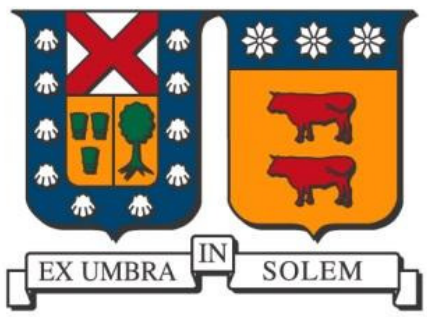
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO - CHILE



“CREACIÓN DE COMPONENTE REUTILIZABLE DE INTELIGENCIA ARTIFICIAL PARA UNITY”

DIEGO ANDRÉS NORAMBUENA VERGARA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Sven von Brand
Profesor Correferente: Nicolás Barriga

Junio - 2023

DEDICATORIA

A Francisca, por molestarme y motivarme a escribir hasta el último día.
A mi mamá y papá, quienes me dieron las bases de quien soy como persona y han permanecido conmigo toda mi vida.

AGRADECIMIENTOS

Bueno, gracias a mis profesores guía y coreferente, Sven y Nicolás. Siempre estuvieron ahí para responder mis dudas y terminar con alguna conversa interesante, incluso cuando desaparecía por harto tiempo y creía que me iban a retar. Me han ayudado a crecer tanto profesionalmente como persona.

También al equipo de Abstract Digital, con quienes compartí un año entero, entre la práctica y el trabajo. Me dieron una oportunidad única para integrarme a la industria del videojuego chileno, aprendí bastante y me gusta saber que de allí resultaron amistades muy valiosas.

A mi mamá que vivía preocupada de mi y se ha sacrificado un montón para que yo pudiera estudiar tranquilo y tuviera todo lo necesario.

Cabe darle un espacio enorme a mi novia Francisca, quien me recordaba constantemente que debía sentarme a escribir y no distraerme, pero que aparte de eso me ha dado momentos hermosos y me ha acompañado por ya casi 7 años. También, le doy las gracias a su familia, quienes me han tratado como si siempre hubiese sido parte de ella.

A mis amigos y amigas de La Ligua: Javi, Leo, Moller, Cáceres, Osorio, Morin. Y los que vine a conocer a aquí en Valparaíso: Reveco, Nico, Mapaz, Pauly, Vale, Maru, Hugo, Trakai, Fran, Catu, Toño, Toro y Pato.

RESUMEN

Resumen—

Crear comportamientos para personajes controlados por computadora (agentes o *Not playable characters*) puede resultar todo un desafío y las técnicas para hacerlo son variadas. En este trabajo se investiga acerca de las soluciones más populares para este problema, con tal de diseñar e implementar una arquitectura para la creación de comportamientos y toma de decisiones de los agentes, que fue evaluada en el marco del desarrollo del videojuego *"Mix, the forgotten"* en la empresa **Abstract Digital**. Los resultados fueron mayormente positivos, demostrando que la solución propuesta basada en sistemas de utilidad, funciona para crear comportamientos que se adaptan a las reglas del juego propuestas por los diseñadores. Además, aparecieron destellos de comportamiento emergente. Por último, sirve para comprender cuáles son las ventajas y limitaciones de este tipo de sistemas, de los cuales existe relativamente poca información sobre su implementación.

Palabras Clave— utilidad, comportamientos, inteligencia artificial, videojuegos

ABSTRACT

***Abstract*—** Creating behaviors for computer-controlled characters (agents or *Not playable characters*) can be quite challenging, and the techniques to do so are varied. This study investigates the most popular solutions to this problem in order to design and implement an architecture for creating agent behaviors and decision-making, which was evaluated within the development framework of the video game "Mix, the forgotten" at Abstract Digital. The results were mostly positive, demonstrating that the proposed solution based on utility systems works for creating behaviors that adapt to the game rules proposed by the designers. Additionally, glimpses of emergent behavior appeared. Lastly, it serves to understand the advantages and limitations of this type of system, for which there is relatively little information available about its implementation.

***Keywords*—** utility, behaviors, artificial intelligence, videogames

GLOSARIO

Agro anglicismo para definir una situación de amenaza o ataque para un agente o el jugador.

BT Behaviour Tree, árbol de comportamiento. Técnica de IA para diseñar e implementar patrones de comportamiento, construida con nodos de distintos tipos.

DOTA : *Defense of the ancients*, juego que surgió como mapa personalizado de *Warcraft III*. DOTA 2 es la versión moderna desarrollada por Valve Corporation.

DRY *Don't repeat yourself*. Filosofía de diseño que busca evitar la duplicación de procesos o lógica, en el caso de la computación.

FPS First Player Shooter, juego de disparos en primera persona. O puede ser *Frames per second*, refiriéndose al número de imágenes mostradas en pantalla por segundo.

FSM Finite State Machine, máquina de estado finito. Técnica de IA que usa estados y transiciones para crear comportamientos.

Gameplay anglicismo que se refiere a la manera en que los jugadores interactúan con el videojuego, incluyendo aspectos como las mecánicas, progresión, historia, desafíos y reglas.

Gaming anglicismo que abarca varios aspectos relacionados a los videojuegos, como las mercancías, empresas, jugadores profesionales y/o servicios de distribución de contenido.

Frame la imagen mostrada en pantalla, que representa el estado actual del videojuego. En cada frame, se lee el *input* del jugador, hacen cálculos y se proyecta la imagen.

Hack algún atajo inteligente hecho en el código para resolver un problema específico, pero que en el largo plazo puede traer problemas aún más difíciles de resolver.

IAUS Infinite Axis Utility System, arquitectura basada puramente en utilidad para la toma de decisiones y creación de comportamientos de agentes.

KISS *Keep it simple, stupid*. Filosofía de desarrollo de software que establece la simplicidad de un sistema como objetivo clave en su diseño.

NPC *Non-Playable Character*. Personaje controlado por computadora.

Out of the box característica de un producto, usualmente de tecnología, que puede ser usado fácilmente pues viene pre-configurado para correr sin problemas.

RPG Role Playing Game, juego de rol.

RTS Real Time Strategy, juego de estrategia en tiempo real.

Renderizar anglicismo para representación gráfica, proceso por el cual se genera una imagen en pantalla a partir de un modelo 2D o 3D.

Runtime tiempo de ejecución, es el periodo en el que un programa comienza a correr. En el caso de Unity, cuando se inicia el *Play Mode* en el Editor o se ejecuta la *build*.

SPA Sentir - Pensar - Actuar. El ciclo que sigue un agente inteligente cuando interactúa con su medio ambiente.

Tag etiqueta. Dentro de Unity, todos los *GameObject* poseen alguna etiqueta que sirve para identificarlos.

YAGNI *You Are'nt gonna need it*, filosofía de desarrollo de software. Indica que no se debe introducir funcionalidad extra a menos que se necesite.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

La industria de los videojuegos es un sector actualmente inmenso y que va a seguir creciendo en los próximos años. Como se puede apreciar en la figura 1, en 2020 el valor del mercado alcanzó $155.89 mil millones de dólares y se estima que alcanzará los $268.81 mil millones para el año 2025.

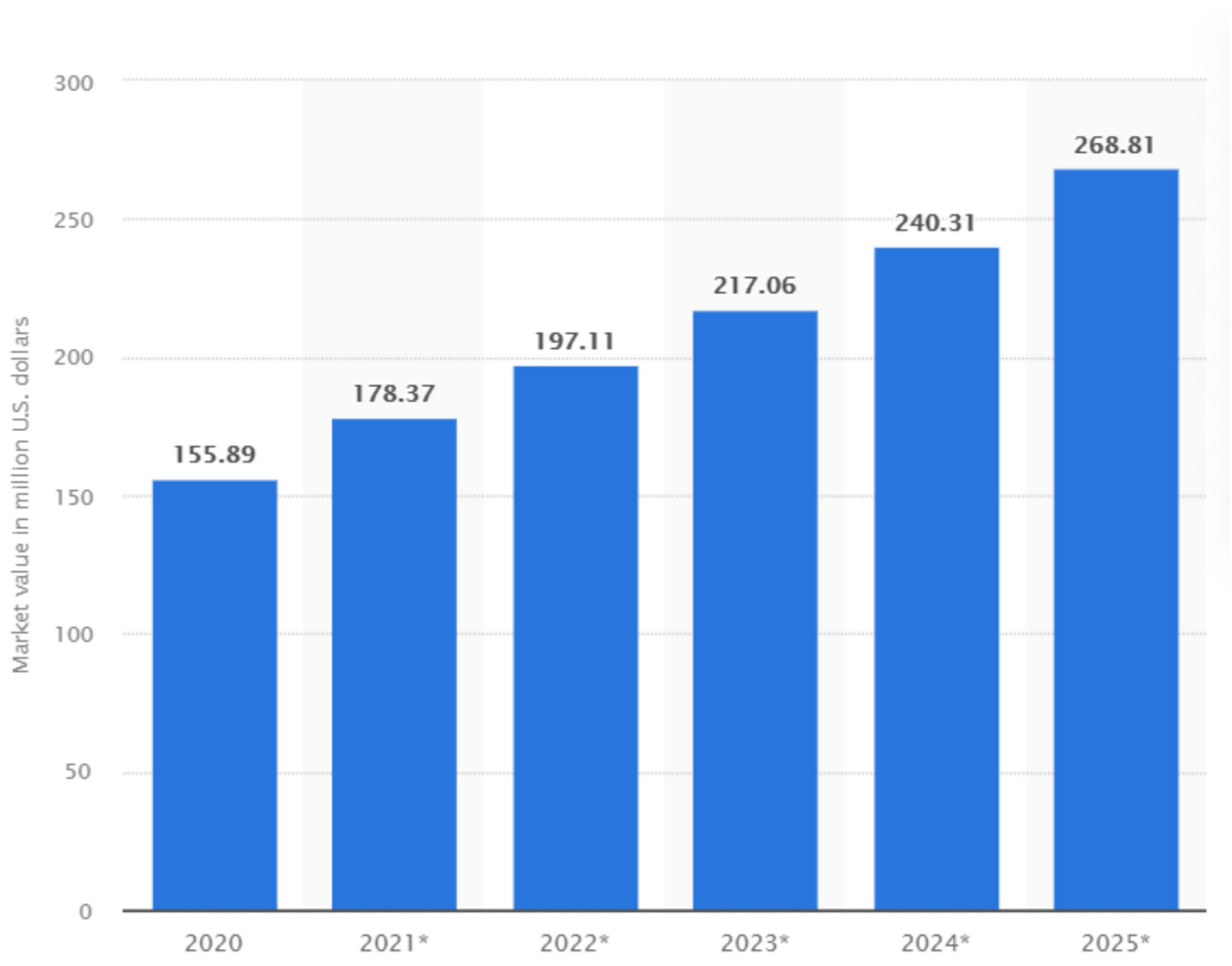


Figura 1: Valor mundial del mercado de videojuegos desde 2020 a 2025
Fuente: [Clement, 2023].

Aunque no todo fue siempre tan glamoroso. En un principio, los videojuegos nacieron en el entorno académico en la década de los 50', donde el acceso a los computadores solo estaba disponible para grandes empresas o unidades de investigación en universidades. Ejemplo de ello es *"OXO"*, conocido comúnmente como *"gato"*, implementado por el profesor británico A. S. Douglas para su tesis de doctorado en la Universidad de Cambridge en 1952. También está *"Tennis for Two"*, predecesor del famoso *"Pong"* de Atari [Smithsonian, 2016], creado en 1958 por William Higinbotham en la Biblioteca Nacional "Brookheaven", Nueva York. En estas tempranas etapas de desarrollo, ya se empezaban a aplicar las primeras versiones de **inteligencia artificial** que rivalizaban contra humanos, como ocurre en 1952, año en que Herbert Koppel, Eugene Grant and Howard Bailer construyen una máquina capaz de jugar (y ganar) *Nim* [Grant and Lardner, 1952], juego donde hay diferentes pilas de objetos y en cada turno se debe remover al menos 1 objeto de alguna pila, y gana quien se lleve el último objeto. Este interés surge debido a que los juegos poseen **reglas definidas**, las cuales crean los límites y guías para crear piezas de software capaces de seguirlas, a la vez de experimentar variaciones que difieren del pensamiento humano convencional.

No fue hasta la década de los $70'$, acompañada del nacimiento de las primeras **consolas** de videojuegos para el hogar, donde la IA se volvió una parte integral de los videojuegos, como se demuestra con *"Space invaders"* de 1978, que contaba con diferentes patrones de movimiento basados en los *inputs* del jugador y diferentes niveles de dificultad. Otros gigantes como *"Pac-man"* y *"Donkey Kong"* llegarían un par de años más tarde (1980 y 1981 respectivamente) provenientes desde Japón, país clave a la hora de hablar del desarrollo del *gaming*

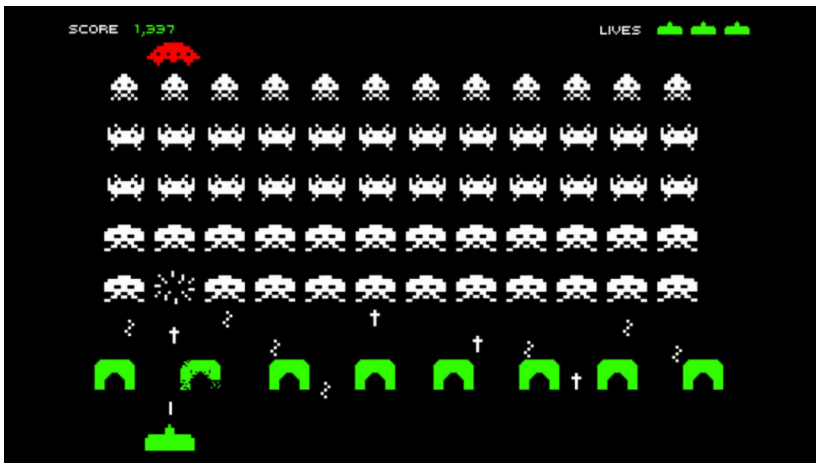


Figura 2: Videojuego Space Invaders
Fuente: MuyComputer

como lo conocemos. Gracias a que la IA se configuró como uno de los elementos importantes de los videojuegos, es que se vió impulsado este mercado y el interés de crear empresas dedicadas al rubro de desarrollarlos y publicarlos, lo que permitió diseñar videojuegos cada vez más atractivos, capaces de generar **comportamientos adaptables** a diversos estilos de juego, basándose en técnicas usadas en el ámbito académico, como las máquinas de estados finitos y algoritmos de búsquedas de caminos (como el algoritmo A*).

Ahora bien, en los últimos años se han realizado investigaciones para desarrollar IAs capaces de *aprender* cómo jugar y superar el rendimiento de los humanos en juegos de mesa y multi-jugadores en linea, utilizando técnicas de aprendizaje de máquinas como redes neuronales, auto-juego con aprendizaje por refuerzo, aprendizaje multi-agente, aprendizaje por imitación, entre otras [Deepmind, 2019]. Ejemplos de esto son *AlphaStar*, que alcanzó el nivel *GrandMaster* con las 3 razas de StarCraft II, *AlphaZero* capaz de alcanzar un nivel sobre humano en shogi, ajedrez y Go con solo 24 horas de entrenamiento [Silver et al., 2017], *OpenAI Five* que fue capaz de vencer a OG, los campeones mundiales de DOTA 2. Sin embargo, estos experimentos requieren enormes cantidades de datos y poder de cómputo para el entrenamiento de los modelos, sin ir más lejos *OpenAI Five* experimentó **45.000 años** de partidas, consumiendo **800 petaflops por segundo** todo el día [OpenAI, 2019]. Claramente, este tipo de pruebas solo son posibles para grandes empresas como Deepmind u OpenAI. Además, cualquier cambio que se quiera realizar en los modelos, implica re-entrenarlos consumiendo más tiempo y energía. En general la meta de la IA en los videojuegos no es crear personajes imbatibles que propongan desafíos casi imposibles a los jugadores, si no ser capaz de generar un ambiente donde los y las jugadoras se sientan motivadas a completar metas, ya sea venciendo enemigos o estableciendo relaciones de compañerismo con los personajes, permitiendo una progresión a lo largo del juego, maximizando la entretención y participación por períodos de tiempo más largos. También es común que personajes controlados por computadora acompañen al jugador en sus misiones, proyectando una sensación de inmersión mayor al avanzar en la historia e interactuar con tales personajes. Sin embargo, lograr que estos personajes expresen comportamientos acordes al mundo propuesto por los diseñadores es usualmente **difícil** por diversos factores detallados más adelante. Por otro lado,

es relevante mencionar que la industria nacional de videojuegos está en auge[1], lo cual abre una ventana de oportunidades para mejorar los procesos de creación y desarrollo locales, de cara a las nuevas posibilidades y desafíos que el mercado ofrece.
Es en este contexto que se enmarca el presente trabajo, donde se desarrollará una herramienta de inteligencia artificial reutilizable en diferentes proyectos, integrada en el motor de videojuegos *"Unity"*, capaz de abstraer en buena medida la complejidad de desarrollar los comportamientos de los diferentes agentes dentro del videojuego y agilizar la entrega de productos con una IA de calidad para los jugadores. La estructura del trabajo se presenta a continuación:

- **Capítulo 1, Definición del problema:** se detalla sobre las dificultades de desarrollar IA en videojuegos y entregan los objetivos del trabajo.
- **Capitulo 2, Marco Conceptual:** Se describen los principales conceptos asociados a la memoria, una discusión bibliográfica que recoge avances en la materia junto con evidencias de los problemas mencionados y una tabla con algunas herramientas ya existentes en el mercado.
- **Capítulos 3, Propuesta de solución:** se describe la arquitectura que se va a construir y se discute sobre decisiones de diseño.
- **Capítulo 4, Validación de solución:** se describen los casos de uso que fueron desarrollados para verificar la utilidad de la arquitectura propuesta y se exponen los resultados medidos.
- **Capítulo 5, Conclusiones:** se expone lo aprendido en el desarrollo, se valida si los objetivos propuestos se cumplieron y eran acordes a lo que se obtuvo, y se dan ideas de trabajo futuro.

Cabe destacar que este trabajo fue apoyado por la Agencia Nacional de Investigación y Desarrollo (ANID), Subdirección de Investigación Aplicada (SIA), proyecto ID21I10363.

[1] https://store.steampowered.com/curator/25113200-Videojuegos-Made-In-Chile/

CAPÍTULO 1
DEFINICIÓN DEL PROBLEMA

Año a año, las grandes empresas como Sony, Nintendo, Microsoft, Valve, entre otras, empujan los límites de las experiencias que los videojuegos nos proveen. Por un lado, el aumento en la capacidad de cómputo, sobre todo en las consolas de última generación como "Play Station 5" y "Xbox Series X" que llegan al orden los 9,2 y 12 Teraflops respectivamente,[2] permite mejoras en diversos aspectos: tiempos de carga de los videojuegos, gráficas de texturas e iluminación cada vez más reales, mayor tamaño de mapas, apartado musical de mayor fidelidad, animaciones fluidas con gran cantidad de partículas, entre otras características que juntas, buscan crear el ambiente óptimo para que el o la jugadora se sientan **inmersos** en el universo que los diseñadores le proponen. El principal objetivo es lograr que la persona disfrute y se entretenga con el juego.
En estos mundos, seguramente nos encontraremos con personajes controlados por computadora, llamados usualmente "agentes", o **"NPC"** (Non-Playable Character) por sus siglas en inglés. Los agentes tienen diferentes tareas, como asignar misiones, narrar partes de la historia, atacar o defender al jugador, o simplemente existir y llenar el espacio de la escena. Independientemente de sus objetivos, son una pieza clave para la inmersión, pues brindan a los jugadores la sensación de estar acompañados por entes que viven bajo las reglas del juego en que se encuentra, que se verán afectados por las decisiones de quien maneja al personaje principal. Por lo anterior, estos agentes deben exhibir un comportamiento adecuado al contexto del juego. Para ello, los diseñadores elaboran documentos donde indican qué comportamientos esperan ver en los agentes, mientras que los desarrolladores se encargan de implementarlos.
Es aquí donde se originan los primeros problemas y existen diversas perspectivas que podemos analizar respecto de la complejidad de crear comportamientos para agentes que interactúan entre si o con el jugador, listadas a continuación:

1. Como desarrollador, ser capaz de programar un comportamiento adecuado para cierta actividad del NPC, implica entender, más o menos, cómo debería hacerlo una persona real, con tal de no romper la trama incidentalmente. Por ejemplo: si vemos que un NPC decide huir del combate porque su vida es baja, entendemos esa decisión como algo razonable. Sin embargo, al encontrar patrones que se repiten constantemente en el comportamiento del agente, podemos predecirlos en minutos, destruyendo la *ilusión de inteligencia*. [Bulitko et al., 2018].

2. Tal como se indica en el artículo *"Artificial Stupidity"* [Lidén, 2003], lo que hace divertido a un juego no es necesariamente que los NPC enemigos sean más inteligentes, pues se supone que el jugador, al final, tiene que ganar. Crear una IA que gane es sencillo haciendo trampa, dado que tenemos maneras de monitorear lo que está haciendo el jugador en todo momento. Sin embargo, dejar que pierda porque la IA está mal dise-

[2] `https://hardzone.es/noticias/juegos/ps5-xbox-series-x-diferencia-potencia-rendimiento/`

ñada no es aceptable. El desafío reside en crear una IA que pierda, pero que parezca inteligente al hacerlo, mostrándole sus habilidades al jugador, pero dándole ventanas para que gane.

3. Otro problema destacado en *"Artificial Stupidity"* [Lidén, 2003] es el sobre-diseño del sistema de IA. Caer en esta trampa es común, creando complejos sistemas cuando no se necesitan. Es más un arte que ciencia, saber cuando podemos hacer pequeñas trampas en el código y cuando aplicar soluciones simples que cumplan con el objetivo: dar la **ilusión** de que el NPC está haciendo algo inteligente. Si el jugador se queda con esta impresión, la forma en que realmente se haya implementado la IA pasa a segundo plano.

4. Algunos desarrolladores no implementan las técnicas de IA conocidas en la academia, tendiendo a ser menos sofisticadas, por razones como tiempo de desarrollo acotado, desconfianza en métodos no deterministas, capacidad de cómputo de los usuarios finales, falta de conocimiento respecto a las técnicas y el hecho de que las mejoras gráficas suelen recibir más atención[Kopel and Hajas, 2018].

5. En [Saagie, 2020] se indica que la IA de los videojuegos inicialmente tenía que sortear más restricciones de cómputo, lo que dificultaba la creación de algoritmos complejos y tenía que solucionarse modificando el diseño del juego. A medida que transcurrían las generaciones, tales modelos simples de IA fueron heredándose a títulos nuevos, por lo que se explica en parte por qué ciertos sistemas de IA siguen pareciendo tan básicos frente a los que uno encontraría en investigaciones tradicionales.

6. Usualmente, las herramientas para desarrollar IA se centran en facilitar la creación de implementaciones específicas por juego [Safadi et al., 2015]. Este enfoque no aprovecha las similitudes existentes entre juegos, que incluso pueden pertenecer a diferentes géneros, lo que impide explotar una posible **reutilización** de algoritmos y módulos de código, por ejemplo, para que los agentes tomen decisiones y exhiban comportamientos.

7. Incluso existiendo herramientas para crear comportamientos de agentes, tanto diseñadores como programadores muestran una brecha de educación con los desarrolladores de IA, pues en cierto punto la complejidad del área crece hasta que se vuelve muy complicado de mantener en nuestra mente y de ser explicada en términos simples, lo que se traduce en mayor dificultad para usar estas herramientas y posteriormente entender por qué los resultados no corresponden a los esperados [Lewis, 2018]. Además, muchos de los requisitos entregados por diseñadores suelen ser vagos y es tarea del desarrollador hacer las preguntas correctas para desenmarañar aquellos detalles que inicialmente no son considerados con la importancia que merecen. Un ejemplo muy reciente de esto lo entrega Jason Storey, experimentado desarrollador de software, donde cuenta cómo un artista, colega suyo, comenzó a usar **ChatGPT** para escribir código, sin embargo al tratar de desarrollar IA no fue capaz de

obtener los comportamientos deseados, principalmente debido a la falta de conocimiento técnico en el área y la poca precisión a la hora de describir requisitos[3].

