TRABAJO FINAL DE GRADO

Implementación de una inteligencia artificial basada en utilidad

Antoni Galmés Oliver
Grado en Diseño y Producción de Videojuegos

CURSO 2020-21

TecnoCampus
Escola Superior Politècnica

Centre adscrit a la

upf. Universitat Pompeu Fabra *Barcelona*

TecnoCampus

Centres universitaris adscrits a la

upf. Universitat Pompeu Fabra Barcelona

Grado en Diseño y Producción de Videojuegos

Implementación de una inteligencia artificial basada en utilidad en Unity Engine

Memoria Final

Antoni Galmés Oliver

Tutor: Dr. Enric Sesa i Nogueras

2020-2021

TecnoCampus Mataró-Maresme

Abstract

A library accompanied by a graphical interface that allows the creation and edition of utility-based artificial intelligence in Unity Engine. Throughout this work, the theoretical concepts that support the idea of usability as a decision-making mechanism are presented. In addition, the development of the library using the UnityEngine API is presented.

Resumen

Una librería acompañada de una interfaz gráfica que permite la creación y edición de inteligencia artificial basada en utilidad en *Unity Engine.* A lo largo de este trabajo se exponen los conceptos teóricos que sustentan la idea de la usabilidad como mecanismo de toma de decisiones. Además, se presenta el desarrollo de la librería usando la API de *UnityEngine*.

Resum

Una llibreria acompanyada d'una interfície gràfica que permet la creació i edició d'intel·ligència artificial basada en utilitat a *UnityEngine.* Al llarg d'aquest treball s'exposen els conceptes teòrics que sustenten la idea de la usabilitat com a mecanisme de presa de decisions. A més, es presenta el desenvolupament de la llibreria utilitzant l'API de *UnityEngine*.

Índice

Índice de figuras

Índice de Tablas

1. Introducción

La inteligencia artificial en los videojuegos abarca una gran cantidad de conceptos y técnicas que permiten crear comportamientos inteligentes. La toma de decisiones necesaria para determinar que comportamientos se deben realizar es un elemento clave. Existen una gran variedad de acercamientos, entre ellos las máquinas de estado, los árboles de comportamiento o sistemas basados en utilidad (entre otros). Cada acercamiento viene acompañado de una serie de ventajas y desventajas, y es a criterio del programador elegir que implementación realizar. (Millington y Funge, 2009).

El objetivo principal de este trabajo es realizar una librería que permita la implementación de una inteligencia artificial basada en utilidad en el motor gráfico *Unity* minimizando la necesidad de codificar. Esta librería ha de disponer de una interfaz gráfica la cual debe permitir la creación y edición de comportamientos.

Este trabajo se estructura en diferentes capítulos. En el capítulo 2 se presentan los objetivos principales y secundarios de este trabajo. El capítulo 3 se divide en 2 partes, la primera presenta al lector la inteligencia artificial aplicada a los videojuegos y los mecanismos de toma de decisiones no basados en utilidad más comunes. La segunda parte expone todos los conceptos teóricos necesarios para poder desarrollar el objeto de este trabajo. El capítulo 4 se realiza un estudio de diferentes obras, librerías y herramientas que implementan la utilidad. En el capítulo 5 se presenta el plan de trabajo, donde se exponen las diferentes fases, metodología y cronograma de este trabajo. En el capítulo 6 se expone el desarrollo del trabajo. Para finalizar, en el capítulo 7 se presentan los objetivos cumplidos y una valoración final del trabajo.

2. Objetivos

2.1. Objetivos principales

El primer objetivo es desarrollar una librería para el motor de videojuegos *Unity Engine* que permita la creación e implementación de comportamientos basados en utilidad.

El segundo objetivo es desarrollar una herramienta para Unity Engine que permita implementar y modificar los comportamientos minimizando la necesidad de codificar.

2.2. Objetivos secundarios

Como primer objetivo secundario se planea dotar a la herramienta de editor con una lista preestablecida de curvas de utilidad que se adapte a la necesidad de cada comportamiento.

En segundo lugar, se planea dotar a la librería con diferentes modificadores que permitan la implementación de *dual-utility reasoning* y *risk-takers*.

3. Marco teórico

En esta sección del trabajo se agrupan todos los conceptos teóricos necesarios para poder implementar correctamente el objeto del trabajo. En el primer apartado se expone el concepto de inteligencia artificial, su rol en los videojuegos y las implementaciones más utilizadas. En el segundo se profundiza en los componentes y la funcionalidad de la Utility-Based AI. Por último, se introducen dos variantes que permiten obtener diferentes resultados respecto a la implementación original.

3.1. Inteligencia artificial en los videojuegos

Esta sección del trabajo se divide en dos partes: la primera se centra en exponer diversos mecanismos de toma de decisiones que no se basan en la utilidad. Estos mecanismos solo se explicarán brevemente para que el lector pueda diferenciarlos correctamente. La segunda parte se centra en explicar detalladamente que es la inteligencia artificial basada en utilidad, sus componentes y su funcionamiento. Una vez se ha explicado detalladamente la inteligencia artificial basada en utilidad se van a exponer dos variantes de esta que permiten obtener diferentes resultados.

La inteligencia artificial es un campo muy amplio que abarca diferentes áreas de estudio como el *machine learning, deep learing* o la simulación de comportamientos. En este trabajo solo se explica la inteligencia artificial en el área de los videojuegos.

Dill (2013), expone que el principal objetivo de la inteligencia artificial en los videojuegos es la de crear una experiencia convincente para el jugador a partir de comportamientos que den la ilusión de inteligencia propia. El autor argumenta que el jugador va a suspender su incredulidad para poder participar en la experiencia que ofrece el juego y que por lo tanto es importante crear comportamientos convincentes que le permitan mantener esa suspensión de la incredulidad.

Para que una inteligencia artificial sea convincente tiene que ser reactiva y no determinista. Los agentes deben reaccionar a los estímulos del jugador o del propio ambiente del juego a partir de unas normas que define el diseñador del juego con el objetivo de crear una experiencia. (Dill, 2013).

“Es importante mantener control autoritario para asegurar que el autor de la inteligencia artificial puede modificarla para asegurarse de que se consigue la experiencia deseada” (Dill, 2013, pág. 5). Para poder mantener un control autoritario sobre la inteligencia artificial, el autor defiende que es importante mantener la simplicidad y escalabilidad para que el diseñador pueda manejar todas las situaciones posibles.

Uniendo las ideas anteriores se puede concluir que una inteligencia artificial en los videojuegos tiene que ser reactiva y no determinista mientras se mantiene un control autoritario para que el diseñador pueda crear una experiencia que se adecúe al juego.

3.1.1. Mecanismos de toma de decisiones

En esta sección se exponen diferentes mecanismos de toma de decisiones no basados en utilidad. El objetivo de su explicación es poner al lector en contexto y que pueda diferenciarlos entre ellos. Ninguno de estos mecanismos son el objeto de este trabajo por lo que su explicación es breve y meramente orientativa.

3.1.1.1. Árboles de decision

Millington y Funge (2009) definen los árboles de decision (del inglés *decision trees*) como una estructura formada por puntos de decisión conectados, también llamados nodos. Los autores exponen que estos puntos de decisión solo revisan una simple variable y devuelven *True* o *False.* Esto implica que las decisiones no pueden estar compuestas por diferentes variables unidas por operadores booleanos (AND y OR).

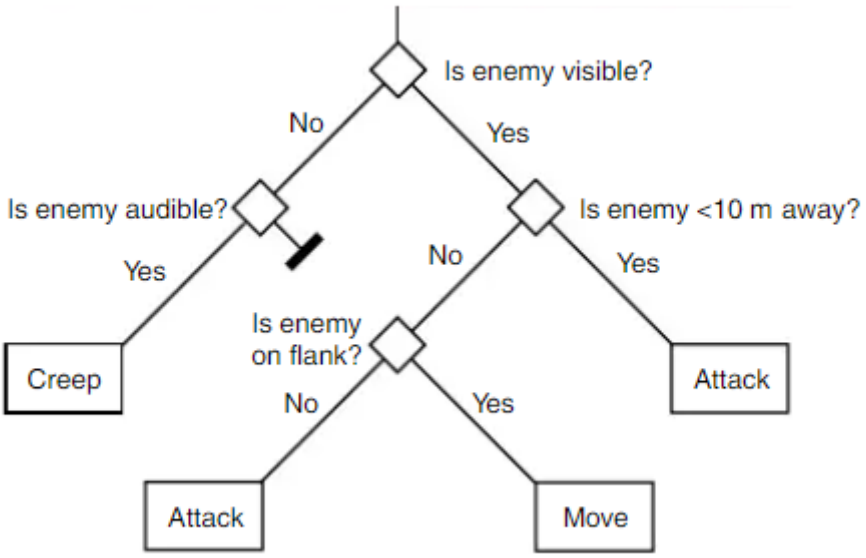

Figura 3.1 Estructura de un árbol de decisión. Fuente: Millington y Funge, 2009.

Como se puede observar en la Fig. 3.1, todos los puntos de decisión parten de un nodo inicial llamado nodo raíz. “El algoritmo continúa a lo largo del árbol, tomando decisiones en cada nodo hasta que no quedan más decisiones por tomar (…) Cuando el algoritmo llega a una acción, la ejecuta de inmediato” (Millington y Funge, 2009, pág. 296).

3.1.1.2. Máquinas de Estados Finitas

Rabin (2002) define una máquina de estados finita (del inglés *Finite State Machine*) como un sistema basado en reglas donde un numero finito de estados están conectados entre sí mediante transiciones. Además, añade que una máquina de estados tan solo puede ocupar un estado a la vez.

Millington y Funge (2009) exponen que las máquinas de estados están formadas principalmente dos elementos: los estados y las transiciones. “En cada iteración (normalmente en cada fotograma), se llama a la función *update* de la máquina de estados” (Millington y Funge, 2009, pág. 311).

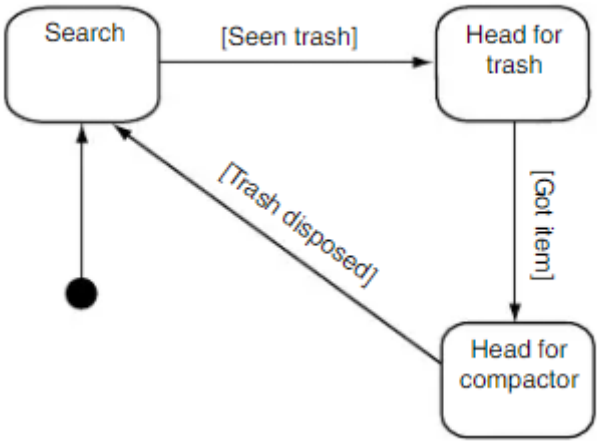

Figura 3.2 Ejemplo de una Finite State Machine. Fuente: Millington y Funge, 2009.

Como se puede observar en la Fig. 3.2, los diferentes estados (*Search, Head for trash* y *Head for compactor*) están unidos por transiciones. Las transiciones se ejecutan cuando se ha cumplido la lógica necesaria (*Seen trash*, *Got item* y *Trash disposed*). Un elemento esencial de las máquinas de estados es el punto de entrada, en la Fig. 3.2 está representado mediante un punto negro. La función del punto de entrada es determinar el estado inicial de la máquina de estados. Como se puede observar, la transición que une el punto de entrada con el estado *Search*, no contiene ninguna lógica, por lo tanto, esta transición se ejecuta en la primera iteración y establece el estado *Search* como el estado actual. (Millington y Funge, 2009).

Aunque el concepto sea claro y simple, Millington y Funge (2009) afirman que las máquinas de estados son difíciles de gestionar cuando hay una gran cantidad de estados debido a que el número de transiciones necesarias para conectar los estados crece significativamente.

3.1.1.3. Máquinas de Estados Jerárquicas

Las máquinas de estado jerárquicas (del inglés *Hierarchical Finite State Machines*) usan el mismo concepto explicado anteriormente: El uso de estados y transiciones. Millington y Funge (2009) exponen que una máquina de estados jerárquica está compuesta por diferentes niveles (jerarquías) donde cada nivel

representa una máquina de estados finita y estos se procesan mediante una función recursiva que recorre toda la jerarquía.

Millington y Funge (2009), explican que el algoritmo empieza a procesar la máquina de estados de nivel superior y si el estado elegido es una máquina de estados, el algoritmo empieza a procesar el nivel inferior.

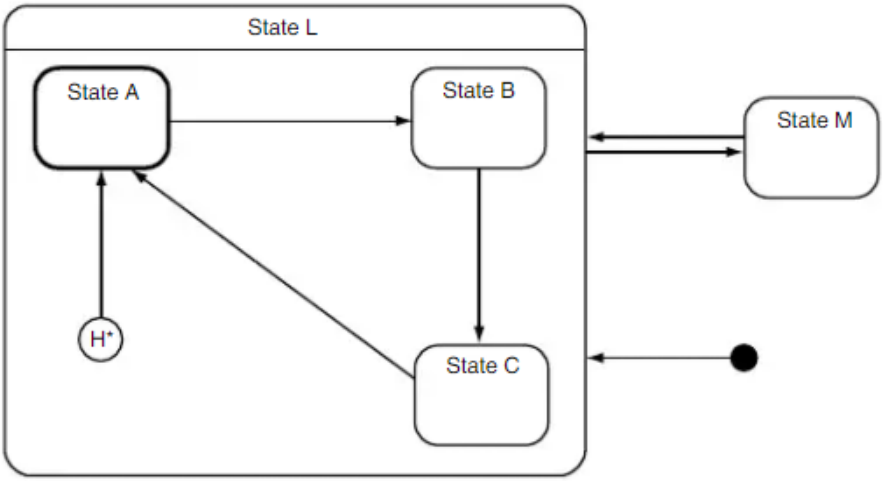

Figura 3.3 Ejemplo de una Hierarchical Finite State Machine. Fuente: Millington y Funge, 2009.

En la Fig. 3.3 se puede observar una máquina de estados jerárquica de 2 niveles. El nivel superior es el compuesto por los estados L y M, el estado M representa una acción mientras que el estado L representa una máquina de estados de nivel inferior compuesta por los estados A, B y C. El proceso de selección de los estados y la transición entre ellos es el mismo que con las máquinas de estado finitas. Como se puede observar, ambos niveles tienen un punto de entrada, el de la máquina de estados de nivel superior está representado como un punto negro, mientras que el la de nivel inferior está representado por el punto H*. Este segundo punto determina que una vez se ha accedido a la máquina de estados de nivel inferior, el primer estado que se ejecuta es el estado A.

Millington y Funge (2009) defienden que las máquinas de estado jerárquicas ofrecen una buena solución al problema de gestión mencionado en el apartado anterior, ya que el uso de una jerarquía reduce considerablemente el número de transiciones.

3.1.1.4. Árboles de Comportamiento

Los árboles de comportamiento (del inglés *Behavior Trees*) son una estructura en forma de árbol inverso. Esta estructura está formada por nodos, los cuales se distinguen en dos categorías, los nodos de control de flujo (nodos internos) y los nodos de ejecución (nodos externos). La diferencia entre un nodo interno y un nodo externo es que un nodo interno tiene al menos un nodo hijo, mientras que los nodos externos no tienen. (Colledanchise y Ögren, 2018).

La ejecución de los árboles de comportamiento se basa en ticks. Este tick es una señal que se genera en el nodo raíz y va navegando de nodo en nodo. Un nodo solo se ejecuta si recibe el tick. (Colledanchise y Ögren, 2018).

Según Colledanchise y Ögren (2018) los nodos de control de flujo se distinguen entre: selector, secuencia, paralelo y decorador. Por otro lado, los nodos de ejecución se distinguen entre: condición y acción. La diferencia entre los diferentes tipos de nodos de control de flujo es el valor que se devuelve según la ejecución de los nodos hijos. Por otro lado, la función de los nodos de ejecución es o bien comprobar una condición (nodo de condición, devuelve: *success* o *failure*) o realizar una acción y devolver el resultado (nodo de ejecución, devuelve: *running, success* o *failure*). (Colledanchise y Ögren, 2018).

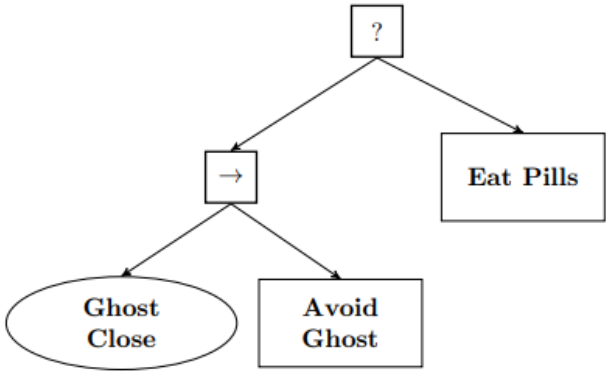

Figura 3.4 Árbol de comportamiento de PAC-MAN. Fuente: Colledanchise y Ögren, 2018.

En la Fig. 3.4 se puede observar un árbol de comportamiento el cual está formado por dos nodos de control de flujo: Un selector, representado como [?] y

una secuencia, representada como [→]. Y tres nodos de ejecución, los cuales se distinguen entre un nodo condición (*Ghost Close*) y dos nodos de acción (*Avoid Ghost* y *Eat Pills*).

