
**Framework de comportamientos de enemigos
para videojuegos 2D**
**An enemy behaviour framework for 2D
videogames**



**Trabajo de Fin de Grado
Curso 2024–2025**

Autor
Francisco Miguel Galván Muñoz
Cristina Mora Velasco

Director
Guillermo Jimenez Díaz

Grado en Desarrollo de Videojuegos
Facultad de Informática
Universidad Complutense de Madrid

Framework de comportamientos de enemigos para videojuegos 2D

An enemy behaviour framework for 2D videogames

Trabajo de Fin de Grado en **Desarrollo de Videojuegos**

Autor

Francisco Miguel Galván Muñoz
Cristina Mora Velasco

Director

Guillermo Jimenez Díaz

Convocatoria: *Junio 2025*

Grado en **Desarrollo de Videojuegos**
Facultad de Informática
Universidad Complutense de Madrid

13 de Junio de 2025

Dedicatoria

*A nuestras gatas por inventar el arte de
convertir el dolor de un arañazo en la alegría
de un beso*

*Y a nuestras familias por el apoyo
incondicional mostrado en la consecución de
este trabajo*

Agradecimientos

A Guillermo por ser el mejor tutor de Trabajo de Fin de Grado posible y por habernos dado la oportunidad de cumplir un sueño al llegar a la consecución de este grado. A todo el resto del profesorado que nos ha enseñado tantas cosas y nos han acompañado estos años y, sobre todo, a nuestras familias que día tras día han estado con nosotros, ayudándonos a crecer, apoyándonos incondicionalmente y dándonos fuerzas para seguir adelante.

Resumen

Framework de comportamientos de enemigos para videojuegos 2D

Los videojuegos han evolucionado hasta volverse cada vez más sofisticados, combinando diversas disciplinas, como arte, sonido, programación y diseño. Además, en todo momento debe estar claro el objetivo del videojuego y los obstáculos que se deben superar. Concretamente en los plataformas 2D, los enemigos representan el principal obstáculo para el jugador. Con dicha sofisticación el sector es cada vez más técnico y el diseño de enemigos más complejo provocando la necesidad de perfiles con conocimientos en otras áreas para poder diseñar enemigos. Para evitar ese problema, surge la necesidad de crear una herramienta accesible que permita diseñar enemigos sin la barrera técnica. Para abordar este problema, se desarrollará un catálogo de componentes para Unity que facilitará la creación de comportamientos de enemigos en juegos de plataformas 2D, basado en una abstracción de patrones identificados en este tipo de juegos.

Palabras clave

Inteligencia Artificial, IA, Unity, Enemigo, Maquinas de Estado, Estado, Sensor, 2D

Abstract

An enemy behaviour framework for 2D videogames

Video games are becoming more and more sophisticated, combining various disciplines such as art, sound, programming and design. In addition, the objective of the video game and the obstacles to be overcome must be clear all the time. Specifically in 2D platformers, enemies represent the main obstacle for the player. With such sophistication, the industry is becoming more and more technical and the design of enemies more complex, causing the need for profiles with knowledge in other areas to be able to design enemies. To avoid this problem, the need arises to create an accessible tool that allows designing enemies without the technical barrier. To address this problem, a catalog of components for Unity will be developed to facilitate the creation of enemy behaviors in 2D platform games, based on an abstraction of patterns identified in this type of games.

Keywords

Artificial Intelligence, AI, Unity, Enemy, State Machine, State, Sensor, 2D

Capítulo 1

Introducción

“Los seres humanos no nacen para siempre el día en que sus madres los alumbran, sino que la vida los obliga a parirse a sí mismos una y otra vez”
— Gabriel García Márquez

1.1. Motivación

A lo largo de los años, los videojuegos han experimentado una notable evolución, transformándose en elementos más complejos. En paralelo, los enemigos han tenido la misma evolución. En el contexto específico de los videojuegos de plataformas en dos dimensiones, los enemigos son más que una simple oposición del jugador, son la clave para mostrar la esencia del juego. Diseñar enemigos, especialmente en el tipo de videojuegos mencionados, es una tarea cada vez más compleja. No se limita a darles cierta apariencia si no que tienen que tener unos comportamientos y características únicas. Como consecuencia, provocamos que la persona encargada de realizar esta tarea tenga que tener ciertos conocimientos multidisciplinares (arte, diseño, programación, ...). En los últimos años, han surgido herramientas destinadas a simplificar significativamente el flujo de trabajo de los diseñadores. No obstante, una proporción limitada de estas se enfoca específicamente a este espacio de trabajo. El propósito de estas herramientas reside en facilitar la labor de los diseñadores, permitiéndoles, incluso sin dominio de la programación, la capacidad de generar enemigos con funcionalidades completas.

1.2. Objetivos

Este trabajo tiene como objetivo principal el diseño y desarrollo de una herramienta en *C#* para el motor de videojuegos *Unity#*, que simplifique y agilice el proceso de creación de enemigos en juegos plataformas 2D y separe los roles de programación y diseño completamente, permitiendo que no sea necesario el conocimiento en programación para un puesto de diseñador. La herramienta contará con un catálogo de comportamientos fácil de manejar para cualquier persona independientemente de sus conocimientos de programación. Así como de un manual de usuario,

que explicará de forma clara cada componente de la herramienta, la instalación y ejemplos de uso.

Para llevar a cabo el desarrollo de nuestra herramienta, nos documentaremos con herramientas similares y analizaremos el comportamiento de los enemigos en este tipo de juegos, tomando como referencia títulos de renombre que serán presentados posteriormente.

1.3. Plan de trabajo

Para llevar a cabo este trabajo, se ha seguido la metodología ágil Scrum. Esta metodología, permite crear un flujo de trabajo enfocado en la iteración y continua mejora, asegurando un avance en el desarrollo eficiente y posibles adaptaciones frente a problemas detectados durante el proceso. El trabajo se dividirá en cuatro bloques: investigación y planificación, desarrollo de la memoria, desarrollo de la herramienta y pruebas con usuarios. Cada bloque a su vez se dividirá en subsecciones explicadas a continuación.

