente

en el apartado 4.3. Por ahora, basta con saber que en un momento dado, el robot Towot, puede moverse hacia una posición y dicha acción la lleva a cabo el BT **TowotMove**, que es ejecutado como subcomportamiento dentro del BT que gobierna al Towot. Esta tarea, además de moverse hasta ese punto, debe activar y desactivar un sonido y una animación mientras se produce el desplazamiento. La tarea podía haber sido desarrollada como una acción programada por código como las acciones anteriores, pero sin embargo fue desarrollada como un BT por los programadores del juego.

Para poder desarrollar una tarea como un BT, el sistema debe contar con todas las "piezas" necesarias para poder llevar a cabo. En este caso, las acciones primitivas *PlayTowotSoundAnimation* que establece un sonido y una animación, *MoveToGameObject* que mueve una entidad del juego a una posición y *StopTowotSoundAnimation* que detiene la ejecución de un sonido y una animación. Estas tres acciones sí están programadas por código.

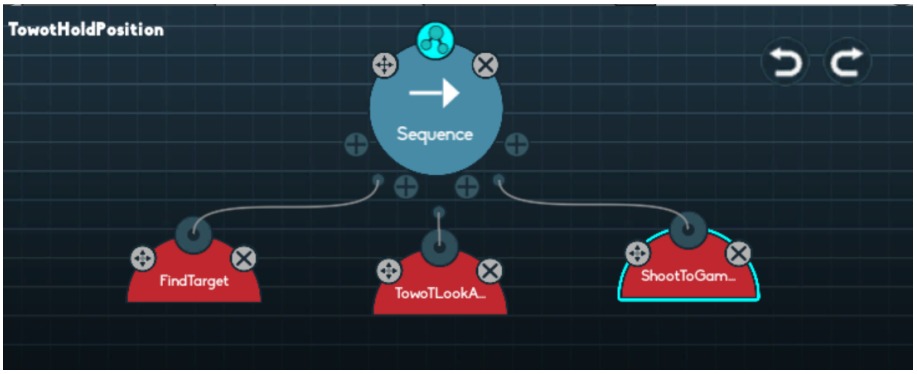


Figura 3.5: Ejemplo de una tarea a bajo nivel usanda en Towot

Esta tarea tiene un parámetro de entrada que es la posición a la que debe ir el NPC y que necesita la acción primitiva MoveTo. Las acciones *PlayTowotSoundAnimation* y *StopTowotSoundAnimation* necesitan dos parámetros que son la animación que se desea ejecutar y el sonido que se desea reproducir. En este caso, estos parámetros son constantes en el árbol. Nótese que ésta no es la mejor solución para hacer el comportamiento totalmente genérico y reutilizable. Lo adecuado hubiera sido que estas acciones leyeran de la entrada del BT la información para poder cambiar la animación y el sonido en cada NPC, pero esta fue la decisión que tomó el programador del juego en su momento y de ahí que especificase que esta tarea es TowotMove, es decir, describe cómo se mueve el Towot en particular y no puede usarse para otro NPC.

Si usamos directamente la tarea en el ejecutor de comportamiento del personaje, desde el inspector de Unity habría que proporcionarle el destino al que debe ir el NPC o no se movería a ningún sitio. En este caso, el destino lo lee de la pizarra. Debe haber otra acción que coloque en esa variable de

la pizarra, la posición del objetivo. Esa tarea, por ejemplo la podría hacer el sistema de percepción del NPC (podemos imaginar cómo el sistema de percepción ha detectado un posible enemigo y ha inyectado la posición como variable en la pizarra del BT). En la Figura 3.6, podemos ver una captura de la pizarra de este subcomportamiento con el parámetro de entrada descrito, que ha sido asignado al subcomportamiento.

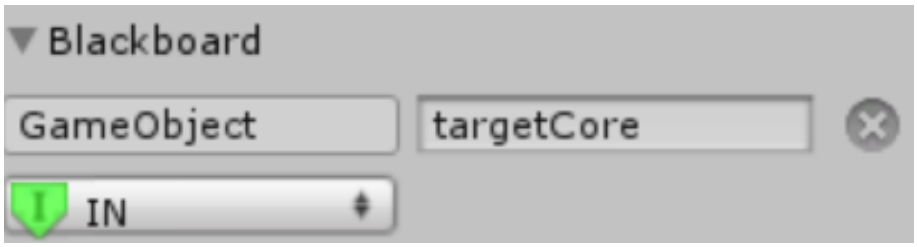


Figura 3.6: Captura de la pizarra con un parámetro de entrada del comportamiento TowotMove

Así es cómo en nuestra metodología, un programador usa *Behavior Bricks*. Antes de comenzar a crear los comportamientos, el diseñador y el programador deben acordar qué primitivas deben ser implementadas por el programador. Antes de realizar una nueva tarea básica, el programador debe comprobar si no dispone ya de una tarea o conjunto de tareas en la biblioteca del sistema con las que construir dicha tarea. La idea es que el diseñador no tenga que editar un árbol general de más de dos niveles de profundidad, para simplificar el proceso y que el diseñador pueda crear comportamientos que puedan ser fácilmente entendibles a simple vista, mientras puede experimentar en las capas de más alto nivel con los comportamientos directamente sobre *Behavior Bricks*, editando él mismo el comportamiento de forma iterativa hasta construir uno que sea satisfactorio.

3.5.2. Behavior Bricks desde el punto de vista de los diseñadores

En principio, el diseñador sólo debe hacerse responsable de las tareas de más alto nivel para aliviarle de posibles complicaciones inherentes de trabajar en los detalles de las acciones más específicas. Sin embargo, los diseñadores con más habilidades técnicas, pueden también ocuparse de los comportamientos de más bajo nivel. Esta separación de roles y tareas propuesta por nuestra metodología, por tanto, no es rígida y dependerá en cualquier caso de la configuración del equipo de desarrollo y sus conocimientos previos.

Por otro lado, a los diseñadores no técnicos es bueno restringirles el uso de ciertos nodos internos complejos. Tener gran variedad de nodos internos proporciona una gran expresividad a los programadores y puede simplificar su trabajo, pero pueden abrumar a los diseñadores, porque el funcionamiento

de algunos nodos no es fácil de entender para ellos y porque pueden no saber cuál elegir en cada momento. Con el objetivo de reducir la complejidad de los BTs para los no programadores, nuestra metodología propone seleccionar un subconjunto de los nodos internos que puedan utilizar, que posteriormente, cuando el diseñador adquiera una mayor experiencia, podría ser ampliado. Este subconjunto propuesto está formado por los nodos: *secuencia*, *selector*, *selector de prioridad* y el *decorador de repetición*. Estos son suficientemente expresivos para construir la mayoría de comportamientos de alto nivel, que son generalmente muy similares como decimos, a las máquinas de estados.

Por ejemplo, la estructura de estado a ejecutar, condición que la dispara, del selector con prioridad es muy similar a la forma de trabajar de las máquinas de estados y las reglas, en la que una acción es iniciada si se cumple una condición. Hemos excluido a los nodos paralelos, los *NOT* y otros nodos y decoradores más complejos que aparecen en la literatura como los selectores aleatorios o las guardas, restringiendo el uso de estas últimas solamente al selector de prioridad para focalizar la atención del diseñador en un menor número de conceptos.

3.5.3. Behavior Bricks en un proceso de desarrollo iterativo

Por último, vamos a describir cómo utilizar *Behavior Bricks* en una metodología ágil de desarrollo de software como scrum (Schwaber y Beedle, 2001).

Scrum y el proceso iterativo de desarrollo son la metodología más utilizada en el desarrollo de videojuegos (Miller, 2008; Keith, 2010b) y más expresamente en el diseño de la IA para videojuegos ya que los comportamientos de los personajes raramente son concebido e implementados a la primera y son fruto de sucesivas iteraciones y ajustes.

Por ejemplo en la GDC 2010, Hudson's (2010) describe cuál fue el proceso por el que se creó la inteligencia artificial de los personajes del videojuego Bioshock 2® en la que describe como, por ejemplo, con uno de los personajes (*Brute*) una vez terminada la primera fase de diseño, modelado, animación e implementación, se dieron cuenta que no encajaba con el diseño de los escenarios que habían planteado y tuvieron que modificarlos, al igual que ajustar los comportamientos a dichos escenarios, o como en las *Big Sister* (otro enemigo del juego) usando lo aprendido con el *Brute*, el prototipo inicial fue rápido, pero tenían un diseño muy inconsistente de ella, debido a que su comportamiento debía ser en cierta manera imprevisible y no estaba muy claro desde el propio diseño, lo que les obligó a refinarlo en varias iteraciones.

La idea básica detrás de la metodología de desarrollo iterativo es desarrollar un prototipo jugable lo más rápidamente posible y a través de pequeños cambios incrementales, ir haciendo crecer el prototipo hasta obtener el juego final. En el caso de aplicarlo al diseño de comportamientos, hasta conseguir

el comportamiento que se espera del NPC.

Para facilitar la colaboración entre programadores y diseñadores en esta metodología de desarrollo iterativo, *Behavior Bricks* sería el contrato entre ambas partes. El diseñador es el encargado de crear el comportamiento a alto nivel, bien en papel o bien directamente en *Behavior Bricks*, dejando las tareas de bajo nivel por implementar para que el programador las vaya construyendo y reutilizando algunas de las ya existentes en la biblioteca de tareas. Cuando va obteniendo las tareas a bajo nivel del programador, el diseñador puede ir probando el comportamiento, modificando si fuese preciso el BT de forma que, si las cosas no funcionan bien, el diseñador pronto se dará cuenta de ello y podrá corregirlo más ágilmente o dando indicaciones al programador para que modifique la tarea entregada.

De esta forma, el BT creado en *Behavior Bricks* puede ser la propia documentación de los primeros prototipos. En consecuencia, los programadores implementarán estas tareas primitivas en C# o con comportamientos de bajo nivel. Este proceso se repetirá hasta que se complete el primer prototipo del comportamiento completamente funcional que, entonces, será validado por el diseñador de forma exhaustiva. En este punto, el diseñador tendrá que realizar los ajustes finales sobre el comportamiento si fuese necesario, hasta ajustarlo a lo que el diseñador quería conseguir.

Cuando el primer prototipo esté totalmente terminado y validado por el diseñador, debería ser testeado por el equipo de control de calidad (Quality Assurance o QA) que proporcionará retroalimentación al diseñador y a los programadores de errores tanto de diseño como de programación. Esta retroalimentación puede ser positiva y cerrarse el ciclo, o lo más normal es que provoque que se vuelvan a modificar o reajustar el comportamiento, ya que normalmente surgirán casos no contemplados, errores, ajustes en la dificultad, etc.

3.6. Experimentación realizada

En el proceso de experimentación que detallamos a continuación, queríamos validar nuestra hipótesis inicial donde suponíamos que *Behavior Bricks* utilizado en base a nuestra metodología, podía ser usado por los diseñadores con un cierto grado de fiabilidad, aunque estos no tengan conocimientos técnicos avanzados. Sin embargo, con aquellos diseñadores técnicamente menos calificados, asumimos que seguirán teniendo más problemas para comprender los BTs que los que tengan mayor cualificación técnica. Lo que queríamos comprobar es si realmente las limitaciones de los diseñadores no técnicos eran tan grandes como para impedir utilizar la herramienta o por el contrario podían superarse con un proceso de formación suficente.

En los roles descritos en la metodología, ya hemos considerado que los diseñadores con conocimientos de programación deberían ser capaces de usar

todas las características del editor visual, incluso implementando subcomportamientos complejos de bajo nivel. Sin embargo, los comportamientos de bajo nivel son más difíciles de lograr por los diseñadores no técnicos, y por lo tanto, sólo deben crear comportamientos de alto nivel, al menos inicialmente, hasta que su experiencia les permita tener mayor control sobre el formalismo de los BTs y la herramienta en sí misma.

Nadie como los diseñadores sabe cómo debe comportarse un NPC, en consecuencia, si crean partes del comportamiento, reducirán los errores de comunicación entre los dos roles principales, paralelizando el trabajo y, por consiguiente, aumentando la productividad frente a la metodología tradicional. Además, el modelo de BT expuesto en *Behavior Bricks* ayuda a reutilizar los comportamientos y tareas primitivas, debido a la abstracción que permite especificar sus parámetros de entrada y salida, lo que facilita tener una biblioteca de comportamientos reutilizables, que el diseñador podrá utilizar en diferentes NPCs.

El experimento que describimos a continuación demuestra que es posible que los diseñadores no técnicos implementen ciertos comportamientos de alto nivel con resultados similares a los programadores, aunque ciertamente inferiores, si han sido entrenados lo suficiente y tienen una herramienta visual como *Behavior Bricks*. Aunque, probablemente, necesitan trabajar más duro para comprender los BTs y *Behavior Bricks* que sus compañeros programadores, sí pueden usar la herramienta para crear ciertos comportamientos. Hemos llevado a cabo este experimento con los estudiantes del Máster de Desarrollo de videojuegos de la Universidad Complutense de Madrid, que incluye dos itinerarios diferentes: uno para programadores y otro para diseñadores. Estos estudiantes tienen diversos antecedentes formativos, a modo de ejemplo, entre los diseñadores se suelen encontrar graduados en: Ciencias de la Computación, Matemáticas, Periodismo o Artes, y por lo tanto, es una población lo suficientemente diversa como para validar nuestra hipótesis.

Antes de hacer el experimento, los sujetos recibieron formación tanto de árboles de comportamiento como del uso básico de *Behavior Bricks*. La formación consistió en una clase teórico-práctica sobre árboles de comportamiento y el uso básico de la herramienta, usando su tutorial oficial. Se realizó esta formación para agilizar el experimento y que todos los participantes tuvieran la información esencial para poder usarlo. La experimentación se realizó en tres partes:

- Para preparar la prueba y obtener los datos, los estudiantes rellenaron un cuestionario con algunas preguntas sobre sus habilidades de programación, su experiencia con BTs, sus estudios previos para clasificarlos y además, tuvieron que completar una pequeña prueba de programación con diez preguntas para evaluar sus conocimientos. Esta prueba se hizo para valorar de una forma cualitativa y más objetiva sus conocimientos, ya que en la encuesta, un usuario puede decir que sabe programar,

pero luego en realidad sus conocimientos no se correspondan con lo que el usuario indicó en la encuesta.

- Posteriormente, se realizó un ejercicio práctico que consistió en dos pequeñas prácticas que se describirán a continuación en la que los sujetos creaban el comportamiento a alto nivel de dos NPCs usando *Behavior Bricks*.

- Y, por último, un cuestionario final para evaluar la experiencia de usuario con la herramienta.

Llevamos a cabo un diseño previo de la experimentación que fue publicado en (Sagredo-Olivenza et al., 2015c) basado en una demostración de Unity que enfatizaba cómo implementar comportamientos de sigilo. Con el feedback obtenido en dicha conferencia, modificamos el diseño de la experimentación y agregamos los conceptos de comportamientos de bajo o alto nivel y la restricción de ciertos nodos, para simplificar la tarea a los diseñadores, ya que nos dimos cuenta de que algunos diseñadores podrían tener problemas para crear los comportamientos de bajo nivel y por tanto, nos centramos en la implementación de los conceptos a más alto nivel. También limitamos el uso de algunos nodos como describimos en la metodología. Para esta experimentación, cambiamos el dominio y usamos una primera versión del juego defensa de torre Towot, descrita en el apartado 3.3

Para realizar los experimentos, usamos la documentación de los diseñadores del juego para dar a los sujetos de la prueba una descripción real de lo que un diseñador suele hacer al describir un comportamiento. De entre todos los comportamientos descritos en el documento de diseño, se seleccionaron dos para el experimento: el enemigo básico y el escudo.

3.6.0.1. El enemigo básico.

El enemigo básico es descrito por los diseñadores de la siguiente forma:

- El enemigo básico recibe una entidad objetivo de su sistema de percepción, en este caso sólo están disponibles el núcleo y el jugador, ya que el resto de posibles objetivos no tienen sentido y se desactivaron. La selección de este objetivo es aleatoria con diferentes probabilidades para cada tipo de objetivo.

- Mientras que la percepción no cambie, el objetivo a atacar será perseguido hasta que éste esté dentro de su rango de ataque.

- Si el objetivo entra en su rango de ataque, el enemigo lo atacará.

- Si la percepción detecta otro objetivo, el enemigo básico también debe cambiar su objetivo.

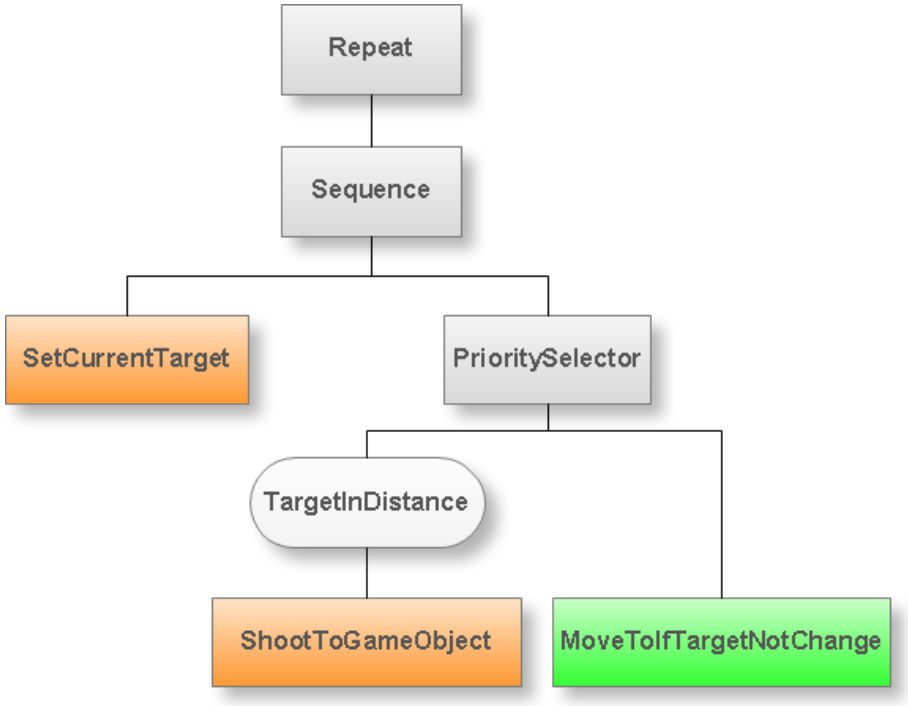


Figura 3.7: La solución esperada para el ejercicio 1: El enemigo básico

La percepción se encarga de seleccionar el objetivo que el NPC debe considerar. Esta percepción está implementada en el experimento como una acción y los usuarios no deben preocuparse por ella, simplemente deben saber usarla dentro del árbol que van a generar. En la Figura 3.7 podemos ver una posible solución al comportamiento descrito por los diseñadores.

3.6.0.2. El escudo.

Este segundo comportamiento está dividido en dos ejercicios. En el primer ejercicio, los estudiantes debían implementar el comportamiento *AttackTheCore*, que se define de la siguiente forma:

- El objetivo inicial del enemigo es el core.
- El NPC deberá moverse junto al core y dispararlo.

Este comportamiento es un comportamiento que podríamos definir de bajo nivel. Pero es muy simple, por lo que se podría implementar por los diseñadores sin problema. Añadimos este comportamiento para medir si realmente estos comportamientos también son posibles de crear por los diseñadores.

la solución esperada para el comportamiento descrito se muestra en la Figura 3.8.

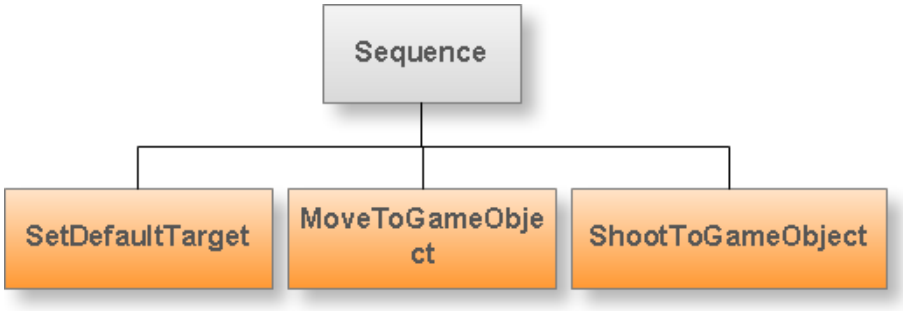


Figura 3.8: Solución esperada para el comportamiento AttackTheCore

En la segunda parte del ejercicio, los usuarios debían usar el comportamiento AttackTheCore para crear el comportamiento final del NPC. La descripción del comportamiento para el Escudo es la siguiente:

- El Escudo intenta proteger a otro enemigo.
- Mientras protege a un enemigo, intenta maximizar el área de protección de este enemigo, es decir intenta cubrir bajo su paraguas de protección, el mayor número de enemigos posibles.
- Si el objeto a proteger es destruido, el NPC intentará buscar otro enemigo a proteger.
- Si no encuentra un enemigo al que proteger, atacará a la base.

Como en el primer ejercicio, la percepción le fue proporcionada a los diseñadores para que no tuvieran que implementarla. La solución esperada para este comportamiento se muestra en la Figura 3.9.

El conjunto de acciones primitivas que los usuarios podían utilizar era el siguiente:

- Comportamientos previamente creados (Comportamientos de bajo nivel):
  - *ProtectEnemy*: Este comportamiento protege a un NPC que se le proporciona por parámetro, persiguiéndolo si es necesario. La posición que ocupa el NPC siempre es la que intente abarcar la mayor cantidad de enemigos posibles. Este comportamiento requiere calcular el baricentro de los enemigos, por este motivo, lo consideramos un comportamiento de demasiado bajo nivel para que sea implementado por un diseñador.
  - *MoveToIfTargetNotChange*: El NPC persigue a un objetivo, mientras este no cambie. Si cambia, la acción devolverá fallo y terminará.

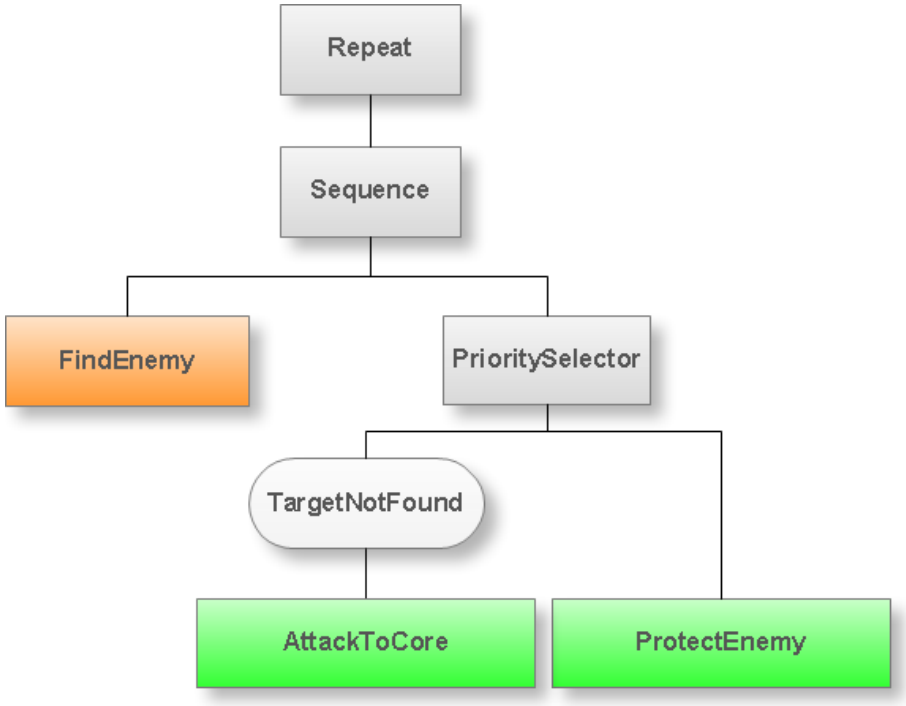


Figura 3.9: Solución esperada para el comportamiento del NPC Escudo (Shield)

- Actiones:
  - *SetCurrentTarget*: Esta acción lee de la percepción del enemigo básico y adquiere el objeto al que debe atacar. Esta información es compartida en la pizarra del comportamiento para que pueda ser leída por el resto de tareas.
  - *ShootToGameObject*: La acción recibe como parámetro el NPC al que debemos disparar.
  - *SetDefaultTarget*: Esta acción establece el objetivo por defecto en la pizarra.
  - *FindEnemy*: La acción lee de la percepción si existe un objetivo a la vista y lo escribe en la pizarra.
  - *MoveToGameObject*: La acción mueve al NPC hasta un punto señalado con un GameObject de Unity (entidad).
- Condiciones:
  - *TargetInDistance*: Comprueba si el objetivo que se le pasa por parámetro está a una distancia inferior a la dada como segundo parámetro.

- *TargetFound*: Esta condición determinan si el objetivo proporcionado por parámetro existe o no (si es nulo).
- *TargetNotfound*: Acción contraria al anterior.

3.6.1. Descripción del experimento

Con este experimento, queremos evaluar si existe una correlación entre los conocimientos de programación de los sujetos del experimento y sus resultados usando *Behavior Bricks*. Para ello, hemos comparado las calificaciones obtenidas en una prueba de programación y las calificaciones obtenidas en los ejercicios prácticos con *Behavior Bricks*.

En nuestras hipótesis, esperamos que los usuarios que sepan programar, resolverán mejor los ejercicios, aunque esperamos que los diseñadores no técnicos puedan implementar algunos de los tres comportamientos planteados.

La muestra de este experimento fue de 25 estudiantes y tuvieron 2 horas y 10 minutos para resolver los dos ejercicios. Una vez completado el experimento, evaluamos los ejercicios. La puntuación de cada ejercicio ha sido la siguiente: 4 puntos para el primero, 2 puntos para el ejercicio 2.1 ya que es similar al primero, y 4 puntos al ejercicio 2.2. Si el ejercicio no estaba completo, se puntuó una fracción de la puntuación máxima.

Durante el ejercicio, hemos registrado el tiempo empleado para resolverlo, y también hemos validado si el ejercicio fue correcto. Los estudiantes no tienen mucho entrenamiento previo, sólo un día de clase y un tutorial de la herramienta, así que, inicialmente los estudiantes tuvieron algunos problemas para completar el primer ejercicio, como se muestra más adelante en los resultados. Sin embargo, pronto comenzaron a comprender cuál era la forma de trabajo de *Behavior Bricks*.

3.6.2. Resultados del experimento

Los resultados del experimento fueron muy interesantes. La gráfica que podemos ver en la Figura 3.10, muestra una correlación evidente entre el resultado del examen de programación y el resultado del ejercicio usando *Behavior Bricks*, con una pendiente de la recta de regresión de 0,58.

Ademas de la gráfica, calculamos el coeficiente de correlación de Pearson (PCC), que sigue la siguiente ecuación:

$$\rho X, Y = \frac{Cov(X, Y)}{\sigma_x \sigma_y} \tag{3.1}$$

Donde:

- X es la puntuación del examen.
- Y es la puntuación del comportamiento.