La ejecución de un árbol de comportamiento se realiza de izquierda a derecha. En el caso de la Fig. 3.4, el tick ejecuta el nodo *Ghost Close,* ya que es el que está más a la izquierda. Este nodo es hijo de un nodo secuencia, el cual devuelve *success* si todos los nodos hijos se ejecuten con éxito. Independientemente de si el nodo *Ghost Close* devuelve *success* o *failure,* el nodo *Avoid Ghost* se va a ejecutar. El nodo *Eat Pills* solo se ejecuta si el su nodo hermano (la secuencia mencionada anteriormente) devuelve *success*, ya que su nodo padre es un selector, el cual solo ejecuta el siguiente nodo si el anterior devuelve *success*. Al no haber más nodos a la derecha de *Eat Pills*, el selector se ejecutará continuamente hasta que devuelva *success*. (Colledanchise y Ögren, 2018).

3.2. Utility-Based AI

El objetivo de esta sección es presentar al lector la inteligencia artificial basada en utilidad. En primer lugar, se tratará la teoría de la utilidad y su aplicación en los videojuegos. En segundo lugar, se entrará más en detalle en el funcionamiento de la inteligencia artificial basada en utilidad y sus componentes. Finalmente se explicarán los conceptos risk-takers y dual-utility, los cuales son variaciones de la implementación original.

3.2.1. Teoría de la Utilidad

Graham (2013) define la teoría de la utilidad como que toda acción o estado posible en un modelo puede ser descrito con un único valor uniforme. Este valor, normalmente referido como utilidad, describe lo útil que puede ser una acción en un contexto definido. El autor remarca que es importante diferenciar entre la utilidad y el valor de una acción o estado. La teoría de la utilidad es especialmente útil cuando es necesario comparar dos cosas que, a priori, no son directamente comparables.

Para poder comparar acciones o estados que no son directamente comparables se usa la utilidad de dicha acción. Este valor indica que conveniente es realizar una acción en una situación concreta. Según Graham (2013), el uso de valores normalizados (valor de 0 a 1) es una buena práctica ya que son fácilmente comparables entre ellos, además de que es fácil identificar a primera vista la utilidad de una acción. Una acción con una utilidad de 0 tiene utilidad nula mientras que una utilidad de 1 indica que esa acción tiene utilidad máxima.

Juan y Pablo van de viaje de Barcelona a Paris. Tienen dos opciones: ir en autobús o ir en avión. El precio para ir en autobús es de 30€, mientras que ir en avión cuesta 200€. El tiempo de viaje es de 15h en autobús mientras que en avión son 2h. Juan es un empresario que va de viaje de negocios, mientras que Pablo es un estudiante que va a visitar a unos amigos. Juan cobra 20€ por cada hora de trabajo, mientras que Pablo no tiene trabajo.

Como se puede observar, Juan es una persona que valora mucho su tiempo ya que cada hora que no trabaja, son 20€ que deja de ganar, mientras que Pablo valora mucho el dinero ya que no tiene ninguna fuente de ingresos.

Siguiendo la teoría de la utilidad, se puede observar que para Juan un viaje rápido es mucho más útil (mayor utilidad) aunque sea más caro. Mientras que para Pablo un viaje más barato tiene mayor utilidad.

En el ejemplo anterior se puede observar la diferencia entre la utilidad y el valor de una opción. La utilidad de cada opción depende de los criterios de cada agente (Juan y Pablo), mientras que la opción más valiosa es el viaje en avión.

3.2.2. Utility-based AI en los videojuegos

Graham (2013) expone que la teoría de la utilidad ya existía antes que los videojuegos o incluso que los ordenadores. El mismo concepto ha sido trasladado a la inteligencia artificial como mecanismo de toma de decisiones. “Uno de los sistemas más robustos y poderosos que nos hemos encontrado es un sistema basado en utilidad” (Graham, 2013, pág. 113).

Además, el uso de una inteligencia artificial basada en utilidad aporta múltiples beneficios comparado con los mecanismos expuestos en el apartado 3.1.1. “Es extremadamente rápido (…) y escala muy bien. Uno de los grandes atractivos de este sistema es la cantidad de comportamientos emergentes que se pueden obtener con sólo unos pocos valores.” (Graham, 2013, pág. 125).

Para poder observar la posibilidad de crear comportamientos emergentes véase el siguiente ejemplo expuesto por Dill (2012):

> Imagine una inteligencia artificial para un personaje que se encuentra en un combate. En este mismo momento los enemigos están disparando, y en el suelo hay una granada y un compañero herido. Una máquina de estados finita siempre seguirá el mismo orden (definido por los estados y transiciones). En esta situación concreta la máquina de estados siempre elige coger la granada y lanzarla a los enemigos.
>
> En cambio, una inteligencia artificial basada en utilidad evalúa todas las posibles acciones a realizar. Dependiendo de la distancia de la granada, la gravedad de las heridas del compañero o encontrar un sitio para cubrirse, la acción a realizar es una u otra. Si la granada está justo a los pies del personaje, este la lanzará ya que la utilidad es muy alta, en cambio, si está a varios metros de distancia, exponerse al fuego enemigo para lanzar la granada no es la mejor opción (poca utilidad). Si las heridas del compañero no son graves y puede moverse, la mejor opción con mayor utilidad es ayudar al compañero.

Como se puede observar en el ejemplo anterior, una inteligencia artificial basada en utilidad permite obtener diferentes comportamientos según la situación en la que se encuentra el agente sin la necesidad de definir todas las situaciones posibles.

3.2.3. Componentes y funcionamiento

3.2.3.1. Acciones

Russell y Norvig (1995) definen las acciones de un agente como los posibles comportamientos que puede realizar un agente. Cada posible acción está compuesta por una o más consideraciones.

3.2.3.2. Consideraciones

Las consideraciones son los factores de decisión que se tienen en cuenta a la hora de elegir una acción. La decisión de realizar cierta acción raramente se basa en una sola consideración, sino que puede ser influenciada por diferentes consideraciones. En el caso de que se utilicen múltiples consideraciones, se asigna un peso a cada una. La función de este peso es determinar el grado de influencia de dicha consideración. (Graham, 2013).

La función de las consideraciones es calcular la utilidad de cada factor que influencia la elección de cierta acción. Este cálculo se obtiene a partir de las curvas de utilidad, las cuales se explican en el apartado 3.2.3.4. Al promediar y normalizar la utilidad de todas las consideraciones se obtiene la utilidad total de dicha acción. (Graham, 2013).

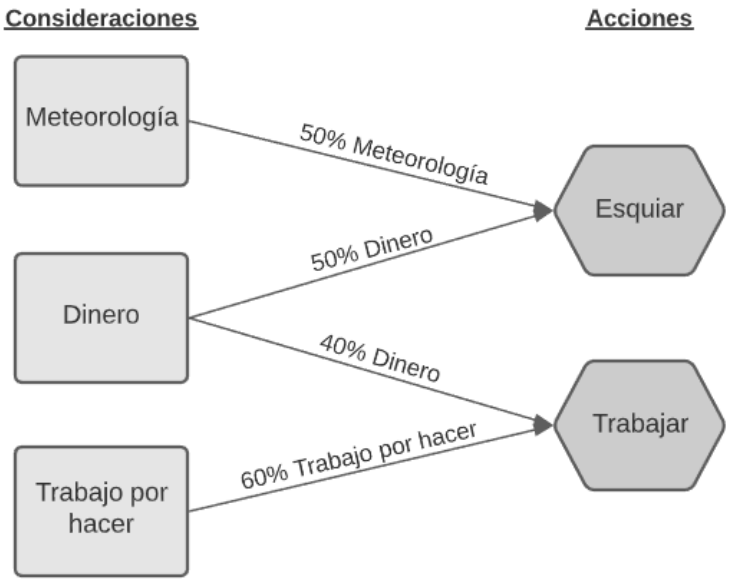

Figura 3.5 Ejemplo de uso de consideraciones. Fuente: Elaboración propia.

En la Fig. 3.5 se puede observar el proceso que se sigue para calcular la utilidad mediante consideraciones. En este ejemplo, Esquiar y Trabajar son las dos posibles acciones que se pueden realizar. La acción Esquiar está formada por las consideraciones Meteorología y Dinero, mientras que la acción Trabajar está formada por las consideraciones Dinero y Trabajo por hacer.

Tal y como se puede observar en la Fig. 3.5, el peso de las consideraciones Meteorología y Dinero es el mismo, ya que se considera que tienen la misma importancia a la hora de considerar la acción Esquiar. En cambio, a la hora de considerar la acción Trabajar, se le da más importancia a la consideración Trabajo por hacer que a la consideración Dinero.

3.2.3.3. Utilidad

La función de la utilidad es representar con un valor numérico la conveniencia de realizar una acción en cualquier contexto. Como se ha expuesto anteriormente, se recomienda el uso de un valor normalizado (de 0 a 1) para expresar la utilidad de la acción.

La utilidad de cada acción está representada per un único valor, el cual varía según las necesidades del agente dependiendo de la curva de utilidad, la cual se explica en el siguiente apartado.

Dill (2015) expone que existen dos aproximaciones a la hora de usar la utilidad para tomar una decisión. La primera es la utilidad absoluta, en la cual se analizan todas las acciones que se pueden realizar y se elige la que tiene mayor utilidad. La segunda, la utilidad relativa, se elige la acción aleatoriamente, pero la probabilidad de elegir la acción (P_A) se determina dividiendo la utilidad de la acción (U_A) entre la utilidad de todas acciones, tal y como se ve en (3.1). Por lo tanto, mayor es la utilidad de la acción, más probable es que se elija.

$$P_A = \frac{U_A}{\sum_{i=1}^{n} U_i} \tag{3.1}$$

3.2.3.4. Curvas de Utilidad

Graham (2013) define las curvas de utilidad como la relación entre un valor arbitrario del juego y su utilidad. Esta relación es la que permite obtener la utilidad a partir de una variable del juego. “Las curvas pueden ser pensadas como un proceso de conversión, donde se convierten uno o más variables del juego a utilidad” (Graham, 2013, pág. 117).

Cada consideración tiene asignada una curva de utilidad la cual se encarga de convertir una variable a utilidad. La utilidad obtenida se representa en el eje Y, mientras que el valor de la variable se representa en el eje X. (Graham, 2013).

Existe una gran variedad de curvas las cuales representan diferentes grados de crecimiento de la utilidad respecto a la variable del juego. La elección de estas está totalmente sujeta al comportamiento que se quiere obtener. (Graham, 2013).

A continuación, se expondrán la tipología de curvas más comunes.

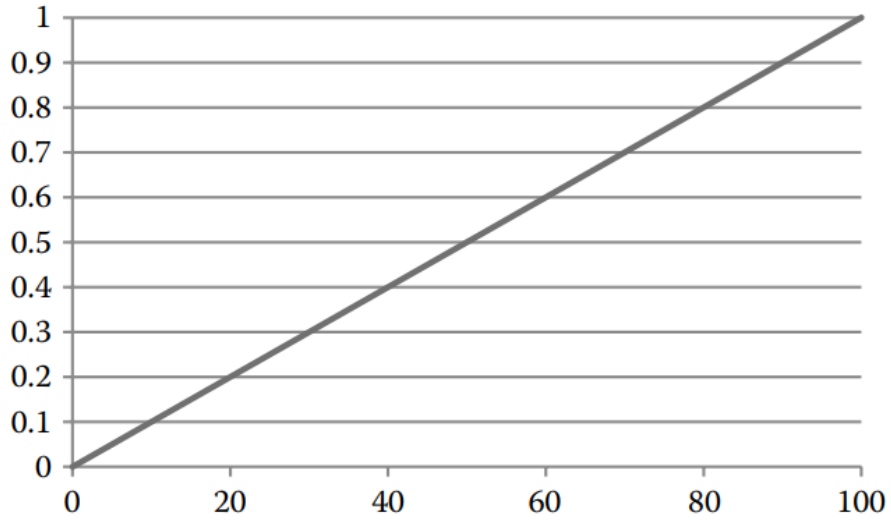

Figura 3.6 Ejemplo de curva lineal. Fuente: Graham, 2013.

$$U = x/m \tag{3.2}$$

En la Fig. 3.6 se puede observar una curva lineal. La utilidad obtenida a partir de esta curva crece o decrece de forma constante. Como se puede observar en

(3.2), la constante *m* determina el grado de inclinación de la curva, y por lo tanto su crecimiento. Si el valor de *m* es negativo, la utilidad obtenida decrece a medida que el valor de la variable crece.

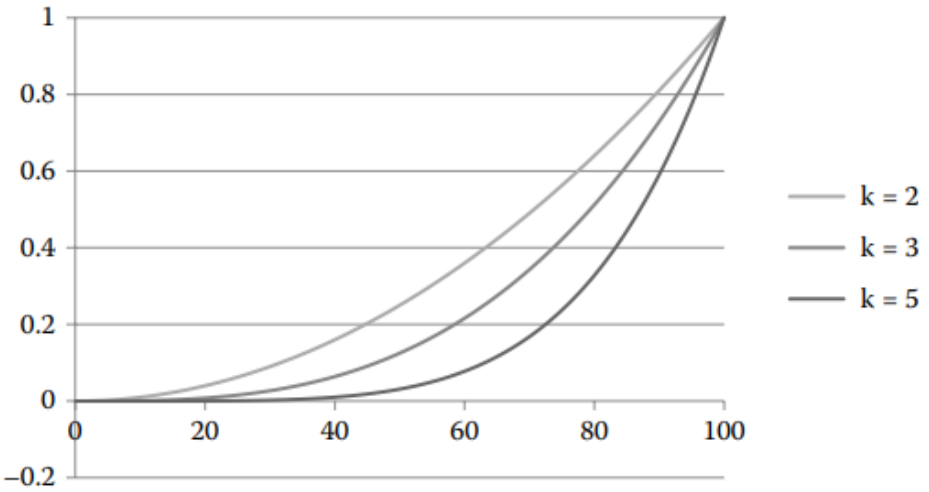

Figura 3.7 Ejemplo de una curva exponencial. Fuente: Graham, 2013.

$$U = \left(\frac{x}{m}\right)^{k} \tag{3.3}$$

En la Fig. 3.7 se pueden observar tres curvas de utilidad de carácter exponencial, resultado de (3.3). En esta curva, la utilidad (U) crece a medida que el valor de la variable del juego (X) aumenta. El exponente *K* determina el grado de crecimiento.

La diferencia respecto a curva representada en la Fig. 3.6 es que el ritmo de crecimiento de la utilidad no es constante. Véase el siguiente ejemplo:

La curva obtenida a partir de $k = 5$ en la Fig. 3.6 se usa para obtener la utilidad de realizar la acción IR A COMER. El eje Y representa la utilidad obtenida y el eje X representa el valor de la variable HAMBRE.

El valor de HAMBRE incrementa 1 unidad por minuto y se establece a 0 cada vez que el personaje realiza la acción IR A COMER.

Como se puede observar en el ejemplo anterior, hasta que el valor de HAMBRE no sobrepasa un valor determinado (en este caso 60), la utilidad de realizar la acción IR A COMER es muy baja. Pero a partir de 70, la utilidad empieza a crecer mucho más rápido.

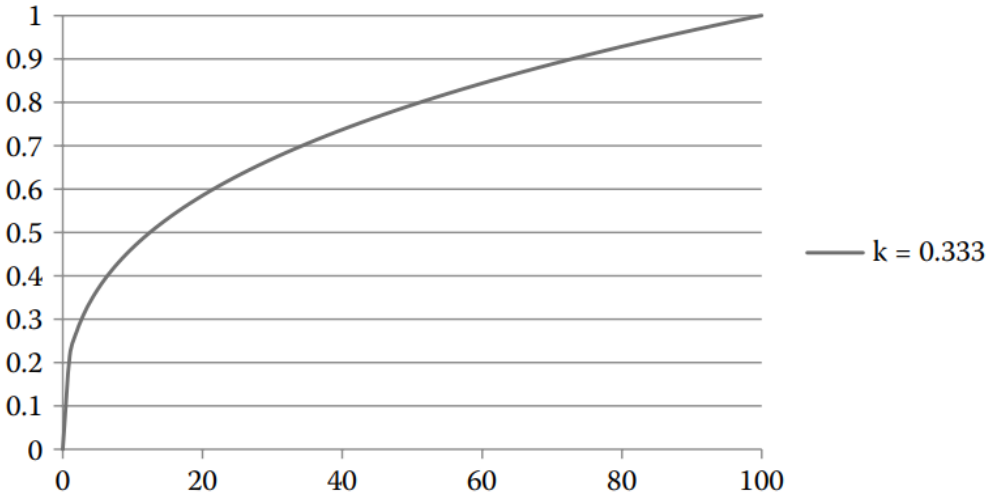

Figura 3.8 Ejemplo de curva exponencial inversa. Fuente: Graham, 2013.

En la Fig. 3.8 se puede otro ejemplo de curva. Esta curva también es resultado de (3.3), lo que la diferencia de la Fig. 3.7 es el valor de *k.* Cuando *k* obtiene un valor inferior a 1, se obtiene el efecto inverso a la Fig. 3.7. El crecimiento de la utilidad es mucho más pronunciado al principio y a medida que el valor del eje X aumenta, el crecimiento es cada vez menos pronunciado. En el siguiente ejemplo se propone un uso de una curva cuadrática rotada:

El eje Y de la Fig. 3.8 representa la utilidad de realizar la acción ATACAR CON RIFLE, mientras que el eje X representa la distancia a la que se encuentra el enemigo.

Como se puede observar, si el enemigo se encuentra a menos de 1 metro de distancia, la utilidad de ATACAR CON RIFLE es mínima, mientras que, a mayor distancia, mayor es la utilidad de dicha acción.