- Investigación y planificación:

- Estudio del problema: En esta primera fase se realizará un estudio del estado del arte, centrado en el papel de los enemigos en los videojuegos, su importancia en la jugabilidad y las diferentes técnicas utilizadas para su diseño y comportamiento.
- Selección y estudio de herramientas: Esta fase implicará un análisis comparativo de distintas técnicas y motores de videojuegos evaluando sus ventajas y desventajas, así como un estudio de su funcionamiento y arquitecturas.
- Diseño de la herramienta: En esta etapa, se definirá la arquitectura de la herramienta propuesta, describiendo las técnicas empleadas, esquemas de funcionamiento y organización de elementos principales.

- Desarrollo de la memoria:

- Redacción inicial: Es esta fase del trabajo se procederá a la redacción inicial de los contenidos cubriendo todos los puntos especificados en el índice.
- Revisión y corrección: Una vez completada la redacción inicial, se realizarán las correcciones necesarias tras revisar exhaustivamente el documento.
- Conclusiones y trabajo futuro: Tras finalizar los desarrollos y las pruebas de usuario, se redactarán las conclusiones obtenidas en base a los resultados y se detallarán los posibles pasos a seguir en un futuro.

- Desarrollo de la herramienta:

- Implementación de funcionalidades principales: En esta etapa se implementarán las funcionalidades principales de los movimientos básicos incluyendo la integración con sensores y emisores permitiendo la interacción entre ellos.
 - Implementación de ayuda visual: Se desarrollarán ayudas visuales destinadas a servir como referencias para los diseñadores, incluyendo elementos gráficos que faciliten la comprensión de los comportamientos.
 - Pruebas y depuración: Se llevará a cabo un proceso iterativo de pruebas que aseguren la funcionalidad de la herramienta, corrigiendo los errores detectados durante su implementación.
- Pruebas con usuarios:
- Primera fase de pruebas: Se harán pruebas con usuarios que no hayan probado la herramienta antes, siguiendo un plan de pruebas especificado en el apartado evaluación con usuarios. Las pruebas estarán centradas en: detectar posibles errores en las funcionalidades principales, validar la funcionalidad y evaluar la usabilidad y claridad.
 - Segunda fase de pruebas: Tras implementar mejoras recibidas de la retroalimentación de la primera prueba, se llevará a cabo una segunda prueba de verificación.
 - Corrección y resultados: Tras analizar los resultados de cada fase de prueba, se documentarán los errores y dificultades encontrados. Tras esto se procederá a implementar las correcciones necesarias para mejorar los resultados.

Capítulo 2

Estado de la Cuestión

En este capítulo se hará una investigación sobre las técnicas, herramientas y formas de crear inteligencias artificiales para enemigos. Para ello vamos a comenzar haciendo un recorrido por elementos generales relacionados con la inteligencia artificial y cómo se usan en videojuegos de plataformas en dos dimensiones, ya sea para crear un Non-Player Characters (NPC) o enemigos. Se mencionarán además herramientas para conseguir los fines descritos anteriormente y se hablará de algunos motores de videojuegos que han inspirado algunos aspectos de nuestra herramienta.

2.1. Introducción a la Inteligencia Artificial

La Inteligencia Artificial (IA) es la capacidad que tiene un sistema o software de realizar tareas diferentes entre sí de manera autónoma aplicando reglas, algoritmos o patrones de aprendizaje automático, simulando así comportamientos propios de la inteligencia humana. El potencial de la IA hoy día y por lo que está generando tanto furor es su capacidad de autonomía, ya que no solo ejecuta direcciones preestablecidas, si no que puede tomar decisiones en función de un contexto.

En el ámbito de los videojuegos, la IA se ha hecho paso a base de demostrar su gran capacidad de adaptación a contextos diversos como la adaptación a las acciones del jugador del Alien en *Alien Isolation*¹, la manera en la que puede generar contenido procedural para juegos roguelike como *Hades*² o, por último, entrenar una IA para que se acerque lo máximo posible al comportamiento de un humano usando redes neuronales como en la serie *Forza Motorsport*³.

¹<https://www.gamedeveloper.com/design/the-perfect-organism-the-ai-of-alien-isolation>

²[https://hades.fandom.com/es/wiki/Hades_\(juego\)](https://hades.fandom.com/es/wiki/Hades_(juego))

³https://forza.fandom.com/wiki/Forza_Wiki

2.2. Técnicas de modelado

Existen muchas técnicas que se pueden abordar a la hora de modelar una IA. Para decidir cuál se ajusta mejor a un problema en concreto, es importante saber ciertos factores, como por ejemplo la complejidad esperada del comportamiento, la adaptabilidad y flexibilidad de la técnica, si queremos invertir mucho tiempo en implementar estas técnicas o queremos algo rápido de hacer y funcional o los recursos que consumen.

A continuación, se presentan técnicas utilizadas en videojuegos para modelar IA, las cuales hemos seleccionado porque comparten una estructura similar a la que planteamos inicialmente. Entre ellas, las máquinas finitas de estados se ajustan especialmente a nuestro enfoque y será la que utilizaremos en nuestro desarrollo.

2.2.1. Maquinas de estado finitas

Las máquinas de estado finitas (FSM), son un modelo computacional que solo permite estar en uno de un número finito de estados en un momento dado. Está definida por un conjunto de reglas que describen las transiciones que se pueden realizar para pasar de un estado a otro.

Una FSM se representa como un grafo, siendo este una representación abstracta de un conjunto de objetos, eventos, acciones o propiedades conectados entre sí, siendo estos elementos nodos (estados) que realizan acciones y comprueban la posibilidad de que haya que cambiar de nodo.

En el mundo de los videojuegos, las Máquinas de Estados Finitas nos dicen cómo se va a comportar un elemento del juego en cada momento. Es una forma de diferenciar distintos "modos"^{en} los que puede estar (corriendo, saltando, atacando...). La FSM define cuáles son esos modos y qué tiene que pasar para que el personaje cambie de uno a otro, asegurándose de que solo esté haciendo una cosa a la vez.

Como se menciona en *Finite State Machine in Game Development*⁴, el primer videojuego documentado que utilizó FSM para implementar la lógica de juego fue *Spacewar!(1961)* desarrollado en el MIT por Steve Russell. Este videojuego implementaba una lógica basada en estados para manejar el comportamiento de las naves, la detección de colisiones y la física del juego. Aunque no usaba una implementación formal de máquinas de estado, sí modelaba cambios entre estados bien definidos, como el movimiento de las naves o la activación de los disparos.