8. En [Sánchez-Ruiz et al., 2008] destacan que usualmente los mapas son diseñados de manera independiente al desarrollo de la IA, sin embargo es esta última la encargada de hacer que los personajes interactúen con el espacio, lo que puede generar problemas a la hora de lograr que el agente se mueva por el mapa, por ejemplo, que se atasque en ciertas zonas o que no pueda obtener información fácilmente sobre el tipo de terreno donde se encuentra. También destacan que las constantes variaciones en el *gameplay*, derivadas de las pruebas y/o cambios de ideas de la producción propias del juego, provocan que la creación de una IA efectiva sea un objetivo elusivo.

Considerando los puntos expuestos, este trabajo hace énfasis en desarrollar una solución para la **toma de decisiones**, es decir, aquellas técnicas que se encargan de simular una capacidad de razonamiento en base al contexto y entregan las acciones que el agente debe realizar, separándose de otras áreas específicas de la inteligencia artificial aplicada en videojuegos como la búsqueda de caminos y la generación procedural de contenido.

Las metas que se desean alcanzar con este trabajo, se alinean con los siguientes objetivos:

1.1. Objetivo general

Implementar una arquitectura reutilizable de inteligencia artificial, para la creación de comportamientos de agentes en el desarrollo de videojuegos.

1.2. Objetivos específicos

- Investigar sobre las ventajas y desventajas de diferentes arquitecturas.
- Facilitar el proceso de creación de comportamiento y toma de decisiones para agentes.
- Evaluar la arquitectura propuesta a través de una prueba de concepto en Unity.

La figura 3 presenta un árbol del problema que resume lo expuesto anteriormente.

[3] `https://youtu.be/EH9Ao_WfLD8?t=1008`

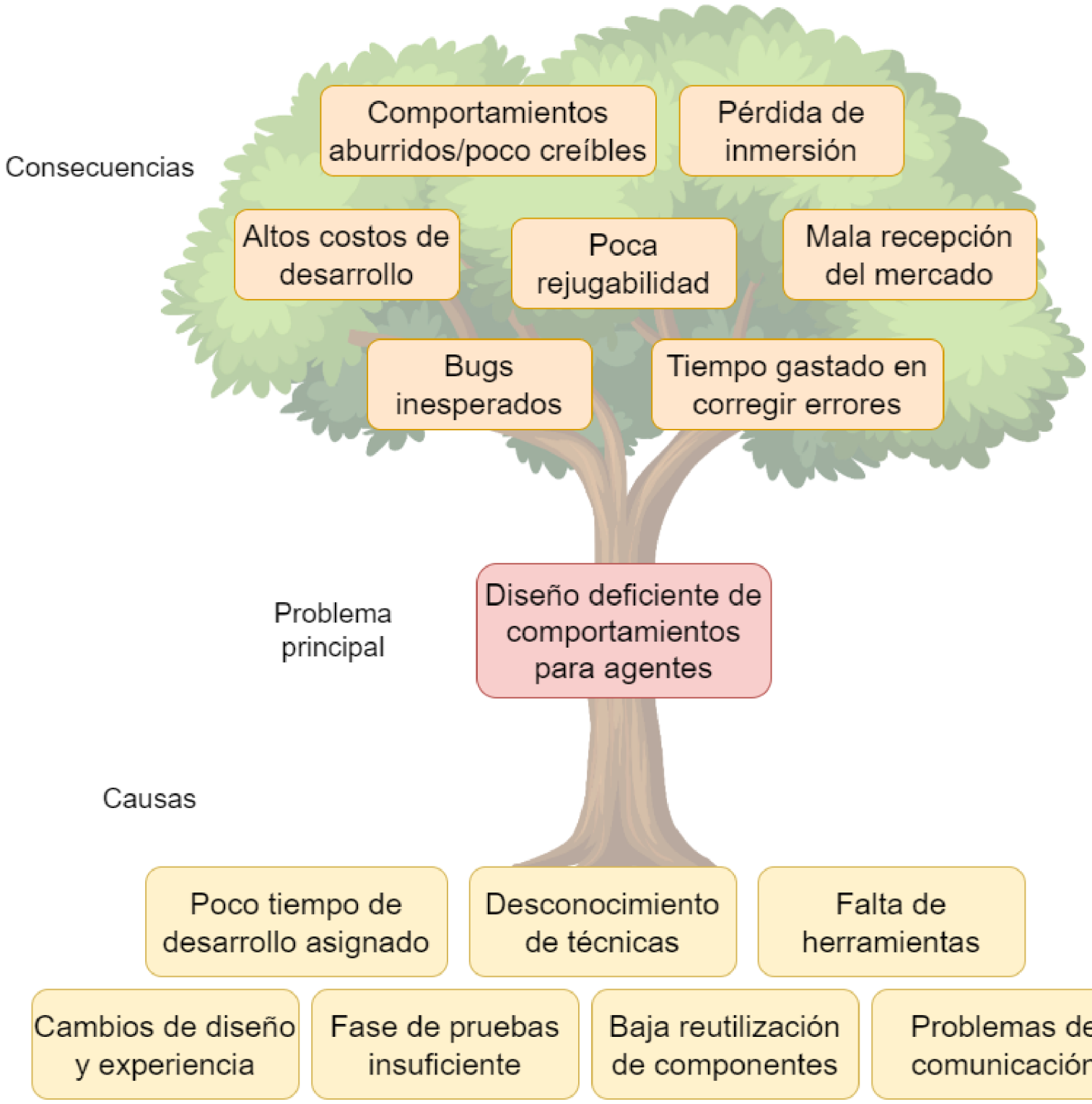


Figura 3: Árbol del problema.
Fuente: elaboración propia

CAPÍTULO 2
MARCO CONCEPTUAL

2.1. Conceptos básicos

En esta sección se revisan las definiciones más importantes, que servirán de base para la propuesta de solución detallada más adelante. Cabe destacar aquí que cuando se habla de desarrollar IA en videojuegos, no buscamos recrear la realidad (en la mayoría de los casos), si no que se buscan crear comportamientos reproducibles y consistentes frente a las decisiones del jugador, acordes al mundo del videojuego que se está planteando, donde la escalabilidad, extensibilidad y depuración son metas a conseguir [Anguelov, 2022].

2.1.1. Motor de videojuegos

Un motor de videojuegos es un software que actúa como una plataforma diseñada para construir aplicaciones interactivas [Valencia-García et al., 2016]. Incluso, podemos construir aplicaciones que no necesariamente son categorizados como juegos [Chernikov, 2018], por ejemplo: simulaciones de ambientes con agentes en un modelo cazador/presa, catálogos 3D interactivos, entornos de realidad virtual[4], entre otros.
Esta plataforma actúa como un ambiente de desarrollo, ofreciéndonos diversas herramientas, configuraciones y ajustes que simplifican, facilitan y optimizan el desarrollo de aplicaciones usando una amplia variedad de lenguajes de programación [Arm, 2021] o, si programar código no es una opción, se puede utilizar *Visual Scripting*, una herramienta que permite programar de manera visual usando grafos o bloques de código [Godot, 2020].
Un motor de videojuegos normalmente incluye una serie de motores adicionales para *renderizar* gráficos 2D o 3D, manejar las propiedades físicas dentro del juego (gravedad, velocidad, traslaciones, etc), controlar efectos de sonido, animar diferentes escenas o personajes, e incluso algunos[5] [6] incorporan módulos para desarrollar inteligencia artificial.
Existen motores de videojuegos más conocidos dada su comercialización, orientados a un uso "general", como el caso de **Unity**, **Unreal Engine**, **GameMaker**, **Godot** (para una lista extensiva ver [Wikipedia, 2020]). A su vez, las productoras de videojuegos como **Blizzard** o **EA**, cuentan con motores propios que usan en su red interna de trabajo, diseñados especialmente para los diferentes proyectos a desarrollar (RTS, FPS, RPG, entre otros).

[4] `https://unity.com/es/unity/features/vr`
[5] `https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/`
[6] `https://learn.unity.com/course/artificial-intelligence-for-beginners`

2.1.2. Unity

Es un motor de videojuegos liberado en 2005 por la empresa “Unity Technologies”. Es uno de los motores más populares para la creación de contenido interactivo, que según su reporte anual 2021[7] es utilizado por el 61 % de los desarrolladores entrevistados, además de contar con un plan de uso Personal gratuito, que permite desarrollar aplicaciones para diversas plataformas (Windows, MacOS, Linux, Android).

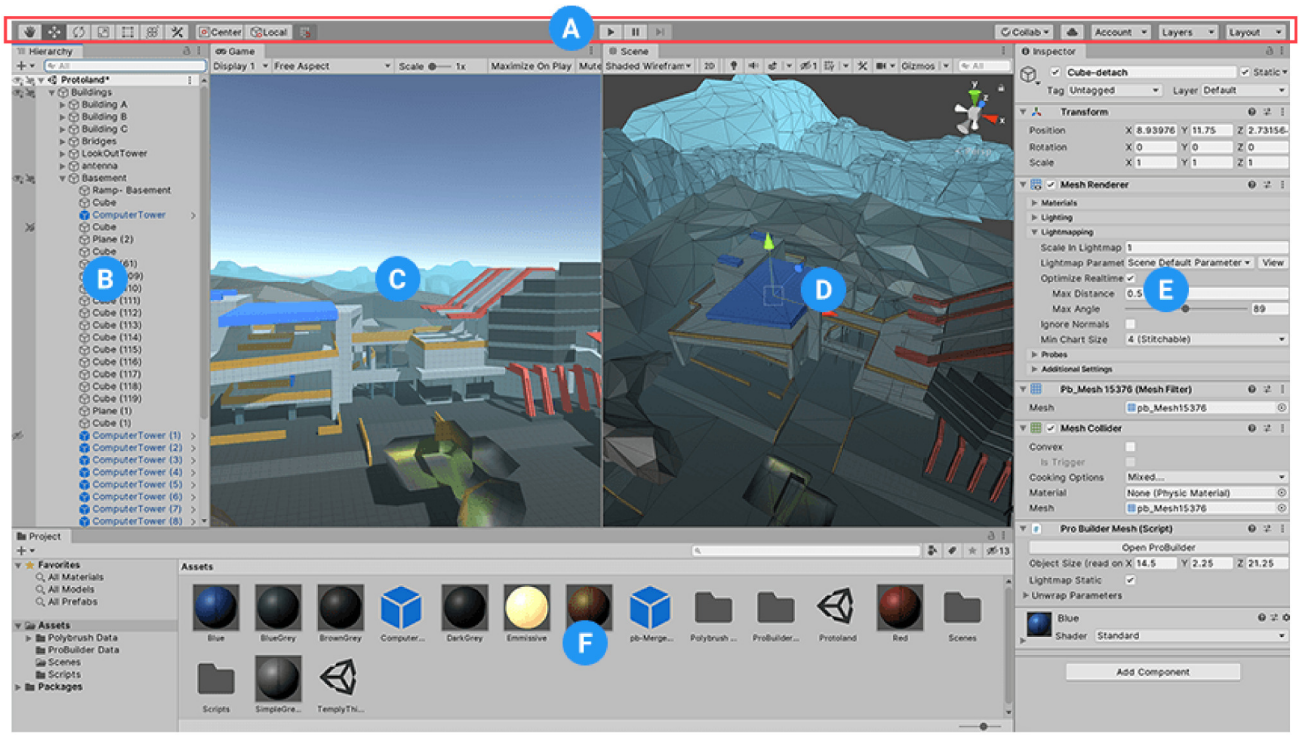


Figura 4: Interfaz de usuario de Unity. A: Barra de herramientas, B: Jerarquía, C: Vista de la escena, D: Vista del juego, E: Inspector, F: Ventana de proyecto
Fuente: Unity.

A continuación se definen algunos de los conceptos más relevantes relacionados al flujo de trabajo dentro de Unity:

- ***Assets:*** es cualquier recurso que podemos usar en nuestro proyecto, ya sea un archivo propio de Unity como de otras fuentes soportadas (scripts, modelos 3D, imágenes, audios, entre otros). Existe una tienda llamada *“Unity Asset Store”* donde los usuarios pueden compartir y adquirir diferentes *assets*, tanto gratuitos como de paga, de categoría 2D o 3D. Podemos encontrarlos en la ventana de proyecto (F) y arrastrarlos a la jerarquía (B), vista de escena (C) o al inspector(E), dependiendo del tipo de archivo seleccionado.

- ***Scene:*** una escena es un *asset* que podemos usar para separar en diferentes partes nuestro proyecto. Supongamos que cada nivel de nuestro juego es una escena nueva, con su propio ambiente, objetos, obstáculos, decoraciones e interfaz de usuario. Se puede crear cualquier número de escenas en un proyecto y se pre-visualizan en la vista de juego (D). Utilizando los botones de la barra de herramientas (A), podemos ejecutar, pausar y detener una escena. También aquí encontramos herramientas

[7] https://create.unity3d.com/2021-game-report

útiles para manejar *GameObjects*, ocultar o mostrar elementos visuales (*gizmos*) y la posición de la cámara en la vista de escena (C).

- ***GameObject:*** es cualquier instancia de un objeto dentro de una escena. Sirven como contenedores de **componentes**, los cuales podemos asignar arrastrándolos hacia el inspector (E) y editar las propiedades que exponen.

- **Componente**: comportamiento o característica que se añade a un *GameObject*. Estos definen la manera en que el objeto al cual están vinculados interactúa con el resto del mundo. Obligatoriamente, cada *GameObject* posee un componente llamado *Transform*, el cual maneja la posición, rotación y escala dentro de la escena, además de habilitar la **paternidad** entre objetos, lo cual permite que objetos sean hijos de otros, característica fundamental de Unity [Unity, 2020b].

- ***Monobehavior***: es la clase de la cual heredan inicialmente todos los *scripts* que se crean en Unity. Además, provee la capacidad de adjuntar estos *scripts* a los *Game Objects* en el inspector [Unity, 2020c].

- ***Coroutines:*** es una característica del motor, no del lenguaje C#, que permite distribuir lógica del juego a lo largo de diferentes *frames*, a través de métodos con retorno de tipo *IEnumerator*. Es importante destacar que estos métodos no son hilos (*threads*), por lo que se siguen ejecutando en el *main thread* de Unity y es importante evitar operaciones bloqueantes, como ciclos infinitos sin condiciones de salida [Unity, 2020a].

- ***Scriptable object***: es un tipo de dato muy útil para guardar información en Unity. Permite editar sus valores en *runtime*, con la particularidad de que estas modificaciones persisten en el disco luego de terminar la ejecución en el Editor. Además, al ser un *asset*, diferentes instancias de otras clases pueden referenciarlo, ayudando a evitar la duplicación de datos en memoria.

- ***Custom Editor***: dentro de Unity existen clases especiales que solo sirven cuando se trabaja en el editor. Estas alteran la manera en que se muestran los miembros de una clase en el inspector, además que permite añadir cualquier lógica que uno desee

Algunos ejemplos de los juegos más populares creados con este motor son: “Cult of the Lamb” (2022), “Genshin Impact” (2020), “Fall Guys” (2020), “Among Us” (2018), “Hollow Knight” (2017), “Cuphead” (2017), “Pokemon Go” (2016), “HearthStone” (2013).

2.1.3. Agente

Un agente es cualquier entidad autónoma capaz de obtener información de su ambiente y tomar decisiones basadas en tal información para lograr uno o más objetivos [Franklin and Graesser, 1996]. Normalmente este concepto se asocia a los mismos NPC que podemos observar en el mundo, pero puede ser algo más abstracto como ocurre en los juegos de estrategia (*“Age of Empires”* o *“Starcraft”*), donde el agente es un sistema encargado

de manejar distintos grupos de NPC para conseguir recursos, atacar y defenderse de enemigos, controlar el terreno, etc.

En el libro “Artificial Intelligence: A modern approach” [Russell and Norvig, 2010], podemos encontrar una abstracción de lo que sería un agente, donde su estado interno queda aislado del resto del ambiente y puede captar datos del medio a través de sus sensores, para así razonar y actuar en consecuencia con la información que posee en el momento. Este pro-

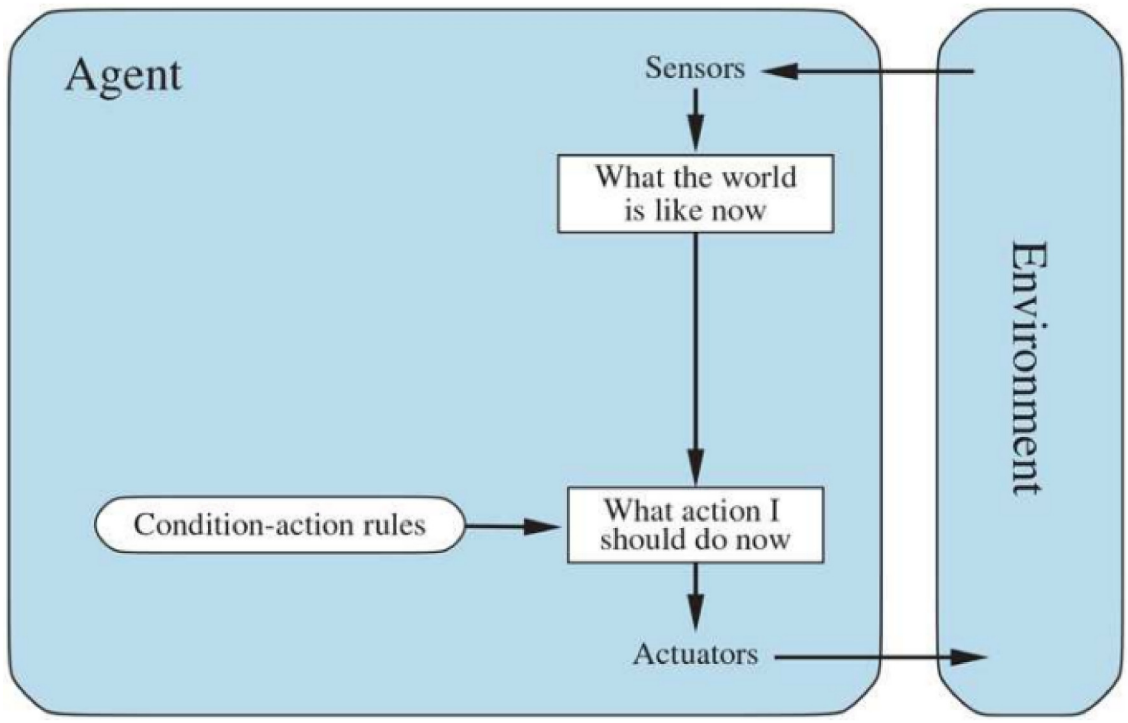


Figura 5: Agente capaz de razonar y actuar
Fuente: [Russell and Norvig, 2010]

ceso de Sentir - Pensar - Actuar (SPA), si bien tiene su origen en el ámbito de la robótica [Siegel, 2003], se ha utilizado ampliamente en el desarrollo de IA para videojuegos, pues es una manera sencilla y robusta de separar responsabilidades a la hora de programar las piezas de código que le darán vida al agente.

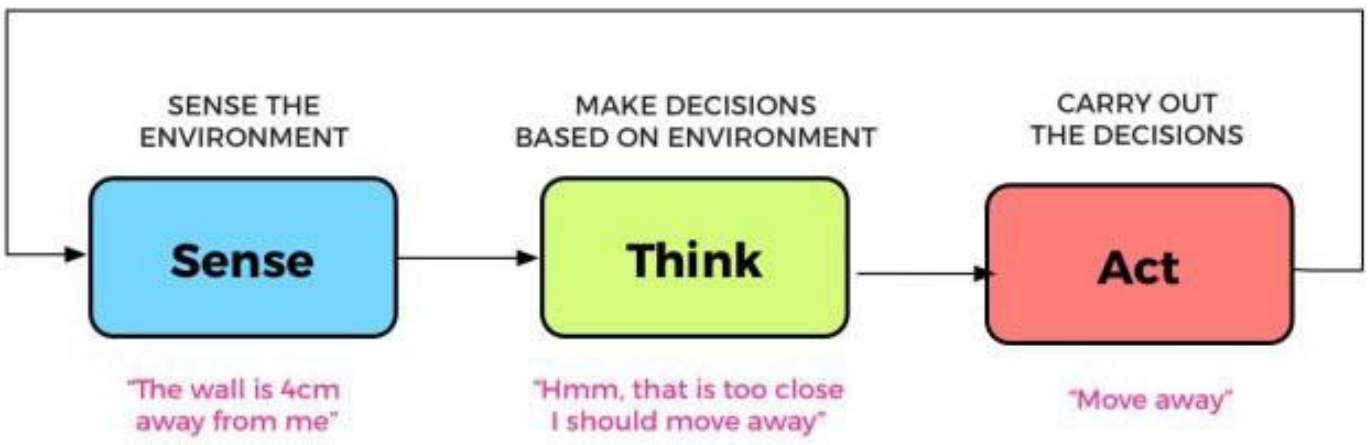


Figura 6: Ciclo SPA
Fuente: [Daeller, 2018]

2.1.4. Estado del juego

Un estado es una configuración única del ambiente donde se localiza el agente, considerando componentes audiovisuales y lógicos. Tal estado, puede cambiar según las acciones del agente o del jugador y existe un conjunto de estados posibles a los cuales un agente puede acceder, llamado **espacio de estados** [Saagie, 2020]. Esta noción es importante pues lo que buscan los métodos de IA, en esencia, es explorar este espacio de estados con tal de encontrar la manera más adecuada de actuar, ante ciertas variaciones del ambiente. Según el contexto y el diseño del juego, pueden existir muchos estados posibles para los agentes, según las propiedades que consideremos relevantes, por ejemplo: estar inactivo, atacar, correr, curar vida, cazar, perder de vista al jugador y una larga lista de otras opciones. Definiendo todas estas situaciones donde el agente puede encontrarse, podemos crear las diferentes acciones que permiten transicionar entre los estados. Esto es, crear un **comportamiento** capaz de responder y elegir las acciones más acordes según el ambiente actual. En ciertos casos, como el ajedrez, donde el espacio de búsqueda es del orden de 10^{47}, recorrerlo completamente se vuelve infactible, por lo que se deben utilizar métodos incompletos más rápidos, pues no podemos mantener ocupada la CPU por demasiado tiempo. Incluso, en el caso de los videojuegos, podemos hacer trampa, permitiendo a las entidades acceder directamente a información que en una situación más realista no sería posible que obtuvieran. Aquí es importante mencionar el concepto de ***Blackboards***, que dentro del contexto de videojuegos, son estructuras donde se guardan datos que uno o más sistemas de IA necesitan, permitiendo la interacción entre estos. Usualmente, se implementan 2 tipos, una local para guardar información propia del agente a la que solo puede acceder este y otra global, que puede ser vista por cualquier entidad [Dill, 2013].

2.2. Principales arquitecturas de IA

A continuación se revisan las arquitecturas más populares usadas en la industria, considerando pros y contras de cada una, junto con ejemplos de aplicaciones en videojuegos lanzados al mercado.

2.2.1. Finite State Machine

Una máquina de estado finito (FSM por sus siglas en inglés) es un modelo matemático de computación usado para simular lógica secuencial. Se compone por nodos que representan a los estados y aristas que los unen, que representan a los eventos que permiten transicionar entre estados [Jagdale, 2021]. Los FSM sirven para representar el flujo de ejecución en un juego. En el caso de los NPC, podríamos decir que su “cerebro” puede representarse usando una FSM, tal como vemos en la figura 7. Cada etiqueta permite conocer cuándo debería ocurrir cierta transición entre un estado y otro. Para cada objetivo, se define uno o más estados, que dictan las acciones del NPC. Los FSM permiten construir comportamientos rápidamente

y gracias a que son representables como grafos, facilitan el seguimiento del curso de acción dado un estado inicial y una serie de eventos. Además, disminuyen la probabilidad de encontrar *bugs* en el código, pues solo un estado puede estar activo a la vez. Sin embargo, cuando la cantidad de nodos y aristas crece, el grafo se vuelve muy complejo de *debuggear*, pues más acciones y transiciones deben ser definidas, intrincando las relaciones entre nodos, los cuales se vuelven fuertemente interdependientes.