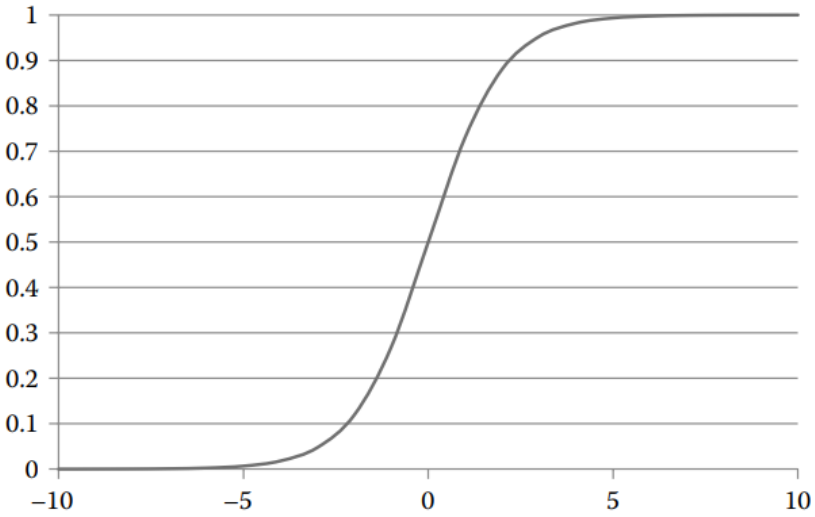

Figura 3.9 Ejemplo de curva logística. Fuente: Graham, 2013.

$$U = \frac{1}{1 + e^{-x}} \tag{3.4}$$

En la Fig. 3.9 se puede observar una curva logística. Este tipo de curva se obtiene a partir de una función sigmoidal, tal y como se puede apreciar en (3.4). La principal característica de las funciones curvas logísticas es que se produce un cambio agresivo del valor del eje Y (la utilidad obtenida) en el centro del eje X, mientras que en los límites apenas se produce un cambio.

Como se puede ver en (3.4), se usa la constante *e* (número de Euler) como base de la exponencial. Este valor se puede ajustar según la pendiente que se quiera obtener, cuánto más alto es el número, más agresiva es la pendiente y, por lo tanto, más repentino es el cambio. (Graham, 2013).

El uso más apto de la curva logística es cuando se quiere obtener un repentino cambio de utilidad una vez alcanzado cierto valor.

En ocasiones una fórmula matemática no es suficiente para expresar la curva debido a la poca flexibilidad que ofrecen para ajustarla y obtener la utilidad deseada. Para solventar este problema se propone el uso de curvas personalizadas. (Graham, 2013).

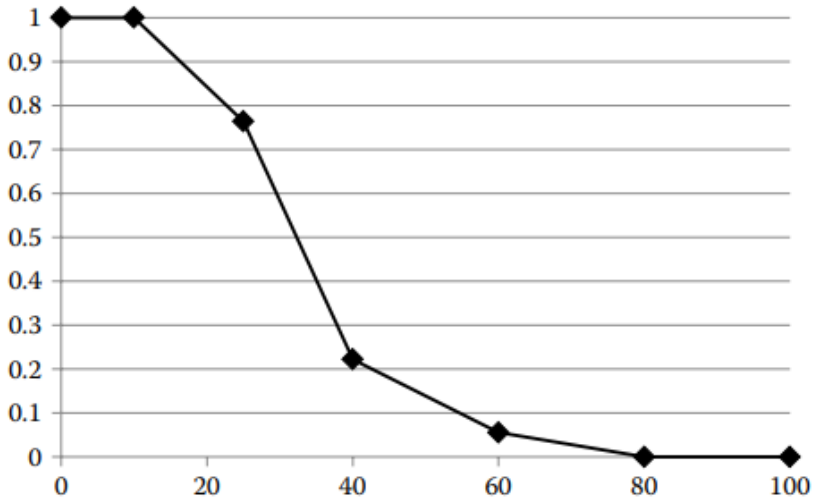

Figura 3.10 Ejemplo de curva por tramos. Fuente: Graham, 2013.

En la Fig. 3.10 se puede observar una curva creada a partir de tramos. Estos tramos se controlan a partir de nodos. En la Fig. 3.10 todos los tramos son lineales, sin embargo, estos pueden ser personalizados para representar diferentes tipos de curvas. La ventaja de usar una curva por tramos es que permite personalizar al máximo la utilidad obtenida. (Graham, 2013).

Aunque la Fig. 3.10 se asimila a una curva logística, se puede apreciar que en los límites de la curva (valores del 0 a 10 y del 80 al 100) la pendiente de la curva es 0 (la utilidad no varía) mientras que, en la curva logística, la pendiente no es totalmente 0. Por otro lado, la zona con mayor cambio de utilidad ya no está centrada en el gráfico, sino que está levemente desplazada.

Como se puede apreciar, hay una gran variedad de curvas, cada una con un uso propio. Graham (2013) concluye que la mejor opción es dotar al diseñador de una herramienta que le permita editar con total libertad las curvas de utilidad y, en consecuencia, obtener los comportamientos deseados.

3.2.3.5. Funcionamiento

En este apartado se explica el funcionamiento de la inteligencia basada en utilidad a través de un ejemplo.

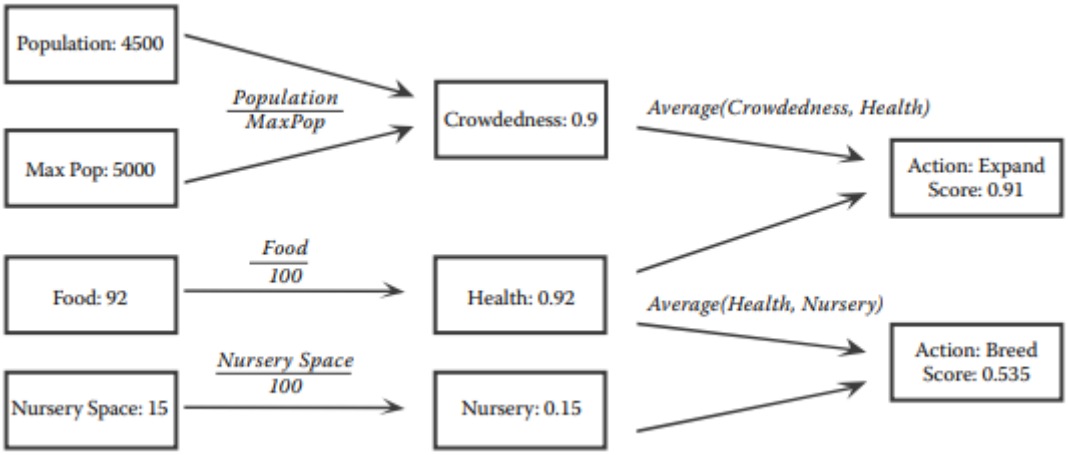

Figura 3.11 Ejemplo de un mecanismo basado en utilidad. Fuente: Graham, 2013.

En la Fig. 3.11 se pueden apreciar tres columnas. La columna de la izquierda contiene las diferentes variables del juego, la columna central contiene las consideraciones a tener en cuenta y la columna de la derecha contiene las posibles acciones a realizar. Como se puede observar, la columna de variables y la columna de consideraciones se relacionan a partir de diferentes fórmulas, estas fórmulas son la representación matemática de la curva de utilidad de cada consideración. Por otra parte, la relación que une las acciones y consideraciones es el peso de la utilidad obtenida de dicha consideración, ya que cada acción tiene más de una consideración en cuenta.

El funcionamiento es el siguiente: En primer lugar, cada consideración debe calcular la utilidad a partir de su curva. En el caso de la consideración *Crowdedness* se calcula a partir de (3.5).

$$Crowdedness = \frac{Population}{MaxPop} \tag{3.5}$$

Como se puede observar en la Fig. 3.11, la consideración *Crowdedness* tiene en cuenta múltiples variables, mientras que la consideración *Health* y *Nursery* calculan la utilidad a partir de una sola variable.

En segundo lugar, cada acción debe calcular la utilidad a partir de los pesos de cada consideración. En este ejemplo, todas las consideraciones tienen el mismo

peso, por lo tanto, se calcula la media aritmética de las consideraciones de cada acción. En el caso de la acción *Expand*, la utilidad de dicha acción se calcula a partir de (3.6)

$$U = \frac{Crowdedness + Health}{2} \tag{3.6}$$

Una vez se ha calculado la utilidad de cada acción, se elige la que tiene mayor utilidad. El proceso de calcular la utilidad de cada acción debe realizarse en cada fotograma, ya que las variables del juego están en constante cambio, por lo tanto, la utilidad de cada consideración varía. (Dill, Ray, Garrity y Fragomeni, 2012).

3.2.4. Variantes de la Utility-based AI

En esta sección se van a explicar dos variantes de la inteligencia artificial basada en utilidad. Ambas variantes parten de los mismos conceptos explicados anteriormente, pero introducen cambios con el objetivo de obtener diferentes resultados comparados con la implementación original.

Antes de exponer estas dos variantes, es necesario explicar los conceptos de utilidad absoluta y utilidad relativa, ya que las variantes que se explican a continuación utilizan estos dos conceptos.

Un agente que implementa la utilidad absoluta siempre escoge la opción con mayor utilidad de todas las existentes. Como se puede apreciar, el concepto de utilidad absoluta es el que se aplica en la implementación original. (Dill, 2015).

En cambio, en el acercamiento de la utilidad relativa se elige la acción de manera aleatoria. La utilidad de cada acción define la probabilidad de que esta sea elegida, por lo tanto, mayor es la utilidad, mayor es la probabilidad de que dicha acción sea elegida. (Dill, 2015).

3.2.4.1. Risk-takers

La variante de *risk-takers* es la que implementa la utilidad relativa. El objetivo de esta variante es reducir la predictibilidad de un agente basado en utilidad. (Dill, 2015).

Dada una situación concreta, un agente que implementa la utilidad absoluta siempre elige la acción de mayor utilidad, por lo tanto, siempre que se repita esta situación, la acción elegida siempre es la misma. Al introducir la aleatoriedad a la hora de elegir que acción realizar, se reduce la predictibilidad. (Dill, 2015).

Lo que diferencia la variante *risk-takers* de un mecanismo de toma de decisiones totalmente aleatorio es que las diferentes acciones a realizar tienen una probabilidad distinta, teniendo mayor probabilidad la acción más útil. Esto reduce la probabilidad de elegir acciones poco útiles.

Sin embargo, aunque se tienda a elegir una acción adecuada, al añadir aleatoriedad a la toma de decisiones, existe la posibilidad de que se elija una acción con baja utilidad. (Dill, 2015).

3.2.4.2. Dual-Utility reasoning

El objetivo de la variante que se expone en este apartado es solventar los problemas que presentan la utilidad absoluta y la utilidad relativa por separado.

Dill (2015) propone el concepto *Dual-Utility Reasoning,* el cual combina la utilidad absoluta con la utilidad relativa. En este acercamiento se asignan dos valores utilitarios a cada acción en vez de tan sólo utilizar la utilidad. Estos dos valores son: rango y peso. (Dill, 2015).

El rango se utiliza para separar las acciones en diferentes categorías dependiendo de su utilidad. Por otra parte, el peso de cada acción se define a partir de la utilidad relativa a la categoría. (Dill, 2015).

El funcionamiento de esta variante es el siguiente: En primer lugar, se eliminan todas las acciones de utilidad 0. En segundo lugar, se elige la categoría con mayor rango y se eliminan todas las otras categorías. Una vez se ha

seleccionado la categoría con el mayor rango, se eliminan todas las acciones con una utilidad mucho menor (peores acciones) que el resto. Finalmente se selecciona una acción de las restantes mediante el acercamiento de utilidad relativa. (Dill, 2015).

Como se puede apreciar, con esta variante se consigue eliminar la posibilidad de realizar acciones no útiles, a la vez que se reduce la predictibilidad de la inteligencia artificial.

4. Análisis de Referentes

En esta sección se analizan varios videojuegos y herramientas que se consideran referentes debido a su relación con la inteligencia artificial basada en utilidad. En el primer apartado se exponen videojuegos los cuales implementan una IA basada en utilidad. El segundo apartado se centra en analizar diferentes herramientas que permiten implementar mecanismos de toma de decisiones basados en utilidad.

4.1. Videojuegos

El objetivo de esta sección es analizar la implementación de los mecanismos basados en utilidad de diferentes obras. El análisis se realiza a partir de la información que se ha compartido públicamente por la compañía o los desarrolladores.

4.1.1. Dragon Age: Inquisition (BioWare, 2014)

Dragon Age: Inquisition es un videojuego de acción RPG desarrollado por *BioWare* (2014). Este juego es el tercero de la franquicia *Dragon Age*. Los jugadores controlan principalmente al Inquisidor (el protagonista) o sus compañeros en una serie de mundos semiabiertos donde el jugador puede interactuar con una gran variedad de NPCs (*non-playable characters*), los cuales pueden ser reclutados como compañeros.

Figura 4.1 Imagen *in-game* de *Dragon Age: Inquisition*. Fuente: Kevin VanOrd, 2014.

Hanlon y Watts (2017) recogen que *BioWare* implementó la inteligencia artificial a partir de un mecanismo basado en utilidad ya que la IA debía cumplir las siguientes reglas:

1. En cualquier momento, hay una serie de acciones que el agente puede realizar.
2. El agente solo puede realizar una acción a la vez.
3. Las diferentes acciones tienen valores utilitarios; algunas acciones son más útiles que las otras.
4. Debe ser posible cuantificar la utilidad de cada acción.

Como se puede apreciar, estas 4 reglas se adaptan a un mecanismo de toma de decisiones basado en utilidad. (Hanlon y Watts, 2017).

En *Dragon Age: Inquisition* se categoriza las acciones en 4 diferentes categorías dependiendo de utilidad, cual va de 0 a 70. Las categorías son las siguientes:

Tipo de acción	Utilidad	Descripción
Básica	0-10	Acción preferible que no hacer nada.
Ofensiva	20-40	Acción preferible que una acción básica.
Defensiva	25-45	Acción preferible que una acción ofensiva del mismo valor.
Reacción	50-70	Todas las acciones en esta categoría se deben ejecutar inmediatamente.

Tabla 4.1 Categorización de las acciones en *Dragon Age: Inquisition*. Fuente: Hanlon y Watts, 2017.

Como se puede apreciar en la Tabla 4.1, *Dragon Age: Inquisition* utiliza la variante *Dual-Utility Reasoning*, explicada en el apartado 3.2.4.2. Se ha implementado esta variante que combina la utilidad absoluta con la utilidad relativa ya que, en un juego basado en combate, evitar la predictibilidad es esencial, del mismo modo que es evitar realizar acciones con poca utilidad. (Hanlon y Watts, 2017).

Por otro lado, *BioWare* también implementa un mecanismo basado en utilidad en la selección de *target*. Esto es debido a que en los combates se enfrentan más de una unidad, por lo tanto, el agente debe seleccionar a que enemigo atacar. Esta elección también se basa en utilidad. Cada enemigo se define como una acción, por consiguiente, habrá tantas acciones como posibles enemigos a atacar. (Hanlon y Watts, 2017).

Para elegir que acción realizar, el agente primero decide que hechizo o ataque va a utilizar. Una vez se ha decidido el tipo de ataque, evalúa a todos los enemigos para calcular la utilidad de atacar a dicho enemigo.

Dragon Age: Inquisition es un referente para este trabajo ya que se pueden ver ejemplificados los conceptos explicados en el marco teórico, especialmente la variante *Dual-Utility Reasoning.*

4.1.2.Guild Wars 2 (ArenaNet, 2012)

Guild Wars 2 es un juego de rol masivo online desarrollado por *ArenaNet* (2012). En el juego, el jugador debe explorar un mundo fantástico en el cual debe completar misiones para subir de nivel y participar en eventos. Estos eventos son el principal atractivo del juego, donde una gran cantidad de jugadores se reúne para derrotar a criaturas con el objetivo de obtener recompensas únicas.

Figura 4.2 Imagen *in-game* de *Guild Wars 2*. Fuente: Joel Helmich, 2015.

La inteligencia artificial que controla estas criaturas es referente para este trabajo ya que los mecanismos de toma de decision están basados en utilidad. (Mark y Lewis, 2015).

Mark y Lewis (2015) afirman que utilizar una inteligencia basada en utilidad es la mejor opción debido a la gran variedad de criaturas que tiene el juego, cada una con diferentes habilidades únicas. Este amplio abanico de posibilidades dificulta la implementación y mantenimiento de otros mecanismos de toma de decisiones

como pueden ser las máquinas de estado finitas o árboles de comportamiento. (Mark y Lewis, 2015).

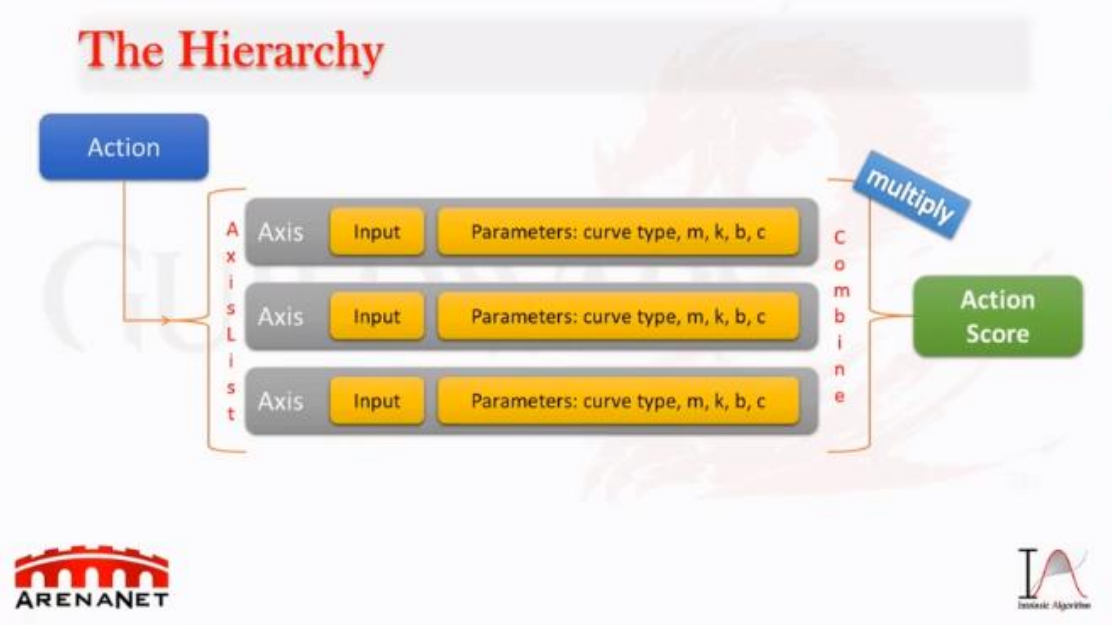

Figura 4.3 Obtención de la utilidad de una acción en *Guild Wars 2*. Fuente: Mike y Lewis, 2015.