*Pac-Man*⁵ es un videojuego que usa FSM, en el que el jugador controla un personaje amarillo en forma de círculo con una boca que se abre y cierra constantemente. Fue lanzado en 1980 por la compañía japonesa Namco (actual Bandai Namco). El objetivo de este videojuego es recorrer un laberinto e ir comiendo todos los puntos mientras evitamos cuatro fantasmas hasta que comemos una píldora de poder que

⁴<https://www.ijarsct.co.in/Paper2062.pdf>

⁵https://pacman.fandom.com/es/wiki/Pac-Man_Wiki:Portada

nos hace invulnerable y nos da la capacidad de comer a los fantasmas. Estos huirán tras comernos la píldora.

La complejidad en la IA de Pac-Man es asombrosa porque se le quiso dar profundidad al juego haciendo que cada fantasma tuviera una personalidad diferente. Para ello se implementó una máquina de estado por fantasma haciendo que la forma en la que interactúan con el entorno sea ligeramente diferente. A continuación se enumerarán los fantasmas y sus formas de comportarse.

- Blinky: es el fantasma rojo y su papel es el de cazador, siendo su personalidad la más agresiva, hecho que se refleja en que es el único fantasma que comienza fuera de la casa de los fantasmas y que tras salir empieza a perseguir al jugador incansablemente. Tiene otra característica propia, a medida que el jugador va comiendo bolitas, comienza a aumentar su velocidad.
- Pinky: como su nombre indica es el fantasma de color rosa. En japonés se llama *Machibuse*, el que tiende emboscadas. Pinky es el interceptor del juego por lo que va a tratar de cortar el camino del jugador. Es un fantasma relativamente rápido, por lo que calculará constantemente hacia donde se dirige el jugador para usar su velocidad para adelantarse y cortar el paso.
- Inky: el fantasma azul es el más impredecible de todos, deambula tranquilo por el laberinto hasta que está cerca del jugador y entonces, lo persigue.
- Clyde: el fantasma naranja y el más tranquilo de todos. Suele ser el último en salir de la casa de los fantasmas y no intentará atrapar al jugador en ningún caso.

Para ilustrar el funcionamiento del juego se usará la Figura 2.1 que representa una posible FSM para el jugador, lo que haría que las decisiones tomadas fueran lo más eficientes posibles en el momento.

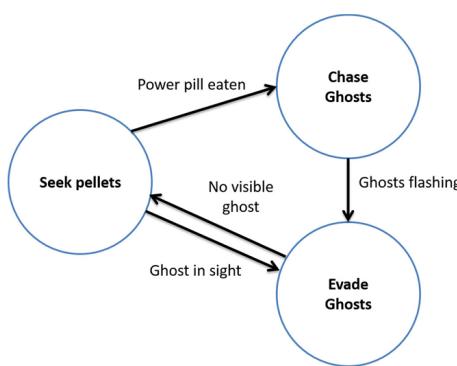


Figura 2.1: Comportamiento jugador Pac-Man, extraído del libro de Yannakakis y Togelius (2018)

Una limitación de las FSMs es que son muy inflexibles y estáticas, de manera que las posibilidades de escalado de la lógica es limitada. Asimismo, son algo predecibles una vez el jugador ha analizado y comprendido el funcionamiento de la entidad.

Sin embargo, esta limitación puede ser atenuada mediante la implementación de probabilidades o reglas que no estén tan claras a la hora de hacer las transiciones.

2.2.2. Árboles de comportamiento

Un árbol de comportamiento o Behavior Tree (BT) es un sistema similar a las FSMs ya que contamos con nodos y transiciones y solo uno de esos nodos, en este caso representando un comportamiento en lugar de un estado, puede estar activo al mismo tiempo.

En esencia, es un árbol de nodos que se organizan jerárquicamente y que atienden a unas normas que controlan el flujo de cambios de nodo.

La principal ventaja respecto a las FSMs es su modularidad, la capacidad que tiene un sistema de dividir la lógica del comportamiento en piezas independientes y reutilizables, pudiendo agrupar estas piezas en grupos que a su vez funcionan como una pieza.

Su facilidad para ser diseñados y testeados han hecho que los árboles de comportamiento se conviertan en una opción real para modelar IA en la industria del videojuego, con juegos como *Bioshock* (*2K Games, 2007*)⁶ y *Halo 2* (*Microsoft Game Studios, 2004*)⁷ como referencias en el uso de BTs.

Un ejemplo no tan conocido de uso de BTs en la industria del videojuego es *Spore* (*2008, Maxis*)⁸. Spore es un videojuego en el que el jugador va a comenzar creando una célula y va encarnarla durante todo el proceso de su evolución hasta que esta se convierta en un ser mucho más complejo llegando incluso a construir una civilización muy avanzada. La inteligencia artificial de las entidades que nos rodean en este videojuego están fundamentadas en BTs.

La gran diferencia de como usa BTs Spore a los juegos anteriormente mencionados es que Spore separa el concepto de *decider* de *behavior*. Los BTs de Halo 2 están compuestos por *behaviors* ya sean estos comportamientos grupales, en los que se elige entre varias opciones, o individuales, que ejecutan acciones específicas, e impulsos que son los saltos que se dan entre comportamientos y que se toman dependiendo de una prioridad, que a futuro resulta en problemas de escalabilidad. Para solucionar este problema, en Maxis deciden unificar el concepto de impulso y comportamientos grupales bajo el nombre de *deciders*, dejando completamente separado el concepto de *behavior* y permitiendo que estos puedan ser reutilizados en múltiples lugares del árbol, asegurando la escalabilidad.

La Figura 2.2 es un ejemplo de un BT sacado de las documentación⁹ que hay publicada del juego. Un punto importante a tener en cuenta es que si un nodo tiene más de un hijo, estos se comprueban si son o no elegidos en un orden determinado hasta que un de los *deciders* acepte la petición del padre o ninguno acepte y se

⁶<https://es.wikipedia.org/wiki/BioShock>

⁷<https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-ha>

⁸<https://www.ea.com/es-es/games/spore>

⁹https://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs

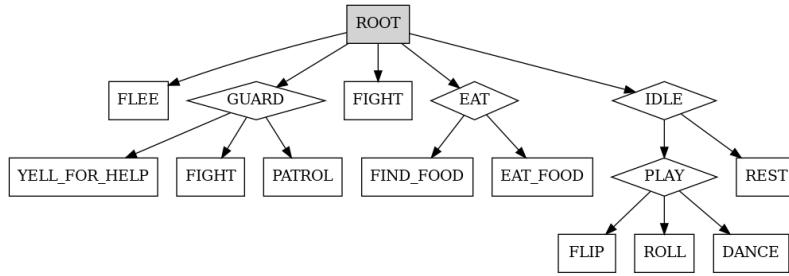


Figura 2.2: Ejemplo BT, Documentación Spore

vuelva al nodo actual volviendo a repetir el proceso.