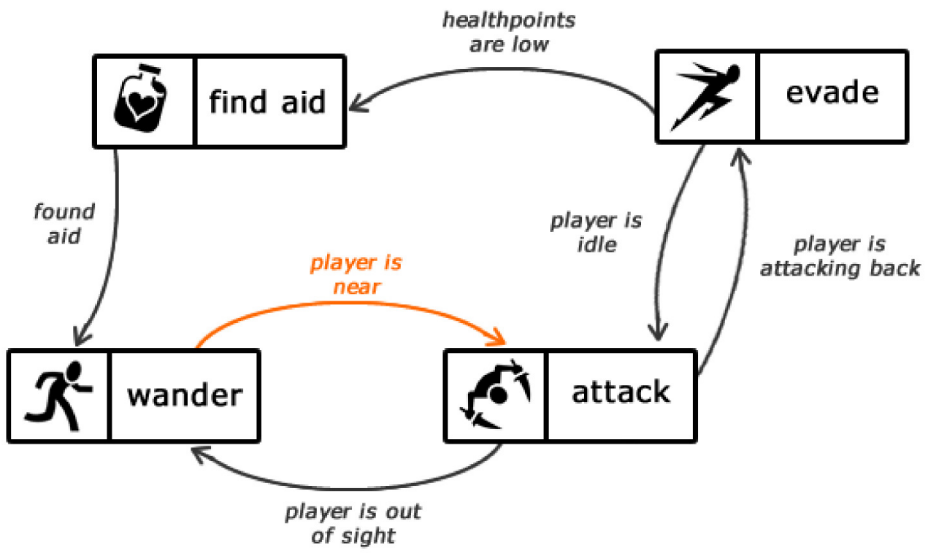


Figura 7: Máquina de estado finito de un NPC
Fuente: [Bevilacqua, 2013]

Por otro lado, un FSM no se adapta a nuevas situaciones (si bien puede introducir aleatoriedad entre ciertas transiciones), por lo que se vuelve predecible tras algunos periodos de juego. Esto puede o no corresponder a la experiencia de juego que se desea entregar. Por ejemplo, en batallas de jefes como las de *Sonic, the hedhedog* para *Sega Genesis*[8], es fácil reconocer el patrón de movimiento y ataque, por lo que el jugador en pocas pasadas puede derrotar al Dr. Eggman. Por otro lado, las FSM adquirieron importancia en otros aspectos del desarrollo de videojuegos, como el control de animaciones. Unity posee un componente *Animator*, el cual representa las diferentes animaciones y transiciones de un personaje como una máquina de estados[9].

Existe una variación conocida como *"Hierarchical FSM"*, la cual mejora algunos aspectos como modularidad y reusabilidad, con el uso de *sub-estados*. Esta idea se sustenta con el hecho de que casi todas nuestras acciones siguen una cierta jerarquía [Nirwan, 2021], que parte de algo abstracto y puede ser descompuesta en acciones simples concretas, por lo que resulta natural agrupar estados relacionados por su naturaleza, en estados de más alto nivel.

Algunos títulos que implementan esta arquitectura son: "Tomb Raider" (2013), *"Batman: Arkham Asylum"* (2009), "Half-Life" (1998), "Pacman" (1980).

[8] `https://www.youtube.com/watch?v=d8HsF338s90`
[9] `https://docs.unity3d.com/Manual/AnimationStateMachines.html`

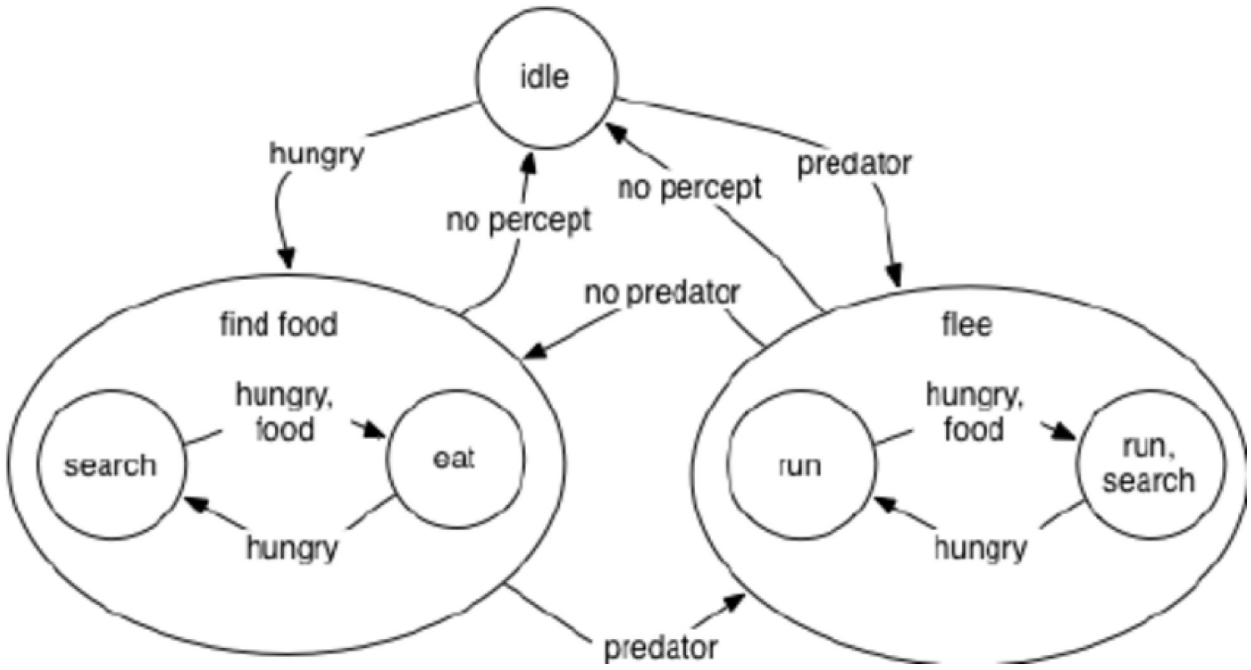


Figura 8: Máquina de estado jerárquica
Fuente: [Heckel et al., 2010]

2.2.2. Behaviour Trees

Un behaviour tree (BT) es otro modelo matemático de ejecución de planes, usado principalmente en videojuegos y que ha ganado popularidad en la robótica ([Ogren, 2012],[Bagnell et al., 2012], [Iovino et al., 2022]). A día de hoy, es la herramienta más utilizada para la creación de comportamientos y toma de decisiones. Son similares a los HFSM, pero su elemento principal de construcción son las **acciones** (a diferencia de los estados). Este modelo describe las transiciones entre diferentes acciones de manera modular, representadas en forma de árbol (ver figura 9). Esta manera de organizar las acciones y transiciones, permite crear comportamientos complejos a base de acciones simples, abstrayéndonos de cómo están implementadas. Estos árboles poseen diferentes tipos de nodos [Saagie, 2020]:

- **Hoja**: no tienen descendientes y representan acciones concretas que el agente ejecutará y que el jugador verá (nodos morados).
- **Compuestos**: poseen al menos un descendiente. Los 3 más comunes son:
  - **Secuenciales**, que deben ejecutar todos sus nodos descendientes en orden (nodos *"Chase player"* y *"Patrol"*).
  - **Selectores**, que permiten ejecutar el primer descendiente que sea válido (nodo *"AI State"*).
  - **Paralelo** que tal como indica su nombre, ejecuta todos sus descendientes al mismo tiempo, esperando por el éxito/fracaso de todos/alguno, según la implementación [Rabin, 2019].
- **Decoradores**: añaden más capacidades a los nodos anteriores, como por ejemplo indicar cuántas veces se ejecutará o cuánto tiempo se le asignará para su ejecución

[Yannakakis and Togelius, 2018]. También se pueden usar para verificar si es válido ejecutar el nodo y en caso contrario, lo descarta. En la figura 9, corresponde al nodo de color azul. Existe una variación definida como **Servicio** que chequea cada cierto tiempo si se cumplen condiciones específicas, reclamando el control de la evaluación donde sea que se encuentre y llevándolo hasta su nodo, cuando el estado del juego lo amerita, como por ejemplo verificar el *agro* de un npc [Rabin, 2019].

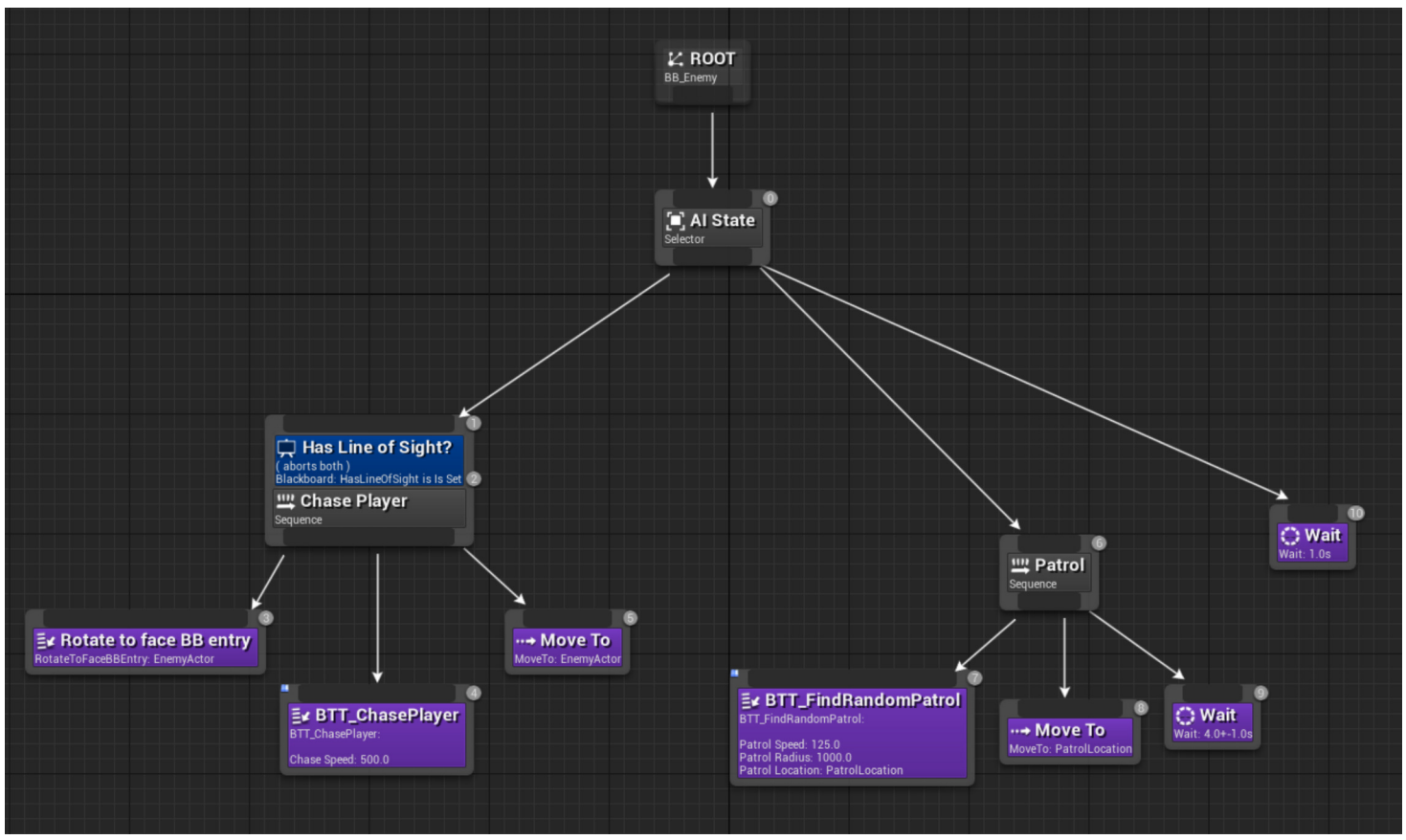


Figura 9: Behaviour tree básico
Fuente: Unreal Engine

Un BT convencionalmente se ejecuta de arriba hacia abajo, de izquierda a derecha, empezando por el nodo raíz, indicando la prioridad existente entre cada comportamiento. Cada nodo es evaluado en orden y puede entregar una de 3 respuestas: éxito, fracaso o trabajando: si es éxito, el nodo devuelve esta respuesta a su padre y este mensaje se puede propagar hacia arriba. Para que un nodo secuencial entregue éxito, cada descendiente tiene que haber entregado éxito, mientras que en el selector, basta con que alguno sea exitoso. En caso de llegar de vuelta a la raíz, el árbol comienza nuevamente con la evaluación; si es fracaso, en el caso del secuenciador, inmediatamente retornará este resultado, en cambio el selector seguirá probando sus descendientes hasta agotar sus opciones. Si bien cuentan con limitaciones similares a los FSM (fijos, predecibles, escalabilidad), su arquitectura los hace menos propensos a errores y facilita el desarrollo, ya que es más amigable para personas que no necesariamente sean desarrolladores [Rabin, 2019].

Es importante notar que existen varias limitaciones de esta solución, algunas detalladas por el desarrollador de Valve, Robert “Bobby” Anguelov en sus charlas [Anguelov, 2019] [Anguelov, 2022], principalmente porque se está modelando un comportamiento cíclico (del agente) con una estructura acíclica como el **BT**, por lo que se necesitan demasiados *hacks* para resolver problemas como interrupciones de acciones.

El juego que popularizó esta arquitectura es *Halo 2* (2004) con la charla de Damian Isla, desarrollador principal de la arquitectura. Otros títulos relevantes que utilizan BT son: *Alient: Isolation* (2014), *Far Cry 4* (2014), *Bioshock: Infinite* (2013), *Spec Ops: The Line* (2012), *Halo 3* (2007) [Thompson, 2019]

2.2.3. Utility based

Un sistema de utilidad permite modelar comportamientos de los agentes, comparando los beneficios relativos entre diferentes acciones, para encontrar el "mejor" curso de acción a seguir. Tal como indica [Graham, 2013]: "la idea central es que cada acción o estado dentro de un modelo, puede describirse con un valor único y uniforme". La utilidad, término derivado del campo económico, es una medición de la satisfacción que nos entrega cierto elemento ante una necesidad [Yannakakis and Togelius, 2018]. Se traduce en puntajes asignados a las diferentes acciones que un agente puede ejecutar en cierto momento, basándose en el contexto actual del mundo más el estado interno del agente, y escogiendo alguna de estas mediante una heurística, que usualmente es tomar aquella acción con el puntaje más alto. De esta manera, el agente puede reaccionar a los cambios de su ambiente en una manera que parece lógica y no necesita ser explícitamente definida por los diseñadores, lo que permite obtener ventajas sobre métodos de selección más estáticos (como los mencionados arriba), por ejemplo añade re-jugabilidad ya que los agentes no actuarán necesariamente de la misma manera si es que el contexto cambia en las siguientes ejecuciones, a la vez que genera un interés en los jugadores por entender cómo funciona la IA y entrega flexibilidad a los desarrolladores, pues no es necesario definir transiciones explícitas entre todos los posibles estados del agente [Graham, 2013]. Además, este sistema permite tomar decisiones incluso en casos donde las opciones son poco deseables, hasta inválidas, lo que evita que el agente se quede atascado. Esta técnica suele asociarse con la aparición de comportamientos emergentes [Graham, 2019], entendiéndose como aquellas interacciones entre agentes y su ambiente, que posiblemente no se diseñaron a propósito, pero que dentro del contexto del juego se sienten naturales e interesantes, o por el contrario, pueden salirse de la propuesta de diseño, por lo que es un arma de doble filo que requiere iterar varias veces para encontrar configuraciones apropiadas a la experiencia que se pretende brindar.

Una arquitectura propuesta por Dave Mark, llamada Sistema de Utilidad de Ejes Infinitos (IAUS por sus siglas en inglés), define una manera de seleccionar comportamientos puramente basada en utilidad [Mark, 2013]. En esta arquitectura, cada acción que el agente puede ejecutar posee diferentes **consideraciones** que debe evaluar para encontrar el puntaje final que tendrá. Una consideración es básicamente una variable del estado actual del juego importante para la acción en particular que desea evaluar y comparar.

Por ejemplo, si tenemos una acción llamada **"Sanarme"** que reproduce una animación y restaura puntos de vida, suena lógico *considerar* cuál es el porcentaje de vida actual del agente para calcular la utilidad de tal acción, pues no es lo mismo sanarme si tengo 95 % de vida que si tengo 5 % restante, así como si tenemos enemigos cerca o la distancia a algún objetivo particular. Podemos añadir todos los factores que creamos relevantes en la etapa de diseño

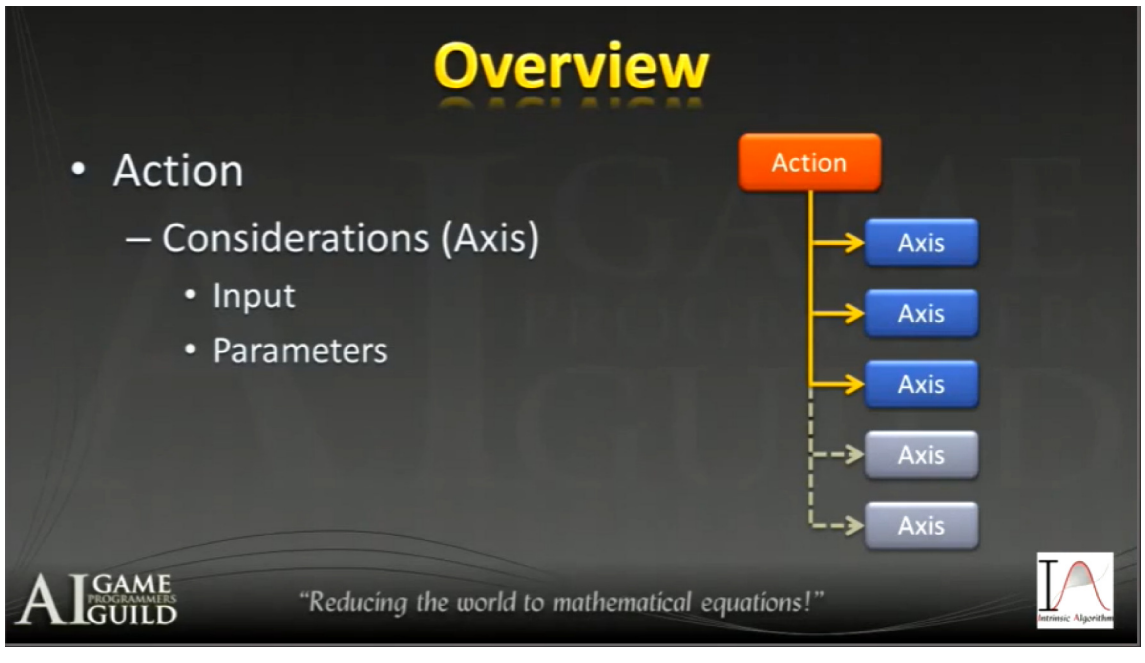


Figura 10: Una acción de IAUS
Fuente: [Mark, 2013]

(ejes). Estas consideraciones se encargan de convertir un valor específico del contexto (*input* usualmente numérico, como salud, distancia, dinero, stamina), a otro valor **normalizado** y posteriormente evaluarlo a través de una **curva de respuesta**. Una vez que se calculan los valores de cada consideración, se multiplican entre si para dar el puntaje final del comportamiento. La razón de que estén normalizados, es que así se ocupa una escala común para los comportamientos, permitiendo la comparación entre ellos, aunque se podría usar cualquier otra escala mientras se mantenga la consistencia, junto a que se pueden definir *bookends*, que son rangos distintos para normalizar cada consideración. Por ejemplo, si se tiene una consideración para alguna acción cualquiera, que evalúa la distancia entre el agente y algún objetivo, se pueden tomar valores de entrada entre [0-20], donde 0 es que el objetivo está prácticamente en la misma posición del agente y 20 sería el límite del agente, simulando un rango de visión o detección de otro tipo. Todo lo que estuviera a una distancia superior a 20 unidades, será interpretado como muy lejano, truncando el valor hasta ese límite.

Luego de haber normalizado el valor, se evalúa en la curva de respuesta y si alguno de estos valores es 0 se interpreta que por alguna razón este comportamiento no es válido, eliminándose automáticamente de la ronda de decisión actual, optimizando la selección. Por otro lado, la curva de respuesta puede ser cualquier función que consideremos adecuada para la experiencia que se propone entregar. Un ejemplo que lo ilustra es el nivel de alerta de los agentes: este aumenta conforme otros agentes o el jugador se acercan para atacarlo, sin embargo, no es lo mismo que un agente se acerque a una distancia "prudente", por ejemplo unos 20 metros, que si está a medio metro, donde fácilmente nos puede atacar. Para representar esta idea, podríamos utilizar una curva lineal como se ve en la figura 11. Sin embargo, aquí se esconde un detalle importante: según esta curva, el incremento en el nivel de alerta es el mismo para un agente que se acerca de 2 a 1 metro de distancia, como para uno que se acerca de 20 a 19 metros. Esto puede o no corresponder a la experiencia que se quiera entregar, pero lo más probable es que en realidad sea mucho más importante para el agente el primer caso. Para representar esto, una curva cuadrática o polinomial de mayor grado po-

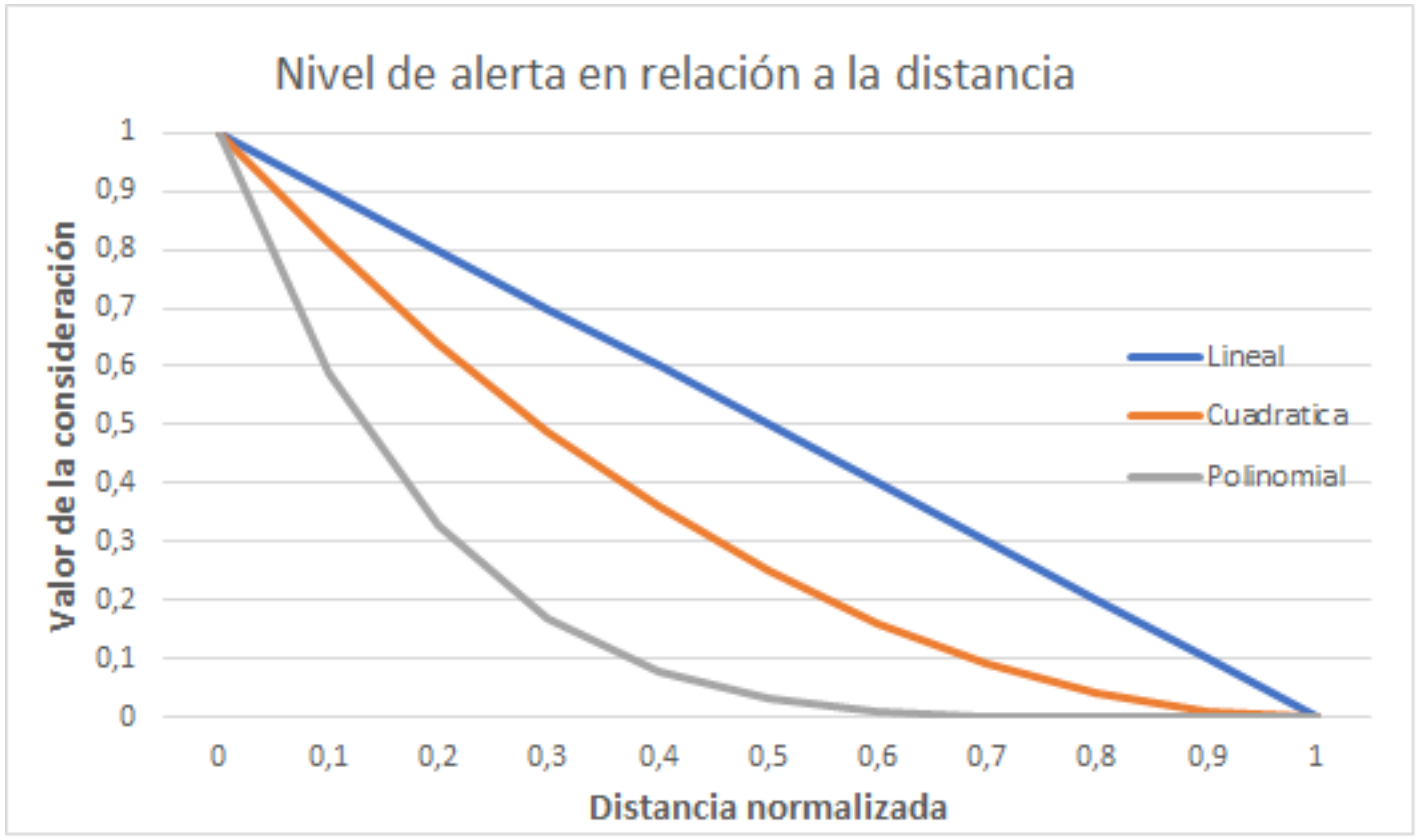


Figura 11: Diferentes curvas de respuesta para evaluar la distancia
Fuente: elaboración propia

drían ser mejores candidatas, puesto que cuando la distancia es poca, aumenta mucho más rápido el valor entregado por la curva.