En la Fig. 4.3 se puede ver el proceso que se sigue para calcular la utilidad de una acción. Como se puede apreciar, se sigue la misma concepción explicada en la sección 3 de este trabajo: Una acción está formada por una o múltiples consideraciones con su propia curva de utilidad. Las consideraciones (Axis) de la Fig. 4.3 están formadas por un *Input* y unos parámetros. El *Input* es la variable del juego de la cual se obtiene la utilidad y los parámetros definen la curva de la cual se obtiene la utilidad de la consideración. Como se puede observar, la utilidad de la acción (*Action Score*) se obtiene combinando la utilidad de todas las consideraciones mediante la aplicación de pesos. (Mike y Lewis, 2015).

Mike y Lewis (2015) exponen que las acciones, consideraciones y parámetros se definen a través de un programa externo, el cual está conectado al motor gráfico y permite la creación de comportamientos sin la necesidad de programar.

4.2. Herramientas y librerías

El objetivo de esta sección es analizar diferentes herramientas y librerías que implementan mecanismos basados en utilidad.

4.2.1. Apex Utility-AI (Apex Game Tools, 2016)

Apex Utility-AI es una herramienta desarrollada por *Apex Game Tools* (2016) para el motor gráfico *Unity* que facilita la implementación de mecanismos basados en utilidad.

La principal característica de esta herramienta es que ofrece un editor con interfaz para la edición de los comportamientos. Aunque el mecanismo de toma de decisiones se basa en los mismos conceptos y funcionamiento explicados en la sección 3 de este trabajo. *Apex Utility*-AI introduce un nuevo elemento: el selector.

Un selector es un elemento de alto nivel que encapsula acciones con sus respectivas consideraciones. Hay dos tipologías de selector: *Highest score wins* y *First score wins.* El primero elige la acción con la utilidad más alta (utilidad absoluta), mientras que el otro selector calcula la utilidad de las acciones en el orden en que están dispuestas en el editor y selecciona la primera acción que supera una cierta utilidad. (Apex Game Tools, 2016).

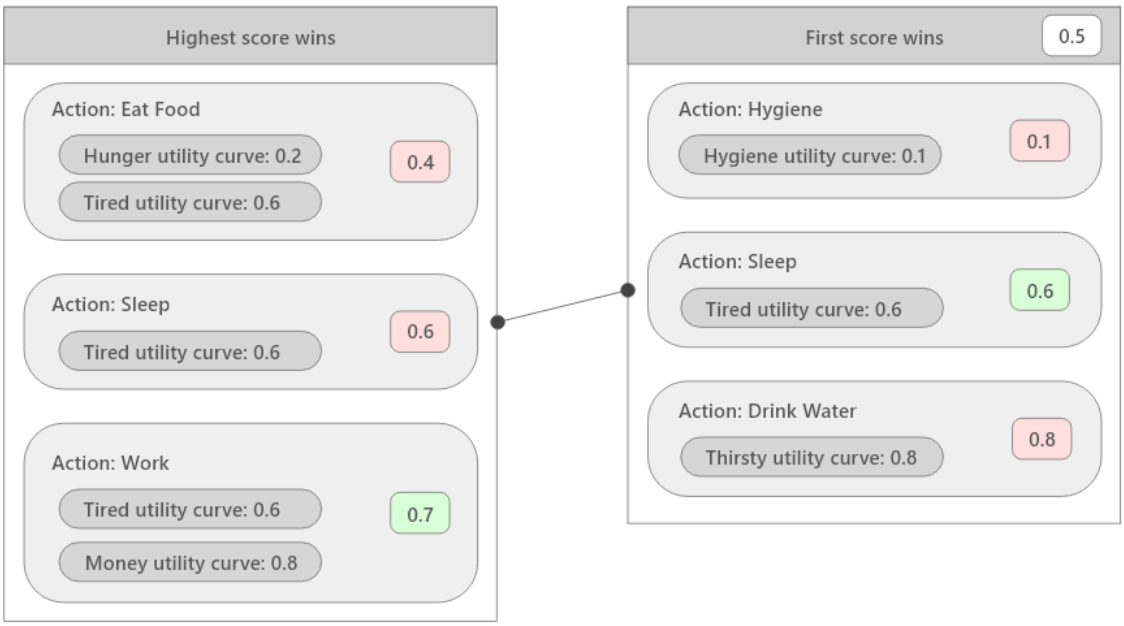

Figura 4.4 Ejemplificación de selectores de *Apex Utility-AI.* Fuente: Elaboración propia.

En la Fig. 4.4 se puede visualizar la diferencia entre un selector de tipo *Highest score wins* (izquierda) y un selector tipo *First score wins*. Como se puede apreciar, el selector que se basa en el principio de utilidad absoluta tiene 3 acciones con sus consideraciones: Comer, Dormir y Trabajar. La acción más útil es trabajar, por lo tanto, esta es la elegida. En cambio, el otro tipo de selector va a elegir la primera acción que supere la utilidad de 0.5, en este caso la acción dormir.

Una inteligencia artificial construida con el editor de *Apex Utility-AI* puede estar formada por uno o más selectores. Como se puede ver en la Fig. 4.4, ambos selectores están conectados, esta conexión indica que una vez se ha completado la acción del primer selector, se pasa al siguiente.

Apex Utility-AI es un referente para este trabajo ya que es una herramienta que permite la implementación de una inteligencia artificial basada en utilidad mediante un editor.

4.2.2.Utility-AI (Ban der Krujis, 2016)

Utility-AI es una librería desarrollada por Krujis (2016) que permite la implementación de inteligencia artificial basada en utilidad en *Unity*. Aunque esta librería no incorpora un potente editor como *Apex Utility-AI,* permite realizar gran parte de la implementación mediante la interfaz gráfica del motor.

Utility-AI sigue la misma concepción que los referentes explicados anteriormente. Las acciones están formadas por una o más consideraciones, las cuales calculan la utilidad a partir de una propiedad (valor del juego) y una curva de utilidad.

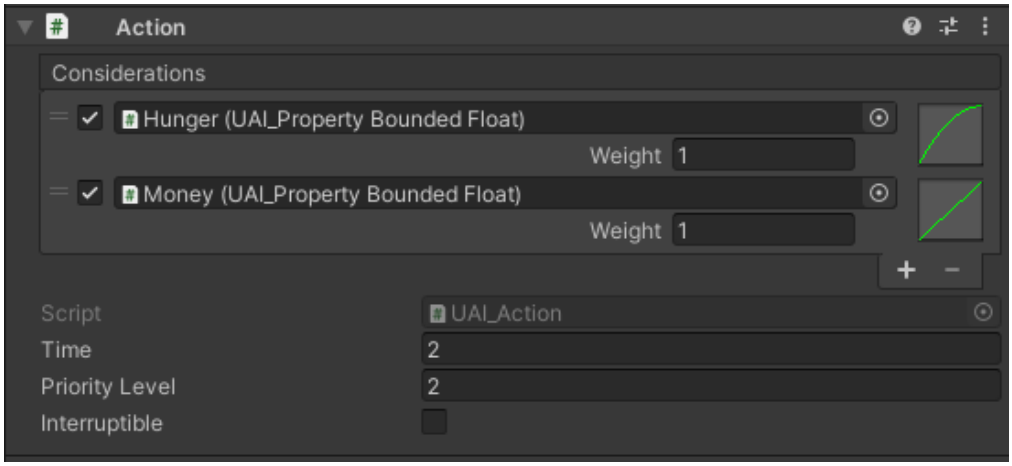

Figura 4.5 Ejemplo de acción en *Utility-AI*. Fuente: Elaboración propia.

En la Fig. 4.5 se puede observar una acción creada a partir las consideraciones *Hunger* y *Money*, cada una con su respectivo peso y curva de utilidad. A parte de las consideraciones, las acciones en *Utility-AI* tienen otros tres parámetros: tiempo, nivel de prioridad y si una acción es interrumpible. El tiempo define lo que se tarda en realizar la acción. El nivel de prioridad define la importancia de una acción sobre otra con una utilidad similar, por lo tanto, dadas dos acciones con la misma utilidad, se elige la que tiene el nivel de prioridad más alto. El parámetro interrumpir define si una acción puede ser interrumpida mientras se realiza para realizar otra de mayor utilidad.

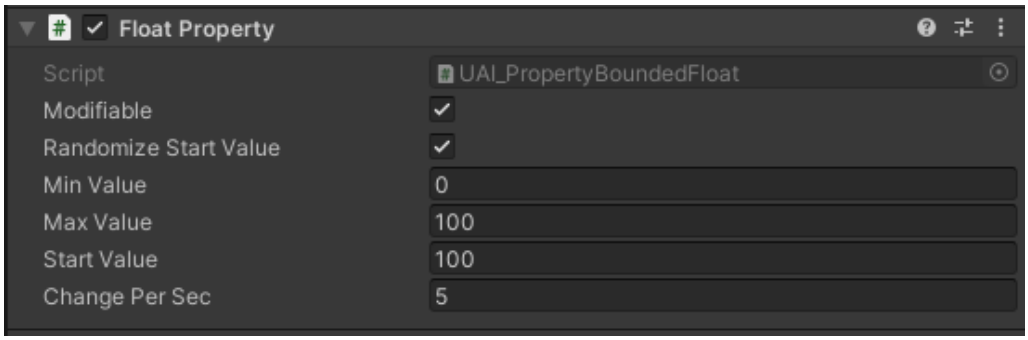

Figura 4.6 Ejemplo de propiedad en *Utility-AI.* Fuente: Elaboración propia.

Una consideración en *Utility-AI* está formada por una propiedad y una curva de utilidad. Tal y como se puede observar en la Fig. 4.6, una propiedad no es sólo un valor, sino que contiene una serie de parámetros editables.

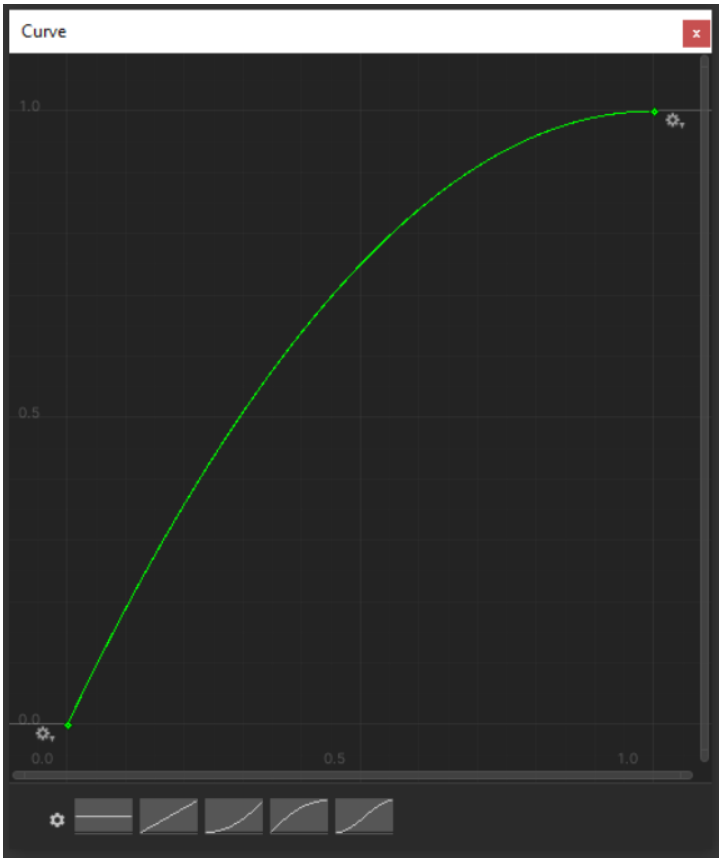

Figura 4.7 Ejemplo de curva de utilidad en *Utility-AI*. Fuente: Elaboración propia.

En la Fig. 4.7 se puede apreciar una curva cuadrática rotada, la cual calcula la utilidad de la consideración *Hunger*. *Utility-AI* implementa el editor de curvas nativo de *Unity*. Este editor permite utilizar curvas predefinidas, como puede ser

una curva lineal, cuadrática o logística. Pero también permite crear curvas por tramos añadiendo nodos.

Esta librería es un referente para este trabajo ya que implementa los conceptos explicados en la sección 3 de forma nativa en Unity, además permite editar o ampliar la interfaz con la que se implementan los comportamientos ya que es *Open Source* y utiliza el lenguaje de *scripting* del motor gráfico.

4.2.3.Curvature (Lewis, 2018)

Curvature es una herramienta desarrollada por Lewis (2018) que permite el prototipado de inteligencia artificial basada en utilidad. Esta herramienta difiere del objeto de este trabajo y por lo tanto solo se toma como referencia para este trabajo el selector de curvas.

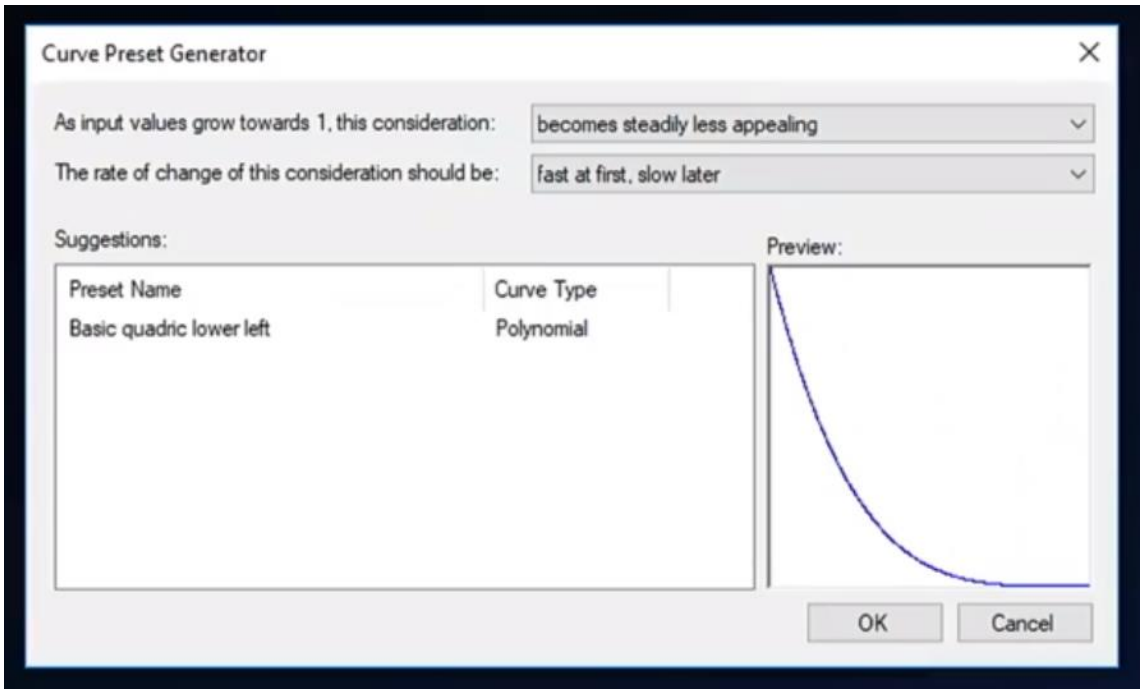

Figura 4.8 Selector de curvas de *Curvature*. Fuente: Lewis (2018)

En la Fig. 4.8 se puede observar el selector de curvas de *Curvature* este selector de curvas propone que tipo de curvas utilizar a partir de dos parámetros: que debe hacer la curva a medida que la utilidad se acerca a 1 (en el contexto de un valor normalizado) y el ritmo al que debe cambiar la curva. Con estos dos

parámetros, el selector ofrece una o más curvas que cumplen los requisitos. (Lewis, 2018).

El selector de curvas de *Curvature* es referente para este trabajo ya que ofrece una solución para la elección de curvas a los usuarios no familiarizados con la utilidad.

5. Plan de trabajo

5.1.1.Estudio previo

El objetivo de este apartado es definir que funcionalidades debe tener el objeto del trabajo. El estudio empieza analizando que funcionalidades se desean desarrollar. La decision de desarrollar dichas funcionalidades se basa en ponderar su funcionalidad en base al tiempo que conlleva desarrollarlas. A continuación, se destacan las funcionalidades que se desean implementar en orden de prioridad:

1	Creación de comportamientos a mediante la interfaz de *Unity*.
2	*Risk-takers.*
3	*Dual-Reasoning Utility*.
5	Guardar las acciones como un *Scriptable Object.*
5	Selector de curvas similar a *Curvature*.

Tabla 5.1 Funcionalidades que se desean implementar. Fuente: Elaboración propia.