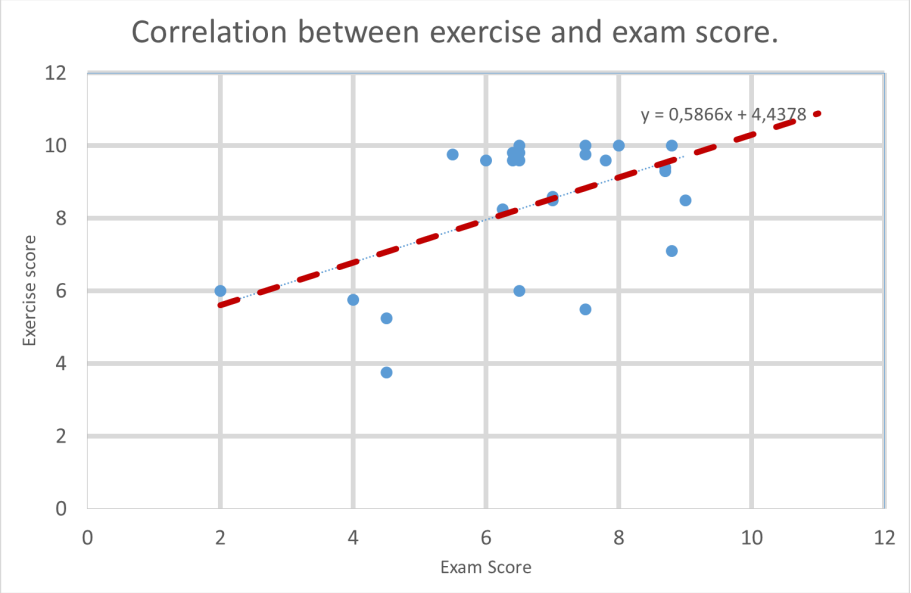


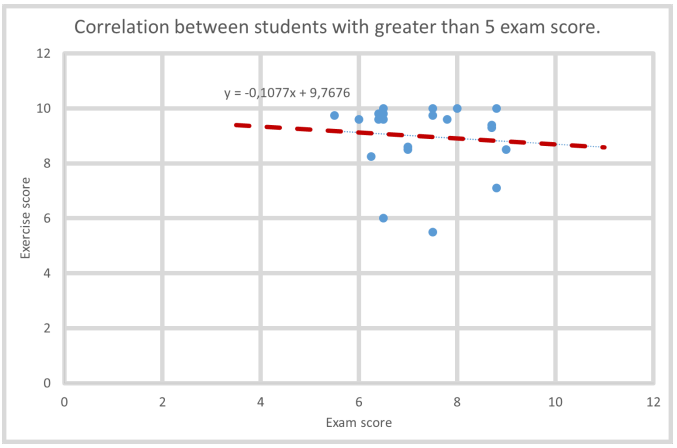
Figura 3.10: Correlación entre los conocimientos de programación y los resultados obtenidos en la creación de comportamientos.

Con el que obtuvimos un valor de 0,525, lo que indica que existe una correlación (su valor es superior a 0,5), pero dicha correlación es débil.

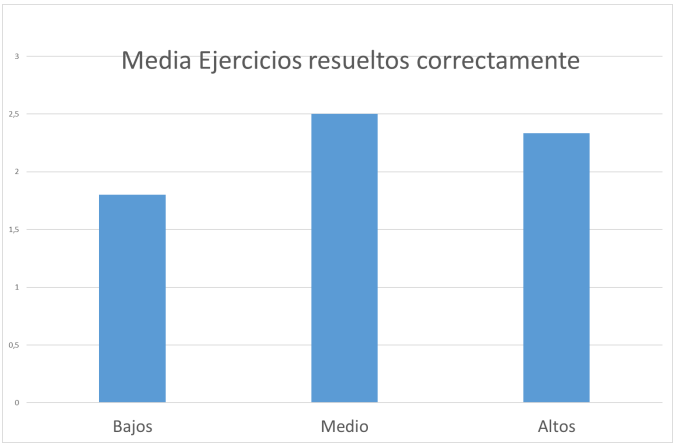
Es decir que, sí que parece que los conocimientos de programación influyen en los resultados obtenidos, pero como se demuestra en los experimentos, muchos diseñadores sin conocimientos de programación, si pueden crear comportamientos de más alto nivel con conciertas garantías, algo que inicialmente nos sorprendió ya que esperábamos una mayor correlación.

Al principio nos sorprendió este resultado, ya que pensábamos que la relación conocimiento de programación y resultados obtenidos con los árboles de comportamiento iba a ser mucho más evidente. Tampoco fue la sensación que percibimos en el experimento, donde notamos que los diseñadores sin conocimientos de programación tenían más dudas que los programadores. Sin embargo, si analizamos los datos con más calma, se puede ver que la dependencia de la creación de BTs, con los conocimientos de programación sí que aparece en ellos.

Si analizamos cuidadosamente los resultados, encontramos que, dado un conocimiento de programación básico, las diferencias entre los resultados del ejercicio y el examen no siguen ningún tipo de correlación. Es decir, que a partir de ciertos conocimientos de programación, no hay evidencias de que cuanto más conocimientos se tenga, mejor se usa *Behavior Bricks*. La barrera está entre los que tienen ciertas nociones de programación y los que no saben programar. Esto creemos que es debido a que la utilización de una herramienta visual como *Behavior Bricks* simplifica mucho la creación de los



(a) Correlación entre el examen aprobado y los comportamientos



(b) Número de ejercicios resueltos correctamente según sus conocimientos de programación.

Figura 3.11: Análisis de los resultados del experimento por categorías.

comportamientos para aquellos que tienen ciertas nociones de programar, haciendo que el ser buen o mal programador, en un experimento de estas características, no se vea reflejado en los resultados. Pesan más otros factores como la experiencia con herramientas similares o la capacidad del propio individuo para realizar los ejercicios planteados.

Sin embargo, si no se tienen los conceptos necesarios para entender o poder extraer de un comportamiento las condiciones que lo gobiernan, o la secuencia de pasos que determina su funcionamiento o simplemente saber asignar los parámetros de entrada de las acciones correctamente, ahí es cuando las diferencias se hacen más evidentes. Esto se puede ver en la Figura 3.11a donde se muestra que no existe ninguna correlación entre ambas magnitudes si eliminamos a aquellos con las notas de los exámenes más bajas

(inferior a 5).

Este hecho está también afectando al coeficiente de correlación de Pearson, ya que hay una parte de la distribución que no está correlacionada y otra parte que sí lo está. Donde realmente se ve la correlación es precisamente en el salto entre programadores y no programadores. Es decir entre los usuarios con menos de 5 en el examen y los usuarios con más de 5 en el examen. Si clasificamos los estudiantes en tres clases según sus conocimientos de programación en: bajos (por debajo del 5), medios (entre el 5 y el 7) y altos (por encima del 7), podemos ver cómo el número medio de ejercicios resueltos correctamente es inferior entre los que tienen menos conocimientos de programación, que en los que tienen algún conocimiento de programación y nuevamente es prácticamente el mismo entre los usuarios que tienen al menos unos conocimientos mínimos y los que sacaron mejores notas en el examen. Podemos ver estos resultados en la Figura 3.11b

Con estos datos podemos concluir que, sin conocimientos de programación, efectivamente como suponíamos se tienen peores resultados creando comportamientos a alto nivel, pero sí que es posible usar *Behavior Bricks* en muchos casos para crear comportamientos más sencillos como los del primer y segundo ejercicio, donde muchos de los sujetos del estudio completaron al menos el primer ejercicio y muchos el segundo, incluso sin saber programar. Tan sólo uno de los participantes con menos conocimientos técnicos consiguió completar todos los comportamientos. Sin embargo, la media de los que sí tenían conocimientos técnicos sí que parece uniforme. Es decir, a partir de unos ciertos conocimientos técnicos sobre programación, no se consigue una mejora sustancial en el uso de la herramienta, al menos con el nivel de complejidad de los comportamientos con los que se hizo el estudio. Sólo si se carece de ellos, es cuando cuesta más entender los árboles de comportamiento a alto nivel y por consiguiente, se producen peores resultados.

Capítulo 4

Programación por demostración en Behavior Bricks

4.1. Introducción

Como vimos en el apartado 3.5, nuestra metodología de uso de BT para diseñadores y programadores, usando como nexo de unión *Behavior Bricks*, es eficaz a la hora de crear ciertos comportamientos de alto nivel por parte de los diseñadores, incluso si estos no disponen de demasiados conocimientos técnicos. Sin embargo, en el estudio empírico realizado, se detectó que los diseñadores con pocos o ningún conocimiento de programación, tenían más problemas a la hora de usarlos que aquellos que sí disponían de estos conocimientos. Por lo tanto, si queríamos dotar a los diseñadores de una herramienta realmente potente y útil para que fuesen más autónomos a la hora de crear comportamientos, necesitábamos ir más allá.

Para conseguirlo, pensamos qué tenían en común la mayoría de los diseñadores de videojuegos, ya que suelen provenir de diferentes perfiles profesionales, diferentes sensibilidades o incluso diferentes culturas. Lo cual no significa que sea malo, sino todo lo contrario, ya que un diseñador, o al menos el equipo de diseño en su conjunto, necesita tener conocimientos de múltiples áreas y artes, ya que los videojuegos actuales tienen multitud de referencias culturales, arquitectónicas, cinematográficas y creativas en general, así como también se valoran mucho otros conceptos más técnicos como tener conocimientos de programación, de negocio o de animación (Schell, 2014). Pero obviamente es muy complicado encontrar perfiles tan variados en una sola persona y es el conjunto del equipo el que suele estar compuesto por miembros de diferentes sensibilidades. Pero queríamos encontrar cuál era el nexo de unión de todos o de la mayoría de los diseñadores. Entonces nos dimos

cuenta que lo que sí saben hacer la mayoría de los diseñadores es jugar a videojuegos.

Jugar a otros juegos es una parte importante en la formación de un diseñador (Granberg, 2014) y de su background como profesional del medio, por lo tanto, es algo que presuponemos, el diseñador sabe hacer y le gusta hacer. De esta forma, pensamos que para cubrir esta necesidad y hablar en el mismo lenguaje que un diseñador conoce, debíamos poder crear una herramienta que permitiera que los diseñadores pudieran generar comportamientos jugando, porque es la forma más intuitiva para ellos de ver si un comportamiento funciona o no funciona, si es a la postre divertido o no. Porque al jugar, el diseñador puede descubrir nuevas mecánicas o situaciones que sobre el papel, no había tenido en cuenta y porque en un proceso interactivo como en el que se desarrollan hoy en día los videojuegos, cuanto antes se vean implementadas las funcionalidades, más rápidamente se ven las carencias o las necesidades que éstas implican.

Por lo tanto, lo más importante para una correcta colaboración entre ambos roles, es que el diseñador juegue cuanto antes para que la propia experiencia de juego guíe sus decisiones a la hora de crear la especificación del comportamiento. Además, el hecho de trabajar en paralelo con el programador y saber de primera mano qué es lo que se quiere conseguir con cada comportamiento, debería ayudar a reducir problemas de comunicación entre ambos. Ante esta necesidad, la aproximación que de forma natural surge es utilizar programación por demostración. Esta tecnología permite programar ejemplificando cómo se debe hacer una tarea, en nuestro caso, cómo debe jugar el NPC.

La programación por demostración (Programming By demonstration o learning by observation, PbD) es una técnica o conjunto de técnicas que permite a una máquina ser programada mediante la observación de las acciones que realiza un actor, normalmente experto en resolver el problema que se pretende resolver. Por lo tanto, la máquina aprende a realizar una tarea por imitación u observando de lo que hace el experto. En nuestra aproximación, el experto es el diseñador del videojuego, que es el más indicado para tener una información más precisa de lo que un NPC debe hacer.

A diferencia de otros dominios donde se busca una respuesta clara del experto (por ejemplo, ante una serie de síntomas, un médico proporcionaría una lista de posibles causas ordenada por probabilidad según su experiencia), en videojuegos, el experto no tiene todas las respuestas de antemano. Es decir, tener el mejor diseñador del mundo no hará que el sistema, aun modelando a la perfección los comportamientos enseñados por este, genere un comportamiento válido que vaya a ser el comportamiento final del NPC en el juego. El comportamiento final del NPC en el juego es el resultado de un proceso de descubrimiento iterativo, en el que el propio experto debe, en cierta medida, encontrar cuál es. Seguramente el diseñador tiene un

comportamiento en mente que cree óptimo, pero es muy probable que no sea el definitivo y que el comportamiento final sea muy diferente al ideado inicialmente.

Así pues, es necesario construir un sistema en el que sea fácil generar diferentes alternativas para un comportamiento, con diferentes opciones, e iterar sobre ellas hasta encontrar la mejor. Por lo que el diseñador necesita de un entorno flexible, que sea fácilmente modificable y ampliable para poder probar estas ideas rápidamente y descartarlas lo antes posible, si no son viables o no son divertidas. A la vez, necesita poder reutilizar en la mayor medida posible los comportamientos o tareas que ya se han realizado previamente, ya que esto facilitará la exploración de soluciones lo más pronto posible y reducirá enormemente el tiempo de desarrollo.

Podemos definir de una forma más formal la programación por demostración como la unión de cinco elementos:

- El comportamiento a aprender (*Behavior*) que denotamos con la letra (B).
- El conjunto de tareas (*Tasks*) posibles a ejecutar, que denotamos con la letra (T).
- El entorno (*Environment*) donde se ejecuta la tarea T, que denotamos con la letra (E).
- El actor (*Actor*) o experto en el problema (A).
- El sistema (*System*) o agente que desea aprender a resolver el problema (S).

De forma general, como hace de forma similar Ontañón et al. (2014), podemos definir el proceso de aprendizaje por demostración como aparece en la Figura 4.1. El actor A recibe como parámetro la información del comportamiento a aprender B y las tareas disponibles T. Una vez conocido el comportamiento a aprender, A percibe el estado del entorno E y elige la tarea $t \in T$ que considere más adecuada (donde la función teórica del comportamiento maximice resultados) en función del entorno E. La tarea t se ejecuta en el entorno E y se guarda una traza de lo ocurrido en el sistema S.

Esta técnica se ha utilizado profusamente en robótica (Lozano-Perez, 1983; Segre, 2012), pero en videojuegos hay algunas reticencias a utilizarla, debido a que los diseñadores suelen querer tener un alto control sobre los comportamientos generados. Ésto es un hecho conocido en la industria, hasta el punto que desde la Universidad se ha intentado que la industria utilice más estas técnicas en videojuegos, prueba de ello son algunas conferencias como la GDC de 2015, donde los creadores del motor de física Havok®, y la universidad de Pennsylvania (Sunshine-Hill, 2015) trataron este hecho en una conferencia, donde se animaba a los desarrolladores a usar estas técnicas.

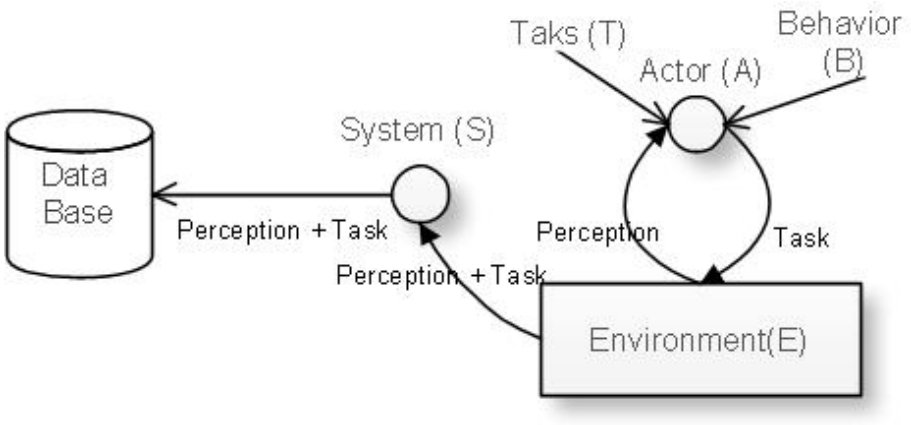


Figura 4.1: Descipción del proceso de aprendizaje de la programación por demostración

Sin embargo, algunos juegos sí la han utilizado, sobre todo en juegos competitivos, porque la programación por demostración modela muy bien el comportamiento humano, lo que hace que la IA vaya adaptándose al jugador para que éste tenga una mejor experiencia de juego. Esta técnica permite que la IA se comporte de una forma más realista, por lo que algunos juegos la han puesto en práctica como por ejemplo en (Derek Neal, 2016). Aun así, no es la tónica habitual, ya que un comportamiento aprendido por demostración, tiene un cierto grado de incertidumbre sobre el resultado obtenido y en ciertos juegos, esto puede suponer un problema de diseño, ya que el NPC puede hacer algo que por diseño no se quiere que suceda. Así que, usar programación por demostración para aprender todo el comportamiento de un NPC, es una tarea muy compleja que además puede producir comportamientos poco fiables que generarán rechazo por parte del diseñador. Así pues, lo interesante de nuestra propuesta es que este tipo de generación de comportamientos por demostración se integra dentro de otra técnica totalmente determinista y controlable por el diseñador y los programadores, como son los árboles de comportamiento.

La idea, por tanto, no es sustituir los árboles de comportamiento por programación por demostración, sino más bien extender las capacidades de los BTs con la posibilidad de aprender ciertas funcionalidades por demostración. De esta forma el diseñador podrá crear por demostración sólo ciertas partes del comportamiento, aquellas que el equipo de desarrollo considere oportunas, con la finalidad de ayudar al diseñador a crear comportamientos por sí mismo, sin necesidad de tener conocimientos avanzados de programación. A estos nodos especiales que son capaces de aprender por demostración los hemos denominado *Trained Query Nodes* y los detallaremos más profusamente en la siguiente apartado.

4.2. Trained Query Nodes

Como hemos comentado en la introducción del capítulo, el objetivo que pretende nuestra aproximación es hacer a los desarrolladores más autónomos de los programadores, permitiendo sustituir partes de un BT por un nodo especial que hemos denominado *Trained Query Node* (TQN), cuya finalidad es aprender por demostración una parte del árbol de comportamiento. En concreto la rama que cuelga bajo el punto del árbol donde se estableció el nodo. El nodo sólo aprende el primer nivel del árbol por debajo de él, es decir, sólo aprende qué hijo del nodo debe seleccionar en cada momento, por lo que si uno de los hijos del nodo no es un nodo termina, dicho sub-arbol debe estar abstraídas en un subcomportamiento o una acción del sistema que lo implemente. En caso contrario, las diferentes alternativas del nodo deberán ser aprendidas también por demostración.

El nombre de *Trained Query Node* proviene de que este nodo, es capaz de hacer una consulta a una base de datos (la base de datos de tareas disponibles) y ejecutar una tarea al vuelvo en tiempo de ejecución que concuerde con la especificación del entorno dada en ese momento. Esta idea proviene de los trabajos realizados por Flórez-Puga et al. (2009) en los que aparece un nodo especial que se denominó *Query Node*. El Query Node según Florez Puga, es capaz de recuperar comportamientos de una base de comportamientos en base a una descripción ontológica de dicho comportamiento. En el Query Node original, se debe especificar una descripción del comportamiento que se quiere recuperar y el nodo busca los comportamientos más similares a dicha especificación, seleccionando uno de ellos, el que considere más similar a la descripción dada. Esto se realiza en tiempo de ejecución, de forma que se seleccionará unas veces un comportamiento y otras veces otro, dependiendo de la información del entorno que más se adecue a la especificación de cada comportamiento.

La idea de los *Trained Query Nodes* es similar, pero tiene algunas diferencias importantes. Para definir el comportamiento no se necesita definir una ontología compleja, simplemente se utiliza un nombre semántico que identifica dicho comportamiento y se construye una relación jerárquica entre las tareas disponibles en la biblioteca. Ese nombre representa la tarea que quiere ser aprendida y el sistema recupera aquellas tareas que implementan la tarea que hemos definido. Estas tareas que implementan la tarea que queremos aprender, pueden implementarla de formas muy diferentes. Para seleccionar cuál es la más óptima, no se utiliza la definición ontológica, sino las trazas generadas por la observación del experto, en nuestro caso, el diseñador. De esta capacidad de aprender en qué contexto debo aplicar qué tarea, en base a una base de casos que el sistema graba del experto, es de donde sale el término *Trained* del nodo.

Así pues, podemos asociar los diferentes elementos de los TQN a la defi-

nición formal de programación por demostración especificada en la introducción del capítulo, para ver que en efecto, lo que hace el *Trained Query Node* es programación por demostración. La tarea a aprender en nuestra aproximación sería el comportamiento a aprender, B en el modelo formal descrito. Las tareas disponibles que implementan la tarea a aprender serían el conjunto de tareas T del modelo formal. El Actor A no es otro que el diseñador y por último, el sistema S es el propio NPC y el entorno de entrenamiento.

Los *Trained Query Node* se implementan sobre el editor de árboles de comportamiento de *Behavior Bricks*, que describimos en el apartado 3.4 y utilizan una de sus principales características en su beneficio, la posibilidad de definir tareas que se implementan de diferentes formas pero que sean compatibles con dicha tarea base, de forma que se crea una relación jerárquica de tareas, donde unas tareas pueden implementar a otras. Teniendo una jerarquía de tareas bien definida, podemos aprender cualquier sub-árbol de esa jerarquía, simplemente estableciendo la raíz de ese sub-árbol como la tarea a aprender en el *Trained Query Node*. Veamos un ejemplo para ayudar a entender este mecanismo. Supongamos que en Towot, nuestro entorno de pruebas, tenemos la siguiente jerárquica de tareas que se muestra en la Figura 4.2

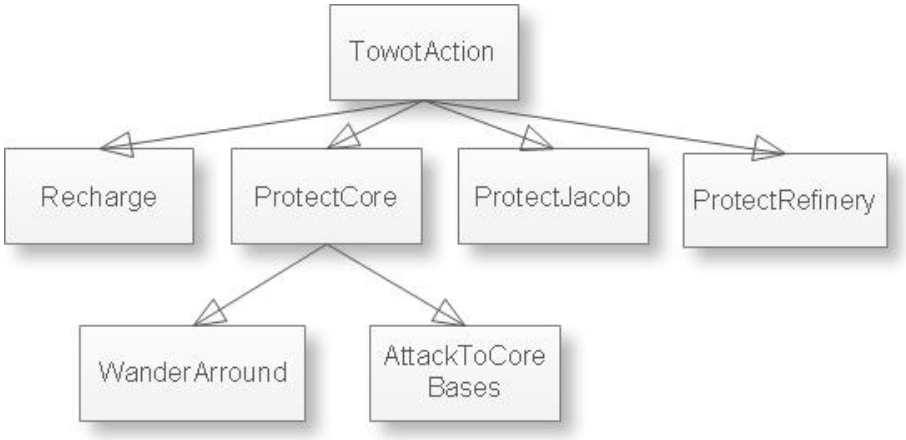


Figura 4.2: Una posible jerarquía de tareas del Towot como ejemplo para ilustrar cómo aprender una tarea usando TQN

Si queremos aprender cuál es la mejor implementación de la tarea *ProtectCore* en este contexto del juego, debemos establecer en el *Trained Query Node* está tarea como la tarea a aprender. Nótese que podemos aprender cualquier tarea dentro de la jerarquía, así pues, podríamos aprender todos los comportamientos de alto nivel del NPC. Cómo veremos más adelante, esto es lo que hemos hecho en la experimentación usando Towot, ya que la tarea a aprender en los experimentos realizados fue precisamente "TowotAction" que está implementada de 4 formas diferentes: *ProtectCore*, *ProtectJacob*,

ProtectBase y *Recharge*.

El TQN tiene dos modos de funcionamiento, el modo de entrenamiento y el modo de ejecución. En el modo de entrenamiento, cuando el BT decide que debe ejecutar este nodo, va tomando muestras del estado del juego cada cierto tiempo y las va guardando en una base de casos. La frecuencia con la que se van guardando estos casos es un parámetro del sistema que el diseñador puede decidir cuando configura el nodo. Un caso está compuesto por un subconjunto de los parámetros que están almacenados en la pizarra del árbol que ejecuta el TQN.

Ese subconjunto de parámetros del comportamiento componen los atributos que formarán parte del caso que se almacenará y son decididos por el diseñador al comienzo del entrenamiento. Estos atributos deben ser relevantes para el comportamiento. No siempre es fácil decidir qué atributos son relevantes para un comportamiento, por lo que el diseñador debe seleccionarlos con cierto cuidado. Sin duda, puede seleccionar todos, pero una buena selección de los atributos que se usarán para construir el modelo es importante, ya que si no incluyen atributos innecesarios se elimina el ruido que puede producir introducir atributos que no aportan información al modelo que se pretende construir. Siempre se puede utilizar un algoritmo de selección de atributos como *prueba de Chi cuadrado*, *ratio de ganancia de información* o el *Correlation Feature Selection* (Hall y Holmes, 2003; Blum y Langley, 1997) entre otros, para seleccionar los atributos de forma automática, aunque actualmente la herramienta no dispone de ningún algoritmo de selección de atributos implementado.

De todas formas, hay que tener cuidado si se usa alguna de estas técnicas de selección de atributos automática, ya que algunas de ellas pueden eliminar atributos que sí son relevantes. Una buena elección de los atributos contribuye a que el entrenamiento sea más eficaz, pero por el contrario, si no se introduce la información suficiente, es posible que no podamos aprender lo que deseamos, así pues, es preferible añadir atributos de más que de menos.

Una vez que el diseñador establezca la tarea a aprender y los atributos de la pizarra de comportamiento que quiere guardar, el diseñador debe ejecutar el *Trained Query Node* en modo entrenamiento. Éste irá captando la información del entorno e irá almacenándola en la base de casos cada cierto tiempo. Esa base de casos es la base del entrenamiento posterior del sistema, así que cuanto mejor esté construida, mejores resultados producirá.

Para generar la base de casos se requiere crear un entorno de entrenamiento dentro del juego que permita al diseñador entrenar cómodamente al NPC. Crear un entorno de este tipo es sencillo ya que los TQN ofrecen, en cada iteración, las acciones que implementan las tareas a aprender que se encuentran disponibles en el sistema, así como el valor de los parámetros que se van a almacenar.

De forma que estos valores se pueden mostrar al usuario y mediante una

sencilla interfaz, permitir al diseñador seleccionar en cada momento la tarea a ejecutar. Es importante desde nuestra perspectiva, mostrar la información del entorno que se va a almacenar, ya que es una ayuda al diseñador para tomar la decisión de qué tarea se debe ejecutar y cuándo es más apropiado hacerlo.

También consideramos que implementar una función de interrupción del juego es una característica interesante para que la toma de decisiones sea pausada y meditada, sobre todo si lo que se está aprendiendo del juego no es una tarea que requiera una reacción inmediata del entrenador (pensemos en un juego de lucha, en este contexto no tendría sentido interrumpir para que el diseñador tomase una decisión), el diseñador podrá parar el juego para elegir la acción a ejecutar cuando él estime oportuno, controlando al NPC.

Vamos a ver este proceso de entrenamiento con el ejemplo de nuestro juego de pruebas nuevamente. Supongamos que queremos entrenar el comportamiento *TowotAction*. Para ello se construye una interfaz de entrenamiento como la que se muestra en la Figura 4.3.

Figura 4.3: Interfaz de entrenamiento del Towot

En ella podemos apreciar por ejemplo en la esquina superior derecha, el valor de los atributos del entorno que se usan para ayudar al diseñador en la toma de decisiones. También aparece en la captura el momento en el que el diseñador está ordenando al Towot una acción. Para que el diseñador tenga tiempo de tomar una decisión tranquila y meditada, cuando el diseñador pulsa la tecla i de su teclado, el juego se interrumpe y se muestra el menú de selección con las tareas disponibles en el sistema. El diseñador selecciona una de ellas y el Towot la ejecuta. Mientras tanto, el TQN va almacenando cada cierto tiempo el estado de la pizarra y la acción que el nodo está ejecutando.

Para conseguir todo esto, el TQN ofrece una interfaz muy sencilla para ser

manejado desde fuera del BT con métodos de parar el BT, leer la información de la pizarra del BT, etc. Por las características del juego de pruebas, donde la cámara no sigue al NPC sino al jugador, se ha colocado en la esquina superior izquierda, un visor que se puede o no ocultar para que el diseñador pueda vigilar lo que hace el Towot, así mismo, la posición del Towot se indica en el minimapa. En otros juegos donde el NPC a entrenar no dependa tanto de la acción del jugador, puede centrarse la cámara en el propio NPC para controlarlo más fácilmente. En nuestro caso la solución del visor nos pareció la más adecuada para poder controlar ambos.