Estas curvas poseen la ventaja de que cualquier persona puede dibujarla, siendo una pieza clave para dar flexibilidad a la arquitectura, además de proveer una manera sencilla de modificar comportamientos sin tocar el código. Otro punto importante es que estas curvas constituyen la personalidad de los agentes, por lo que modificándolas podemos alterar percepciones subjetivas (para los jugadores), como que tan miedoso o valiente pareciera ser un personaje, basándonos en las acciones que suele ejecutar más seguido debido a los puntajes obtenidos a través de las curvas [Rabin, 2019].

Cabe mencionar que esta arquitectura aborda el problema de decidir a qué objetivo se aplica cierta acción, en caso de tener múltiples opciones, como en el caso de disparar, buscar cobertura, recolectar objetos etc. Para esto, se calcula el puntaje de cada par acción-objetivo, usando consideraciones comunes, y se añaden por separado a la lista de opciones disponibles a ejecutar. De esta manera, se posibilita al agente cambiar de objetivos, hacia aquellos que tengan mayor utilidad, en tiempo de ejecución, cuando el estado del juego lo amerita.

2.2.4. Goal-Oriented Action Planning

Este sistema de Inteligencia artificial fue implementado en el juego *“F.E.A.R.”*, lanzado en el año 2005. Desde entonces, se popularizó y ha sido aplicado en diferentes juegos como *“Silent Hill: Homecoming”* (2008), *“Tomb Raider”* (2013), *“Middle Earth: Shadows of Mordor”*

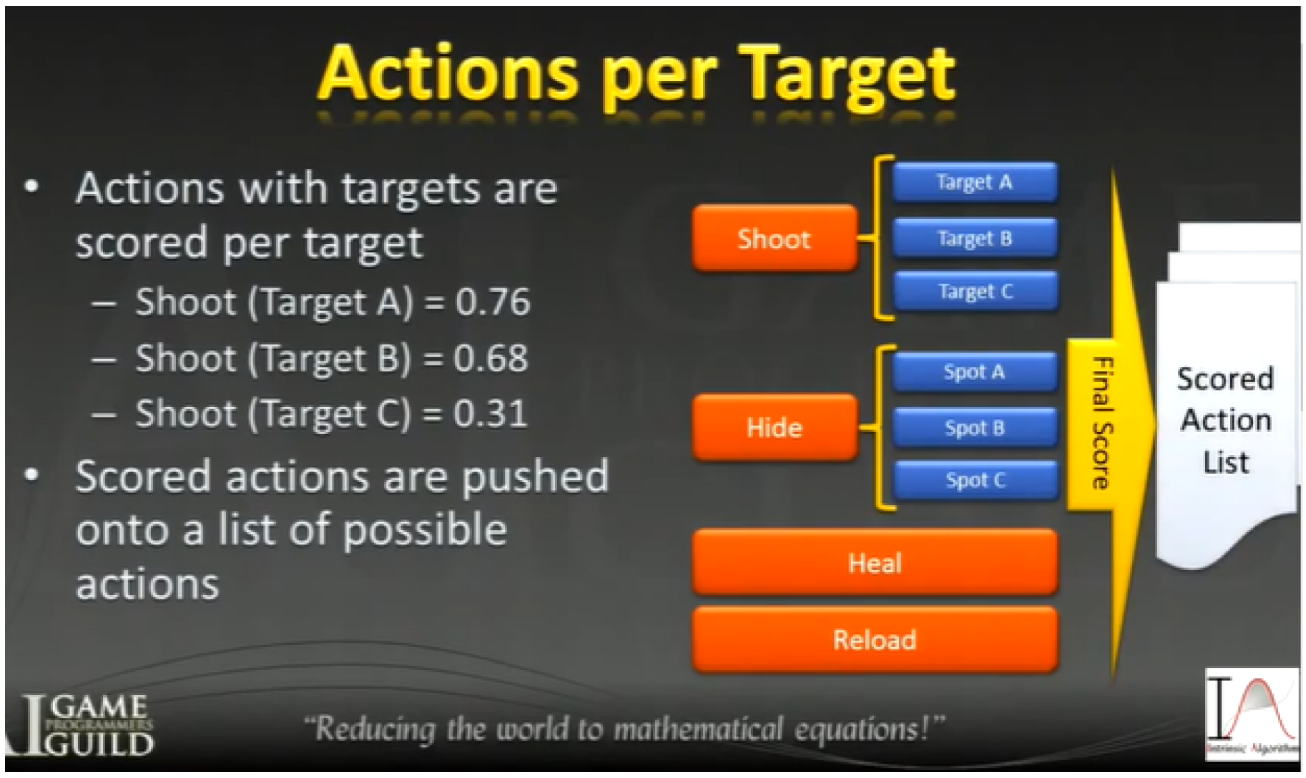


Figura 12: Evaluación de acciones por objetivos
Fuente: [Mark, 2013]

(2014), entre otros. De acuerdo a su desarrollador principal, Jeff Orkin, es una arquitectura de planeación simplificada derivada de STRIPS, un solver de problemas desarrollado en el Instituto de Investigación de Stanford, diseñado para controlar personajes autónomos de videojuegos en tiempo real [Orkin, 2006].

Esta arquitectura funciona de la siguiente manera: en cada momento, el agente se encuentra en un **estado**, el cual consiste en la combinación de valores de las diferentes variables que lo representan. Por ejemplo, un agente podría considerar dentro de su estado, variables como su salud, dinero, energía y posición, las cuales tendrán valores definidos durante la ejecución del juego. Luego, diferentes tipos de eventos (tick de reloj, jugador en rango de vista, etc) provocarán que el agente quiera cambiar su estado, es decir, cambiar el valor de alguna de sus variables para cumplir una **meta**. Para lograrlo, el agente debe elaborar un plan que permita llevarlo de su estado inicial **A** al estado deseado **B**, donde la meta está satisfecha. Este plan estará compuesto por una secuencia de **acciones**. Estas acciones son la pieza clave de la arquitectura. Están compuestas por pre-condiciones y efectos, donde las primeras son un conjunto de variables que deben ser ciertas en un determinado momento para que la acción sea válida, mientras que las segundas reflejan el cambio que tal acción provocará en el estado del agente. Se puede hacer una analogía al juego de Dominó, donde podemos encadenar fichas que tengan la misma pinta, tal como podemos unir acciones siempre que los efectos de la primera coincidan con las pre-condiciones de la siguiente (ver figura 13)

Volviendo a la elaboración del plan, este puede visualizarse como una búsqueda de caminos en un grafo, entre los estados (nodos) **A** y **B**, donde las acciones representan a las aristas. La búsqueda suele realizarse con el algoritmo **A***, pues a cada acción se le asigna un costo (tiempo, dinero, energía), lo que permite encontrar un plan óptimo y que en teoría, es lógico desde el punto de vista humano. Si existe más de un plan válido con el mismo costo mínimo,

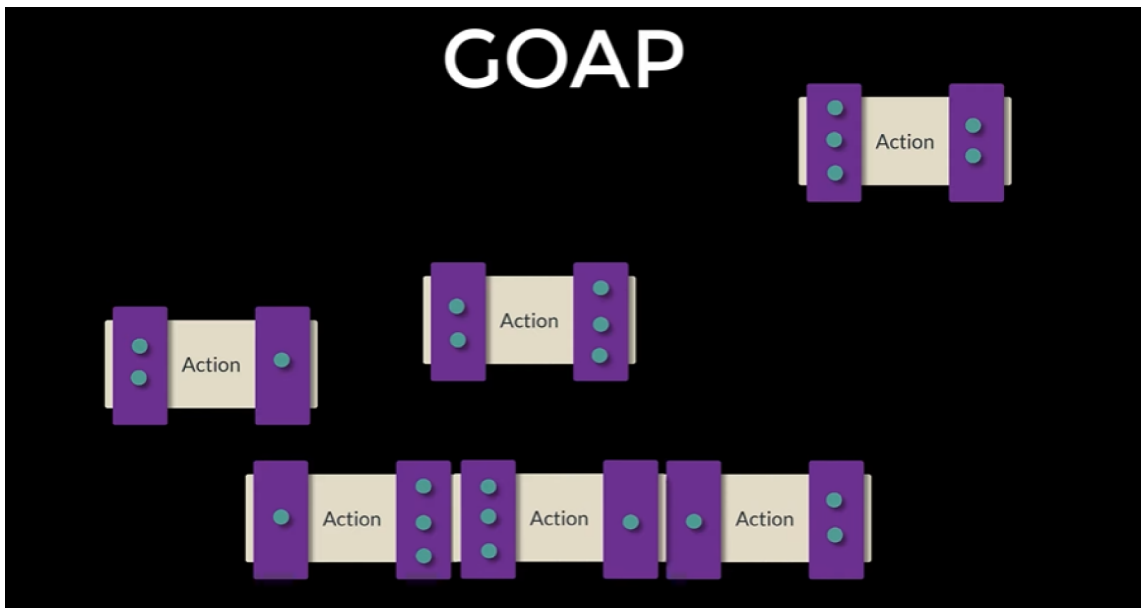


Figura 13: Representación de acciones en GOAP
Fuente: Holistic3D

normalmente se elige alguno por aleatoriedad.

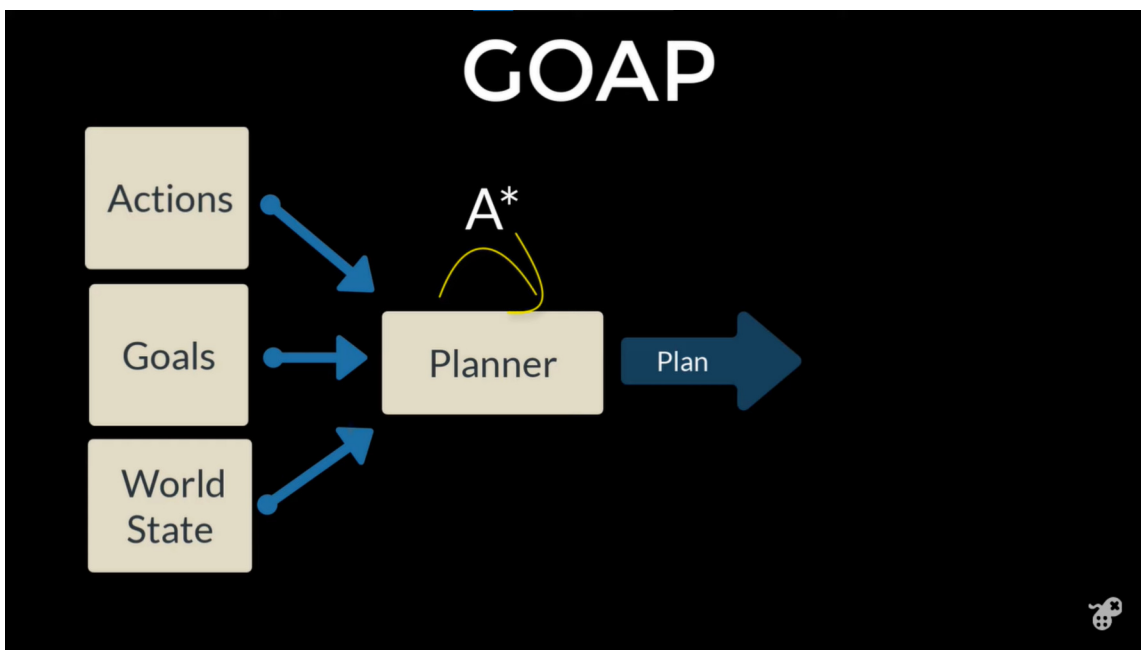


Figura 14: Arquitectura base de GOAP
Fuente: Holistic3D

El plan siempre se valida sobre una copia del estado actual del juego. Si tiene éxito, el sistema de planeación se encarga de transformar el estado actual. Si no, se desecha el plan y comienza nuevamente el proceso.

Una de las principales ventajas, y a la vez desventaja, de esta arquitectura es el sistema de planeación. El algoritmo de búsqueda puede encontrar diferentes planes cada vez que se ejecute, entregando rejugabilidad a los niveles lo cual suele llamar la atención de los usuarios. Sin embargo, se pierde parte del control sobre el comportamiento del agente, lo cual podría ser perjudicial para el equipo de desarrollo en el caso de que el agente se comporte de manera “óptima” según la evaluación de costo de los planes, pero la secuencia de acciones no se vea acorde al estilo del juego.

2.2.5. Hierarchical Task Network

HTN es un sistema de planeación, que permite generar comportamientos similares a los que haría un humano, pues descompone tareas complejas en otras más simples hasta llegar a un nivel donde la computadora puede ejecutar sin problema la acción, similar a la manera en que una persona enseña a otra una actividad compleja, separándola en pasos fáciles y ordenados [Humphreys, 2013]. Con HTN modelas el mundo con acciones, combinaciones de tareas que crean comportamientos complejos. Poseen precondiciones y efectos, tal como GOAP, pero nos centramos en las precondiciones al inicio de la acción y no por cada tarea que la compone. Lo mismo se aplica a los efectos, solo se evalúan los cambios al estado del juego al final de la acción.

Existen tareas **primitivas**, tal como las de GOAP y tareas **compuestas**, las cuales hacen uso de primitivas y otras compuestas, con tal de solucionar un gran sub-conjunto de objetivos en la planeación. Ej: Asaltar al jugador : Correr a cobertura → Cubrirse → Lanzar granada → Disparar

Lo bueno de HTN es su similitud con los BT, pero a diferencia de estos, se construye en tiempo real, considerando el estado y metas actuales del agente, lo que genera comportamientos más llamativos, mientras mantiene el control que los diseñadores tienen sobre el comportamiento.

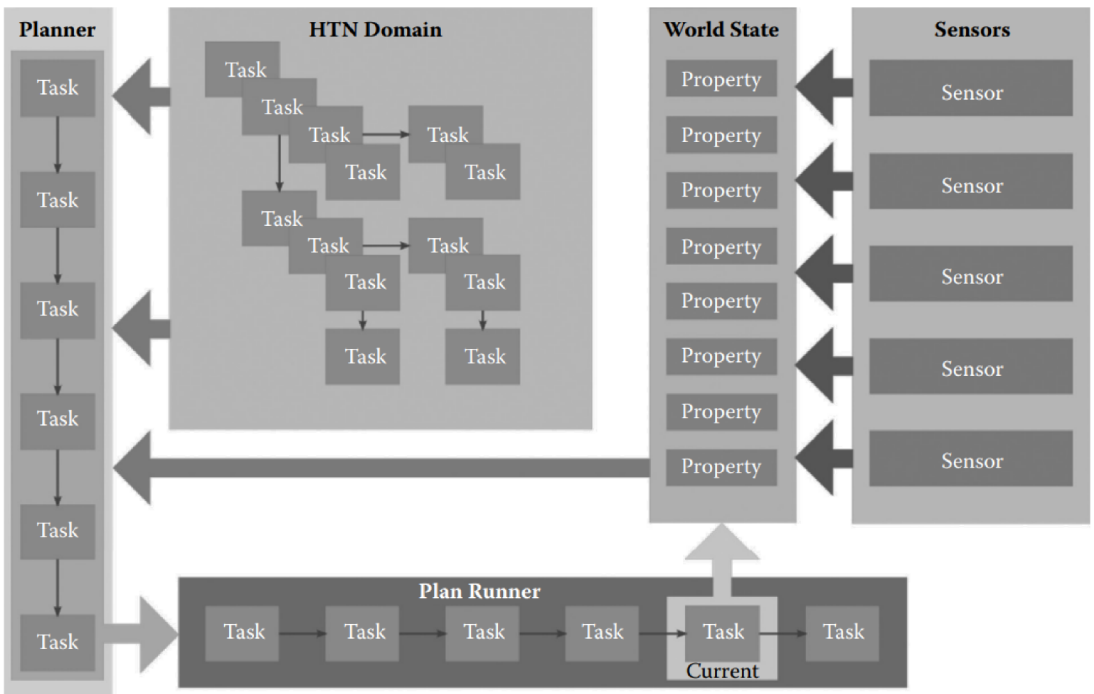


Figura 15: Diagrama general de HTN
Fuente: GameAI Pro

2.3. Discusión bibliográfica

El libro “Artificial Intelligence & Games”[Yannakakis and Togelius, 2018] provee una amplia cobertura de cómo se aplica Inteligencia Artificial en las diferentes áreas de los videojuegos:

game-play (la experiencia de jugar), generación de contenido y modelado de jugadores. Es un texto más académico, centrado en cubrir técnicas que se han popularizado entre los desarrolladores de la industria, aunque entrega varias referencias al estado de la práctica. En particular, es de interés el capítulo 2, dedicado a describir en más detalle las técnicas, utilizando como campo de pruebas el juego "Ms Pac-Man", dada su popularidad y complejidad, ya que es un juego no determinista a diferencia del "Pac-Man" original. Se distinguen 6 categorías principales de técnicas: **autoría de comportamiento**, relacionada a métodos estáticos sin ningún tipo de aprendizaje, como máquinas de estados finitos, árboles de comportamiento e IA basada en utilidad; **Árboles de búsqueda**, donde los métodos expanden árboles con posibles decisiones a tomar como GOAP, HTN, el algoritmo *Minimax* y *Monte Carlo Tree Search*; **Computación evolutiva** relacionada a algoritmos genéticos; y 3 tipos de métodos en el área de aprendizaje de máquinas: **supervisado**,**reforzado** y **no supervisado**.

2.3.1. NPC e inmersión

En la tesis de Henrik Warpefelt [Warpefelt, 2016], se crea una tipografía y un modelo con el fin de entender mejor lo que es un NPC, describir de manera íntegra sus diferentes características y visualizar, a través del modelo, cómo afectan en la credibilidad de su representación en el juego. Separa a los NPC en tipos genéricos propios de narrativas que aparecen en diferentes categorías de juegos (villano, princesa, héroe, etc), uniendo esta clasificación a otra creada en base a los roles dentro del juego (vendedor, guardias, narrador, relleno de espacio, entre otras). Esta fusión de tipologías, entrega una amplia y sólida guía para el desarrollo de los comportamientos que el componente tendrá.

Savaglia [Savaglia, 2021] estudia 2 patrones de diseño para crear IA en juegos, basados en la Prueba de Turing [Turing and Haugeland, 1950]: *top-down* y *bottom-up*. En el primero, la IA debe llegar a conclusiones usando las herramientas o "Fórmulas" que se le entregan, mientras que la segunda depende del *machine learning* para entregar respuestas en base a datos. Para probar cuál patrón resulta más convincente, Savaglia crea 2 IA (1 por patrón) que simulan ser una mascota virtual, y le pide a los usuarios que interactúan con ellas, cuál es su grado de satisfacción con ambas. Si bien la mayoría prefirió interactuar con la IA *top-dow*, la IA *bottom-up* recibió una mejor calificación promedio de realismo (5.3 BU vs 4.7 TD, en escala de 1 a 10).

2.3.2. Reusabilidad

En [Safadi et al., 2015], se inspiran en la capacidad de los humanos para encontrar analogías y proponen el diseño de un framework para diseñar IA conceptual, es decir, que sea capaz de analizar su entorno, traspasar los estados del juego a conceptos siguiendo ciertas políticas y entender lo que debería hacer. Tal framework permite que la IA diseñada se adapte a diversos escenarios, en diferentes juegos que no necesariamente pertenecen al mismo género.

2.3.3. Comportamientos

Uno de los comportamientos más usados en juegos de estrategia y sigilo, es **patrullar**. Enezi et. al [Al Enezi and Verbrugge, 2020] argumentan que los caminos generados automáticamente, ya sea de manera aleatoria o siguiendo algún objetivo, obtienen poca cobertura de los mapas y lucen poco naturales. Por ende, proponen un enfoque basado en modelar la "obsolescencia" de la posición actual de manera similar a los mapas de ocupación, aplicado en grillas como en mallas de visibilidad dinámica (en estas últimas, el NPC no tiene conocimiento del mapa en que se encuentra y lo explora "de a poco"). El desarrollo es en Unity y demuestra la efectividad del método en las mallas dinámicas.
Zhou et. al [Zhou et al., 2006] toca el campo de las emociones en NPC. Destaca que estas mejoran la calidad e inteligencia aparente de los agentes. En el artículo, se propone la creación de un modelo simple para transicionar entre 6 emociones básicas: sorpresa, tristeza, disgusto, susto, alegría y enojo. Para cada escena propuesta en el desarrollo, una emoción se destaca del resto, resultante de una serie de ecuaciones que toman en cuenta al NPC, la escena y el jugador. El trabajo resulta de interés dados sus resultados positivos, y el hecho de que [Warpefelt, 2016, Yannakakis and Togelius, 2018] destacan el campo de las emociones como uno de las fronteras en el desarrollo de NPC más reales.

2.3.4. Costos de desarrollo

[Bulitko et al., 2018] destaca que los costos de desarrollar IA en grandes juegos, como los multijugadores masivos online, son cada vez más altos, ya que se espera una rica variedad de interacciones, pero desarrollar a mano cada una no suele ser eficiente, por términos de tiempos de entregas y porque a veces tales interacciones se vuelven repetidas, dado que un mismo desarrollador se debe encargar de muchas de ellas. Por lo tanto, se busca resolver tales problemáticas usando técnicas de generación procedural, orientadas a crear NPC de manera más económica, y usando algoritmos evolutivos de manera *offline* en proxies manejados por IA, para sacar provecho a la adaptación de la IA frente al jugador, sin que a este último le tome demasiado tiempo darse cuenta de los avances.

2.4. Herramientas ya existentes en el mercado

A continuación se muestra una tabla con los *assets* en la tienda de Unity que sirven para la creación de comportamientos de IA para agentes. Para hallarlos, se buscó *"artificial intelligence"*, con el filtro de solo herramientas ("*Tools*")[10]. De estos resultados, se fueron revisando aquellos que más se alineaban a los objetivos de este trabajo. Se pueden sacar algunos datos interesantes de esta recopilación, como que las arquitecturas que más se repiten son

[10] https://assetstore.unity.com/?category=tools&q=artificial%20intelligence&orderBy=1

Behavior trees y FSM, mientras que no se encontraron HTN. Respecto a sistemas de utilidad, algunas soluciones mencionan que utilizan este sistema, pero no se puede acceder a su implementación sin antes comprarlas.