Para poder implementar dichas funcionalidades se debe realizar un estudio de la API de *Unity*, la cual utiliza el lenguaje C#. Además, se debe realizar un estudio sobre la arquitectura que se va a implementar en el proyecto e identificar los referentes más importantes respecto a la inteligencia artificial basada en utilidad.

5.1.2.Investigación

El objetivo de la investigación es recopilar información respecto a los mecanismos de toma de decisiones basados en utilidad con el objetivo de establecer un marco teórico donde se exponen los conceptos necesarios para poder desarrollar este trabajo.

Además, se deben identifican los principales referentes que utilizan mecanismos basados en utilidad para poder realizar un análisis de su funcionamiento. Los

referentes pueden ser herramientas o videojuegos que implementan total o parcialmente mecanismos basados en utilidad.

5.1.3.Metodología

El objetivo de este apartado es definir la metodología de trabajo a seguir. Debido a que el tiempo de desarrollo es corto, es necesario establecer una metodología que permita minimizar el riesgo de no implementar las funcionalidades establecidas en el apartado 5.1.

La metodología elegida se basa en el sistema *Agile*. Esta metodología de trabajo se basa en la rápida iteración entre versiones con un desarrollo evolutivo acompañada de una planificación adaptativa y el *feedback* obtenido de las versiones anteriores. Debida a la rápida iteración los errores pueden ser detectados mucho más pronto comparado con la metodología convencional (*Waterfall*) en la que se testea el producto una vez se ha terminado el desarrollo. La detección precoz de un error permite minimizar el tiempo utilizado en solventar el problema. (Cockburn, 2007).

Se utilizará *Git* para controlar las versiones de la librería con la siguiente estructura de ramas:

- *Release Branch:* Se actualiza con versiones estables completas y estables para el uso público de la librería.
- *Development Branch:* Se actualiza con la nueva versión estable de la librería.
- *Version Branch:* Se actualiza cuando se ha desarrollado una nueva funcionalidad. Cuando se termina la iteración de esta versión se debe realizar una fase de testeo antes de actualizar *Development Branch*. Si se encuentra algún error, este de ser solucionado en la *Fix Branch*. Se debe crear una nueva *Version Branch* cada vez que se inicia una nueva iteración.
- *Feature Branch:* Se actualiza durante el desarrollo de dicha funcionalidad. Pueden existir diferentes ramas de funcionalidad que se desarrollan

paralelamente. Si se detecta un error de la funcionalidad durante el desarrollo de esta, el error se soluciona en esta misma rama.

- *Fix Branch:* Esta rama se utiliza en caso de detectarse un error en la *Version Branch*. Ya que es posible que, al combinar diferentes funcionalidades, se produzcan errores.

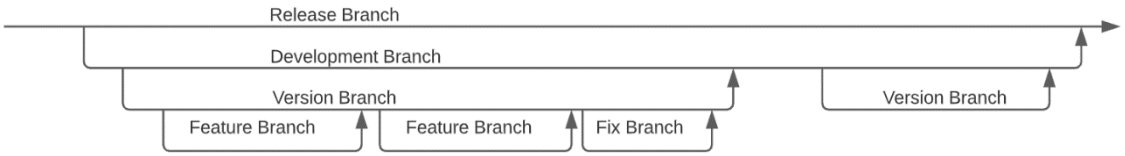

Figura 5.1 Estructura de ramas en *Git*. Fuente: Elaboración propia.

5.2. Cronograma

A continuación, se expone el cronograma con la planificación del proyecto, desde la asignación hasta la defensa final (mes 10).

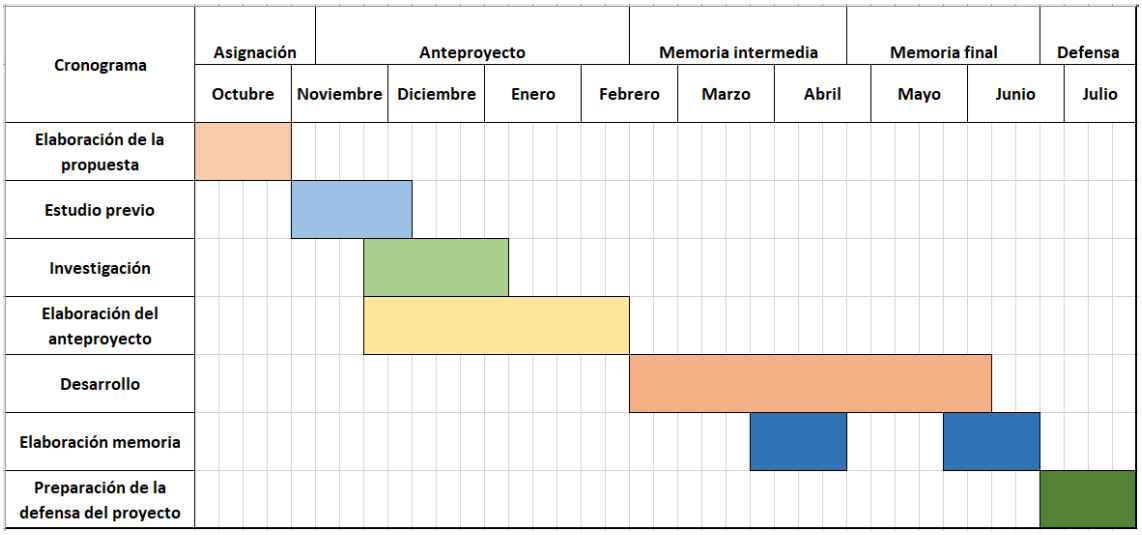

Figura 5.2 Cronograma del proyecto. Fuente: Elaboración propia.

6. Desarrollo del trabajo

Una vez expuesta la metodología utilizada para la elaboración de la parte práctica, este capítulo se centra en registrar la evolución del trabajo. El trabajo se desarrolla en base a dos iteraciones.

La estructura de este capítulo se divide en dos bloques (primera y segunda iteración) y cada bloque se segmenta en 4 secciones. Inicialmente se expone brevemente en que consiste la iteración, a continuación, se nombran los objetivos principales y secundarios de la iteración. Una vez aclarados los objetivos de la iteración, se expone detalladamente la arquitectura y los elementos desarrollados. Finalmente se realiza una valoración de la iteración.

6.1. Primera iteración

Esta primera iteración parte de una base nula a nivel de desarrollo, pero a nivel conceptual tanto la arquitectura como los elementos a desarrollar están establecidos en base al trabajo realizado en el capítulo 3 y 4.

El marco general de esta primera iteración es desarrollar y enlazar los componentes principales de la arquitectura con el fin de tener una primera versión de la herramienta estable. En el caso que se modificaran los elementos expuestos en esta primera iteración, se mencionará dicho cambio en su iteración pertinente.

6.1.1. Objetivos de la iteración

Como se ha nombrado en el apartado anterior y siguiendo la metodología establecida en el apartado 5.1.3. el objetivo a más alto nivel de esta iteración es obtener una versión estable y funcional de la herramienta en la *Release Branch*.

Para obtener esta versión estable se deben cumplir los siguientes objetivos principales. Los objetivos secundarios solo se deben desarrollar si se han cumplido todos los objetivos principales.

Objetivos principales:

- Implementación de las clases: *Action*, *Consideration* y *Property*.
- Implementación de la clase *Sorter*.
- Implementación de la clase *Agent.*
- Implementación de la clase *ActionLogic.*
- Desarrollar una herramienta que permita la construcción de agentes.
- Elaboración de una escena para poder corroborar que la arquitectura funciona correctamente.

Objetivos secundarios:

- Modificación de variables del agente mediante la herramienta.
- Edición de comportamientos mediante la herramienta.

6.2. Desarrollo de la iteración

6.2.1.1. Arquitectura

El mayor peso de trabajo de esta iteración se ha centrado en el diseño e implementación de la arquitectura para la selección de acciones. La arquitectura diseñada en esta iteración sigue los conceptos teóricos expuestos en el capítulo 3.

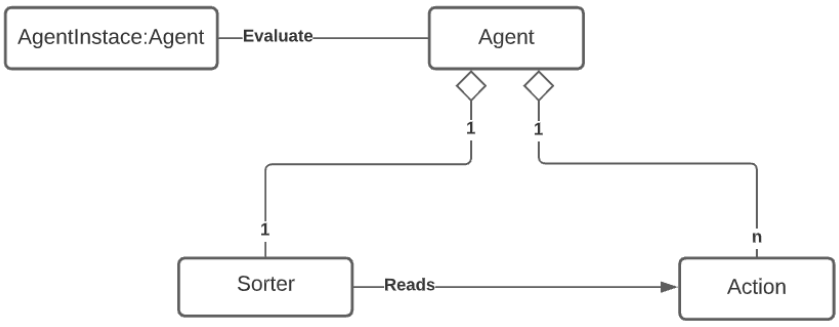

Figura 6.1 UML Evaluación de acciones.

En la Fig. 6.1 se puede observar la arquitectura que sigue una instancia de la inteligencia artificial (agente de ahora en adelante) para obtener la acción con

más utilidad. El agente obtiene esta acción mediante la clase *Sorter,* la cual devuelve una lista de acciones ordenadas según su utilidad. Este procedimiento se realiza cada vez que se llama la función *Evaluate*. Cada agente está compuesto por una instancia de *Sorter* y una lista de *Action*.

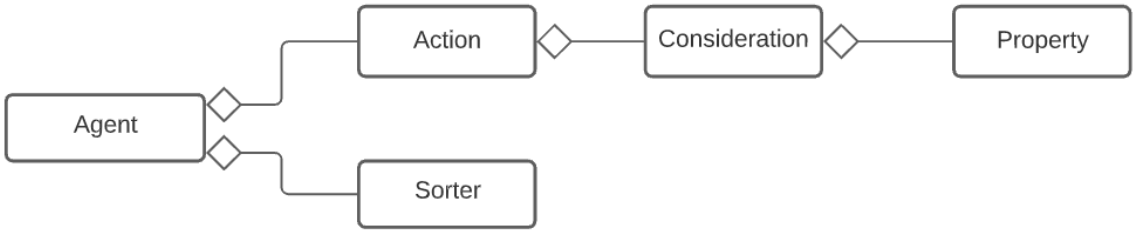

Figura 6.2 Relación de clases que forman un *Agent*.

En la Fig. 6.2 se puede observar la composición de la clase *Agent*. Como se ha mencionado anteriormente, la clase agente se compone de una única instancia de *Sorter* y una única lista de *Action*. Esta lista puede contener un número indefinido de *Action* y cada *Action* puede estar formada por un número indefinido de *Consideration*. En cambio, cada *Consideration* está formada por una única *Property*.

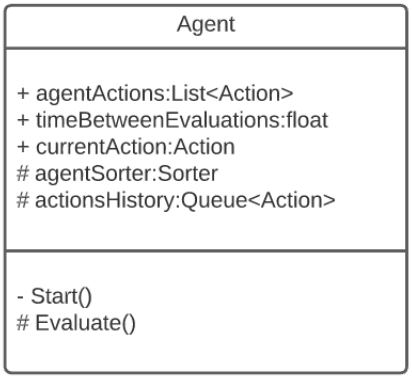

Figura 6.3 UML de la clase *Agent*.

En la Fig. 6.3 se puede observar el diagrama UML de la clase *Agent*. Todos los agentes creados a partir de la herramienta de este trabajo heredan de esta clase.

Tal y como se ha expuesto anteriormente, *Agent* contiene una instancia de *Sorter y* una lista de *Action.* Tal y como se puede observar en la Fig. 6.4, la función

virtual *Evaluate* es la encargada de llamar al *Sorter*, el cual devuelve una lista de acciones ordenadas según la utilidad. La implementación original de esta función compara la acción actual almacenada en *currentAction* con la acción con mayor utilidad de la lista proporcionada por el *Sorter.* Si esta acción es diferente, añade *currentAction* a *actionHistory* y luego guarda la acción de mayor utilidad en *currentAction*. Al ser una función virtual, esta lógica puede ser sobrescrita. Esta función se llama a un ritmo constante marcado por el atributo privado *timeBetweenEvaluations*.

```
protected virtual void Evaluate()
{
    Action l_bestAction = agentSorter.GetSortedActionWithUtility()[0].action;

    float utility = agentSorter.GetSortedActionWithUtility()[0].utility;
    print( message: l_bestAction.actionName + ":  " + utility);

    if (l_bestAction != currentAction)
    {
        actionsHistory.Enqueue(currentAction);
        currentAction.actionLogic.OnExit();
        currentAction = l_bestAction;
        currentAction.actionLogic.OnEnter();
    }
}
```

Figura 6.4 Función *Evaluate* de la clase *Agent*.

La clase *Agent* hereda de *Monobehavior*, por lo tanto, se pueden usar las funciones llamadas mediante eventos que proporciona *UnityEngine.* En esta clase se utiliza la función *Start* para inicializar la instancia de *Sorter* mediante constructor por parámetros además de inicializar *actionHistory* y *currentAction*.

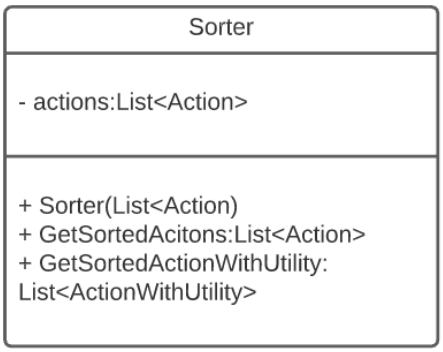

Figura 6.5 UML de la clase *Sorter*.

En la Fig. 6.5 se puede observar el diagrama UML de la clase *Sorter*. El único atributo de esta clase es una lista de *Action*, la cual se inicializa en el constructor. Esta lista contiene las mismas acciones que el agente a la cual pertenece dicha instancia de la clase *Sorter.*

La función *GetSortedActions* devuelve una lista con las acciones ordenadas según la utilidad de cada acción. Mientras que la función *GetSortedActionsWithUtility* devuelve una lista de *ActionWithUtility* (una estructura formada por un atributo *Action* y un atributo *float*).

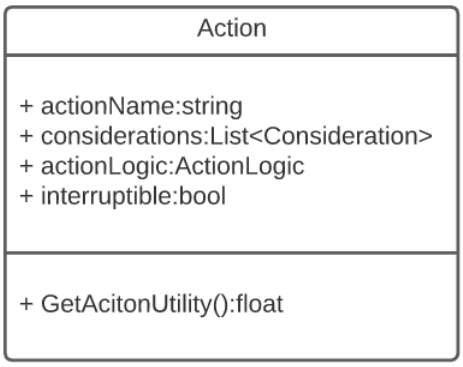

Figura 6.6 UML de la clase *Action*.

En la Fig. 6.6 se puede observar el diagrama UML de la clase *Action*. Esta clase cumple la función de representar una acción / comportamiento dentro de la librería, separando así la lógica de dicha acción de la selección de comportamientos.

La clase *Action* está formada por el nombre de la acción (*actionName*), una lista de *Consideration,* la lógica asociada a dicha acción (*actionLogic*) y si esta acción es interrumpible (*interruptible*). La función *GetActionUtility* se encarga de calcular la utilidad de la acción.

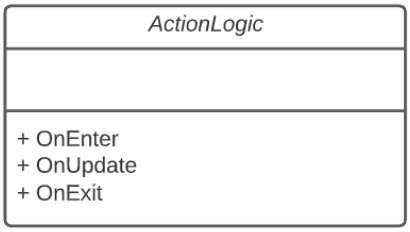

Figura 6.7 UML de la clase *ActionLogic*.

La clase abstracta *ActionLogic* cumple la funcionalidad de desacoplar la lógica de los comportamientos y la selección de estos.

Tal y como se puede observar en la Fig. 6.7, la clase *ActionLogic* se asimila a una clase *State* de una máquina de estados finita, ya que contiene un método de entrada (*OnEnter*), un método para actualizar mientras se está ejecutando esa acción (*OnUpdate*) y un método de salida (*OnExit*). Se ha optado por esta estructura ya que nunca se ejecutan dos acciones al mismo tiempo, por lo tanto, se debe salir de una acción (método *OnExit*), antes de entrar en otra acción (método *OnEnter*).

La clase *ActionLogic* hereda de *Monobehavior* para así facilitar la integración de la lógica con todos los objetos y componentes que proporciona *UnityEngine*.

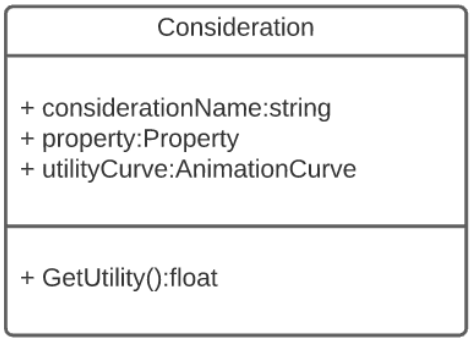

Figura 6.8 UML de la clase *Consideration*.

En la Fig. 6.8 se puede observar el diagrama UML de la clase *Consideration*. Esta clase contiene los atributos de nombre de la consideración (*considerationName*), una instancia de la clase *Property* y la curva de utilidad *(utilityCurve)*.

La función *GetUtility* calcula la utilidad de dicha consideración a partir del valor normalizado de la propiedad.

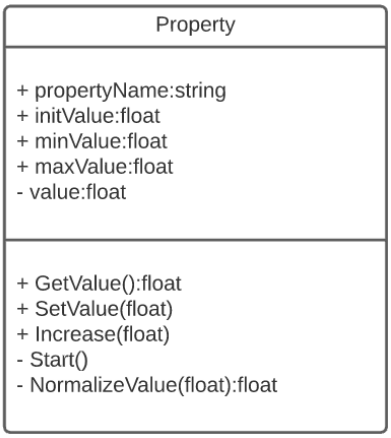

Figura 6.9 UML de la clase *Property*.

