Desarrollo de un videojuego 3D e implementación de Agentes Inteligentes

TITULACIÓN: Grado en Ingeniería Informática

AUTOR: Aarón Hernández Álvarez

TUTORIZADO POR:

Agustín Rafael Trujillo Pino

Fecha: marzo de 2022

INDICE

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE ASSETS EMPLEADOS

1. *Boxes pack*. Publisher: Aleksey Kozhemyakin
2. *Crate and Barrels*. Publisher: Kobra Game Studios
3. *FPS Icons Pack*. Publisher: Infima Games
4. *Free Pack Woden Planks*. Publisher: Lord Enot
5. *Free Zombie Character Sounds*. Publisher: Idia Software LLC.
6. *Pallet Plywood LODs*. Publisher: Gerhald3D.
7. *Post Apocalypse Guns Demo*. Publisher: Sound Earth Game Audio.
8. *Shotgun*. Publisher: Adequate.
9. *Snaps Prototype | Sci-Fi / Industrial*. Publisher: Assets Store Originals.
10. *Stylized Sci-fi Texture.* Publisher: LowlyPoly.
11. *War FX*. Publisher: Jean Moreno.
12. *Weapon Pack - AK (Kalashnikov) – FREE*. Publisher: TessaraOxygen.
13. *Weapon Pack - Makarov (PM) – FREE*. Publisher: TessaraOxygen.
14. *Wooden Floor Materials*. Publisher: Casual2D.
15. *Yughues Free Concrete Materials*. Publisher: Nobiax /Yughues.
16. *Yughues Free Metal Materials*. Publisher: Nobiax /Yughues.
17. *Zombie.* Publisher: Pxltiger.

RESUMEN

Uno de los objetivos más importantes dentro del desarrollo de un videojuego es la implementación e integración de los elementos que conforman el entorno jugable, los cuales facilitan y encauzan la inmersión del jugador durante la experiencia de juego. Entre ellos hay que destacar el papel de los agentes inteligentes. Un desarrollo bien ejecutado de los sistemas que componen los agentes, así como sus estrategias de interacción con el entorno puede generar en el jugador retos durante el tiempo de juego, que contribuyan a una experiencia de juego satisfactoria. El objetivo de este proyecto es la implementación de un sistema inteligente compuesto por agentes coordinados dentro de un videojuego de disparos en primera persona, estudiando estructuras ya establecidas como las máquinas de estados finitos, los sistemas de rutas o los árboles de comportamiento. Como resultado se dispone de una versión de demostración del videojuego propuesto, en la que es posible escoger diferentes niveles de dificultad que generan en la escena de juego agentes con sistemas de diferente nivel de complejidad. Para el desarrollo se empleó el motor Unity, que provee de herramientas para una puesta en producción rápida de un producto de este tipo, y el lenguaje de programación C#, nativo del propio motor. Además, para la prueba y verificación se hizo uso de la librería NUnit, orientada a la prueba y verificación del código implicado. Por otro lado, se emplearon herramientas de modelado, como SketchUp para modelos y bocetos del entorno. Se codificaron varias versiones de un mismo enemigo, centrándose el proceso en el análisis e implementación dos estructuras habitualmente empleadas para construir sistemas inteligentes, las máquinas de estado y los árboles de comportamiento. Se realizaron además simulaciones sobre las diferentes versiones de los árboles construidos para comprobar el aumento de dificultad para un jugador de características similares. Una vez finalizadas las pruebas y analizadas ambas estructuras, se concluye que los árboles ofrecen mayores beneficios respecto a las máquinas de estado, dada su estructura modular, su escalabilidad pocencial y su capacidad de optimización, entre otros aspectos.

ABSTRACT

One of the key goals in the development of a videogame is the implementation and integration of the elements that make up the playable environment, which ease and channel the player's immersion during the gaming experience. Among them, the role of intelligent agents should be highlighted. The aim of this project is to implement an Intelligent System on a First Person Shooter, composed by coordinated Intelligent Agents system, on a First Person Shooter, analysing already established structures and paradigms, such as finite states machines, waypoint systems or behavior trees. As a result, a demo version of the game proposed is available. It is possible to between different levels of difficulty, each one generating agents with escalating complexity within the game scene. Unity engine will be used for the development of the game, as it provides tools for a fast production start-up of an asset of this type, and the C# programming language, native to the engine itself. In addition, the NUnit library will be used for testing and verification of the code involved. On the other hand, modelling tools, such as Blender3D, will be used for the realization and modification of models within the game. Modelling tools, such as SketchUp, were used for sketching during preliminary stages. Several versions of a default enemy were coded, pivoting the process around analysis and implementation of two commonly used structures in this area, finite state machines and behavior trees. Simulations were conducted over the behavior trees built during development to verify the increase on difficulty level between versions. Once the tests have been completed and both structures have been analyzed, it is concluded that trees offer greater benefits compared to state machines, given their modular structure, their potential scalability and their optimization capacity, among other aspects.

1 PREFACIO

1.1 ¿Por qué un videojuego?

Al formar parte de una rama de la ingeniería en la que la investigación y aplicación directa de nuevas soluciones de negocio son altamente demandadas casi a diario, haber tomado como objetivo un proyecto orientado al entretenimiento puede parecer extraño, o una locura.

Mientras una parte significativa del estudiantado con el que he tenido contacto a lo largo de la carrera ha optado por aplicar soluciones informáticas en trabajos relacionados con áreas del conocimiento casi opuestas a la informática, como pueden ser la medicina general o la psicología, o bien llevar a cabo proyectos ya establecidos con anterioridad por el profesorado, yo he tomado la decisión de desarrollar un videojuego desde cero, en aproximadamente cuatro meses, documentándolo y con el objetivo de conseguir funcione de forma meridanamente correcta. La pregunta más obvia que surge es *¿Por qué?*

Desde que tengo uso de razón he tenido contacto con ordenadores, y a día de hoy no hay día que pase sin que me siga sorprendiendo de la cantidad de detalles que esconden y que pasamos por alto. Me apasiona la informática, cualquiera de sus posibles vías de conocimiento me resulta fascinante, desde el mantenimiento e instalación de equipos hasta el *Machine Learning*, pasando por cosas tan diametralmente opuestas entre sí como la configuración de una sumatoria de dos números en un simulador MIPS, la instalación de una DMZ en una red local o el diseño de una interfaz de usuario para una aplicación de reserva de vuelos. Pero si hay una disciplina que siempre me ha llamado la atención, esa es el desarrollo de videojuegos.

Quizás sea porque el primer contacto del que tengo constancia fue con un juego sencillo de pintar y colorear, o tal vez por pasarme días enteros hasta altas horas de la noche viendo análisis y tutoriales técnicos acerca del diseño y programación de videojuegos, o incluso por compartir momentos con amigos en partidas a lo largo de los años, pero después de dos años de un ciclo superior y otros cuatro de carrera en los que he tocado un poco de todo, desarrollar un videojuego por mí mismo supone afrontar un reto personal que llevo tiempo esperando poder atajar.

1.2 Objetivos del proyecto

1.2.1 Objetivos académicos

Al tratarse de un proyecto eminentemente académico, quedan incluidos en él una serie de objetivos de dicha índole. Dentro de los planteados para este proyecto, se destacan los siguientes:

- Aplicar los conocimientos obtenidos durante el Grado en materia de investigación.
- Poner en práctica los conocimientos en materia de gestión de proyectos de software, adquiridos en la especialidad.
- Poner en práctica los conocimientos relativos a Inteligencia Artificial aprendidos en la asignatura de Fundamentos de Sistemas Inteligentes.
- Ampliar conocimientos actuales sobre Sistemas Inteligentes, sus componentes y metodologías de implementación dentro de un proyecto de software orientado a un público general.

1.2.2 Objetivos técnicos

A nivel técnico, se plantean los siguientes objetivos:

- Entender con cierta soltura el funcionamiento a nivel interno de un motor de videojuegos presente en el mercado.
- Analizar, comprender e implementar patrones arquitectónicos y de diseño orientados al desarrollo de videojuegos en un proyecto real.
- Analizar, comprender e implementar metodologías y patrones de optimización, comparando los resultados en versiones con y sin dichas estrategias.

1.3 Competencias

1.3.1 Competencias generales

De acuerdo con los establecido en el plan de estudios actual, y teniendo en cuenta las características del proyecto en el ámbito de la informática, las competencias generales incluidas en este proyecto, las competencias generales a destacar son las siguientes:

- **G2:** Aplicar sus conocimientos a su trabajo o vocación de una forma profesional y posean las competencias que suelen demostrarse por medio de la elaboración y defensa de argumentos y la resolución de problemas dentro de su área de estudio;
- **G3:** Reunir e interpretar datos relevantes (normalmente dentro de su área de estudio) para emitir juicios que incluyan una reflexión sobre temas relevantes de índole social, científica o ética;
- **G4:** Transmitir información, ideas, problemas y soluciones a un público tanto especializado como non especializado;

- **G5:** Desarrollar aquellas habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía.

1.3.2 Competencias específicas

En su entidad de proyecto de software, las competencias específicas relativas a la Mención de Software implicadas en el desarrollo son las siguientes:

- **IS01:** Capacidad para desarrollar, mantener y evaluar servicios y sistemas software que satisfagan todos los requisitos del usuario y se comporten de forma fiable y eficiente, sean asequibles de desarrollar y mantener y cumplan normas de calidad, aplicando las teorías, principios, métodos y prácticas de la ingeniería del software.
- **IS04:** Capacidad de identificar y analizar problemas y diseñar, desarrollar, implementar, verificar y documentar soluciones software sobre la base de un conocimiento adecuado de las teorías, modelos y técnicas actuales.
- **IS05:** Capacidad de identificar, evaluar y gestionar los riesgos potenciales asociados que pudieran presentarse.
- **IS06:** Capacidad para diseñar soluciones apropiadas en uno o más dominios de aplicación utilizando métodos de la ingeniería del software que integren aspectos éticos, sociales, legales y económicos.

2 ESTADO DEL ARTE

Este apartado describe el estado actual del desarrollo de sistemas multiagente y, concretamente de su integración en los elementos IA dentro del sector de los videojuegos, así como su evolución desde mediados del siglo XX hasta la actualidad.

2.1.1 Agentes Inteligentes

Las entidades nucleares de los Sistemas Inteligentes son los denominados Agentes Inteligentes. Sobre su morfología, Wooldridge (2009) afirmó lo siguiente:

> Son sistemas de computación con dos capacidades muy importantes. Primero, son capaces en, al menos, algún grado de extensión de actuar autónomamente – de decidir por sí mismos que necesitan para satisfacer sus objetivos de diseño.
>
> Segundo, son capaces de interactuar con otros agentes – no solamente para intercambiar datos, sino conjuntarse con otros llevando a cabo el tipo de actividad social que implicamos en cada día de nuestra vida: cooperación, coordinación, negociación, etc. (pág. 1)

En esencia, el objetivo de los agentes es emular, dentro de un contexto establecido, actitudes y comportamientos propios de especies vivas, interactuando entre sí con el único fin de lograr el objetivo para el que han sido diseñados.

2.1.2 Sistemas Multi-Agente (SMA)

Mediante la combinación de múltiples tipos de agentes en entornos ideados para su interacción logran los Sistemas Multi-Agente, *SMA* en su versión abreviada, en los cuales cada interacción entre dos o más agentes se toma como un evento puntual dentro de una suma global de ocurrencias, esto es, el comportamiento del sistema en sí.

Para la construcción de un sistema de estas características, existen dos inconvenientes. El primero lo comprenden los agentes que lo integran, mientras que el segundo está compuesto por las interacciones sociales que se pretenden establecer entre ellos. Wooldridge (2009) describe esta dicotomía como la distinción *micro/macro* (pág. 5). No solo es necesario construir agentes capaces de trabajar de forma autónoma (*nivel micro*), sino que el análisis y desarrollo del sistema en su completitud exigen estudiar los comportamientos sociales esperados y prever en cierta medida los resultados de éstos, o en su defecto, acotarlos (*nivel macro*).

Es necesario plantearse una serie de esquemas a la hora de implementar un sistema de estas características. En su desarrollo conceptual sobre los SMA, Wooldridge (2009) destaca las siguientes cuestiones:

- ¿Como emerge la cooperación en una sociedad de agentes independientes?
- ¿Cómo pueden los agentes reconocer comportamientos o esquemas de actuación y coordinarse entre ellos sin generar conflicto?
- ¿Cómo pueden coordinarse para logar objetivos de forma conjunta?
- ¿Qué tipos de lenguajes comunes emplean para comunicarse entre ellos?
- ¿Cómo pueden estar seguros de que, en el momento de compartir información, ésta es interpretada de forma correcta? (págs. 5-6)

Estas cuestiones revelan en primera instancia la complejidad que supone crear SMA's consistentes y compactos. Si no se estudia pormenorizadamente las respuestas para cada una de ellas a la hora de crear el sistema, el desarrollo se puede volver muy complejo o abstracto.

2.1.3 SMA en videojuegos.

Planteándolo en la disciplina que plantea el proyecto, antes de diseñar la IA del juego, hay que verificar que aspectos o acciones que realizase el jugador pueden ser fuente de información para los agentes, qué agentes estarán presentes y que comportamientos individuales pueden desarrollar sin necesidad de coordinarse, qué comportamientos precisan de coordinación y qué proceso han de seguir cada uno, y por supuesto, qué resultados conllevan consigo.

Un aspecto importante que puede ayudar a esclarecer este análisis son las mecánicas, es decir, todas aquellas acciones, procesos o actos que hacen que el estado del contexto actual del juego sea modificado. Por ejemplo, en las fases de un juego en las que el jugador precisa de ser sigiloso, la mecánica principal sería la de evitar hacer ruido, de lo contrario, el estado del juego se ve modificado y la situación de la partida se ve modificada en su contra.

Estudiar y analizar las mecánicas que se quieren establecer puede guiar la configuración de la lista de preguntas anteriormente mencionadas. Continuando con el ejemplo de la mecánica de sigilo, una posible configuración puede ser la siguiente:

- ¿Cómo saben los enemigos, estando a distancia los unos de los otros que hay un jugador en su zona?
- ¿Cuál es el método de actuación en caso de que el jugador active la mecánica de sigilo?
- ¿Los agentes han de cubrir zonas o atacar directamente? ¿En grupo o como individuos?

- ¿Cómo se van a comunicar entre ellos?
- ¿Cómo sabe que han alcanzado al jugador? ¿Qué pasa si este muere?

2.1.4 Inteligencia artificial en los videojuegos

La inteligencia artificial, como campo dentro de la computación, ha sido objeto de una fuerte evolución en los últimos cincuenta años. Su aplicación a las ramas que emanan de éste tampoco ha pasado desapercibida. Concretamente en el ámbito de los videojuegos, la introducción de metodologías, tecnologías y patrones destinados a la creación de Sistemas Inteligentes más complejos ha sido un constante conforme el sector ha ido evolucionando.

Si bien podría decirse que las primeras aproximaciones aprecieron a principios de los años 70, con títulos como Pong (Atari Inc., 1972) o Gotcha (Atari Inc., 1973), lo cierto es que estas empleaban principalmente sistemas de lógica discreta y estructuras preprogramadas de fábrica, por lo que las posibilidades que ofrecían eran más bien reducidas. A principios de los años 80, Pac-Man (Namco Ltd., 1980) introdujo estructuras más complejas dentro de los enemigos, así como patrones específicos para cada uno de ellos.

Con los avances en computación, la mejora y reformulación de los métodos de producción de software y la llegada asociada a éstos de nuevos procedimientos, sumados a la expansión del medio en los años 90, se introdujo progresivamente en uso de herramientas formales en el desarrollo de las IA, como Máquinas de Estados Finitos, algoritmos de búsqueda, heurísticas, etc. (Wikipedia, 2022)

La aparición de nuevos géneros en el medio dio lugar a sistemas inteligentes cada vez más especializados. Un ejemplo de ello es *Civilizations* (MicroProse, 1991), obra de Sid Meier. Este proporcionaba, entro otros, rutinas personalizadas para los diferentes personajes según su función en la partida, árboles tecnológicos que permitían cambiar sus estadísticas o estrategias de interacción. Desde entonces, el avance en la implementación ha contribuido a crear experiencias de juego más realistas, con personajes que ejecutan comportamientos más fluidos dentro de sus entornos.

De todo el abanico de títulos que han sido publicados en los últimos años se pueden destacar los *sandboxes Horizon Zero Dawn* (Guerrilla Games, 2020), y GTAV (Rockstar North, 2015), donde el entorno actúa como un sistema interconectado en el que los múltiples personajes no jugables, como peatones y fuerzas del orden actúan en consecuencia a la las acciones del jugador. En el nicho de un jugador, podemos mencionar a Dark Souls (From Software, 2011), Bloodborne (From Software, 2015) o (From Software, 2019), denominados

Souls-Like por el primero, que proponen combates contra jefes finales que suponen un desafío constante al presentar cada uno mecánicas y comportamientos diferentes.

Figura 1: Enemigos en Left 4 Dead 2

En el género estrategia se pueden destacar las sagas *Hearts Of Iron* y *Europa Universalis*, de Paradox Interactive, en cuyos últimos lanzamientos. HoI4 (Paradox Development Studio, 2016) y EU4 (Paradox Development Studio, 2013), la inteligencia artificial integrada propone estrategias de balanceo en tiempo real cada vez más complejas en lo que a gestión diplomática y militar se refiere, provocando que el jugador deba calcular cuidadosamente qué acciones quiere ejecutar y qué consecuencia podría tener, dando lugar a un amplio abanico de opciones al comienzo de cada partida.

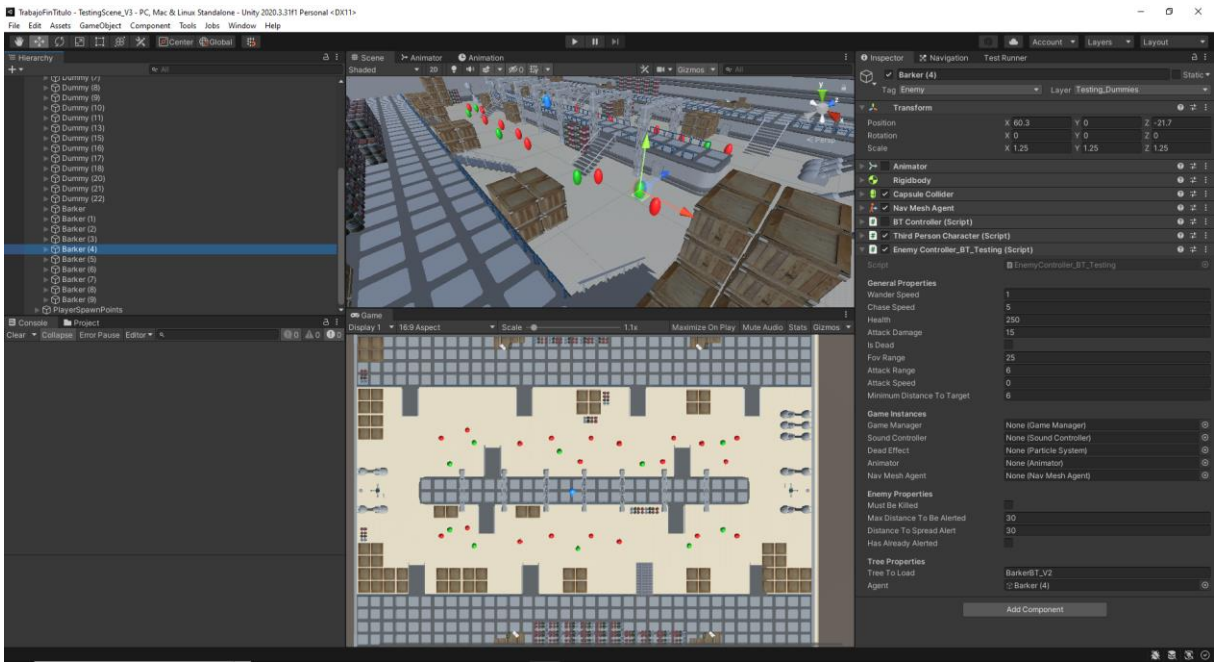
3 HERRAMIENTAS DE DESARROLLO

3.1 Unity3D

Unity 3D es un motor de desarrollo de videojuegos creado por David Helgason, Nicholas Francis y Joachim Ante en 2004. Se orienta principalmente a desarrolladores indie y compañías de medio tamaño, aunque actualmente proporciona licencias de múltiples tipos y precios, por lo que cualquier persona interesada puede hacer uso de el para introducirse en el sector.

Durante las últimas dos décadas ha ido siendo actualizado para adaptarse a las exigencias del mercado, siendo añadidas más características con cada nueva versión publicada. La última versión de soporte a largo plazo publicada es la 2021.3, aunque por motivos de estabilidad y compatibilidad, para el proyecto se ha decidido emplear la versión 2020.3.

Figura 2: Entorno de desarrollo en Unity



3.1.1 Características principales

Otorga compatibilidad con otras tecnologías del sector, como Blender, Adobe Fireworks, Maya o 3DSMax entre las existentes, por lo que la puesta en funcionamiento en equipos que las empleen se realiza de forma transparente. Asimismo, los cambios integrados en los modelos e instancias son implementados de forma automática, sin tener que reimportar. (Wikipedia, 2022).

Como todo motor de videojuegos, incluye las funcionalidades habituales, entre las cuales destacaremos:

- Motor gráfico para renderizar imágenes en 2 y 3 dimensiones, basado en OpenGL.
- Motor de físicas para gestión de las mismas en las escenas.
- Apis propias para la gestión de animaciones, sonidos, datos en partidas, etc.
- Complementos para Inteligencia Artificial.
- Funcionalidades añadidas para Realidad Virtual.

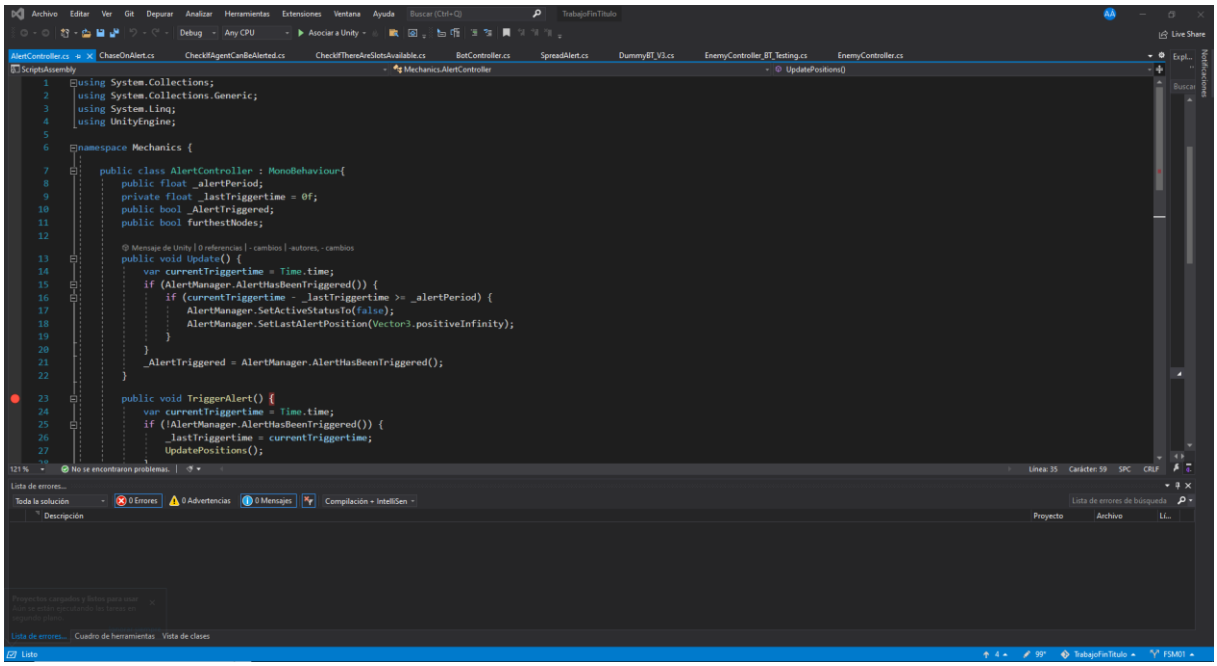
Todas estas funcionalidades vienen incluidas en la mayoría de versiones abiertas al público, siendo el montante obtenido por el juego y lo servicios extra ofrecidos al desarrollador los límites entre unas y otras. Aparte de esto, dispone de una plataforma de *Assets*, complementos desarrollados por la comunidad, tanto que pago como gratuitos, que pueden ser descargados y añadidos al proyecto de forma sencilla. Abarcan desde modelos de personajes simples hasta generadores de terreno o, incluso, componentes de Inteligencia Artificial.