2.2.3. Goal-Oriented Action Planning

GOAP es un sistema basado en planificación de acciones. En lugar de definir comportamientos fijos como hemos visto anteriormente, la entidad analiza la situación y construye un plan óptimo para alcanzar el objetivo designado. Esta técnica fue desarrollado en el *MIT* por ? a principio de siglo.

La entidad pasa a ser un agente autónomo que tiene la capacidad de planificar de manera dinámica una secuencia de acciones para satisfacer una meta. Para llegar a esa meta se tendrán en cuenta el contexto del agente, por lo que dependiendo de este se podrá llegar a la meta de varias maneras, aunque la utilizada sea la mejor valorada, de esta manera se reduce lo repetitivo que pueda llegar a ser lidiar con un tipo de agente, ya que siendo el enemigo, por ejemplo, el mismo este abordará al jugador de manera distinta dependiendo de la situación en la que se encuentre.

El enfoque de GOAP es muy parecido a una FSM, pero en GOAP las acciones y metas no van de la mano, si no que se separan para abrir la posibilidad de tener un proceso de planificación dinámico y adaptativo. La escalabilidad que ofrece GOAP es mayor a todas las técnicas vistas anteriormente.

El considerado primer videojuego que usa GOAP es *F.E.A.R, Monolith Productions*¹⁰. El propio Jeff Orkin explica que en el videojuego se quería llegar a una complejidad en la IA a la que las FSMs no podían llegar, por lo que optaron por no excluirlas pero solo tener tres estados y usar un algoritmo A* para planear las acciones a realizar. Un ejemplo es que si el jugador cierra una puerta mientras persigue un enemigo, el enemigo puede dinámicamente volver a rehacer su plan y decidir si buscar un hueco para disparar, por ejemplo una ventana, o buscar una entrada alternativa. Esta libertad en las acciones de los agentes liberan a los desarrolladores para que estos puedan enfocarse en el manejo de grupos, como fuego de supresión, cobertura y búsqueda del jugador.

? nos ofrece otro ejemplo ilustrativo de lo que es GOAP. El videojuego al que se refiere es *NOLF*¹¹, siendo este considerado el primer videojuego que usa un sistema parecido a GOAP, los agentes estaban dirigidos por metas aunque no podían

¹⁰<https://www.gdcvault.com/play/1013282/Three-States-and-a-Plan>

¹¹https://nolf.fandom.com/wiki/No_One_Lives_Forever_2:_A_Spy_in_H.A.R.M.%27s_Way

planificar sus acciones. A continuación vamos a definir una serie de términos clave para definir el comportamiento de GOAP.

- Objetivos: lo que la entidad quiera lograr. Un agente puede querer cumplir más de un objetivo. En el videojuego *NOLF 2* los personajes tenían típicamente alrededor de 25 objetivos, aunque en cada instante solo un objetivo esté activo y este determine las acciones del agente. Un objetivo sabe como calcular su relevancia actual y sabe cuando ha sido alcanzado.
Aunque conceptualmente son similares, hay una diferencia clave entre los objetivos en *NOLF 2* y GOAP y es que en el primero cada objetivo tiene un plan predefinido con pasos fijos y ramas condicionales establecidas de antemano y en GOAP los objetivos solo definen las condiciones que deben cumplirse para darse por terminado, los pasos para alcanzarlos se generan dinámicamente en tiempo real.
- Plan: forma de denominar una secuencia de acciones. Un plan válido es aquel que lleva a un personaje desde un estado inicial hasta un estado que cumple con el objetivo. El plan se ejecuta hasta que se complete, se invalide u otro objetivo se vuelva más importante lo que obligará a que se cree formule un nuevo plan. Hay que tener en cuenta que GOAP necesita el funcionamiento de las FSMs y las simplifica enormemente, teniendo esto en cuenta, un plan es también una secuencia de acciones donde cada acción representa una transición entre dos estados.
- Acción: una acción es un paso único y atómico dentro de un plan que hace que un personaje haga algo. Algunas acciones posibles en *NOLF 2* es *Go To Point*, *Draw Weapon...* La duración de una acción puede variar, por ejemplo la acción *Reload Weapon* terminará cuando la acción acabe, mientras que la acción *Attack* puede continuar indefinidamente hasta que el objetivo muera. Cada acción determina cuándo puede ejecutarse y qué impacto tendrá en el mundo del juego, es decir una acción conoce sus *precondiciones* y sus *efectos*.
- Mundo: estado actual del contexto de la entidad.
- Planificador: encuentra la secuencia óptima de acciones para lograr el objetivo partiendo del estado actual del agente. Si tiene éxito devolverá un plan que el personaje seguirá para guiar su comportamiento.

Para ilustrar el funcionamiento del planificador usaremos la Figura ???. Los rectángulos representan el estado inicial y el estado objetivo, los círculos representan las acciones disponibles. En este caso, el objetivo es matar a un enemigo, por lo que el estado final es aquel en el que el enemigo está muerto, en otras palabras, el planificador tiene que encontrar una secuencia de acciones que tome el mundo desde un estado en el que el enemigo está vivo hasta otro en el que este está muerto.

Este proceso se aborda como un *pathfinding*. El planificador tiene que encontrar un plan válido y no siempre este es el esperado por el usuario. Para ello debemos darle ciertos requisitos al algoritmo de *pathfinding*, por ejemplo, encontrar la secuencia de acciones más baratas que nos lleven al objetivo.