Nombre	**Descripción**	**Precio ($USD)**	**Última versión**
Behavior Designer	Probablemente la herramienta más completa. Incluye elementos de Teoría de Utilidad para calcular el puntaje de acciones y scripting visual con nodos.	$90	May 5, 2023
GOAP	Los comentarios apuntan a que es un framework poco pulido que no vale el precio, solo serviría para empezar un proyecto. Destacan que **no tiene UI**.	$39.95	Nov 2, 2016
Utility AI Framework	Pareciera implementar una arquitectura de utilidad muy similar al IAUS. Incluye herramientas de *debugging*, junto con videos extensos para demostrar el funcionamiento de la herramienta.	$35	Feb 9, 2023
Advanced Enemy AI	Apunta a crear prototipos rápidamente con comportamientos predefinidos y sensores comunes, junto con sistemas de salud/daño del agente.	$10	Jan 30, 2020
Blazed AI Engine	Motor de IA para cualquier género de juego, viene con escenas demo. Es compatible con visual scripting, destacando que no usan un framework particular que restrinja la creatividad a la hora de usarlo. Posee buenas reseñas respecto al feedback que el desarrollador entrega.	$29.99	May 8, 2023
AI Behavior	Tiene comportamientos útiles como Idle, Follow, Search, etc. Es básico y personalizable. Usa FSM como arquitectura base, aunque parece estar abandonado.	Gratis	Mar 28, 2019
Candice AI for Games	Bastante recomendado por sus usuarios. Usa Behavior Trees como arquitectura base y tiene bastantes configuraciones que se pueden añadir, además de contar con comportamientos ya establecidos. Tiene desarrollada una interfaz de usuario en el Editor.	Gratis	Jan 27, 2023

Instinct AI	Similar a Behavior Designer, tiene interfaz de usuario dentro de Unity. Usa curvas de utilidad. Destacan que no usan blackboards como medio de comunicación entre sistemas. Tiene acciones personalizables predefinidas.	$75	Dec 26, 2017
Easy NPCs AI	Incluye varias acciones predefinidas para NPCs, similares a los de GTA V y San Andreas, como pelear y hablar. Viene con su código fuente para modificarlo como se estime conveniente.	Gratis	Apr 21, 2023
DecisionFlex	Una herramienta que utiliza teoría de utilidad para calcular puntajes y acciones. Usa la misma jerarquía de Unity para agrupar acciones y consideraciones asociadas. Posee buenos comentarios, indican que es un asset útil para comenzar con la materia y que funciona a día de hoy incluso sin recibir actualizaciones.	$35	Oct 10, 2017
S GOAP	Posee buena cantidad de likes y reseñas. Cuenta con un debugger para ver cómo está haciendo el planning y las acciones. Se indica que no funciona *out of the box*, pues hay que escribir código para las acciones, pero se dan ejemplos para ayudar al desarrollo.	$49.99	Jun 16, 2022

Tabla 1: Herramientas para creación de comportamientos en Unity.

Fuente: Elaboración propia

CAPÍTULO 3
PROPUESTA DE SOLUCIÓN

3.1. Sobre la metodología

De acuerdo a lo expuesto en los capítulos anteriores y tomando en cuenta las reuniones donde se discutieron los enfoques más interesantes para resolver el problema, se resolvió utilizar la arquitectura de **I.A.U.S.** como modelo principal para implementar la lógica detrás de la toma de decisiones de los agentes. Los principales atributos que sostienen esta decisión se relacionan con la **flexibilidad** que posee, pues se adapta al contexto en tiempo de ejecución a través de la evaluación de las consideraciones; es **extensible**, pues podemos añadir tantas acciones y consideraciones como estimemos convenientes; provee una interfaz relativamente sencilla a los diseñadores para cambiar el comportamiento de los agentes, a través de las curvas de respuesta del editor; es **reusable**, ya que la arquitectura en si se encarga de pensar qué hacer, no cómo hacerlo, por lo que se puede exportar el código principal como paquete a diferentes proyectos, independiente de su género.
Además, se utiliza el proceso **SPA** como referencia para que los agentes muestren capacidades de interacción con el ambiente y se pueda validar el funcionamiento de la arquitectura en un caso de prueba dentro del videojuego en desarrollo ***Mix, the forgotten***.
A continuación se presentan diagramas con los componentes principales de la arquitectura a desarrollar, junto con sus definiciones para entender las responsabilidades asociadas a cada uno. También se detalla el funcionamiento y comunicación entre las diferentes partes, para comprender el flujo de la solución.

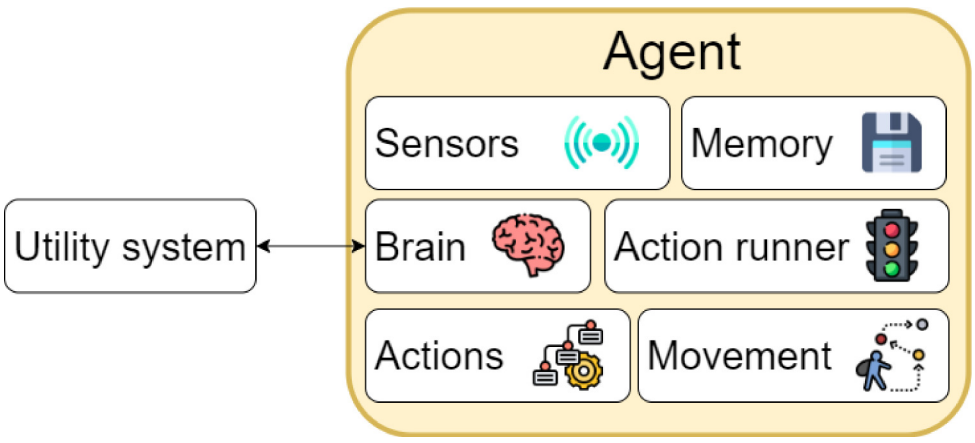


Figura 16: Partes principales de un agente
Fuente: Elaboración propia

3.2. Descripción general del sistema

Como se observa en la figura 16, un agente se compone de 6 partes principales que se comunican entre si para completar el ciclo SPA. Además, existe una entidad externa (Utility system) con la cual cualquier agente se puede comunicar para obtener el comportamiento más adecuado para su contexto.

A través de estímulos externos e internos, representados por eventos en C#, el agente es capaz de recibir, actualizar y evaluar constantemente la información que posee en su memoria. El diagrama de la figura 17 muestra una visión general de las relaciones existentes entre los componentes y el flujo de información entre ellos. Se observa que los eventos principales, representados por óvalos amarillos, son captados por el **Agent Brain** y desencadenan el comportamiento cíclico que le da vida al agente. Los recuadros marcados en rojo (Utility System y Action Runner), esconden parte de la complejidad del sistema y son detallados más adelante en la sección.

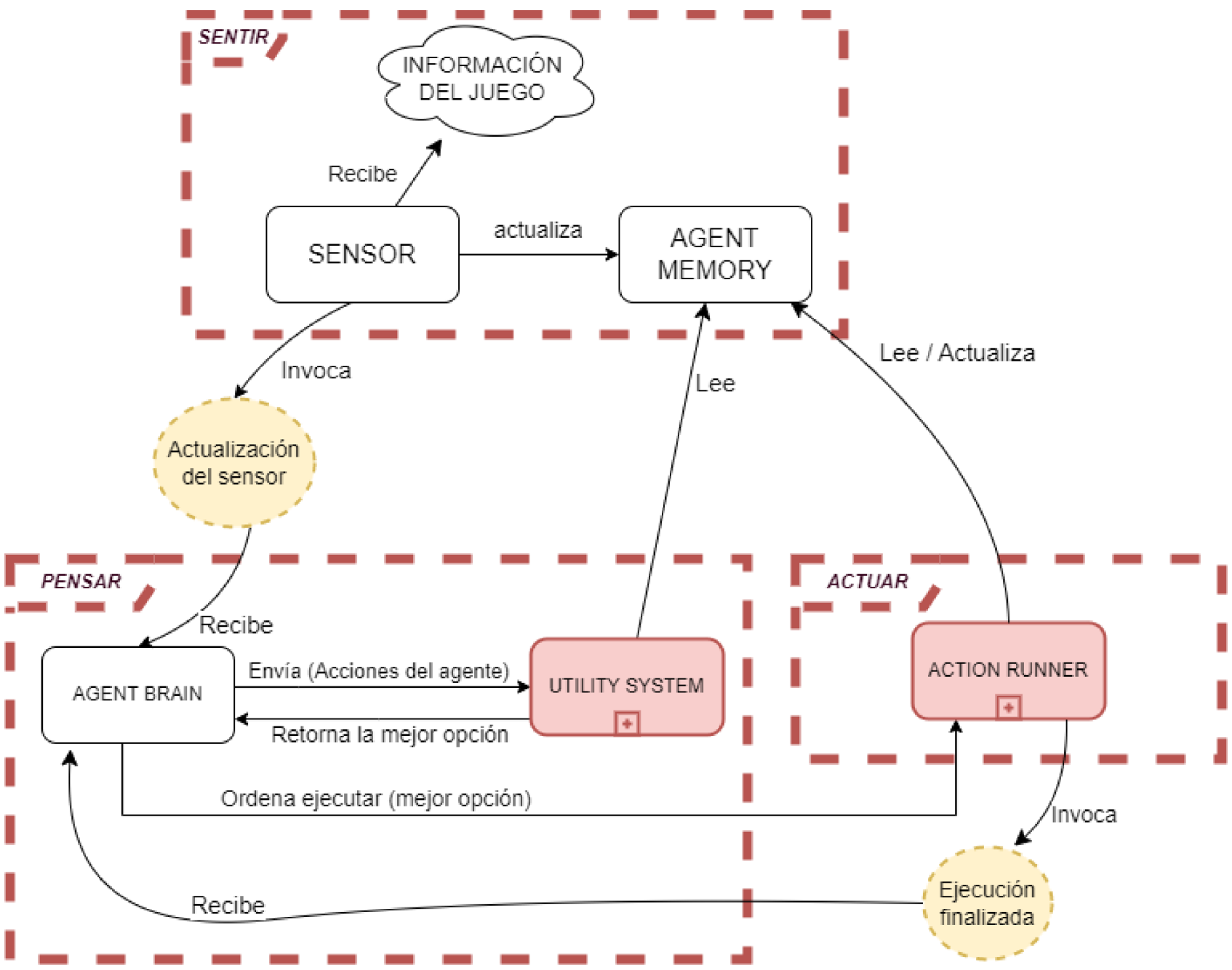


Figura 17: Diagrama general del sistema
Fuente: Elaboración propia

3.3. Sensors

Un agente puede tener uno o más sensores. Estos se encargan de captar datos relevantes dentro del juego, como enemigos, zonas de terreno especiales, sonidos, objetos interactuables, etc. y posteriormente dar aviso de este evento de interés. Adicionalmente, el sensor puede leer o escribir en la memoria local (`AgentMemory`) del agente según lo requiera.

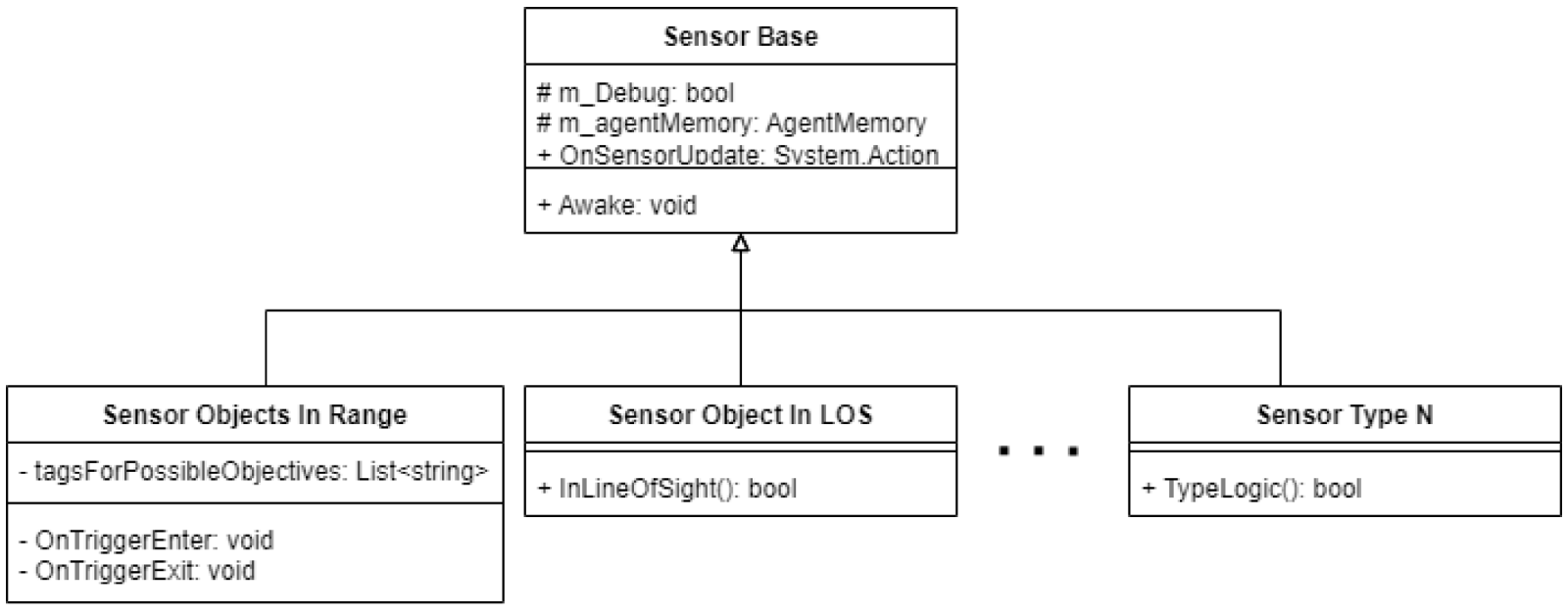


Figura 18: Herencia de sensores
Fuente: Elaboración propia

Para implementarlo, se crea una clase base abstracta `SensorBase` con los campos comunes para todos los sensores, mientras que cada sensor específico, hereda de esta clase e implementa su propia lógica en la que debe invocar el evento *OnSensorUpdate*, pues a través de este mensaje se da aviso al componente `AgentBrain`, y a cualquier otra entidad que lo necesite, que el estado del juego ha cambiado y el agente ahora debería *pensar* en qué hacer al respecto.

3.4. Memory

Es la clase donde se guardan los datos que describen el estado actual del agente, como objetivos, items que ha tomado, vida, *stamina*, etc. Es leída y/o actualizada por el sistema de utilidad, los sensores y las acciones asociadas al agente, actuando como una *blackboard* local.

Cuenta con un campo llamado **Configuration**, que permite inicializar las estadísticas con valores predefinidos en una instancia de la clase `AgentConfiguration`. Esto facilita la tarea de configurar diferentes tipos de agentes, pues cada instancia del mismo tipo, hará referencia al mismo archivo de configuración, por lo que si se quiere cambiar algún valor, basta hacerlo una vez en ese *asset*.

3.5. Agent Brain

Esta clase sirve como unidad de control central para dirigir la IA del agente, dándole la capacidad de *pensar*, evaluando la utilidad de las diferentes opciones que puede ejecutar. En este contexto, una opción es una instancia de la clase `Option`, que agrupa a una acción, su objetivo (en caso de ser necesario) y el puntaje obtenido en la actual evaluación. El **Agent Brain** busca la memoria, acciones, sensores, **Action Runner** y movimiento asociados al agente, guardando referencias a cada uno de estos componentes. Se puede asignar una de estas acciones al campo **Default action**, para que sea ejecutada si todo lo demás falla. Aquí normalmente iría alguna acción simple como quedarse quieto (*idle*), emitir algún sonido y/o reproducir una animación.

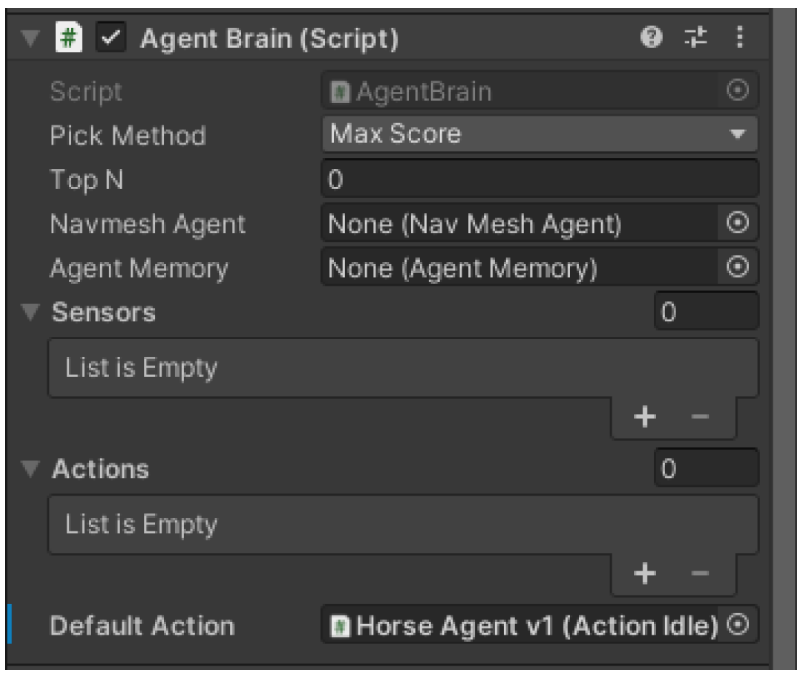


Figura 19: Agent Brain en Unity

Por cada acción, le entrega una referencia a la memoria y movimiento del agente, pues en la mayoría de los casos, los necesitan para ejecutarse y guardar sus efectos. Esto también libera al diseñador de tener que asignar referencias manualmente en el Editor, lo cual consume tiempo cuando el número de acciones es alto. También, se suscribe al evento *OnFinishedExecution* del **Action Runner**.

Luego, por cada sensor dentro de la jerarquía del agente, se suscribe al evento *OnSensorUpdate*. De esta manera, el cerebro sabe cuando ocurre un evento importante, comenzando así un ciclo de pensamiento para evaluar qué debería hacer dada la nueva configuración de su ambiente. Para ello, se comunica 2 veces con la clase **Utility system**: Primero solo le entrega la lista de acciones del agente al método `ScorePossibleOptions`, recibiendo otra lista con todas las opciones válidas. Luego, envía esa lista de opciones, el método de selección y el campo `top N` como argumentos al método `PickFromScoredOptions`, recibiendo la mejor **opción** a ejecutar, siendo esta la que se envía a la clase **Action Runner**, para ser ejecutada por el agente. Esta doble comunicación puede parecer innecesaria, pues el `UtilitySystem` podría entregar directamente la mejor opción al **Agent Brain**, pero se implementa así para lograr *debuggear* al agente, pues luego de recibir la primera respuesta, el agente envía un evento con la lista de opciones válidas, al cual se suscribe una clase complementaria llamada `AgentDebugger` que permite ver en la interfaz del juego cómo varían las diferentes opciones (ver figura 25 como ejemplo). Así es más sencillo elegir a qué agente se desea inspeccionar en tiempo de ejecución, a diferencia de lo que sería tratar de obtener esta misma información desde la clase `Utility System`, pues al ser estática, no posee memoria del agente ni sus acciones, y si bien podría invocar el mismo evento cuando termina de crear una lista de

opciones, como es accedida por todos los agentes presentes en la escena, sería difícil saber a quién corresponden tales opciones.

3.6. Utility system

El sistema de utilidad se encarga de asignar puntajes a las acciones que el agente puede ejecutar, basándose en el contexto actual, y elegir alguna de ellas según el método que el **Agent Brain** le indique. Como es una clase estática, no posee memoria del estado del juego, por lo que el contexto en realidad se obtiene de las consideraciones que poseen las diferentes acciones que el agente le envía. Por cada acción recibida, obtiene una lista de 0 (null), 1 o N opciones, dependiendo si esta acción es válida (según sus consideraciones) y si se puede aplicar o no sobre distintos objetivos. Como se observa en el código, solo se agregan opciones que no sean nulas a la lista retornada. Una opción es nula cuando el puntaje de la acción es 0.

```
public static List<Option> ScorePossibleOptions(List<ActionBaseClass> actions)
{
    List<Option> scoredOptions = new();
    foreach (ActionBaseClass action in actions)
    {
        var options = action.ScoreThisAction();
        if(options != null) scoredOptions.AddRange(options);
    }
    return scoredOptions;
}
```

Luego, esta lista es filtrada y/o reordenada según el método de selección, siguiendo el proceso detallado al final de la sección del **Agent Brain**. Estos métodos, descritos a continuación, cambian la personalidad expresada por el agente dentro de la ejecución, sin embargo se asegura que la opción a ejecutar siempre sea lógica según las consideraciones evaluadas.

- `Max Score`: se elige la opción con el puntaje más alto.
- `All random`: se elige cualquier opción de manera aleatoria.
- `Weighted all random`: una selección aleatoria con pesos, según el puntaje de cada opción. Aquí, es más probable que se elija la opción con mayor puntaje, pero no será así en todos los casos.
- `Top N`: todas las opciones se ordenan de manera descendente según su puntaje. Luego solo las primeras N más altas son consideradas para hacer una selección aleatoria.
- `Weighted Top N`: una mezcla de los 2 métodos anteriores. Se eligen las primeras N opciones con mayor puntaje y entre ellas se hace una selección aleatoria con pesos.

3.7. Action Runner

Esta clase se encarga de comenzar e interrumpir la ejecución de opciones. Estos aspectos son clave para evaluar el comportamiento exhibido por el agente, pues queremos que sea capaz de reaccionar a los cambios en su entorno de manera controlada, para evitar casos donde alterne constantemente entre comportamientos válidos de manera muy rápida, lo cual le impediría terminar cualquiera de sus acciones.

Cuando recibe una opción desde el **Agent Brain**, primero verifica si ya hay una opción en memoria que esté ejecutándose. Si no es así, simplemente ejecuta la nueva opción y se suscribe al evento **OnFinishedAction** de la acción encapsulada en la opción.

```
private void BeginNewExecution(Option option)
{
    m_currentOption = option;
    option.Action.OnFinishedAction += FinishExecution;
    option.Action.StartExecution(option.Target);
    IsRunning = true;
}
```

Este evento, como su nombre indica, permite saber cuándo la acción ha llegado a su fin. Esto es útil, pues permite reiniciar el ciclo de pensar-actuar a través de una simple cadena de mensajes: cuando la acción finaliza exitosamente, envía un aviso (*OnFinishedAction*), luego el método de **Action Runner** suscrito a este evento también dispara otro evento (*OnFinishedExecution*), el cual es detectado por el **Agent Brain**, quien vuelve a evaluar la situación actual del agente.

```
private void FinishExecution(){
    if (m_currentOption != null)
    {
        m_currentOption.Action.OnFinishedAction -= FinishExecution;
        m_currentOption = null;
    }
    IsRunning = false;
    OnFinishedExecution?.Invoke();
}
```

En el caso de que ya hubiera una opción ejecutándose, se verifica que esta no se encuentre bloqueada. Si lo está, se ignora la nueva opción. Esto para evitar que ciertas acciones sean interrumpidas en momentos críticos, que dejarían al agente en un estado corrupto. En caso de estar desbloqueada, significa que podemos interrumpir su ejecución de manera segura, llamando al método `InterruptExecution(option)`, y luego ejecutando la nueva opción, siguiendo los pasos descritos anteriormente.

3.8. Actions

Estas son las clases que definen cómo actuará el agente en algún momento dado del juego. Además, poseen una lista de consideraciones que son evaluadas cuando el agente necesita decidir qué hacer. Las acciones son *Monobehaviour* y siguen una estructura de herencia tal como los sensores, es decir, existe una clase base que define los miembros comunes y cada heredera sobrescribe métodos y/o añade miembros específicos del comportamiento. Las acciones se implementan principalmente usando *Coroutines*, pues necesitan ejecutarse en periodos extensos de tiempo, no solo en un *frame*.