3.2 C# y Visual Studio Community 2019

El lenguaje de programación sobre el que se sustenta el entorno de desarrollo de Unity es C#. Es un lenguaje de alto nivel orientado a objetos y con seguridad de tipos, desarrollado por Microsoft orientado al entorno *.NET*. Este compone un sistema de ejecución virtual denominado Common Language Runtime (Microsoft Corporation, 2022). Permite además el control de versiones.

Para programar en este, Microsoft provee del entorno Visual Studio. VS es un entorno de desarrollo integrado (IDE), que provee de soporte necesario para programar en los lenguajes del entorno .NET, entre ellos C#. Al igual que Unity, provee de múltiples licencias adaptadas a las necesidades del desarrollador. La versión empleada para este proyecto será la edición Community 2019, puesto que es compatible con Unity 2020.3

Figura 3: Entorno de desarrollo en Visual Studio



3.2.1 Características principales

Al tratarse de un entorno integrado de desarrollo, Visual Studio ofrece al programador de una serie de herramientas que le facilitan su tarea, no solo a la hora de verificar el código, sino también para otros procesos, entro los cuales destacan:

- Creación de proyectos y soluciones.
- Integración de librerías propias de Microsoft y externas que amplían las capacidades de la solución.
- Soporte para control de versiones, pudiendo alternar entre ramas si se necesitase.
- Editor de código personalizable.
- Soporte para múltiples Sistemas Operativos.

Provee además una herramienta propia, *IntelliSense*, configurable y que permite disponer de documentación básica acerca de del código que se ha construido, así como sugerencias a la hora de escoger sobrecargas de métodos incluidos en las librerías disponles, etc. (Microsoft Corporation, 2018)

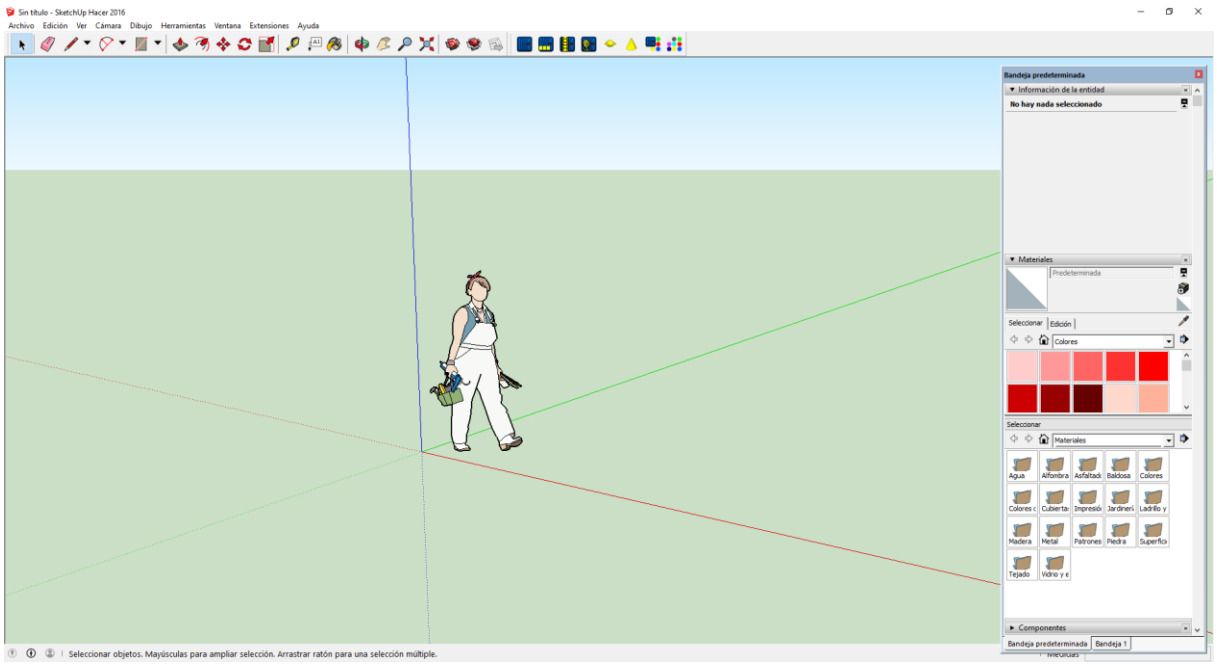
Para la gestión del código en sí, ofrece la herramienta *CodeLens*, que posibilita ser las referencias de los métodos, posibles correcciones sobre éstos y unidades de prueba asociadas con éstos. Otra de las utilidades que ofrece es la generación de grafos basados en las llamadas

de código, otorgando una visualización más clara acerca de la estructura actual del proyecto. (Microsoft Corporation, 2022).

3.3 SketchUp 2016

SketchUp es una herramienta para modelado en tres dimensiones, creado originalmente por Last Software, adquirido por Google, y actualmente en propiedad y desarrollo por Trimble. Se trata de un software profesional para el bocetado, modelado y renderizado básico de estructuras, principalmente orientado al sector de la arquitectura y el urbanismo.

Figura 4: Entorno de desarrollo en SketchUp



3.3.1 Características principales

Provee de herramientas básicas para generar modelos en tres dimensiones de estructuras, asignación de texturas a capas y gestión de geometrías. Se pueden realizar operaciones sobre superficies y volúmenes, tales como extrusión, proyección sobre contornos, traslación de vértices en volúmenes ya creados. Además de estas funcionalidades, permite obtener modelos ya creados por otros usuarios en la plataforma SktechUp Warehouse.

La versión empleada para el desarrollo ha sido la 2016, algo antigua, pero que conserva las capacidades descritas. El ámbito de aplicación se ha enfocado en el bocetado de elementos dentro del proceso de construcción y análisis del nivel de juego.

4 METODOLOGÍAS

4.1 Control de Calidad del Software (Software Quality Assurance)

4.1.1 Marco de aplicación

Al tratarse de un proyecto software de trabajo moderado, se precisó la adopción de metodologías ya vistas y practicadas durante la titulación. Si bien es preciso matizar que hubo que hacer ligeros ajustes, al tratase de un equipo de desarrollo de dos miembros.

Se optó por un enfoque ágil, siguiendo un sistema de trabajo de tipo *pull*, en el cual cada implicado generaba y tomaba una serie de tareas a realizar durante un período de tiempo concreto. Una vez finalizado, se concertaba una reunión para verificar el estado del trabajo realizado, pendiente y para refinar apartados que se considerasen de relevancia para el correcto funcionamiento del equipo en las subsiguientes iteraciones.

4.1.2 SCRUM

SCRUM fue la metodología escogida, principalmente por la familiaridad de los implicados en su uso. La Guía oficial de Scrum lo define como "un marco ligero que ayuda a las personas, equipos y organizaciones a generar valor a través de soluciones adaptables para problemas complejos." (Schwaber & Sutherland, 2020, pág. 3)

Propone una serie de directrices a la hora de organizar un equipo de trabajo, centrándose en el análisis, medida y reajuste continuo de las dinámicas de equipo y en el trabajo realizado, por hacer y en planificación.

El trabajo a realizar se integra en la pila general, o *Product Backlog*, y es extraído a la pila de trabajo de la iteración, o *Sprint Backlog*, que será materializada en un incremento potencial una vez finalizado el período de desarrollo o iteración. El objeto final es lograr llegar a un consenso entre los integrantes del equipo de forma que cada entrega crea valor para el producto final.

Para el desarrollo del juego, se acordó establecer la duración de 3 semanas por sprint, de forma que el tiempo de desarrollo no se alargase demasiado ni fuese lo suficientemente corto para observar cambios notables en el producto final.

4.1.3 Metodología TDD

Para el desarrollo diario se optó por aplicar el Desarrollo Guiado por Pruebas (*Test Driven Development*). Propone una serie de pasos a seguir a la hora de codificar una solución

en los cuales la prueba es el elemento nuclear, debiendo el programador adaptar el código resultante a ésta.

Martin (2009) desarrolla las tres leyes a tener en cuenta cuando se aplica TDD:

- **Primera ley:** No debe crear código de producción hasta que haya creado una prueba de unidad que falle.
- **Segunda ley:** No debe crear más de una prueba de unidad que baste como fallida y no compilar se considerará un fallo.
- **Tercera ley:** No debe crear más código de producción que el necesario para superar la prueba de fallo actual. (pág. 158)

Siguiendo estas directrices, se implementó el código de los diferentes objetos de juego, probando desde los casos unitarios más simples hasta los más complejos, refactorizando el código cuando se considerase necesario una vez que las pruebas hubiesen pasado satisfactoriamente.

4.1.4 Unity Testing Framework

Unity provee al desarrollador de un entorno propio específicamente preparado para la creación de pruebas unitarias y de integración dentro del juego a lo largo de su desarrollo, Unity Testing Framework (UTF).

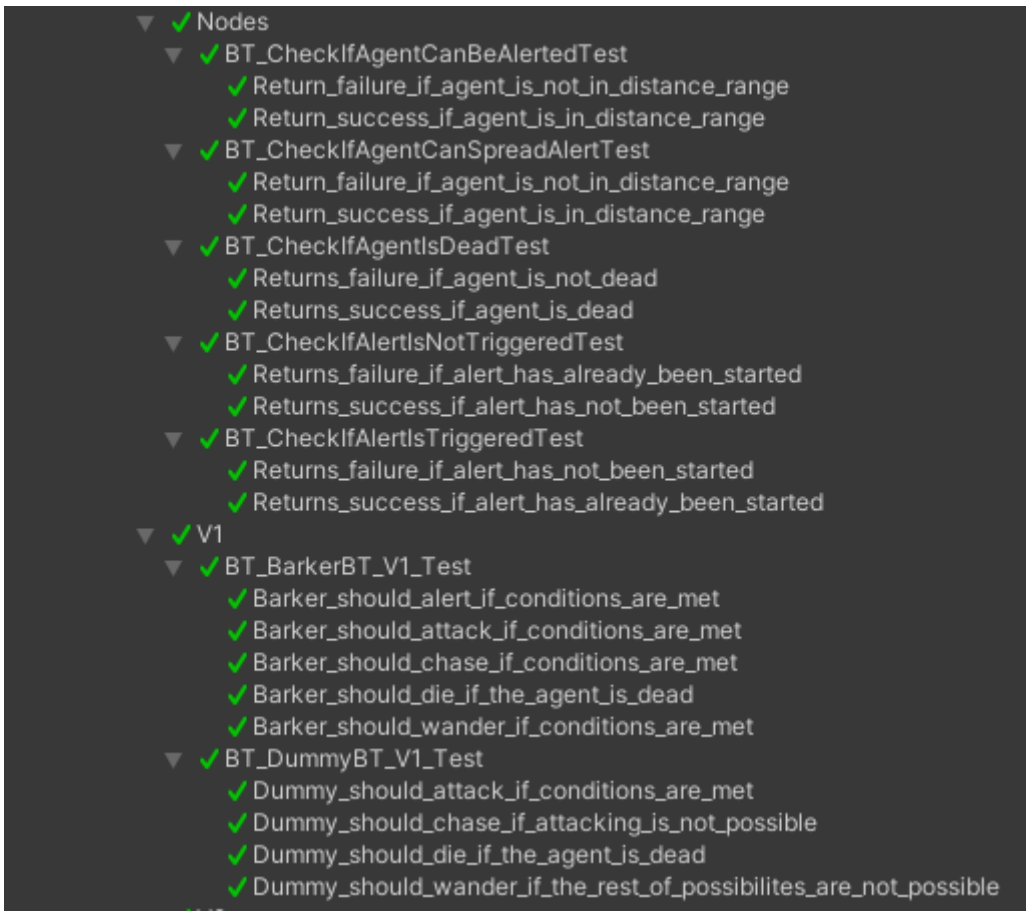
Emplea una integración específica de la librería NUnit (Unity Technologies, 2022). Esta librería de código abierto es la versión para entorno .NET del framework de pruebas xUnit, que permite la creación de subsistemas automatizados de pruebas para proyectos software en lenguajes como F#, C# o Visual Basic (.NET foundation, 2022).

Una de las características más relevantes del UTF es la posibilidad de establecer pruebas en dos modos diferentes, edición y juego. El primero abarca el sistema clásico de pruebas, en el que el código es probado directamente. El segundo permite establecer pruebas dinámicas que se verifican durante el tiempo de juego. Para el desarrollo se optó por testar el código en modo edición, debido a la familiaridad con este tipo de pruebas.

Para el correcto funcionamiento de las pruebas fue necesario reensamblar todo el código en un subproyecto propio, al cual se le hubieron de añadir posteriormente las referencias a las librerías empleadas. Una vez configurado el entorno de pruebas, codificaron y verificaron las correspondientes pruebas para los objetos principales de la escena de juego, como muestra la figura 5.

Figura 5: Test del sistema

Detalle de una parte de batería de pruebas para el proyecto durante el desarrollo.



4.1.5 Control de versiones

Como complemento a la metodología y para facilitar la gestión de cambios, tanto código como pruebas se administraron empleando control de versiones establecido sobre Git, con el correspondiente repositorio alojado en GitHub.

El modelo de creación de ramas tomado fue GitFlow (Driessen, 2010). Se trata de una forma de organizar la creación de ramas basada en una jerarquía preestablecida a nivel de proyecto y en ramas locales a nivel de funcionalidad. Permite llevar un control adecuado a nivel de proyecto sin comprometer la rama de producción.

Los cambios son integrados sobre la rama de desarrollo durante todo el proceso, y se vuelcan sobre la rama de producción únicamente cuando se finaliza la iteración y han sido completamente testeados.

Asimismo, hubo de hacer modificaciones al repositorio, debido a los problemas en compatibilidad con Unity que presenta Git, en especial con los archivos tipo *meta* (metadatos

locales), empelados por Git para hacer las discreciones de cambios entre ficheros modificados o sin modificar.

Por ello, se configuró un fichero tipo *gitignore*, indicando en él los datos que deben ser obviados por el sistema Git cuando se realizan operaciones tipo *commit* en la rama local, del cual se muestra un extracto en la figura 6.

Figura 6: Fichero gitiignore.

Detalle del fichero gitignore del proyecto, configurado para evitar, entre otros, los ficheros tipo meta y los reportes de errores.

```
# This .gitignore file should be placed at the root of your Unity project directory

///

# Unity3D generated meta files
*.pidb.meta
*.pdb.meta
*.mdb.meta
*.meta

# Unity3D generated file on crash reports
sysinfo.txt

# Builds
*.apk
*.aab
*.unitypackage
*.app
```

5 ANALISIS Y DISEÑO PREVIO

Este apartado describe las consideraciones tomadas a la hora de establecer qué clase de juego quería llevarse a cabo y qué partes del mismo se implementarían durante el tiempo previsto para el desarrollo.

5.1 Conceptualización

5.1.1 Selección de la tipología

Se llevaron a cabo diferentes aproximaciones respecto del tipo de juego que se quería abordar, puesto que el formato y condiciones adjuntas a éste condicionarían en gran media la integración de los diferentes componentes dentro del entorno de juego.

Con respecto a la *tipología*, se valoraron los siguientes enfoques:

- Juego de disparos en primera persona, en formato jugador contra jugador (*Player vs Player,* abreviado *PvP*).
- Juego de disparos en primera persona, en formato jugador contra entorno (*Player vs Environment,* abreviado *PvE*).
- Juego de estrategia en tiempo real, con zonas objetivos (*Real Time Strategy, abreviado RTS*).
- Juego de estrategia por turnos, tipo *Risk* que permitiera al jugador desempeñarse contra la IA (*Turn Based Strategy,* abreviado *TBS*).

Tras valorar las diferentes propuestas y considerar ventajas e inconvenientes de cada una, optamos por implementar la segunda opción.

Esta decisión estuvo condicionada a la escalabilidad potencial del proyecto en sí, puesto que un entorno *PvE* permitiría realizar pruebas con diferentes tipos de agentes, y su implementación en una perspectiva de primera persona permitiría integrar, entro otros, componentes de Realidad Virtual y multijugador, pudiéndose además distribuir estas acciones de forma paralela, facilitando con ello el proceso de desarrollo para ambos estudiantes.

5.1.2 Estudio de los objetivos del jugador

Una vez decidido el tipo de juego, se analizaron los objetivos primarios y secundarios del jugador como elemento de juego.

Para este punto, las opciones consideradas fueron:

- Sobrevivir a hordas de enemigos durante un tiempo preestablecidos.

- Sobrevivir hordas de enemigos preestablecidas generadas a lo largo de diferentes rondas.
- Alcanzar un punto concreto de la escena de juego sorteando enemigos.

Finalmente se optó por la segunda opción, debido a su facilidad para ser implementada y a que permite la reutilización de los modelos de enemigos base, reajustando sus características al comienzo de cada ronda.

5.1.3 Condiciones de victoria y derrota

Al tratarse de un juego de supervivencia por hordas, la condición de victoria base es sobrevivir al mayor número posible de rondas. El juego tiene una duración *infinita* a priori. Después de cada ronde las características de los agentes son modificadas antes de aparecer estos en la escena de juego. Entre éstas, la cantidad de puntos de salud y el daño que realizan aumentan, de forma que resulte más complejo eliminar a un enemigo conforme el jugador va sobreviviendo más rondas.

De forma análoga, la condición de derrota base es la muerte del propio jugador. Una vez los puntos de salud de éste disminuyen a cero, el jugador *muere* y finaliza la partida. Llegado a este punto, el jugador puede reiniciar la partida, comenzando nuevamente desde la primera ronda.

A pesar de considerarlo plausible, se desestimó el incluir la posibilidad de guardar el progreso durante la partida, por la propia idiosincrasia del juego. El objeto de victoria es, pues, que el jugador sobreviva al mayor número posible de rondas.

5.2 Mecánicas del jugador

Para lograr una experiencia de juego más inmersiva y ampliar el catálogo de opciones del jugador a la hora de afrontar cada ronda. Se estimaron las siguientes mecánicas:

- El jugador ha de disponer de un arma principal con la que iniciará cada ronda.
- De forma alternativa, el jugador dispondrá de un arma secundaria. Las características de ésta serán más reducidas en comparación con el arma principal.
- El jugador podrá recuperar sus estadísticas de salud en puntos definidos en el mapa. Estas zonas de salud se activarán si el jugador pulsa una tecla predefinida.

- El jugador podrá recuperar la munición de todo su arsenal en puntos definidos en el mapa. En dichos puntos, deberá pulsa una tecla predefinida.
- El jugador podrá desplazarse frontal y lateralmente sobre los ejes vectoriales correspondientes. Se permite además la acción de salto, pudiéndose realizar una vez.

5.3 Mecánicas de los agentes

Siguiendo con el punto previo, se plantean las siguientes funcionalidades base para cada uno de los agentes presentes.

- El agente podrá desplazarse frontal y lateralmente sobre los ejes vectoriales correspondientes.
- El agente podrá reconocer al jugador y tomar las acciones previstas
- El agente podrá finalizar su comportamiento si el jugador muerte a lo largo del transcurso de la partida.
- Según el nivel de dificultad, el agente podrá coordinarse con otros para atacar al jugador por diferentes rutas de aproximación.

5.3.1 Modelos de agentes planteados

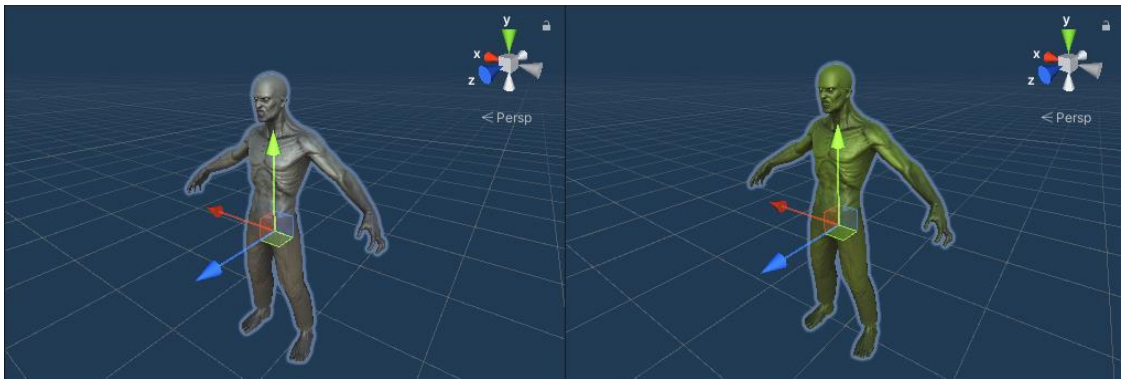
Para dotar al juego de una variedad aceptable de enemigos, se plantean los siguientes modelos de agentes, con sus respectivos comportamientos.

- Dummy (Simplón): es el enemigo más débil de los tres. Si el jugador aparece cerca de él, lo perseguirá para atacar a menos que el jugador consiga escapar a tiempo. Su rango de ataque es corto.
- Barker (Aullador): de cualidades promedio, no perseguirá al jugador. Sin embargo, si se encuentra lo suficientemente cerca, gritará alertando a otros de su presencia. Si otros enemigos de diferente clase están presentes, también se verán alertados y ejecutarán sus acciones pertinentes.

La figura 7 muestra el modelo empleado para ambos, con sus texturas. Es importante destacar que estos modelos son conceptuales y forman parte de la primera versión presentada para este proyecto.

Figura 7: Vistas de modelos de zombis

A la izquierda, el zombi normal. En verde, el zombi aullador.



5.4 Análisis y diseño del nivel

Con respecto al diseño del mapa se tuvieron presentes tanto las mecánicas a integrar como la sensación de dificultad resultante. El objeto de estas consideraciones no era otro que evitar generar frustración en el jugador durante la partida.

Se tomaron referencias de niveles ya diseñados en otros títulos que incluyen modos de supervivencia de hordas como como *Left 4 Dead 2* (Valve Corporation, 2009) y *Team Fortress 2* (Valve Corporation, 2007), así como otros que, por su depurado diseño de niveles, se consideraron interesantes de cara a al estudio previo, como *Portal 2* (Valve Corporation, 2011), *Call of Duty: Modern Warfare 2* (Infinity Ward, 2009) y DOOM Eternal (Bethesda Softworks, 2020)

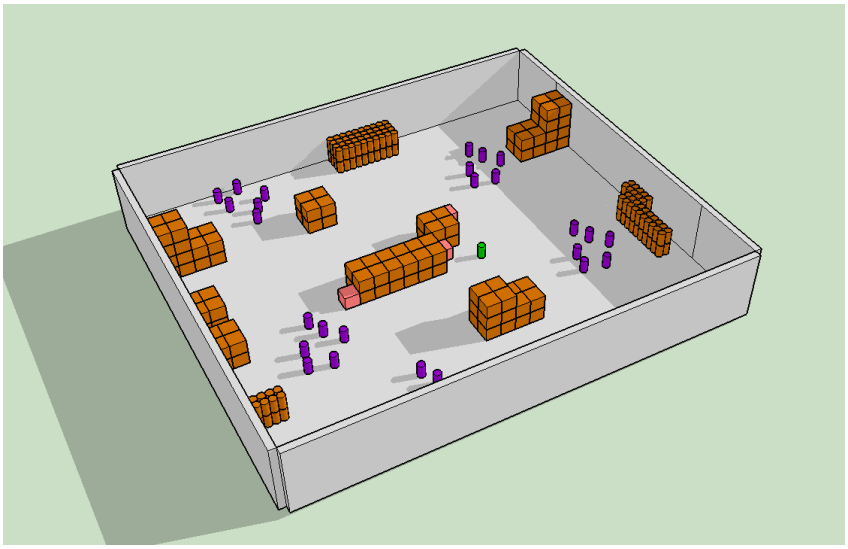
Por ello se concretaron los siguientes puntos a la hora de llevar a cabo el diseño del nivel.