La clase *Property* cumple la función de representar directa o indirectamente una variable numérica del juego. Puede ser de asignación directa como puede ser la salud del personaje o puede ser una variable que no se puede encontrar directamente dentro del juego, la cual se debe calcular *a posteriori,* como puede ser el porcentaje de aciertos con un arma.

En la Fig. 6.9 se puede observar el diagrama UML de la clase *Property*. Esta clase está formada por el nombre de la propiedad (*propertyName*), el valor inicial del valor de la propiedad (*initValue*), el valor mínimo (*minValue*), el valor máximo (*maxValue*) y el valor de dicha propiedad (*value*).

El atributo *value* es privado, por lo tanto, se debe realizar un *Get* y *Set* de dicho atributo para poder modificar su valor desde fuera de la clase *Property*. La función *Increase* se encarga de incrementar el valor de *value* (o decrementar si el parámetro es negativo). La función *Start* se encarga de inicializar el valor de *value* con el valor de *initValue*. La función *NormalizeValue* es la encargada de normalizar el valor de la propiedad a partir de su valor máximo y mínimo, definido por los atributos *minValue* y *maxValue*.

Como se puede observar en la Fig. 6.10, las clases *Action, Consideration* y *Property* heredan de la clase *Monobehavior,* para así poder construir los agentes con la herramienta que se desarrolla en este trabajo además de utilizar las funciones de eventos que proporciona *UnityEngine.*

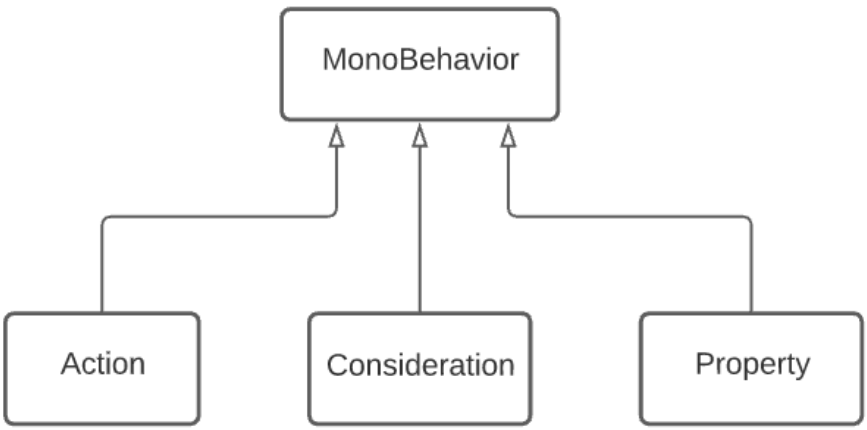

Figura 6.10 Diagrama de la herencia de las clases *Action, Consideration* y *Property.*

6.2.1.2. Desarrollo de la herramienta

La herramienta desarrollada en este trabajo permite la creación de agentes basados en utilidad a través de una interfaz. En esta primera iteración la funcionalidad de la herramienta es limitada debido a que la carga principal de

trabajo ha sido centrada en el desarrollo de la interfaz de la herramienta y todos los elementos asociados a ella.

En la Fig. 6.11 se puede observar la interfaz provisional de la herramienta. La zona lateral izquierda contiene la lista de agentes creados. En la parte superior se encuentra un encabezado donde se pueden editar la opción de autoguardado y la visualización de componentes globales. Si esta opción está activada se van a visualizar todas las consideraciones y propiedades del agente, mientras que si está desactivada solo se van a visualizar las consideraciones y propiedades de la acción seleccionada.

En la parte derecha de la interfaz se encuentra el inspector de la herramienta. En esta sección es donde el usuario puede editar tanto los parámetros como los componentes del agente. La interfaz del inspector varía según el componente que está seleccionado. En la Fig. 6.11 el componente seleccionado (*Time*) es una consideración, por lo tanto, en el inspector se pueden apreciar los diferentes campos editables relacionados con dicho componente.

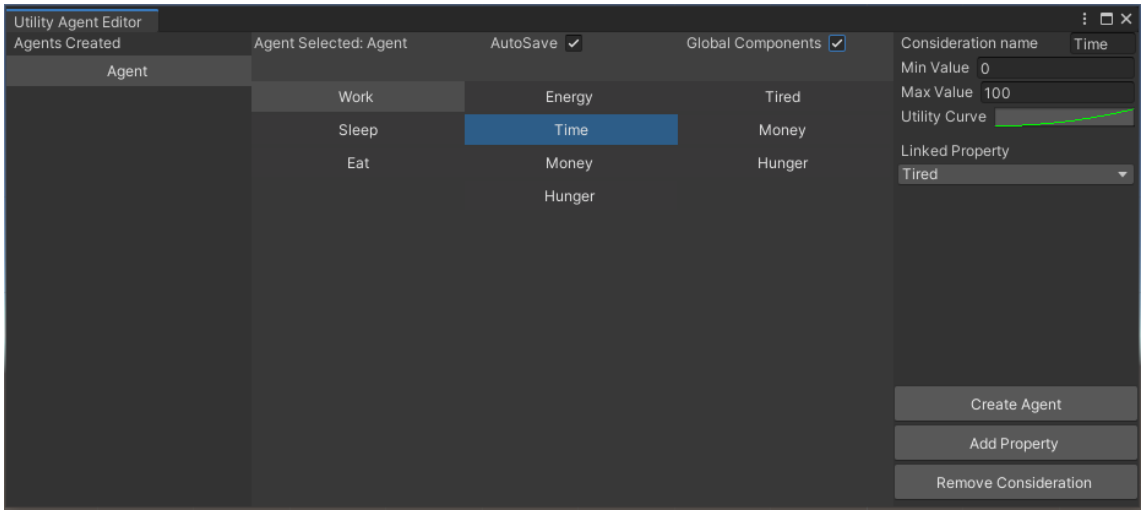

Figura 6.11 Herramienta desarrollada en la primera iteración.

En esta iteración el desarrollo de la herramienta se ha centrado en la creación de agentes y en la adición y sustracción de componentes (Acciones, consideraciones y propiedades).

La creación de agentes se basa en dos elementos clave: la dirección de creación (*path*) y el tipo del agente. En este trabajo los agentes son creados a partir de los *prefabs* que proporciona *UnityEngine* y estos para ser guardados necesitan un *path*. Por otra parte, el tipo de agente viene definido por la clase de dicho agente, la cual debe heredar de *Agent*.

Como se puede ver en la Fig. 6.12, un agente se construye a partir de 4 *GameObjects* principales: El *GameObject* padre y los encapsuladores de acciones, consideraciones y propiedades, los cuales son hijos del *GameObject* padre.

Véase que al final de la función se destruye el *GameObject* creado. Esto es debido a que *UnityEngine* necesita crear el *GameObject* en una escena para que este pueda ser guardado como un *prefab*, por lo este debe ser destruido para que no afecte el estado de dicha escena.

```
public static void Create(string _name, string _path, Type _agentType)
{
    GameObject l_agent = new GameObject {name = _name};
    GameObject l_actionObject = new GameObject{name = "Actions"};
    GameObject l_considerationsObject = new GameObject{name = "Considerations"};
    GameObject l_propertiesObject = new GameObject{name = "Properties"};

    l_agent.AddComponent(_agentType);
    l_actionObject.transform.parent = l_agent.transform;
    l_considerationsObject.transform.parent = l_agent.transform;
    l_propertiesObject.transform.parent = l_agent.transform;

    PrefabUtility.SaveAsPrefabAssetAndConnect(l_agent,
        assetPath: _path + "/" + l_agent.name
       + ".prefab", InteractionMode.AutomatedAction);
    Object.DestroyImmediate(l_agent);
}
```

Figura 6.12 Función para la creación de agentes.

Las acciones, consideraciones y propiedades creadas con la herramienta serán añadidas a sus correspondientes encapsuladores como componentes. Esta

operación se puede realizar sin la necesidad de instanciar el *GameObject* en la escena, por lo tanto, se obtiene una pequeña mejora de rendimiento.

Como se puede observar en la Fig. 6.13, se crea una instancia local del *prefab*, se añade un componente de tipo *Action* y luego se guarda dicha instancia como *prefab* en su correspondiente *path*.

La adición de consideraciones y propiedades sigue el mismo procedimiento, pero se añade el componente en el *GameObject* encapsulador correspondiente.

```
public static class AddAction
{
    1 usage
    public static void ToAgent(AssetWithPath _asset)
    {
        GameObject l_prefabInstance = PrefabUtility.LoadPrefabContents(_asset.path);

        l_prefabInstance.transform.Find("Actions").gameObject.AddComponent<Action>();

        PrefabUtility.SaveAsPrefabAsset(l_prefabInstance, _asset.path);
        PrefabUtility.UnloadPrefabContents(l_prefabInstance);
    }
}
```

Figura 6.13 Adición de una acción a un agente.

Para la substracción de componentes si se necesita crear una instancia del *prefab* en la escena. Esto es debido a que la función *DestroyImmediate* solo obtiene por parámetro componentes que estén instanciados en la escena.

En la Fig. 6.14 se puede observar la sustracción de una acción a un agente, esta sustracción se realiza mediante la destrucción de dicho componente. La destrucción solo se realiza si el componente existe en dicho *prefab*. Véase que para determinar si un prefab tiene dicho componente, este se compara con todos los componentes de su tipo que tiene el *prefab*.

```
public static class RemoveAction
{
    1 usage
    public static void FromAgent(AssetWithPath _asset, Action _action)
    {
        GameObject l_prefabInstance = (GameObject)PrefabUtility.InstantiatePrefab(_asset.gameObject);
        Action[] l_actions = FindAgentComponents.GetActions(l_prefabInstance);

        foreach (var l_action in l_actions)
        {
            if (l_action == _action)
            {
                Object.DestroyImmediate(_action);
                break;
            }
        }
        PrefabUtility.ApplyRemovedComponent(l_prefabInstance, _action, InteractionMode.AutomatedAction);
        PrefabUtility.SaveAsPrefabAsset(l_prefabInstance, _asset.path);
        Object.DestroyImmediate(l_prefabInstance);
    }
}
```

Figura 6.14 Sustracción de una acción a un agente.

La sustracción de consideraciones y propiedades sigue el mismo procedimiento que el de la Fig. 6.14, comparando cada componente con los de su tipo obtenidos de la clase *FindAgent*Components.

Como se puede observar, tanto en la Fig. 6.13 como en la Fig. 6.14, se utiliza la clase estática *FindAgentComponents* para la obtención de los componentes de un agente. Véase la Fig. 6.15.

```
public static class FindAgentComponents
{
    3 usages
    public static Action[] GetActions(GameObject _agent)
    {
        return _agent.GetComponentsInChildren<Action>();
    }
    5 usages
    public static Consideration[] GetConsiderations(GameObject _agent)
    {
        return _agent.GetComponentsInChildren<Consideration>();
    }
    5 usages
    public static Property[] GetProperties(GameObject _agent)
    {
        return _agent.GetComponentsInChildren<Property>();
    }
```

Figura 6.15 Obtención de componentes dado un agente.

En esta clase se encuentran varios métodos los cuales devuelven los componentes dado un *GameObject*, el cual debe seguir la estructura de un agente presentado en la Fig. 6.12. Además, esta clase contiene diferentes métodos los cuales devuelven los nombres de dichos componentes. Estos métodos se utilizan para la construcción de la interfaz de la herramienta, la cual ofrece diferentes selectores y desplegables que requieren de estos nombres. Véase la Fig. 6.16.

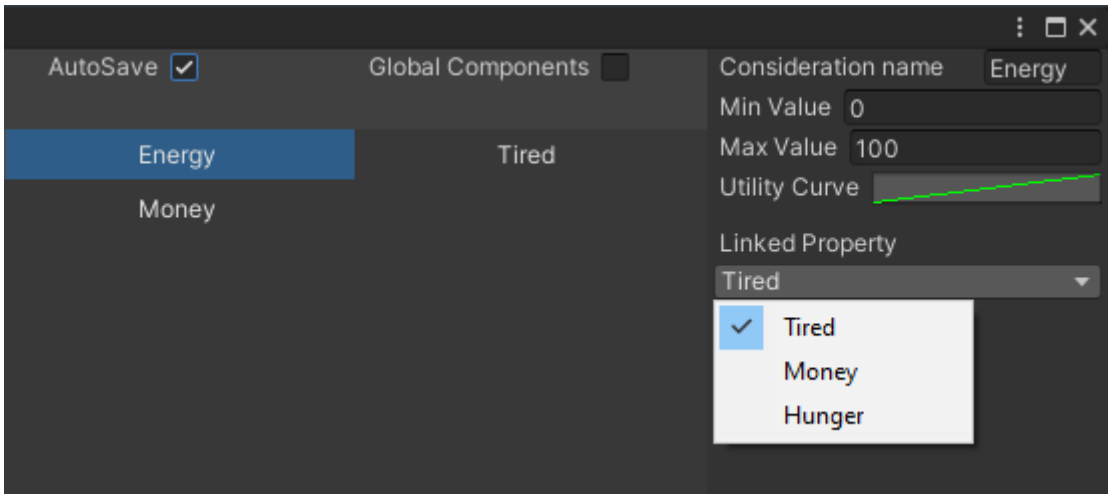

Figura 6.16 Selección de la propiedad vinculada a la consideración.

Como se ha mencionado anteriormente, esta herramienta trabaja con todos los *GameObjects* que contengan un componente heredado de la clase *Agent* en su *GameObject* padre. Para poder trabajar con estos *prefabs*, la herramienta tiene que detectar que *assets* son compatibles con la herramienta. Este procedimiento se realiza en la clase *FindAssets* con el método *GetAgents,* tal y como se puede ver en la Fig. 6.17.

```
public static List<AssetWithPath> GetAgents()
{
    string[] l_guids = AssetDatabase.FindAssets( filter: "t:prefab");
    string[] l_paths = new string[l_guids.Length];

    for (int i = 0; i < l_guids.Length; i++)
    {
        l_paths[i] = AssetDatabase.GUIDToAssetPath(l_guids[i]);
    }

    List<AssetWithPath> l_agentAssets = new List<AssetWithPath>();
    foreach (var l_path :string  in l_paths)
    {
        GameObject l_gameObject = (GameObject)AssetDatabase.LoadAssetAtPath(l_path, typeof(GameObject));
        if(l_gameObject.TryGetComponent(out Agent l_agentComponent))
            l_agentAssets.Add( item: new AssetWithPath(l_gameObject, l_agentComponent, l_path));
    }

    return l_agentAssets;
}
```

Figura 6.17 Método *GetAgents*.

En este método se obtienen todos las direcciones de guardado de los *assets* de tipo *prefab*. Una vez se ha obtenido la lista de prefabs, se itera sobre esta para

comprobar si tienen un componente de tipo *Agent*. Esta comprobación se realiza con el método *TryGetComponent,* proporcionado por la librería de *UnityEngine*. Si el *prefab* tiene un componente *Agent*, este es añadido a la lista de *prefabs* aceptados como válidos.

La interfaz de la herramienta se construye con la librería *IMGUI* en la clase *UtilityAgentCreatorWindow.* La interfaz se divide en tres secciones principales: los agentes creados, los componentes y el inspector.

La sección de agentes creados muestra todos los *Prefabs* del proyecto que contienen un componente de tipo *Agent*, obtenidos del método *GetAgents*, el cual se puede observar en la Fig. 6.17. Los agentes creados pueden ser seleccionados, lo cual revela sus componentes (acciones, consideraciones y propiedades).

La sección de componentes enseña los componentes del agente, obtenidos de los métodos de la clase *FindAgentComponents*, el cual se puede observar en la Fig. 6.15. Los componentes enseñados en esta sección son seleccionables, lo que permite editar sus atributos en el inspector de la herramienta. Además, cuando se selecciona una Acción o una consideración se pueden observar los componentes de su ámbito local. Si se quieren observar todos los componentes del agente se debe seleccionar la opción *ViewGlobalComponents*.

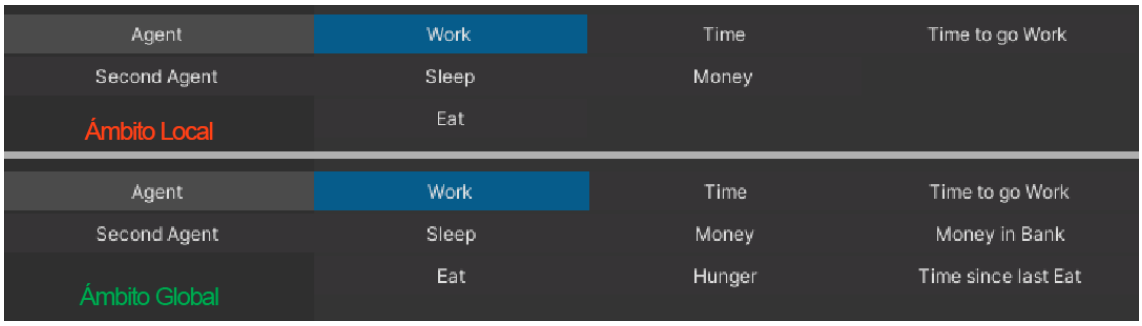

Figura 6.18 Diferenciación entre componentes globales y locales

En la Fig. 6.18 se puede observar la diferenciación entre los componentes en un ámbito local o global, según si la opción *ViewGlobalComponents* está activada. Se define ámbito local como todos los componentes directamente relacionados

a la acción seleccionada, en cambio, el ámbito global comprende todos los componentes de un agente, estén vinculados o no a una acción. Como se puede ver en la Fig. 6.18, la acción seleccionada es *Work*, su ámbito local comprende las consideraciones *Time* y *Money.* Mientras que en el ámbito global se puede ver la consideración *Hunger*, la cual no pertenece a la acción *Work,* pero sigue formando parte del agente seleccionado.