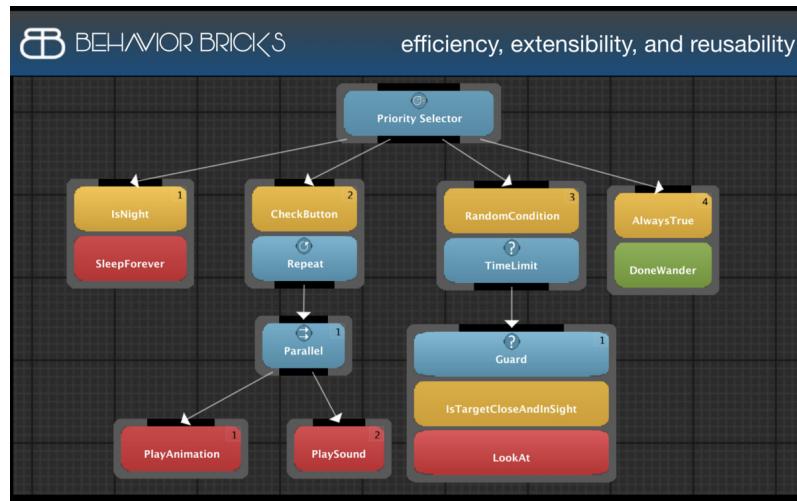


Figura 2.3: Ejemplo de uso de Behavior Bricks

(COMENTARIO: [HTTPS://GITHUB.COM/CRASHKONIJN/GOAP HERRAMIENTA QUE PODRIAMOS ANALIZAR](https://github.com/crashkoniJN/GOAP HERRAMIENTA QUE PODRIAMOS ANALIZAR))

2.3. Análisis de herramientas

Tras abordar técnicas usadas para crear IAs, se ha visto que en ocasiones la complejidad de crear esas IAs no es del todo trivial. Por ello, surge la necesidad de crear herramientas que ayudasen a los desarrolladores a crear IAs de todo tipo. Teniendo en cuenta que la creación de IAs debe estar al alcance de personas con nulo conocimiento de programación, estas herramientas deben implementar una interfaz gráfica para que el funcionamiento de la IA sea más visual. A continuación, se han seleccionado algunas herramientas como ejemplo.

2.3.1. Behavior Bricks

*Behavior Bricks*¹² es una herramienta disponible para el motor de videojuegos *Unity* centrada en cubrir todas las necesidades de implementación de comportamientos para juegos. Con *Behavior Bricks* el usuario es capaz de modelar de manera visual tanto FSMs como BTs.

Un aspecto a destacar de esta herramienta es que es totalmente gratuita, fue desarrollada por *PadaOne Games*¹³, empresa asociada a la Universidad Complutense de Madrid.

Behavior Bricks es una herramienta de scripting visual, por lo que favorece una colaboración entre diseñadores y programadores un aspecto que frecuentemente plantea desafíos.

¹²<https://bb.padaonegames.com/doku.php>

¹³<https://www.padaonegames.com/>

Otra particularidad de *Behavior Bricks* es su modularidad, permite modificar cada una de las partes con muy poco acoplamiento entre ellas, lo que incita a su reutilización en distintos proyectos. Esta modularidad también permite que podamos tener un estado/comportamiento que sea en sí mismo un grupo de estados/comportamientos.

Esta herramienta hace hincapié en no ralentizar significativamente el rendimiento del juego, gracias a su optimización y la gestión de memoria la cual a veces es compartida entre estados/comportamientos para ahorrar recursos.

2.3.2. PlayMaker

PlayMaker¹⁴ es un editor visual de FSM diseñado especialmente para artistas y diseñadores, ya que permite desarrollar IA sin necesidad de escribir código.

Su interfaz es altamente visual e intuitiva. Al crear una FSM, se genera automáticamente un estado inicial llamado *START*, como podemos ver en la Figura 2.4, seguido de un estado predeterminado llamado *State 1*, al cual se transiciona al ejecutar *Unity*. A partir de ahí, el usuario puede agregar más estados y definir eventos que permiten cambiar entre ellos según ciertas condiciones.

Una de sus principales ventajas es la facilidad con la que se pueden modificar los valores de los componentes de *Unity*. Basta con arrastrar un componente a la pestaña *State* para ajustar sus propiedades dentro de un estado determinado. Además, PlayMaker proporciona una amplia colección de acciones predefinidas, como la detección de entrada de teclas, temporizadores y movimientos entre otros. Estas acciones pueden activar eventos que actúan como transiciones entre estados, facilitando la creación de mecánicas de juego complejas sin necesidad de programación.

Gracias a su flexibilidad y facilidad de uso, PlayMaker es ideal para diseñar IAs, lógica de juego, animaciones, interacción con interfaces de usuario y prototipos rápidos, convirtiéndolo en una herramienta poderosa tanto para principiantes como para desarrolladores experimentados que buscan agilizar su flujo de trabajo. Otro punto fuerte de PlayMaker es que permite que la herramienta sea escalable con scripts propios.

PlayMaker está disponible en la *asset store* de *Unity*, aunque su precio hace que desarrolladores con pocos recursos tengan que descartar esta opción, no deja de ser una herramienta usada ampliamente por la comunidad de desarrolladores, habiendo sido utilizada en juegos como el aclamado por la crítica *Hollow Knight*¹⁵, *Firewatch*¹⁶ la aventura narrativa del estudio norteamericano Campo Santo o el juego de plataforma de los creadores de *Limbo*, *INSIDE*¹⁷.

¹⁴<https://hutonggames.com/>

¹⁵https://hollowknight.fandom.com/wiki/Hollow_Knight_Wiki

¹⁶<https://www.firewatchgame.com/>

¹⁷https://inside.fandom.com/wiki/Inside_Wiki

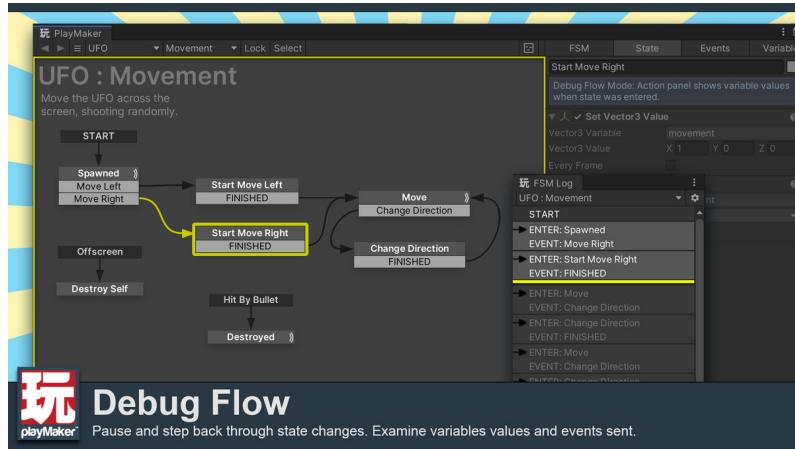


Figura 2.4: Ejemplo de uso de Play Maker

2.3.3. Decisión

Tras el análisis de las herramientas con capacidad para integrar comportamientos de inteligencia artificial, surgieron opciones relevantes que nos permiten la integración con Unity. Si bien esas herramientas son completamente válidas, se ha optado por la creación de una herramienta propia, partiendo desde cero. Esta decisión se fundamenta en la necesidad de personalización y control exhaustivo. Superando así las posibles limitaciones económicas de la investigación y las pisibles faltas de adecuación total con otras herramientas.