- Se ha de favorecer el movimiento, esto es, la experiencia de juego debe ser lo más fluida posible, pudiendo el jugador adoptar diferentes estrategias a lo largo del desarrollo de cada ronda.
- Es preferible un mapa con un número reducido de rutas de ataque a uno con múltiples caminos pero que puedan llegar a inducir confusión o sobrecarga de información en el jugador.
- El espacio navegable debe aportar información para su comprensión. Se ha de evitar en la media de lo posible que existan partes similares en el mapa. Los propios elementos del mapa, o *props*, deben ayudar al jugador a ubicarse espacialmente.

- El propio diseño del mapa debe ser reconocible por el jugador una vez haya experimentado con el juego durante un número reducido de rondas.
- Se han de ofrecer diferentes alternativas de altura. El jugador debe poder generar estrategias que permitan explotar la ventaja obtenida de su posición en diferentes alturas.
- Los enemigos deben aparecer en zonas que les permitan cubrir todo el mapa, siempre y cuando no lo hagan en aquellas en las que esté presente el jugador.

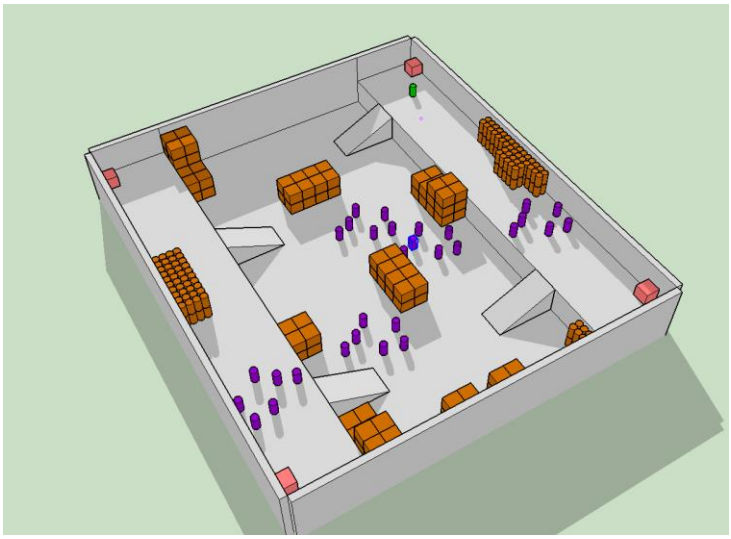
Con estos puntos concretados se llegó a un primer esbozo, que se muestra en la figura8, en el que toma la forma de una sala. Esta primera versión pasó por un proceso de refinamiento para mejorar algunos aspectos de jugabilidad.

Figura 8: Maqueta de la primera versión realizado en Sketchup



La segunda propuesta ya se acercaba a lo que se pretendía alcanzar en este apartado, ofreciendo dos zonas y múltiples obstáculos. La figura 9 muestra los cambios realizados al diseño original después de la iteración, con dos zonas a diferente altura y una zona central donde se presume se desarrolle la acción principal.

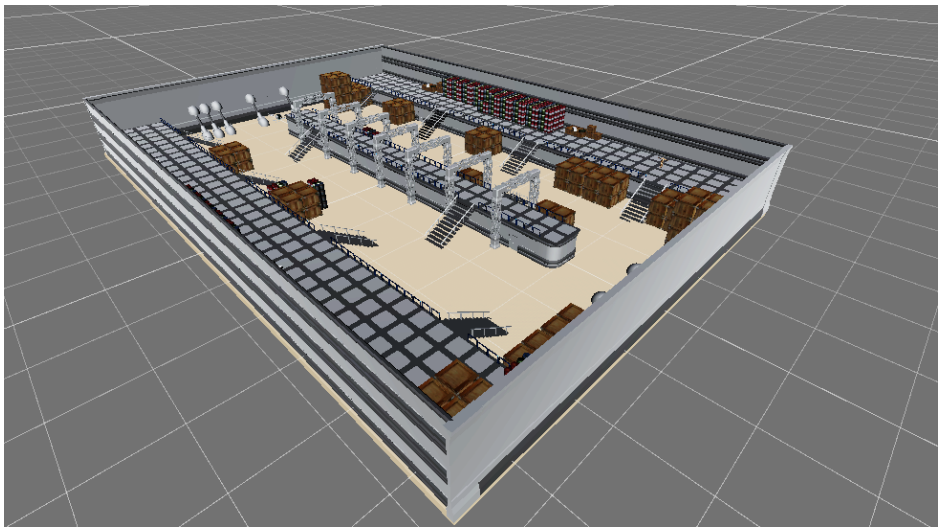
Figura 9: Boceto de la segunda versión realizado en Sketchup



Finalmente, la tercera y definitiva versión del mapa ya incorpora las modificaciones y ajustes necesarios para poder integrar el resto de elementos secundarios. Se puede observar en la figura 10, que en esta versión el mapa queda dividido en tres zonas a diferente altura.

Dos se encuentran ubicadas en los laterales, abarcando todo el espacio longitudinal del espacio jugable. La tercera consiste en una plataforma central, accesible a través de rampas, que cubre parcialmente el área central, ubicua en la segunda versión.

Figura 10: Modelo en Unity de la tercera versión del mapa



Empleando esta distribución se establecen cuatro zonas de aparición para el jugador y los agentes. De forma análoga se distribuye el mismo número de zonas de recarga, ubicadas en los corredores elevados, una por cada cuadrante del mapa.

6 PRIMERA APROXIMACIÓN: MAQUINAS DE ESTADO

Este apartado tratará sobre la conceptualización, análisis e implementación de la primera parte del desarrollo de la IA, que se corresponde con las máquinas de estados finitos.

6.1 La Máquina de Estado Finitos

6.1.1 Definición

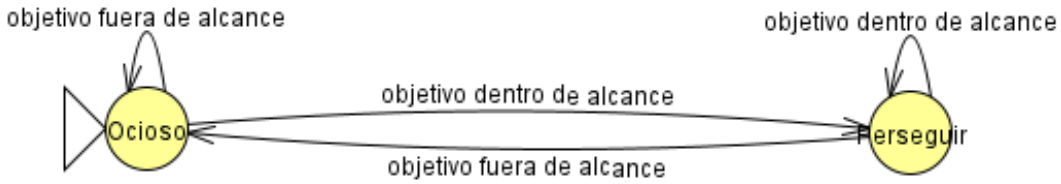
En primera instancia es preciso definir el concepto de Máquina de Estado Finitos (*Finite State Machine*, en inglés), pues proporcionará un enfoque mucho más específico a posteriori.

En el marco de la computación, es una estructura compuesta por un número finito de estados conectados en un grafo por transiciones entre ellos. Una entidad establece su comportamiento en un estado inicial y luego verifica eventos y reglas que puedan disparar una transición hacia otro estado. Dicha Entidad solo puede estar en un único estado en un momento dado. (Aversa, Kyaw, & Peters, 2018, pág. 9)

En la figura 11, podemos observar la distribución de una Máquina de Estados Finitos compuesta por dos estados, ocioso y persecución. Su entidad asociada verificará en un momento de tiempo dado las condiciones que le permitan tomar la transición más apropiada y modificará su estado si estas lo permiten.

Figura 11: Ejemplo de Máquina de Estado Finitos

Máquina de Estado Finitos bi-estado para un sistema de persecución. El estado inicial propuesto es "Ocioso" (idle).



Si las condiciones del entorno le son propicias, es decir, si el objetivo se encontrase dentro del rango del alcance preestablecido, la máquina cambiará de estado de *Ocioso* a *Perseguir*. Ejecutará las acciones contenidas en dicho estado mientras las condiciones que lo permiten se sigan verificando.

Por el contrario, si el objetivo pasase a estar fuera del rango de alcance preestablecido, el esquema de transiciones disponible provocaría que la entidad asociada pasase a estado *Ocioso*, repitiendo el mismo esquema de comportamiento ya descrito.

Esta aproximación teórica permite implementar esquemas de comportamiento más complejos sin mucha dificultad, exceptuando aquella que conlleven el análisis y verificación transicional previos.

Como beneficio colateral, al tratarse de un sistema cerrado, permite evitar posibles problemas asociados, como pueden ser las tomas de decisiones no controladas o infinitas, que dan lugar a problemas de resolución de tipo No Polinómico.

6.1.2 Aplicación al desarrollo de videojuegos

Como hemos visto, la implementación de una máquina de estados, aunque simple, requiere de un análisis previo, en el cual han de estar incluidos:

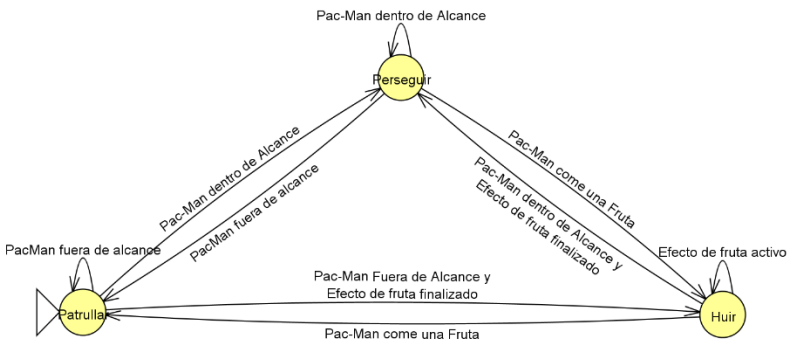
- Estados o puntos de comportamiento que tomará el agente.
- Mapa de transiciones entre estados.
- Condiciones verificables que permitan la toma de transiciones.
- Sensores que provean a la entidad de información sobre el entorno, que será empleada para la verificación.

Es importante hacer hincapié en el último punto, pues a diferencia de un entorno únicamente tangible, la información del entorno en un videojuego puede ser provista por dos fuentes, el controlador físico (usuario) o el controlador virtual (motor de juego).

La figura 12 muestra la MEF de un enemigo del videojuego Pac-Man, publicado por Namco en 1980. Se trata de un esquema de tres estados, en el cual la entidad puede perseguir, patrullar o huir del jugador en función a las condiciones del entorno.

Figura 12: Máquina de Estados del Pac-Man

Máquina de Estado Finitos tri-estado de un enemigo del videojuego Pac-Man.



Nota: Adaptado de (Cossu, 2021, pág. 118)

A pesar de ser una concepción simple en primera instancia, este esquema de comportamientos se ajustó para que las entidades que lo implementaban, pudieran dar la sensación de coordinarse.

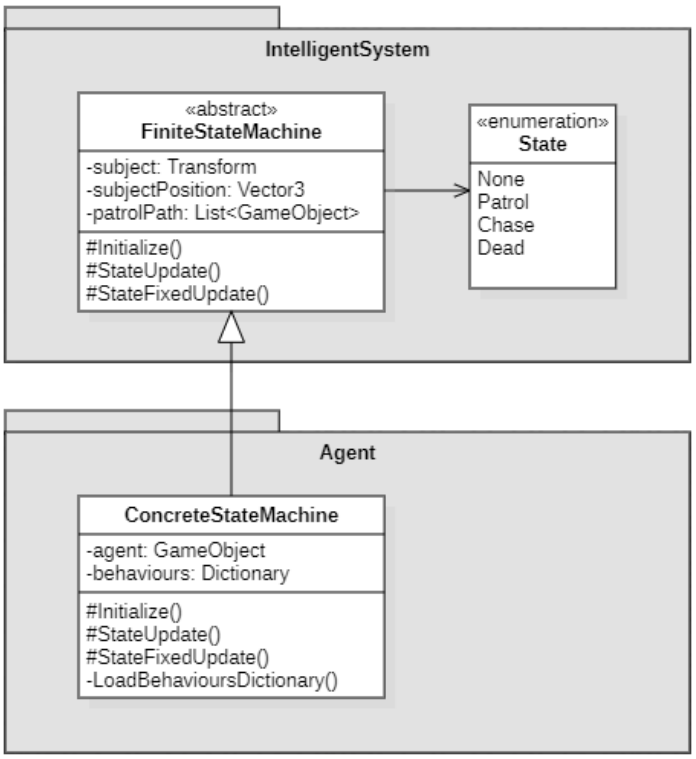
6.1.3 Planteamiento inicial

Dado que el objetivo de este proyecto es el desarrollo de sistemas inteligentes, en su primera versión implementaremos una MEF simple, la cual se irá expandiendo e integrando en un sistema más complejo conforme prosigan las iteraciones. Para facilitar la integración de las instancias que la implementen, la constituiremos como clase abstracta, de forma que todas las instancias que se extiendan de ella hereden los métodos asociados.

Como se puede observar en la figura 13, el modelo de implementación planteado consta de una clase abstracta *FiniteStateMachine,* con las funciones de la máquina, y su implementación concreta *ConcreteStateMachine,* que adoptarán los agentes inteligentes.

Figura 13: Clases de la Máquina de Estados

Diagrama de clases de la Máquina de estados.



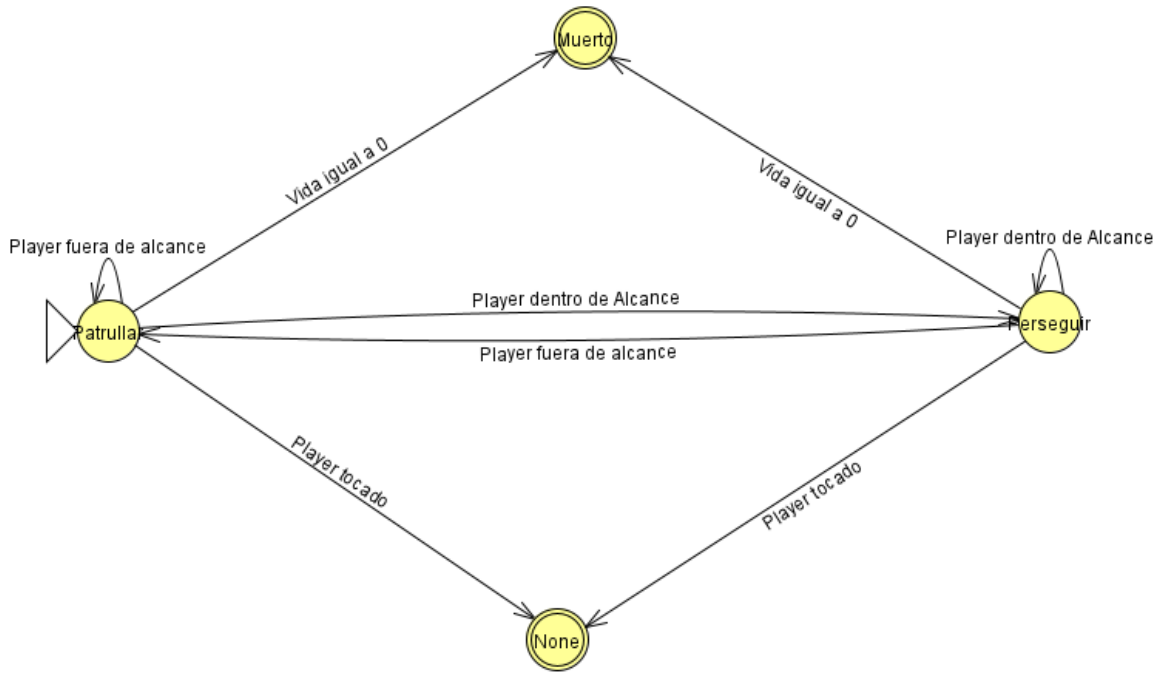
De esta forma disponemos de una estructura común, escalable y de sencilla implementación en caso de que el juego pueda tener varios agentes con patrones de comportamientos diferentes.

Para esta primera versión estableceremos una máquina de estados de cuatro estados muy parecida a la ya creada para el juego Pac-Man: *None, Perseguir, Patrullar y Muerto*. El objetivo del agente deberá ser perseguir al jugador hasta tocarlo. El jugador humano tiene como objetivo eliminar a los agentes sin ser tocado por éstos, perdiendo la partida si no lo logra.

Definimos en primera instancia los estados y transiciones de nuestra máquina en un grafo, como se muestra en la figura 14.

Figura 14: MEF del primer prototipo.

Máquina de Estado Finitos tetra-estado de un enemigo de nuestro primer prototipo.



Cada agente tomará como viales de entrada los vectores de posición y sus respectivos cálculos de distancia. El motor Unity proporciona estos cálculos mediante la llamada a la clase interna *Vector3*, que puede instanciada en cada momento de la ejecución de la escena de juego.

Se plantean pues las condiciones necesarias para que el agente pueda tomar decisiones en función al estado de la escena de juego en cada instante de tiempo. Se detallan más en profundidad en la tabla 1.

Tabla 1: Transiciones en la MEF inicial

Transiciones en la máquina de estados propuesta.

Estado Inicial	Condición/es de disparo	Estado Final
Patrullar	Distancia(Jugador) > Valor	**Patrullar**
Patrullar	Distancia(Jugador) <= Valor	**Perseguir**
Patrullar	Player Tocado por si o por otro agente	**None***
Patrullar	Salud <= 0	**Muerto***
Perseguir	Distancia(Jugador) <= Valor	**Perseguir**
Perseguir	Distancia(Jugador) > Valor	**Patrullar**
Perseguir	Player Tocado por si o por otro agente	**None***
Perseguir	Salud <= 0	**Muerto***

Nota: Los estados finales, sombreados en gris, representan el fin del flujo normal del comportamiento.

Al tratarse de un circuito cerrado en el que las transiciones dependen de variables contextualmente independientes, se canaliza de forma simple la adopción de decisiones y se establece, a su vez, un orden jerárquico en éstas. Si el agente muere, la instancia es eliminada de la escena. Si, por el contrario, el jugador es tocado por el propio agente u otro de los presentes, pasará a estado de inactividad (*None*). Ambos casos dan como resultado el final del flujo normal del comportamiento, bien sea con la eliminación o con su desactivación tras la victoria de la IA.

6.1.4 Implementación en el prototipo

Habiendo realizado el diseño, creamos nuestro primer elemento prefabricado, o *Prefab*. En este caso se trata de un tanque compuesto por dos elementos, el chasis y la torreta. Ambos podrán moverse de forma independiente en cada estado.

Asimismo, tendrá asociado un script que implementa la máquina de estados concreta para su instancia, *StateMachine.* Esta clase hereda los métodos de actualización de estado e implementa a su vez los comportamientos asociados a cada uno en sus respectivas funciones. Como ya se ha comentado, éstas serán ejecutadas por requerimiento en la ejecución del método *StateUpdate()*, heredado de la clase padre.

Es decir, durante la ejecución de la escena, el agente partirá desde el estado en que se encuentra, verificará los datos que dispone, cambiará de estado si es necesario y disparará las acciones asociadas a dicho estado, tal y como hemos descrito en puntos anteriores. Este flujo se repetirá hasta que el agente alcance alguno de los dos estados finales.

6.2 Refactorizaciones posteriores

Ya hemos comentado que uno de los aspectos más relevantes del desarrollo de cualquier proyecto de software es la optimización de recursos y, muy especialmente, del código que lo integra.

En palabras de Robert C. Martin, "no basta con escribir código correctamente. El código debe limpiarse con el tiempo. Todos hemos visto que el código se corrompe con el tiempo, de modo que debemos adoptar un papel activo para evitarlo" (Martin, 2009, pág. 41)

6.2.1 El patrón State

En nuestra primera iteración de la máquina de estados hemos optado por incluir la ejecución de las acciones asociadas como una función propia dentro de cada implementación de la clase *ConcreteStateMachine*.

A futuro esta estructura generará redundancias y problemas en el mantenimiento del código, pues por cada implementación que se añada, habrán de reescribirse los métodos correspondientes a los estados. Asimismo, si se plantea la existencia de estados comunes, como el de inactividad (*None*), existirán n repeticiones de dicho estado, donde n es el número de máquinas que lo incluyen entre sus posibles salidas.

Para solventar este problema plantearemos la introducción del patrón de diseño *State* como primer paso a la hora de optimizar el código ya presente. Para ello es preciso extraer las funciones que encapsulan los estados a clases independientes. Para este proyecto se ha optado por incluirlas dentro de su propio espacio de nombres dentro del contexto general.

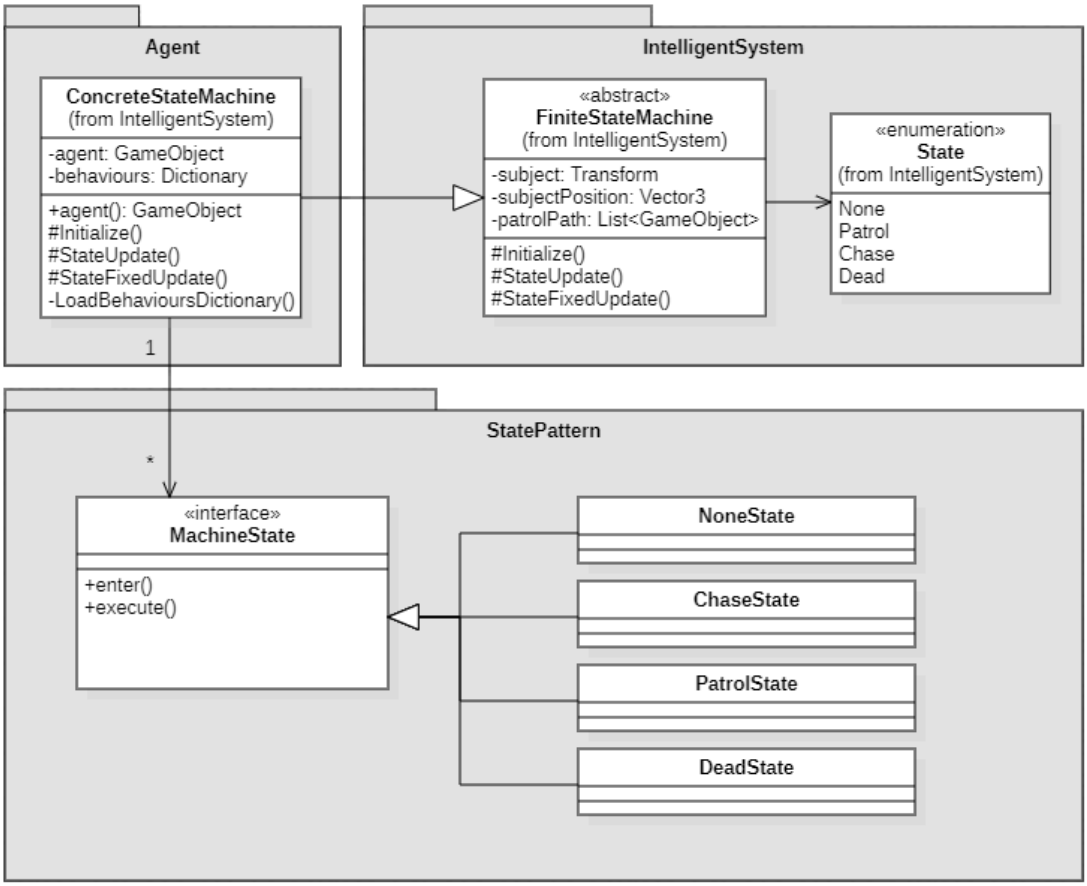
De la misma manera, todas implementarán la interfaz *MachineState*, que contiene los métodos *enter* y *execute*. El primero especifica la entrada en el estado, así como las condiciones en las que este se presentará, mientras que el segundo ejecutará las acciones que corresponda.

Tomando como referencia el diagrama de clases de la primera versión, añadimos la nueva jerarquía al modelo, como se detalla en la figura 15. De esta manera, establecemos a partir de esta iteración la independencia entre el contexto de los estados y las máquinas que los toman, lo que nos permitirá reutilizar dichos estados en el futuro si fuese necesario. Nystrom

(2014) resume esta explicación afirmando que el encapsulamiento de los estados permite que las acciones se ejecuten sin necesidad de saber el estado de procedencia.

Figura 15: Implementación del patrón State

Refactorización del modelo de clases de la primera versión con el patrón State.



De forma paralela, con este cambio hemos conseguido que la infraestructura de las máquinas concretas se simplifique aún más, pues el mapa de comportamientos ya no estará definido por la pareja <*State, Action*>, sino que pasará a ser <*State, Machine State*>. De forma análoga, la llamada en el método de actualización será reescrita como *behaviours[currentState].enter().*

Otra ventaja añadida es la posibilidad de establecer una jerarquía dentro del propio paquete StatePattern, pudiendo introducir paquetes internos que permitan organizar todas las clases derivadas presentes en el proyecto.

6.3 Navegación por el entorno de juego

Las máquinas de estado finito pueden ser útiles para controlar determinados circuitos cerrados, al coste de estar limitadas únicamente al conjunto de estados que se han definido previamente para el agente. Se hace patente en el estado de persecución de nuestro primer prototipo, pues el agente simplemente se traslada hacia la posición del jugador hasta tocarlo.