En el inspector de la herramienta se enseñan los diferentes atributos editables del agente o componente seleccionado. Además, se presentan diferentes botones que permiten añadir o eliminar componentes. Tanto los atributos como los botones expuestos en el inspector varían según el elemento seleccionado.

Los atributos expuestos mediante la herramienta están directamente vinculados al agente seleccionado, por lo tanto, al realizar cambios en estos parámetros, los cambios se realizan directamente en el *prefab* del agente seleccionado.

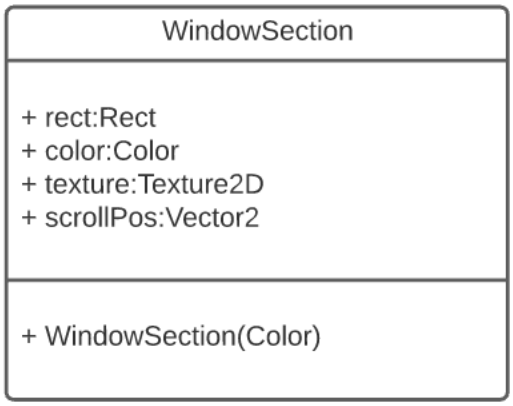

Figura 6.19 Clase *WindowSection*.

Como se ha expuesto anteriormente, la interfaz de la herramienta se divide en diferentes secciones principales. Estas secciones principales están construidas a partir de diferentes secciones, por ejemplo, la sección de componentes está dividida en la sección de acciones, la sección de consideraciones y la sección de propiedades. Para construir una interfaz compleja en la misma ventana con la librería *IMGUI* es necesario dividir las diferentes áreas de contenido en secciones. En dicha librería, cada sección debe tener un atributo de tipo *Rect*, una textura y un color. Para no cargar la clase *UtilityAgentCreatorWindow* con

una gran cantidad de atributos, se ha creado la clase *WindowSection* para mejorar la gestión de dichos atributos. Véase la Fig. 6.19.

Debido a que diversas secciones de la interfaz requieren de un componente *Scroll* para poder desplazarse por todos los elementos de la sección, *WindowSection* contiene un atributo para guardar la posición del desplazamiento de dicho *Scroll*.

Para finalizar esta primera iteración se ha desarrollado un sistema de guardado que permite guardar los cambios realizados en la herramienta. Actualmente, los cambios realizados en la herramienta se aplican a una instancia “virtual” del agente seleccionado. Esta instancia se sobrescribe cada vez que se selecciona un agente en la herramienta, por lo tanto, si se realizan cambios y después se selecciona otro agente, la instancia es sobrescrita y se pierden los cambios realizados.

```
public static class SaveChanges
{
    Frequently called  1 usage
    public static void Agent(AssetWithPath _asset)
    {
        GameObject l_prefabInstance = (GameObject)PrefabUtility.InstantiatePrefab(_asset.gameObject);

        PrefabUtility.ApplyObjectOverride(l_prefabInstance, _asset.path, InteractionMode.AutomatedAction);
        PrefabUtility.SaveAsPrefabAsset(l_prefabInstance, _asset.path);
        Object.DestroyImmediate(l_prefabInstance);
    }
}
```

Figura 6.20 Clase *SaveChanges*.

Para guardar los cambios se debe sobrescribir el prefab del agente guardado en el proyecto por la instancia “virtual” de la herramienta. Para realizar este proceso se utiliza la clase *SaveChanges* con el método *Agent*, al cual se le debe pasar la estructura que contiene el *GameObject* que se va a guardar y el *path* para saber dónde se tiene que guardar.

Tal y como se puede observar en la Fig. 6.20, se crea una instancia del agente en la escena, se aplican los cambios y se guarda como *prefab* en el *path* correspondiente. Por último, se destruye la instancia creada en la escena.

6.2.2. Valoración de la iteración

La valoración de esta primera iteración es positiva ya que se han cumplido todos los objetivos principales y secundarios, aunque la escena para validar que la herramienta funciona se podría mejorar con elementos más visuales.

El principal punto por destacar es la diferencia de tiempo de desarrollo entre la arquitectura y la herramienta. El tiempo invertido en desarrollar la herramienta ha sido mucho mayor que el de la arquitectura, principalmente por los problemas de escalabilidad que presenta la librería IMGUI. Estos problemas de escalabilidad presentan varias desventajas para futuras iteraciones las cuales pueden frenar la implementación de futuras características.

La herramienta debe ser reescrita cuanto antes utilizando la librería UIElements que ofrece *UnityEngine* para resolver los problemas expuestos anteriormente. Si bien reescribir la herramienta puede suponer una carga de trabajo adicional, tanto la arquitectura como los elementos expuestos en el apartado 6.1.2.2 ofrecen un núcleo fuerte sobre el que trabajar.

Actualmente el sistema de guardado está basado en el tiempo (un guardado cada *n* unidades de tiempo). Esto no es óptimo ya que se realizan guardados, aunque no se hayan producido cambios, además que se guardan los cambios sin que el usuario decida si deben ser guardados o no. En la siguiente iteración se pretende implementar una opción de autoguardado basado en cambios además de ofrecer una opción de guardado manual.

6.3. Segunda Iteración

La segunda iteración parte de una versión funcional de la herramienta con algunos errores menores. El trabajo inicial de esta iteración se centra en resolver estos errores para así poder añadir funcionalidad a la herramienta más fácilmente.

La funcionalidad añadida en esta iteración parte de la arquitectura expuesta a lo largo de este trabajo, por lo tanto, su implementación no debe presentar mayores problemas.

6.3.1. Objetivos de la iteración

Los objetivos por cumplir en esta iteración añaden funcionalidad a la herramienta o mejoran lo implementado en la primera iteración del trabajo. Nuevamente, los objetivos se dividen en objetivos principales y objetivos secundarios.

Objetivos principales:

- Resolver errores de la iteración anterior.
- Implementación Risk-Takers.
- Implementación Dual-Utility Reasoning.
- Implementación acción no interrumpible.
- Mejorar el sistema de guardado.
- Añadir y eliminar ActionLogic.

Objetivos secundarios:

- Mejorar la distribución de la interfaz.
- Mejorar el apartado estético de la herramienta.
- Implementar un lista preestablecida de curvas.

Una vez cumplidos los objetivos principales y secundarios de esta iteración, la herramienta debe ser funcional y no debe presentar errores.

6.3.1.1. Desarrollo de la herramienta

La primera iteración de la herramienta, aunque es funcional, presenta una serie de errores que deben ser resueltos antes de empezar a trabajar en nuevas funcionalidades. Estos errores aparecen cuando se intenta añadir, eliminar o modificar un componente y algún elemento relacionado con este componente es nulo. Por ejemplo, si se elimina una propiedad que está vinculada a una

consideración, la consideración devuelve un error ya que la propiedad vinculada ya no existe. La solución que se ha implementado es revisar que estos elementos no sean nulos, y en el caso de que lo sean (debido a la naturalidad de la arquitectura), se avisa al usuario mediante la interfaz.

En la primera iteración, añadir y eliminar *ActionLogic* se debe realizar desde el *GameObject* mientras que en esta iteración ya se puede realizar la configuración desde la herramienta. Los clases que permiten esta funcionalidad son: *AddActionLogic* y *RemoveActionLogic*. Estas clases son estáticas y contienen un solo método, el cual configura un agente.

```
public static class AddActionLogic
{
    Frequently called  1 usage
    public static void ToAgent(AssetWithPath _asset, Type _actionLogicType)
    {
        GameObject l_prefabInstance = PrefabUtility.LoadPrefabContents(_asset.path);

        l_prefabInstance.transform.Find("ActionsLogic").gameObject.AddComponent(_actionLogicType);

        PrefabUtility.SaveAsPrefabAsset(l_prefabInstance, _asset.path);
        PrefabUtility.UnloadPrefabContents(l_prefabInstance);
    }
}
```

Figura 6.21 Clase *AddActionLogic*

Como se puede observar en la Fig. 6.21, la función *ToAgent* sigue el mismo método de la Fig. 6.13, se carga una instancia local del agente a partir de su *path* y se añade el componente deseado en el objeto hijo llamado *ActionsLogic*. Una vez aplicado el cambio, se guarda la instancia del agente como *prefab* en el *path* especificado.

```
public static class RemoveActionLogic
{
    Frequently called  1 usage
    public static void FromAgent(AssetWithPath _asset, ActionLogic _actionLogic)
    {
        GameObject l_prefabInstance = (GameObject)PrefabUtility.InstantiatePrefab(_asset.gameObject);
        ActionLogic[] l_actions = FindAgentComponents.GetActionsLogic(l_prefabInstance);

        foreach (var l_action in l_actions)
        {
            if (l_action == _actionLogic)
            {
                Object.DestroyImmediate(_actionLogic);
                break;
            }
        }

        PrefabUtility.ApplyRemovedComponent(l_prefabInstance, _actionLogic, InteractionMode.AutomatedAction);
        PrefabUtility.SaveAsPrefabAsset(l_prefabInstance, _asset.path);
        Object.DestroyImmediate(l_prefabInstance);
    }
}
```

Figura 6.22 Clase *RemoveActionLogic*

Como se ha mencionado en la introducción de esta iteración, la arquitectura de la librería es la misma, por lo tanto, la adición y substracción de componentes se realiza siguiendo el mismo procedimiento. Véase la Fig. 6.14 y la Fig. 6.22. Como se puede observar en ambas figuras, lo único que varía es el tipo de componente a eliminar.

Lo que si varía respecto a los otros componentes es cómo se añaden y eliminan los componentes de tipo *ActionLogic* en la herramienta. Debido a que se debe seleccionar que *ActionLogic* se debe añadir o eliminar del agente, la interfaz debe proporcionar soporte a dicha elección.

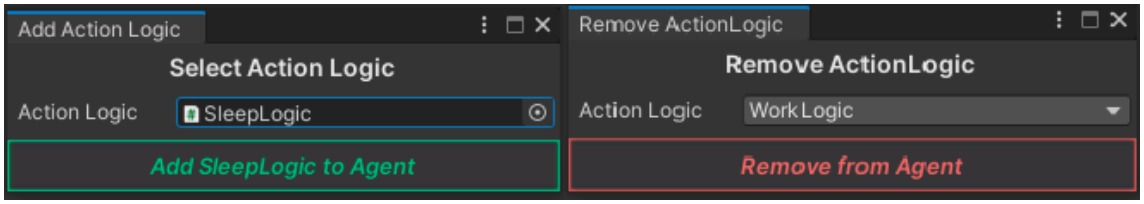

Figura 6.23 Interfaces para poder seleccionar la *ActionLogic*

En la Fig. 6.23 se pueden observar las dos ventanas que dan soporte a la elección de *ActionLogic*. La ventana para añadir un *ActionLogic*, se debe elegir el script que contiene la lógica, mientras que para eliminar un *ActionLogic*, se debe elegir uno de los que ya forman parte del agente. La interfaz para añadir

ActionLogic se construye en la clase *AddActionLogicWindow*, mientras que la interfaz para eliminar *ActionLogic* se construye en la clase *RemoveActionLogicWindow*.

La implementación de los Risk-takers y Dual-Utility Reasoning se ha realizado en la clase *Agent*. Un agente puede ser de tres tipologías: Un agente estándar, un agente *risk-taker*, o un agente *dual-utility*. La definición del tipo de agente se realiza a través del inspector en la herramienta en el atributo *AgentType*, tal y como se puede ver en la Fig. 6.24.

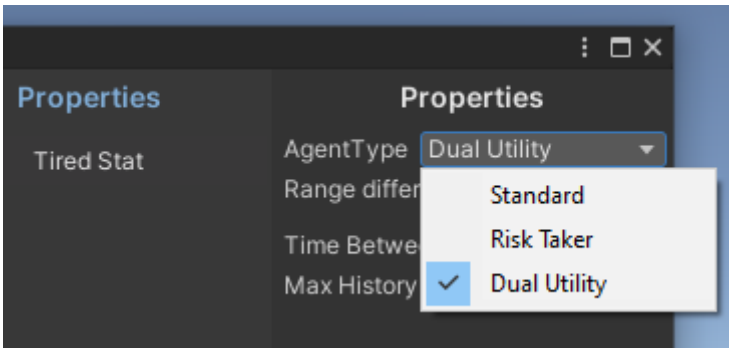

Figura 6.24 Elección del tipo de agente.

Según la tipología del agente, el método de evaluación de acciones será distinto. Actualmente, las tres funciones de evaluación son las siguientes:

- *EvaluateStandard.*
- *EvaluateRiskTaker.*
- *EvaluateDualUtility.*

La función *EvaluateStandard* (anteriormente nombrada *Evaluate*) es la misma que en la primera iteración, y al ser virtual, permite al usuario escribir su propio algoritmo para seleccionar la acción a realizar.

Tanto en la evaluación de los agentes risk-takers como en los agentes Dual-Utility, se utiliza la clase *WeightedRandom* para obtener una acción a partir de su peso respecto al total. En este trabajo, el peso de cada acción es la utilidad de esta.

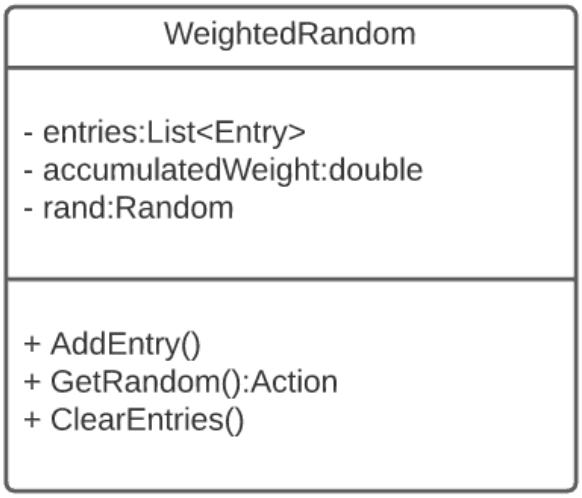

Figura 6.25 Clase *WeightedRandom*.

Como se puede observar en la Fig. 6.25, la clase *WeightedRandom*, tiene una lista de tipo *Entry*. Un *Entry* (entrada), es una estructura que contiene un atributo de tipo *double* para el peso de la entrada y un atributo tipo *Action* para guardar la acción de dicha entrada. Para que una acción sea considerada por el algoritmo, esta se debe insertar como una nueva entrada. Al realizar esta operación se añade el peso de la acción al peso acumulado de todas las acciones.

```
public Action GetRandom() {
    double r = rand.NextDouble() * accumulatedWeight;

    foreach (Entry l_entry in entries) {
        if (l_entry.ownAccumulatedWeight >= r) {
            return l_entry.action;
        }
    }
    return entries.Last().action;
}
```

Figura 6.26 Método *GetRandom*.

Para obtener la acción de forma aleatoria se utiliza el método *GetRandom*, el cual se puede observar en la Fig. 6.26. Este método devuelve la primera acción que tiene su peso acumulado por encima del valor un valor aleatorio entre 0 y el

peso acumulado total (*r*). El peso acumulado de cada entrada representa un % del peso acumulado total, por lo tanto, mayor es el peso acumulado de una entrada, más probabilidades hay de que esa entrada sea la elegida.

```
private void EvaluateRiskTaker()
{
    foreach (var l_action in agentActions)
    {
        riskTakerRandom.AddEntry(l_action, _weight: l_action.GetActionUtility());
    }

    Action l_bestAction = riskTakerRandom.GetRandom();
    riskTakerRandom.ClearEntries();

    if (l_bestAction != currentAction)
    {
        if (actionsHistory.Count >= maxHistoryLength) actionsHistory.Dequeue();
        actionsHistory.Enqueue(currentAction);
        currentAction.actionLogic.OnExit();
        currentAction = l_bestAction;
        currentAction.actionLogic.OnEnter();
        if(!currentAction.interruptible) currentAction.actionLogic.OnActionFinished += InterruptibleActionFinished;
    }
}
```

Figura 6.27 Método *EvaluateRiskTaker.*

Como se puede ver en la Fig. 6.27, se añaden todas las acciones del agente al selector aleatorio, por lo tanto, cualquier acción puede ser elegida, sin importar su utilidad. Al haber seleccionado la mejor acción se debe limpiar la lista de acciones del *WeightedRandom* y poner el peso acumulado a 0, ya que en la próxima evaluación la utilidad de las acciones habrá cambiado y, por lo tanto, la probabilidad de ser elegidas será diferente.

```
private void EvaluateDualUtility()
{
    riskTakerRandom.AddEntry(agentSorter.GetSortedActionWithUtility()[0].action, agentSorter.GetSortedActionWithUtility()[0].utility);

    for (int i = 1; i < agentActions.Count; i++)
    {
        if (Differential(_best: agentSorter.GetSortedActionWithUtility()[0].utility, _compare: agentSorter.GetSortedActionWithUtility()[i].utility) > dualRangeDiff) break;

        riskTakerRandom.AddEntry(agentSorter.GetSortedActionWithUtility()[i].action, agentSorter.GetSortedActionWithUtility()[i].utility);
    }
```

Figura 6.28 Selección de entradas en el método *EvaluateDualUtility*.