Una vez realizado el proceso de entrenamiento del NPC, la base de casos generada se puede asignar al TQN para que, en modo ejecución, sea capaz de modelar el comportamiento que el diseñador le ha indicado por demostración. Para ello, se pueden aplicar diferentes técnicas para generar un modelo que tome la decisión de qué tarea se ha de ejecutar en cada momento. Cada una de ellas tiene unas determinadas ventajas e inconvenientes. En nuestro trabajo hemos explorado tres diferentes: Árboles de decisión, Razonamiento basado en Casos y Redes de neuronas.

Independientemente de la técnica utilizada para seleccionar la acción a ejecutar, el nodo cada cierto tiempo debe consultar el estado de las variables que el diseñador seleccionó como relevantes en el árbol de comportamiento y en base a esta información, consultar el modelo para que le devuelva la tarea que el sistema considera más acertada. Una vez realizado esta selección, el nodo ejecuta la tarea. Mientras la tarea no termine o no lleve ejecutándose más tiempo del indicado en el tiempo de reevaluación, la tarea sigue en ejecución. Si la tarea termina, el TQN termina y el valor de terminación del TQN será el valor de terminación de la tarea que acaba de finalizar. Si se cumple el tiempo de re-evaluación y la tarea seleccionada es la misma, nada sucede y ésta continúa ejecutándose sin problemas. Si hay un cambio de tarea, es decir, la tarea seleccionada es otra diferente, entonces se interrumpe la tarea en curso y se ejecuta la nueva a continuación.

Para ser consecuentes con cómo se fue creando la herramienta y el porqué se añadieron los tres tipos de modelos disponibles, en la siguiente subapartado sólo vamos a detallar el modelo basado en CBR, ya que fue el primero que se creó y en el que se centra el experimento de el apartado 4.4. Posteriormente describiremos por qué se añadieron los demás modelos y también explicaremos su funcionamiento.

4.2.1. Modelado del comportamiento usando razonamiento basado en casos

Una de las técnicas a emplear para generar el modelo de comportamiento es la basada en razonamiento basado en casos (Aamodt y Plaza, 1994; Kolodner, 2014). Este método consiste en buscar las soluciones a nuevos problemas basándose en las soluciones similares a problemas anteriores. Se

ha argumentado que el razonamiento basado en casos no sólo es un método poderoso para el razonamiento de computadoras, sino que es usado por las personas para solucionar problemas cotidianos.

Este sistema de razonamiento se basa en una unidad mínima llamada caso. Un caso se puede definir como una representación de una experiencia anterior, o, dicho de otro modo, una vivencia. Estos casos son extraídos de un contexto y deben estar basados en la experiencia previa, por lo que podemos considerarlos ciertos desde el inicio. Además, el hecho de que hablemos de experiencia, implica que este sistema está muy ligado a la adquisición de conocimiento externo, lo que favorece a que se vaya adquiriendo nuevas experiencias para mejorar su razonamiento con el tiempo.

En ese sentido, el algoritmo es muy flexible ya que no crea un modelo a partir de los datos almacenados, si no que dicho modelo se infiere por similitud en tiempo real, lo cual hace que aunque los datos cambien y se añadan nuevos, el modelo no debe ser re-calculado como en otros algoritmos que veremos más adelante. Las experiencias que tenemos en el sistema no se refieren a cualquier experiencia sino sólo a aquellas que nos aportan alguna información sobre el tema tratado por el sistema experto, además es importante no repetir experiencias ya existentes con el mismo contexto, ya que no aportan nueva información al sistema y lo hacen más caro de ejecutar.

El CBR por tanto se adapta muy bien a las características de nuestro problema. El entrenamiento del NPC genera casos que se almacenan en la base de casos. Estos casos forman una biblioteca de experiencias previas para un dominio concreto. A la información aportada, se le pueden añadir nuevos ejemplos, siempre que éstos no sean repetidos, para perfeccionar el entrenamiento. Esto lo podemos llevar a cabo si detectamos que no hay suficientes casos que representen una acción concreta del NPC.

Por lo tanto, la fase de captación de ejemplos es el propio aprendizaje del sistema y el diseñador no necesita hacer una segunda fase de entrenamiento con los datos capturados para generar el modelo, siendo muy intuitivo y trasparente para el diseñador usar todo el conjunto. El sistema, antes de ejecutar, hace una pequeña comprobación de casos repetidos o muy similares para reducir el número de casos en ejecución y limpiar la base de casos.

En la fase de recuperación, se usa el algoritmo del os k vecinos más cercanos o k-Nearest Neighbors (k-NN) (B. W. Silverman, 1989) que es un algoritmo de reconocimiento de patrones, que puede ser usado tanto en clasificación como en regresión. En nuestro caso se usa para clasificar los casos de entrenamiento en las tareas a ejecutar.

Por lo tanto, el algoritmo utiliza como premisa para resolver un problema actual, la búsqueda de un problema similar en el pasado, para comprobar cómo se resolvió. La forma en la que se resolvió el problema, si el caso es muy similar, debería servir para resolver el actual. Para conseguir ésto, se necesita estar en el mismo contexto y además disponer de algún mecanismo

que permita comparar la similitud del caso actual con los almacenados.

En nuestro sistema, al realizar el entrenamiento por observación, guardamos en un vector los parámetros seleccionados por el diseñador como relevantes, seguidos de la acción a resolver. Estos parámetros tienen un rango que también es definido por el diseñador y que sirven para normalizar las variables a un valor comprendido entre 0 y 1. La información que no sea numérica, debe ser discretizada y codificada numéricamente para que el algoritmo funcione. Como medida de similitud existen numerosas en la literatura como en (Deza y Deza, 2009), nosotros hemos utilizado la distancia euclídea ponderada (Ecuación 4.1)

$$d(A,B) = \sqrt{\sum_{i=1}^{n}(\frac{w_i}{W})(A_i - B_i)^2} \tag{4.1}$$

donde w_i es el peso del atributo i, W es la suma de los pesos, A_i, B_i es el valor de los atributos del caso íesimo.

Los pesos de los atributos (w_i) deben elegirse por los diseñadores en función de la importancia de ese parámetro en la toma de decisión en cada entorno y sólo son usados en la fase de recuperación.

Esta información se puede considerar como una heurística que ayuda o guía al algoritmo a dar más peso a un parámetro que a otro, pero es una información que a priori puede no conocer el diseñador, ya que una correcta selección de los pesos implica conocer bien la estructura de los datos que se están almacenando y cómo funciona k-NN.

Con esta función de similitud seleccionada, se compara el estado actual del entorno con la base de casos almacenada, seleccionandos los k más similares. Este parámetro k es un parámetro que el diseñador puede modificar. Si $k = 1$, entonces seleccionará el más similar. En cualquier otro caso, de entre los k seleccionado, nos quedaremos con la tarea $t \in T$ que más se repite de entre los k más similares.

Uno de los problemas de usar k-NN es su alto coste computacional si el número de casos almacenados es muy grande, ya que se debe calcular la similitud con todos los casos de la base de casos en cada ejecución. Existen optimizaciones como el particionado del espacio usando un *árbol k-dimensional* o *k-dimensional tree* (K-d tree) (Hajebi et al., 2011) que es una estructura de datos que permite dividir el espacio mediante hyperplanos, usando los diferentes parámetros de la base de casos. De esta forma, podemos reducir el número de casos con los que comparamos, ya que sólo procesaremos aquellos que están próximos en el espacio, de forma que los propios atributos guían la búsqueda. Aun así, el proceso es más costoso que otros algoritmos y tiene unas necesidades de memoria también muy elevadas.

En nuestro caso, para reducir el coste computacional en un sólo fotograma, se difiere el cálculo en varios fotogramas, de forma que el NPC tardará

algunos milisegundos en encontrar la acción a ejecutar, pero no se sobrecargará con un pico que provoca una ralentización cuando decida tomar la decisión. De esta forma se puede decidir el número de casos máximo que el NPC puede evaluar por fotograma. Así se puede tener en todo momento controlado el coste de calcular la IA en cada actualización y evitar oscilaciones en el número de imágenes por segundo.

También hay que tener en cuenta el tamaño en memoria que requiere este algoritmo, significativamente mayor que otras técnicas como las redes de neuronas o los árboles de decisión. Hay que tener cuidado con el retardo de reacción del NPC. Normalmente un NPC no debe reaccionar instantáneamente, debido a que eso implica que el usuario perciba la IA como tramposa. Debe tener unos tiempos de reacción humanos, por lo que normalmente no es un problema diferir en N fases el cálculo, pero en cualquier caso hay que tenerlo en cuenta.

4.3. Entorno de experimentación

Los experimentos realizados en esta fase del trabajo se efectuaron sobre una versión evolucionada del videojuego que ya introducimos en el apartado 3.2 denominado Towot. En esta versión, se han introducido numerosos elementos jugables nuevos. Por ejemplo, inicialmente el juego tenía dos fases dentro de escenario, una en la que se colocaban las torretas y que tenía una vista cenital y otra en la que se jugaba al juego en primera persona. Ahora el juego sólo tiene una vista en tercera persona y las torretas son colocadas por el propio jugador donde él considere oportuno, para defender la base. Sigue existiendo una parte en la que el jugador puede colocar las torretas de defensa, pero todo está integrado en el juego. Simplemente hay un tiempo hasta la primera oleada, que el jugador puede aprovechar para diseñar su estrategia de defensa, pero no existe una fase estratégica como tal, sino que todo está integrado dentro del juego. Además, se añaden nuevos objetivos al juego. Ya no solamente hay que proteger la base, sino que además en cada nivel hay que extraer un preciado mineral denominado *Uridium* de los diferentes planetas que visita el jugador.

Cada planeta es una fase del juego, en la cual se debe extraer una cantidad de mineral determinado. El mineral se encuentra ubicado en diferentes localizaciones en los escenarios y se necesita crear una refinería para poder extraerlo. Esta refinería puede ser destruida por los enemigos y por tanto hay que defenderla de éstos. Ahora, el objetivo del juego es extraer la mayor cantidad de mineral como sea posible, debiendo superar un mínimo para superar la fase, antes de que los insectos droides destruyan al jugador o a su nave espacial. En la Figura 4.4 podemos ver al fondo una mina de mineral. En primer plano, de espaldas, el avatar del jugador que se denomina *Jacob* y en la esquina inferior derecha el minimapa con el nivel del juego.

Figura 4.4: Captura de Towot en su versión actual.

Para extraer dicho mineral, se debe colocar una torreta extractora que debe ser defendida de los ataques de los enemigos, ya que es vital para completar la misión. Así pues, en esta versión del juego, no sólo hay que defender el núcleo o base, sino que también hay que defender las torretas que se creen para extraer el mineral.

Además, en esta versión no se pueden crear robots como en la versión anterior del juego. El único robot que te acompaña es *Towot*, que da nombre al juego y que es una especie de mascota cibernética, que sirve al jugador como ayuda, estando a su disposición desde el principio del nivel. El robot le ayuda a defender su base (su nave espacial) y las torretas extractoras, atacando a los enemigos que se acercan a ellas. Este robot además necesita recargar su energía para poder disparar, por lo que debe haber instalada una torreta de recarga a la que el robot debe acudir periódicamente. Por tanto, el robot debe mantener su energía en unos valores óptimos sin descuidar la defensa de los diferentes elementos del escenario.

Como todo juego de defensa de torre, los enemigos parten de zonas alejadas del núcleo o base a defender y se van aproximando por una serie de caminos predefinidos por el propio escenario. Estos enemigos van llegando en oleadas. En cada oleada, normalmente se van alternando diferentes tipos de enemigos y en teoría cada una de ellas es más complicada que la anterior, ya que tiene o bien mayor número de enemigos o enemigos más poderosos.

El comportamiento de los enemigos no es determinista, puede variar de una ejecución a otra ya que el objetivo al que atacan se elige con una probabilidad. La probabilidad de elegir la refinería o el core como objetivo es de un 50 %. Además, hay una probabilidad de un 20 % de que mientras se dirigen a su objetivo primario, el enemigo se desvíe si ve al jugador, para

atacarlo. Por lo tanto, dos ejecuciones del juego nunca será idénticas.

En el escenario de pruebas, se han diseñado tres oleadas de enemigos que podemos ver en la Figura 4.5. La más a la izquierda es la primera oleada, la del medio la segunda oleada y la que está más a la derecha la tercera oleada. En las tres capturas podemos ver la estación petrolífera, en el centro del mapa, que indica la ubicación de la refinería. Los puntos azules representan puntos de extracción de mineral, la figura amarilla en la parte baja centra es la base y en cada una de las esquinas se marca las zonas donde aparecen los enemigos. Por último, el icono de la gasolinera situado cerca de la estación petrolífera indica un punto de recarga para el Towot.

En la primera oleada, los enemigos parten del generador situado en la esquina superior izquierda y atacarán a la refinería y al núcleo. En la segunda oleada, los enemigos parten del generador inferior izquierdo y atacarán principalmente al core. Por último, la tercera oleada generará enemigos desde los dos generadores superiores y desde el generador situado en la esquina inferior derecha. Esta última podemos verla como un ataque masivo final desde varios puntos, para que los enemigos ataquen simultáneamente la refinería y el núcleo.

Figura 4.5: Las diferentes oleadas en el escenario del experimento

El mapa que muestra la figura fue el utilizado para realizar las pruebas de nuestro sistema. Por simplificar los experimentos, el jugador no puede colocar ninguna torreta. Las necesarias para superar el nivel ya están colocadas desde el principio.

Las acciones que el Towot puede ejecutar han sido abstraidas en 4 subcomportamientos.

- DefendCore: Protege al core de los enemigos.
- DefendRefinery: Protege a la refinería de los enemigos
- DefendJacob: Persigue a Jacob, el jugador, para atacar junto a él.
- Recharge: Se mueve hacia el punto de recarga, para recuperar su energía. Sin energía, el Towot no puede disparar.

Con la combinación de estos factores de juego, el jugador puede plantear diferentes estrategias a llevar a cabo en un nivel y lo ideal es que Towot complemente dicha estrategia. De esta forma, Towot puede optar por ejecutar diferentes estrategias dependiendo de lo que pretende hacer el jugador en el nivel.

Por ejemplo: puede mantenerse cerca de la base para protegerla mientras el jugador se mueve por el escenario. Puede moverse por las torretas extractoras (o refinerías) para protegerlas mientras el jugador protege la base, puede acudir a donde vaya el jugador como fuego de apoyo o simplemente dejar que el Towot decida en cada momento que es lo más prioritario que debe defender y que lo haga de forma autónoma.

Todas estas estrategias y algunas otras más son las que el diseñador puede crear usando programación por demostración con nuestra herramienta.

En cuanto a los atributos que definen el entorno del NPC en este dominio, la siguiente lista muestra los que se han utilizado para los experimentos y que han sido reducidos a 8, ya que son los que hemos considerado como relevantes a la hora de crear el comportamiento del Towot. Estos atributos, que posteriormente serán almacenados en la base de casos, forman parte de la percepción del NPC. Sus significados son los siguientes:

- TowotEnergy: es la energía del Towot. Sin ella no puede disparar, aunque si moverse para que el Towot pueda recargarse incluso sin energía. Esta energía se puede recargar en las torretas de recarga. El rango de valores que puede tomar es de [0,100].

- RefineryRiskLevel: Es el número de enemigos que rodea a la refinería. Su rango va de 0 a 4, siendo 4 si está rodeado por 4 o más enemigos.

- CoreRiskLevel: Mismo parámetro que el anterior, pero esta vez en para el core.

- JacobRiskLevel: Mismo parámetro que el anterior, pero esta vez Jacob, el avatar del jugador.

- CoreLife: la vida del core, que es un valor comprendido entre [0,1000].

- RefineryLife: la vida de la refinería, que es un valor comprendido entre [0-500]

- JacobLife: la vida de Jacob, que es un valor comprendido entre [0-500]

- DistanceToCore: la distancia de Jacob al core. En algunos experimentos usamos esta medida para que el sistema pueda detectar si el jugador está atacando o defendiendo en función de su distancia al core. Sus valores están comprendidos entre [0-200]

En el siguiente apartado explicaremos los experimentos realizados con los usuarios para validar los *Trained Query Nodes* como método de generación de comportamiento por demostración en juegos y medir la satisfacción de los diseñadores en el uso de los mismos.

4.4. Experimentación realizada

En la validación de los *Trained Query Nodes*, se han realizado diferentes experimentos en los que nos hemos focalizado en unos aspectos más que en otros. Por ejemplo, para la validación del sistema en su conjunto, se usaron usuarios reales, donde pretendíamos validar si el sistema podía ser usado por diseñadores noveles que no tenían fuertes conocimientos previos. Para ello usamos como sujetos de la experimentación a alumnos del Máster de Desarrollo de Videojuegos y del grado de desarrollo de videojuegos de la Universidad Complutense de Madrid. En el momento de realizar el experimento, sólo estaba implementado como técnica de generación del modelo de comportamiento CBR con k-NN. Posteriormente se han realizado experimentaciones sobre la precisión de los diferentes métodos empleados en el TQN y para valorar las ventajas e inconvenientes de cada una de ellas. Estos experimentos se detallan en el capítulo 5.

En el experimento que detallamos en el siguiente apartado, veremos como los diseñadores crearon por demostración diferentes comportamientos y posteriormente comprobaron si los comportamientos enseñados fueron aprendidos correctamente por el NPC. Así mismo, se les consultó acerca de la herramienta y su utilidad como herramienta de apoyo a los diseñadores para crear comportamientos.

4.4.1. Experimentación para validación del entrenamiento

Para esta experimentación, como ya hemos mencionado en el apartado 4.3, se ha usado como domino de pruebas Towot. El NPC a entrenar fue el robot que acompaña al jugador en dicho juego, denominado también Towot. La experimentación fue llevada a cabo por treinta estudiantes de diseño del Máster de diseño de videojuegos de la Universidad Complutense de Madrid con diferentes perfiles académicos. Algunos de ellos provenía de estudios relacionados con la programación, otros sin embargo provenían de profesiones más artísticas o relacionadas con la comunicación. El experimento fue llevado a cabo en diferentes fases para que fuese más fácil de explicar a los usuarios.

Al principio del experimento, se realizó una presentación del juego, su estructura y la disposición de los elementos en el mapa. Se mostró una pequeña demo donde el Towot era controlado por una IA entrenada por nosotros mismos, pero que no implementaba ninguna de las estrategias que se les pedía a

los diseñadores en el posterior ejercicio, para evitar condicionar su forma de jugar. La configuración del experimento ha sido descrita en el apartado 4.3.

En los experimentos, los usuarios jugaron tres veces el juego, cada vez con una estrategia diferente. Con cada estrategia, los usuarios primero entrenaban al Towot y posteriormente volvían a jugar para validar si el comportamiento del Towot era o no era el adecuado en función de lo entrenado. Es decir, si el Towot había aprendido correctamente el comportamiento que se le enseñó. Cada sesión de entrenamiento y test duró unos 6 minutos por lo que el experimento total, junto con la explicación inicial duro aproximadamente 40 minutos.

La primera estrategia la denominamos *estrategia defensiva*, en la que el Towot protegía la base y recargaba su energía entre las diferentes oleadas, mientras tanto, el jugador atacaba a los enemigos cerca de las zonas donde estos aparecían.

Como en la primera y segunda oleada sólo aparecen en un punto concreto, normalmente el jugador podía acabar con todos los enemigos y si alguno se le escapaba hacia la base, el Towot se encargaría de eliminarlo. No sucede lo mismo en la tercera oleada, donde aparecen desde tres puntos diferentes. En esta tercera oleada, el jugador ve como es atacado tanto el core como la refinería, por lo que es imposible que el jugador contenga el ataque por sí solo, antes de que alguno de los enemigos alcance uno de los dos puntos a defender.

Por lo tanto, la estrategia aquí varía en función de lo que decida el jugador, pudiendo atacar uno de los generadores de enemigos, hasta que vea comprometida la refinería y tenga que defenderla; o puede esperar a que los enemigos lleguen a la refinería y defenderla allí. Por otro lado el Towot tendrá que defender el core. En principio es suficiente para vencer a los enemigos, pero si se complica la situación, el jugador deberá estar atento para acudir en su ayuda.

El segundo comportamiento a entrenar lo denominamos *comportamiento ofensivo*. En él, el Towot acompaña al jugador en la vanguardia, es decir, protege a Jacob en todo momento, mientras no se vea comprometido el core o la refinería. En caso de que alguno de ellos sea atacado con suficiente contundencia, si el jugador no puede defender alguna de las estructuras, debe ordenar al Towot que lo haga. En cualquier caso, entre oleadas, el Towot deberá recargar su energía. Este comportamiento es algo más complicado, por lo que se espera que el sistema sea más complejo de aprender.

El tercer entrenamiento es un entrenamiento libre, es decir, el jugador puede jugar con la estrategia que quiera. Simplemente debe comprobar que, en la fase de test, el comportamiento entrenado y el aprendido es semejante.

Como estamos entrenando comportamientos concretos dentro de los posibles comportamientos que puede tener el Towot, pedimos a los usuarios que, en la ejecución de validación, jueguen de una forma similar a como lo

hicieron en el modo de entrenamiento, así se podrán comparar mejor los resultados con el entrenamiento realizado. El resultado obviamente no será exactamente el mismo ya que, dentro de una misma estrategia, siempre existen diferentes situaciones y no se puede jugar exactamente igual, además, como explicamos en el apartado 4.3 el juego no es determinista.

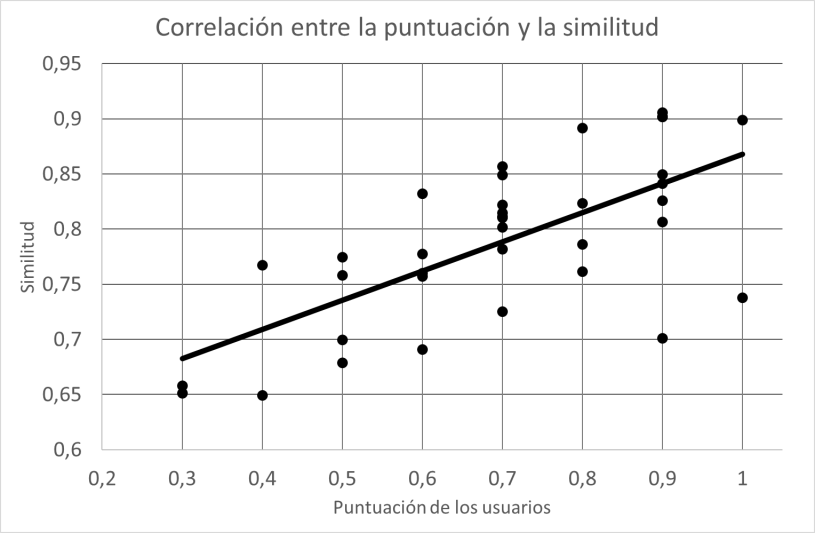


Figura 4.6: Recta de regresión entre la puntuación de los usuario y la distancia de edición entre la fase de entrenamiento y la de validación

El algoritmo de aprendizaje utilizado para construir el modelo del NPC en el experimento fue la aproximación basada en CBR con k-NN como algoritmo de aprendizaje. Los atributos que se almacenaron como contexto en el experimento son los descritos en el apartado 4.3 excepto *DistanceToCore* que fue introducido en experimentos posteriores. La configuración de k-NN fue fijada de antemano en base a experimentos realizados previamente, donde detectamos que un valor de $k = 10$ ofrecía, en general, buenos modelos en las estrategias que pretendíamos aprender.

El criterio de selección entre los k casos más similares que se utilizó para el experimento fue la tarea mayoritaria, es decir, de entre los diez casos más parecidos, se cuenta la tarea que más veces se repite en esos diez casos más similares y esa es la tarea que se seleccionará el TQN. En caso de empate, es decir que haya dos o más tareas con el mismo número de apariciones, la tarea seleccionada es la que tuviera una similitud media más alta. En la fase de entrenamiento se tomaron muestras del entorno y la acción que se estaba ejecutando cada 0.1 segundos.

Con la misma frecuencia sí hizo en la fase de validación para después comparar los resultados. Estas últimas muestras no están involucradas en el aprendizaje, sólo sirven para comparar el resultado de lo ejecutado en

la validación del modelo, con el entrenamiento realizado por el diseñador. La medida de similitud utilizada por el algoritmo para seleccionar los K casos más similares fue la distancia euclídea ponderada como se describe en el apartado 4.2.1. Los pesos de cada atributo fueron seleccionados en base a diferentes pruebas realizadas previamente al experimento. Esto se hizo así para simplificar el experimento, debido a que era bastante largo y no era recomendable intentar re-entrenar cambiando los pesos, para no hacerlo demasiado tedioso. Los pesos elegidos utilizados se muestran en la Tabla 4.1

Atributos	Rango	Peso
TowotEnergy	[0, 100]	2
RefineryRiskLevel	[0, 4]	1
CoreRiskLevel	[0, 4]	1
JacobRiskLevel	[0, 4]	1
CoreLife	[0, 1000]	0.25
RefineryLife	[0, 500]	0.25
JacobLife	[0, 500]	0.25

Tabla 4.1: Atributos almacenados en la fase de entrenamiento y el peso de los mismos en el cálculo de la similitud

Como se puede observar en la tabla, la energía tiene un peso mayor que el resto de los atributos. Esto es debido a que en las pruebas previas detectamos que este atributo tenía mucha importancia para que el comportamiento aprendido fuese efectivo, ya que si no se recargaba a tiempo el Towot éste no puede disparar, con lo cual, todo el comportamiento se veía afectado. También se puede observar como la vida del core, Jacob y la refinería tienen un peso menor, ya que es un valor que fluctúa mucho en función de cómo juegue cada jugador (si ha conseguido frenar a los enemigos antes de que lleguen a los objetivos) y que un jugador juegue mejor o peor no debería implicar un cambio en el comportamiento del Towot, a menos a corto plazo.

Sin embargo, que la vida del core o de la torreta sea baja, sí que puede ser interesante para el comportamiento, ya que podría decantar a la IA hacia qué es más prioritario defender en caso de duda. En cualquier caso, no se eliminaron para no simplificar demasiado el aprendizaje, ya que asumir que los diseñadores serían capaces inicialmente de saber seleccionar correctamente los atributos, era algo que no podíamos presuponer y por tanto eliminar estos atributos implicaría ayudar de forma artificial al algoritmo.

Es razonable pensar que un diseñador elegiría la vida de los tres objetivos, como un parámetro importante a tener en cuenta en el entrenamiento del Towot, en el marco de los comportamiento que queremos aprender, donde estamos intentando defender los diferentes objetivos. Para concluir, todos los atributos se normalizaron a valores comprendidos entre 0 y 1, usando sus rangos de representación mínima y máxima para que la magnitud de los

mismos no afecte a la similitud.

En este experimento queríamos validar la utilidad de la programación por demostración para los diseñadores, es decir, si realmente ésta permitía crear comportamientos de calidad y reproducibles. También queríamos medir si el diseñador era capaz de conseguir un comportamiento razonable, simplemente haciendo un único entrenamiento.

Nuestra hipótesis de partida era que, en comportamientos sencillos, la programación por demostración, incluso sin la posibilidad de re-entrenar ni modificar los pesos de los parámetros, era suficientemente eficaz como para que el diseñador pudiera crear comportamientos de forma autónoma, sin preocuparse de los detalles de implementación, simplemente haciendo un entrenamiento por demostración. Pero que cuando los comportamientos se volvían más complejos, la programación por demostración no era suficiente y podía producir comportamientos más erráticos que provocaría el rechazo del jugador.

Para medir la precisión del comportamiento aprendido se usa una doble validación. Por un lado, se pidió una validación subjetiva del propio diseñador, dando al final de cada fase de entrenamiento y validación una puntuación para indicar la calidad del comportamiento aprendido. Por otro lado, se miden las trazas de las tareas ejecutadas por el Towot, tanto en el modo de entrenamiento como en el modo de pruebas, para compararlas de una forma más objetiva. Las trazas guardan los valores de los atributos relevantes y la acción realizada.