Sin embargo, en un entorno donde pueda haber obstáculos este sistema no es aplicable, ya que el agente no tiene información del medio que hay a su alrededor ni que vías puede tomar en caso de que el jugador cambie de ubicación.

Para lograr esto último, es preciso aplicar una capa más de complejidad al agente, dotándolo de herramientas que le permitan tener, hasta cierto punto, control sobre donde se encuentra, cuál son los posibles caminos hacia el jugador y cuál de éstos es la mejor opción a escoger en función de su posición dado un momento de tiempo concreto.

6.3.1 Proceso de Búsqueda como resolución algorítmica

En el plano teórico, el comportamiento de la inteligencia artificial, puede ser reducido a un nivel conceptual una vez se establecen las condiciones de base para que ésta cumpla su objetivo.

Hay que destacar que, a pesar que dicho objetivo puede variar en función a número y tipo de instancias inteligentes que se hayan planteado implementar en el entorno accesible al jugador, todos compartirán el mismo concepto común: su comportamiento debe asumirse como resolución de un problema planteado en un estado E como conjunto de condiciones C en un momento de tiempo T, sobre el cual se aplicará un conjunto de decisiones:

$$R \rightarrow \mathrm{D}^*(\mathrm{ECT})$$

Es posible plantear la definición previa dentro del entorno de una escena de la siguiente forma: Dado un elemento inteligente con un objetivo preestablecido, éste deberá completar dicho objetivo basándose en un conjunto de reglas preprogramadas para resolverlo bajo unos parámetros de coste y rendimiento asumibles.

Hay que hacer hincapié sobre todo en estos dos últimos parámetros, coste y rendimiento. Esto se justifica por el hecho de que a nivel computacional se llevan a cabo múltiples subprocesos asociados que limitan la capacidad de cómputo disponible durante la ejecución de la instancia de juego.

6.3.2 *Estrategia de exploración*

Para resolver de forma efectiva el problema preestablecido, es preciso dotar al elemento inteligente de una estrategia que le permita acotar el dominio de posibles resoluciones que puede adoptar.

Aplicando esta dinámica al entorno de una escena, podemos determinar que dicha estrategia deberá poder tomar información existente en el medio. Una vez procesada dicha información, la decisión escogida deberá ajustarse nuevamente al espacio posible definido, para a continuación repetir el proceso hasta la resolución del problema planteado.

6.3.3 Aproximación computacional

En esencia, lo que se plantea acoplar al agente es la capacidad de resolver un problema de búsqueda informada desde su posición hacia un objetivo no estático (el jugador humano) a través de un medio predefinido (el nivel de juego).

La resolución de este tipo de tareas requiere de alguna fuente que les permita ajustar el rango de soluciones viables. Dicha solución es la introducción del componente heurístico, que se puede describir como "criterios, métodos o principios para decidir cuál de entre diversos cursos de acción alternativos 'promete' ser el más efectivo para alcanzar algún objetivo" (Hernández, 2021)

Es importante reseñar el concepto de promesa, pues siempre existe la posibilidad de que el problema no se pueda resolver dada una situación de juego concreta, como, por ejemplo, el hecho de que el jugador se encuentre en una localización no alcanzable por el agente.

Asimismo, es preciso emplear un algoritmo que permita la admisión del componente heurístico y que a su vez ofrezca un rendimiento óptimo en la toma de decisiones.

En el caso del motor Unity, el algoritmo predefinido para iterar las mallas de navegación es el A*, o *A estrella*. Como todo algoritmo de búsqueda en amplitud, es un algoritmo completo: en caso de existir una solución, siempre dará con ella. (Algoritmo de búsqueda A*, 2022). Esto implica que los agentes encontrarán la solución adecuada en tiempo óptimo. La figura 16 muestra un breve resumen del algoritmo.

Figura 16: Alagoritmo A*

*Detalle del procedimiento lógico del algoritmo A**

1. Introducir en ABIERTA una trayectoria inicial que contenga solamente el nodo raíz. Inicializar a vacía la lista CERRADA
2. Hasta que lista esté vacía o se encuentre el objetivo, examinar la primera trayectoria de la lista
   a. Si el primer nodo es el Objetivo, entonces salir del bucle
   b. Si en primer nodo no el Objetivo, entonces:
      - Eliminar la primera trayectoria de la lista ABIERTA, incluyéndola en la lista cerrada en el caso de que sea de coste más óptimo que una similar ya contenida, eliminando en su caso la antigua trayectoria contenida en CERRADA
      - Formar nuevas trayectorias a partir de la trayectoria eliminada de ABIERTA, ramificando el último nodo de la misma
      - Añadir las nuevas trayectorias a la lista ABIERTA, si existen
      - Ordenar la lista ABIERTA en base al costo acumulado de cada una, colocando la de mejor valor de función evaluación al inicio de la lista
      - Si dos o más trayectorias de ABIERTA acaban en un nodo común, borrar las mismas excepto la que posee mínimo valor de función de evaluación de entre ellas. Eliminar esta última también si existe una similar con menor valor de función de evaluación en la lista CERRADA. Al eliminar trayectorias de ABIERTA deben insertarse en CERRADA salvo que ya exista allí una similar de menor coste
3. Si se ha encontrado el objetivo, finaliza con ÉXITO, y *la solución es la primera trayectoria en la lista*
4. Si no, *Problema sin Solución*

Nota: adaptado de (Hernández, 2021, pág. 66)

6.4 Navegación en Unity

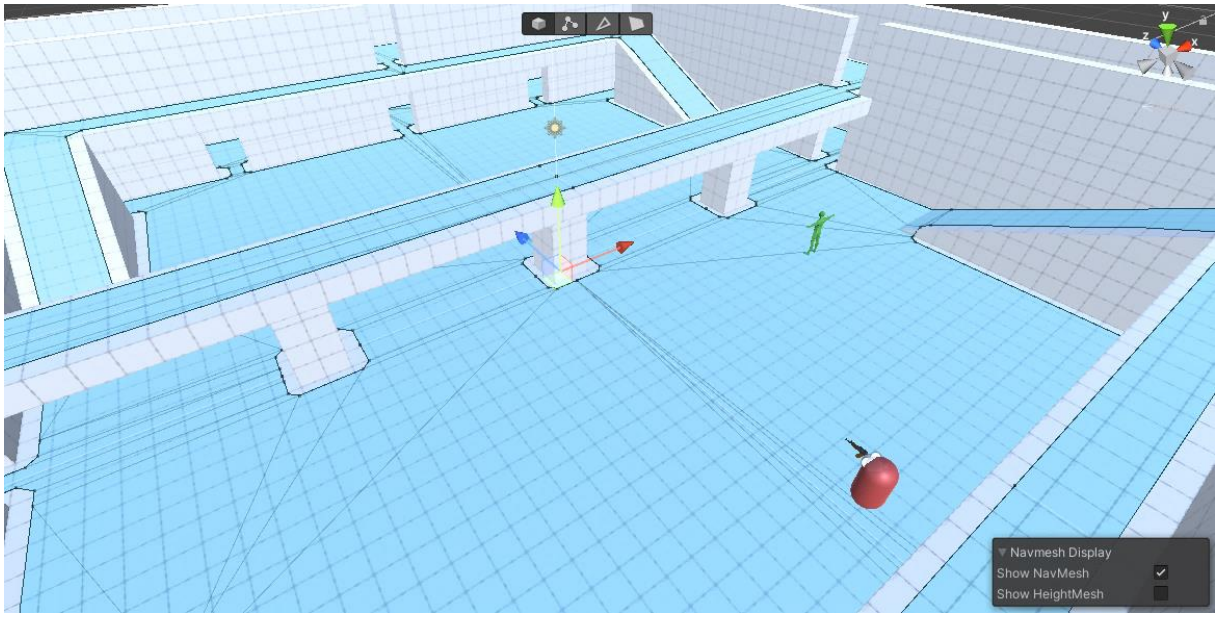
6.4.1 Mallas de Navegación en Unity

De acuerdo con la documentación oficial de Unity, se define una malla de navegación como “una estructura de datos que describe las superficies navegables del mundo del juego y permite encontrar el camino de una ubicación caminable a otra en el mundo del juego. La estructura de datos es construida de manera automática de la geometría de su nivel.” (Unity Technologies, 2022).

En esencia, el motor provee al programador de una herramienta de análisis de la geometría existente en el nivel. Una vez que se ejecuta el análisis, se genera un conjunto de polígonos, que son almacenados como entidad de malla de navegación. La figura 17 muestra una captura de una malla ya generada, en color azul, sobre el mapa del prototipo.

Figura 17: Malla de navegación en prototipo

Ejemplo de malla de navegación de una de los prototipos base. Se observan las líneas divisorias de los diferentes polígonos que forman el grafo (en color azul).



Dicho conjunto es explorado como un grafo empleando el algoritmo A*, es decir, llevando a cabo siempre la opción más optima disponible en el instante de tiempo en el que la función de exploración es llamada. En esencia, los objetos de juegos que tienen integrada la capacidad de navegar a través de mallas se acoplarán a la primera disponible, iniciando exploraciones para llegar al punto vectorial designado como objetivo.

6.4.2 Gestion de obstáculos

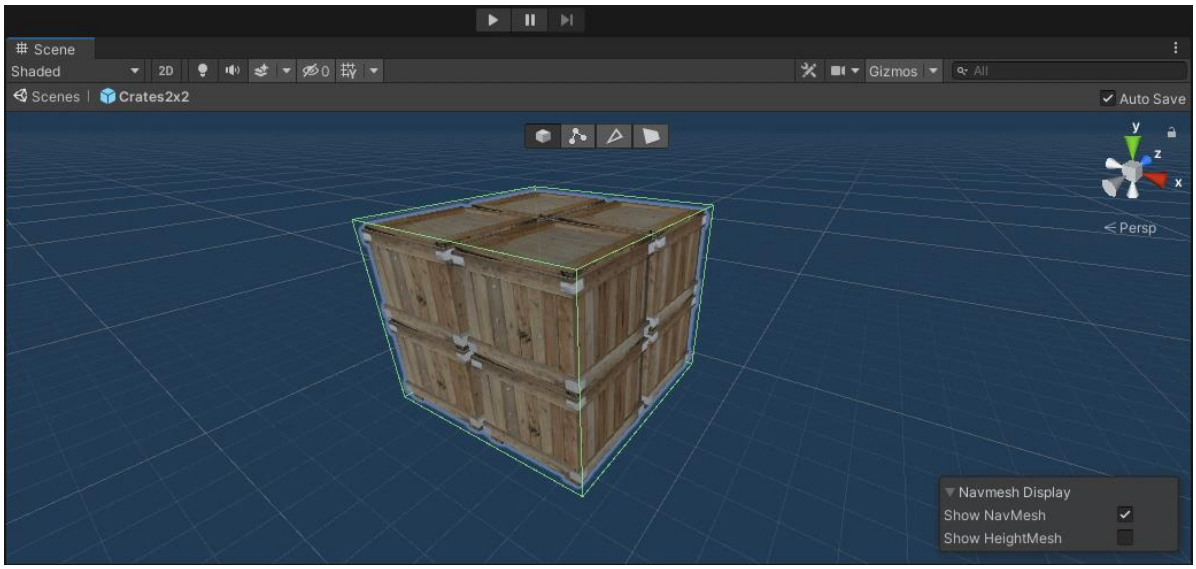
A nivel general, se permite la exclusión de elementos del proceso de construcción mediante el empleo del sistema de capas interno, pudiéndose configurar de forma personalizada las diferentes secciones del mapa que se van a considerar para generar la malla resultante. Además de esto, el generador debe poder discernir si existen o no objetos dentro de dichas capas que puedan representarse como no transitables, esto es, objetos volumétricos dentro del mapa. Para explicar este punto, hay que hacer mención a los colisionadores o *Colliders*.

Un colisionador es un componente asociado a un objeto de juego que permite delimitar el espacio físico ocupado por el objeto dentro del entorno y su interacción física con el resto de elementos. La documentación oficial de Unity relativa a estos resume su función en que “definen la forma de un objeto para los propósitos de colisiones físicas” (Unity Technologies, 2022) Estos se añaden a los objetos, pudiendo tomar formas primitivas, como un cubo, una

esfera o una cápsula, o bien otras más complejas, adaptadas a la textura o forma del modelo que las integra. Como se observa en la figura 18, la inclusión del colisionador debe ajustarse en la media de lo posible a la forma del objeto.

Figura 18: Colisionador en un componente

Detalle un componente de colisión en Unity. En este caso, el colisionador toma la forma de un cubo y ha sido ajustado al tamaño del objeto que encapsula.



Si bien las propiedades físicas de los objetos encapsulados por colisionadores son irrelevantes cuando se trata de navegabilidad, el delimitar su espacio geométrico es imprescindible para que sean excluidos del análisis. A nivel interno, una vez se han analizado las capas correspondientes, los colisionadores asociados actuarán de límites dentro de este contexto para verificar hasta donde mapear el área de navegación resultante.

6.4.3 Gestion de alturas

De forma análoga a la exclusión de obstáculos, la gestión de alturas es otro de los puntos que se tienen en cuenta a la hora de construir la malla. Por ejemplo, a la hora de implementar mecánicas en las que los agentes pueden realizar acciones de salto entre zonas con diferentes alturas en su trayectoria hacia el jugador, la malla de navegación construida debe proveer de datos referencia a la hora de poder calcular qué zonas son accesibles y cuáles no.

Por ello, durante el proceso de construcción se puede tener en cuenta la inclusión de las mallas de alturas, o *HeightMeshes*, que son esencialmente mallas de navegación limitadas únicamente a las superficies navegables que se encuentran a diferente altura. La malla final se

compondrá de un plano principal, por el que los agentes podrán desplazarse, y planos secundarios, ubicados a diferentes alturas definidas por los objetos que representan.

Para el desarrollo del proyecto se ha optado por no incluir mallas de altura mútiple, puesto que las mecánicas que se desean implementar no implican el salto o desplazamiento a través de diferentes alturas.

6.4.4 Agentes de Navegación

En los primeros prototipos se ha empleado esta A la hora de realizar la toma de decisiones por parte de cada agente. Usando la posición del agente y la distancia a éste medida pasos hacia su ubicación en el grafo, el agente es capaz de llegar

Esto puede solventase dotándole de un cierto conjunto de acciones que permitan forzar la toma de cursos de acción en caso de aproximarse a situaciones límite.

6.4.5 Implementación en el proyecto

La infraestructura de mallas se ha implementado mediante el sistema provisto por Unity. Este sistema de creación y modificación de mallas, así como diferentes tipos de agentes, ya sean humanoides u otros tipos.

Para ello los diferentes agentes integran un componente tipo *NavMeshAgent*, que les permite navegar por la malla de navegación generada. Asimismo, al integrar este tipo de componente, se incluyen las llamadas y métodos públicos que proveen. Mediante su inyección en el método de ejecución de los estados de la MEF, el agente puede obtener o recargar un destino, desplazarse hacia otro lugar o detenerse.

Una vez integrado en la instancia de zombi base, el componente tiene la particularidad de requerir de forma implícita una malla para poder inicializarse. El entorno de desarrollo permite generarlas teniendo en cuenta una serie de parámetros como la anchura, su altura, el ángulo máximo de escalada o el tipo de superficie por la que puede desplazarse. Además, en la generación de la malla se pueden configurar qué capas, o *layers*, entran dentro del análisis de geometrías, pudiendo generar mallas para diferentes enemigos que permitan una mayor variedad de opciones dentro del mapa.

Con la construcción de la malla, se generan datos asociados dentro del proyecto, que contienen datos relativos al análisis y al grafo producto de éste. Es importante reseñar que estos datos serán utilizados durante el tiempo de juego por los agentes.

7 SEGUNDA APROXIMACIÓN: ARBOLES DE COMPORTAMIENTO

Este apartado versará sobre la conceptualización, análisis e implementación de la segunda parte del desarrollo de la IA, que se corresponde con los árboles de comportamiento.

7.1 Arboles de comportamiento

Las máquinas de estado finitas, aunque útiles, son muy limitadas a la hora de expandir los comportamientos que el agente puede presentar. Esto es debido a que el número de transiciones posibles crece con cada nuevo estado que se agregue a la máquina. Por consiguiente, el sistema se torna inasumible en coste y desarrollo.

Como contraparte a la FSM, para las siguientes versiones de los agentes se emplea una nueva estructura: el árbol de comportamiento, o *Behavior Tree* (BT).

7.1.1 Definición

Un árbol de comportamiento es una estructura de grafo de tipo arbóreo, como indica su nombre. Dispone de una serie de nodos interconectados entre sí de forma jerárquica, que pueden ser explorados mediante un algoritmo de búsqueda.

El objeto final del proceso de búsqueda es dar con una solución o no terminal, que tendrá una serie de acciones. Esas acciones serán acordes con el objetivo de la búsqueda realizada. Por ejemplo, si un agente decide llevar a cabo una persecución, la solución obtenida del nodo terminal debe ir acorde con esa decisión.

En esencia, el árbol de comportamiento permite al agente que lo implementa emular el razonamiento humano empleando métodos de descarte. Presenta una arquitectura basada en subprocesos de selección y secuenciación. A su vez, estos subprocesos pueden ser combinados e intercambiados durante el momento del desarrollo del árbol, lo cual lo hace más flexible y moldeable que una máquina de estados finitos.

Añadido a esto hay que destacar el hecho de que la propia estructura del árbol promueve la priorización de recursos a la hora de ser construido. Las acciones que se planteen para el agente deben ser sometidas a un análisis previo que tenga en cuenta, entre otros, el contexto de acciones a realizar y la importancia de éstas dentro de la interacción del agente con el medio.

En otras palabras, es preciso definir qué acciones tienen mayor peso sobre el comportamiento del agente y establecer la construcción del árbol acorde a dicha definición de prioridades. El agente resultante debe poder ejecutar el comportamiento adecuado para únicamente aquella o aquellas situaciones concretas para las que este fuese planteado.

7.1.2 Componentes

Como se ha comentado, el árbol de comportamiento se basa en una serie de componentes que permiten descartar ramas enteras si así requiere para el proceso de razonamiento. Dichos componentes se denominan *nodos*, y pueden ser analizados en función a su posición en la jerarquía o su función en el proceso de razonamiento.

Según el primer aspecto, destacamos tres tipos:

- Nodos hoja, *Leaf*: puntos finales de rama. No tienen hijos asociados. Los nodos hojas integran el código para realizar una acción específica.
- Nodos de modificación, o *Decorators*: son nodos que modifican el resultado de la evaluación del subárbol que se encuentra por debajo. Solo pueden tener un hijo asociado.
- Nodos compuestos, o *Composite*: puntos intermedios en la jerarquía. Tienen múltiples hijos asociados. Permiten realizar la evaluación de los subárboles que surgen de ellos y, según su resultado, tomar las acciones contenidas en los nodos hoja de la rama resultante.

Según el segundo, encontramos los siguientes tipos:

- Nodos de verificación o *Checkers*: nodos hoja que comprueban una condición u otro elemento necesario para la ejecución de un comportamiento.
- Nodos de acción, o *Tasks*: nodos hoja que aglutinan las instrucciones de un comportamiento específico.
- Nodos de selección, o *Selectors*: nodos compuestos que permiten la resolución del subárbol que cuelga de ellos siempre que al menos una de sus ramas se resuelva correctamente.
- Nodos de secuencia, o *Sequence*: nodos compuestos que permiten la resolución del subárbol que cuelga de ellos siempre y cuando todas sus ramas se resuelvan correctamente.

Todos estos elementos son acogidos bajo un mismo contrato para estandarizar el método común de evaluación. De esta forma, las llamadas a la evaluación de cada nodo pueden ser encapsuladas su interfaz, evitando la dependencia de la implementación.

Para establecer cuándo una exploración se considera aceptable, se plantean tres estados posibles:

- En curso, o *Running*: el nodo se encuentra en proceso de evaluación y no hay resultado concreto aún.

- Éxito, o *Success*: el nodo ha finalizado su evaluación y las condiciones que solicita se han cumplido de forma satisfactoria.
- Fracaso, o *Failure*: el nodo ha finalizado su evaluación y las condiciones que requiere no se han cumplido.

La exploración de cada rama debe dar alguno de estos resultados. Según dicho resultado, el agente tomará las acciones pertinentes.

Además, los nodos deben disponer de un mecanismo interno para intercambiar información que pueda ser relevante a la hora de realizar las evaluaciones. Este mecanismo recibe el nombre de *blackboard* (Aversa, Kyaw, & Peters, 2018), y permite que un nodo registre datos que luego pueden ser visibles para resto del árbol. Por ejemplo, si un agente requiere de la existencia de un objeto en su posesión para realizar determinadas acciones, registrar dicho objeto en la *blackboard*. En las sucesivas exploraciones el resultado del razonamiento se verá alterado al haber sido modificado el contenido del *blackboard*.

7.1.3 Estrategia de búsqueda

Dada una situación particular, el agente debe ser capaza de resolver un problema planteado como el final de un proceso de descarte.

En términos formales, árbol de comportamientos puede ser tomado como un espacio de estados que el agente ha de recorrer. Por cada estado que se evalúa, el agente podrá tomar una serie de transformaciones que lo llevarán a descartar dicho camino o a continuar explorando por el estado consecuente resultado de las trasformaciones aplicadas.

La estrategia de búsqueda que se aplique sobre el árbol determinará el diseño del comportamiento, debido que el empleo de un algoritmo u otro influirá en la resolución del proceso evaluación de cada rama de éste y en los estados a explorar. También es preciso definir la existencia o ausencia de conocimiento asociado que pueda influir a la hora de resolver las evaluaciones en los nodos intermedios.

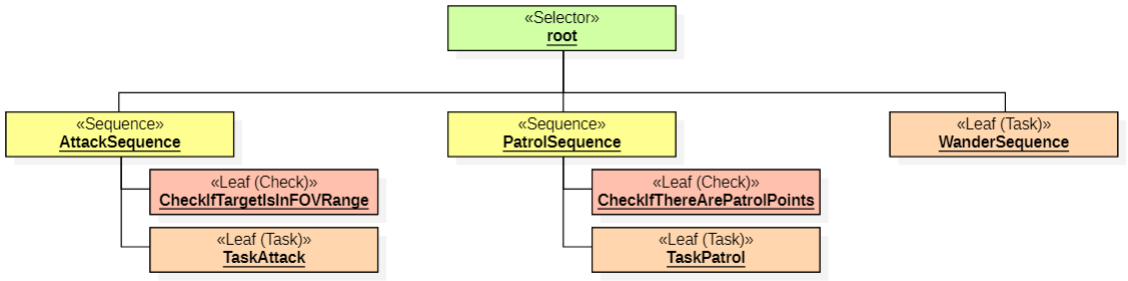
Para el caso, la metodología que se aplicará queda enmarcada en las estrategias una búsqueda no informada, pues no se considera la existencia de una heurística per se, al contrario que con las mallas de navegación. La prioridad de evaluación de los elementos empleará una cola tipo FIFO (*First In First Out*), por lo que los primeros elementos que se inserten en el nodo raíz serán los primeros en ser evaluados. Por otro lado, su morfología obliga a comprobar cada subrama hasta verificar con el comportamiento adecuado. En principio, la estrategia de búsqueda en profundidad (*DFS*) se ajusta a esta descripción.

La figura 19 muestra un árbol de comportamiento básico para un agente predeterminado. Las acciones permitidas son las mismas que para la FSM presentada anteriormente, es decir, atacar (*Attack*), patrullar (*Patrol*) y vagabundear (*Wander*). Quedan distribuidas en este mismo orden de izquierda a derecha, siendo la acción de ataque la más prioritaria al ser la primera en la cola, y la de vagabundear la menos prioritaria por su consecuente posición en la cola.

Se muestran además los arquetipos asociados a cada nodo del árbol. El nodo raíz es de tipo selector, seguido por los nodos secuencia que representan los comportamientos a realizar. Bajo cada nodo secuencia se encuentran los nodos de verificación y acción asociados, que permitirán definir si ese comportamiento puede o no ejecutarse y que acciones conlleva.

Figura 19: Ejemplo de Árbol de Comportamiento

Árbol de comportamiento con tres vías de actuación. Los arquetipos de cada nodo quedan representados encima de su nombre.