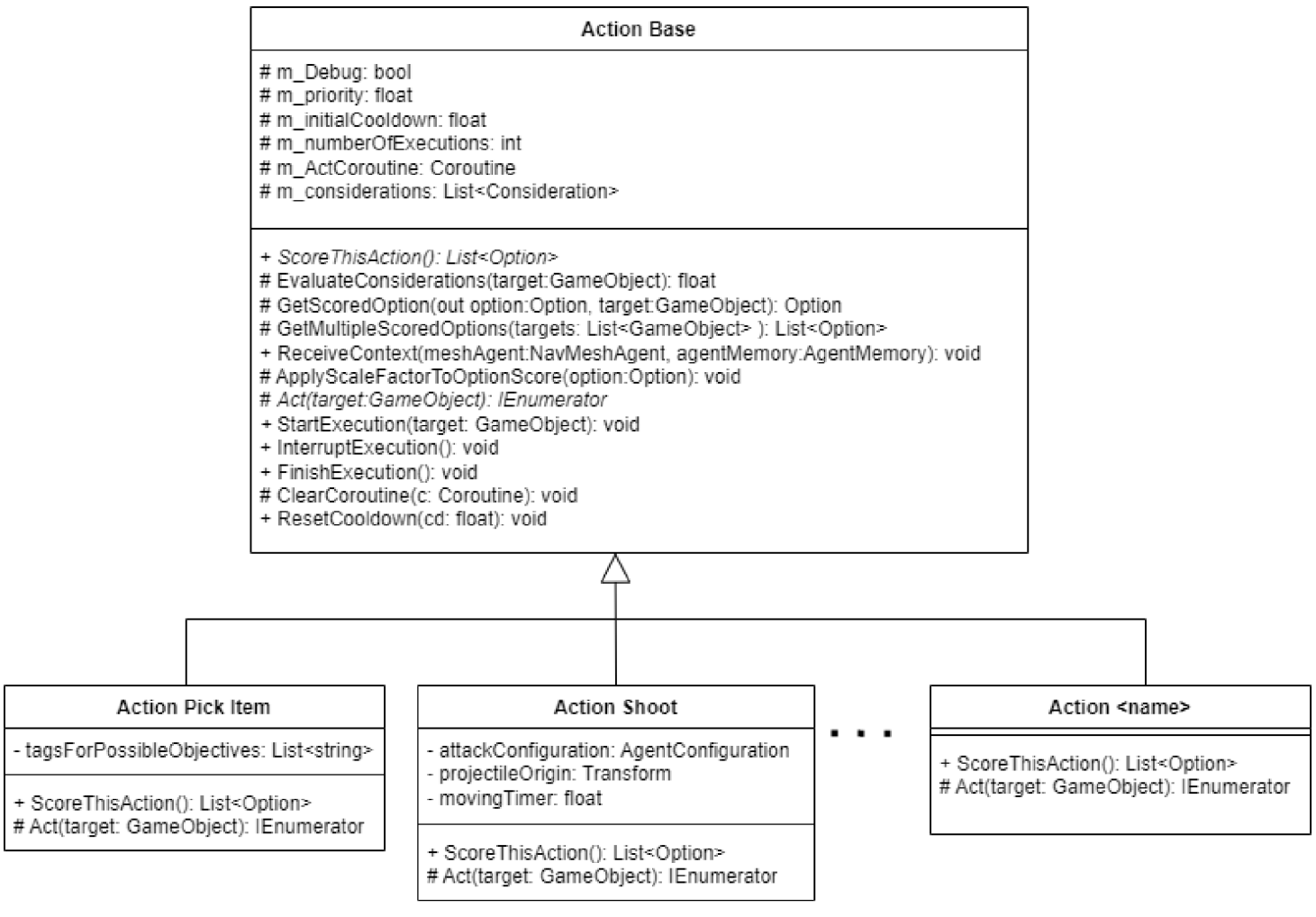


Figura 20: Diagrama de herencia de acciones
Fuente: Elaboración propia

Una acción debe definir en su implementación cómo comenzar, interrumpir y finalizar su ejecución. Estos 3 aspectos son necesarios para que el **Action Runner** sea capaz se manejarla.

Una acción comienza llamando a la *coroutine* principal `Act`, que debe implementar la lógica específica de esa acción donde pueden llamarse una o más *coroutines* con comportamientos complementarios, como moverse a una cierta posición, verificar distancias respecto a un objetivo, emitir sonidos, comenzar animaciones, etc. Durante su ejecución, es posible que cierta lógica deba completarse sin interrupciones para evitar que el agente y/o algún posible objeto del juego quede en un estado corrupto, por lo que las acciones tienen una propiedad pública que permite bloquearlas en esas situaciones específicas. Es importante señalar que esta propiedad debe manejarse con cuidado, pues tal como se indicó en la sección de **Action Runner**, si no se desbloquea no se pueden ejecutar nuevas acciones.

En línea con lo anterior, al interrumpir una acción se deben manejar los procesos que hayan quedado a medio terminar, para que el agente no rompa la inmersión del juego y su estado sea adecuado para ejecutar una nueva opción. Esto variará dependiendo de la implementación, pero lo común en este caso es detener *coroutines* y limpiar variables propias de la acción o de la memoria del agente.

Para finalizar una acción, se debe seguir un proceso similar a la interrupción, validando que el estado del agente sea adecuado. Sin embargo, aquí se debe dar aviso, mediante el evento *OnFinishedAction*, para reiniciar el ciclo de pensamiento, tal como se describió en la sección de **Action Runner**.

Por último, para participar en la ronda de selección, una acción debe definir cómo se calculará el puntaje de las diferentes opciones posibles, por lo que debe tener al menos una consideración en su lista. Cabe recordar que cuando se elige algún comportamiento, este se escoge de la lista de **opciones válidas** que el **Agent Brain** recibe del **Utility System**. La clase `Action Base` define dos métodos auxiliares para ello, uno para cuando existen múltiples objetivos y otro para cuando solo hay uno (ya sea el mismo agente u otro objeto predefinido) o ningún objetivo. Sin embargo, cada acción puede implementar su propio método de evaluación, solo debe asegurarse de devolver una lista de opciones válidas o `null`. Si una acción retorna `null`, significa que de acuerdo a sus consideraciones, ninguna opción relacionada es válida, por lo que el agente la descarta.

3.9. Considerations

Tal como se describió en la sección 2.2.3 las consideraciones, o ejes, son piezas de información, del estado actual del juego, relevantes para evaluar la utilidad de ejecutar una acción en este contexto. En Unity, la clase `Consideration` hereda de `ScriptableObject`. Así, cada consideración creada en el proyecto es un *asset* que puede ser referenciado por una o más acciones de diferentes agentes. Esto provee una manera sencilla de reutilizarlas y evita duplicaciones innecesarias en memoria. Además, tal como se mencionó en el marco conceptual, al ser `ScriptableObject`, se puede editar su configuración en *runtime* y tales cambios persistirán en el disco luego de la ejecución, lo que facilita crear comportamientos que se adapten a la experiencia solicitada por los/las diseñadoras, mientras se va probando el juego.

En esta implementación, una consideración posee 3 piezas clave:

1. Un contenedor de métodos implementados, que extraen información del mundo.
2. El método escogido, del contenedor anteriormente mencionado.
3. La curva de respuesta.

El rango para truncar y normalizar el valor obtenido por el método antes de entregarlo como entrada a la curva de respuesta es opcional.

El proceso que sigue una consideración para retornar un valor hacia la acción que la referencia, es el siguiente: primero se invoca su método asociado, que busca y devuelve el valor actual v del dato de interés, por ejemplo, la salud del agente. Luego, en caso de ser necesario, v se trunca y normaliza según los parámetros configurados en el editor. Finalmente, v se entrega como *input* a la curva de respuesta, que devuelve un valor normalizado hacia la acción. Cabe señalar que como los valores devueltos por cada consideración se multiplican para crear el puntaje final de una nueva opción, cualquiera que retorne 0 hará que la opción se anule automáticamente. Por ejemplo, un agente que posee una personalidad muy estricta con la higiene, pero que también ama los flanes de vainilla, tiene hambre, por lo que va a su cocina a **comer** algo. Es probable que pueda elegir entre varias opciones y para decidirse compara aspectos como qué tanto le gusta el alimento, qué tan satisfecho estará, cuánto demorará en prepararlo, el estado del alimento (podrido o no), entre otros atributos. Digamos que encuentra carne, tomates y un flan de vainilla abierto hace 2 días. Internamente, el agente *piensa* que la carne y el tomate suenan como opciones adecuadas, con un puntaje de 4 y 5 en una escala de 1 a 10 (escala puesta de manera arbitraria), pero el flan, que podría tener un puntaje de 10, dados los gustos del agentes, termina siendo ignorado, porque tiene hongos y se encuentra en mal estado, o sea, que esa consideración anuló al resto (o sea, retornó 0) y con justa razón pues coincide con la personalidad esperada del agente (y de cualquiera con sentido común). Esta es la idea que justifica cómo se evalúan e interpretan las consideraciones.

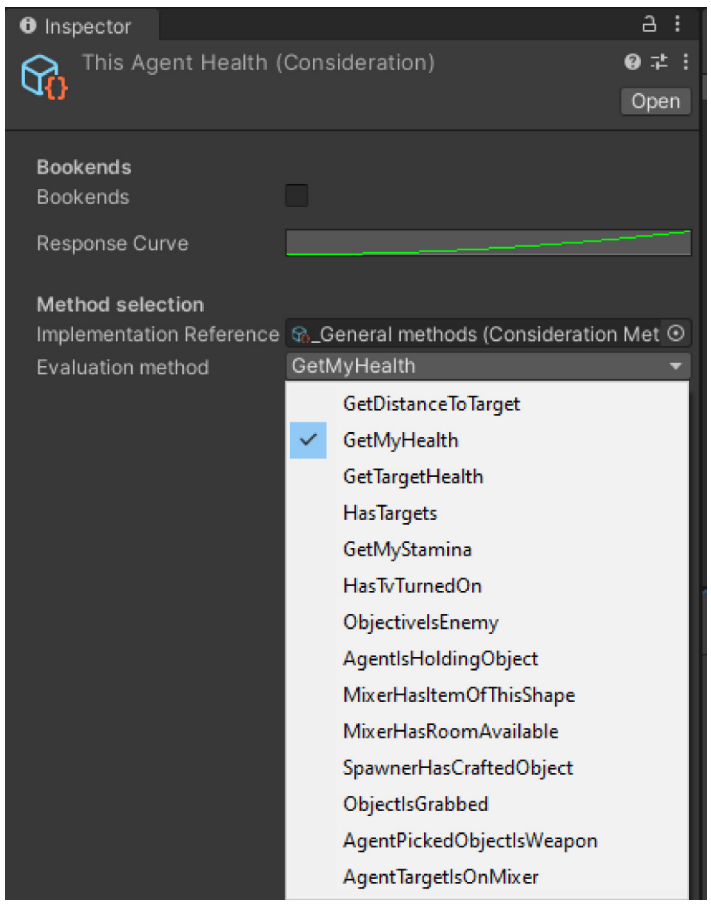


Figura 21: Instancia de una consideración en Unity
Fuente: elaboración propia

El método asociado a una consideración puede configurarse en el editor, facilitando en gran medida la creación de estas, pues a diferencia de los sensores o las acciones, no es necesario crear una nueva clase por cada consideración que requiere obtener una pieza distinta de información. En cambio, basta crear una instancia de `Consideration`, a través del menú de creación de *assets*, indicar un nombre adecuado que represente claramente lo que se está *considerando*, por ejemplo “This agent health”, y modificar sus parámetros, asignando la referencia a la instancia de la clase donde se implementa el método de interés y eligiéndolo en la lista correspondiente, como se ve en la figura 21.

Esto se logra utilizando reflexión, una característica del lenguaje C#[11] y un *Custom Editor* programado específicamente para mostrar instancias de consideraciones en el inspector. Para que un método sea listado debe tener al menos 3 *flags*: `BindingFlags.Public`, `BindingFlags.Static`, `BindingFlags.DeclaredOnly`. Estas *flags* le indican al sistema de reflexión que debería buscar solo métodos públicos y estáticos que sean declarados a nivel del tipo que se asignó en el campo **Implementation Reference** de la consideración.

Para mantener un código reutilizable, todos los métodos elegibles por una consideración deben poseer la misma firma, de tal manera que reciban los mismos tipos de argumentos, de donde obtendrán el contexto, y retornar un valor numérico de punto flotante, que representa a la pieza de información buscada.

3.10. Movement

Se decidió utilizar la solución de navegación incorporada en Unity, a través de las *Nav Mesh* y el componente *Nav Mesh Agent*. Una razón de esta decisión es que ya se conocían las características ofrecidas por este sistema, además que es fácil de integrar usando su *API*. De esta manera, basta invocar una función para indicarle al agente donde debe dirigirse, mientras el sistema esconde mucha de la complejidad asociada a tal orden, como el cálculo de caminos óptimos, la evasión de obstáculos u otros agentes, cuando debería detenerse, etc. Adicionalmente, este paquete incluye componentes para actualizar las *navmesh* en tiempo real, lo cual es útil cuando queremos interactuar con el terreno donde transitan los agentes, ya sea cambiando el tipo de área donde se encuentran o destruyendo zonas, las cuales serán evitadas automáticamente al momento de calcular caminos.

[11] `https://learn.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection`

CAPÍTULO 4
VALIDACIÓN DE LA SOLUCIÓN

Se desarrollan 2 casos de pruebas para verificar la utilidad de la arquitectura propuesta, como parte del desarrollo del videojuego *"Mix, the Forgotten"*, propiedad de la empresa chilena **Abstract Digital**[12]. En cada uno se detallan los requisitos entregados por los diseñadores, que corresponden a las descripciones de alto nivel sobre cómo debería comportarse un agente, los sensores implementados, las acciones que cada agente posee, las consideraciones evaluadas (por acción) y los resultados obtenidos.

Mix, the Forgotten es un videojuego narrativo de acción y aventura 3D en tercera persona, que transporta a los jugadores a un entorno enigmático y lleno de vida, habitado por objetos perdidos. En el papel de Mix, el protagonista, te enfrentas a desafiantes acertijos mientras recorres la Dimensión perdida. La misión es hacerte amigo de quienes encuentras en tu camino, ayudándolos a superar las diferentes etapas del duelo, representadas por biomas, pues los objetos en realidad son personas atrapadas en sus propios limbos tras su muerte. Resolviendo problemas a través de la construcción de herramientas y dialogando con los personajes, Mix debe llegar a la etapa de la Aceptación, donde finalmente él también deberá superar la realidad de que no puede ayudar a todos como quisiera.

La construcción de objetos es una mecánica que implica combinar diferentes materiales (ingredientes) del ambiente en una mesa de *crafteo*, llamada *mixer*, que verifica si se ordenaron de una manera específica y tengan propiedades (forma, material, función) que coincidan con alguna receta conocida. De ser así, aparecerá un nuevo objeto sobre el *weapon spawner* (objeto asociado al *mixer* donde aparecen los elementos creados), que el jugador o los agentes pueden tomar. Al hacerlo, los ingredientes desaparecen de la mesa. Ciertos objetos pueden crearse con diferentes materias primas, siempre que estas cumplan las restricciones de la receta. Por ejemplo, un martillo se construye combinando 2 ingredientes: un objeto cilíndrico que actúa como el mango y otro rectangular que se convierte en la cabeza. Incluso, se puede crear un martillo recursivo, es decir, utilizar uno ya construido como ingrediente para elaborar otro. De esta manera, tanto una tubería, una linterna y hasta otro martillo, pueden servir como el mango para crear uno nuevo, mientras que objetos como una televisión o cajas de madera servirían para hacer la cabeza. Esta mecánica abre la posibilidad a los jugadores (y agentes) para experimentar variaciones de todo tipo.

4.1. Caso 1: Shooter

Este agente debe actuar como un enemigo, disparando a objetos relevantes dentro de su rango de detección, dándole prioridad a los más cercanos. Si no posee un objetivo, entonces debería deambular por el terreno. Este comportamiento sirve como primera prueba pues,

[12] `https://www.abstractdw.com/`

si bien puede parecer simple en un principio, requiere evaluar distintos objetivos a la vez, manejar varios sub-comportamientos de manera paralela e interrupciones.

Implementación

El agente posee un sensor **Sensor Objects In Range** que utiliza colisiones de tipo *OnTrigger* para detectar cuando un objeto entra o sale de su rango, usando un componente *Sphere Collider*. El sensor cuenta con una lista donde se indican los *tags* que tienen los objetos de interés. Esta lista permite filtrar las colisiones con otros objetos. En este caso, se buscan aquellos con *tags* “Player” y “Enemy”.

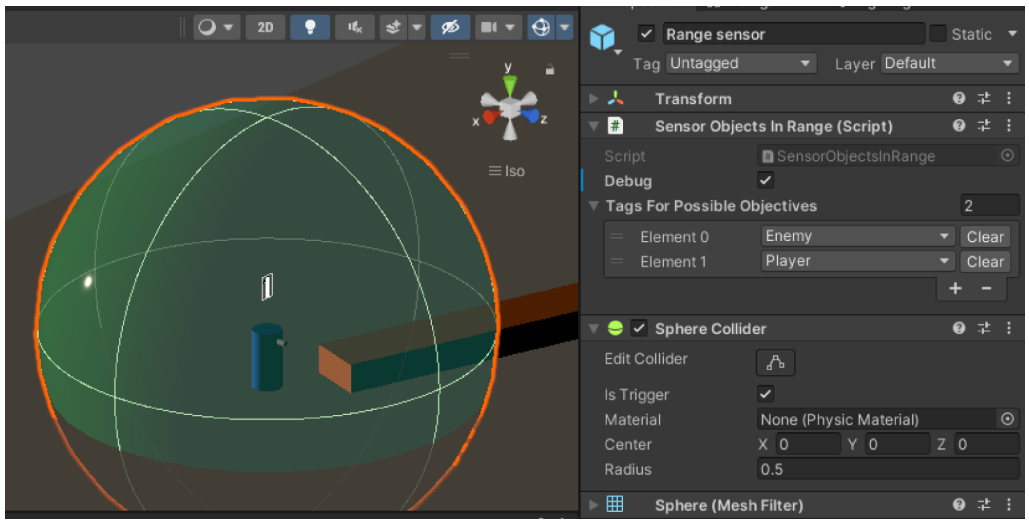


Figura 22: Sensor de *Shooter* implementado en Unity

Una vez verificadas las condiciones, el sensor actualiza la memoria del agente, añadiendo (o removiendo, según sea el caso) el objeto a su lista de objetivos, para luego disparar el evento ***OnSensorUpdate*** como se ve en el código.

```
private void OnTriggerEnter(Collider other)
{
 if (tagsForPossibleObjectives.Contains(other.tag))
 {
  if (!HelperFunctions.CheckTarget(m_agentMemory.Objectives,other.gameObject))
  {
   if(m_debug) Debug.Log(other.name + " found");
   HelperFunctions.AddTargetInList(m_agentMemory.Objectives, other.gameObject);
   OnSensorUpdate?.Invoke();
  }
 }
}
```

Para disparar, se creó la acción **Action Shoot**. Cuando es ejecutada inicia 3 *coroutines*: una encargada de perseguir al objetivo, otra de apuntar hacia este y la última, de efectuar el disparo y dar término a la acción, invocando el evento respectivo. Esta acción posee una consideración: la distancia entre el agente y su objetivo (ver figura 23). De esta manera, por

cada objetivo en la memoria del agente, se evalúa la distancia que los separa con una curva de respuesta decreciente, similar a una cuadrática invertida, logrando que mientras mayor sea la distancia menor sea el puntaje que el objetivo recibe.

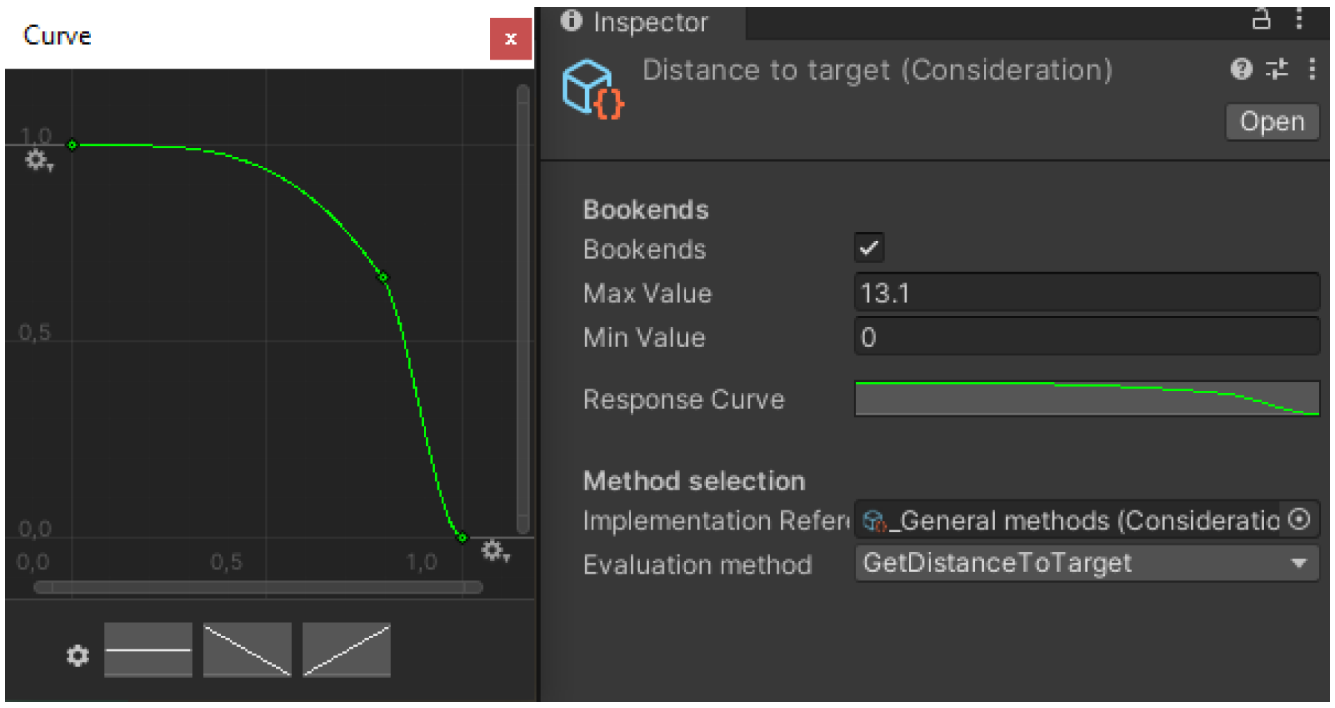


Figura 23: Consideración de distancia en el inspector
Fuente: Elaboración propia

La segunda acción implementada es **Action Move to Random Direction**, la cual permite al agente moverse aleatoriamente por el terreno. Es importante añadir pausas en esta clase de comportamientos, pues dan la ilusión de que se está explorando y no solo siguiendo una rutina predefinida. Para ello, cada cierto tiempo se elige una nueva posición dentro de la *navmesh* donde se encuentra el agente, se mueve hacia ella y al llegar al destino, toma una pausa pequeña.

En este caso, la única consideración que esta acción posee es la de verificar que no tenga objetivos en su memoria. Si fuera así, automáticamente se interpreta que esta acción deja de ser válida, pues debería estar persiguiéndolo en vez de deambular sin rumbo. El método asociado *Has Targets* devuelve 1 si el agente tiene objetivos y 0 en caso contrario, pero usando la curva de respuesta, se puede invertir este resultado. De esta forma, siempre que el agente tenga al menos un objetivo en su memoria, esta consideración puntuará 0, y por consiguiente la acción no será parte del proceso de selección.

El agente fue puesto a prueba en un campo con diferentes obstáculos estáticos, junto con otros agentes que se mueven aleatoriamente por el terreno, además del propio jugador. Se puede encontrar un video en el Anexo que muestra el funcionamiento. La figura 25, muestra la ventana de *debug* del agente, cuando se encuentra en una situación donde posee múltiples objetivos. Esta ventana refleja en tiempo real todas las **opciones** que el agente tiene disponible. Cabe mencionar que, si bien el agente tiene la acción `Move To Random Direction` asignada, esta no aparece listada pues tiene objetivos cerca, por lo que su única consideración tiene un valor de 0, y como se mencionó previamente, las acciones que no tienen opciones válidas, son anuladas y descartadas de la ronda de selección.