La medida de similitud utilizada para comparar ambas trazas es la distancia de edición (o también llamada distancia de Levenstein (Deza y Deza, 2009)) y la distancia euclídea entre las muestras tomadas en ambas fases.

La distancia de edición mide el número de modificaciones (inserciones y borrados) que hay que hacer sobre una secuencia de símbolos, para convertirla en otra secuencia de símbolos diferente. En nuestro sistema, esa secuencia de símbolos está formada por las acciones realizada por el Towot, recopiladas en forma de vector. De esta forma, comparamos el vector que se genera con los datos de entrenamiento con el vector generado con las trazas de ejecución, midiendo la similitud entre ambas. Las muestras para formar el vector se tomaron cada 100 milisegundos, con lo que en media había aproximadamente 1500 a 2000 acciones en el vector, dependiendo de la duración.

De las dos medidas que se evaluaron, la que mejores resultados obtenía en función de las valoraciones de los usuarios era la distancia de edición, debido a que ésta tiene en cuenta el orden de las tareas realizadas y en nuestro caso, las tareas que el Towot ejecuta en ambas trazas están secuenciadas en función de las oleadas de enemigos configuradas para el experimento. Por lo tanto, la distancia de edición representa mejor la similitud que la distancia euclídea en este caso. Para validar que la medida tenía sentido, medimos la correlación existente entre las valoraciones de los usuarios y la similitud entre

las trazas de entrenamiento y validación. En la Figura 4.6 se puede ver la recta de regresión entre ambas medidas, que indica que efectivamente existe una correlación entre ambas magnitudes.

Si aplicamos el coeficiente de correlación de *Pearson*, podemos observar una correlación entre ambas magnitudes, con un valor obtenido de **0.68**, (superior a 0.5), de lo cual, podemos deducir que la distancia de edición nos puede servir como una medida de valoración similar a la valoración subjetiva de los usuarios, pero que podemos calcular nosotros de una forma más objetiva, para valorar si el aprendizaje del comportamiento entrenado ha sido el correcto. Obviamente la valoración subjetiva es importante ya que es la percepción del usuario la que nos interesa, así pues en los resultados de este experimento mostraremos ambas.

Con todo esto en mente, queremos demostrar con este experimento que integrar la programación por demostración dentro de los BTs tiene sentido, ya que ambos se complementarían. Algunos comportamientos pueden ser aprendidos por demostración, sin embargo, otros más complejos es más difícil que se consigan obtener por demostración, sin que el algoritmo cometa errores que lo alejen del comportamiento esperado por el diseñador. En esos casos, el entrenamiento por demostración sirve más bien como una forma de ejemplificar al programador como debe comportarse el NPC o incluso para incidir en los errores que éste comete, para que el programador lo solucione, más que en utilizarlo directamente en el juego final.

Hay que recordar en todo momento el fin último de utilizar programación por demostración en nuestro trabajo, que no es otro que ayudar al diseñador a ser más autónomo a la hora de crear comportamientos y de que ésta sirva de ayuda en la coordinación y cooperación entre ambos roles.

Como se puede ver en los resultados obtenidos, que se condensan en la Tabla 4.2, sin re-entrenar ni ajustar parámetros para conseguir afinar estos entrenamientos, las estrategias 1 y 2, las más sencillas, han obtenido unos resultados del 83 % de similitud entre el entrenamiento y la validación, usando *la distancia de edición* y una valoración de 3.75 sobre 5 en la primera estrategia y del 71 % y una valoración de 3.6 en la segunda estrategia. En la tercera estrategia, sin embargo, los resultados han sido sustancialmente peores.

	defensiva	ofensiva	libre
Similitud	0.832	0.719	0.624
Puntuación (1-5)	3.75	3.6	2.9

Tabla 4.2: Media de los resultados de las tres estrategias: Ofensiva, defensiva y libre

La distribución de los resultados usando la distancia de edición como re-

ferencia, para cada estrategia, se puede observar en la Figura 4.7, donde se aprecia claramente que la dispersión de los datos es mayor en la segunda y tercera estrategia. Como era de esperar, estas estrategias son más complicadas que la primera y por tanto la dispersión es mayor. Podemos ver también si nos fijamos en los datos atípicos, que aunque entrenar parece más sencillo que programar, hay entrenamientos realizados que producen muy bajos resultados y que son debidos en parte a que los diseñadores no han conseguido entrenar de forma correcta al sistema de aprendizaje.

Figura 4.7: Diagrama resumen de la distribución de los resultados del experimento usando la distancia de edición

En la segunda estrategia, a pesar de que el resultado medio es satisfactorio, hay casos donde el resultado del entrenamiento fue muy malo, véanse los casos atípicos que se muestran en el diagrama con una similitud de menos del 50 %, obteniendo, por el contrario, otros casos con una similitud cercana al 95 %. Esto tiene mucho que ver con la forma en la que se ha entrenado. Se necesitan buenos ejemplos que cubran bien el espacio de búsqueda para que el algoritmo pueda recuperarlos y deducir la acción a ejecutar.

Una conclusión que podemos sacar, por tanto, de este experimento es que el diseñador puede aprender con el tiempo a mejorar los resultados obtenidos, entrenando con mejores ejemplos al NPC para conseguir los resultados que se buscan, ya que aunque es más sencillo e intuitivo que generar un BT con un editor, si se entiende como funcionan las técnicas de modelado de comportamiento utilizadas, se pueden obtener mejores resultados, siendo cuidadosos con la forma de enseñar al sistema. En los casos en los que el entrenamiento ha fallado, lo más recomendable hubiera sido re-entrenar el comportamiento para intentar perfilarlo y conseguir mejores resultados. Al utilizar una aproximación basada en razonamiento basado en casos, una

forma de mejorar el resultado sería añadir nuevos ejemplos a la base de casos que cubran los huecos que el entrenamiento inicial no supo cubrir en el espacio de búsqueda.

En la valoración de los usuarios, cuyo resumen se muestra en la Figura 4.8, podemos sacar conclusiones similares. Podemos ver como la distribución 2 y 3 son similares, con la media algo más baja en el caso de la segunda y con mas casos atípicos en la parte superior de la distribución que en la inferior. En general se puede ver como la valoración de los usuarios es más estricta en la segunda estrategia que la similitud. Esto lo podemos achacar a una de las principales quejas de los usuarios del experimento cuando se les preguntó a cerca del uso de la herramienta. Muchos dijeron que el Towot no se recargaba en el momento adecuado, y por tanto, su comportamiento se volvía muy ineficiente, aunque el comportamiento general fuese bueno. Es decir, un pequeño error hacía que el Towot no se comportase correctamente a pesar de que la similitud de las acciones fuese muy similar. A pesar de estas pequeñas diferencias, los resultados usando ambas medidas son muy similares.

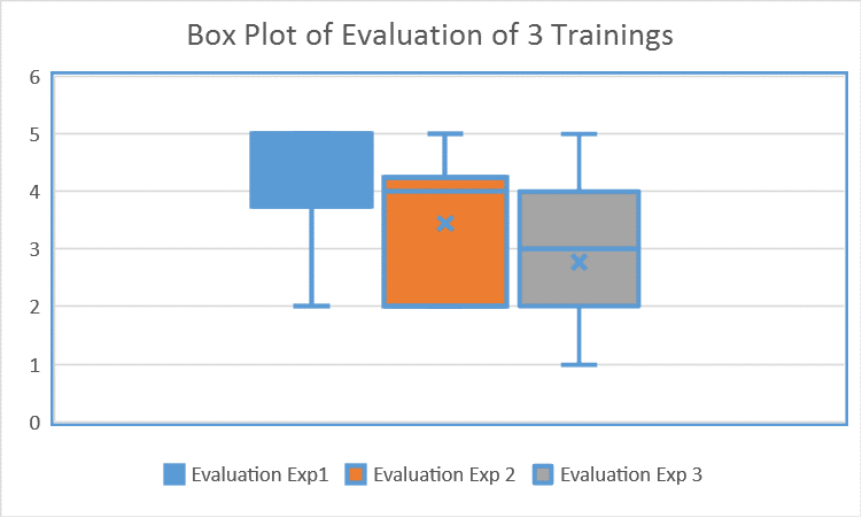


Figura 4.8: Diagrama resumen de la distribución de los resultados del experimento con la valoración de los usuarios

Para ejemplificar un poco esta ultima situación, imaginemos que, en el momento que el usuario recargó el Towot, este disponía de 50 unidades de energía aproximadamente. Existen por tanto, una serie de trazas con diferentes valores del resto de atributos, pero con unos valores en torno a 50 unidades de energía (supongamos de 50 a 40) en las que el diseñador ordenó recargar al Tawot.

En la fase de pruebas, sin embargo el Towot no decidió recargarse cuando la energía estaba en torno a 40 o 50unidades, porque encontró un caso con

mayor similitud que le decía, por ejemplo, que defendiera el core, ya que éste estaba siendo atacado. En ese momento, el algoritmo pensó que era más importante defender el core, algo que seguramente el propio jugador le enseñó en algún ejemplo e incluso es algo razonable que un jugador humano quizás hubiera decidido hacer, ya que el Towot aún podía seguir combatiendo un rato más sin recargar. Sin embargo, después de defender al core, el jugador humano habría mandado a recargar el Towot, para evitar que se quedase sin energía. Incluso, aunque no se diera cuenta de que el Towot estaba sin energía, en algún momento del juego lo haría y lo mandaría sin demora. Sin embargo, en nuestro caso, si el comportamiento no tiene ningún caso en el que el jugador recargara energía por debajo de 40, k-NN es muy probable que nunca mande a recargar al Towot cuando su energía esté sustancialmente por debajo de esa cifra, y por consiguiente, el comportamiento del robot se echará a perder en su conjunto.

La solución obvia en este caso sería añadir más casos a la base de casos, con situaciones en las que el Towot se recargue con energía por debajo de 40 o incluso forzar a que el Towot se quede sin energía y aportarle casos de recarga con energía a 0. Obviamente, llegar a esa conclusión por parte del diseñador no es algo trivial, se necesitaría cierto aprendizaje previo o conocer como funciona k-NN, algo que no podemos asumir. Lo ideal es que el diseñador jugase sin preocuparse de estas cosas y el algoritmo fuese lo suficientemente inteligente como para sacar conclusiones, pero tener en cuenta esto, ayudará mucho al algoritmo a tener mejores resultados.

Por otra parte, buena parte del feedback conseguido en este experimento nos instaba a tener alguna certeza de qué modelo había aprendido el NPC. Algunos diseñadores, sobre todo los más técnicos, sentían la necesidad de saber si el algoritmo que había aprendido, tenía cierto sentido o que incluso pudiese editarlo para mejorarlo.

Es decir, el uso de CBR no aporta información al usuario de cómo se está construyendo el modelo y por tanto puede generar rechazo a algunos diseñadores. Por este motivo se decidió incluir los árboles de decisión que describiremos en el apartado 5.3.1. Éstos son capaces de generar modelos que son legibles por los usuarios, ya que pueden ser fácilmente convertidos en reglas y estas reglas pueden describir nodos priority o parallel con guardas que se pueden generar automáticamente, dando información al diseñador de cómo está construido el modelo, manteniendo la estructura de los árboles de comportamiento que están habituados a utilizar.

Capítulo 5

Metodología integradora de Árboles de Comportamiento y Programación por Demostración

5.1. Introducción

A medida que los videojuegos han crecido de tamaño, su proceso de desarrollo y producción se ha vuelto mucho más complejo. Las primeras soluciones utilizadas en los videojuegos de los años 80 ya no tienen cabida en la industria actual, donde los equipos de desarrollo son multidisciplinares y están compuestos por decenas o incluso algunos cientos de trabajadores, se tardan años en desarrollar un juego e intervienen una gran cantidad de tecnologías. Incluso estudios especializados en un género de juego concreto y con herramientas y procedimientos ampliamente probados en multitud de juegos, en un desarrollo normal suelen tardar unos dos años.

Este hecho se puede ver por ejemplo en juegos como Call of Dutty®, que, aunque se produce un juego al año de esta franquicia, se llevan a cabo por dos estudios simultáneamente. Cada uno de los estudios generan un videojuego cada dos años y se van alternando para conseguir generar un videojuego anual; o la serie de videojuegos de conducción Forza®, donde también su periodicidad es aproximadamente de 2 años entre diferentes entregas, intercaladas por una versión del juego más arcade[1] (Forza Horizon®), realizada también por otro estudio diferente.

En otras producciones con más incertidumbre, es fácil encontrarnos con desarrollos de más de dos años. Los videojuegos son, por tanto, obras au-

[1] Un juego de conducción se considera arcade, cuando el foco de la jugabilidad no está en el realismo del comportamiento del vehículo si no en que el juego sea divertido.

diovisuales muy complejas y costosas, con muchas incertidumbres y pocas certezas cuando se comienza su desarrollo, que no se van despejando hasta que el desarrollo ya está en una fase avanzada del mismo. Por lo tanto, es importante tener un sistema de producción eficaz, que reduzca lo más posible los tiempo de desarrollo, que permita reutilizar la mayor cantidad posible de recursos, etc. Así pues, es muy típico reutilizar el motor en diferentes juegos, reutilizar animaciones o comportamientos.

El proceso de producción típico de un videojuego puede ser dividido en cuatro etapas (Chandler, 2009):

- **La preproducción:** en esta fase se deben sentar las bases del desarrollo. Es importante tener una idea clara del juego a realizar y de su envergadura, así como de definir el concepto de juego y sus requisitos correctamente. Esta fase suele llevar entre el 10 y el 25 % del tiempo total. El objetico de la preproducción es crear el plan de juego, una especie de *hoja de ruta* de los pasos a seguir para llevar a cabo el proyecto.

  Como resultado suele generarse, entre otras cosas, un prototipo del juego que es evaluado para ver si es divertido y tiene suficiente potencial como para continuar con el proyecto. En caso de que no sea así, es posible que el juego se cancele y no llegue a ver la luz, por lo que una buena preproducción ahorra muchísimos costes si se determina a tiempo la viabilidad o inviabilidad del proyecto.

- **La producción:** es la fase en la que el equipo produce los recursos (assets) y las características (features) finales del juego. La línea entre preproducción y producción es del todo clara. Podemos establecer esta línea a partir del primer prototipo aceptado. Es decir, cuando el equipo de desarrollo crea un prototipo que supera el visto bueno del equipo de evaluación, que dependiendo del estudio o del proyecto estará formado por unas personas o por otras. Ésta es normalmente la fase más larga, al menos tradicionalmente y en ella se cierra el concepto de juego totalmente, así como su implementación.

  Esta fase tiene como finalidad crear una versión de juego para ser testeada. Generalmente un juego en producción se encuentra en tres estados, que se suelen denominar con letras griegas. El estado *pre-alpha* es aquel en el que el juego aún no ha alcanzado el estado de Alpha. La versión Alpha la podemos definir como la primera versión jugable del juego en la que apenas falta contenido importante del mismo[2] y la versión *Beta* sería ya una versión muy cercana a la final, donde los usuarios pueden probar el juego completo y sólo faltan pequeños ajustes, errores y equilibrado. Muchas veces los juegos en Beta son lanzados

[2]NEXT Genereti on magazine, Vol 15, páginas 29–31, Imagine Media

para ser testeados por los propios usuarios finales, por lo tanto, el estado del juego debe ser prácticamente el estado final. Podemos decir que la producción finaliza con el juego en estado Beta.

- **Las pruebas:** esta fase es muy importante en un videojuego, debido a que en un desarrollo tan complejo es normal encontrar muchos errores de implementación. Pero no sólo se prueba si la implementación es correcta, sino también si el juego es divertido, si cumple con lo que el diseñador pretende, si está equilibrado o tiene una curva de dificultad ajustada, entre otras muchas cosas. La fase de pruebas normalmente abarca todo el proceso de desarrollo, pero se intensifica cuando los juegos alcanzan su estado de *Alpha* en adelante. El resultado es hacer que el juego se distribuya con el menor número de errores posible.
- **La postproducción:** Cada vez más importante en la industria ya que los modelos de juego actuales incentivan la construcción de juegos que se van modificando con el paso del tiempo, porque no son juegos cerrados si no juegos que van evolucionando y se van mejorando, adaptándose a las necesidades de los jugadores. Así se han conseguido que juegos como Starcraft®, Leage of Leyends® o World of Warcradft®, sigan años y años en activo con miles o incluso millones de jugadores. Además, los nuevos modelos de negocio como los *free to play* o los *Digital Downlader Contents* (DLCs) hacen que esta fase tenga cada vez más peso. En cualquier caso, esta fase queda fuera del alcance de este trabajo y, por tanto, no la consideraremos en el presente capítulo.

Estas fases que se aplican al desarrollo en un videojuego completo, son también aplicables a la creación de una parte del juego, por ejemplo a la creación un personaje concreto y por tanto de su comportamiento. Desde este punto de vista, podemos definir el ciclo de creación de un comportamiento como sigue: inicialmente se crea el concepto del personaje. Este concepto no es más que una idea de cómo debe ser, que aspecto tendrá y cómo se comportará. A continuación, en la fase de preproducción se genera un prototipo del personaje para evaluarlo. Si el prototipo es aceptado, se pasa al proceso de producción del mismo, para cerrar sus características hasta conseguir el personaje final que se añadirá al juego.

Actualmente estas fases, como veremos más adelante, no están tan delimitadas y están mucho más entremezcladas unas con otras, ya que el desarrollo no es secuencial como lo hemos descrito, si no iterativo.

Nuestra metodología pretende precisamente intentar reducir esas iteraciones, involucrando al diseñador en el proceso de desarrollo y no sólo en el proceso creativo o de supervisión. Dado este marco general, en la siguiente sección detallaremos cuál ha sido la evolución del proceso de creación de NPCs en la industria que sigue en paralelo la evolución del proceso de creación de un videojuego en sí mismo y su estado actual. En el apartado 5.3

describiremos la extensión de la herramienta realizada para incorporar árboles de decisión y redes de neuronas a los modelos de aprendizaje disponibles. Después, en el apartado 5.4, detallaremos nuestra propuesta de metodología y como se integra en el proceso de creación actual de un NPC y finalmente explicaremos con un ejemplo una aplicación práctica en nuestro juego de pruebas Towot.

5.2. Desarrollo de comportamientos en la industria

Tradicionalmente, el desarrollo de videojuegos ha sido un proceso secuencial y con él, la creación de los NPCs asociados al mismo. Cómo afirma Kent Hudson, Senior System Designer de 2K Marin en la GDC 2010 (Hudson's, 2010), y también corroboran otros autores como (Chandler, 2009) o en (Keith, 2010a), la forma tradicional de desarrollar videojuegos era usando un ciclo de desarrollo en cascada, en la que las fases de producción del juego van progresando una después de otra, hasta crear el juego final. Puede verse un esquema de este proceso en la Figura 5.1.

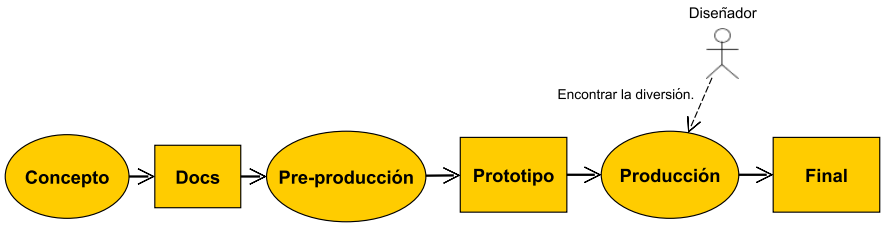


Figura 5.1: El ciclo tradicional de la industria

Como hemos dicho y sostiene Kent Hudson, este proceso es aplicable a cómo se crea el comportamiento de un NPC. Así pues, tradicionalmente el método para crear comportamientos era un proceso de creación en cascada, donde inicialmente el diseñador creaba el concepto del personaje. Este concepto no es más que una descripción del mismo, su aspecto, su estética, sus características (fuerza, peso, tamaño, defensa, ataque, etc), sus animaciones y su comportamiento. También una descripción, si fuese necesario, de en qué ambientes, zonas o situaciones este NPC debe aparecer. Entiéndase por NPC cualquier personaje no controlable por el jugador en un videojuego, desde un simple aldeano que deambula por un poblado haciendo sus quehaceres y con el que apenas podemos intercambiar unas palabras, como compañeros que pelean junto al jugador y que deben coordinarse con él para avanzar en el juego, o enemigos que intentan complicar el avance del jugador para que el juego le suponga un reto. Este concepto es presentado al resto del equipo de desarrollo y es tomado como base para crear un prototipo del mismo.

Este primer prototipo del personaje sirve para validar si éste tiene sentido

dentro del juego y si se continúa con su producción, o bien se re-formula o se elimina del juego final. Una vez superada esta fase y con el feedback del diseñador, el equipo de desarrollo termina el personaje y lo entrega al equipo de QA, que se encargará de testearlo y darle por finalizado. Este proceso es el proceso ideal, pero en la práctica esto no es así ya que QA normalmente desechará el comportamiento del personaje o su implementación en multitud de ocasiones, incluso pudiendo llegar a tener que re-hacer todo el personaje de nuevo.

Con el tiempo, los desarrolladores vieron como este ciclo clásico no era el más adecuado. Con este enfoque se desperdiciaba muchísimo trabajo y, cada vez más, intentaron acortar los tiempos en los que se supervisara el estado del proyecto, para poder corregir las desviaciones lo antes posible. De esta forma, se ha ido evolucionado paulatinamente a un modelo de producción de juegos en general y los NPCs en particular, mucho más iterativo y ágil.

Así, han surgido modelos de desarrollo ágiles como *scrum* que han sido adoptados muy rápidamente por la industria del videojuego (Keith, 2010a; Miller, 2008), cambiando a un estado de desarrollo basado en prototipos rápidos e incrementales y a una menor jerarquización del equipo de desarrollo, para que la toma de decisiones sea ágil y los equipos estén más cohesionados.

En este nuevo enfoque, como se está iterando constantemente sobre el prototipo, el diseñador siempre tendrá un juego con el que poder jugar y donde poder probar nuevas características. Viendo su evolución, los testers y los diseñadores pueden ver la evolución del juego y sus mecánicas muy rápidamente y en constante evolución. De esta forma, el diseñador tiene cuanto antes valiosa información con la que crear el juego y una retroalimentación muy rápida de sus peticiones, ya que puede probar rápidamente las ideas que van surgiendo y viendo sus posibles implicaciones.

Para que este tipo de ciclo de desarrollo funcione y tenga sentido, los prototipos han de ser rápidos y centrados en la funcionalidad e implementación de las mecánicas que se quieren conseguir. Se divide el proceso de desarrollo en una serie de hitos y se planifica que tareas se van a llevar a cabo para cada hito. En scrum, a estos hitos se les denomina *Sprints*.

Es importante comprobar primero las más complejas de llevar a cabo o aquellas de las que el diseñador tenga dudas de su funcionamiento, para así reducir lo más rápidamente posible la incertidumbre que estas generan y poder avanzar con él proyecto.

En la Figura 5.2 se puede ver un esquema de este proceso iterativo. Cómo se puede apreciar, el proceso comienza de una forma similar al ciclo tradicional en cascada. Se crea un concepto de juego y los documentos asociados al mismo para compartir la idea con el resto del equipo de desarrollo.

Con estos documentos iniciales, se crea un prototipo que permite algunas aproximaciones a nivel de arte del juego. Dicho prototipo puede ser considerado como el primer *entregable* de la preproducción. Separar la creación

del prototipo de las sucesivas iteraciones hasta conseguir el estado Alpha o Beta del juego en el esquema tiene sentido, ya que el proceso de creación del prototipo inicial es donde las principales características del juego deben ser resueltas, al menos a nivel de lo expresado en el concepto inicial, para poder saber si dicho concepto es divertido o transmite lo que los diseñadores pretenden.

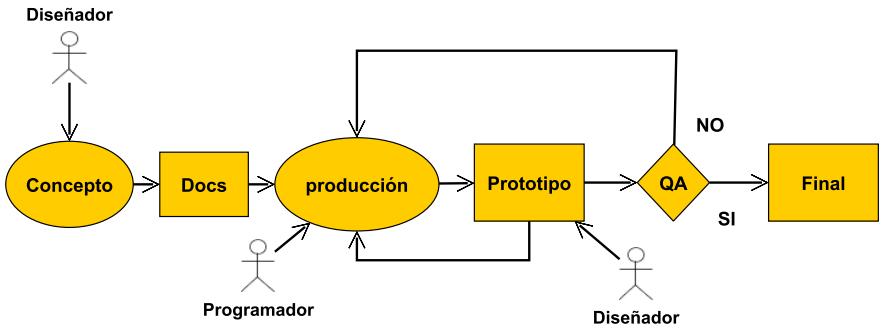


Figura 5.2: El ciclo iterativo de prototipado rápido utilizado actualmente

Así pues, el prototipo de comportamiento implicará un primer acercamiento al comportamiento del NPC, que evolucionará con el tiempo y que deberá ser revisado.

El proceso de revisión de este prototipo será doble ya que, una primera revisión la realiza el diseñador, ajustándose los parámetros que disponga el comportamiento, para que cumpla con sus especificaciones. Pero cuando el diseñador cree que el comportamiento está suficientemente pulido, se produce una segunda revisión de QA, que valida dicho comportamiento. QA puede propiciar cambios no sólo a nivel de programación, si no del propio diseño, por lo que el ciclo puede repartirse en varios niveles.

El objetivo de la metodología descrita en este capítulo, pretende minimizar las iteraciones entre el diseñador y el programador. La idea es que cuando se cree el prototipo del comportamiento, éste sea entregable para la fase de pruebas, con mínimas revisiones del diseñador, ya que éste ha estado involucrado en el propio proceso de desarrollo, creando la IA a alto nivel.

Cuando QA acepta el comportamiento, éste pasa a versión final y no se vuelve a modificar a no ser que se produzca algún cambio de diseño.

Otro de los aspectos que se puede apreciar en a Figura 5.2 y que ya hemos comentado anteriormente, es que en la creación de comportamientos, hay una clara distinción entre los roles de programadores y diseñadores. En otras áreas y cada vez más, los diseñadores pueden tener una parte relativamente importante en el proceso de desarrollo, gracias a las herramientas existentes. Sin embargo, en la creación de comportamientos para personajes, normalmente los diseñadores no entran en producción, como sí hacen a me-

nudo en el montaje del nivel o en el diseño de misiones, si no que supervisan y ajustan en el prototipo, en base a lo que los programadores han hecho y a lo que éstos les permitan modificar.

Normalmente los programadores dejan parámetros que los diseñadores puede modificar para ajustar estos comportamientos, pero estos no suelen crearlos directamente. Lo que conseguimos con nuestra metodología es que el diseñador no sólo interaccione con el prototipo, si no que sea parte activa del proceso de creación de dicho prototipo de comportamiento. Es decir, con nuestro cambio en el esquema de desarrollo, el diseñador cambie su rol de supervisor y de ajuste en el prototipo, por un rol mucho más activo en el desarrollo. Es decir, en nuestra aproximación, el diseñador pasa de ser evaluador del comportamiento creado por el programador a producir dicho comportamiento (véase Figura 5.3), gracias a que ahora cuenta con una herramienta para poder crear comportamientos de forma autónoma.

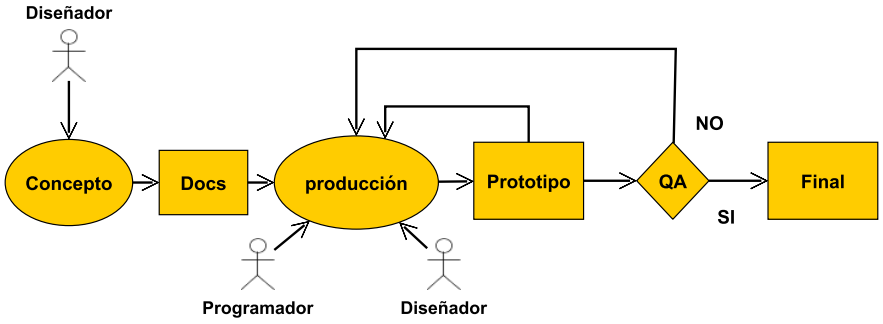


Figura 5.3: En nuestra aproximación el diseñador interviene en la producción.