2.4. Motores de videojuegos

La siguiente sección consistirá en el análisis de distintos motores de videojuegos. El objetivo de este análisis es proporcionar una comprensión sobre cuales son las fortalezas y debilidades de cada uno de ellos. Todos los motores escogidos tienen gran renombre en la industria ofreciendo gran variedad y distintas filosofías de diseño.

2.4.1. Unity

*Unity*¹⁸ es un motor de videojuegos desarrollado por *Unity Technologies* que se ha convertido en una de las herramientas más utilizadas en la industria del desarrollo de videojuegos. Su versatilidad y facilidad de uso han permitido la creación de títulos de gran éxito como *Hollow Knight*¹⁹, *Cuphead*²⁰ y *Genshin Impact*²¹. La versión más actual del motor, *Unity 2023*, incorpora mejoras en su sistema de renderizado, herramientas avanzadas de optimización y un motor de físicas más eficiente.

¹⁸<https://unity.com>

¹⁹<https://www.hollowknight.com>

²⁰<https://cupheadgame.com>

²¹<https://genshin.hoyoverse.com/es>

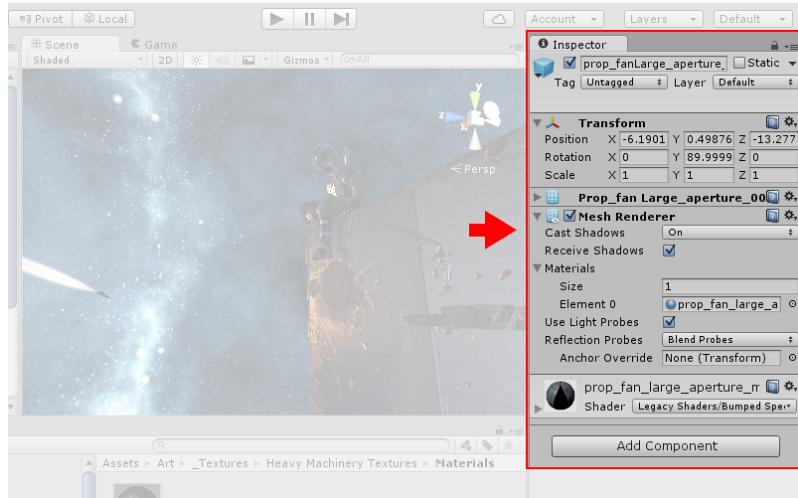


Figura 2.5: Inspector de Unity, con una serie de scripts

Uno de los aspectos más destacados de *Unity* es su capacidad para desarrollar videojuegos tanto en 2D como en 3D, lo que lo convierte en una opción ideal para una amplia variedad de proyectos. El motor ofrece dos principales opciones para la programación: el lenguaje *C#*, utilizado para la creación de scripts avanzados, y el sistema visual *Bolt*, que permite desarrollar mecánicas sin necesidad de escribir código.

El sistema de scripting en *Unity* está basado en *C#* y funciona a través del uso de *MonoBehaviour*, una clase base que permite definir el comportamiento de los objetos del juego. Los scripts se adjuntan a los objetos dentro del editor (Figura 2.5) y pueden controlar aspectos como la física, la inteligencia artificial y las interacciones del jugador.

Por otro lado, el sistema de programación visual *Bolt*²² permite a los desarrolladores sin experiencia en programación crear juegos mediante una interfaz basada en nodos, este sistema permite definir lógica de juego conectando bloques de funciones y eventos sin necesidad de escribir una sola línea de código.

Además de su versatilidad en la programación, *Unity* cuenta con un conjunto de herramientas avanzadas para la creación de entornos, animaciones y físicas. Su sistema de renderizado *Universal Render Pipeline (URP)* permite optimizar los gráficos para múltiples plataformas, mientras que el *High Definition Render Pipeline (HDRP)* está diseñado para juegos con gráficos de alta calidad en PC y consolas de última generación.

Si se compara con otros motores como *Unreal Engine*, del cual se hablará más adelante, *Unity* destaca por su flexibilidad y menor consumo de recursos. Mientras que *Unreal Engine* es ampliamente reconocido por su calidad gráfica superior, *Unity* ofrece un entorno más ligero y optimizado, lo que lo convierte en una mejor opción

²²<https://docs.unity3d.com/2019.3/Documentation/Manual/VisualScripting.html>

para desarrolladores independientes o proyectos móviles. Sin embargo, su sistema visual de nodos es menos avanzado que el de *Unreal*, lo que puede requerir el uso de *C#* para acceder a funcionalidades más complejas.

Un punto negativo de *Unity* con respecto a otros motores es su modelo de licencias y sus cambios recientes en la política de precios, lo que ha generado controversia entre los desarrolladores. A pesar de esto, su comunidad activa, su gran cantidad de recursos educativos y su compatibilidad con una amplia variedad de plataformas lo mantienen como una de las opciones más accesibles y populares para la creación de videojuegos.

La combinación de herramientas avanzadas y la facilidad de uso hace que cualquier persona pueda desarrollar desde juegos móviles y experiencias en realidad virtual hasta títulos en 3D de gran escala sin necesidad de contar con un equipo grande o conocimientos avanzados de programación.

2.4.2. Unreal Engine

*Unreal Engine*²³, desarrollado por *Epic Games*²⁴, es uno de los motores de videojuegos más potentes y utilizados en la industria de los videojuegos en títulos como la saga *Hellblade*²⁵ o el éxito mundial *Fortnite*²⁶ y la versión más actual es Unreal Engine 5 que cuenta, entre otras cosas, con *Lumen*, un sistema de iluminación global dinámica o la mejora sustancial del sistema de simulación de físicas *Chaos*.