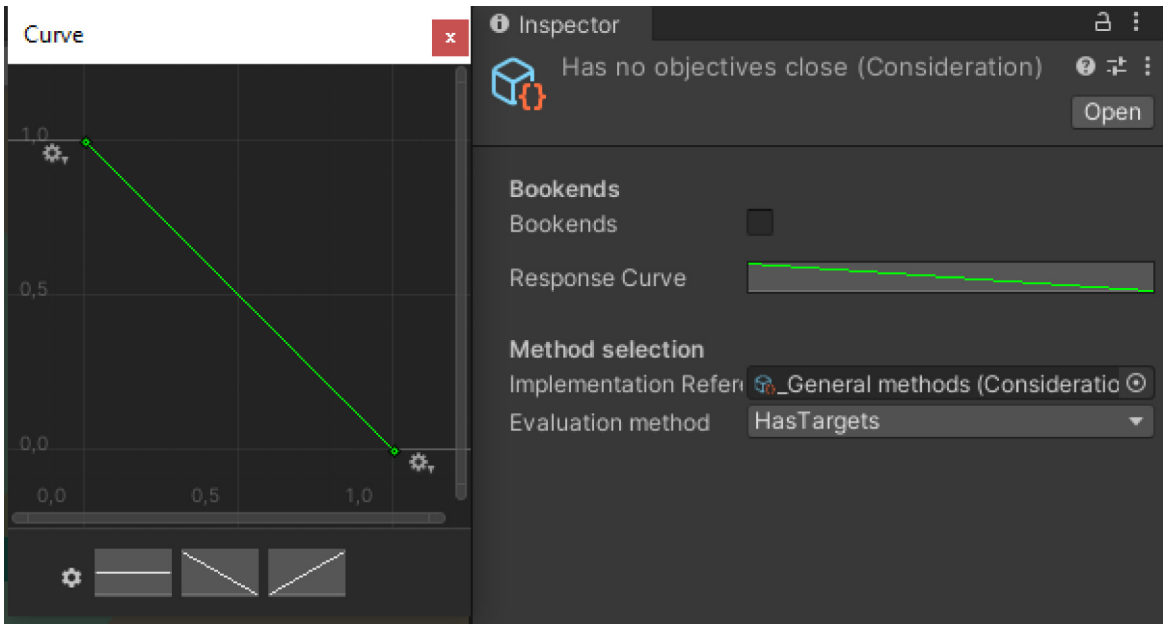


Figura 24: Consideración de objetivos en memoria
Fuente: Elaboración propia

Resultados

Se logró comprobar que la arquitectura es funcional pues el agente era capaz de razonar adecuadamente con la información que tenía, sin quedarse estancado pensando. Las cadenas de eventos que se programaron para comunicar los componentes principales del agente fueron de utilidad para que pudiera seguir ejecutando acciones cuando terminaba la anterior. Cuando se enfrentaba a múltiples objetivos, logró identificar aquel con mayor utilidad de acuerdo a las consideraciones de sus acciones y podía interrumpir su acción de disparo, con tal de cambiar su objetivo por el que tuviera mayor puntaje, mientras que cuando no tenía a quien seguir, automáticamente optaba por deambular en el terreno. Este comportamiento mostraba una ligera tendencia a caminar cerca de los bordes del mapa, pero esto tiene que ver con la manera en que se elige un punto aleatorio de la *navmesh* en la implementación de su acción. En resumen, este caso entregó buenos resultados, pues se cumplieron los requisitos propuestos inicialmente en el diseño y el agente exhibió el comportamiento esperado.

4.2. Caso 2: Chess horse

Habiendo probado que la arquitectura ya funciona, se continúa con la implementación del comportamiento del agente *"Chess Horse"*, un agente relativamente más complejo.

De acuerdo al documento de diseño del juego, existen personajes representados por caballos de ajedrez que en uno de los niveles, desafían al jugador a una batalla y debe superarlos para avanzar a la siguiente etapa de su viaje. El enfrentamiento ubica tanto al jugador como los caballos en tableros de ajedrez flotantes sobre un río. Cada tablero tiene un *mixer* de 2 casillas. Gana quien sea capaz de botar al rival al agua. Para ello, pueden construir martillos y arrojarlos hacia la plataforma contraria, rompiendo los bloques que la componen, haciendo

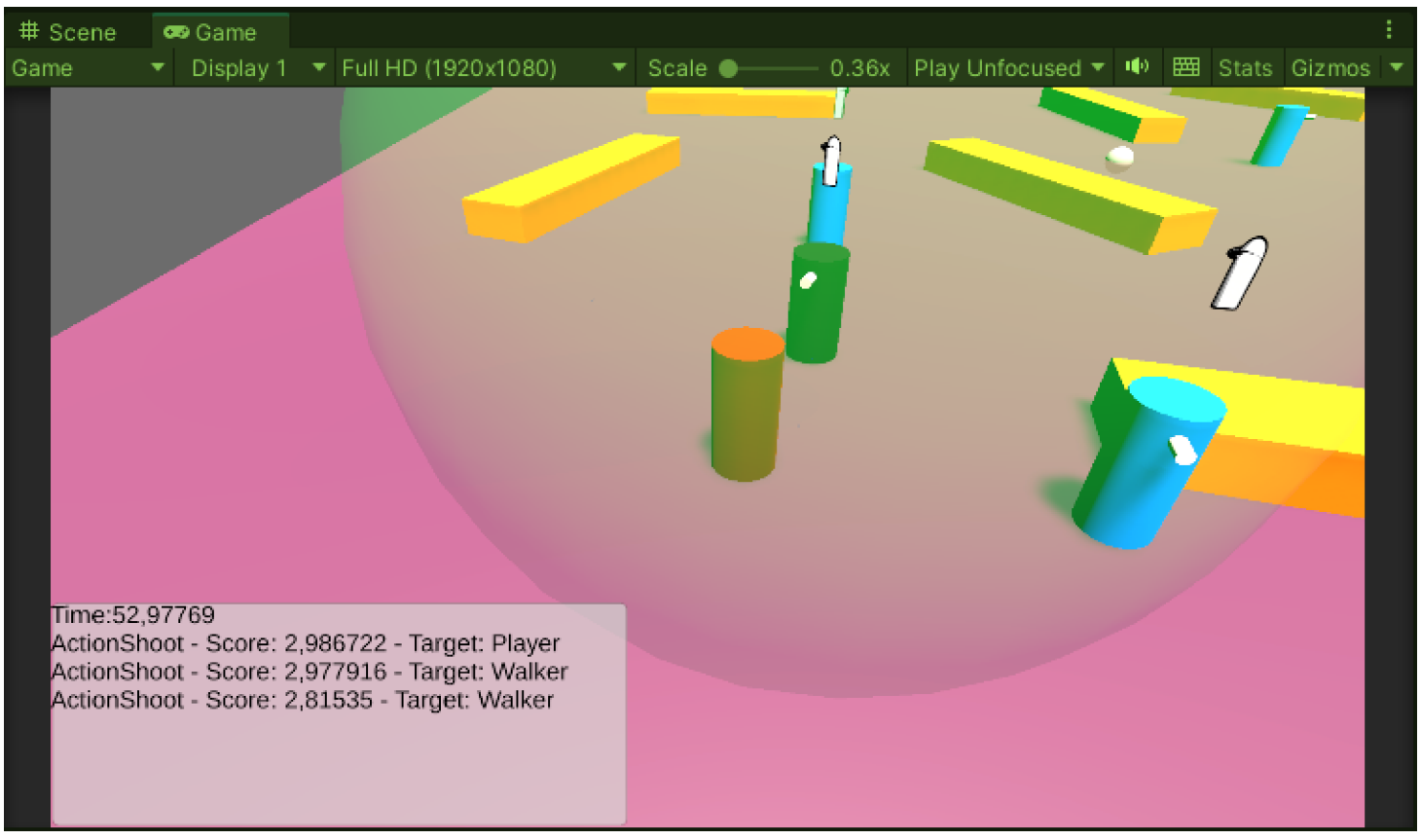


Figura 25: Ventana de *debugging* del agente **Shoter**
Fuente: Elaboración propia

caer a quien esté sobre él. Los ingredientes básicos para crear los martillos, son lanzados de manera aleatoria cada cierto tiempo sobre el tablero, desde el reloj que se encuentra allí.

Implementación

El agente usa el mismo sensor descrito anteriormente, **Sensor Objects In Range**. Se modifica la forma del *collider* asociado, con uno de tipo *Box Collider*, que es más adecuado para el nivel. También se indica que los objetos de interés deben tener el *tag "Ingredient"*, pues el agente necesita detectar aquellos ingredientes que le servirán para construir el martillo.

Para exhibir el comportamiento deseado, cuenta con 6 acciones:

1. **Action Move To Random Direction**: reutilizada del caso anterior. Al moverse, va encontrando los objetos. Se ajustó el radio de exploración para evitar que siempre se moviera a los bordes del tablero.

2. **Action Idle**: deja al agente en un estado inanimado por un tiempo determinado, configurable en el editor. Su única consideración es verificar que no tenga objetivos cerca. Sirve para reiniciar el ciclo de pensamiento en caso de no haber opciones válidas en el momento.

3. **Action Pick Object**: con esta acción se pueden tomar objetos del terreno y se asignan a la memoria del agente, para que sepa qué objeto ha tomado. Esta acción considera

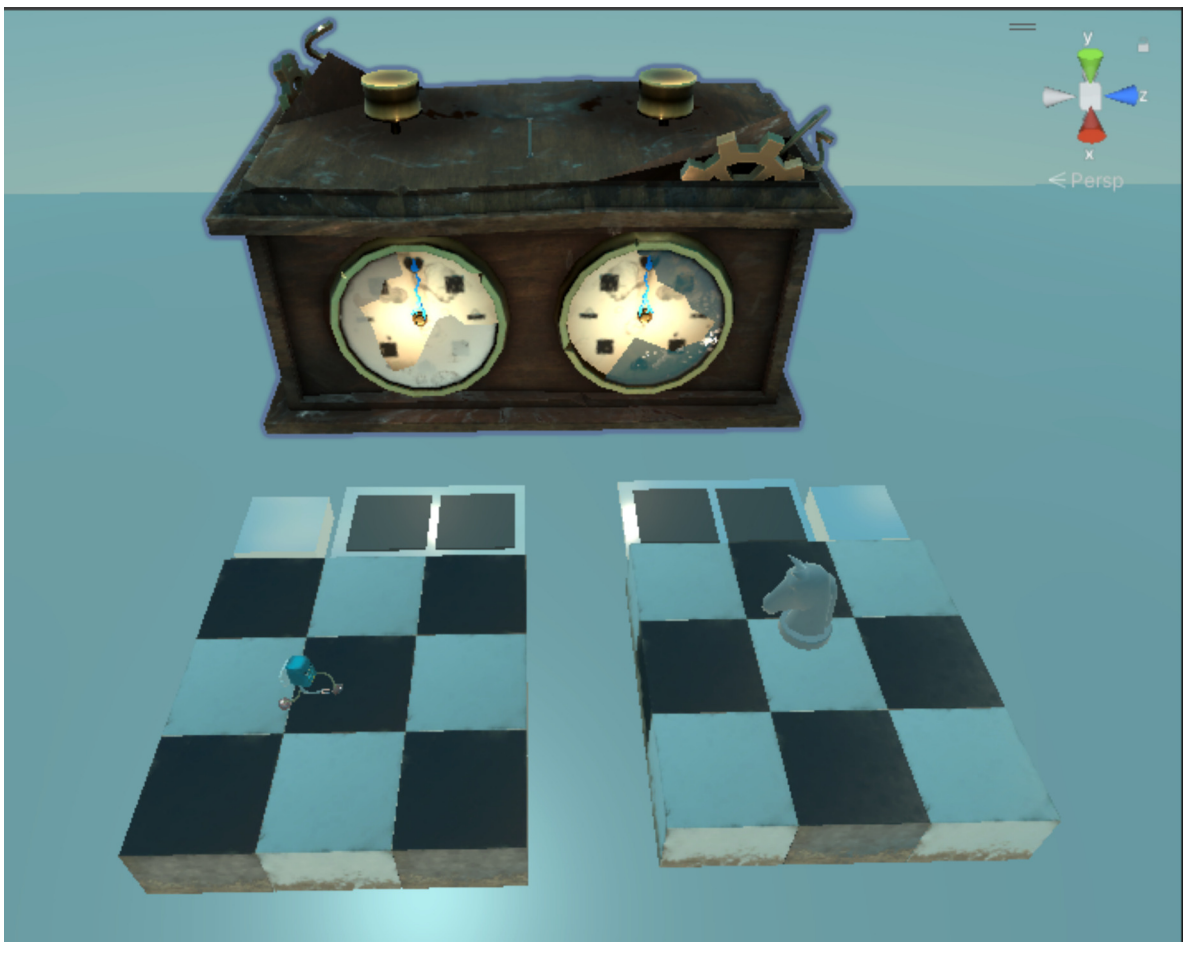


Figura 26: Nivel de batalla con el caballo

la distancia al objeto a tomar. Se puede configurar en el editor qué *tag* deben tener los objetos para ser tomados.

4. **Action Drop Item Into Mixer**: utilizada para llevar el item que el agente tomó, hacia la mesa de *crafteo*. Considera 3 aspectos: si el agente está sosteniendo algún objeto, si el *mixer* aún tiene espacio disponible para recibir ingredientes y si el *mixer* no tiene algún ingrediente con la forma del ítem que el agente tomó. Esta última consideración se elaboró porque evita el caso donde el agente llena el mixer con 2 ingredientes iguales, quedando sin posibilidad de elaborar martillos.

5. **Action Pick Object From Spawner**: si el *weapon spawner* tiene un objeto creado, el caballo intentará acercarse y tomarlo, para usarlo posteriormente. En este caso, son los martillos que debe lanzarle al jugador. Se consideran 2 aspectos para evaluarla: si el caballo no está sosteniendo algún objeto y si el *spawner* tiene un objeto listo para ser tomado. Esta acción recibe una prioridad más alta que las anteriores pues es más importante que tome el martillo creado que un nuevo material del tablero.

6. **Action Throw Object To Player**: como indica el nombre, esta acción le permite al agente arrojar objetos hacia la posición del jugador. Su puntaje de evaluación depende de 2 consideraciones: que el agente tenga tomado un objeto y que ese objeto sea un arma. El martillo es un arma, mientras que los ingredientes no, razón por la cual se verifica la segunda consideración. De lo contrario, el agente podría ayudar al jugador, arrojándole ingredientes, lo que no corresponde al comportamiento solicitado en un inicio.

Resultados

En el desarrollo de la IA del agente *Chess horse* se observó la aparición de un comportamiento no programado explícitamente: el agente fue capaz de crear un martillo recursivo. Aunque en principio no se consideró como un objetivo, pues construir un martillo simple basta para destruir uno de los bloques del tablero, dadas las reglas del juego, esto era posible. Otro factor importante fue la separación del comportamiento en varias clases de acciones "más pequeñas", que representan los pasos necesarios para completarlo. Inicialmente, se intentó programar una sola acción para construir martillos, pues parecía lo más lógico: bastaba asignar una receta en el inspector de la acción para indicarle al agente qué ingredientes debía buscar y cómo ordenarlos. Sin embargo, esto probó ser muy complejo, incluso con una receta de 2 ingredientes, principalmente porque se debían verificar muchas condiciones en cada *frame*, haciendo que el código de la clase creciera y costara seguir el flujo de su ejecución. En cambio, con acciones más pequeñas (y sus respectivas consideraciones), resultó más fácil identificar dónde habían *bugs*, pero más importante aún, se fueron ensamblando de manera natural gracias a la configuración que se les asignó en el editor.

Respecto al *feeling* del agente, en la primera validación con el profesor guía, se indicó que actuaba muy robótico, y si bien lograba llevar a cabo las acciones individuales para lanzar el martillo al jugador, se sentía un tanto aburrido y no iba acorde a la experiencia que se quería lograr. Además, en ciertas ocasiones la acción de tomar un martillo del *weapon spawner* era interrumpida por la señal recibida del sensor cuando un nuevo ingrediente aparecía cerca del agente, quien iniciaba un nuevo ciclo de evaluación, resultando en la ejecución de otra acción. Esto dejaba al martillo en un estado corrupto, flotando en el aire, supuestamente tomado por el caballo, quien a su vez intentaba lanzarle el martillo al jugador, pero no lo lograba porque perdía la referencia al objeto, debido a la interrupción. Si bien esto último se pudo corregir usando la propiedad bloqueante de las acciones, este segundo caso de prueba demostró las limitaciones y falencias de la solución propuesta.

El Anexo contiene 2 videos mostrando el funcionamiento del agente.

CAPÍTULO 5
CONCLUSIONES

La creación de comportamientos y toma de decisiones de agentes es una tarea compleja, pero necesaria para que los jugadores no se frustren y logren disfrutar de una grata experiencia, donde se puedan sentir entretenidos e inmersos en el mundo del videojuego. La arquitectura propuesta e implementada en este trabajo facilita este proceso, entregando un marco de trabajo y una base de código pequeña y robusta para el desarrollo de diferentes IA's en videojuegos, independiente del género al que pertenezcan. Los casos de prueba demostraron que si bien existen varias áreas donde mejorar el sistema, este ya puede ser utilizado para su objetivo principal. Además, separar el sistema propuesto en varios componentes permite asignarles responsabilidades claras y definidas a cada uno , lo cual es un requisito fundamental para luego identificar fácilmente dónde y por qué aparecen problemas en el desarrollo de los comportamientos y la toma de decisiones del agente.

I.A.U.S. es una arquitectura que probó ser flexible y eficaz para la toma de decisiones, por lo que su elección como guía para el desarrollo fue correcta. El uso de las consideraciones fue un factor clave que permitió crear y probar comportamientos de manera rápida y fácil, sin tener que tocar el código relacionado al *core* del sistema implementado. Como punto negativo, se puede indicar que en casos con pocos comportamientos a escoger, pareciera que se está "matando una mosca con un cañón", pues se tienen que configurar las consideraciones y *testear* varias veces , cuando otras soluciones populares como los **Behavior Trees** o **FSM** discutidos anteriormente, pueden obtener los mismos resultados, si es que no mejores, en una fracción del tiempo. Esto muestra que el sistema de utilidad adquiere relevancia cuando el número de acciones crece y las posibilidades para elegir comportamientos superan el umbral manejable por los BT's o FSM, siendo la **escalabilidad** uno de sus fuertes.

Respecto al desarrollo, una de las lecciones aprendidas es que es muy amplia la variedad de cosas que se pueden construir para crear comportamientos de agentes y uno tiende a caer en la trampa de desarrollarlas antes de necesitarlas realmente, creyendo que las usará en casos específicos, cuando en realidad terminas resolviéndolo con los mismos componentes ya creados. Irónicamente este es uno de los problemas que se listó al inicio del trabajo. Por ello, es necesario definir con anterioridad objetivos alcanzables y medibles[13], pero sobre todo, apegándose a la planificación. Filosofías de desarrollo de software como *DRY*, *YAGNI* o *KISS*, que apuntan a construir sistemas simples, evitando la duplicación de código, son guías útiles para mantener una base minimalista con la cual trabajar. A juicio del autor, menos es más.

Por otro lado, fue una buena decisión utilizar la solución de navegación de Unity, pues permitió ahorrar bastante tiempo en la creación de las pruebas que validaban la funcionalidad de la arquitectura. Algo peculiar ocurría en el caso del caballo de ajedrez, pues si se modificaban sus parámetros de velocidad en el componente de movimiento *NavMeshAgent*, con tal de

[13]Ver objetivos SMART `https://www.mindtools.com/a4wo118/smart-goals`

que se mueva más rápido por el mapa, pareciera que pensara y actuara con mayor fluidez. Esto indica que la configuración específica del agente puede modificar la percepción en el juego de cara a los jugadores, a pesar de que la arquitectura es la misma. Es decir, la manera en que las personas perciben la inteligencia del agente depende de ambas, pero pareciera que la configuración pesa más.

Usar *coroutines* como primera manera de implementar acciones tiene aspectos positivos como negativos. Estas son relativamente sencillas de usar dentro de los *scripts* de Unity y tienen la ventaja de que su tiempo de vida está ligada al *Monobehaviour* donde se está ejecutando, por lo que si el agente es destruido, es el motor el que se encarga de detenerlas y borrarlas de la memoria, liberando al programador de esta responsabilidad. Esto permite centrarse más en la lógica del *gameplay* que en aspectos de bajo nivel, relacionados al manejo de la memoria que pueden causar impactos negativos en el rendimiento, como el caso de los *memory leaks*. Separar acciones en sub-comportamientos (diferentes *coroutines* corriendo en paralelo y/o secuencialmente) es una manera efectiva de disminuir su complejidad a la hora de programarlas, pues durante su ejecución siempre se deben ir verificando pequeñas condiciones que pueden pasar desapercibidas si se hace una sola gran rutina. Eso si, saber hasta qué punto se deben separar los comportamientos en acciones pequeñas, es más arte que ciencia. Un punto negativo relacionado es que, para el caso de las acciones, es ideal tener una manera de consultar su estado de ejecución, pero las *coroutines* no tienen esta característica ni un tipo de retorno como tal (`IEnumerator` solo provee una manera de iterar a lo largo de los frames) por lo que se deben usar variables de control manual, usualmente de tipo `bool`, que pueden llevar a errores y enredan el código.

Implementar la arquitectura dentro del motor de Unity tiene inconvenientes similares a los mencionados por [Lewis, 2018] en su charla: él describió su herramienta desarrollada que implementa el **I.A.U.S**, sin depender de algún motor de videojuegos específico, por lo que dispone de muchos más recursos exclusivos para alcanzar sus objetivos. Sin embargo, para hacer pruebas de un sistema en Unity (o cualquier otro *engine*), se tiene que compartir con los demás sistemas integrados en el editor, como la *renderización* de gráficos, sonidos, físicas, *serialización* de recursos, etc., aumentando los tiempos de iteración pese a que estos sistemas no influyen directamente en la toma de decisiones del agente. Aunque es este mismo aspecto el que permite detectar tempranamente problemas de sincronización entre los sistemas, y ya que el producto final requiere que todos estos trabajen en armonía (o al menos, den esa ilusión), mientras antes se comiencen a integrar, mejor serán los resultados.

Respecto a las consideraciones, se comprendió que es mejor separarlas por tipo de agente, incluso si 2 o más tipos distintos requieren las mismas consideraciones para sus acciones. Esto quiere decir que si, por ejemplo, un agente A y otro B tienen acciones que requieren evaluar su salud actual, es mejor tener 2 consideraciones independientes, o sea 2 *assets*, que estén en diferentes carpetas, por ejemplo, *"AgenteTipoA/Consideraciones/" y "AgenteTipoB/Consideraciones/"* . De lo contrario, si ambos tipos de agentes comparten la misma referencia a la consideración en sus acciones, corremos el riesgo de alterar involuntariamente el comportamiento de alguno cuando se quiere modificar algún parámetro de la consideración para que sea más adecuada para las acciones del otro tipo de agente. En cambio, si se

tienen consideraciones independientes, pese a que en teoría hacen lo mismo, evitamos este riesgo y solo se modificará el comportamiento de un tipo de agente cuando se implementen cambios. Dado que es fácil crear nuevas instancias, esto no supone demasiados problemas, siempre y cuando se siga una estructura de nombres adecuada, para evitar confusiones al asignar referencias de consideraciones en las acciones, al editarlas en el inspector, lo que desembocaría en el problema descrito anteriormente. Otra posible solución sería implementar variables basadas en *ScriptableObjects* y sus correspondientes *Property Drawers*, como lo demuestra Ryan Hipple en su charla de Unite 2017 [Hipple, 2017]. Siguiendo este ejemplo, se podrían crear consideraciones de 2 maneras: la que ya existe, a través de *assets* que pueden referenciarse por varias acciones, y otra nueva donde se puede crear una consideración directamente en el inspector, por lo que solo estaría ligada al componente/acción que la contiene y no se crearía un nuevo archivo en el proyecto. Así, se pueden editar los parámetros de la consideración y luego se podría crear un *prefab* del agente para guardar la configuración. Lo interesante de esta segunda opción es que evita crear nuevos archivos, por lo que ya no es necesario organizarlos en carpetas ni asignarles nombres, a costa de la reutilización, ya que solo las instancias del *prefab* del agente tendrían la consideración configurada, y si otro agente quisiera tener una similar, deberá crear su propia versión, aumentando el consumo de memoria. Estos son intercambios que se deben evaluar dependiendo del tipo de juego, cantidad de agentes y las diferentes acciones que deben ejecutar.

Una de las limitaciones de la solución propuesta guarda relación con la naturaleza sensorial de los agentes, pues cada vez que detectan un evento de interés a través de los componentes que heredan de `SensorBaseClass`, su ciclo de pensamiento se activa. En situaciones donde se reciben muchas señales interesantes, por ejemplo, muchos objetos que entran al rango de detección del agente, se producen muchas evaluaciones en el cerebro (instancia de **Agent Brain**), lo que puede provocar caídas de los *FPS*.

Respecto a la usabilidad de la solución implementada, se considera que sigue siendo más difícil de aprender que una *FSM* o *BT*. Se requiere dar una buena inducción al sistema para que el o la diseñadora puedan usarlo correctamente, pues hay ciertas cosas que pasan desapercibidas como lo mencionado anteriormente sobre las mismas consideraciones siendo referenciadas por diferentes acciones, ya sea del mismo o incluso distintos agentes, ya que se puede alterar indirectamente el comportamiento de cualquiera de ellos y sin que haya un aviso claro de esto en el editor. También, si bien las curvas de respuesta proveen un gran poder para editar comportamientos, requieren al menos nociones de las funciones matemáticas más comunes y el concepto de normalización, lo que podría ser una barrera de entrada para quien no sepa del tema.