5.3. Extensión de la herramienta para soportar múltiples modelos

Como vimos en la experimentación realizada en el apartado 4.4.1, hay algunos comportamientos que son más difíciles de conseguir, simplemente entrenando con demostración con nuestro modelo de comportamiento basado en CBR. Además, recibimos algunas peticiones sobre la necesidad de que el sistema mostrase de alguna forma qué había aprendido. Disponer de esta información genera más seguridad a los usuarios a la hora de usar esta tecnología, ya que pueden validar que lo aprendido sea realmente correcto. También pueden retocar el comportamiento, ajustándolo exactamente a lo que necesitaban.

El problema del CBR es que la interpretación de lo aprendido por el sistema por parte del diseñador, requiere que éste tenga conocimientos de

cómo funciona el algoritmo. Sin embargo, existen otros modelos más amigables para ser interpretados. Por ejemplo, los árboles de decisión, permiten mostrar el modelo en forma de reglas *Si...Entonces*, que son más accesibles para los diseñadores.

Así pues, modificamos la arquitectura para que el sistema pudiera ejecutar múltiples algoritmos de aprendizaje de forma fácil, simplemente creando una nueva clase que los implementase, permitiendo añadir entre otros los árboles de decisión. Un esquema de la arquitectura del TQN se muestra en al Figura 5.4. En ella, se puede ver como el nodo utiliza un modelo abstracto que puede ser implementando por diferentes algoritmos.

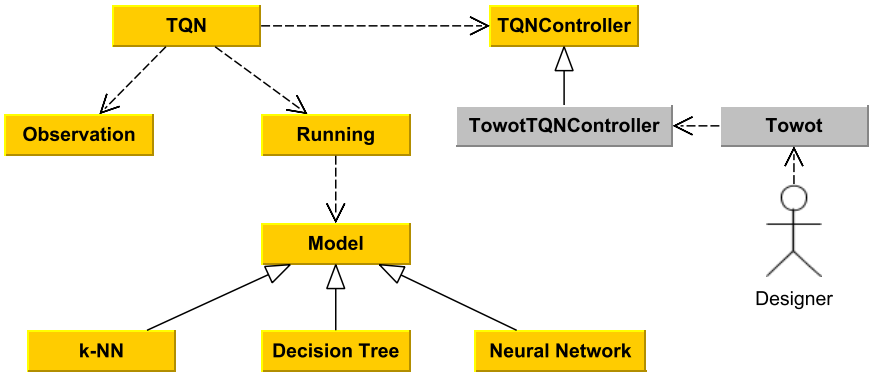


Figura 5.4: Esquema de la arquitectura del TQN que permite utilizar diferentes algoritmos para construir el modelo de ejecución

Algunos algoritmos, como los árboles de decisión o las redes de neuronas, necesitan un proceso de aprendizaje y procesado de los datos adicional a la captación de ejemplos por observación. Los datos obtenidos por observación serán usados en este proceso de aprendizaje para construir un modelo. Este modelo necesita ser guardado, sobre todo si el proceso de aprendizaje es costoso, para no tener que repetir el proceso de aprendizaje. En el caso de los árboles de decisión, como su algoritmo de aprendizaje es bastante rápido, se puede prescindir de almacenar el modelo, generándolo justo antes de ejecutar el nodo por primera vez, lo que simplifica los pasos que debe realizar el diseñador, siempre y cuando el numero de datos almacenados no sea muy grande y haga este proceso demasiado lento.

Aprovechando el cambio de arquitectura y la flexibilidad que nos proporcionaba, añadimos dos nuevas formas de generar el modelo de comportamiento: usando árboles de decisión y redes de neuronas. Las redes de neuronas no generan un modelo entendible por el diseñador, como si lo hacen los árboles de comportamiento. Se han incluido para tratar de tener diferentes modelos que permitan al diseñador tener diferentes opciones a la hora de crear los comportamientos por demostración. Como veremos en el experimento del

apartado 5.6, según la estrategia a aprender, unos modelos funcionan mejor que otros.

En los siguientes subapartados explicaremos cómo se crean estos modelos y veremos un experimento que muestra los resultados comparativos del mismo al final del presente capítulo, así como un ejemplo donde se puede ver las ventajas de nuestro enfoque híbrido con árboles de comportamiento. En este punto, vamos a describir completamente el estado final de la herramienta, antes de explicar nuestra metodología, para que el lector tenga la idea global de sus posibilidades.

El experimento final intenta demostrar que nuestro enfoque híbrido es muy útil para asegurar la confiabilidad del sistema y tiene sentido dentro del proceso de desarrollo. Veremos como un entrenamiento poco satisfactorio ha sido re-entrenado desde otro punto de vista, mezclando dos comportamientos más sencillos usando nuestro enfoque.

5.3.1. Modelado del comportamiento con árboles de decisión

La primera de las nuevas técnicas incluidas para generar el modelo de comportamiento fueron los árboles de decisión (Rokach y Maimon, 2014).

Ésta técnica genera un modelo de predicción que ha sido utilizado en diversas áreas, desde las ciencias de la computación a la economía. Estos árboles son muy útiles para visualizar las diversas opciones de las que se dispone para resolver un problema o modelar un comportamiento y cuál es la secuencia de pasos necesaria para llegar a dicha decisión, ya que se pueden convertir en reglas, que son mucho más legibles para el que investiga el funcionamiento del modelo. Además, se les pueden aplicar métodos de inducción, como por ejemplo la inducción hacia atrás, gracias a los cuales mediante sencillos razonamientos, se puede conseguir el razonamiento que el sistema ha tomado para llegar a seleccionar una acción.

Un árbol de decisión está compuesto de dos tipos de nodo, un nodo condicional y una clase. Los nodos condicionales son los nodos internos del árbol y la clase, los nodos hoja o terminales.

Cada nodo condicional es una pregunta que se hace a las variables del entorno y sólo tiene dos caminos, que la condición sea cierta o falsa. En función de si es una y otra, se tomará un camino y se llegará a un nodo hoja u otro. Cuando se llega a un nodo hoja, este determina la clase a la que pertenece el ejemplo suministrado. La clase, en nuestra aproximación, es la tarea a realizar.

Como ejemplo, en la Figura 5.5 podemos ver el árbol de decisión generado en uno de nuestros experimentos, donde se puede ver cómo la energía del Towot es el parámetro que se coloca en la raíz y en función de su valor o se recarga el Towot o se realiza otra de las acciones.

Su utilización en nuestro sistema viene dada precisamente de esta últi-

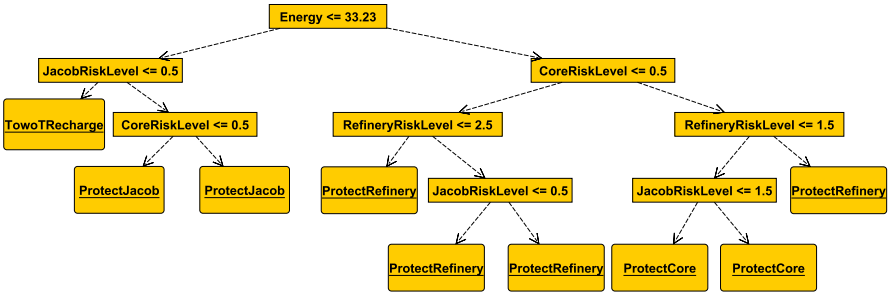


Figura 5.5: Arbol de decisión generado para uno de nuestros experimentos

ma característica, los árboles de decisión generan un modelo que puede ser entendido por el diseñador, de forma que además de modelar el comportamiento del NPC, le aporta a éste mucha información acerca de qué reglas ha aprendido en la fase de observación o entrenamiento. Esta información, como veremos en el apartado 5.3.2 además nos sirve para poder construir árboles de comportamiento con la información obtenida del árbol de decisión, por lo que el diseñador puede ver el comportamiento aprendido en el propio lenguaje de los BTs y modificar el árbol, ajustándolo manualmente si fuese necesario.

Este mecanismo proporciona una herramienta muy poderosa al diseñador para crear comportamientos de forma autónoma. El algoritmo concreto que hemos elegido para construir el árbol de decisión ha sido C4.5 (Quinlan, 2014). Este algoritmo utiliza la entropía (medida de incertidumbre o de desorden) para ayudar a decidir qué atributo debe ser el siguiente en ser evaluado en el árbol. También nos puede ayudar a descubrir que atributos son los más relevantes, ya que el algoritmo los colocará más cerca de la raíz. En concreto se pretende maximizar la ganancia de información, donde la entropía se utiliza como una medida del orden de los datos. Es decir, el atributo seleccionado es aquel que deja la información más ordenada o dicho de otra forma, mejor clasificada. La entropía se calcula con la ecuación 5.2

$$Entropia(s) = \sum_{n=1}^{c} -p_i log_2 p_i \tag{5.1}$$

donde c son los posibles valores de clasificación, S es el conjunto de todos los ejemplos y p_i es la proporción de ejemplos de S que están en la clase i. Obsérvese que se usa el logaritmo en base 2 porque la entropía es una medida de la longitud de codificación esperada medida en bits.

La ganancia de información se calcula según la ecuación 5.2

$$Ganancia(S, A) = Entropia(S) - \sum_{v \in Valores(A)} \frac{|S_v|}{|S|} Entropia(S_v) \tag{5.2}$$

donde $Valores(A)$ es el conjunto de todos los valores posibles para el atributo A, y S_v, es el subconjunto de S para el cual el atributo A tiene el valor v.

En general, los atributos que separan mejor las clases, tienden a reducir más la entropía, y por tal motivo, debe ser seleccionado primero. Nótese que la entropía se utiliza como heurística en una búsqueda de primero en profundidad, que realiza el algoritmo, por lo que el proceso de aprendizaje es muy rápido y no requiere de la intervención del diseñador. Por lo tanto, esta técnica es poco intrusiva para el usuario y el proceso de aprendizaje adicional a la fase de captación de ejemplos es transparente.

5.3.2. Generación de árboles de comportamiento

Las validaciones realizadas con usuarios de nuestro sistema, inicialmente se realizaron usando simplemente CBR como método de selección de la tarea en la fase de ejecución. Se seleccionó primero, debido a que no requería de una fase de entrenamiento previo (*offline*) más allá de la fase de captación de información, así como su versatilidad a la hora de poder mejorar su aprendizaje, simplemente añadiendo nuevos casos a la base de casos. Sin embargo, una de las cuestiones que los usuarios formularon fue que el sistema era una caja negra. Es decir, era un sistema que no aportaba información de cómo estaba tomando las decisiones.

Ésto a algunos diseñadores le producía recelo, ya que tenían la sensación de que no tenían el control de lo aprendido. Entonces surgió la idea de usar árboles de decisión por ser una técnica que es capaz de mostrar la información del modelo que infiere de una forma más entendible que otras técnicas que se evaluaron. Sobre todo si la convertimos en reglas.

Por las características de los árboles de decisión, las reglas que se pueden inferir de ellos son hyperplanos que particionan todo el espacio de posibilidades del algoritmo en conjuntos disjuntos. De esta forma, podemos garantizar que las reglas que se generan son también disjuntas. Por lo cual, se puede construir un árbol de comportamiento, donde cada una de estas reglas sean una condición protegida por una guarda sobre la acción a realizar, generando un árbol mucho más legible para los diseñadores y programadores.

Por lo tanto, para generar el árbol de comportamiento, primero se calcula el árbol de decisión sobre los datos de entrenamiento y posteriormente lo convertimos en reglas, usando un recorrido de *primero en profundidad*. Habrá una regla por cada hoja. La condición de la regla se forma concatenando con operadores AND, todos los nodos intermedios del camino que va desde la raíz hasta la hoja que determina la acción que dispara la regla.

Como ejemplo podemos ver este conjunto de reglas extraído de uno de los experimentos realizados, el gráfico del árbol lo mostramos previamente en la Figura 5.5:

```
1. Recharge= (ENERGY<=33.2) && (JACOB_RISK<=0.5)
2. ProtectJacob= (ENERGY<=33.2) && (JACOB_RISK>0.5)
    && (CORE_RISK<=0.5)
3. ProtectJacob= (ENERGY<=33.2) && (JACOB_RISKL>0.5)
    && (CORE_RISK>0.5)
4. ProtectRef= (ENERGY > 33.2) && (CORE_RISK<=0.5)
    && (REFINERY_RISK<=2.5)
5. ProtectRef= (ENERGY > 33.2) && (CORE_RISK<=0.5)
    && (REFINERY_RISK>2.5) && (JACOB_RISK<=0.5)
6. ProtectRef= (ENERGY>33.2) && (CORE_RISK<=0.5)
    && (REFINERY_RISK>2.5) && (JACOB_RISK>0.5)
7. ProtectCore=(ENERGY>33.2) && (CORE_RISK>0.5)
    &&(REFINERY_RISK<=1.5) && (JACOB_RISK<=1.5)
8. ProtectCore= (ENERGY>33.2) && (CORE_RISK>0.5)
    && (REFINERY_RISK<=1.5) && (JACOB_RISK>1.5)
9. ProtectRef= (ENERGY>33.2) && (CORE_RISK>0.5)
    && (REFINERY_RISK>1.5)
```

Si nos fijamos detenidamente en las reglas, podemos ver cómo estas son redundantes, por ejemplo, la regla 2 y 3 se pueden condensar en una sola regla, porque el parámetro CORE_RISK produce la misma tarea en ambos lados de la condición.

El sistema es capaz de simplificar estas reglas automáticamente. Para ello se utiliza un algoritmo voraz, que busca un nodo que tenga dos hijos hoja con la misma acción, lo que generará una regla redundante. Para simplificarlo, se sustituye dicho nodo completo (la condición y las dos hojas) por la acción que aparece en las hojas. El proceso se va repitiendo sucesivamente, hasta que en una iteración no se modifica el árbol. Aplicado a reglas, el pseudocódigo del algoritmo se muestra en 5.1.

Algoritmo 5.1: Algoritmo de simplificación de reglas

```
Reglas[] R
reglassimplificadas = 0
Reglas[,] grupoReglas = ClasificarPorTarea(R)
foreach(Reglas[] r in grupoReglas)
{
 reglassimplificadas = 0
 do{
  for i=0 to r.Count-1
    for j=i+1 to r.Count{
    if(MismaReglaSalvoUltimoParm(r[i],r[j])){
     BorramosUltimoParm(r[i])
     r.BorramosRegla(j)
     reglassimplificadas++
     }
   }
 } while (reglassimplificadas>0);
}
```

Aplicando esta reducción, las reglas se simplifican como se muestran a continuación.

```
1. Recharge= (ENERGY<=33.2) && (JACOB_RISK<=0.5)
2. ProtectJacob= (ENERGY<=33.2) && (JACOB_RISK>0.5)
3. ProtectRef= (ENERGY>33.2) && (CORE_RISK<=0.5)
4. ProtectRef= (ENERGY>33.2) && (CORE_RISK>0.5)
        &&(REFINERY_RISK>1.5)
5. ProtectCore= (ENERGY>33.2) && (CORE_RISK>0.5)
        &&(REFINERY_RISK<=1.5)
```

Como se puede observar, el número de reglas y su complejidad es mucho menor y por tanto es mucho más legible.

Lo interesante para construir el árbol de comportamiento y que éste sea legible, es que tengamos una única regla por cada comportamiento. Así pues, si unimos las reglas que siguen ofreciendo la misma tarea con una condición OR podremos conseguir una regla por acción, como se muestra a continuación:

```
1. Recharge= (ENERGY<=33.2) && (JACOB_RISK<=0.5)
2. ProtectJacob= (ENERGY<=33.2) && (JACOB_RISK>0.5)
3. ProtectRef= ((ENERGY>33.2) && (CORE_RISK<=0.5))
  || ((ENERGY>33.2) && (CORE_RISK>0.5)
  && (REFINERY_RISK>1.5))
4. ProtectCore= (ENERGY>33.2) && (CORE_RISK>0.5)
  &&(REFINERY_RISK<=1.5)
```

Finalmente, podemos crear el comportamiento usando árboles de comportamiento mediante un nodo paralelo que ejecuta cada una de las tareas disponibles. Estas tareas están protegidas por un nodo guarda con la condición extraída de las reglas del árbol de decisión. Si la condición se cumple, se ejecutará la tarea y si no se cumple no se ejecutará. Como las condiciones son mutuamente exclusivas, nunca se ejecutan dos tareas simultáneamente.

El nodo paralelo puede ser sustituido por un nodo de selector de prioridad, si el nodo paralelo resultase confuso para el diseñador. Éste nodo lleva implícita una prioridad de ejecución de los hijos. Es decir, los hijos de la izquierda son más prioritarios que los de la derecha. Si un nodo más prioritario cumple la condición para ser ejecutado, la tarea actual se interrumpe y comienza la nueva. Esto es algo que, en este caso, no tiene ninguna relevancia debido a que no se pueden dar dos condiciones simultaneas como ciertas, ya que todas son mutuamente excluyentes. De no darse esta característica, usar el priority selector implicaría cambiar la semántica del comportamiento. Usar el selector normal no sería apropiado ya que si una acción entra en estado de ejecutandose, esta nunca puede ser interrumpida una vez evaluada la guarda, mientras que un selector de prioridad, reevalua la guarda constantemente.

En la Figura 5.6 se muestra el árbol generado usando *Behavior Bricks*. El sistema exporta el árbol en el formato que *Behavior Bricks* puede entender y este queda disponible para ser usado en sustitución del Trainde Query Node del comportamiento que se estaba aprendiendo.

Existe una simplificación adicional que se puede realizar para generar reglas mas sencillas. Si establecemos un error tolerable para las reglas, po-

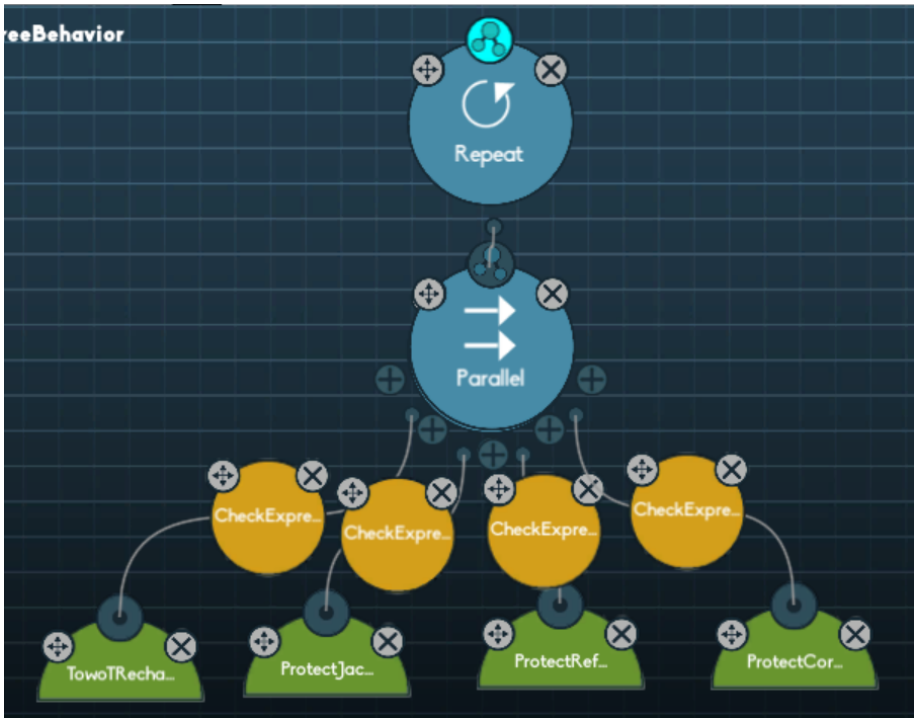


Figura 5.6: BT inferido

demos simplificarlas iterativamente hasta conseguir que la eliminación de un parámetro, implique que dicho margen error no se supera. De esta forma podemos simplificar aún más las reglas a costa, eso sí de reducir su precisión (Quinlan, 1987). En cualquier caso, si la reducción resulta beneficiosa para la legibilidad de las mismas, puede aplicarse ya que en este caso, es más importante la información que aporta al diseñador que la propia precisión de las reglas, ya que el diseñador una vez tenga el árbol construido, puede modificar el árbol manualmente para ajustarlo a sus necesidades. El algoritmo se puede ver en el pseudocódigo 5.2:

Algoritmo 5.2: Algoritmo de simplificación de reglas con pérdida de precisión

```
Reglas[] R.
error = Error(R)
foreach(r in R)
 p = Precondicion(r);
 foreach (p in P)
  nuevoError = ErrorEliminar(r,p)
  if(|nuevoError - error| < delta)
   eliminar(p,r)
```

El resultado de esta simplificación variará en función del error permitido. Si el error que se permite es mayor las reglas quedarán más simplificadas, pero perderemos precisión al ejecutar el árbol.

5.3.3. Modelado del comportamiento con redes de neuronas

Las redes de neuronas son una analogía del funcionamiento del cerebro humano, que permiten a las computadoras adoptar ciertas capacidades que se pueden definir como inteligentes y que resuelven un importante subconjunto de problemas que otras técnicas computacionales o matemáticas no resuelven o lo hacen de forma más ineficiente, tanto en tiempo como en los resultados obtenidos.

Debido a su semejanza con el cerebro, posee muchas capacidades similares a la mente humana. Por ejemplo, son capaces de aprender de la experiencia, de generalizar de casos anteriores a nuevos casos, de abstraer características esenciales a partir de entradas que representan información irrelevante, etc. Esto hace que ofrezcan numerosas ventajas y que este tipo de tecnología se esté aplicando en múltiples áreas, como aprendizaje adaptativo, auto-organización, tolerancia a fallos, etc. (Matich, 2001).

Las redes de neuronas se basan en los trabajos realizados por Warren McCulloch y Walter Pittis en 1943 que han servido de base para la aparición de la multitud de arquitecturas obtenidas durante los últimos setenta años.

Las redes de neuronas se caracterizan por contar con un número de neuronas organizadas de diversas formas, que están conectadas unas a otras a través de enlaces ponderados por unos pesos, que suelen ser recalculables por la propia red, de forma más o menos autónoma, para proporcionar una de las capacidades más importantes de la mayoría de las redes de neuronas, que es su capacidad de aprendizaje.

Al igual que el cerebro humano, las neuronas y las redes de neuronas artificiales sustentan su capacidad de cálculo no en el núcleo de las neuronas sino principalmente en las interconexiones que se realizan entre ellas. En las neuronas humanas existen unas zonas denominadas sinapsis, donde se realizan las conexiones entre dos neuronas. Esas conexiones tienen enlaces químicos de diferentes intensidades que se van modificando cuando aprendemos.

La neurona artificial no posee la complejidad electroquímica de la neurona biológica, ya que es una analogía matemática de su comportamiento, y simula estos enlaces con un peso. Dicho peso determina la intensidad del enlace entre dos neuronas.

La gran diferencia entre la computación neuronal y la computación clásica estriba en el hecho de que la primera no ejecuta un algoritmo predefinido para resolver un problema, sino que ejecuta un algoritmo genérico, que resuelve un número muy alto de problemas de características dispares. Con un paradigma convencional de programación clásica, el objetivo del programador es modelar el problema a tratar y formular una solución que no es más que un algoritmo que resuelva dicho problema. En contraposición, la aproximación basada en las redes de neuronas parte de un conjunto de datos de entrada suficientemente significativo y el objetivo es conseguir que la red

aprenda automáticamente las características del problema y sea capaz de obtener una solución lo suficientemente aproximada, como para que el margen de error de la red en su solución sea despreciable.

Así pues, podemos definir una red neuronal como una colección de neuronas cuya salida está conectada a las entradas de otras neuronas formando un *grafo dirigido* G(V,A) donde los vértices (V) son las neuronas y las aristas (A) son las conexiones entre ellas. En dichas conexiones reside la inteligencia de la red. Estas conexiones llevan asociada un peso que simula la complejidad de la sinapsis y que es modificable por mecanismos de aprendizaje. Al conjunto de pesos de una red se le denomina *conocimiento de la red.*

Normalmente se suelen diferenciar tres tipos de neuronas en una red, dependiendo de su función:

- Neuronas de entrada: Son las encargadas de recibir los datos del exterior de la red e inician la propagación de dichos datos por toda la estructura. Las conexiones de entrada de éstas neuronas (sus dentritas), no realizan ningún cómputo, sólo el hecho de captar la información para la red.

- Neuronas ocultas: Son aquellas neuronas que permiten enlazar las neuronas de entrada y las neuronas de salida, aunque también pueden estar conectadas a otras neuronas ocultas; y en sus interconexiones es donde precisamente se realiza el proceso de cómputo, tanto al enviar la información a todas las neuronas con las que está interconectada, como al recibir la información.

- Neuronas de salida: Son las que muestran al exterior los resultados obtenidos después de que la entrada haya recorrido toda la red. Las conexiones de salida de las neuronas de salida y no realizan ningún cómputo.

Así, se puede definir el concepto de *capa* como una colección de neuronas que tienen una función común dentro de la red, que reciben exclusivamente como entradas las salidas de neuronas pertenecientes a una capa denominada anterior (o del exterior si nos referimos a la capa de entrada) y que propagan su salida hacia neuronas que pertenezcan a una misma capa denominada siguiente (o al exterior de la red si nos referimos a la capa de salida). (véase Figura 5.7)

Una de las redes más conocidas es el *Perceptrón multicapa*, que posee un número variable de capas ocultas, con conexiones hacia delante y usa el algoritmo de retropropagación (Williams y Hinton, 1986), para su aprendizaje.

Podemos definir por lo tanto de forma matemática una red de neuronas por capas como una ecuación vectorial (Ecuación 5.3)

$$\vec{S} = F(F(F(\vec{X} \cdot \vec{W}_1) \cdot W_2) \ldots \cdot W_n) \tag{5.3}$$

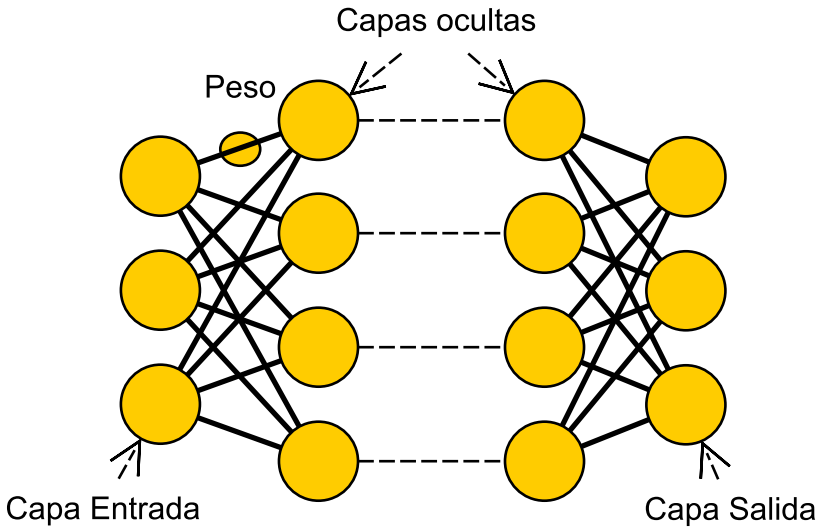


Figura 5.7: Esquema de una red neuronas

Donde W_i son los pesos de la capa i-esima, $\vec{X}$ es el vector de entrada y $\vec{S}$ el de salida y F es la función de activación.

Existen muchas funciones de activación en la literatura pero en nuestro sistema hemos optado por la sigmoidal (Ecuación 5.4)

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5.4}$$

El algoritmo de retropropagación se basa en calcular la diferencia existente entre la salida real de los datos de entrada y la salida producida por la red. El descenso del gradiente del error se propaga hacia tras por las distintas capas de la red, modificando sus conexiones para que la próxima vez que se vuelva a leer el dato, la red produzca una salida más próxima a la real.