El proceso de selección de comportamiento es simple. En cada iteración del tiempo de juego, el motor dispara, a través de su método de actualización la llamada al evaluar del nodo raíz. Este comienza a verificar sus hijos a partir del primero que se añadió en su cola. El contexto de evaluación cambia al hijo si se trata de un nodo compuesto o un decorador y repite de forma recursiva el proceso hasta llegar un nodo hoja.

Una vez en este punto, cada uno de los nodos hojas de la última capa retorna el resultado de su evaluación al padre y el proceso se repite de forma recursiva hasta llegar al nodo raíz. Si el resultado de esa evaluación es de éxito, el agente ejecutará es comportamiento. Si el resultado del proceso recursivo sobre esa rama es de fallo, se procederá a evaluar la siguiente rama en orden desde el nodo raíz.

Por ejemplo, supongamos que planteamos una situación en la que la acción que ha de ejecutar el agente es vagabundear por el entorno. Para que esta situación se de forma propicia, se deben de cumplir las siguientes condiciones:

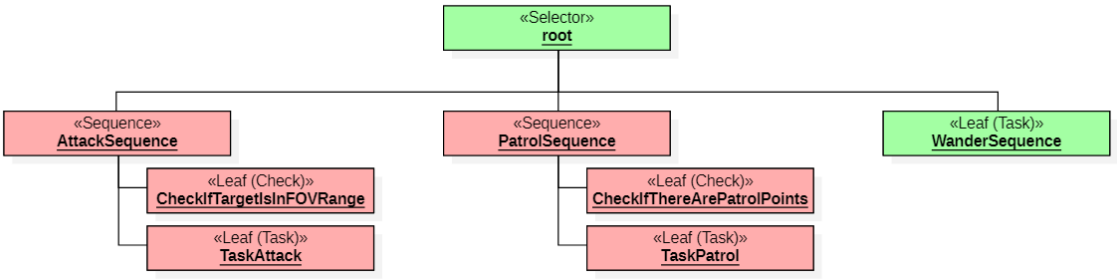
- Existe una instancia de agente.

- La evaluación de la rama de ataque falla, no hay una jugador u objetivo al que atacar.
- La evaluación de la rama de patrulla falla, no existen puntos de patrulla a los que el agente pueda acceder para iniciar un ciclo de patrulla.

El resultado del árbol evaluado quedaría como se muestra en la figura 20, con todas ramas de mayor prioridad completamente recorridas con resultado infructífero y la última rama existente con resultado exitoso.

Figura 20: Recorrido de un BT empleando DFS

Resultado de la evaluación del árbol de la figura 19. En verde los nodos exitosos y en rojo los nodos infructuosos.



En este caso las secuencias de ataque y patrulla evalúan todos sus nodos y como resultado se obtiene fallo en ambas. Por descarte la única opción posible para el agente es vagabundear esperando un cambio en el entorno en próximas iteraciones.

7.1.4 Optimización

No obstante, a pesar de ser superior en rendimiento y escalabilidad a las máquinas de estados finitos, el árbol de comportamientos también adolece de carencias en cuanto a optimización. La principal radica en el algoritmo de búsqueda en profundidad. Aunque es sencillo de implementar, DFS es propicio a generar problemas de optimización si el árbol es demasiado grande o si se generan bucles dentro de la propia estructura.

Si bien es un algoritmo completo para este caso, pues el árbol presenta un espacio de estados finito, hay que recordar que no asegura la optimalidad, ya que puede haber una solución mejor en una rama con menor profundidad que aún no haya sido explorada. Asimismo, sus órdenes de complejidad espacial y temporal pueden llegar, en el peor de los casos, a ser $O(b^d)$ y $O(b^m)$, siendo b el factor de ramificación efectivo, d la profundidad de la solución menos costosa y m la máxima profundidad del espacio de estados, respectivamente (Wikipedia, 27).

Para verificar el orden de complejidad del ejemplo anterior tendremos que calcular el factor de ramificación efectivo, b*. Este se aplica a árboles balanceados, es decir, aquello en los que todas sus posibles trayectorias tienen el mismo número de hijos. Es posible calcularlo resolviendo la ecuación trascendente correspondiente:

$$N + 1 = 1 + b^* + (b^*)^2 + (b^*)^3 + \cdots + (b^*)^d$$

Para el árbol ejemplo en particular, el factor de ramificación se debe calcular de forma específica empleando el factor de ramificación promedio, debido a que está desbalanceado hacia la izquierda y no todos los intermedios dan el mismo número de hijos.

Entiéndase el factor de ramificación efectivo como el cociente entre los nodos no raíz divididos entre los nodos no hoja (Hmong.es, s.f.). Para el ejemplo presentado anteriormente, el resultante de esta operación queda como:

$$b = \frac{7\ nodos\ no\ raíz}{3\ nodos\ no\ hoja} = 2{,}33$$

Lo cual nos deja como resultado valor alejado de 1, que es el que se considera deseable. Teniendo en cuenta la notación de órdenes, tendríamos que:

- En el peor de los casos, su orden espacial sería de $O(2{,}33^2)$. Su factor de ramificación efectivo es de 2,33, y su profundidad es de 2 niveles.
- Su orden temporal sería de $O(2{,}33^1)$, pues el factor de ramificación efectivo es de 2,33 y la solución que se encuentra a menor profundidad está en el primer nivel.

Si bien es un ejemplo, cuando la estructura del árbol aumente en profundidad o expansión de nodos, el tiempo de exploración aumentará en consecuencia y, por ende, el agente podría no ejecutar el comportamiento esperado antes de la siguiente iteración.

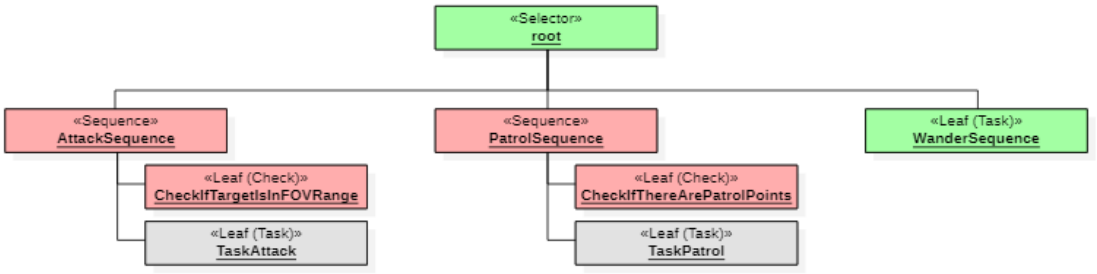
Para paliar este problema establecemos que los nodos de secuencia se ajusten estrictamente a su comportamiento lógico durante su tiempo de evaluación. Esto es, los nodos secuencias dejarán de explorar sus hijos desde el momento en el que se reciba un fallo. Al evaluarse como puertas AND de facto, no tiene sentido seguir evaluando una vez que una de las condiciones ha fallado.

Este cambio no es aplicable a los nodos selectores, pues su comportamiento es el de una puerta OR, ya que toman la primera vía que da resultado satisfactorio. Sin embargo, si estructuramos el árbol de forma que de cada selector cuelguen únicamente nodos secuencia o

de tarea y aplicamos el cambio mencionado, logramos reducir en gran medida el tiempo de decisión obviando la evaluación de aquellos nodos hoja no necesarios, como se muestra en la figura 21

Figura 21: Recorrido de un BT usando DFS con optimización

Resultado de la evaluación del árbol de la figura 20. En este caso, los nodos secuencian obvian los hijos siguientes al primero que da Fallo como resultado. En verde los nodos exitosos, en gris los no evaluados y en rojo los nodos infructuosos.



Como se ve, aplicando el cambio al método de exploración obtenemos, para este caso, que el factor de ramificación promedio en el peor de los casos se reduce de forma considerable, puesto que solo exploramos un nodo por cada secuencia. De facto queda que:

$$b = \frac{5\ nodos\ no\ raíz\ efectivos}{3\ nodos\ no\ hoja} = 1{,}66$$

Con ello obtenemos un valor mucho más próximo a 1. Por extensión, esto genera que el orden del árbol se vea reducido, quedando tal que:

- En el peor de los casos, su orden espacial optimizado sería de $O(1{,}66^2)$. Su factor de ramificación efectivo es de 1,66, y su profundidad es de 2 niveles.
- Su orden temporal sería de $O(1{,}66^1)$, pues el factor de ramificación efectivo es de 1,66 y la solución que se encuentra a menor profundidad está en el primer nivel.

Como se ve, este cambio en la evaluación del algoritmo nos garantiza reducir el orden de complejidad para el peor de los casos posibles, es decir el ultimo comportamiento en orden de prioridad. Con ello hemos mejorado significativamente el rendimiento en la exploración, pudiendo seguir expandiéndolo para dar cabida más subárboles de secuencia o selectores que abarquen más comportamientos.

7.2 Implementación de la arquitectura BT en el proyecto

Una vez estimadas y analizadas las principales características y funcionalidades de la arquitectura a implementar, así como su mesura. Se realizó la implementación en código.

Para facilitar el proceso de desarrollo se optó por aplicar abstracción extrayendo los métodos comunes a una clase padre abstracta compartida y sobrescribiéndolos localmente las hijas de esta. A continuación, los métodos abstractos serían extraídos a una interfaz superior para poder ser instanciados en los test como instancias *mock*.

Esta organización sería replicada en la codificación de las clases contenedoras de árboles, pudiendo generar múltiples instancias de árboles diferentes para probar y añadir rutinas de forma escalada antes de llegar a una primera versión estable. Para facilitar la organización, todas las clases quedan englobadas dentro del espacio de nombre *BehaviorTree*.

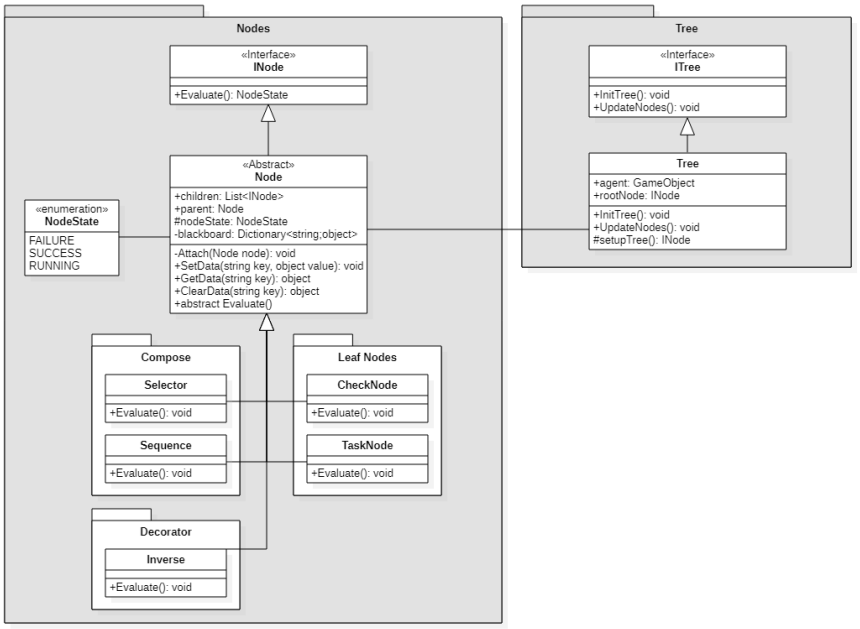
A nivel abstracto, la jerarquía de clases quedaría de la siguiente forma:

- Interfaz *INode*: representa las instancias de nodo existentes en el proyecto. Contiene el método de evaluación.
- Clase abstracta *Node*: contiene la lógica de las instancias *nodo* del árbol. Incluye además la estructura de *blackboard* o pizarra ya descrita.
- Clase Enumerada *NodeState*: contiene los tres estados posibles resultado de la evaluación de cada nodoa, *Failure*, *Success* y *Running*.
- Interfaz *ITree*: contiene los métodos de inicialización y cargado de los árboles.
- Clase abstracta *Tree*: contiene la lógica de las instancias *árbol*, incluyendo además una referencia hacia el *GameObject* agente que lo emplea para seleccionar los comportamientos.

La figura 22 muestra un el modelo de clases generalizado que ha resultado de aplicar la estructura descrita. Como se puede constatar, los árboles tienen un nodo raíz a partir del cual disparan el proceso de evaluación, el nodo raíza puede o no tener múltiples hijos y tanto él como su descendencia implementan el mismo contrato.

Figura 22: Modelo de clases para creación de BT's

Modelo de clases general para la creación de árboles de comportamiento.



7.2.1 Clase Sequence

El método de evaluación de la clase Sequence actúa tal y como se ha descrito a nivel conceptual. En primer lugar, revisa la lista de hijos que tiene asociados. Comienza a explorar por el primero que tiene disponible. Si este hijo devuelve estado de terminación fallido, finalizará la exploración y retornará *Failure*. Si, por el contrario, dicho hijo retorna *Success*, continuará con el proceso de exploración hasta finalizar con toda la secuencia asociada. Cabe destacar que puede verificar si existe algún nodo por el cual no ha recibido respuesta. Si se da ese caso, entonces devuelve el estado *Running*, y el agente que ejecutando esa rutina.

Por ejemplo, si en el caso de ejemplo se dieran las condiciones para atacar, el agente exploraría hasta el nodo de tarea de ataque. Este retornaría *Running*, propagando el resultado a su nodo secuencia padre y, por ende, al selector *root*. El agente quedaría en estado de ataque hasta la siguiente evaluación. La figura 23 muestra un detalle del método de evaluación del nodo secuencia.

Figura 23: Evaluación en la clase Sequence

Detalle del método de evaluación en la clase Sequence.

```
public override NodeState Evaluate() {
       bool anyChildIsRunning = false;
       foreach (Node child in _children) {
              switch (child.Evaluate()) {
```

```
                case NodeState.FAILURE:
                        state = NodeState.FAILURE;
                        return state;
                case NodeState.SUCCESS:
                        continue;
                case NodeState.RUNNING:
                        anyChildIsRunning = true;
                        continue;
                default:
                        state = NodeState.SUCCESS;
                        return state;
                }
        }

        state = anyChildIsRunning ? NodeState.RUNNING : NodeState.SUCCESS;
        return state;
}
```

7.2.2 Clase Selector

El método de evaluación de la clase Selector, al contrario que la clase Sequence, no contempla el uso del estado *Running*. Acorde con su funcionalidad dentro del esquema del árbol, su objetivo es dar salida a bifurcaciones en éste.

Cuando se lanza el proceso de evaluación, toma al primer hijo existente en la lista de hijos que tiene asociada. Evalúa de forma recursiva y si dicho hijo retorna exitoso, finaliza su ejecución y el agente queda ejecutando el comportamiento asociado a esa rama. Si por el contrario el resultado es insatisfactorio, pasará al siguiente nodo hijo y asía sucesivamente hasta que, o bien no haya resultado satisfactorio, o bien uno de los restantes hijos de éxito, en cuyo caso retornará *Failure* y *Success*, respectivamente.

Como se puede apreciar en la figura 24, el detalle del método de evaluación del selector fallará si ninguno de sus hijos es exitoso. Ante cualquier otro caso, el resultado será el estado correspondiente a la resolución de dicha rama.

Figura 24: Evalucación en la clase Selector

Detalle del método de evaluación en la clase Selector.

```
public override NodeState Evaluate() {
        foreach (Node child in _children) {
                switch (child.Evaluate()) {
                        case NodeState.SUCCESS:
                                state = NodeState.SUCCESS;
                                return state;
                        case NodeState.RUNNING:
                                state = NodeState.RUNNING;
                                return state;
                        default:
                                continue;
                }
        }
```

```
        state = NodeState.FAILURE;
        return state;
}
```

7.2.3 Estructura Blackboard en los nodos

Para implementar la estructura del *Blackboard*, se consideró emplear una estructura de diccionario. Ésta permite almacenar como pareja de clave-valor aquellos datos que se consideren útiles para el proceso de razonamiento. Éstos pueden ser almacenados, consultados o eliminados durante el proceso de evaluación, facilitando el proceso de descarte de ramas si son relevantes para la resolución de dichas ramas del árbol.

Por ejemplo, para que un agente decida si perseguir al jugador o no, es necesario verificar si existe o no existe un jugador en las inmediaciones de su posición. Si existe, puede comprobarse si está a distancia para atacar, lo cual podemos considerar como caso genérico, o iteración *n*. El inconveniente aparece en la primera y última iteraciones del ciclo de evaluación, es decir, cuando el agente *no sabe* que existe un jugador cerca de él o cuando el jugador desaparece y el agente debe verificar que comportamiento realizar.

En esos el *blackboard* hace la función de "memoria" del agente, mediante transacciones contra éste, el agente puede verificar si existe algún agente mediante una consulta de tipo *get* al diccionario. Si la consulta es exitosa, el proceso de comprobación continuará su curso habitual. Si es infructuosa, el nodo de comprobación puede verificar si esa condición se cumple y a continuación registrarlo en el diccionario, o dar por finalizada su ejecución y retornar *Failure*.

Para la gestión de datos, la clase abstracta *Node* incluye tres métodos que permiten realizar todas las transacciones necesarias:

- GetData(): retorna el objeto especificado con la clave que se facilita por parámetro, si se encuentra en el diccionario, *null* en caso contrario.
- SetData(): crea o actualiza el valor asociado a la clave que se la facilita. Tanto valor como clave entran como parámetros.
- ClearData(): elimina del diccionario el objeto asociado con la clave pasada como parámetro, si se encuentra registrado en éste.

Cabe destacar que el proceso de comprobación se realiza de forma recursiva mediante *backtracking*. Cuando un nodo requiere de transaccionar con en el diccionario sobre algún objeto especifico. Lo busca en su entorno local y no lo encuentra, verifica en los nodos superiores de forma iterativa hasta encontrar uno cuyo diccionario lo contenga, o en caso contrario, retornando *null* si no está registrado. El sistema de limpieza de datos funciona de

forma equivalente, retornando verdadero o falso si el objeto pudo ser, o no borrado del diccionario.

Con respecto a la inserción, si se diese el caso de no existencia, el nodo puede tomar dos cursos de acción. El primero es insertar el nodo en el diccionario. El segundo es abortar el proceso de evaluación y retornar el estado correspondiente.

Esta aproximación nos permite obtener dos ventajas de cara a la construcción de los árboles:

- La responsabilidad de obtener los datos necesarios para las funciones locales reside en cada nodo, por lo que se puede asumir que, si existen *n* instancias de un nodo en el árbol, la primera escritura del mismo valor puede realizarse entre 0 y *n* veces, según se desarrolle el curso de la partida.
- La existencia de nodos intermedios que almacenen información relevante ayuda en consecuencia a restringir el proceso de *backtracking*. Por ejemplo, si dos nodos situados en la misma rama a diferente profundidad necesitan de un dato concreto, una vez que el situado a menor profundidad verifique ese dato, el nodo más profundo no requerirá de ascender hasta la raíz sino hasta *j-k* niveles, donde *j* es su nivel y *k* es el de su nodo ancestro.

7.2.4 Tree y Tree Generator

Para englobar los diferentes árboles planteados dentro de la estructura de clases existentes, se decidió añadir la clase abstracta *Tree*. Esta integra la configuración y jerarquía de nodos de cada árbol existente en el proyecto. Al tratarse de una clase propia, nos aseguramos de que cada instancia se corresponde con lo explicitado en su modelo correspondiente.

Sin embargo, el funcionamiento de Unity impide acoplar directamente estos scripts por dos motivos esenciales:

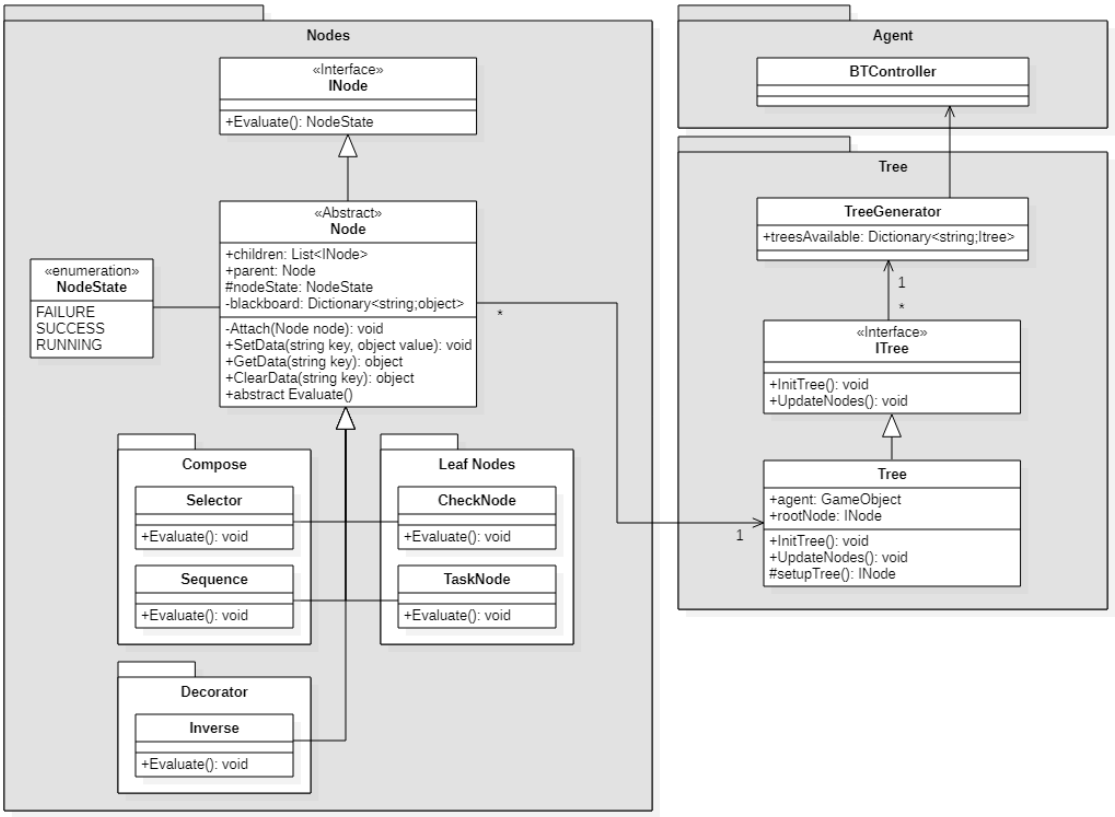
- El primero es que al no ser instancias derivadas de la clase abstracta *MonoBehavior*, no pueden ser localizadas en el entorno de desarrollo.
- El segundo es que dentro de cada instancia de árbol se encuentra una configuración diferente, por lo que conviene tener una estructura superior que permita al controlador de agente precargar el árbol correcto de maneara automática al momento de iniciar la escena de juego.

Para solventarlo se decidió aplicar la misma dinámica que con los nodos, es decir, extraer a una interfaz *ITree* los métodos comunes de la clase abstracta y a continuación

encapsular la creación de las implementaciones de dicha interfaz bajo un generador de árboles, *TreeGenerator*, que es la entidad consultada en el momento en el que cada *GameObject* agente carga sus atributos. La figura 25 muestra el modelo de clases asociado a este subsistema.

Figura 25: Modelo de clases para generar árboles.

Detalle del sistema de generación de árboles completo. Obsérvese la presencia del TreeGenerator como intermediario entre controlador e instancias ITree.



Cuando el agente solicita al TreeGenerator la instancia de árbol que se le ha especificado, este busca en el diccionario interno el valor asociado con la clave representada por el nombre del árbol, retornando una nueva instancia de dicho árbol.

Esta estructura garantiza, además, una total independencia entre los agentes y conjunto de árboles. Cada prefabricado de agente puede ser configurado con cualquier instancia de árbol, lo cual permite plantear partidas con enemigos de un mismo tipo, pero con comportamientos diferentes.

A nivel puramente técnico, este planteamiento arquitectónico nos asegura también el poder aumentar el número de prototipos de forma inocua al proyecto, pues no se considerarán existentes en tanto que no haya ningún agente que los solicite al *TreeGenerator*, por lo que se

pueden probar versiones del árbol en la zona de pruebas y al mismo tiempo depurar otros aspectos de la escena actual sin que suponga conflicto alguno.