A pesar de que permite la programación en *C++*, también ofrece un sistema visual llamado Blueprints, diseñado para que cualquier persona, sin conocimientos de programación, pueda crear videojuegos completos mediante una interfaz gráfica intuitiva.

El sistema de Blueprints funciona de manera similar a un lenguaje de programación visual basado en nodos. En lugar de escribir código manualmente, el usuario conecta bloques de lógica (Figura 2.6) para definir comportamientos, interacciones y mecánicas dentro del juego. Esto permite crear desde movimientos de personajes y mecánicas de combate hasta sistemas complejos de inteligencia artificial y físicas sin necesidad de escribir una sola línea de código.

Cabe destacar que se pueden crear Blueprints en caso de que sea necesario.

Además, Unreal Engine incluye un conjunto de herramientas preconfiguradas que facilitan el desarrollo, como sistemas de animación, iluminación, renderizado de alta calidad y físicas avanzadas. Gracias a estas características, cualquier usuario puede desarrollar videojuegos en 2D y 3D de manera accesible y rápida, sin necesidad de aprender un lenguaje de programación tradicional.

Si se compara con otros motores como Unity, Unreal Engine destaca por su calidad gráfica y su sistema visual más robusto. Mientras que en Unity se requiere programación para acceder a ciertas funciones avanzadas, en Unreal es posible construir mecánicas complejas únicamente con Blueprints. Esto lo convierte en una

²³<https://www.unrealengine.com/es-ES>

²⁴<https://www.epicgames.com/site/es-ES/home>

²⁵https://thehellblade.fandom.com/wiki/Hellblade:_Senua%27s_Sacrifice

²⁶<https://www.fortnite.com/?lang=es-ES>

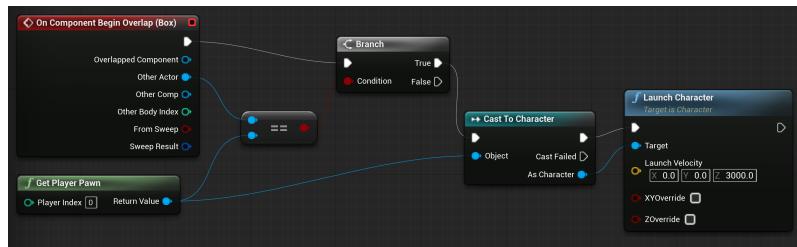


Figura 2.6: Blueprints en Unreal Engine

opción ideal para desarrolladores novatos que buscan facilidad de uso sin sacrificar potencia y flexibilidad.

Un punto negativo de Unreal Engine con respecto a Unity es la necesidad de recursos hardware que tiene para poder usarlo sin ningún tipo de ralentizaciones ni crasheos fortuitos, necesitando equipos muy potentes para que el desarrollo sea ameno o utilizar versiones anteriores de Unreal Engine.

Gracias a los Blueprints, cualquier persona puede diseñar enemigos, implementar IA, construir niveles interactivos y desarrollar mecánicas de juego avanzadas sin tocar código.

2.4.3. Godot

Godot²⁷, desarrollado por Juan Linietsky y Arenanet, es un motor de videojuegos de código abierto que ha ganado popularidad por su accesibilidad, flexibilidad y sencillez. Ofrece una plataforma poderosa para desarrollar videojuegos tanto en 2D como en 3D, y uno de sus mayores atractivos es que está diseñado para ser fácil de usar sin necesidad de tener conocimientos avanzados de programación.

A diferencia de otros motores como Unreal Engine o Unity, Godot permite el desarrollo de videojuegos con una interfaz intuitiva, pero también brinda herramientas más accesibles para aquellos que no desean escribir código. Su sistema de GDScript, un lenguaje propio diseñado específicamente para ser fácil de aprender y usar, facilita la creación de juegos sin necesidad de un conocimiento profundo de programación. GDScript es similar a Python (comparten sintaxis y tipado dinámico), lo que lo hace accesible y amigable para desarrolladores noveles.

Para aquellos que prefieren una experiencia más visual y menos enfocada en la programación, Godot incluye un sistema de "VisualScript" (Figura 2.7), un lenguaje visual basado en nodos que permite desarrollar mecánicas sin escribir código. Similar a los sistemas de programación visual en otros motores como Unreal Engine. Este sistema ha sido eliminado del núcleo de Godot a partir de la versión 4.0 del motor aunque en lanzamientos futuros VisualScript será re-implementado como una extensión²⁸.

Godot también destaca por ser un motor muy optimizado para el desarrollo de juegos en 2D, proporcionando una serie de herramientas específicas para este tipo

²⁷<https://godotengine.org/>

²⁸https://docs.godotengine.org/es/3.5/tutorials/scripting/visual_script/index.html

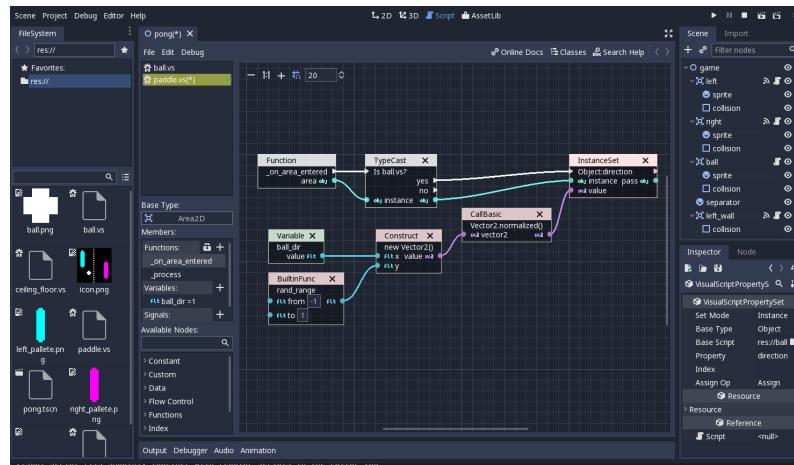


Figura 2.7: Sistema VisualScript en Godot

de desarrollo, como un sistema de mallas 2D, animaciones, efectos y un conjunto de físicas dedicadas al mundo 2D. De esta manera, los desarrolladores pueden crear juegos con un rendimiento óptimo, incluso para dispositivos de baja gama, y con un flujo de trabajo ágil.