Dicha derivada del error se va propagando desde las salidas a las entradas, modificando los pesos de las conexiones para ajustarlos al error con pequeños incrementos, el grado con el que se ajustan los pesos viene dado por una razón de aprendizaje (α) que determina la intensidad con la que se modifican los pesos. Valores bajos hacen que el entrenamiento sea más largo, pero evitan saltos y comportamientos erráticos de la red y de su error en el aprendizaje, aunque también es más difícil que el algoritmo explore nuevas soluciones si la función que representa la red entra en un mínimo local (Isasi Viñuela y Galván León, 2004). El proceso de aprendizaje precisa de una medida del error total de la red para un ciclo de entrenamiento.

Este mecanismo no es siempre tan ideal en la practica, porque se pretende

que la red adquiera capacidad para todos los datos que constituyen el universo del discurso del problema, y no sólo para los datos de entrenamiento, por lo que si se entrena demasiado se produce un fenómeno de *sobreadaptación* (Isasi Viñuela y Galván León, 2004). Si se produce sobreadaptación, la red de neuronas funciona muy bien con los datos de entrenamiento, pero pierde capacidad de generalización para ejemplos que no estén en el corpus de aprendizaje.

Se puede definir ciclo de entrenamiento como el número de veces que un Perceptron propaga y calcula el error de todas las entradas del conjunto de aprendizaje. En cada ciclo se evalúa el error cometido por la red en su conjunto. Este error debe ser mostrado al usuario que realiza el entrenamiento para que decida si la red ha aprendido correctamente. La función matemática que expresa el error de toda la red (E) es el error medio para uno de los patrones del conjunto de entrenamiento que la red está aprendiendo y se define en la ecuación 5.5

$$E = \frac{1}{n} \sum_{p=1}^{n} e(p) \tag{5.5}$$

donde $e(p)$ es el error producido por el patrón p en la red y n es el total de patrones del conjunto de entrenamiento.

Por lo tanto, para entrenar la red se necesita informar al usuario de qué error se está produciendo. Mientras el error siga reduciéndose, la red puede seguir entrenándose, sin embargo, si la red comienza a producir más error del deseable, hay que detener el algoritmo para evitar sobreadaptación.

Así pues, las redes de neuronas tienen un hándicap a la hora de ser utilizas por un diseñador en nuestra herramienta. No sólo debe configurar la red, sino que debe visualizar el error cometido y detener el entrenamiento cuando lo considere oportuno.

Para simplificar las decisiones que el diseñador debe tomar, hemos tomado algunas decisiones de diseño. Por ejemplo, la red sólo dispone de una capa oculta (técnicamente, añadir más capas ocultas no aporta mayor expresividad a la red de neuronas (Gardner y Dorling, 1998)) y las capas de salida y de entrada se configuran automáticamente. Se han probado diferentes codificaciones de la capa de salida y la que mejor resultado nos ha ofrecido es una codificación de una neurona por cada posible tarea de salida y la clase a la que pertenece la salida es de la neurona de salida con un mayor valor de activación. Todos los valores de la red y las entradas han sido normalizadas a valores comprendidos entre 0 y 1 y hay una entrada por cada atributo que se guarda de los ejemplos.

Así pues, el diseñador sólo debe establecer como parámetros el número de neuronas de la capa oculta, la razón de aprendizaje y realizar el entrenamiento, vigilando que el error no se incremente. El error se calcula con un conjunto de validación para ver si la red pierde o no capacidad de ge-

neralización, por lo tanto, también requiere de un segundo entrenamiento para obtener este conjunto de validación. Este segundo conjunto de entrenamiento se puede obtener de la misma forma que se obtuvo el entrenamiento inicial, ejecutando el *Trained Query Node* en modo observación y guardando una traza del mismo. De esta forma tendremos dos conjuntos diferentes de aprendizaje, uno para entrenar y otro para validar. En el apartado 5.6 se pueden ver los resultados obtenidos en nuestra experimentación con las redes de neuronas, en comparación con las anteriores técnicas descritas.

5.4. Metodología integradora de árboles de comportamiento y programación por demostración

Como se ha podido ver en el Capítulo 4, utilizando programación por demostración dentro de una herramienta de desarrollo de comportamientos como los árboles de comportamiento, los diseñadores pueden crear comportamientos complejos sin perder de vista la confiabilidad necesaria en un videojuego. Gracias al enfoque híbrido del sistema, el diseñador puede entrenar por demostración los comportamientos que ha ideado y comprobar in situ si el entrenamiento ha sido el correcto. Mientras entrena el comportamiento tiene retroalimentación directa de si el comportamiento ideado tiene sentido y funciona correctamente.

Gracias a este sistema, el diseñador tiene una potente herramienta para poder crear comportamientos sin necesidad de saber programar, por lo que le proporcionamos una mayor autonomía. Como hemos explicado anteriormente, los experimentos realizados con los usuarios mostraron que la programación por demostración en algunos comportamientos no tenían la suficiente precisión y algunos usuarios comentaron que les parecía muy útil poder saber qué estaba aprendiendo el NPC, lo que nos llevó a extender el modelo, primero con árboles de decisión y más tarde, aprovechando que el sistema fue adaptado para permitir múltiples modelos, con redes neuronales para estudiar si este modelo mejoraba los resultados obtenidos hasta el momento con árboles de decisión y k-NN.

Al añadir estas nuevas formas de crear el modelo, se hace necesario rediseñar la metodología presentada en el apartado 3.5, añadiendo el uso de los *Trained Query Nodes* y sus múltiples métodos de generación del modelo de aprendizaje, dentro del proceso de desarrollo de un videojuego. La metodología descrita previamente sigue plenamente vigente, es decir, seguimos recomendando a los diseñadores sólo crear comportamientos de alto nivel, mientras que los programadores se encargarán de crear aquellos comportamientos de bajo nivel que el diseñador necesite para construir la IA. Estos comportamientos se irán almacenando como una biblioteca, de forma que, si estos están suficientemente generalizados y bien construidos, cada vez será más sencillo crear nuevos comportamientos, reutilizando los ya existentes.

Partiendo de esta base metodológica inicial, queremos concentrar la intervención del diseñador, no en el prototipo como hasta ahora se solía hacer, si no en la propia creación del comportamiento en sí mismo. Hasta ahora, en la mayoría de estudios, el diseñador modificaba los parámetros que ofrecía el programador para conseguir ajustar el comportamiento, dentro de los márgenes que el programador le había permitido modificar. Sin embargo, nuestra intención es que sea el diseñador el que cree la IA de alto nivel, para que se involucre activamente en el proceso. Para ello, el proceso de producción debe ser diferente a como se hacía hasta ahora. Anteriormente, la producción consistía en crear o modificar los recursos necesarios para completar las tareas establecidas para el siguiente hito o *sprint*, según la nomenclatura de scrum.

Estas especificaciones surgen de los comentarios del propio diseñador y del feedback de los tester al probar el juego. Una vez estas nuevas características eran implementadas, los programadores dejaban una interfaz sencilla a los diseñadores para que, modificando ciertos parámetros en el prototipo, el diseñador pudiera ajustar el comportamiento a lo que él esperaba. Ahora, con nuestra aproximación, la interacción del diseñador se produce en el propio desarrollo del comportamiento y no sólo en el prototipo en sí. De forma que, el prototipo no necesita la fase de supervisión del diseñador ya que él ha formado parte de la creación del mismo y por tanto, los errores de diseño que éste contenga, ya son conocidos por él, y los que no conozca, serán revelados por el equipo de pruebas. Para ello, el proceso de hacer el juego divertido se centra, no en el prototipo como hasta ahora, si no en el proceso de entrenamiento. Mientras el diseñador entrena los comportamientos, los está en cierta manera validando no sólo a nivel de lo bien o mal que el modelo de aprendizaje haya modelado dicho comportamiento, si no de la utilidad o lo divertido que es el comportamiento en sí mismo.

Esta retroalimentación es clave para que el diseñador pueda darse cuenta, mientras crea los comportamientos, si estos eran los comportamientos correctos o no. De forma que el diseñador y el propio concepto del comportamiento se retroalimentan de dicho entrenamiento. Detallando la fase de producción, podemos ver con más el proceso de aprendizaje que será el centro de nuestra metodología que se muestra en la Figura 5.8.

Si el diseñador tiene claro cómo crear el comportamiento usando directamente los árboles de comportamiento, éste puede hacerlo usando el editor de *Behavior Bricks* sin necesidad de realizar ningún entrenamiento. En caso de que el diseñador no sea capaz de programar estos comportamientos usando el editor, podrá entrenarlos y validarlos de forma iterativa, hasta conseguir que dicho comportamiento sea correcto. Una vez el diseñador considere que el comportamiento generado es el correcto, se envía a QA para que lo valide definitivamente. QA puede encontrar errores y devolverlos al proceso de entrenamiento o aceptarlo y dar el comportamiento como final. De esta forma

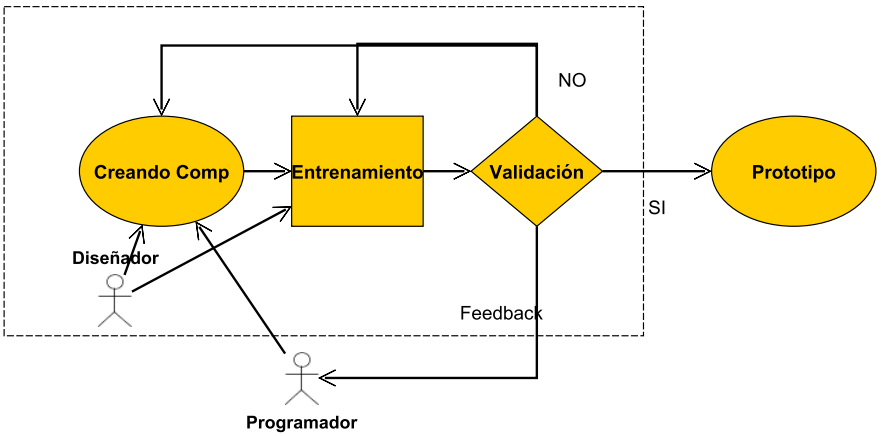


Figura 5.8: Esquema de construcción de comportamientos

reducimos las iteraciones que se producen entre programador diseñador y los testers.

El prototipo cuando es finalizado, ya ha sido supervisado por el diseñador y por tanto esa fase la eliminamos, moviendo al diseñador del prototipo al proceso de creación de la IA. Los cambios que se han ido produciendo y las iteraciones entre diseñador y programador, siguiendo esta aproximación, son mucho más pequeños e incrementales, ya que ambos trabajan codo con codo en la creación del comportamiento, eliminando el tiempo de *desconexión* en el que el programador desarrolla todo el comportamiento y, una vez ha terminado, lo entrega para validarlo al diseñador. Además, como parte del propio comportamiento lo implementa un diseñador, la parte en la que el diseñador cuenta al programador lo que quiere conseguir con el comportamiento se reduce, minimizando imprecisiones.

De esta colaboración, además, surgen una serie de sinergias que también permiten mejorar la comunicación entre ambos roles. Por ejemplo, si el diseñador detecta que alguna de las acciones básicas no funciona correctamente, informa al programador para que la corrija; si mediante el entrenamiento, surgen nuevos comportamientos básicos, estos pueden ser creados por el programador rápidamente, etc.

Para conseguir todo esto, hay que realizar algunos cambios en el proceso de creación del primer prototipo, ya que hay que construir también el entorno de entrenamiento. Este hecho implica que hay un incremento en el tiempo de desarrollo del prototipo inicial (pre-producción). También el diseñador debe aportar más información inicial en sus documentos de diseño. En la figura 5.9 se muestran algunos de estos documentos adicionales que deben surgir de la reunión con los programadores para crear el primer prototipo. De esta reunión deben salir los siguientes documentos:

- Un documento de requisitos del entrenador de comportamientos *Trainer Requirements*, donde se especifica las características que debe tener el entorno de pruebas.

- Una tabla de estrategias *StrategyTable* donde se debe especificar para los diferentes escenarios del juego, que tipo de estrategia o comportamiento debe tomar el NPC.

- El desglose de las tareas que deben realizar los programadores y los diseñadores *Tasks* y que debe consensuarse entre ambos roles. Las tareas de los programadores serán de bajo nivel o comportamientos especialmente sensibles, que no se desee que sean entrenados. Las tareas del diseñador por el contrario, se centrarán en entrenar comportamientos a alto nivel.

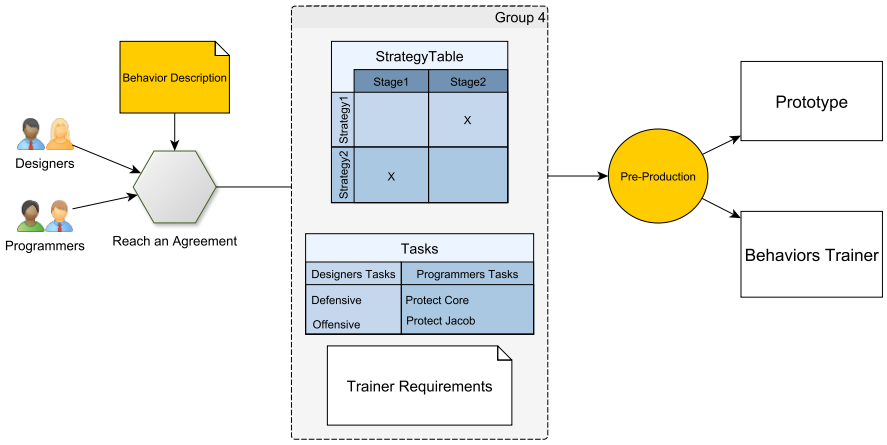


Figura 5.9: Proceso de creación del prototipo inicial

Así pues, el entorno de entrenamiento es una parte muy importante en esta metodología y debe ser creado con cuidado para que sea intuitivo de usar para el diseñador. El entorno debe cumplir las siguientes características:

- Debe poseer una interfaz donde el diseñador pueda seleccionar la acción que desee ejecutar. Dependiendo del juego, esta interfaz puede estar mapeada en los botones de un mando, en una interfaz gráfica de usuario dentro del juego o en el teclado. Dependiendo de la naturaleza del NPC y su comportamiento, tendrá más sentido una cosa o la otra. Si el NPC debe reaccionar con rapidez a diferentes situaciones, entonces se espera una respuesta rápida. Mapear las acciones que puede realizar a botones o teclas seria la aproximación más efectiva y el proceso de entrenamiento sería, simplemente, jugar al juego siendo el NPC el protagonista. Con este sistema se pierde capacidad de reflexión

por parte del diseñador y las decisiones erróneas que tenga, serán datos que se le aporta al sistema. Sin embargo, en juegos más estratégicos, se necesitan más tiempo de reflexión y sería más recomendable detener el juego mientras se muestran las tareas disponibles para ejecutar, para que la toma de decisiones sea sosegada. En general, esta forma de entrenar, si se puede llevar a cabo, genera menores trazas espurias. En cualquier caso, la frecuencia de adquisición de la información debe estar especificada acorde con el tipo de mecanismo que utilicemos para no perder eventos importantes en el juego, si la toma de decisiones por parte del diseñador se puede llevar en lapsos de tiempo muy cortos.

- Mostrar las variables de entorno al diseñador. Esta característica es clave ya que le aporta mucha más información al diseñador de la que tendría un jugador normal en un momento dado y por tanto, puede tomar una decisión mucho más informada sobre la tarea a realizar en cada momento por el NPC.

- El NPC debe ser visible en todo momento. Normalmente la cámara del juego no sigue a los NPCs sino al jugador, pero normalmente el entrenador debe poder ver a ambos. Esto implica ciertos cambios en el juego cuando el diseñador está entrenando. Si no se puede dejar de ver al jugador por algún motivo (porque lo tengamos que manejar como parte del entrenamiento, por ejemplo), hay dos soluciones posibles: que un visor muestre siempre al NPC a entrenar, o bien que haya dos diseñadores entrenando al NPC, uno moviendo al jugador y otro analizando los datos y dándole órdenes al personaje. En nuestro juego de pruebas optamos por crear un visor donde mostrábamos al Towot constantemente para ver cómo se comportaba.

- Tanto en modo entrenamiento, como en modo de ejecución, es conveniente mostrar la tarea que está ejecutando el NPC para facilitar su seguimiento.

Suele ser recomendable realizar diferentes entrenamientos en diferentes escenarios, para conseguir un comportamiento bien generalizado. Esto es importante si queremos aprender un comportamiento de aplicación general y no uno muy adaptado al entorno de entrenamiento.

Sin embargo, si queremos hacer comportamientos más específicos y que estos difieran en función del escenario donde se estén ejecutando, se recomienda entrenar un comportamiento concreto para cada escenario donde queramos que el comportamiento sea diferente. Podemos definir escenario en este contexto como una escena o pantalla de juego o simplemente una situación concreta dentro de una pantalla, dependerá de cada juego. No confundir con el escenario o el entorno de entrenamiento, que hemos usando anteriormente.

Además de definir la lista de tareas básicas y los comportamientos de alto nivel, el diseñador debe generar como documento de diseño para crear el prototipo, una tabla que relacione cada uno de los comportamientos o estrategias del NPC a un escenario. Esta tabla da información a los programadores para construir los entornos de entrenamiento, pero también muestra información al diseñador de qué comportamiento es aplicable en qué escenario, así como las posibles dudas del propio diseñador si las hubiera.

Lo ideal sería que, dado un escenario, sólo se pueda aplicar una estrategia o comportamiento predefinido, ya que esto implicaría que el comportamiento en ese escenario está claro por parte del diseñador, la menos inicialmente. Pero puede suceder que exista la posibilidad de aplicar diferentes estrategias en un mismo escenario. Si el diseñador no sabe cuál es la mejor estrategia a elegir, el proceso de entrenamiento y validación será una importante fuente de información para que el diseñador pueda decidir cuál utilizar. En el caso de que los dos sean factibles, se pueden fusionar ambos comportamientos mediante un BT, como se mostrará en los experimentos de el apartado 5.7. En cualquier caso, el entrenamiento y la posterior validación le aportará mucha información al diseñador para decidir qué hacer en caso de duda en sucesivas iteraciones.

Un ejemplo de cómo sería esta tabla se puede ver en la Tabla 5.1, que muestra una posible tabla de estrategias de un soldado en diferentes pantallas de un juego.

	Base	Frontline	Final Stage	Enemy Base
Defensive	X			
Offensive		X		
Suicide			X	
Stealthy				X
Nervious				X

Tabla 5.1: Ejemplo de la tabla de asignación de estrategias a escenarios.

En un caso como el que presenta la Tabla 5.1, en general, se crearán conjuntos de entrenamiento por demostración para cada estrategia en cada escenario y se entrenará por separado posteriormente. Si no hay dependencia de las acciones al escenario, en caso de que ambas estrategias sean válidas, se podrían mezclar ambos entrenamientos. Si un BT tiene más de un TQN, necesitará una base de casos por cada uno, a no ser que podamos utilizar el mismo comportamiento en varios de ellos.

Con toda esta información, el prototipo construido y el sistema de entrenamiento finalizado, el diseñador comenzará a crear el comportamiento. Inicialmente debe crear un BT, bien él o bien el programador, donde se coloquen uno o varios TQN en los sub-comportamientos que queramos aprender. Se debe establecer un nombre para el sub-comportamiento a aprender.

Por ejemplo, Atacar, patrullar o simplemente "comportamiento del NPC" indicando que el comportamiento a aprender es en sí mismo, todo el comportamiento de ese NPC. La cuestión es que todas las posibles tareas que pueden ser seleccionadas por el TQN deben implementar la tarea que queramos aprender. De esta forma, el TQN elegirlas y seleccionar una de ellas para ser ejecutadas en tiempo real en cada momento.

Como decimos, puede haber más de un TQN, aunque recomendamos que para entrenar sólo haya uno y que se entrene poco a poco si es un comportamiento más complejo, para que sea más sencillo llevarlo a cabo por el propio diseñador. Una vez tengamos el TQN colocado dentro del BT general del comportamiento y hayamos especificado la tarea a aprender, así como los atributos que consideramos relevantes para llevarla a cabo de la pizarra del BT, comenzamos el proceso de entrenamiento.

En este proceso, el diseñador debe mostrar al sistema qué debe hacer, cambiando la tarea a realizar en los momentos que el diseñador considere relevantes. Tome o no tome decisiones el diseñador, el sistema cada cierto tiempo guarda una fotografía de su estado de juego, es decir del valor de las variables que han sido definidas previamente como relevantes para el comportamiento del NPC.

Esa fotografía incluye la tarea que en ese momento se está ejecutando en el NPC y que se toma como tarea correcta en ese estado de juego. Si se ha producido algún suceso que enturbie el entrenamiento (por ejemplo, se ha tomado una decisión demasiado tarde o se ha olvidado hacer alguna de las cosas que se suponía que el diseñador debía enseñarle hacer al sistema), se puede repetir el proceso de entrenamiento para no contaminar los datos con trazas erróneas. Algunos algoritmos soportan mejor el ruido que otros, así que, dependiendo del ruido del entrenamiento, posteriormente podremos elegir uno u otro algoritmo, pero si los datos no contienen errores conocidos, el resultado será más preciso.

Una vez satisfecha la fase de entrenamiento, debemos validar que lo entrenado pueda ser utilizado en el juego. Para ello, se cambia el modo de entrenamiento a ejecución en el TQN. Ahora, el diseñador no tiene que darle ordenes al NPC, simplemente debe manejar al jugador (si es necesario) y observar el comportamiento del NPC. La validación del comportamiento puede necesitar varias ejecuciones consecutivas para estar seguros que el comportamiento aprendido es correcto, probando diferentes alternativas. Es interesante que el diseñador fuerce las situaciones más comprometidas para asegurarse de que el modelo responde bien ante ellas.

También se debe decidir qué algoritmo utilizar para el aprendizaje. No hay ninguno de ellos globalmente mejor que el otro a nivel de capacidad de representación, como veremos en el apartado 5.6. Así pues, se pueden probar diferentes alternativas para elegir cuál es el que produce mejores resultados en el contexto actual. En la Figura 5.10 se nuestra un esquema de las diferentes

alternativas con las que cuenta el diseñador a la hora de crear el modelo a modo de resumen. En el gráfico las tareas están ordenadas según nuestra recomendación de prioridad de arriba a abajo.

Los árboles de decisión apenas tienen coste computacional, comparado con las redes de neuronas, ya que se entrenan sin intervención del usuario. Sin embargo, las redes de neuronas se tardan más en entrenar y requieren de la intervención del diseñador (de ahí el actor *Designer* que supervisa el entrenamiento de la red de neuronas y no del árbol de decisión). Nuestra recomendación es usar primero árboles de decisión, ya que ofrecen un buen rendimiento y son fáciles y rápidos de entrenar. Así mismo, apenas requieren intervención del diseñador y suelen funcionar muy bien.

Además, una vez generados, pueden exportarse a un BT que proporcionará información explícita de qué comportamiento ha aprendido el NPC. En el caso de que el árbol de decisión no funcione correctamente, si no tenemos requisitos de espacio de memoria o de ejecución que desaconsejen usar KNN, recomendamos en segundo lugar usar este algoritmo, debido a que es más sencillo de usar por el diseñador que las redes de neuronas.

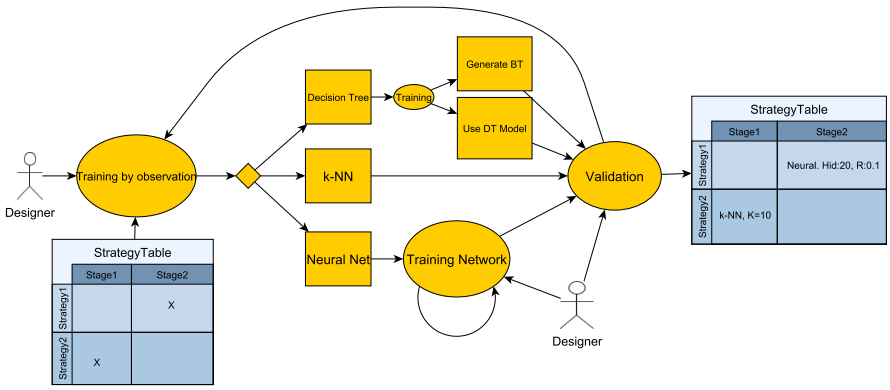


Figura 5.10: Diferentes alternativas de entrenamiento.

En el caso de que tampoco obtenga buenos resultados o simplemente no sea posible utilizarlo por cuestión de rendimiento, entonces la tercera opción a utilizar serían las redes de neuronas. Éstas suelen modelar muy bien el comportamiento y generalizan muy bien, siendo además muy rápidas en ejecución y compactas en memoria, pero tiene un proceso de entrenamiento offline (a parte del proceso de captación de trazas común a todos los algoritmos) bastante tedioso, que necesita normalmente sucesivas iteraciones. También soportan muy bien el ruido, por lo que son ideales cuando los datos del entrenamiento no son muy precisos.

Su principal inconveniente proviene de la necesidad de realizar un entrenamiento de la propia red con los datos de entrenamiento capturados en la fase de entrenamiento. Hay que configurar la red con diferentes paráme-

tros y realizar diferentes entrenamientos para conseguir un modelo pulido del mismo.

Una vez decidido el modelo, se realizarán diferentes procesos de entrenamiento y validación en el proceso iterativo, mostrado en la Figura 5.8. Se puede repetir el entrenamiento del sistema (la captura de los datos del diseñador) bien porque no se consigue el comportamiento esperado, bien porque el propio entrenamiento modifica el concepto del comportamiento en sí mismo al ver su resultado, o porque se quieren probar otras estrategias. También porque es muy posible que durante el entrenamiento hayan surgido modificaciones de las tareas básicas que se usan como primitivas, y que necesitan de una revisión por parte del programador y una posterior secuencia de entrenamiento y validación.

Finalmente, si entrenando con las sucesivas técnicas o incluso generando el árbol de comportamiento equivalente al árbol de decisión y editándolo a mano, no se consigue un comportamiento correcto, entonces el diseñador deberá acudir al programador para que implemente ese comportamiento. Pero no irá con las manos vacías.

El proceso de aprendizaje le habrá servido para perfilar el comportamiento y además, acude al programador con un comportamiento aprendido que puede servirle a éste como guía de lo que debe programar. Es posible incluso que pueda partir del BT generado por el comportamiento y editándolo, construir el comportamiento final. De esta forma, el programador tiene mucha más información que la descripción del comportamiento en un documento de diseño.

En cualquier caso, el proceso deberá terminar con un conjunto de comportamientos finales, listos para ser ejecutados dentro del juego, y que deberán ser evaluados por QA para darles su visto bueno definitivo. Junto a los comportamientos creados, se debe actualizar la tabla de estrategias con las configuraciones con las que se han obtenido dichos comportamientos, como muestra la Figura 5.10, así como, los ficheros donde se han almacenado las trazas de los entrenamientos. De esta forma, los diseñadores que monten los niveles finales, tendrán información precisa de cómo construir los comportamientos en cada nivel.

Esta documentación es importante generarla ya que puede haber una traza de comportamiento generada diferente para cada estrategia y cada nivel, por lo que, si no está perfectamente documentado, es posible que alguna de ellas pueda perderse.

Para finalizar, surge una pregunta obvia, ¿debemos aplicar esta metodología a todos los comportamientos? Nuestra respuesta es no. Sólo tiene sentido aplicar la metodología si el comportamiento es suficientemente complejo o si no está claro y queremos tener rápidamente un prototipo jugable con ese personaje. Si el personaje tiene un comportamiento claro, puede ser fácilmente implementable con unas pocas reglas o programarlo directamente

en una máquina de estados sencilla por código o con un BT simple. En estos casos, no tiene sentido aplicar la metodología.