7.3 Arboles implementados

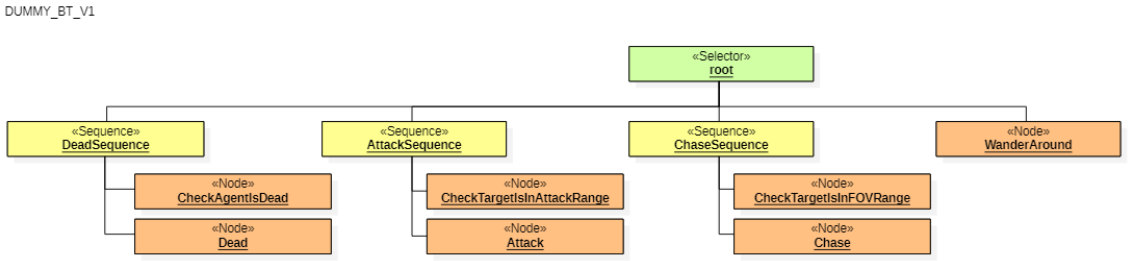
7.4 Primera versión

7.4.1 Zombi Simple

La estructura planteada es una extrapolación de la máquina de estados planteada para su primera versión. Consta de cuatro posibles comportamientos disponibles para ejecutar, de los cuales tres secuencias y un comportamiento final por defecto. La priorización establecida para éste se puede apreciar en la figura 26, que presenta los estados ordenados por orden de prioridad.

Figura 26: Primera versión del Zombi Simple

Detalle del árbol genérico del proyecto. Las están establecidas de acuerdo al orden de prioridad, siendo la ubicada más a la izquierda la más prioritaria, y la ubicada a la derecha la menos prioritaria.



Como se puede constatar el árbol resultante es la representación de la máquina de estados planteada en la etapa inicial de desarrollo. El agente puede ejecutar el procedimiento de muerte y desactivarse, atacar si se encuentra a la distancia adecuada del jugador, perseguirlo si, no pudiendo atacar, el jugador se encontrase en las inmediaciones de un rango preestablecido. Si ninguna de estas opciones fuese posible, tomará la acción por defecto y vagabundea por el mapa hasta que se genere la situación propicia para cualquiera de las otras posibilidades.

En el plano teórico, el árbol presenta un total de diez nodos no raíz y cuatro nodos no hoja. Al no estar balanceado, el factor de ramificación resultante queda:

$$b = \frac{10\ nodos\ no\ raiz}{4\ nodos\ no\ hoja} = 2{,}5$$

Sin embargo, al haberse optimizado para evitar la exploración de ramas una vez el primer nodo hijo falla, el número de nodos hijos efectivo queda en siete. En consecuencia, el factor de ramificación efectivo se aproxima mucho más a la unidad resultando:

$$b = \frac{7\ nodos\ no\ raiz}{4\ nodos\ no\ hoja} = 1{,}75$$

Por tanto, los órdenes de complejidad espacial y temporal para este esquema quedan de la siguiente forma:

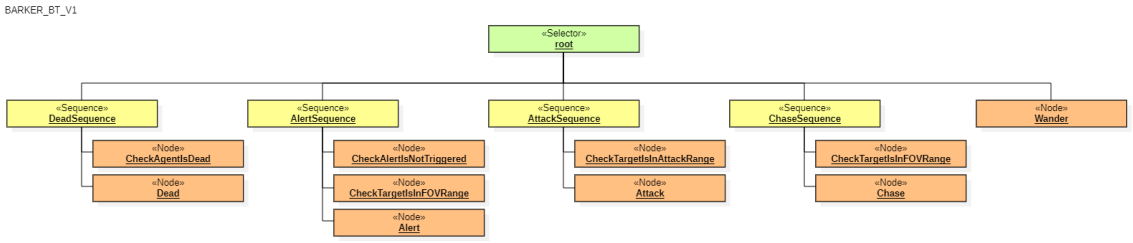
- En el peor de los casos, su orden espacial optimizado sería de $O(1{,}75^2)$. Su factor de ramificación efectivo es de 1,75, y su profundidad es de 2 niveles.
- Su orden temporal sería de $O(1{,}75^1)$, pues el factor de ramificación efectivo es de 1,75 y la solución que se encuentra a menor profundidad está en el primer nivel.

7.4.2 Zombi Aullador

Una vez determinado el árbol genérico, procedimos a crear una versión extendida que incluyese los comportamientos específicos para el segundo tipo de agente, el aullador o *Barker*. La figura 27, muestra la ampliación de la disposición en una rama que encapsulando el comportamiento de alertar para esta primera versión.

Figura 27: Primera versión del Zombi Aullador

Detalle del árbol del aullador en su primera versión. Nótese la inclusión del nuevo comportamiento de alerta



En esta versión, al aumentarse el número de nodos hoja y nodos no raíz hay que recalcular el factor de ramificación. Obviaremos el factor general, ya que no es un árbol balanceado, por lo que el cálculo del factor efectivo teniendo en cuenta la optimización aplicada queda:

$$b = \frac{9\ nodos\ no\ raiz}{5\ nodos\ no\ hoja} = 1{,}8$$

Los órdenes de complejidad espacial y temporal para este esquema quedan, por tanto:

- En el peor de los casos, su orden espacial optimizado sería de $O(1{,}8^2)$. Su factor de ramificación efectivo es de 1,8, y su profundidad es de 2 niveles.
- Su orden temporal sería de $O(1{,}8^1)$, pues el factor de ramificación efectivo es de 1,8 y la solución que se encuentra a menor profundidad está en el primer nivel.

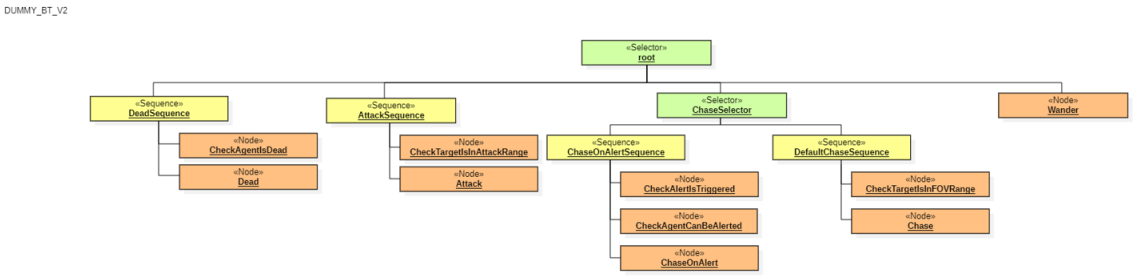
7.5 Segunda versión

7.5.1 Zombi Simple

Para el caso del personaje simplón, el cambio se aplica sobre el estado de persecución. Ahora puede determinar si persigue al jugador en función al efecto una alarma o, en su defecto, persigue al jugador usando la rutina base. Para el primer caso se añaden dos nodos de verificación, como se aprecia en la figura 28, que comprueban si se ha iniciado una alerta si el agente se encuentra en una posición lo suficientemente cercana para poder ser alertado.

Figura 28: Segunda versión del Zombi Simple

Detalle del árbol del aullador en su segunda versión. Nótese la bifurcación de la rama de alerta.



El cálculo del factor de ramificación efectivo en este caso queda, para el peor caso posible:

$$b = \frac{10\ nodos\ no\ raiz}{6\ nodos\ no\ hoja} = 1{,}66$$

Y para el caso en el que se resuelve como rutina la persecución por defecto, posible camino más largo después de la peor situación plausible:

$$b = \frac{10\ nodos\ no\ raiz}{6\ nodos\ no\ hoja} = 1{,}66$$

Como en el caso anterior, ambos resultados son equivalentes, por lo que los órdenes de complejidad y espacio, para esta versión, quedan de la siguiente forma:

- En el peor de los casos, su orden espacial optimizado sería de $O(1{,}66^3)$. Su factor de ramificación efectivo es de 1,66, y su profundidad es de 3 niveles.
- Su orden temporal sería de $O(1{,}66^1)$, pues el factor de ramificación efectivo es de 1,66 y la solución que se encuentra a menor profundidad está en el primer nivel.

7.5.2 Zombi Aullador

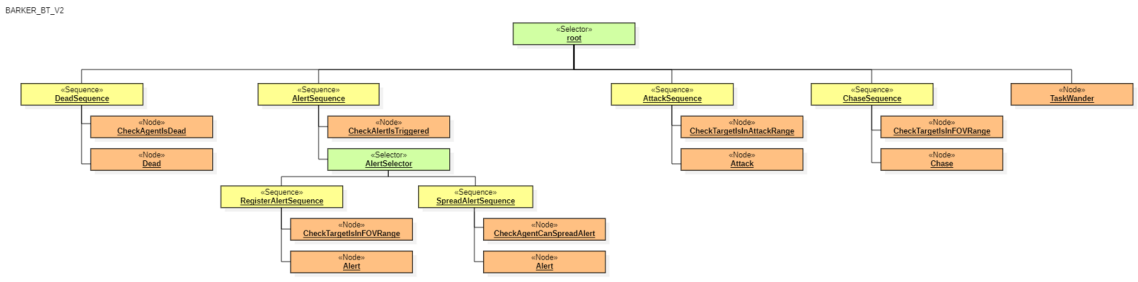
El principal inconveniente de la primera versión es la incapacidad entre ambos tipos de agentes para poder comunicarse. El aullador puede lanzar una señar de alerta y alertar durante intervalo de tiempo T, pero al no existir un nexo común, este comportamiento queda reducido a un simple cambio de estado. Para solventar este problema se decidió añadir al proyecto dos nuevas clases:

- AlertManager: se trata de una instancia de tipo *Singleton*, que puede ser consultada por los agentes durante el proceso de razonamiento. En esta versión, simplemente establece si se ha dado una alarma o no, permitiendo a los aulladores verificar si pueden alertar o propagar la alerta.
- AlertController: Se trata de una clase controlador, integrada en el objeto de juego del jugador. Cuando el agente es alertado, recurre a este recurso para inicializar el estado de alarma en la clase singleton.

De esta manera el árbol queda modificado como se puede observar en la figura 29, con la secuencia de alerta bifurcada en dos subramas mediante un selector. Ahora el aullador puede decidir entre iniciar un proceso de alerta, expandir el área de alerta, o continuar razonando.

Figura 29: Segunda versión del Zombi Aullador

Detalle del árbol del aullador en su segunda versión. Nótese la bifurcación de la rama de alerta.



Este incremento en el número de ramas conlleva un aumento en la complejidad del árbol y de sus órdenes. El factor de ramificación para esta versión queda tal que:

$$b = \frac{19\ nodos\ no\ raiz}{8\ nodos\ no\ hoja} = 2{,}38$$

Que, aplicando el esquema optimizado, y suponiendo el peor caso posible, deja el factor efectivo en:

$$b = \frac{9\ nodos\ no\ raiz}{5\ nodos\ no\ hoja} = 1{,}8$$

El factor efectivo se mantiene constante debido a que si el nodo verificador de alerta falla, el resto del árbol conserva el mismo número de nodos restante que la primera versión. Comprobamos además si el factor de ramificación para el caso en el que al menos una de las dos ramas del árbol de alarma es recorrida de forma satisfactoria es significativamente mayor. Por ejemplo, supongamos que el razonamiento da como resultado propagar la alerta. Para ese caso, el agente ha tenido que fallar a la hora de verificar si puede lanzar una alerta. En consecuencia, el factor efectivo queda como:

$$b = \frac{11\ nodos\ no\ raiz}{6\ nodos\ no\ hoja} = 1{,}83$$

Que queda muy aproximado al peor caso posible establecido. En resumen, los órdenes de complejidad espacial y temporal quedan tal que:

- En el peor de los casos, su orden espacial optimizado sería de $O(1{,}8^3)$. Su factor de ramificación efectivo es de 1,8, y su profundidad es de 3 niveles.
- Su orden temporal sería de $O(1{,}8^1)$, pues el factor de ramificación efectivo es de 1,8 y la solución que se encuentra a menor profundidad está en el primer nivel.

7.6 Tercera Versión

7.6.1 Zombi Aullador

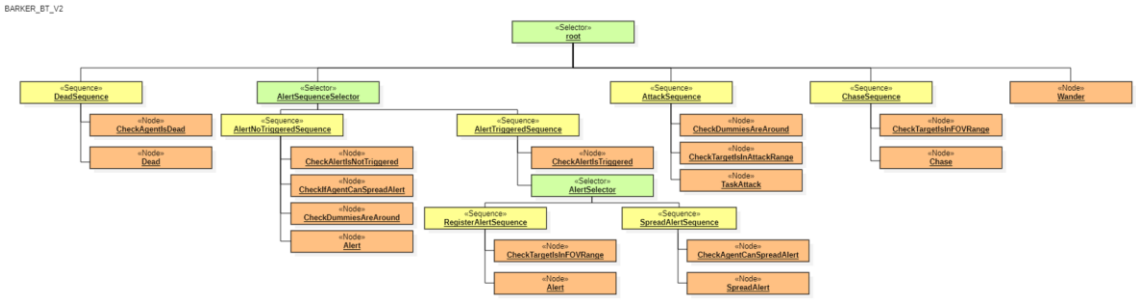
La segunda versión del aullador, aunque a mejora la original, generó problemas durante las pruebas en la escena de juego. Se detectó que era debido a la propia jerarquía establecida.

El agente entraba en estado de alarma y no lograba salir del mismo debido a que no existía ninguna estructura de control que lo impidiese, o en su defecto, que lo desactivase durante un período de tiempo establecido.

Para solventarlo, se añadió una subrutina en el script *EnemyController* que establecía un período de congelamiento (*Cooldown*) una vez el proceso de alertar se hubiera iniciado, se habría de esperar un tiempo establecido para poder reiniciarse. La subrama del proceso de alerta fue alterada, como se ve en la figura 30.

Figura 30: Tercera versión del Zombi Aullador

Detalle del árbol del aullador en su tercera versión. Nótese la modificación de la rama de alerta respecto del modelo previo.



En esta versión, el agente verifica en primera instancia si se ha disparado alguna alarma. Si no se cumple, intenta iniciar el proceso de propagación. Si se da la condición opuesta, el proceso se mantiene equivalente a la versión anterior.

Para este modelo, el factor de ramificación efectivo en el peor caso queda tal que:

$$b = \frac{12\ nodos\ no\ raiz}{6\ nodos\ no\ hoja} = 2$$

Mientras que en el caso en el que el agente deduce que propagar la alerta es la opción viable (se explora hasta el nodo hoja más profundo posible), el cálculo del factor efectivo resulta:

$$b = \frac{14\ nodos\ no\ raiz}{7\ nodos\ no\ hoja} = 2$$

Constatando que, de forma análoga a los modelos anteriores, las situaciones más infructuosas posibles tanto en anchura como en profundidad no diferencian los órdenes máximos asumible, que quedan definidos de la siguiente forma:

- En el peor de los casos, su orden espacial optimizado sería de $O(2^4)$. Su factor de ramificación efectivo es de 2, y su profundidad es de 4 niveles.
- Su orden temporal sería de $O(2^1)$, pues el factor de ramificación efectivo es de 2 y la solución que se encuentra a menor profundidad está en el primer nivel.

7.7 Cuarta versión

7.7.1 Zombi Simple

En esta versión el árbol se mantiene estructuralmente igual que la versión 3, pero se realiza un cambio en la rama de persecución para poder integrar un sistema de coordinación. En este caso, la clase controladora del AlertManager se modifica para integrar la posibilidad de emplear o no el sistema de asignación de puestos. La figura 31 muestra el cambio en la rama correspondiente.

Figura 31: modificación del código

Detalle de la modificación en el código que permite la adquisición de un puesto en el nodo más próximo.

```
public override NodeState Evaluate() {
        ClearData("nextSlot");
        var lastAlertZone = AlertManager.GetLastZoneReportedInNodes();
        foreach (GameObject gameObject in lastAlertZone) {
                var slotsController = gameObject.GetComponent<SlotsController>();
                if (!slotsController.IsFull()) {
                        slotsController.TakeSlot(agent);
                        _parent.SetData("nextSlot", slotsController.gameObject);
                        state = NodeState.SUCCESS;
                        return state;
                }
        };

        state = NodeState.FAILURE;
        return state;
}
```

Este nuevo sistema permite que, en lugar de posiciones, los agentes se acercan a los nodos en función a cuentas posiciones o *slots* quedan disponibles. En esencia, cuando se activa el sistema de alarma y se posiciona al jugador, cada uno de los nodos informa a los agentes implicados en la propagación de la alarma de cuantas posiciones quedan disponibles, de forma que, si un nodo se llena, los agentes restantes no podrán ir hacia él, sino hacia otro de los restantes que quedan con huecos libres.

Con esto se pretende orientar a los enemigos hacia zonas donde *es posible* encontrar al jugador, en lugar de fijar su posición en los puntos donde se le avistó, buscando generar situaciones en las que el jugador se vea rodeado con más frecuencia. Este sistema de nodos se ha combinado con estrategias de posicionamiento, para disponer de más combinaciones para la parte de pruebas.

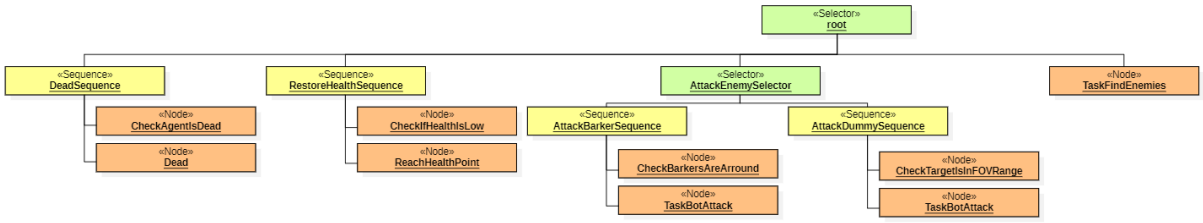
7.8 Robot de pruebas

Para las pruebas de los comportamientos se añadió un personaje de *sparring*, con sus clases de control correspondientes. Este robot implementará un árbol sencillo, cuyo objetivo será buscar y eliminar a los enemigos presentes en la escena de juego. Asimismo, una vez gane o muera, registrará los datos en un fichero de texto correspondiente.

En esencia, su objetivo es proveer de un interfaz que permita verificar la dificultad estimada de cada uno de los árboles montados durante el proyecto. A priori, posee rutinas sencillas de ataque, búsqueda de enemigos y selección de objetivos. La figura 32 muestra el esquema del árbol generado.

Figura 32: Árbol del Bot

Detalle del árbol del zombi simple en su cuarta versión. Nótese la modificación de la rama de alerta respecto del modelo previo



Al igual que sus homólogos, el árbol es bastante compacto. El conjunto de comportamientos permite que el robot muera, anotando los resultados del test, busca un punto restauración de salud para recuperarse en caso de que su salud se reduzca por debajo de un umbral, localiza y apunta hacia los enemigos en función a su tipo, y en defecto de cualquiera de las anteriores, se mueve por el mapa intentando localizar enemigos.

7.9 Comparativa de árboles

A modo de resumen, la tabla 2, muestra a modo ilustrativo un resumen entre los tres modelos y sus órdenes máximos, con optimización implementada y sin ella.

Tabla 2: Compartiva de órdenes de complejidad

Tabla comparativa de órdenes de complejidad de los modelos de agente implementados.

Modelo	**Sin optimización**		**Con optimización**	
	Espacial	**Temporal**	**Espacial**	**Temporal**
Dummy_V1	$O(2{,}5^2)$	$O(2{,}5^1)$	$O(1{,}75^2)$	$O(1{,}75^1)$
Dummy_V2	$O(2{,}5^3)$	$O(2{,}5^1)$	$O(1{,}66^3)$	$O(1{,}66^1)$
Barker_V1	$O(2{,}8^2)$	$O(2{,}8^1)$	$O(1{,}8^3)$	$O(1{,}8^1)$
Barker_V2	$O(2{,}37^3)$	$O(2{,}37^1)$	$O(1{,}66^3)$	$O(1{,}66^1)$
Barker_V3	$O(2{,}6^4)$	$O(2{,}6^1)$	$O(2^4)$	$O(2^1)$
Barker_V4	$O(2{,}6^4)$	$O(2{,}6^1)$	$O(2^4)$	$O(2^1)$

7.10 Estrategias de coordinación

Paralelamente al desarrollo de los árboles y del sistema de alerta, se diseñaron dos estrategias de coordinación para que los agentes localicen y ataquen al jugador. Ambas estrategias se centran en establecer una zona por donde el jugador pudo haber pasado una vez se ha dado la señal de alarma, diferenciándose únicamente en los puntos extraídos para determinar dicha zona.

7.10.1 Posicionamiento del jugador

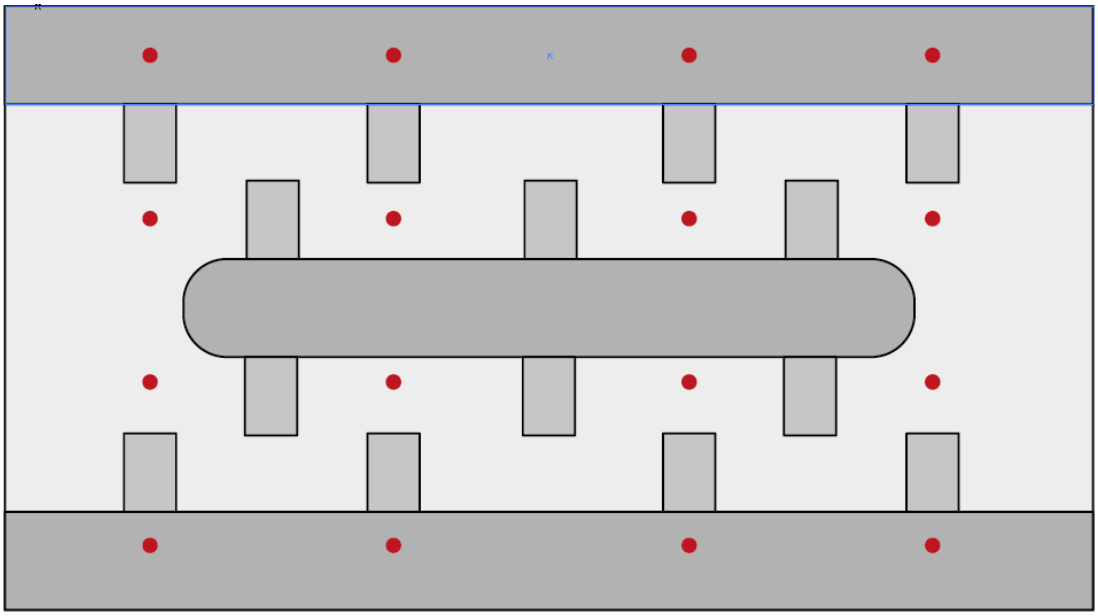
El método de localización empleado fue el de posicionamiento por nodos. Cuando el jugador provoca que un agente dispare una alerta, se registran en la instancia de *AlertManager* su última posición conocida y los nodos más próximos a dicha posición en forma de elementos tipo *Vector3*.

Cuando el resto de agentes susceptibles de ser alertados consultan los valores de ultima zona conocida, es decir, los nodos cercanos al jugador en el momento de la alerta. Cada agente establece a continuación su ruta de aproximación al jugador tomando como destino el nodo más próximo al que se encuentra, o si le es posible, la posición del jugador. A continuación, navega hacia ese punto empleando el componente NavMesh asociado, empleando el algoritmo A* ya descrito.

Con respecto al conjunto de nodos de posicionamiento, se optó por una disposición en forma de rectángulo, de cuatro por cuatro. Se respetaron las diferencias de altura entre las plataformas. La figura 33 muestra la vista cenital del esquema de descrito.

Figura 33: Mapa de nodos de detección

Detalle cenital de la disposición de conjunto de nodos de posicionamiento en el mapa. En color rojo aparecen indicados las posiciones de cada uno de los nodos.