En el trabajo se describió que todas las consideraciones requieren métodos que tengan la misma firma. Esto permite que, usando reflexión, podamos cambiar los métodos asignados en el inspector y que en tiempo de ejecución, sean capaces de obtener la información necesaria del estado del juego, sin arrojar excepciones a causa de los argumentos recibidos. Como se dijo anteriormente, esta decisión de diseño se tomó para aumentar la reusabilidad del código (no tener una clase distinta por cada consideración) y por la facilidad con la que se pueden crear y editar nuevas consideraciones, sin tener que modificar el código subyacen-

te. Sin embargo, en los casos donde el método asignado verifica alguna condición binaria, como si el agente tiene o no objetivos, es más intuitivo retornar tipos de datos booleanos (*true* o *false*), en vez de flotantes (1 o 0). Si bien es común hacer una analogía entre estos valores, 1 es *true* y 0 es *false*, esto no es una regla de C#, pues aquí no existen los valores *truthy/falsy* como en Pyhton o JavaScript. Sumado a esto, la interacción entre el valor de retorno del método y la curva de respuesta, hace un poco más difícil entender el valor final de la consideración para alguien que estuviera aprendiendo a usar la herramienta.

Trabajo a futuro

Implementar interfaces que permitan desacoplar los sistemas de IA, específicamente las acciones implementadas, y movimiento, de tal manera que si se decide utilizar otra solución de navegación, se pueda reemplazar sin que las acciones de la IA se vean afectadas.

Suena interesante implementar una arquitectura híbrida con componentes similares a la de un Behavior Tree, con selectores y secuenciadores, pero manteniendo la flexibilidad del sistema de Utilidad, esto es, que el agente sea capaz de razonar y actuar según sea lo más apropiado en ese contexto, en vez de seguir una estructura rígida de pensamiento en cada situación. Esto permitiría obtener lo mejor de ambas. A su vez, las acciones se podrían agrupar en comportamientos de más alto nivel, parecido a lo que hacen las *Hierarchical FSM* o *HTN*, para que sean más fáciles de modelar y comprender por diseñadores, mientras que faciliten su implementación a los programadores. De hecho, ya existe una implementación de nodos de utilidad para mejorar la manera en que priorizan los distintos caminos de opciones de un BT, donde se explican las ventajas de esta unión, como la posibilidad de evaluar el árbol paralelamente a la ejecución de acciones o la de aplicar este tipo de evaluaciones con utilidad solo donde se estime conveniente [Merril, 2013]. Esto es algo que Kevin Dill, en su paper sobre el *framework* GAIA, definía como “complejidad concentrada” [Dill, 2019].

Continuando con las acciones, en este caso aquellas que pueden aplicarse a diferentes objetivos, actualmente se revisan todas las consideraciones por cada uno de ellos. Sin embargo, a veces estas evalúan aspectos que no dependen del objetivo, por lo que se está consultando la misma información m veces, donde m es el total de objetivos en la actual ronda de evaluación, cuando podría hacerse solo una vez, ahorrando recursos valiosos en el caso de que hubieran múltiples agentes con una o más de este tipo de acciones. Un ejemplo claro de esto sería si un agente quiere disparar, pero no tiene balas, basta que tal consideración se verifique una vez para anular el comportamiento, en vez de pasar por cada objetivo, sabiendo que tendremos el mismo resultado. En este aspecto, se propone dividir entre **requisitos** y consideraciones. Los requisitos serían muy similares a las consideraciones, pero con la diferencia de ser evaluados una vez y antes que cualquier consideración por objetivo. Esto facilitaría el proceso de eliminación de una acción en la ronda de selección de comportamiento, pues tal como se hace actualmente con las consideraciones, si algún **requisito** de la acción no se cumpliera, automáticamente quedaría fuera y no se necesitaría evaluar consideraciones por objetivo. Estas propuestas apuntan a mejorar la **escalabilidad** del sistema.

El sistema de IA actualmente no considera la dificultad del juego, ni existe una manera directa de alterar su comportamiento general para que sea más fácil o difícil de superar. Para efectos del desarrollo, este aspecto no fue impedimento para crear y probar los comportamientos de los agentes. El o la diseñadora tiene la capacidad de ajustar parámetros en las distintas instancias de las acciones que componen al agente, sin embargo esto requiere períodos de *testing* relativamente largos para encontrar aquellos ajustes adecuados a la experiencia.

En la charla de Johannes Ahvenniemi en Unite 2019 [Ahvenniemi, 2019], se contrasta en gran detalle las diferencias de usar el flujo de trabajo *async/await*, introducido originalmente al *engine* de Unity en 2017, en relación a las *coroutines*, que fueron implementaciones de rutinas asíncronas antes de la llegada de esta característica.

Figura 27: Diferencias entre coroutines y async
Fuente: [Ahvenniemi, 2019]

Si bien se indica en la charla que *async* no es un reemplazo de las *coroutines*, suena bastante tentador hacerlo, puesto que en el flujo *async* existen las *Tasks*, que permiten saber en qué estado se encuentra la ejecución del proceso asíncrono. Esto podría resolver el problema mencionado de que las *coroutines* no tienen valores de retorno y por ende es difícil *debuggear* en qué etapa va la acción. Además, se menciona que los métodos *async* retienen la pila de ejecución, por lo que podemos saber de dónde fue llamada para *debuggearla*, mientras que las *coroutines* perdemos parte de esa información y es más complicado encontrarlo en el *Profiler* de Unity.

Se podría complementar el sistema con otras fuentes de información como mapas de influencia y sistemas de etiquetas. Esto es algo que ya hicieron en la compañía de Dave Mark[14]

[14] `https://intrinsicalgorithm.com/techs.php`

y serviría para ampliar las capacidades sensoriales y de actuación de los agentes, pues estos complementos entregan información útil, difícil de obtener y habilitan al agente para ser más inteligente.

En foros[15] se menciona que en vez de multiplicar los puntajes de las consideraciones, se podría hacer un promedio de los valores, eliminando automáticamente los comportamientos que tengan alguna consideración con valor 0. De esta manera, se evitaría el cómputo del factor de escala que al parecer genera variaciones importantes en la puntuación final de una acción, inflando aquellas que poseen mayor cantidad de consideraciones. Para comprobar esto, se necesita establecer *benchmarks* y correr pruebas de escalabilidad que permitan visualizar el rendimiento de la lógica encargada de asignar puntajes y elegir qué comportamiento tomar. Así, se pueden conocer los límites de la solución (cuántos agentes pueden estar en la escena a la vez, qué tantas consideraciones se pueden evaluar por unidad de tiempo, etc.) e identificar los cuellos de botellas para optimizarlos.

[15] https://www.reddit.com/r/gameai/comments/lj8k3o/infinite_axis_utility_ai_a_few_questions/

REFERENCIAS BIBLIOGRÁFICAS

[Ahvenniemi, 2019] Ahvenniemi, J. (2019). Best practices: Async vs. coroutines - unite copenhagen - youtube. https://www.youtube.com/watch?v=7eKi6NKri6I. Recuperado el 14/04/2023.

[Al Enezi and Verbrugge, 2020] Al Enezi, W. and Verbrugge, C. (2020). Dynamic guard patrol in stealth games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 160–166.

[Anguelov, 2019] Anguelov, R. (2019). AI Arborist: Proper Cultivation and Care for Your Behavior Trees.

[Anguelov, 2022] Anguelov, R. (2022). Game AI Basics.

[Arm, 2021] Arm (2021). What is a gaming engine? https://www.arm.com/glossary/gaming-engines Recuperado el 01/12/2021.

[Bagnell et al., 2012] Bagnell, J. A., Cavalcanti, F., Cui, L., Galluzzo, T., Hebert, M., Kazemi, M., Klingensmith, M., Libby, J., Liu, T. Y., Pollard, N., Pivtoraiko, M., Valois, J.-S., and Zhu, R. (2012). An integrated system for autonomous robotics manipulation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2955–2962.

[Bevilacqua, 2013] Bevilacqua, F. (2013). Finite-state machines: Theory and implementation. https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867. Recuperado el 08/05/2023).

[Bulitko et al., 2018] Bulitko, V., Walters, M., and Brown, M. R. (2018). Evolving npc behaviours in a-life with player proxies. In *AIIDE Workshops*.

[Chernikov, 2018] Chernikov, Y. (2018). What is a game engine? https://www.youtube.com/watch?v=vtWdgtMo1T4 Recuperado el 01/12/2021.

[Clement, 2023] Clement, J. (2023). Video game industry revenue & market share (jun 2023). https://www.statista.com/statistics/1344668/revenue-video-game-worldwide/. (Recuperado el 20/06/2023).

[Daeller, 2018] Daeller, F. (2018). Machines That Sense, Think, And Act.

[Deepmind, 2019] Deepmind, I. (2019). AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning.

[Dill, 2013] Dill, K. (2013). Structural architecture common tricks of the trade. http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter05_Structural_Architecture_Common_Tricks_of_the_Trade.pdf. Recuperado el 16/02/2023).

[Dill, 2019] Dill, K. (2019). Introducing GAIA: A Reusable, Extensible Architecture for AI Behavior. In *Game AI Pro 360*. 1 edition.

[Franklin and Graesser, 1996] Franklin, S. and Graesser, A. (1996). Is it an agent, or just a program?: A taxonomy for autonomous agents. pages 21–35.

[Godot, 2020] Godot (2020). What is visual scripting? `https://docs.godotengine.org/es/stable/getting_started/scripting/visual_script/what_is_visual_scripting.html` Recuperado 03/12/2021.

[Graham, 2019] Graham (2019). Stop Fighting! Systems for Non-Combat AI.

[Graham, 2013] Graham, D. (2013). An introduction to utility theory.

[Grant and Lardner, 1952] Grant, E. and Lardner, R. (1952). It. *The New Yorker*.

[Heckel et al., 2010] Heckel, F. W. P., Youngblood, G. M., and Ketkar, N. S. (2010). Representational complexity of reactive agents. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 257–264.

[Hipple, 2017] Hipple, R. (2017). Unite Austin 2017 - Game Architecture with Scriptable Objects.

[Humphreys, 2013] Humphreys, T. (2013). Exploring HTN Planners through Example.

[Iovino et al., 2022] Iovino, M., Scukins, E., Styrud, J., Ögren, P., and Smith, C. (2022). A survey of Behavior Trees in robotics and AI. *Robotics and Autonomous Systems*, 154:104096.

[Jagdale, 2021] Jagdale, D. (2021). Finite state machine in game development. pages 384–390.

[Kopel and Hajas, 2018] Kopel, M. and Hajas, T. (2018). Implementing ai for non-player characters in 3d video games. In *ACIIDS*.

[Lewis, 2018] Lewis, M. (2018). Winding Road Ahead: Designing Utility AI with Curvature.

[Lidén, 2003] Lidén, L. (2003). Artificial stupidity: The art of intentional mistakes. *AI game programming wisdom*, Vol 2:41–48.

[Mark, 2013] Mark, D. (2013). Architecture tricks: Managing behaviors in time, space, and depth.

[Merril, 2013] Merril, B. (2013). Building Utility Decisions into Your Existing Behavior Tree.

[Nirwan, 2021] Nirwan, D. (2021). Hierarchical finite state machine for ai acting engine. `https://towardsdatascience.com/hierarchical-finite-state-machine-for-ai-acting-engine-9b24efc66f2` Recuperado el 07/12/2021.

[Ogren, 2012] Ogren, P. (2012). Increasing modularity of uav control systems using computer game behavior trees.

[OpenAI, 2019] OpenAI (2019). OpenAI Five defeats Dota 2 world champions.

[Orkin, 2006] Orkin, J. (2006). Three states and a plan: the ai of fear. In *Game developers conference*, volume 2006, page 4. CMP Game Group SanJose, California.

[Rabin, 2019] Rabin, S. (2019). Game AI Architecture: The Problem of Human Authored Behavior.

[Russell and Norvig, 2010] Russell, S. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition.

[Saagie, 2020] Saagie (2020). Artificial intelligence in video games. `https://saagie.com/blog/artificial-intelligence-in-video-games/` Recuperado el 07/12/2021.

[Safadi et al., 2015] Safadi, F., Fonteneau, R., and Ernst, D. (2015). Artificial intelligence in video games: Towards a unified framework. *Int. J. Comput. Games Technol.*, 2015:271296:1–271296:30.

[Savaglia, 2021] Savaglia, L. (2021). Artificial intelligence in gaming: Creating a living world and its npcs. Technical report, EasyChair.

[Siegel, 2003] Siegel, M. (2003). The sense-think-act paradigm revisited. In *1st International Workshop on Robotic Sensing, 2003. ROSE' 03.*, pages 5 pp.–.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

[Smithsonian, 2016] Smithsonian (2016). Video game history. `https://www.si.edu/spotlight/the-father-of-the-video-game-the-ralph-baer-prototypes-and-electronic-gam video-game-history` Recuperado el 10/12/2021.

[Sánchez-Ruiz et al., 2008] Sánchez-Ruiz, A., Stephen, R., Héctor, L.-U., Noz-Avila, M., Belén, D.-A., and Gonzalez-Calero, P. (2008). Game ai for a turn-based strategy game with plan adaptation and ontology-based retrieval.

[Thompson, 2019] Thompson, T. (2019). Behaviour Trees: The Cornerstone of Modern Game AI | AI 101.

[Turing and Haugeland, 1950] Turing, A. M. and Haugeland, J. (1950). *Computing machinery and intelligence*. MIT Press Cambridge, MA.

[Unity, 2020a] Unity (2020a). Unity - manual: Coroutines. `https://docs.unity3d.com/Manual/Coroutines.html`. Recuperado el 17/05/2022.

[Unity, 2020b] Unity (2020b). Unity - manual: Important classes - gameobject. `https://docs.unity3d.com/Manual/class-GameObject.html`. Recuperado el 17/05/2022.

[Unity, 2020c] Unity (2020c). Unity - manual: Important classes - monobehaviour. `https://docs.unity3d.com/Manual/class-MonoBehaviour.html`. Recuperado el 17/05/2022.

[Valencia-García et al., 2016] Valencia-García, R., Lagos-Ortiz, K., Alcaraz-Mármol, G., Cioppo, J., and Vera-Lucio, N. (2016). *Technologies and Innovation: Second International Conference, CITI 2016, Guayaquil, Ecuador, November 23-25, 2016, Proceedings*. Communications in Computer and Information Science. Springer International Publishing.

[Warpefelt, 2016] Warpefelt, H. (2016). The non-player character : Exploring the believability of npc presentation and behavior.

[Wikipedia, 2020] Wikipedia (2020). Motores de juego. `https://es.wikipedia.org/wiki/Anexo:Motores_de_juego`, Recuperado: 07/12/2021.

[Yannakakis and Togelius, 2018] Yannakakis, G. N. and Togelius, J. (2018). Artificial intelligence and games. In *Springer International Publishing*.

[Zhou et al., 2006] Zhou, C.-N., Yu, X.-L., Sun, J.-Y., and Yan, X.-L. (2006). Affective computation based npc behaviors modeling. In *2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology Workshops*, pages 343–346. IEEE.

ANEXOS

Videos de las pruebas

- Agente **Shooter** funcionando: `https://drive.google.com/file/d/1iZ3GRDx0iabQQ0mXKMjgYDN95OoLQvfB/view?usp=sharing`
- Agente **Chess horse** creando martillos recursivos: `https://drive.google.com/file/d/1DIgFxHaEifyLJlH9zRLzoudNn63Ona2X/view?usp=sharing`
- Batalla con el **Chess horse**: `https://drive.google.com/file/d/1hnfhx59VauAFkcY-EUeOzh3m4v7nPpmA/view?usp=sharing`

Documentación de las clases implementadas

Aquí se detallan miembros definidos en las diferentes clases principales de la arquitectura implementada.

Sensor Base

Campos

- `bool` **Debug**: útil para depurar en el Editor si los sensores están detectando la información requerida.
- `AgentMemory` **Agent Memory**: referencia a la memoria asignada del agente, en caso de que el sensor necesite actualizar alguna de sus propiedades.
- `System.Action` **OnSensorUpdate**: evento que cada sensor debe disparar para dar aviso de que ha encontrado nueva información del ambiente.

Agent Brain

Campos

- `enum PickMethod` **PickMethod**: Un método de selección para escoger entre las diferentes opciones. Los métodos son `MaxScore`, `AllRandom`, `WeightedAllRandom`, `TopN`

- `int` **top N**: Solo se usa si el método de selección es `TopN` y sirve para considerar hasta N opciones de la lista de opciones en la actual evaluación.
- `ActionBase` **Default Action**: Una acción por defecto que el agente ejecutará cuando no existan otras opciones válidas. Usualmente, esta es una acción de tipo *Idle*.

Métodos

- `private Option GetNewOption`: envía las acciones (activas en el Editor) del agente hacia la clase **Utility System**, que le retorna una opción válida o `null` en caso que no exista alguna.
- `public void TryStartNewAction`: llama a `GetNewOption`. Si existe la opción válida, se pasa al **Action Runner** para que verifique si se puede ejecutar o no. Si no existe opción válida, se ejecuta la acción por defecto del agente.

Action Base

Campos

- `float` **Priority**: indica la importancia relativa de esta acción en comparación con el resto de acciones asociadas al agente. Multiplica al valor final resultante de la evaluación de las consideraciones de esta acción, de manera que se pueda aumentar o disminuir su relevancia según se estime conveniente. Esto es útil, si tenemos acciones que son urgentes y necesitan ser escogidas cuando se cumplen las condiciones, como por ejemplo, tomar una poción de vida cuando la salud del personaje está críticamente baja.
- `float` **Initial Cooldown**: indica cuánto tiempo estará en enfriamiento una acción luego de su ejecución.
- `bool` **Debug**: útil para depurar los procesos internos de una acción particular dentro del Editor, como los puntajes obtenidos, cuándo termina o es interrumpida, número de ejecuciones totales, etc.
- `List<Consideration>` **Considerations**: es una lista que define los aspectos (ejes según la nomenclatura del I.A.U.S.) que una acción evaluará para obtener su puntaje, como distancia a objetivos, nivel de alerta, salud del agente, etc.

Propiedades

- `NavMeshAgent` **Local Navmesh Agent**: referencia al componente con el cual el agente se puede desplazar por las *navmesh* del terreno.
- `AgentMemory` **Local Agent Memory**: referencia al componente *Agent Memory*, asignado al agente, donde se aloja su estado interno.
- `float` **Action Cooldown**: representa el enfriamiento restante de la acción. Es actualizado en cada frame.
- `bool` **Is Running:** indica si esta acción se está ejecutando aún.
- `System.Action` **OnFinishedAction**: evento que da aviso cuando una acción ha culminado exitosamente.
- `bool` **Is Blocked**: indica si una acción está ejecutando un proceso que no debería ser interrumpido por otras acciones.

Métodos

- `public abstract List<Option> ScoreThisAction()`: asigna un puntaje a la acción según el contexto y su número de objetivos. Retorna la lista de opciones posibles a ejecutar. Cada acción debe definir su propia implementación de este método.
- `public void StartExecution(GameObject target = null)`: Comienza la ejecución de la acción correspondiente, recibiendo un objetivo (*target*) de ser necesario.
- `public void InterruptExecution()`: si el **Action Runner** recibe una nueva opción a ejecutar, invoca este método sobre la opción que ya se esté ejecutando, para interrumpirla. Este método debe asegurar que el agente no quede en un estado corrupto, por lo que debe limpiar variables, *coroutines* y otras propiedades correspondientes.
- `public void FinishExecution()`: se llama este método cuando la acción llega a su término. Aquí se debe disparar el evento **OnActionFinished** para que el ciclo de pensamiento se reinicie, además de asignar las variables necesarias para indicar los efectos de esta acción, ya sea en el estado del mundo o la memoria del agente.
- `protected private abstract IEnumerator Act(GameObject t = null)`: esta es la *corutine* principal que se invoca cuando comienza la ejecución de una nueva acción. Dentro de esta se implementa la lógica propia del agente (moverse, reproducir animaciones, etc) y debe tener alguna condición de término donde llame al método `FinishExecution`.
- `private protected float EvaluateConsiderations(GameObject t = null)`: aquí se calcula el puntaje de cada consideración. En caso de encontrar que alguna retorne 0, inmediatamente termina la ejecución del método, retornando `null`.

- `private protected void ApplyScaleFactorToOptionScore(Option option)`: toma una opción y le aplica un factor de escala, para corregir el puntaje que tiene según el número de consideraciones asignadas a la acción. Esto pues mientras mayor es el número, el puntaje final tiende a decaer puesto que se están multiplicando valores normalizados.

- `private protected List<Option> GetMultipleScoredOptions(arg)`: obtiene el puntaje de una acción cuando tiene varios objetivos posibles, retornando la lista de opciones válidas. Recibe un argumento (*arg*) de tipo `List<GameObject> targets`.

- `public void ReceiveContext(NavMeshAgent nma, AgentMemory am)`: función auxiliar que permite recibir las referencias del componente de movimiento y memoria del agente desde el **Agent Brain**. De esta manera, no es necesario asignarlas en el Editor de manera manual.

Utility System

Métodos

- `public static List<Option> ScorePossibleOptions(List<ActionBase> a)`: recibe una lista de acciones, obtiene sus puntajes y los agrupa junto a sus objetivos (si es que existen) en diferentes opciones. Acciones que sean inválidas no se agregan a la lista final, por lo que no serán parte de la ronda de decisión del agente.

- `public static Option PickFromScoredOptions(args)`: recibe una lista de opciones y escoge alguna según el método que se le indica, retornándola al **Agent Brain** del agente que invocó el método. Los argumentos (*args*) son:
  `List<Option> options,`
  `PickMethod pickMethod = PickMethod.MaxScore,`
  `int topOptionsToConsider = 1`

Action Runner

Propiedades

- `public System.Action` **OnFinishedExecution**: se invoca este evento cuando se termina de ejecutar una acción correctamente (no cuando se interrumpe).

- `public bool` **IsRunning**: permite saber si se está ejecutando alguna acción en este momento.

Métodos

- `public void TryExecuteOption(Option newOption)`: recibe una nueva opción, evaluando si es posible ejecutarla o no, dependiendo si en ese frame la opción anterior no está bloqueada.
- `private void BeginNewExecution(Option option)`: este método inicia la ejecución de una opción, suscribiéndose al evento disparado por esta cuando termine de ejecutarse y guardando la referencia en caso de tener que interrumpirla más adelante.
- `private void InterruptExecution(Option option)`: interrumpe la ejecución de una opción, de-suscribiéndose de esta y eliminando su referencia.
- `private void FinishExecution()`: se llama cuando termina la ejecución de la opción actual, disparando el evento **OnFinishedExecution** y eliminando la referencia de la opción actual.

Consideration

Campos

- `bool` **Bookends**: permite decidir si esta consideración necesita definir cotas para normalizar el valor de entrada para la curva de respuesta.
- `float` **minValue**: cota inferior para el valor de entrada. Solo se muestra en el editor si **Bookends** se asigna a `true`.
- `float` **maxValue**: cota superior para el valor de entrada. Solo se muestra en el editor si **Bookends** se asigna a `true`.
- `AnimationCurve` **responseCurve**: curva de respuesta de la consideración, que puede ser editada para modificar la importancia relativa de esta propiedad en el puntaje de la acción a la cual se asigne.
- `Object` **ImplementationReference**: instancia de alguna clase que implementa los métodos que se pueden escoger para la consideración, de manera que pueda obtener información del estado del juego.
- `BindingFlags flags`: un `enum` que define las características que deben cumplir los métodos definidos en **ImplementationReference** para ser mostrados en el inspector. Por defecto deben ser públicos, estáticos y declarados.
- `List<string>` **methods**: la lista de métodos válidos de **ImplementationReference** que se pueden escoger para que la consideración obtenga información del estado del juego.

Métodos

- `public float GetValue(AgentMemory am, GameObject t = null)`: este método usa reflexión para obtener el valor de la propiedad de interés de la consideración. A través de la referencia asignada en el editor, se puede invocar un método en una clase distinta pasando como parámetros la memoria del agente y un objetivo (si es que existe). Siempre retorna un `float`.