5.5. Ejemplo en el entorno de pruebas Towot

Para ejemplificar como se implementaría esta metodología, vamos a utilizar nuevamente Towot como ejemplo práctico. Supongamos que queremos desarrollar el comportamiento de Towot para el juego final. Siguiendo el esquema de nuestra metodología, el diseñador ideará el concepto del comportamiento del Towot en líneas generales y redactará un documento de diseño con la descripción a alto nivel del mismo. En este documento, se deben incluir diagramas o alguna máquina de estados dibujada para ayudar a entender el comportamiento siempre y cuando el diseñador está familiarizado con ellas. El documento de diseño del personaje puede ir acompañado de *concepts arts* sobre el mismo, creados por los diseñadores conceptuales.

En el caso del Towot, en la Tabla 5.2, se muestra un ejemplo de una tabla de estrategias aplicada al Towot. Los nombres de los planetas son ficticios y están sacados del videojuego Starcraft®, así como sus configuraciones también son figuradas y su finalidad es simplemente para servir de ejemplo.

	Auir	Korhal	Sara	Tarsonis
Defensivo/ofensivo	X	X		
Arriesgada			X	
Avanzada	X			X

Tabla 5.2: Ejemplo de tabla de asignación de estrategias a escenarios.

La estrategia defensiva y la estrategia ofensiva de los entrenamientos se han fusionado en la estrategia mixta defensiva/ofensiva ya que la unión de ambas estrategias es aplicable también cuando se presentan situaciones en las que sólo se defiende o sólo se ataca. En lugar de estas dos estrategias, hemos añadido como ejemplo otro posible estrategia del Towot que hemos denominado *Estrategia Arriesgada*:

- *Estrategia defensiva/Ofensiva*: Cuando el jugador ataca, el Towot defiende el core. Cuando el jugador defiende el Towot ataca protegiendo la refinería.
- *Estrategia arriesgada*: El Towot defiende todo el tiempo al jugador, hasta que vea comprometido el core o la refinería. En esos casos, el Towot dejará de defender a Jacob y acudirá en defensa del core o la refinería, siendo más importante el core.
- *Estrategia avanzada*: El Towot protege la refinería, mientras no haya

nada en riesgo. Si alguno de los objetivos primarios está en riesgo, debe defender primero al core, luego a la refinería y luego a Jacob.

Una vez realizado el documento de diseño, el equipo de desarrollo tiene una serie de reuniones hasta fijar el prototipo del personaje. En dicho prototipo, participarán los programadores y los artistas por un lado y el diseñador o diseñadores por otro. Para crear el prototipo, se deben seleccionar un subconjunto de estrategias entrenables, así como los escenarios donde se van a utilizar. Dichos escenarios pueden ser mapas o pantallas concretas o simplemente situaciones para generar la tabla de estrategias, como vimos en el apartado anterior. Imaginemos que, en nuestro caso, el juego dispone de cuatro planetas donde recoger el mineral del juego y en cada planeta (cada escenario del juego) el Towot tomará una estrategia u otra en función de la forma del escenario.

- El escenario 1 (Auir) supongamos que es un escenario especial, donde se podría aplicar tanto la estrategia defensiva/ofensiva como la estrategia avanzada y el diseñador no tiene claro que sería la mejor.
- Pensemos el escenario 2 (Korhal) como un escenario donde hay una similar proporción de generadores cerca de la refinería y cerca del core, por lo tanto, el jugador unas veces, en función de la oleada, se comportará de forma más defensiva y otras más ofensiva. En cualquier caso, Towot deberá hacer lo contrario para evitar que los enemigos que consigan superar al jugador acaben destruyendo alguno de los objetivos.
- En el escenario 3 (Sara) la mayor parte de los generadores de enemigos estarían lejos del core, por lo que tiene sentido atacar todo el rato, tanto el jugador como el Towot y que solamente si se ve el core afectado o alguna refinería lejos del alcance del jugador, que sea el Towot quien la defienda.
- En el escenario 4 (Tarsonis) supongamos que es un escenario donde la estrategia avanzada funciona muy bien debido a que vienen enemigos desde cualquier parte y en grandes cantidades, por lo que el jugador y el Towot, van moviéndose en función del orden de llegada de los enemigos, unas veces atacando y otras defendiendo, de forma que el Towot debe tener mucha movilidad y acudir en ayuda del jugador con cierta inteligencia, independientemente de la acción del jugador.

También se deben detallar los comportamientos a bajo nivel que serán necesarios y que serán implementados por los programadores y aquellos que serán entrenados por los diseñadores. En el caso del Towot, las tareas a bajo nivel que implementarían los programadores serían las usados en los experimentos: *Protect Core*, *Protect Jacob*, *Protect Refinery* y *Recharge* además

de la percepción que introduce los diferentes valores que necesita el BT en la pizarra.

Con esta información en el prototipo, no sólo se construye el comportamiento básico del Towot, su diseño gráfico inicial y sus animaciones, sino también el entorno de entrenamiento para cada uno de los escenarios. En nuestro caso, el entorno de entrenamiento, que se describió en el apartado 4.2, está compuesto por un menú de selección que interrumpe el juego cuando el entrenador decide dar una orden al Towot, la cámara de seguimiento del Towot y la información de los atributos que se van a almacenar. Además, se simplificó el juego para hacerlo más fácil de jugar, eliminando la parte estratégica de colocación de torretas en el juego.

Cuando el prototipo está listo y se dispone de los entornos de prueba requeridos, el diseñador comienza a realizar los entrenamientos. Realizará un ciclo de entrenamiento y validación por cada escenario con la estrategia establecida como idónea para ese escenario, hasta conseguir que el comportamiento satisfaga las necesidades del juego.

En caso de que haya varias posibilidades, es decir, que haya diferentes estrategias a realizar en el mismo contexto (por ejemplo, en nuestro caso la estrategia ofensiva/defensiva y la estrategia avanzada podrían ser aplicables en el escenario 1), entonces el proceso de entrenamiento ayudará al diseñador a decidir cuál es la estrategia más idónea, ya que éste verá cual es el resultado de utilizar ambas estrategias. Un ejemplo práctico de esto se puede apreciar en la preparación de los experimentos del apartado 4.4.1.

La estrategia inicial de recarga del Towot al principio era que se recargara cuando la energía estuviera baja. Sin embargo, nos dimos cuenta que era mejor recargar al Towot justo en medio de las oleadas, porque tuviera o no tuviera energía suficiente para soportar una oleada más, era un momento ideal para recargarlo sin dejar desprotegido ninguno de los objetivos y de afrontar con garantías la siguiente oleada. Esto se descubrió haciendo entrenamientos de prueba para la experimentación realizada de una forma natural. Había que darles a los participantes del experimento una estrategia clara a seguir y está se fue adaptando conforme fuimos haciendo sucesivos entrenamientos.

Otra conclusión que se ha sacado de los entrenamientos ha sido que aunque el core es más importante porque la refinería puede ser reparada o incluso repuesta, ésta tiene mucha menos vida, por lo cual, puede ser interesante valorar cambiar el orden de las prioridades de algunos comportamientos y defender antes la refinería que el core, ya que ésta será destruida antes y reponerla puede suponer un gran coste de recursos para el jugador.

Así mismo, algunas de las tareas de bajo nivel, como por ejemplo la de moverse a un punto, también tuvieron que ser modificadas tras realizar los entrenamientos, al ver que en algunas ocasiones el Towot no alcanzaba los objetivos indicados.

También se detectaron nuevas acciones básicas, que, aunque no fueron implementadas en ningún experimento, surgieron al entrenar los comportamientos. Por ejemplo, en el diseño inicial de Towot, éste no puede atacar a los generadores de enemigos; sin embargo, al diseñar la estrategia ofensiva, esta necesidad surge de forma natural. De forma que, una posible modificación de la estrategia ofensiva puede ser mandar al Towot a atacar a los generadores en vez de mandarlo a proteger la refinería. Esta tarea no estaba en el documento de diseño del juego ni en la IA implementada cuando integramos la herramienta, sin embargo, al entrenar surgió de forma natural.

Estos son sólo algunos ejemplos prácticos de características de diseño y de programación que fueron modificadas o descubiertas entrenando al Towot, que pretenden ejemplificar que el propio proceso de entrenamiento, no sólo ha dado información de si los comportamientos aprendidos eran buenos o malos, si no que han proporcionado mucha información adicional útil en un proceso de producción, que de otra forma, probablemente no hubieran sido obtenidos hasta la fase de pruebas.

De esta forma, el propio entrenamiento retroalimenta a ambos roles, mucho antes de llegar a tener un prototipo del comportamiento cerrado, lo cual hace que dicho prototipo tenga mucha mayor consistencia, tanto a nivel jugable como técnico y que se adelanten algunos problemas que normalmente se descubrían en la fase de pruebas, en la fase de producción.

Dentro del entrenamiento en sí, el diseñador deberá probar las diferentes técnicas para generar el modelo que dispone el sistema. Como ya dijimos en el apartado anterior, recomendamos por sencillez utilizar primero árboles de decisión o k-NN. Si el rendimiento del juego es un problema, es preferible usar árboles de decisión, mucho más livianos en memoria y tiempo de ejecución. Sin embargo, si estos no modelan correctamente el comportamiento hay dos opciones más: o bien generar el árbol equivalente a lo entrenado y editarlo a mano o bien utilizar redes de neuronas. Las redes de neuronas implican un mayor esfuerzo de entrenamiento, ya que necesitan diferentes entrenamientos con diferentes parámetros donde el diseñador debe comprobar que el error de validación no crezca demasiado, pero pueden generar buenos modelos y sobre todo son muy compactos en memoria y muy rápidos en tiempo de ejecución. Algunos de los resultados obtenidos en los experimentos del apartado 5.7 (los más complejos) costaron varias iteraciones conseguirlos, cambiando los parámetros de la red hasta encontrar uno que se comportara adecuadamente.

Si ninguna de estas técnicas consigue un buen resultado, una última alternativa es dividir el comportamiento en comportamientos más sencillos. Por ejemplo, en el comportamiento defensivo/ofensivo, como veremos en la experimentación, si lo dividimos en defensivo y ofensivo y mezclamos usando el propio editor de BTs, los resultados obtenidos mejoran sustancialmente frente a intentar aprender el comportamiento todo junto de una sola vez (véase Tabla 5.5 del apartado 5.7). Así pues, suele ser más fácil entrenar compor-

tamientos lo más específicos posibles porque así serán más reutilizables en otros entrenamientos.

Como última opción, el diseñador puede desistir de entrenar alguno de los comportamientos, si no consigue crear por sí mismo el comportamiento óptimo y acudir al programador con un modelo del comportamiento y el BT que se ha entrenado. Así pues, con la ayuda del propio comportamiento generado, el diseñador puede explicarle al programador cómo debe comportarse el Towot, en qué cosas el entrenamiento aprendido falla y qué es lo que espera que haga realmente en esos casos. Imaginemos que no se consigue entrenar el Towot para que se recargue correctamente entre oleadas, algo que es habitual en las estrategias más complejas usando k-NN.

El diseñador puede mostrar el comportamiento general del Towot e incidir en el hecho de que cuando la energía esté vacía o cerca de acabarse, debe recargarse el Towot, cosa que, en el modelo generado, imaginemos que no hace. El programador con este modelo y las correcciones podría crear la IA sabiendo exactamente lo que el diseñador quiere que haga el comportamiento. De esta forma, el programador tiene un ejemplo empírico de qué es lo que tiene que conseguir y no una descripción textual ambigua. Además, puede obtener información extra si genera el BT extraído del modelo obtenido por el árbol de decisión, que le aportará información extra a la hora de crear el comportamiento usando el editor de BTs.

Finalmente, el proceso de entrenamiento terminaría con un nuevo prototipo de comportamiento para cada uno de los escenarios planteados y con la tabla de comportamientos actualizada como se muestra en la Tabla 5.3. A la tabla se añade no sólo la estrategia que finalmente usaremos en el escenario, sino también la configuración utilizada para cada una de las estrategias y el nombre de la base de casos utilizada para cada uno. De esta forma, simplemente mirando la tabla de estrategias, un diseñador podrá montar el nivel del juego final. Estos comportamientos, listos para ser ejecutados en el juego, se revisarían por el equipo de QA para darles el visto bueno final. Si QA cree que el comportamiento es correcto, no tiene bugs y es divertido, se dará por finalizado y se incluirá en el prototipo del juego. Esto no cambiará hasta que alguna circunstancia haga revisar el comportamiento (o bien un cambio de diseño o bien se ha detectado un error posterior).

	Auir	Korhal	Sara	Tarsonis
Def/ofen		PdB & BT		
Arriesgada			Net,Hid:10,R:0.1	
Avanzada	kNN k=10			Knn k=10

Tabla 5.3: Ejemplo de tabla de asignación de estrategias a escenarios actualizada con la información de cómo se han obtenido los comportamientos.

5.6. Comparación entre las diferentes tecnologías utilizadas

Finalmente describiremos la experimentación realizada para estudiar la precisión de los diferentes algoritmos incluidos en la herramienta, para generar el modelo del personaje. También queremos validar si efectivamente la aproximación mixta de usar programación por demostración y árboles de comportamiento, permite crear comportamientos de una forma más flexible y con mejores resultados. Nuestra hipótesis de partida es que la programación por demostración puede ayudar a crear comportamientos a diseñadores sin conocimientos de programación. Sin embargo, por sí sola, la programación por demostración no es suficiente cuando el comportamiento es complejo. Una aproximación mixta entre BTs y programación por demostración debería conseguir mejores resultados. Estos experimentos y sus conclusiones fueron publicados en (Sagredo-Olivenza et al., 2017b).

En el experimento hemos definido 4 estrategias para el Towot:

- Estrategia defensiva: el Towot defiende el core mientras el jugador ataca en la parte de arriba del escenario, junto a la refinería y los generadores superiores.
- Estrategia ofensiva: el Towot defiende la refinería, mientras el jugador defiende el core y los generadores inferiores.
- Estrategia mixta: dependiendo de si el jugador ataca o defiende, el Towot ejecuta la estrategia contraria.
- Estrategia compleja: el Towot, por defecto, defiende la refinería si nada está en peligro, pero si detecta que está en peligro el core, la propia refinería o el jugador, entonces los defenderá y abandonará su posición. Ante la duda de qué defender primero, se utiliza un orden de prioridad, así pues en caso de duda, el core es más prioritario que la refinería y ésta lo es más que el jugador.

Los TQNs pueden grabar los parámetros que seleccionemos del árbol de comportamiento que los ejecute, junto con la acción indicada por el diseñador en la fase de entrenamiento. Así pues, hemos realizado una grabación de entrenamiento y 10 grabaciones de validación con cada estrategia, pero jugando de diferentes formas para validar si los modelos utilizados consiguen generalizar correctamente. Para el experimento hemos utilizado, por un lado, KNN con $k = 10$ y seleccionamos la tarea mayoritaria entre los diez casos más similares y por otro, un perceptrón multicapa con aprendizaje por retropropagación con siete neuronas de entrada, cuatro neuronas de salida y diferentes configuraciones de neuronas ocultas y razón de aprendizaje. También se utilizó el árbol de decisión en la comparativa para cada estrategia.

La selección del valor de K se realizó en base a los experimentos previos realizados en 4.4 donde pudimos comprobar que dicho valor ofrecía buenos resultados.

Para entrenar la red de neuronas, se usa una de las partidas grabadas de cada estrategia como entrenamiento y otra como validación y es el diseñador el que debe, mirando el error de entrenamiento y el de validación, detener la red cuando detecte que el error de validación comienza a incrementarse. En el caso de la estrategia mixta, la base de casos utilizada para entrenar es la unión de las trazas almacenadas en el entrenamiento defensivo y ofensivo.

Una vez entrenada la red, se le van proporcionando ejemplos de las 10 partidas grabadas y comparando el resultado de la red con el resultado grabado. Los resultados que se muestra en la Tabla 5.4 representan el promedio y la desviación típica de cada uno de ellos. Además, en el caso de las redes de neuronas, se indica el número de capas ocultas y la razón de entrenamiento utilizada para conseguir los resultados.

	Defensivo	Ofensivo	Mixta	Compleja
k-NN Med	0,939	0,9422	0,7468	0,7833
k-NN Desv	0,01	0,004	0,08	0,032
k-NN pesos=1 Med	0,91	0,945	0,676	0,779
k-NN pesos=1 Desv	0,006	0,007	0,054	0,032
Árbol D Med	0,9175	0,945	0,837	0,705
Árbol D Desv	0,04	0,01	0,056	0,045
Neuronas Med	0,9418	0,9465	0,78385	0,7536
Neuronas Desv	0,011	0,009	0,063	0,071
Neuronas ocultas	10	14	21	10
Factor aprendizaje	0,1	0,15	0,2	0,1

Tabla 5.4: Resumen de la experimentación realizada comparando las diferentes técnicas utilizadas usando la precisión media

Como se puede apreciar en la tabla, los resultados son similares y tienen diferencias en función de la estrategia utilizada. En la primera y segunda estrategia, la precisión es muy alta con una tasa de acierto entre el 93 % y 95 %. Sin embargo, la tasa de acierto, aunque alta, disminuye en la estrategia mixta y la estrategia compleja. Los árboles de decisión se comportan muy bien en la estrategia mixta, con respecto a k-NN o al perceptrón multicapa.

Sin embargo, en la cuarta estrategia, funcionan mejor las otras dos técnicas. Lo que también podemos apreciar es que k-NN en general funciona peor si no establecemos unos pesos que ayuden a la función de similitud a seleccionar los casos más representativos. Esto es interesante ya que el diseñador puede no saber que pesos elegir. Se podrían calcular estos pesos de forma automática usando, por ejemplo algoritmos genéticos (Kelly Jr y Davis, 1991),

pero actualmente esta característica no está implementada.

5.7. Comparación usando programación por demostración unida a BTs

Una posible solución para mejorar la tasa de acierto en el caso de la estrategia mixta del apartado 5.6 es seleccionar que estrategia usar mediante unos cuantos nodos en el árbol de comportamiento que ejecuta el TQN. Cómo se puede observar en los resultados de la Tabla 5.4, por separado las estrategias defensiva y ofensiva consiguen una tasa de acierto muy alta.

Así pues, construimos un nuevo BT con dos TQN, uno entrenado con las trazas defensivas y el otro con las ofensivas y un selector de prioridad que determina si Jacob está atacando o defendiendo en función del parámetro *JacobDistanceToCore*. La técnica utilizada para obtener el modelo de comportamiento fue redes de neuronas con la misma ocnfiguración que la descrita en los experimentos del apartado 5.6. Los resultados obtenidos se muestran en la Tabla 5.5

Tercera estrategia	k-NN	Árbol D	Neuronas	PdB & BT
3º Estrategia Media	0,7468	0,8376	0,78385	0,839
3º Estrategia Desv	0,08	0,0557	0,063	0,07

Tabla 5.5: Tasa de acierto de los diferentes modelos utilizados en la tercera estrategia

Como vemos en la tabla, la tasa de acierto de las redes de neuronas con dicha estrategia mejora y se sitúa muy pareja a la de los árboles de decisión. Esto demuestra que mezclar los comportamientos usando árboles de decisión, puede ayudar a mejorar los resultados del entrenamiento. Si la tasa de acierto no es suficientemente buena para el diseñador, por ejemplo, al mezclar los comportamientos, una posibilidad es mezclar dichos comportamientos mediante el BT, añadiendo programación manual. Esto es posible gracias a nuestra aproximación mixta donde podemos mezclar ambas de forma natural. Así que editando el árbol para que sea éste el que decida qué estrategia utilizar, la tasa de acierto se incrementa.

En la Figura 5.11 se muestra el árbol que se ha utilizado para combinar la estrategia defensiva y la estrategia ofensiva.

Los resultados mostrados en la Tabla 5.5, son generados simulando el nodo del BT mediante una condición en el evaluador del experimento. Dependiendo del parámetro: distancia al core, decidimos si Jacob está atacando o defendiendo y seleccionamos en consecuencia el TQN entrenado con la estrategia defensiva o el TQN entrenado con la estrategia ofensiva.

Probando los comportamientos en el juego, se puede apreciar como la

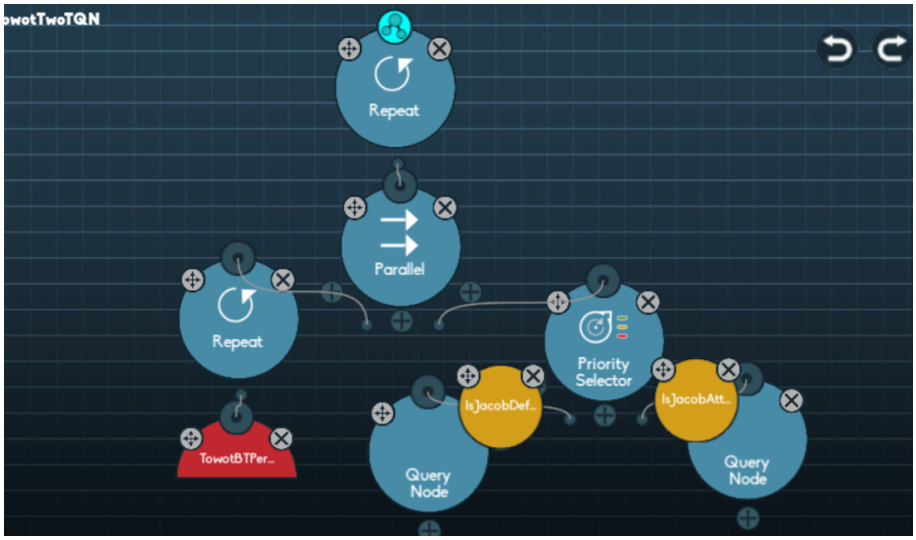


Figura 5.11: BT donde hemos mezclado ambos comportamientos: Defensivo y ofensivo

aproximación mixta se comporta generalmente como se espera. La versión con redes de neuronas tiene mejores resultados en cuanto a experiencia de juego que k-NN. Éste tiene más oscilaciones. Por ejemplo, en ocasiones cuando el Towot debería defender la refinería, baja a defender durante unos pocos segundos el core y luego rectifica su decisión o viceversa. En 10 entrenamientos realizados, con k-NN siete han sufrido de cierto comportamiento errático. Con redes de neuronas tan solo tres y con el nodo intermedio sólo en una ocasión el Towot intentó por unos segundos bajar a defender cuando no debía.

Nótese, que este nodo que selecciona una de los dos *Trained Query Nodes*, en esta implementación está codificado manualmente. Pero podría ser un tercer *Trained Query Node* y el diseñador entrenar dicho nodo, indicándole cuándo debe atacar el Towot y cuándo debe defender. De esta forma, con nuestra aproximación, podemos descomponer problemas complejos en problemas más simples, más fáciles de aprender por demostración y utilizar estos comportamientos aprendidos en otros nodos consulta, que podemos entrenar de forma que podemos generar comportamientos más complejos y de mayor nivel de abstracción que si abordásemos el problema en su conjunto. Este nuevo nodo debería aprender a como implementar una nueva acción de la que heredaran los comportamientos aprendidos. Para poder permitir esto, los *Trained Query Nodes* que apareden en la figura deben ser almacenados en un subcomportamiento que debe ser etiquetado como implementación de la nueva tarea a aprender. En nuestro caso nuestra nueva tarea a aprender se podría denominar *Modo de Ataque* que tendría dos acciones que la implementarían, *ofensivo* y *defensivo*. Dichos subcomportamientos simplemente

tendrían un nodo *Trained Query Node* y recibirían los datos de la pizarra del comportamiento padre a través de sus parámetros de entrada.

Con estos resultados, podemos concluir que nuestra aproximación mixta es una muy buena herramienta para crear comportamientos sin programar, ya que permitimos con esta aproximación crear comportamientos más complejos con mayores tasas de acierto, donde además damos mucha libertad para que el diseñador pueda usar diferentes técnicas para generar el modelo. Al permitir dividir los comportamientos más complejos en sub-comportamientos más sencillos para aprenderlos por PbD para posteriormente mezclarlos usando BTs u otros TQN. De esta forma se pueden crear una gran cantidad de comportamientos por los diseñadores de forma autónoma.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Contribuciones

Como conclusión final a la tesis doctoral, el objetivo perseguido en el presente trabajo es facilitar el proceso de creación de comportamientos inteligentes para NPC en videojuegos, por parte de los diseñadores, permitiendo su integración en el proceso de desarrollo del comportamiento y creando un modelo y una metodología que les permita colaborar mejor con los programadores, para conseguir minimizar las iteraciones necesarias para crear un comportamiento final. El objetivo final es ayudar a que los diseñadores y programadores a que puedan complementarse y cooperar mejor en el campo de la inteligencia artificial aplicada a los NPCs de un juego.

Para conseguirlo, ponemos en manos del diseñador una herramienta denominada *Behavior Bricks*, un editor de comportamientos que implementa el modelo de los árboles de comportamiento (BTs) descrito en este trabajo, con dos características diferenciadoras con respecto a otras herramientas: la capacidad de reutilizar los comportamientos gracias a que estos están parametrizados, por lo que pueden ser instanciados desde cualquier otro comportamiento, como si de una acción básica se tratase y la posibilidad de crear comportamientos como una o varias tareas genéricas, que puedan ser implementadas de forma concreta de múltiples formas, cuando estos comportamientos se instancian en un NPC concreto.

Además, debido a los estudios realizados con usuarios, nos dimos cuenta que los editores de comportamiento basados en BTs, incluso con una metodología que simplifica la creación de los comportamientos y que limita su uso sólo a los comportamientos a alto nivel del NPC, no se adaptaban bien a diseñadores sin demasiados conocimientos técnicos, debido a que sus resultados eran peores a los de sus compañeros con conocimientos de programación. Por lo que quisimos ir más allá y dotar a la herramienta de una forma de crear comportamientos por demostración, para poder crear comportamientos sin programación, de forma que los diseñadores sin tener en cuenta su

formación técnica, puedan crear los comportamientos de los NPCs de forma autónoma. Integrando los BTs y la programación por demostración en un mismo modelo, como demostramos en los experimentos realizados, no sólo permitimos a los diseñadores entrenar por demostración partes de ese comportamiento, también podemos crear comportamientos más complejos, con mayores tasas de aciertos que utilizando la programación por demostración convencional.

Así mismo, ofrecemos tres técnicas diferentes de creación del modelo del NpC en base a los datos proporcionados en la fase de observación, cada una de estas técnicas con sus ventajas e inconvenientes, pero que dotan al usuario de múltiples alternativas para crear el comportamiento, ya que, en principio, cada técnica ofrece mejores resultados en unos comportamientos que en otros. Las tres técnicas utilizadas las resumimos a continuación:

- Usando razonamiento basado en casos: Con esta técnica el diseñador, para crear el modelo de comportamiento del NPC, sólo tiene que seleccionar el conjunto de pesos con el que ponderar los atributos y algunos parámetros relacionados con el algoritmo k-NN que utiliza como base, por ejemplo el número de vecinos que tendrá en cuenta el algoritmo para seleccionar el comportamiento (el parámetro k de k-NN) o el criterio de selección con el que se elije el comportamiento (el más repetido, el más similar, etc). En tiempo de ejecución y espacio en memoria, es el más costoso de los tres algoritmos, pero obtiene buenos resultados y es sencillo para el diseñador. Su gran ventaja es que no necesita un proceso de aprendizaje posterior a la captura de los requisitos y es muy fácil añadir nuevos ejemplos a la base de casos, para mejorar el entrenamiento sin tener que re-entrenar el sistema.

- Usando árboles de decisión: es el más fácil de utilizar para el diseñador, ya que no requiere configuración. Genera un modelo con los datos, que precisa un entrenamiento posterior a la captación de los datos. Por suerte, el proceso de aprendizaje es bastante rápido y no influye demasiado en el proceso de entrenamiento, como sí sucede con las redes de neuronas. Su gran ventaja es su sencillez, su poca necesidad de espacio y de tiempo de ejecución, así como la posibilidad de generar, en base a la descripción del árbol de decisión, un árbol de comportamiento equivalente al modelo observado y aprendido por el árbol de decisión, que puede ser ejecutado o modificado tanto por el diseñador como el programador para ajustarlo.