7.10.2 Selección de los nodos

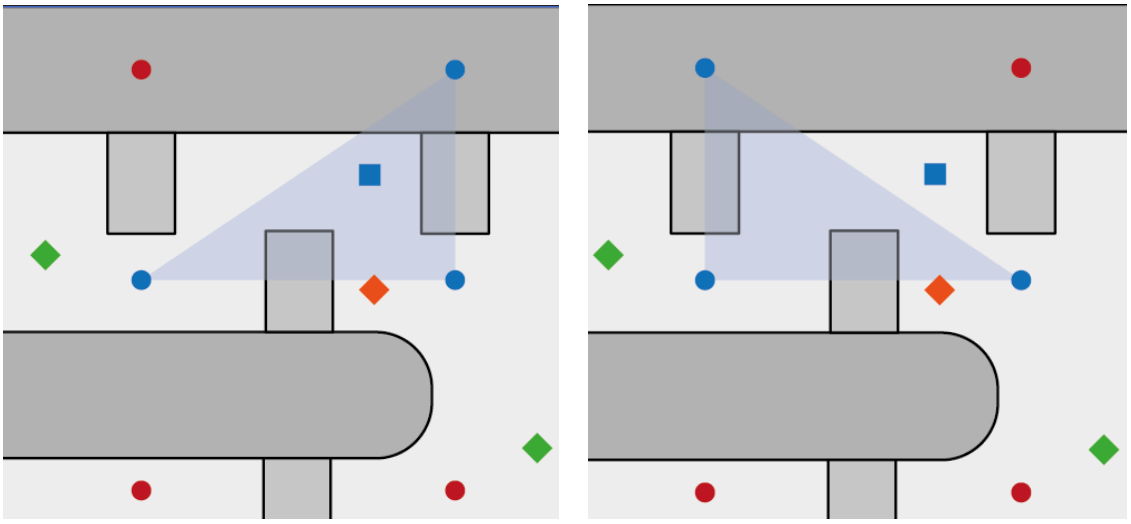
Para seleccionar los nodos se optó por dos posibles enfoques:

- Seleccionar los *n* nodos más cercanos al jugador. El área generada es más pequeña, pero facilita a los agentes la detección a corto alcance del jugador. Como desventaja, otras posibles vías de ataque quedan abiertas.
- Seleccionar los *n* nodos más alejados del jugador dentro de un radio de detección. En consecuencia, el área generada es mayor, lo que permite que los agentes puedan tomar rutas más amplias y cubrir zonas expuestas. Como contraparte, la detección del jugador es susceptible de complicarse en tiempo de juego.

La figura 34 muestra la representación gráfica de ambas detecciones sobre una de las esquinas del mapa cenital mostrado anteriormente. Se puede apreciar la diferencia entre el primer método de detección y el segundo.

Figura 34: Sistemas de detección

Detalle de los dos sistemas de detección de nodos. El área de detección se representa sombreada en azul.



Nota: Los rombos de color verde y naranja se corresponden con enemigos alertados. El cuadrado azul representa la posición del jugador.

La detección de los nodos en le momento de disparar la alerta se realiza empleando el método *Physics.OverlapSphere*¸ integrado dentro del motor de físicas. De acuerdo con la documentación oficial de Unity, computa y almacena los colisionadores existentes que toquen o se encuentren dentro del área generada por la esfera resultante de su invocación (Unity Technologies, 2022).

Para facilitar aún más este proceso, una de sus sobrecargas permite la inclusión de las máscaras de capa, o *LayerMasks*, que son una representación binaria de las capas empleadas en la escena. Ubicando los nodos de detección en una capa concreta y especificando al motor la máscara que debe rastrear conseguimos optimizar el rastreo de forma considerable.

Una vez se tienen los nodos, se extraen de ellos sus posiciones vectoriales llamando a los objetos *transform*. Éstas son almacenadas en una lista, para posteriormente ser ordenadas empleando LINQ.

LINQ, acrónimo de *Language-Integrated Query* es un conjunto de tecnologías orientadas a consulta integrado dentro del entorno de desarrollo de C# (Microsoft Corporation, 2022). Permite, entre otras funcionalidades, construir consultas sobre estructuras de datos comunes, como listas. Aprovechando esta utilidad, loa lista resultante de posiciones es ordenada usando como referencia el resultado de la función de distancia relativa entre dos vectores, ya

comentada en puntos anteriores. Esta sucesión de pasos se puede ver reflejada en el fragmento de código indicado en la figura 35.

Figura 35: Ordenamiento de nodos de posición

Detalle de la codificación del proceso de ordenamiento de los nodos de posicionamiento. En este caso, se muestra el correspondiente con el enfoque de cercanía.

```
private List<Vector3> TriangulatePlayerPosition(Vector3 position) {
       var layerMask = (1 << 16);
       var alertPoints = Physics.OverlapSphere(position, 30f, layerMask);
       alertPoints.OrderBy(x => Vector3.Distance(x.transform.position, position));
       List<Collider> filteredPoints = new List<Collider>(alertPoints);

       List<Vector3> lastZone = new List<Vector3>();
       foreach (var filteredPoint in filteredPoints.GetRange(0, 3)) {
              lastZone.Add(filteredPoint.transform.position);
       }
       return lastZone;
}
```

La secuencia *x => Vector3.Distance(x.transform.position, position)* es una función tipo lambda, en la cual el parámetro de la izquierda se corresponde con el valor de entrada y el valor de la derecha con el cuerpo de la función a aplicar para el ordenamiento, formalmente *cuerpo lambda* (Microsoft Corporation, 2022). En lengua vernácula, representa en código la sentencia *ordenar los objetos de mayor a menor de la colección según distancia vectorial a la posición especificada*. Con esa sencilla implementación los nodos quedan ordenados antes de filtrarse y ser registrados.

8 ASPECTOS DE IMPLEMENTACIÓN

Este apartado tratará acerca de los aspectos más importantes de la implementación del juego, tanto técnicos como de diseño. Se incluyen, entre otros:

- Gestión del flujo de Juego
- Detección con Raycasting.
- Pooling y Generación de Enemigos.
- Otros esquemas de diseño considerados relevantes.

8.1 Gestión del flujo de Juego

Gestionar el flujo de Juego conlleva verificar que las diferentes fases de la partida se van cumpliendo correctamente. Resumido brevemente, el flujo queda definido de la siguiente manera:

- La partida se inicia en la ronda 1. Los enemigos son generados y comienza el tiempo de juego.
- Si el jugador vence en la ronda actual, el sistema muestra un mensaje y el escenario se reinicia, generando una ronda nueva. El sistema puede definir si aumenta o no la dificultad en función a la ronda en la que se encuentra.
- Si, por el contrario, el jugador muere durante una de las rondas, aparece una pantalla de finalización y puede escoger reiniciar la partida, reiniciando o, en su caso, volver al menú principal.

8.1.1 Inicio de la ronda

En esta fase del flujo, los siguientes subsistemas quedan implicados:

- *RoundManager*: Gestiona todos los subsistemas generales de la ronda, como la puntuación y los enemigos restantes.

- *EnemyGenerator*: Controla las instacnias de enemigos que deben ser creadas en la partida.
- *SpawnManager*: Inicia la creación de los enemigos que van a aparecer en la partida.Desarrollo de la ronda.

8.1.2 Desarrollo de la ronda

En esta fase cobran protagonismo los siguientes componentes:

- *AlertManager*: Controla las alertas y la propagación de éstas durante la ronda, así como loa posición del jugador.
- *HealthController*: Gestiona la salud del jugador durante la ronda.
- *WeaponController*: Gestiona el control de armas durante la ronda
- *EnemyController*: Gestiona la actualización de las estructuras de Inteligencia Artificial asociada y otros aspectos de los agentes

8.1.3 Finalización de la ronda

Al finalizar la ronda, puede repetirse el flujo de reinicio de ronda o dar por concluida la partida, por lo que sólo contaremos al *RoundManager* como principal implicado.

8.2 Raycasting

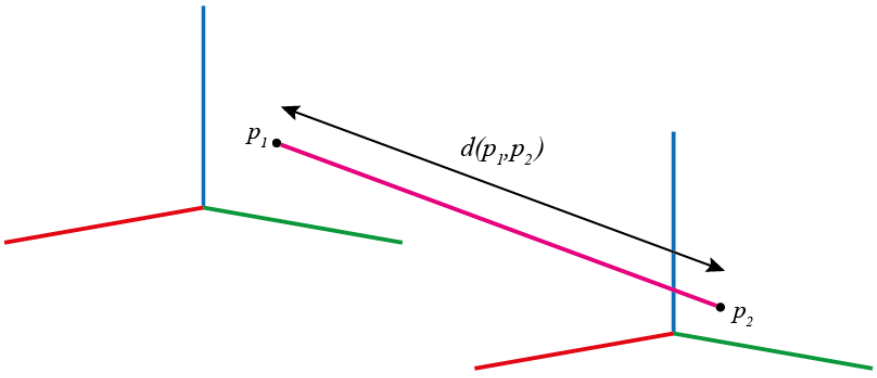
Raycasting (traza de rayos) es un método para construcción de gráficos por computadora y renderizado de modelos 3D. Permite el renderizado por proximidad de los objetos de la escena de juego, renderizando la imagen desde el punto de vista del observador. (Wikipedia, 2022)

8.2.1 Enfoque matemático

En esencia, la construcción de un rayo en el espacio vectorial que conforma la escena de juego emplea conceptos de cálculo vectorial en tres dimensiones. En el momento en el que rayo es trazado, los puntos de referencia p_1 y p_2 son recuperados y se calcula la distancia vectorial entre éstos. La figura 36 muestra un diagrama sencillo del cálculo entre ambos puntos.

Figura 36: Distancia vectorial

Detalle del cálculo de la distancia vectorial entre dos puntos en espacios 3D.



La distancia resultante del cálculo viene definida por la distancia euclidiana clásica:

$$d(y,x) = \sum_{i=1}^{n} \sqrt{(x_i - y_i)}$$

Que aplicada en entre dos puntos tridimensionales queda de la siguiente forma:

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Este sencillo cálculo permite calcular distancias obtenidas de trazar rayos desde un punto de vista concreto sin que suponga un sobrecoste en el procesamiento de la escena.

8.2.2 RayCasting en Unity

El motor de físicas de Unity permite la generación de rayos de forma dinámica durante el tiempo de juego. Añadido a esto, los datos de la colisión del rayo pueden ser almacenados si así se estima oportuno.

Mediante el uso la función *Physics.Raycast* (Unity Technologies, 2022) en cualquiera de sus sobrecargas, es posible obtener datos desde el punto de mira que se tome como base del trazado. Permite además la detección de colisionadores durante el trazado.

Entre los posibles parámetros a inyectar es preciso destacar la distancia máxima a la que se quiere aplicar el rayo y las capas de la escena a las que puede afectar. Ajustando estos parámetros es posible discretizar qué capas se han de tener en cuenta y cuál es la distancia máxima sobre la cual se tendrán en cuenta los resultados del trazado.

8.2.3 Aplicación en el proyecto

Por su versatilidad y facilidad de uso, se consideró usar el trazado de rayos en varias características del modelo del jugador, siendo la más notoria el mecanismo de disparo. Cuando el jugador invoca la acción de disparar pulsando el botón primario del ratón, el motor realiza un trazado de longitud preestablecida partiendo del centro de la posición hacia donde se encuentra enfocada la cámara. La longitud queda definida en las propiedades del arma que está empuñándose en el momento del trazado.

Este proceso puede verse reflejado en código en la figura 37. Cuando se ha finalizado el trazado, el motor devuelve el resultado en una variable de salida tipo *out*. Si se ha golpeado a un enemigo, se extrae el componente tipo *EnemyController* correspondiente y se aplica el daño llamando al método *ApplyDamage*. Si, por el contrario, el impacto ha sido contra otro elemento, se reproduce una pequeña animación de impacto.

Figura 37: Método RaycastShot()

Método RaycastShot para el control de los impactos en el juego.

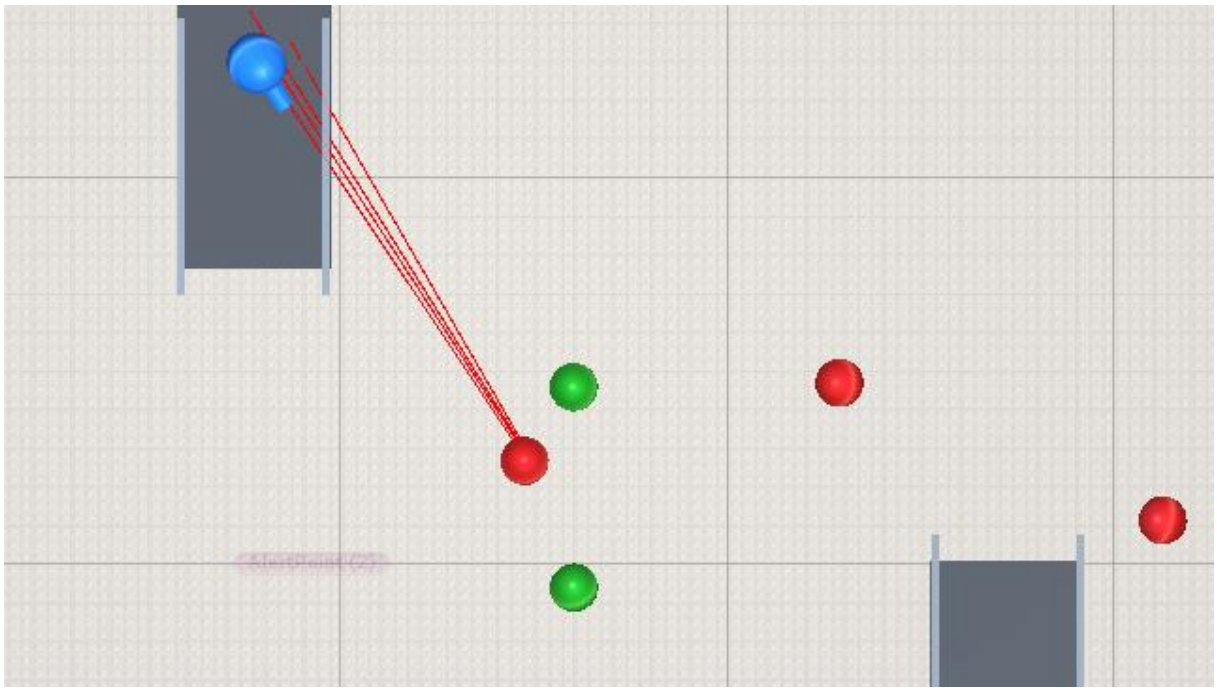
```
private void RaycastShot() {
       if (Physics.Raycast(pointingCamera.position, pointingCamera.forward,
                                  out impactInfo, weaponRange)) {

              Ray ray = new Ray(
                     pointingCamera.position,
                     pointingCamera.forward);
              Debug.DrawLine(ray.origin, impactInfo.point, Color.red, 0.45f);

              if (impactInfo.collider.tag.Equals("Enemy")) {
                     PlayShootAnimation();
                     ApplyDamageOnTarget(impactInfo);
              } else {
                     PlayImpactAnimation();
              }
       }
}
```

Este método también se aplicó en la interacción con los objetos de la escena. Por ejemplo, cuando el jugador necesita curarse, la cámara debe apuntar específicamente a un kit médico a una distancia determinada para que se active el efecto de curación. El mismo sistema se aplicó para gestionar los disparos del robot, como se puede ver en la figura 38

Figura 38: Muestra del disparo por Raycasting



8.3 Pooling

8.3.1 Instanciar vs Activar

A la hora de gestionar la optimización de recursos dentro de la escena de juego, es preciso entender que la creación y destrucción de una nueva instancia de un elemento no solo conlleva el consumo de una serie de recursos del motor, también genera consecuencias no deseables en la memoria.

La más notoria de éstas es el fenómeno conocido como *fragmentación*. Es la partición de la memoria disponible en fragmentos, cada uno de ellos asociados con algún elemento que se usa en tiempo de cómputo. Si no se gestiona bien puede provocar problemas severos de consumo de memoria en un juego, lo cual afecta no solo a la optimización del sistema en sí, sino también al rendimiento del propio juego.

La fragmentación se produce principalmente cuando se instancian y destruyen objetos de juego. Debido en gran medida a que la infraestructura principal ya ha sido cargada en memoria y el espacio reservado restante se empleará para las escenas que integra el juego y lo elementos que contenga.

Sabido esto, supóngase, por ejemplo, que se quieren generar tres instancias de enemigos, dos de un tipo A y una restante de un tipo B. El tipo A consume unos 15kB de memoria y el B 20kB. Si tuviésemos un espacio de 45kB de memoria reservada para objetos, puede darse la siguiente situación:

- Si se cargan los dos objetos A, la memoria queda ocupada casi al 60%. Suponemos que la asignación es al primer hueco libre.
- Si se destruye la primera instancia, la memoria queda fragmentada en tres trozos, uno de 15kB libres, otro del mismo tamaño ocupado por la segunda instancia A y otro de 15kB desde el final de ésta hasta la última posición en memoria disponible.

Como se puede intuir, si se desease instanciar el objeto tipo B, habría que hacer una redistribución o reservar más memoria para el programa, ya que ocupa más espacio del que los fragmentos disponibles pueden ofrecer. Ambas soluciones son ineficientes, puesto que ocupan espacio de memoria de forma innecesaria o porque el consumo de redistribuir periódicamente los fragmentos de memoria puede ser inasumible según la plataforma en la que se trabaje.

Para solventar este problema se aplica el patrón *Object Pool*. Este patrón propone instanciar en memoria todos los objetos que vayan a ser empleados en las escenas durante el

proceso de carga del juego, e ir activándolos y desactivándolos conforme se necesite. Dado que activar y desactivar son procesos más baratos en términos computacionales, realizar la carga de los objetos una única vez no solo resuelve el inconveniente de la fragmentación, sino que evita la asignación innecesaria de recursos al proceso de carga, ya que son empleados una única vez.

8.3.2 Pooling en el proyecto

El patrón *Object Pool* se ha implementado principalmente en la clase *SpawnManager*, encargada de generar las instancias de enemigos durante las rondas de juego. (Nystrom, 2014) define los siguientes casos como adecuados para la implementación del patrón:

- Se necesita crear y destruir objetos frecuentemente.
- Los objetos son similares en tamaño.
- Alojar objetos en la memoria se realiza de forma lenta o puede provocar fragmentación.
- Cada objeto encapsula un recurso, como una conexión a una base de datos o de red que es costosa de adquirir y podría reusarse. (pág. 307)

La clase mencionada cumple con tres de los puntos indicados en la lista, pues crea y destruye constantemente enemigos, todos de similares características y cargarlos o destruirlos puede provocar fragmentación a la larga.

Por ello, se creó un generador de enemigos, que otorga al *SpawnManager* la lista de enemigos a instanciar y en la primera ronda, cuando el método *SpawnEnemies* es llamado, verifica si las instancias han sido creadas o desactivadas, aplicando el procedimiento necesario para ello. La figura 39, muestra un detalle el código implicado en el pooling de los enemigos.

Figura 39: Método RespawnEnemies()

Detalle del método RespawnEnemies, que implementa el patrón Object Pool.

```
public void RespawnEnemies(int enemyLifePoints, int enemyDamage, int roundsPlayed) {
       if (!enemiesHaveBeenCreated) {
              switch (PlayerPrefs.GetString("difficulty")) {
                     case "1":
                            SpawnEasyMode();
                            break;
                     case "2":
                            SpawnMediumMode();
                            break;
                     case "3":
                            SpawnHardMode();
                            break;
              }
              enemiesHaveBeenCreated = true;
              ActivateEnemies(enemyLifePoints, enemyDamage, roundsPlayed);
```

```
        } else {
                ReloadPositions();
                ActivateEnemies(enemyLifePoints, enemyDamage, roundsPlayed);
        }
}
```

Una vez instanciados, los objetos son almacenados en una lista interna de la clase, que es consultada en las funciones *ReloadPositions* y *ActivateEnemies*, en los cuales se reinician las posiciones de los enemigos y sus propiedades, respectivamente.

8.4 Generación de enemigos

Disponer de diferentes niveles de dificultad exige de un mecanismo que permita controlar qué enemigos van a ser creados en función a qué nivel de dificultad. Una vez resuelto el sistema de generación usando *Object Pooling*, se añadió una clase intermedia encargada de proveer al *SpawnManager* de los objetos de tipo enemigo que tiene que instanciar.

8.4.1 PlayerPrefs y la clase EnemyGenerator

PlayerPrefs, abreviatura de *Player Preferences*, es una clase que almacena los datos correspondientes a las preferencias del jugador entre sesiones de juego. El almacenamiento se realiza en función al sistema operativo en el que se ejecuta el juego (Unity Technologies, 2022).

En esencia, dispone de un diccionario vacío en el que se pueden realizar las operaciones clásicas. Las clases que integran el juego pueden acceder a este en cualquier momento de la ejecución llamando a los métodos *get* y *set* asociados.

En el proyecto, la interacción queda establecida en el menú de opciones. Cuando el jugador cambia el nivel de dificultad, se actualiza la clave "*difficulty*" en el diccionario con el valor asociado a la opción seleccionada. En el momento en el que se inicia la partida, durante el proceso de carga de la ronda, el *EnemyGenerator* obtiene este valor consultando al *PlayerPrefs* y provee a *SpawnManager* de la combinación de enemigos que debe instanciar. Finalmente, este último crea el pool de enemigos para la nueva partida.

Aunque tosco, este sistema provee de una solución eficaz y extensible si se plantean varios niveles de complejidad en los agentes. Únicamente se ha de añadir al generador la nueva combinación y al *SpawnManager* el método para generar enemigos.

8.5 Sistema de Pausa

Se consideró esencial que el jugador pudiera pausar el juego durante la ejecución de la ronda. Por ello se añadió un sistema de pausa. Integrado en el *GameManager*, que es una clase destinada a gestionar aspectos de infraestructura del juego, como, por ejemplo, la selección de escenas o las pausas de ejecución.

8.5.1 Estructura

El sistema de pausa se inicia en el *GameManager*. Cuando se pulsa la tecla *P*, el *GameManager* ordena al *RoundManager* que pause la instancia de juego. La variable de escala de tiempo *Time.TimeScale* se establece a cero. A continuación, el Round Manager ordena a todos los agentes que pausen su ejecución, e intercambia el canvas de juego por otro de pausa. Mostrando una pantalla de pausa, con la opción de volver al menú principal o continuar jugando. Si el jugador quita la pausa, todo el proceso es revertido y el juego vuelve a su estado original de juego.

8.6 Diseño del HUD

Al tratase de un desarrollo en un plazo muy comprimido de tiempo, el diseño del HUD es sobrio, fácil de entender para el jugador y que contenga los principales datos que éste necesita saber durante el desarrollo de las partidas, y cuáles puede modificar.

8.6.1 Menú principal y menú de opciones

Para el menú principal, la interfaz gráfica se divide en dos pantallas, la pantalla inicial, que contiene los botones de inicio de partida y acceso al menú de opciones, y la pantalla de opciones, que abarca los elementos que pueden ser modificados por el jugador de cara a personalizar su experiencia de juego, que para este proyecto son el volumen general, la dificultad y la sensibilidad del movimiento del ratón. Las figuras 40 y 41 muestran una comparativa de ambos.

Figura 40: Menú principal

Figura 41: Menú de opciones

8.6.2 Escena de juego

Para la escena de juego, los siguientes datos se encuentran disponibles durante el desarrollo de la partida, que pueden contrastar en la figura 42:

- Marcador de vida del jugador: se actualiza con cada impacto o recuperación de salud del jugador. Ubicado en la esquina inferior izquierda de la pantalla.
- Marcador de enemigos: se, actualiza con el número de enemigos que quedan vivos en la partida. Ubicado en el centro de la parte superior de la pantalla.
- Marcador de munición: Se ajusta a cada arma en función de la munición que queda disponible. Ubicado en la e esquina inferior izquierda de la pantalla.
- Marcador de ronda: Indica la ronda actual en la que se encuentra el jugador. Se actualiza conforme el número de rondas que se jueguen.
- Puntero: Indica hacia donde apunta el jugador actualmente.

Figura 42: Pantalla de juego

9 CONCLUSIONES

9.1 FSM vs Arboles

9.1.1 FSM vs BT

La comparativa entre las máquinas de estado y los árboles de comportamiento resulta en las siguientes conclusiones:

- A nivel arquitectónico, el árbol de comportamiento sobrepasa en todos los sentidos a la máquina de estados. La simpleza de su estructura del otorga una capacidad de ser escalado inalcanzables por la máquina de estados una vez se alcanza determinado nivel de complejidad.
- En lo relativo a la optimización, el árbol de comportamiento nuevamente supera a la máquina de estados. Como ya se ha visto, por la propia morfología de éstas, desplegar en forma de grafo una máquina de estados puede generar un árbol con bucles, lo que fuerza a seleccionar otros algoritmos de búsqueda más complejos para ese tipo de estructuras. Asimismo, los ajustes hechos a los nodos compuestos en los árboles construidos en este proyecto posibilitan mantener órdenes de complejidad cercanos a la unidad.
- En el plano conceptual, resulta más sencillo entender el despliegue de comportamientos de un agente como la suma de resoluciones lógicas dentro de un conjunto establecido de posibilidades. Siguiendo el patrón "*si no, entonces*" la propia jerarquía del árbol emana de forma casi automática. Ello puede facilitar el desarrollo al programador, así como agilizar los procesos de análisis y diseño en proyectos más grandes.