Para los agentes de tipo dual-utility, lo único que cambia respecto a los agentes de tipo risk-takers son las entradas que se añaden al selector aleatorio. Mientras que en los risk-takers se añaden todas las acciones, en los agentes dual-utility solo se añaden las acciones que tienen la utilidad en un rango respecto a la mejor acción. Como se puede observar en la Fig. 6.28, se añade la mejor acción al selector y luego se itera sobre el resto de las acciones, las cuales están

ordenadas de mayor a menor utilidad. El resto de las acciones se van añadiendo mientras su utilidad se mantenga dentro del rango. Este rango lo define el usuario mediante la interfaz de la herramienta.

Con los agentes de tipo dual-Utility se consigue obtener comportamientos pseudo aleatorios manteniendo la seguridad que las acciones que se realizan siguen siendo útiles.

Respecto a la creación de agentes, se ha implementado una nueva funcionalidad que permite crear un agente a partir de un prefab ya existente. Para crear un agente a partir de un prefab ya existente se debe seleccionar el prefab y el comportamiento asociado. El método *FromExistingPrefab* de la clase *CreateAgent* se encarga de añadir todos los elementos necesarios. Lo único que varía respecto a la creación de un agente desde cero es que se crea una instancia local del prefab en vez de crear un nuevo *GameObject*. Véase la Fig. 6.29.

```
public static void FromExistingPrefab(GameObject _prefab, string _path,Type _agentType)
{
    GameObject l_agent = (GameObject)PrefabUtility.InstantiatePrefab(_prefab);
    GameObject l_actionObject = new GameObject{name = "Actions"};
    GameObject l_considerationsObject = new GameObject{name = "Considerations"};
    GameObject l_propertiesObject = new GameObject{name = "Properties"};
    GameObject l_actionsLogic = new GameObject{name = "ActionsLogic"};

    l_agent.AddComponent(_agentType);
    l_actionObject.transform.parent = l_agent.transform;
    l_considerationsObject.transform.parent = l_agent.transform;
    l_propertiesObject.transform.parent = l_agent.transform;
    l_actionsLogic.transform.parent = l_agent.transform;

    PrefabUtility.ApplyObjectOverride(l_agent, _path, InteractionMode.AutomatedAction);
    PrefabUtility.SaveAsPrefabAsset(l_agent, _path);
    Object.DestroyImmediate(l_agent);
}
```

Figura 6.29 Método *FromExistingPrefab* de la clase *CreateAgent*.

La implementación de acciones no interrumpibles se desarrolla a partir de un sistema de eventos. La clase *ActionLogic* contiene un evento llamado *OnActionFinished* el cual es invocado mediante el método *ActionFinished*

cuando una acción ha terminado. Al invocarse este evento se llama al método suscrito a dicho evento; *InterruptibleActionFinished* de la clase *Agent*.

```
private void InterruptibleActionFinished(object _sender, EventArgs _e)
{
    currentAction.actionLogic.OnActionFinished -= InterruptibleActionFinished;
    Evaluate();
}
```

Figura 6.30 Método *InterruptibleActionFinished*

Como se puede observar en la Fig. 6.30, se elimina la suscripción al evento *OnActionFinished*. Esto es debido a que solo se realiza la suscripción cuando una acción no es interrumpible, como no es posible saber si la próxima acción será o no interrumpible, el evento debe quedar limpio de suscripciones.

En esta segunda iteración se ha cambiado el sistema de guardado para obtener una mejora de rendimiento. En la primera iteración el sistema de guardado estaba basado en el tiempo, sin tener en cuenta si se habían realizado cambios o si el usuario quería guardarlos. En esta iteración el sistema de guardado se ha dividido en dos: el guardado manual y guardado automático. El usuario puede elegir en cualquier momento que sistema de guardado desea utilizar. Si el guardado es automático, se realiza un guardado en cada cambio realizado, mientras que, si el guardado es manual, aparece un botón para guardar los cambios. Véase la Fig. 6.31.

Figura 6.31 Opción de autoguardado.

Los cambios en un agente se pueden realizar en dos niveles; a nivel de componente o a nivel de atributo. Un cambio a nivel de componente es cuando se añade o elimina un componente del agente (acción, consideración…) mientras que se considera un cambio a nivel de atributo cuando se edita un atributo de un componente en el inspector.

Los cambios a nivel de componente siempre se guardan automáticamente, aunque la opción de guardado automático esté desactivada ya que, si el usuario añade un componente, edita un atributo de este, y luego guarda los cambios de dicho atributo sin haber guardado la adición del componente añadido, se produce un conflicto.

Los cambios a nivel de atributo sí pueden ser guardados a criterio del usuario. El autoguardado detecta los cambios mediante las funciones *BeginChangeCheck* y *EndChangeCheck* de la clase *GUILayout* ofrecida por la librería *IMGUI*. La función *EndChangeCheck* devuelve cierto o falso según si se ha producido un cambio. Si se ha producido un cambio se llama al método *CheckSave* el cual va a guardar los cambios si se trata de autoguardado o cambiará el estado de *currentChangesSaved* a falso para así notificar a la interfaz de que se han producido cambios y no se han guardado. Véase la Fig. 6.32.

```
void CheckSave()
{
    if (autoSave) SaveCurrentChanges();
    else currentChangesSaved = false;
}
```

Figura 6.32 Método *CheckSave*.

Para finalizar esta iteración se han realizado cambios en la interfaz de la herramienta. Las opciones de autoguardado y ver los componentes globales se encuentran en la esquina inferior izquierda en vez de encontrarse en el encabezado, el cual se ha eliminado. Además, se ha mejorado el aspecto visual de diversos elementos de la interfaz mediante los objetos GUISkin. Estos objetos son nativos del motor *UnityEngine* y permiten realizar cambios estéticos en la interfaz sin la necesidad de editar el código. Véase la Fig. 6.33 para ver la comparación entre la interfaz de la primera iteración comparada con la interfaz de la segunda iteración.

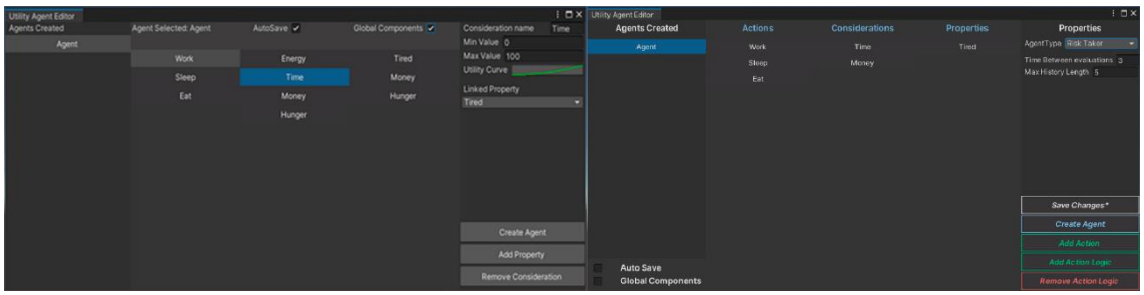

Figura 6.33 Interfaz de la herramienta en la primera y segunda iteración

6.3.2. Valoración de la iteración

La valoración de esta segunda iteración es positiva ya que de nuevo se han cumplido todos los objetivos principales y 2 de los 3 objetivos secundarios.

Aunque no se ha conseguido implementar un selector de curvas personalizado debido a la falta de tiempo, el selector de curvas nativo de *UnityEngine* ofrece una pequeña lista de las curvas más utilizadas. Si bien no es la funcionalidad ideal, la lista de curvas que ofrece puede ser útil para el usuario.

El tiempo que se invirtió en la primera iteración en diseñar la arquitectura ha permitido que los nuevos elementos desarrollados en esta iteración se integren fácilmente en la herramienta. Además, resolver los errores de la primera iteración antes de desarrollar las nuevas funcionalidades, ha conseguido identificar malas praxis en el desarrollo de estas, lo que ha reducido el trabajo en la fase de testeo de esta iteración.

En la valoración de la primera iteración se mencionó los problemas de escalabilidad que presentaba la librería *IMGUI*. Estos problemas no han sido tan graves como se esperaba ya que la carga de trabajo a nivel de interfaz ha sido mucho menor en esta iteración. Si es verdad que las clases que controlan la interfaz pueden ser extensas debido a la gran cantidad de elementos, desacoplar la arquitectura de la interfaz permite gestionar el código más fácilmente.

7. Resultados del trabajo

En este capítulo del trabajo se muestran los resultados obtenidos a lo largo del desarrollo de este trabajo. El capítulo se divide en dos secciones: Resultados de objetivos y la valoración final. En el primero se enseñan todos los resultados de los objetivos propuestos en las iteraciones. La segunda sección consiste en discutir las diversas decisiones tomadas a lo largo del desarrollo y hacer una valoración final del trabajo realizado.

7.1. Resultados de objetivos

En la siguiente tabla se puede observar una lista de los objetivos mencionados a lo largo de este trabajo. De color verde se pueden ver los objetivos implementados correctamente, de color rojo el objetivo no implementado.

ID	Nombre del objetivo
1	Implementar clase Action
2	Implementar clase Consideration
3	Implementar clase Property
4	Implementar clase Agent
5	Implementar clase ActionLogic
6	Desarrollar una interfaz para la herramienta
7	Modificar variables del agente mediante la herramienta
8	Añadir y eliminar componentes mediante la herramienta
9	Implementar Risk-Takers
10	Implementar Dual-Utility Reasoning
11	Implmentar acción no interrumpible
12	Implementar un sistema de autoguardado y guardado manual
13	Implementar una lista preestablecida de curvas

Tabla 7.1 Resultados de objetivos.

7.2. Valoración final

A lo largo de este trabajo han aparecido diversas situaciones donde se ha tenido que tomar una decisión que no fue la que se planteó inicialmente. Estas son las más relevantes:

- *ActionLogic* es una clase abstracta en vez de una interfaz.

- No reescribir el código de la interfaz a UI Elements
- Creación de un agente y añadir componente *ActionLogic* en una ventana emergente.
- Enseñar el ámbito global del agente al añadir un componente.

La propuesta original para un script *ActionLogic* es que este heredara de la clase *Monobehavior* e implementara la interfaz *ActionLogic* para así forzar la implementación de los métodos *OnEnter, OnUpdate* y *OnExit*. El problema es que *UnityEngine* no permite exponer atributos de tipo interfaz en el inspector, lo que provocaría que el atributo de tipo *ActionLogic* de la clase *Action* no fuera visible. Esto supondría un problema en el caso de que un usuario quiera editar un agente directamente desde el *prefab*. De ahí que, la clase *ActionLogic* sea una clase abstracta la cual hereda de *MonoBehavior*. Como resultado, se conserva la herencia de *MonoBehavior* y se implementan los métodos *OnEnter*, *OnUpdate* y *OnExit.*

En la valoración final de la primera iteración se mencionan los problemas de escalabilidad de la librería *IMGUI* y la necesidad de reescribir la interfaz de la herramienta utilizando la librería UI Elements. Aunque esta librería escala mucho mejor, presenta dos problemas. El primero es que la documentación de dicha librería es muy escasa debido a que ha sido lanzada recientemente lo cual ralentiza considerablemente el desarrollo. El tiempo estimado en reescribir la interfaz de la herramienta era de 2 semanas debido a la falta de familiarización con la librería, pero con la falta de documentación, el tiempo estimado para reescribir la herramienta se estableció como desconocido. El otro problema que presenta la librería es que la sintaxis de la librería varía ligeramente entre las diferentes versiones. Esto supone un problema de cara al futuro ya que la sintaxis con la que se escribe la interfaz de la herramienta puede quedar obsoleta. Ambos problemas presentan una desventaja mucho mayor desde el punto de vista de producción respecto a la escalabilidad de la librería *IMGUI*.

Para minimizar los problemas de escalabilidad mencionados anteriormente se tomó la decisión de presentar diversas funcionalidades mediante ventanas emergentes. El criterio para presentar una funcionalidad de esta forma es que

contenga varios elementos visuales que no se encuentren en la interfaz principal de la herramienta y que dicha funcionalidad sea momentánea, es decir, que no sea necesario mantener la ventana abierta para configurar el agente. Tanto la creación de un agente como la adición y supresión de *ActionLogic* cumplen ambos criterios. Presentar una funcionalidad en una ventana emergente permite escribir la interfaz de esta en una clase distinta, lo cual consigue reducir considerablemente la gestión de código en la clase que construye la interfaz de la ventana principal.

Por último, se puede observar que al añadir una acción, consideración o propiedad se habilita la opción de ámbito global del agente. Se ha tomado esta decisión debido a que cuando se añade un componente y la herramienta se encuentra en ámbito local, si no se vincula el componente añadido, este no aparece en la interfaz, ya que no es considerado del ámbito local de la acción seleccionada. Esto puede provocar cierta confusión al usuario ya que se ha añadido un componente, pero este no aparece en la interfaz. Habilitar el ámbito global permite al usuario ver directamente el cambio realizado.

La valoración final del trabajo realizado es muy positiva ya que se ha conseguido desarrollar una herramienta funcional y estable con todas las funcionalidades propuestas al inicio del trabajo. Aunque han aparecido problemas a lo largo del desarrollo, se ha encontrado una solución válida para todos ellos. La metodología de trabajo se ha seguido correctamente, aunque algunos esprints han sido considerablemente más largos que otros debido a que algunas tareas no han sido estimadas correctamente.

El resultado final es una librería con una interfaz gráfica a modo de herramienta que permite la creación e implementación de inteligencia artificial basada en utilidad.

8. Referencias

Almeida, M. y da Silva, F. (2013). Requirements for game design tools: A Systemic Survey. P. Notargiacomo (coord.), *Proceedings do XII Simpósio Brasileiro de Jogos e Entretenimiento*. Congreso celebrado en el Simposio Brasileño de Juegos en Sao Paulo, Brasil.

Apex Game Tools (2016, abril 19). *Unity Apex AI Editor Selectors*. Recuperado de: https://www.youtube.com/watch?v=1ZX_kDdz96g

ArenaNet (2012). *Guild Wars 2* (PC) [Videojuego]. ArenaNet: Bellevue, Washington.

Bates, B. (2004). *Game Design*. Boston: Thomson Course Technology.

BioWare (2014). *Dragon Age: Inquisition* (PC) [Videojuego]. BioWare: Edmonton, Canadá.

Cockburn, A. (2007). *Agile Software Development: The Cooperative Game.* (2a Edición). Londres: Pearson.

Colledanchise, M. y Ögren, P. (2018). *Behavior Trees in Robotics and AI. DOI:* https://doi.org/10.1201/9780429489105

Dill, K. (2012). Introducing GAIA: A Reusable, Extensible architecture for AI behavior. En *Proceedings of the 2012 Spring Simulation Interoperability Workshop* (pág. 14-19). URL: https://www.sisostds.org/DesktopModules/Bring2mind/DMX/API/Entries/Download?Command=Core_Download&EntryId=35466&PortalId=0&TabId=105

Dill, K. (2013). What is Game Ai? En S. Rabin (Ed.), *Game AI Pro: Collected Wisdom of Game AI Professionals (p. 3-9).* URL: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter01_What_is_Game_AI.pdf

Dill, K (2015). Dual-Utility Reasoning. En S. Rabin (Ed.) *Game AI Pro 2: Collected Wisdom of Game AI Professionals (p. 23-26).* URL: http://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter03_Dual-Utility_Reasoning.pdf

Dill, K., Ray, E., Garrity, P. y Fragomeni, G. (2012). Design Patterns for the Configuration of Utility-Based AI. D. Langelier (coord.), *Interservice/Industry Training, Simulation, and Education Conference*. Congreso celebrado en Orlando, Estados Unidos.

Graham, D. (2013). An Introduction to Utility Theory. En S. Rabin (Ed.), *Game AI Pro: Collected Wisdom of Game AI Professionals (p. 113-126).* URL: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf

Hanlon, S. y Watts, C. (2017). Dragon Age Inquisition's Utility Scoring Architecture. En S. Rabin (Ed.), *Game AI Pro: Collected Wisdom of Game AI Professionals (p. 371-379). URL:* http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter31_Behavior_Decision_System_Dragon_Age_Inquisition%E2%80%99s_Utility_Scoring_Architecture.pdf

Helmich, J. (Octubre 2015). *Introducing the Action camera*. Recuperado de: https://www.guildwars2.com/en/news/introducing-the-action-camera/

Krujis, B. (2016). Utility-AI [Software]. Recuperado de: https://github.com/Bartvanderkruys/utility-ai

Lewis, M. (2018) *Winding Road Ahead: Designing Utility AI with Curvature*. Recuperado de: https://www.gdcvault.com/play/1025310/Winding-Road-Ahead-Designing-Utility

Mark, D. y Lewis, M. (2015). *Building a Better Centaur: AI at Massive Scale* [Video]. Recuperado de: https://www.gdcvault.com/play/1021848/Building-a-Better-Centaur-AI

Millington, I. y Funge, J. (2009). *Artificial Intelligence for Games*. Boca Raton: CRC Press.

Rabin, S. (2002). AI Game Programming Wisdom (Game Development Series). Newton: Charles River Media.

Russell, S. y Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Londres: Pearson.

Smith, A., Nelson, M. y Mateas, M. (2010). LUDOCORE: A logical game engine for modeling videogames. *IEEE Symposium on Computational Intelligence and games*, 91-98. doi: https://doi.org/10.1109/ITW.2010.5593368

Toftedahl, M. y Engström, H. (2019). A Taxonomy of Game Engines and the Tools that Drive the Industry. W. Huber (ed.), *DiGRA '19 - Proceedings of the 2019 DiGRA International Conference: Game, Play and Emerging Ludo-Mix*. Congreso celebrado en el encuentro del Digital Games Research Association en Kyoto, Japón.

VanOrd, K. (Noviembre 2014). *Dragon Age: Inquisition Review*. Recuperado de: https://www.gamespot.com/reviews/dragon-age-inquisition-review/1900-6415949/