- Usando redes de neuronas: Estas modelan muy bien algunos comportamientos y crean comportamientos con una gran sensación de inteligencia. Son capaces de abstraer muy bien comportamientos cuando estos son más complejos, como por ejemplo ser capaz de recargar al Towot en los experimentos realizados, en escenarios de juego que diferían sustan-

cialmente de los escenarios que cubrían el entrenamiento. A modo de ejemplo, durante las pruebas realizadas, la redes conseguían recargarse con muy poca energía, no habiendo ningún ejemplo con una recarga en estas circunstancias, situación en la que otros algoritmos, como por ejemplo usando razonamiento basado en casos, fallarían. También genera un modelo muy rápido en ejecución y muy compacto en memoria. Su principal hándicap es el costoso proceso de configuración de la red, así como el tedioso proceso de entrenamiento necesario, posterior a la adquisición de los datos de entrenamiento.

Así mismo, para ayudar a los diseñadores y programadores a cooperar entre sí, se ha definido una metodología de uso de nuestra herramienta, que permite dividir el trabajo entre programadores y diseñadores, colocando al diseñador en el centro del proceso de creación del comportamiento, en vez de su tradicional rol de diseñador y supervisor del resultado. Haciendo que ambos roles (programador y diseñador) puedan trabajar en paralelo y cooperar. Además, situamos el centro de la especificación del comportamiento en el modelo generado por el diseñador por demostración. El programador, si el diseñador no consigue crear un comportamiento que satisfaga sus necesidades por demostración, ahora tiene un modelo de comportamiento que puede ejecutar, mucho más concreto que las especificaciones textuales tradicionales, normalmente muy vagas e imprecisas.

El programador puede ver in situ, cómo debe comportarse el comportamiento, incluso puede partir de dicho modelo entrenado si genera el árbol de comportamiento equivalente, que le servirá de mucha ayuda. De esta forma, sabe exactamente cómo tiene que comportarse el NPC y las desviaciones a lo que el diseñador quiere que no han sido capaces de ser modeladas, son solamente la parte que el diseñador debe documentar y explicar al programador. De esta forma, conseguimos reducir las iteraciones entre diseño, programación y evaluación y por tanto el tiempo de desarrollo.

6.2. Discusión de los resultados

Las principales contribuciones de esta tesis están orientadas hacia la extensión del modelo de árboles de comportamiento mediante el nodo denominado *Trained Query Node*, que permite extender la funcionalidad de los árboles de comportamiento, permitiendo aprendizaje automático en los mismos. Para ello usando el modelo de programación por demostración, lo que permite crear comportamientos sin necesidad de programar, sino simplemente, enseñando al sistema en su propia interfaz cómo debe comportarse.

El sistema consigue aprender en base a los ejemplos que el diseñador le proporciona en una fase de captación de dichos ejemplos, para que con esos datos posteriormente, el nodo pueda modelar el comportamiento del NPC usando diferentes técnicas. Parte de este trabajo está ligado a la creación

de *Behavior Bricks* y de la creación del *middleware* de interfaces en Unity (UHotDraw), con el que se construyó el editor. Dicho editor tiene unas características que le permiten crear comportamientos que pueden ser reutilizados en otros NPCs, gracias a sus capacidades de abstracción mediante parámetros de entrada y de salida (de forma similar a como lo hacen los procedimientos y funciones de un lenguaje de programación) y la posibilidad de poder realizar diferentes implementaciones de la misma tarea de alto nivel que se adaptan a diferentes NPCs, cuando el comportamiento es instanciado.

Pero pensamos que nuestra principal aportación, no utilizada previamente en la literatura que hemos evaluado, es la integración de la programación por demostración dentro de los árboles de comportamiento sobre una herramienta como *Behavior Bricks* así como la elaboración de una metodología para poder utilizar esta herramienta en un entorno de trabajo donde los diseñadores y programadores puedan cooperar en la creación de los comportamientos de una forma que hasta ahora no era habitual.

Hemos demostrado con los experimentos del Capítulo 5, que la integración entre ambos modelos (los BTs y la programación por demostración) no sólo es buena desde un punto de vista del diseñador, por la posibilidad de crear comportamientos sin necesidad de programar, manteniendo un modelo de comportamiento familiar con la industria, sino que la combinación de ambos permite mejorar los resultados que se obtienen con la programación por demostración por sí sola, creando una herramienta que es capaz de generar, con poca programación de carácter visual, y un buen entrenamiento y planificación de lo que se va a entrenar, una gran cantidad de comportamientos con una tasa de acierto bastante elevada. Dándole al diseñador una potente herramienta para crear comportamientos sin prácticamente tener conocimientos de programación. El diseñador tiene múltiples formas de afrontar el entrenamiento y si no consigue entrenar un comportamiento que satisfaga sus requisitos, pueda darle al programador un modelo y una demostración ejecutable de cómo debe comportarse el NPC. Esto simplifica enormemente la traslación de información entre el diseñador y el programador.

Uno de los principales riesgos del presente trabajo es la implantación del sistema en un entorno de desarrollo real. Pensamos que sería más fácil que otros sistemas que utilizan aprendizaje automático o programación por demostración, gracias a que está integrado en una herramienta como *Behavior Bricks* que durante la fecha de la publicación del presente trabajo, ha estado normalmente entre las 5 herramientas de scripting visual gratuitas de Unity más usadas y que se basa en un modelo ampliamente conocido como los árboles de comportamiento. Sin embargo, existen unos riesgos que no podemos obviar.

El principal de ellos es que hay que convencer a los productores, diseñadores y programadores a cambiar su filosofía de trabajo. Esto no es nada fácil ya que muchos estudios llevan años trabajando de una forma determinada.

Los programadores son recelosos de dejar en manos de los diseñadores la creación de comportamientos y los productores deben percibir que esta herramienta les producirá un beneficio, reduciendo los tiempos de desarrollo. Si esto no es percibido de esta forma, hay un riesgo de que la herramienta no sea aceptada por ciertos equipos de desarrollo. También actualmente la herramienta está fuertemente ligada a Unity y por tanto no podrían utilizarla equipas de desarrollo que no trabajen con este motor gráfico.

Por otro lado, hay que valorar el tiempo que puede tardarse en crear el entorno de entrenamiento. Para la creación de NPCs muy complejos, la interfaz que permita manejar el NPC al diseñador será también compleja y repleta de opciones y necesitará un tiempo de desarrollo y de aprendizaje por parte del diseñador.

Por otro lado, también hay que tener en cuenta que ciertos comportamientos pueden no ser entrenados fácilmente y se requieran sucesivas iteraciones o incluso dividir el entrenamiento en diferentes subcomportamientos más sencillos de aprender. Esta tarea puede no ser algo natural para el diseñador, por lo que al principio requerirá un cierto tiempo para aprender a usar la herramienta y detectar las posibles acciones que puede realizar en caso de que el comportamiento no se comporte de la forma esperada.

Así mismo, existen características que mejorar en el presente trabajo. Por ejemplo, existen multitud de aspectos que consideramos pueden ampliarse y perfeccionarse, que describiremos en el apartado de trabajo futuro.

6.3. Trabajo futuro

Creemos que la herramienta creada y su implementación en el juego Towot para realizar los experimentos, sirve como un magnifico campo de pruebas para probar otras técnicas de inteligencia artificial y aprendizaje automático. La arquitectura de *Trained Query Node* es flexible y permite añadir nuevos métodos para generar comportamientos de forma fácil. Prueba de ello es que en el estado actual de la herramienta se han implementado 3 métodos diferentes. Así pues, una vía para ampliar el trabajo en el futuro es incluir nuevas técnicas para medir su eficacia con respecto a las tres utilizadas.

En cuanto a las técnicas que actualmente están implementadas, se pueden ampliar con nuevas mejoras para incrementar la tasa de acierto. Por ejemplo, en el caso del razonamiento basado en casos, se puede mejorar el rendimiento aplicando diferentes técnicas de optimización que mejoren su rendimiento. Esto es importante debido a que para usarlo en la práctica en un videojuego, cuanta menos memoria ocupe y más rápido se ejecute mucho mejor, ya que los videojuegos son aplicaciones que deben reaccionar en tiempo real, muy sensibles al rendimiento. Por otro lado, una posible mejora es utilizar algoritmos genéticos para calcular los pesos que se utilizan en k-NN como en (Kelly Jr y Davis, 1991).

En cuanto a las redes de neuronas, uno de los problemas de su uso como técnica de modelado del comportamiento es su complejo y tedioso sistema de entrenamiento. Una futuro trabajo puede abordar un enfoque neuroevolutivo (Floreano et al., 2008; Yao, 1999). Este enfoque utiliza una red de neuronas como base para que un algoritmo genético encuentre una configuración de la red de forma automática. Cada individuo es una configuración de una red de neuronas y el *fitness* que guía el algoritmo de selección natural es el error de convergencia que alcanza la red. De esta forma, el usuario puede utilizar redes de neuronas sin necesidad de configurar la topología de la red y sin tener que probar diferentes configuraciones y entrenamientos para conseguir aprender. Esto se delega a un proceso de búsqueda automático, que es costoso en tiempo de ejecución pero que se realiza sin la intervención del diseñador.

Un interesante campo a explorar sería cómo mejorar el proceso de entrenamiento del sistema, para permitir correcciones de lo entrenado mientras se evalúa el sistema. Por ejemplo, si el diseñador se percata que el comportamiento en fase de validación se equivoca, se podría forzar a ejecutar la tarea corregida y modificar la base de casos, para que ésta guardase los nuevos casos e invalidara los casos que en dicho contexto llevaron al error. Esta técnica pensamos que se integraría muy bien con el aprendizaje basado en casos, pero habría que estudiar cómo utilizarla en las demás técnicas sin tener que re-entrenar el modelo, sobre todo en el caso de las redes de neuronas, por su especial coste. Así mismo, se podrían intentar genera algoritmos que permitieran al NPC descubrir el comportamiento de forma autónoma sin necesidad de ser entrenado por el diseñador, aunque habría que estudiar si estos comportamientos serían aplicables en un entorno de desarrollo real y no generarían demasiados comportamientos emergentes.

Y, por último, la metodología propuesta por la presente tesis no ha sido evaluada de forma empírica. Un posible trabajo futuro sería poner en practica la metodología para evaluarla en un proyecto real, midiendo si realmente mejora o no los tiempos de desarrollo y adaptando las premisas aquí planteadas si fueran necesario. Es un experimento complejo y a largo plazo, pero creemos que sería muy interesante para perfeccionar la metodología expuesta en la presente tesis.

Bibliografía

Aamodt, A. y Plaza, E. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, vol. 7(1), páginas 39–59, 1994.

Aaron Canary, J. S. Free-range AI: Creating compelling characters for open world games. En *Game Developers Conference*. 2014. [Online; accessed 02/02/2017].

Aha, D. W., Molineaux, M. y Ponsen, M. Learning to win: Case-based plan selection in a real-time strategy game. En *International Conference on Case-Based Reasoning*, páginas 5–20. Springer, 2005.

Almeida, M. S. O. y da Silva, F. S. C. A systematic review of game design methods and tools. En *International Conference on Entertainment Computing*, páginas 17–29. Springer, 2013.

Argall, B. D., Chernova, S., Veloso, M. y Browning, B. A survey of robot learning from demonstration. *Robotics and autonomous systems*, vol. 57(5), páginas 469–483, 2009.

Avery, P., Louis, S. y Avery, B. Evolving coordinated spatial tactics for autonomous entities using influence maps. En *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, páginas 341–348. IEEE, 2009.

B. W. Silverman, M. C. J. E. Fix and J.L. Hodges (1951): An important contribution to nonparametric discriminant analysis and density estimation: Commentary on fix and hodges (1951). *International Statistical Review / Revue Internationale de Statistique*, vol. 57(3), páginas 233–238, 1989. ISSN 03067734, 17515823.

Baskerville, R. y Pries-Heje, J. Short cycle time systems development. *Information Systems Journal*, vol. 14(3), páginas 237–264, 2004.

Blum, A. L. y Langley, P. Selection of relevant features and examples in machine learning. *Artificial intelligence*, vol. 97(1), páginas 245–271, 1997.

BONET, J. D. y STAUFFER, C. Learning to play pac-man using incremental reinforcement learning. En *Proceedings of the Congress on Evolutionary Computation*. 1999.

BOURG, D. M. y SEEMANN, G. *AI for game developers*. O'Reilly Media, Inc., 2004.

CHAMPANDARD, A. Behavior trees for next-gen game ai. En *Game Developers Conference, Audio Lecture*. 2007a.

CHAMPANDARD, A. J. Behavior trees for Next-Gen AI. En *Game Developers Conference*. 2005.

CHAMPANDARD, A. J. 10 Reasons the Age of Finite State Machines is Over. `http://aigamedev.com/open/article/fsm-age-is-over/`, 2007b. [Online; accessed 25/03/2017].

CHAMPANDARD, A. J. Top 10 most influential ai games. 2007c. `http://aigamedev.com/open/highlights/top-ai-games/`.

CHAMPANDARD, A. J. Getting started with decision making and control systems. En *AI Game Programming Wisdom*, vol. 4, capítulo 3, páginas 257–264. Course Technology, 2008.

CHANDLER, H. M. *The game production handbook*. Jones & Bartlett Publishers, 2009.

COLLEDANCHISE, M. y ÖGREN, P. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Transactions on Robotics*, 2016.

COLLEDANCHISE, M., PARASURAMAN, R. y ÖGREN, P. Learning of behavior trees for autonomous agents. *arXiv*, 2015.

CORDONE, R. *Unreal Development Kit Game Programming with UnrealScript: Beginner's Guide*. Packt Publishing Ltd, 2011.

DEREK NEAL, B. H. Designing ai for competitive games. En *Game Developers Conference*. 2016.

DERESZYNSKI, E. W., HOSTETLER, J., FERN, A., DIETTERICH, T. G., HOANG, T.-T. y UDARBE, M. Learning probabilistic behavior models in real-time strategy games. En *AIIDE*. 2011.

DEY, R. y CHILD, C. QL-BT: Enhancing behaviour tree design and implementation with Q-learning. En *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, páginas 1–8. IEEE, 2013.

DEZA, M. M. y DEZA, E. Encyclopedia of distances. páginas 1–583. Springer, 2009.

DREW RECHNER, P. D. Blending autonomy and control: Creating npcs for tom Clancy's The Division. En *Game Developers Conference*. 2016.

FLOREANO, D., DÜRR, P. y MATTIUSSI, C. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, vol. 1(1), páginas 47–62, 2008.

FLÓREZ-PUGA, G., GÓMEZ-MARTÍN, M. A., GÓMEZ-MARTÍN, P. P., DÍAZ-AGUDO, B. y GONZÁLEZ-CALERO, P. A. Query enabled behaviour trees. *IEEE Transactions on Computational Intelligence And AI In Games*, vol. 1(4), páginas 298–308, 2009.

FLOYD, M. W., ESFANDIARI, B. y LAM, K. A case-based reasoning approach to imitating robocup players. En *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008, Coconut Grove, Florida, USA* (editado por D. Wilson y H. C. Lane), páginas 251–256. AAAI Press, 2008. ISBN 978-1-57735-365-2.

GARDNER, M. W. y DORLING, S. Artificial neural networks, the multilayer perceptron, a review of applications in the atmospheric sciences. *Atmospheric environment*, vol. 32(14), páginas 2627–2636, 1998.

GONZALEZ-PEREZ, C., HENDERSON-SELLERS, B. y DROMEY, G. A metamodel for the behavior trees modelling technique. En *Information Technology and Applications, ICITA 2005.*, vol. 1, páginas 35–39. IEEE, 2005.

GRANBERG, C. *David Perry on game design: a brainstorming toolbox*. Cengage Learning, 2014.

HAJEBI, K., ABBASI-YADKORI, Y., SHAHBAZI, H. y ZHANG, H. Fast approximate nearest-neighbor search with k-nearest neighbor graph. En *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, página 1312. 2011.

HALBERT, D. C. *Programming by example*. Tesis Doctoral, University of California, Berkeley, 1984.

HALL, M. A. y HOLMES, G. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data engineering*, vol. 15(6), páginas 1437–1447, 2003.

HARRIS, R. L. *Information graphics: A comprehensive illustrated reference*. Oxford University Press, 2000.

HU, D., GONG, Y., HANNAFORD, B. y SEIBEL, E. J. Semi-autonomous simulated brain tumor ablation with ravenii surgical robot using behavior tree. En *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, páginas 3868–3875. IEEE, 2015.

HUDSON'S, K. The AI of BioShock 2: Methods for iteration and innovation. En *Game Developers Conference*. 2010.

HUNICKE, R., LEBLANC, M. y ZUBEK, R. MDA: A formal approach to game design and game research. En *Proceedings of the AAAI Workshop on Challenges in Game AI*, vol. 4. 2004.

ISASI VIÑUELA, P. y GALVÁN LEÓN, I. Redes de neuronas artificiales. *Un Enfoque Práctico, Editorial Pearson Educación SA Madrid España*, 2004.

ISLA, D. Handling complexity in the Halo 2 AI. En *Game Developers Conference*. 2005.

ISLA, D. Halo 3 - building a better battle. En *Game Developers Conference*. 2008a.

ISLA, D. Halo 3 objective trees: A declarative approach to multi-agent coordination. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2008b.

JAIDEE, U., MUÑOZ-AVILA, H. y AHA, D. W. Case-based goal-driven coordination of multiple learning agents. En *Case-Based Reasoning Research and Development*, páginas 164–178. Springer, 2013.

JALLOV, D., RISI, S. y TOGELIUS, J. Evocommander: A novel game based on evolving and switching between artificial brains. *IEEE Transactions on Computational Intelligence and AI in Games*, 2016.

JOHN E. LAIRD, M. Machine learning for computer games. En *Game Developers Conference*. 2005.

KARPOV, I. V., SCHRUM, J. y MIIKKULAINEN, R. Believable bot navigation via playback of human traces. En *Believable Bots*, páginas 151–170. Springer, 2013.

KEITH, C. *Agile game development with Scrum*. Pearson Education, 2010a.

KEITH, C. The state of agile in the game industry. 2010b.

KELLEY, M. *No-Code Video Game Development Using Unity and Playmaker*. CRC Press, 2016.

KELLY, J.-P., BOTEA, A. y KOENIG, S. Planning with hierarchical task networks in video games. En *Proceedings of the ICAPS-07 Workshop on Planning in Games*. 2007.

KELLY JR, J. D. y DAVIS, L. Hybridizing the genetic algorithm and the K-Nearest Neighbors classification algorithm. En *ICGA*, páginas 377–383. 1991.

KITANO, H., ASADA, M., KUNIYOSHI, Y., NODA, I. y OSAWA, E. Robocup: The robot world cup initiative. En *Proceedings of the first international conference on Autonomous agents*, páginas 340–347. ACM, 1997.

KOLODNER, J. *Case-based reasoning*. Morgan Kaufmann, 2014.

KOUTONEN, J. y LEPPÄNEN, M. How are agile methods and practices deployed in video game development? A survey into finnish game studios. En *International Conference on Agile Software Development*, páginas 135–149. Springer, 2013.

LEMAITRE, J., LOURDEAUX, D. y CAROLINE, C. Towards a resource-based model of strategy to help designing opponent ai in rts games. En *7th International Conference on Agents and Artificial Intelligence (ICAART 2015)*, vol. 1, páginas 210–215. 2015.

LENAT, D. Learning from observation and discovery. *Machine Learning: An Artificial Intelligence Approach, Morgan Kaufmann, Los Altos, CA*, 1983.

LIM, C.-U., BAUMGARTEN, R. y COLTON, S. Evolving behaviour trees for the commercial game defcon. En *European Conference on the Applications of Evolutionary Computation*, páginas 100–110. Springer, 2010.

LLANSÓ GARCÍA, D. *Metodología ontológica para el desarrollo de videojuegos*. Tesis Doctoral, Universidad Complutense de Madrid, 2014.

LOIACONO, D., LANZI, P. L., TOGELIUS, J., ONIEVA, E., PELTA, D. A., BUTZ, M. V., LONNEKER, T. D., CARDAMONE, L., PEREZ, D., SÁEZ, Y. ET AL. The 2009 simulated car racing championship. *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2(2), páginas 131–147, 2010.

LOZANO-PEREZ, T. Robot programming. *Proceedings of the IEEE*, vol. 71(7), páginas 821–841, 1983.

MATICH, D. J. Redes neuronales: Conceptos básicos y aplicaciones. *Cátedra de Informática Aplicada a la Ingeniería de Procesos–Orientación I*, 2001.

MCGUIRE, M. y JENKINS, O. C. *Creating games: Mechanics, content, and technology*. CRC Press, 2008.

MILLER, P. Top 10 pitfalls using scrum methodology for video game development. 2008.

MITCHELL, M. *An introduction to genetic algorithms*. MIT press, 1998.

MIZUGUCHI, M., BUCHANAN, J. y CALVERT, T. Data driven motion transitions for interactive games. *Eurographics 2001 - Short Presentations*, 2001.

MOHOV, S. *Practical game design with Unity and Playmaker*. Packt Publishing Ltd, 2013.

MORIARTY, C. L. y GONZALEZ, A. J. Learning human behavior from observation for gaming applications. En *FLAIRS Conference*. 2009.

MUÑOZ, J., GUTIERREZ, G. y SANCHIS, A. Towards imitation of human driving style in car racing games. En *Believable Bots*, páginas 289–313. Springer, 2013.

NEIL, K. Game design tools: Time to evaluate. En *Proceedings of the DiGRA Nordic Conference*. 2012.

ONTAÑÓN, S., MISHRA, K., SUGANDH, N. y RAM, A. On-line case-based planning. *Computational Intelligence*, vol. 26(1), páginas 84–119, 2010.

ONTAÑÓN, S., MONTAÑA, J. L. y GONZALEZ, A. J. A dynamic-bayesian network framework for modeling and evaluating learning from observation. *Expert Systems with Applications*, vol. 41(11), páginas 5212–5226, 2014.

ONTAÑÓN, S. y RAM, A. Case-based reasoning and user-generated artificial intelligence for real-time strategy games. En *Artificial Intelligence for Computer Games*, páginas 103–124. Springer, 2011.

ORTEGA, J., SHAKER, N., TOGELIUS, J. y YANNAKAKIS, G. N. Imitating human playing styles in super mario bros. *Entertainment Computing*, vol. 4(2), páginas 93–104, 2013.

VAN OTTERLO, M. y WIERING, M. Reinforcement learning and markov decision processes. En *Reinforcement Learning*, páginas 3–42. Springer, 2012.

POMERLEAU, D. A. Alvinn, an autonomous land vehicle in a neural network. Informe técnico, Carnegie Mellon University, Computer Science Department, 1989.

POWERS, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2011.

QUINLAN, J. R. Simplifying decision trees. *International journal of man-machine studies*, vol. 27(3), páginas 221–234, 1987.

QUINLAN, J. R. *C4.5: programs for machine learning*. Elsevier, 2014.

RABIN, S. *Implementing a State Machine Language*, capítulo 3, páginas 314–320. AI Game Programming Wisdom. Cengage Learning, 2002.

RABINER, L. R. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, vol. 77(2), páginas 257–286, 1989.

RASMUSSEN, J. Are behavior trees a thing of the past. `http://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php`, 2016. Accessed 2017.02.27.

RIEDMILLER, M. y BRAUN, H. A direct adaptive method for faster backpropagation learning: The rprop algorithm. En *Neural Networks, 1993., IEEE International Conference on*, páginas 586–591. IEEE, 1993.

ROBERTSON, G. y WATSON, I. Building behavior trees from observations in real-time strategy games. En *Innovations in Intelligent SysTems and Applications (INISTA), 2015 International Symposium on*, páginas 1–7. IEEE, 2015.

ROKACH, L. y MAIMON, O. *Data mining with decision trees: theory and applications*. World scientific, 2014.

RUBIN, J. y WATSON, I. D. On combining decisions from multiple expert imitators for performance. *IJCAI-11* , 2011.

SAGREDO-OLIVENZA, I., FLÓREZ-PUGA, G., GÓMEZ-MARTÍN, M. y GONZÁLEZ-CALERO, P. UHotDraw: a GUI framework to simplify draw application development in Unity 3D. En *Actas del Primer Simposio Español de Entretenimiento Digital*, páginas 131–142. 2013.

SAGREDO-OLIVENZA, I., FLÓREZ-PUGA, G., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Supporting the construction of a GUI component for specifying the behavior of non-player characters in unity. *International Journal of Creative Interfaces and Computer Graphics (IJCICG)*, vol. 6(1), páginas 38–55, 2015a.

SAGREDO-OLIVENZA, I., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Un modelo integrador de máquinas de estados y árboles de comportamiento para videojuegos. En *CoSECivi*, páginas 185–198. 2014.

SAGREDO-OLIVENZA, I., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Supporting the collaboration between programmers and designers building game AI. En *Entertainment Computing - ICEC 2015* (editado por K. Chorianopoulos, M. Divitini, J. B. Hauge, L. Jaccheri y R. Malaka), vol. 9353 de *Information Systems and Applications, incl. Internet/Web, and*

HCI, páginas 496–500. Springer International Publishing, 2015b. ISBN 978-3-319-24588-1.

SAGREDO-OLIVENZA, I., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Programming by demonstration in a complex 3D game. En *CoSE-Civi*, páginas 101–112. 2016.

SAGREDO-OLIVENZA, I., GÓMEZ-MARTÍN, P. P., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Diseño de la experimentación para evaluar la efectividad de behavior bricks. En *Actas del X Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*, páginas 487–494. Centro de Estudios de Mérida, Universidad de Extremadura., 2015c. ISBN 978-84-697-2150-6.

SAGREDO-OLIVENZA, I., GÓMEZ-MARTÍN, P. P., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Using Program by Demonstration and visual scripting to supporting game design. En *The 30th International Conference on Industrial, Engineering, Other Applications of Applied Intelligent Systems (IEA/AIE 2017)*. 2017a.

SAGREDO-OLIVENZA, I., GÓMEZ MARTÍN, P. P., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Using program by demonstration and visual scripting to supporting game design. En *The 30th International Conference on Industrial, Engineering, Other Applications of Applied Intelligent Systems*. 2017b.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, vol. 3(3), páginas 210–229, 1959.

SCHELL, J. *The Art of Game Design: A book of lenses*. CRC Press, 2014.

SCHWABER, K. y BEEDLE, M. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edición, 2001. ISBN 0130676349.

SEGRE, A. M. *Machine learning of robot assembly plans*, vol. 51. Springer Science & Business Media, 2012.

STUTZ, M. Get started with gawk: Awk language fundamentals. 2006.

SUNSHINE-HILL, B. Using machine learning like a responsible adult. En *Game Developers Conference*. 2015.

VASUDEVAN, G. Advanced Behavior Tree Structures. `https://gamedevdaily.io/advanced-behavior-tree-structures-4b9dc0516f92`, 2015. [Online; accessed 25/03/2017].

WEBER, B. G., MATEAS, M. y JHALA, A. Building human-level AI for real-time strategy games. En *AAAI Fall Symposium: Advances in Cognitive Systems*, vol. 11. 2011.

WIEGERS, K. y BEATTY, J. *Software requirements*. Pearson Education, 2013.

WILLIAMS, D. y HINTON, G. Learning representations by back-propagating errors. *Nature*, vol. 323(6088), páginas 533–538, 1986.

WIRFS-BROCK, R. y WILKERSON, B. Object-oriented design: A responsibility-driven approach. En *ACM SIGPLAN Notices*, vol. 24, páginas 71–75. ACM, 1989.

YANNAKAKIS, G. N. Game AI revisited. En *Proceedings of the 9th conference on Computing Frontiers*, páginas 285–292. ACM, 2012.

YAO, X. Evolving artificial neural networks. *Proceedings of the IEEE*, vol. 87(9), páginas 1423–1447, 1999.