Esto no significa que las máquinas de estado sean inútiles, sino todo lo contrario. La debilidad de éstas se manifiesta en el momento de introducir mayor complejidad o vías alternativas de resolución a un comportamiento específico. Si la aplicación que se plantea se basa en patrones muy cerrados y concretos de comportamiento, es preferible, incluso recomendable, implementar una MEF antes que un árbol.

Resumiendo, este punto, ambas estructuras son útiles en su contexto. Si el agente a implementar tiene un rango de acciones corto, es muy recomendable implementar una MEF. Si el sistema plantea varios tipos de agentes y comportamientos, es cuasi imperativo implementarlos como árboles.

9.1.2 Comunicación vs Aislamiento

En lo que a jugabilidad se refiere, este proyecto ha contado con dos aproximaciones de interacción entre los agentes, una enfocada en agentes aislados, sin contacto entre ellos, y otra en la que los agentes comparten un sistema de comunicación que les permite tomar diferentes comportamientos. Después de las pruebas las conclusiones se pueden resumir en:

- La dificultad de los agentes que no interactúan entre ellos genera situaciones en las que el conjunto se mueve como una horda. El jugador puede actuar intentando evitar la horda y así acabar con los enemigos. Las posibilidades de que evada situaciones comprometidas son elevadas.
- En el caso en el que los agente si interactúan entre sí, permite que el jugador se vea rodeado por varios agentes y, por tanto, con más dificultades a la hora de evadir situaciones de flanqueo o rodeo. Esto reduce de forma sistemática las ocasiones en las que el jugador puede evitar situaciones comprometidas.

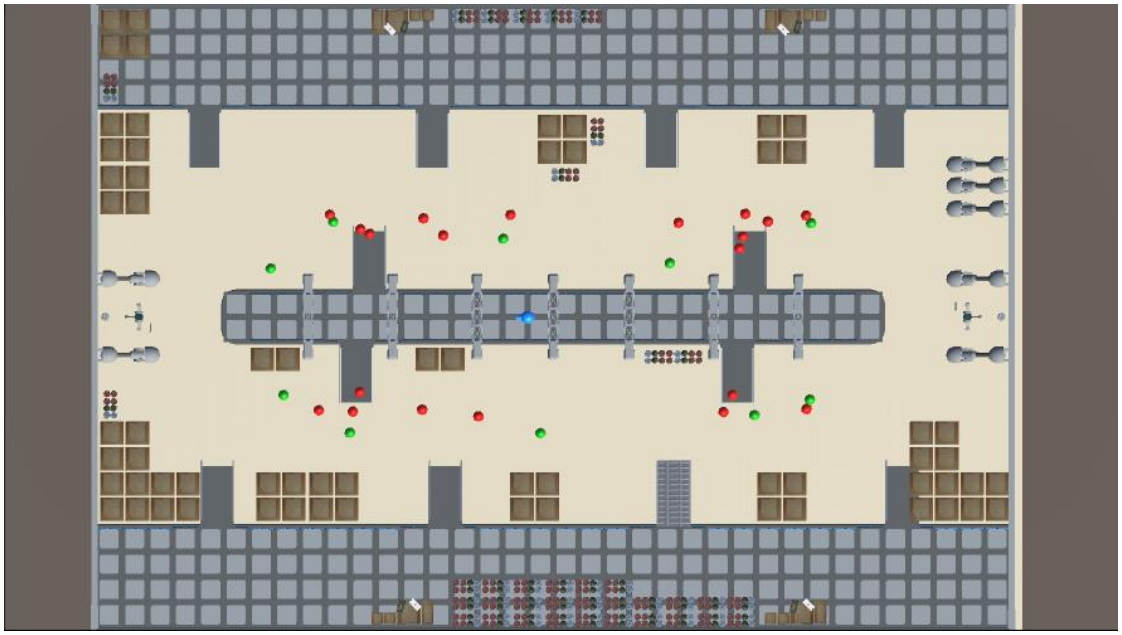
9.1.3 Pruebas del nivel de dificultad

La verificación de los diferentes niveles de dificultad se hizo en entornos de prueba simulado, como se muestra en la figura 43. Para cada entorno se siguieron las siguientes reglas:

- Una instancia de robot se despliega en uno de los puntos de aparición establecidos.
- Busca por el mapa a los enemigos presentes, debiendo curarse si sus puntos de vida baja por debajo de un umbral.
- Una vez que el robot hay finalizado su objetivo, guarda los resultados de la simulación en un fichero de texto.
- Si el robot muere durante el desarrollo de la partida, guardará de igual forma los resultados en el fichero de simulación.

Figura 43: Muestra del mapa de pruebas, segunda versión

En color ver se muestran los zombis aulladores, en color rojo los zombis simples. En azul el robot de pruebas.



Cada prueba se repitió entre 35 y 40 iteraciones, con los mismos parámetros para el jugador, detallados en la tabla 3.

Tabla 3: Parámetros del jugador para las baterías de pruebas.

Puntos de Salud	Velocidad	Daño	Ratio de disparo (Disparos/Segundo)	Rango de Visión	Rango de Ataque
200	7 Uds.	15	0.1	15 Uds.	30 Uds.

De cara a presentar un análisis concreto y verificar el aumento progresivo de la dificultad entre las diferentes versiones, los siguientes datos fueron puestos bajo observación:

- Impactos recibidos de promedio por el total de las partidas.
- Daño total recibido por partida. Contabiliza el daño total recibido durante el trascurso de la partida.
- Curaciones promedio realizadas durante el trascurso de la batería de pruebas.
- Número de enemigos derrotados durante la partida.
- Promedios totales de victorias y derrotas.

9.1.4 Análisis de los resultados

Una vez establecidos en la subrutina de muerte del robot, los resultados obtenidos para las tres baterías de pruebas arrojaron conclusiones interesantes para el desarrollo y posterior ampliación de la inteligencia artificial del juego. Las tablas 4 y 5 muestran de forma resumida los valores obtenidos para cada uno de los parámetros puestos al análisis.

Tabla 4: Resultados de las pruebas

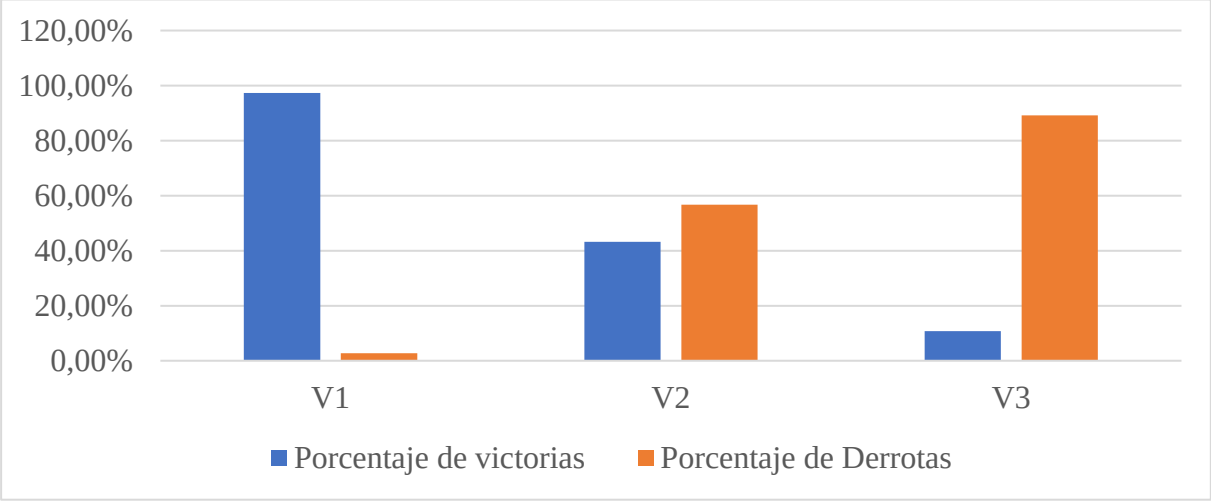
	Daño total recibido	**Impactos**	**Enemigos derrotados**	**Curaciones promedio**
V1	40,54	4,05	29,35	0,14
V2	238,78	18,68	18,16	0,95
V3	224,05	18,84	8,89	0,41

Tabla 5: Ratios de victorias y derrotas para cada batería de pruebas

	Porcentaje de victorias	**Porcentaje de Derrotas**
V1	97,30%	2,70%
V2	43,24%	56,76%
V3	10,81%	89,19%

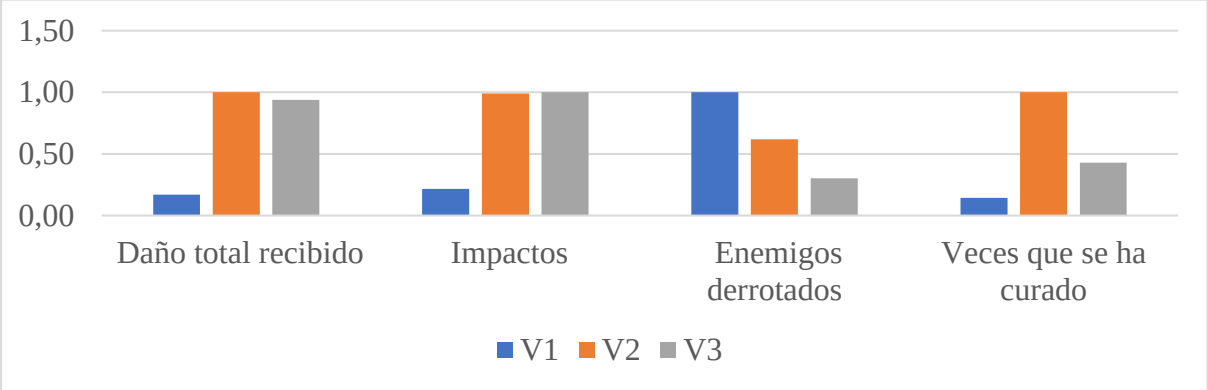
Como se puede observar, la versión básica no supone un reto para el robot de pruebas, puesto que obtiene la victoria en casi la totalidad de las partidas. En cambio, conforme se ve expuesto a inteligencias más complejas, los resultados en la tasa de victorias se resienten severamente. En el caso de la segunda versión, donde ya existe coordinación entre los enemigos aulladores, el porcentaje de victorias se reduce hasta quedar por debajo del 50%. Este descenso es más notorio en la tercera versión, donde la tasa de victorias sobrepasa ligeramente el 10% del total de partidas ejecutadas. La figura 44 muestra de forma gráfica la diferencia entre las tres versiones.

Figura 44: Relación porcentual de victorias/derrotas.



Sin embargo, el detalle más significativo reside en los parámetros intrínsecos a la partida. A primera vista puede parecer confusa la relación que hay entre el número de impactos recibidos y el daño total recibido en la partida. En teoría el daño recibido debería ser menor en la segunda versión con respecto a la tercera, y, de forma análoga el número de impactos debería ser mayor en la tercera versión con respecto a la segunda. La figura 45 muestra la diferencia entre los valores normalizados respecto de su máximo valor local.

Figura 45: Resultados normalizados de las simulaciones



Si se descartan los enemigos eliminados de promedio por partida, el factor que resta por analizar es el número de curaciones que se realizan por partida. Como se observa, el robot es capaz de realizar casi una curación por partida, mientras que en el segundo caso apenas logra una décima parte de ese valor. Esto supone que en la tercera versión el robot se ve sometido de forma más frecuente a situaciones en las que escapar se torna complicado, a debido a que se encuentra rodeado, o en su defecto, cercado por enemigos.

10 AMPLIACIONES FUTURAS

10.1 Mejoras en el sistema actual

10.1.1 Sistema de alerta

El sistema de alerta actual es ligeramente rudimentario. En su versión más compleja se basa en la asignación de slots a las posiciones adyacentes al agente. Aunque se contemplan varias pruebas en el desarrollo, lo cierto es que el sistema es mejorable.

Entre las posibles mejoras se plantean:

- Combinar el sistema de propagación general con otro de corte local, que afecte a los agentes en un radio.
- Disgregar los comportamientos de los agentes de forma que no se vean únicamente forzados a perseguir ni a atacar. En esencia sería ampliar aún más el rango de comportamientos.
- Añadir algún extra que permita simular al jugador una alarma. Por ejemplo, una granada que emita ruido, de forma que los agentes puedan verse atraídos hacia ella.

10.1.2 Enemigos disponibles

Actualmente existen dos enemigos disponibles, el zombi simple y el aullador. En principio se han esbozado dos más que, por motivos de tiempo, no se han podido implementar.

- El primero introduciría una mecánica de sigilo, siendo visible únicamente a determinada distancia del jugador. No se vería afectado por el sistema de alerta, sino que atacaría cuando el jugador no este apuntando hacia él. Mientras es invisible, el jugador solo puede detectarlo por el ruido que hace al caminar y aproximarse a él.
- El segundo sería una versión ligeramente más grande del zombi simple, aunque se está viendo si debería o no verse afectado por el sistema de alerta. Introduciría una mecánica de ataque a distancia, con proyectiles lanzados parabólicamente hacia el jugador y que generan daño en un área una vez que explotan.

10.1.3 Control de animaciones

El control de animaciones está distribuido sobre las estructuras de control de los jugadores. Sin embargo, este aspecto puede mejorarse considerablemente. Unity proporciona la posibilidad de delegar el control al componente de animación usando estructuras tipo

BlendTrees. Una posible ampliación podría tomar esta vía, verificando cómo tomar la posición del agente y suministrársela a su componente Animator para que gestione las transiciones entre animaciones.

10.2 Inclusión de Inteligencia artificial

10.2.1 ML-Agents

Unity ML, abreviatura de Unity Machine Learning, es un proyecto open-source que integra conjunto de herramientas que permite desarrollar agentes automatizados empleando técnicas de aprendizaje por refuerzo, imitación o neuroevolución, entre otros (Unity Tecnologies, 2021).

Dispone de modelos ya entrenados de redes neuronales, que pueden emplearse para entrenar a los agentes que vayan a ser incluidos en el juego. Éstos están soportados por el Motor de Inferencia de Unity, que usa sombreadores de computación para hacer funcionar los modelos en el motor (Unity Technologies, 2022). A bajo nivel, el proyecto está construido en base una API en Python, que proporciona las instrucciones y la infraestructura necesaria para realizar las tareas de entrenamiento de los modelos integrados en los agentes (Unity Technologies, 2021).

10.2.2 Agentes entrenados por refuerzo

Este enfoque dentro del campo de la IA comprende, como dice su nombre, una serie de estrategias por las cuales una red neuronal obtiene conocimiento a través de la repetición iterativa de una prueba en un contexto determinado. En esencia, lo que se pretende es que una vez que el modelo haya finalizado el período de entrenamiento sea capaz de afrontar situaciones similares de la manera en la que se ha ido entrenando, cometiendo la menor cantidad de errores posible.

Las posibles ampliaciones y mejoras que se han planteado para este aspecto son las siguientes:

- Integrar los comportamientos de los agentes dentro de modelos y entrenarlos
- Probar los modelos entrenados en entrono de juego
- Comparar los resultados y ver si resulta más eficiente desarrollar la IA de un juego usando arboles prediseñados o machine learning.
- Optimizar los modelos originales y ver si es posible obtener mejores tiempos de entrenamiento para un mismo conjunto de comportamientos.

GLOSARIO DE TERMINOS

1. Agente: Entidad de computación que interacciona de forma autónoma con el entorno y razona sobre qué acciones son más convenientes a realizar dado un contexto concreto.
2. BT: Abreviatura de *Behavior Tree*, traducción de Árbol de Comportamiento. Estructura lógica arbórea, que se contiene los comportamientos integrados por los agentes. Es explorado empleando un algoritmo de búsqueda.
3. FSM: Abreviatura de *Finte State Machine*, traducción de Máquina de Estados Finitos. Estructura lógica que contempla una serie de transiciones y estados, los cuales son consultados si se dan las condiciones adecuadas.
4. HUD: Abreviatura de *Head-Up Display*. es la información que aparece visible en pantalla y que proporciona al jugador datos sobre el estado actual de la partida y de su avatar en esta.
5. PVE: Abreviatura de *Player versus Environment*, traducción de Jugador contra entorno. Es un formato en el que el jugador se ha de enfrentar a los obstáculos que se plantean en el entorno del juego.
6. PVP: Abreviatura de *Player versus Player*, traducción de Jugador contra Jugador. Es un formato en el que el jugador se ha de enfrentar a otros jugadores y, opcionalmente, a obstáculos que se plantean en el entorno del juego.
7. RTS: Género en el que la mecánica principal consiste en vencer a un enemigo o grupo de enemigos comandando una serie de unidades en tiempo real, pudiendo gestionar recursos en el proceso.
8. Shooter: Género de en el que la mecánica principal rota alrededor de disparar a una serie de objetivos empleando un conjunto de armas.
9. SMA: Sistema Multi-Agente, sistema compuesto por varios agentes que interaccionan entre sí compartiendo información.
10. TBS: Género en el que la mecánica principal consiste en vencer a un enemigo o grupo de enemigos comandando una serie de unidades a lo largo de una serie limitada de turnos, pudiendo gestionar recursos en el proceso.

REFERENCIAS

.NET foundation. (2022). *xUnit.net*. Recuperado el 23 de abril de 2022, de https://xunit.net/

Atari Inc. (1972). Pong [Videojuego].

Atari Inc. (1973). Gotcha [Videojuego].

Aversa, D., Kyaw, A. S., & Peters, C. (2018). *Unity Artificial Intelligence Programming.* Packt Publishing Ltd.

Bethesda Softworks. (2020). DOOM Eternal (Versión para Computadora) [Videojuego].

Cossu, S. (2021). *Beginning Game AI with Unity.* Apress Media LLC.

Driessen, V. (5 de enero de 2010). *A successful Git branching model*. Recuperado el 10 de marzo de 2022, de https://nvie.com/posts/a-successful-git-branching-model/

From Software. (2011). Dark Souls (Versión para computadora) [Videojuego].

From Software. (2015). Bloodborne (Versión para computadora) [Videojuego].

From Software. (2019). Sekiro: Shadows Die Twice (Versión para computadora) [Videojuego].

Guerrilla Games. (2020). Horizon Zero Dawn (Versión para videoconsolas) [Videojuego].

Hernández, M. (2021). *Representación y Control en la Resolución de Problemas, IV Parte.*

Hmong.es. (s.f.). *Factor de Ramificación*. Recuperado el 2022 de mayo de 27, de El factor de ramificación promedio se puede calcular rápidamente como el número de nodos no raíz (el tamaño del árbol, menos uno; o el número de bordes) dividido por el número de nodos no hoja (el número de nodos con hijos).

Infinity Ward. (2009). Call of Duty: Modern Warfare 2 (Versión para Computadora) [Videojuego].

Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall.

MicroProse. (1991). Civilization (Versión para Computadora) [Videojuego].

Microsoft Corporation. (25 de mayo de 2018). *IntelliSense in Visual Studio*. Recuperado el 2022 de abril de 4, de Microsoft Docs: https://docs.microsoft.com/es-es/previous-versions/visualstudio/visual-studio-2017/ide/using-intellisense?view=vs-2017

Microsoft Corporation. (5 de mayo de 2022). *Find code changes and other history with CodeLens*. Recuperado el 12 de junio de 2022, de Microsoft Docs: https://docs.microsoft.com/es-es/previous-versions/visualstudio/visual-studio-2017/ide/find-code-changes-and-other-history-with-codelens?view=vs-2017

Microsoft Corporation. (05 de mayo de 2022). *Language Integrated Query (LINQ) (C#)*. Recuperado el 31 de mayo de 2022, de https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/concepts/linq/

Microsoft Corporation. (06 de abril de 2022). *Operador =>(referencia de C#)*. Recuperado el 31 de mayo de 2022, de https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/lambda-operator

Microsoft Corporation. (6 de junio de 2022). *Paseo por el lenguaje C#*. Recuperado el 8 de junio de 2022, de Microsoft Docs: https://docs.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/

Namco Ltd. (1980). Pac-Man [Videojuego].

Nystrom, R. (2014). *Game Programming Patterns.* Genever Benning.

Paradox Development Studio. (2013). Europa Universalis IV (Version para computadora) [Videojuego].

Paradox Development Studio. (2016). Hearts Of Iron IV (Version para computadora) [Videojuego].

Rockstar North. (2015). Grand Theft Auto V (Versión para computadora) [Videojuego].

Schwaber, K., & Sutherland, J. (noviembre de 2020). The Scrum Guide. Recuperado el 13 de mayo de 2022, de https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Spanish-European.pdf

Unity Technologies. (2 de diciembre de 2021). *Unity ML-Agents Python Low Level API*. Recuperado el 1 de junio de 2022, de ML-Agents: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Python-API.md

Unity Technologies. (18 de enero de 2021). *Unity ML-Agents Toolkit*. Recuperado el 2022 de junio de 3, de ML-Agents: https://github.com/Unity-Technologies/ml-agents

Unity Technologies. (29 de 04 de 2022). *Colliders Overview*. Obtenido de Unity Documentation: https://docs.unity3d.com/es/2018.4/Manual/CollidersOverview.html

Unity Technologies. (21 de mayo de 2022). *Physics.OverlapSphere*. Recuperado el 31 de mayo de 2022, de Unity Documentation: https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Physics.OverlapSphere.html

Unity Technologies. (7 de mayo de 2022). *Physics.Raycast*. Recuperado el 2022 de mayo de 14, de Unity Documentation: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

Unity Technologies. (3 de junio de 2022). *PlayerPrefs*. Recuperado el 5 de junio de 2022, de Unity Documentation: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

Unity Technologies. (24 de marzo de 2022). *Sistema de Navegación de Unity*. Obtenido de Unity Documentation: https://docs.unity3d.com/es/2020.2/Manual/nav-NavigationSystem.html

Unity Technologies. (24 de marzo de 2022). *Unity Documentation*. Obtenido de Sistema de Navegación de Unity: https://docs.unity3d.com/es/2020.2/Manual/nav-NavigationSystem.html

Unity Technologies. (7 de enero de 2022). *Unity Inference Engine*. Recuperado el 30 de mayo de 2022, de ML-Agents: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Unity-Inference-Engine.md

Unity Technologies. (2022). *Unity Test Framework*. Recuperado el 5 de mayo de 2022, de https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html

Unity Tecnologies. (8 de noviembre de 2021). *About ML Agents*. Recuperado el 3 de junio de 2022, de Unity Technologies: https://docs.unity3d.com/Packages/com.unity.ml-agents@2.0/manual/index.html

Valve Corporation. (2007). Team Fortress 2 (Versión para computadora) [Videojuego].

Valve Corporation. (2009). Left 4 Dead 2 (Versión para Computadora) [Videojuego].

Valve Corporation. (2011). Portal 2 (Versión para Computadora) [Videojuego].

Wikipedia. (9 de enero de 2022). *Algoritmo de búsqueda A**. Obtenido de https://es.wikipedia.org/w/index.php?title=Algoritmo_de_b%C3%BAsqueda_A*&oldid=140834447

Wikipedia. (1 de mayo de 2022). *Artificial intelligence in video games*. Recuperado el 2 de mayo de 2022, de https://en.wikipedia.org/w/index.php?title=Artificial_intelligence_in_video_games&oldid=1085624262

Wikipedia. (26 de febrero de 2022). *Ray casting*. Obtenido de https://es.wikipedia.org/w/index.php?title=Ray_casting&oldid=141928501

Wikipedia. (11 de junio de 2022). *Unity (motor de videojuego)*. Recuperado el 2022 de junio de 12, de https://es.wikipedia.org/w/index.php?title=Unity_(motor_de_videojuego)&oldid=144131762

Wikipedia. (2021 de junio de 27). *Wikpedia*. Recuperado el 2022 de mayo de 27, de https://es.wikipedia.org/w/index.php?title=B%C3%BAsqueda_en_profundidad&oldid=136615639

Wooldridge, M. (2009). *An introduction to Multiagent Systems.* Chichester, West Sussex: John Wiley and Sons Ltd.

Wooldridge, M. (2009). *Introducción, An introduction to Multiagent Systems.* Chichester, West Sussex: John Wiley and Sons Ltd.