Comparado con otros motores como Unreal Engine o Unity, Godot se distingue por su flexibilidad y ligereza. No requiere licencias ni regalías, lo que lo convierte en una excelente opción para proyectos indie como el roguelite *Brotato*²⁹ y desarrolladores independientes. También es especialmente potente para aquellos interesados en el desarrollo de juegos 2D, ya que ofrece herramientas específicamente diseñadas para este propósito, algo en lo que otros motores como Unity o Unreal Engine no se enfocan tanto.

La combinación de su accesibilidad y herramientas solventes hace que Godot permita a sus desarrolladores sin experiencia en programación crear juegos completos, desde mecánicas simples hasta sistemas complejos, sin necesidad de escribir código avanzado. Esta facilidad de uso, combinada con un motor robusto y libre, hace que Godot sea una opción popular para quienes buscan comenzar en el mundo del desarrollo de videojuegos o aquellos que desean una solución completamente gratuita y personalizable para sus proyectos.

Aunque Godot cuenta con grandes ventajas, cabe destacar que tanto Unity como Unreal son motores más establecidos en la industria, ya sea por la cantidad de assets, plugins o expansión de estos.

2.4.4. GameMaker

*GameMaker*³⁰ es un motor de videojuegos desarrollado por *YoYo Games* que se especializa en la creación de juegos en 2D, aunque también cuenta con soporte limitado para gráficos en 3D. Su accesibilidad y facilidad de uso lo han convertido en una de las herramientas más populares entre desarrolladores indie, permitiendo

²⁹https://brotato.wiki.spellsandguns.com/Brotato_Wiki

³⁰<https://gamenmaker.io/en>

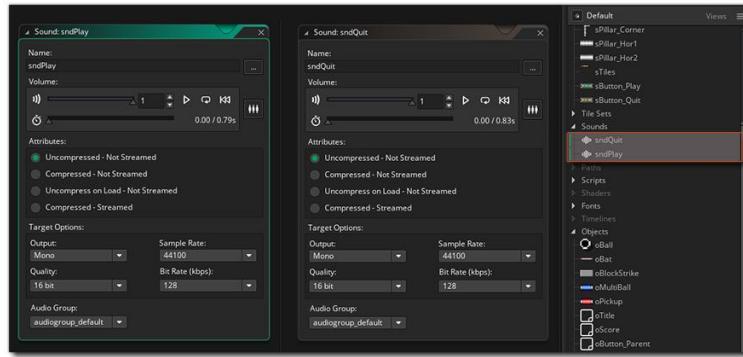


Figura 2.8: Drag and Drop en GameMaker

la creación de títulos exitosos como *Undertale*³¹, *Hotline Miami*³² o *Katana ZERO*³³.

A diferencia de otros motores, GameMaker ofrece dos formas principales de desarrollo: un sistema de programación visual basado en eventos llamado *Drag and Drop* (*D&D*) y un lenguaje de scripting propio llamado *GameMaker Language* (*GML*). La opción de *Drag and Drop* permite a los usuarios sin experiencia en programación crear juegos completos mediante una interfaz intuitiva de bloques gráficos (Figura 2.8), mientras que *GML* proporciona mayor control y flexibilidad a desarrolladores con conocimientos de programación.

El sistema de *Drag and Drop* funciona mediante la asignación de eventos y acciones a los objetos del juego. Por ejemplo, se pueden definir eventos como "cuando el jugador presione una tecla", seguido de una acción como "mover el personaje en una dirección". Esta metodología facilita la creación de juegos sin necesidad de escribir código, aunque también permite una transición fluida a *GML* en caso de que el usuario desee más control sobre la lógica del juego.

Además de su sistema de scripting y su interfaz intuitiva, GameMaker incluye diversas herramientas preconfiguradas que agilizan el desarrollo, como un editor de sprites incorporado, un sistema de animación, un motor de colisiones y soporte para efectos visuales mediante *shaders*. Gracias a estas características, cualquier usuario puede desarrollar videojuegos en 2D sin necesidad de aprender un lenguaje de programación desde cero.

Si se compara con otros motores como *Unity*, GameMaker se destaca por su ra-

³¹<https://undertale.com>

³²https://store.steampowered.com/app/219150/Hotline_Miami/

³³<https://katanazero.com>

pidez y facilidad de uso en proyectos en 2D. Mientras que en *Unity* la configuración de un juego en 2D puede requerir una mayor curva de aprendizaje, GameMaker permite comenzar a desarrollar desde el primer momento con una interfaz optimizada para este tipo de juegos. Sin embargo, su soporte para gráficos en 3D es limitado en comparación con *Unreal Engine* o *Unity*, lo que lo hace menos adecuado para proyectos que requieran entornos tridimensionales complejos.

Un punto negativo de GameMaker con respecto a otros motores es que algunas funciones avanzadas, como la exportación a consolas o la personalización del motor, requieren licencias de pago, lo que puede representar una barrera para algunos desarrolladores. Sin embargo, su modelo de suscripción y la posibilidad de utilizar la versión gratuita para prototipado lo convierten en una opción accesible para quienes buscan una herramienta de desarrollo rápida y eficiente.

Gracias al sistema de *Drag and Drop* y *GML*, cualquier persona puede crear plataformas interactivas, implementar inteligencia artificial básica, desarrollar mecánicas de combate y programar juegos completos sin necesidad de utilizar motores más complejos.

2.5. Conclusiones del análisis

A lo largo del análisis, se han recopilado los aspectos positivos tanto de las herramientas como de los motores de videojuegos estudiados, con el objetivo de utilizarlos como referencia e inspiración en el desarrollo del proyecto. Este proyecto está diseñado específicamente para desarrolladores noveles, proporcionando una solución accesible y flexible para la creación de enemigos en videojuegos de plataformas 2D.

El proyecto consistirá en una serie de componentes modulares, que permitirán a los desarrolladores generar enemigos de manera sencilla y eficiente. Estos componentes estarán diseñados para facilitar la implementación de comportamientos básicos sin requerir un conocimiento profundo de programación o inteligencia artificial. Además, la arquitectura del sistema garantizará que la herramienta sea escalable, permitiendo la incorporación de nuevos componentes en el futuro para ampliar su funcionalidad según las necesidades del usuario.

En cuanto a técnicas de modelado analizadas, se han identificado varias características destacables que servirán de referencia para el diseño del proyecto. PlayMaker sobresale por su sistema de scripting visual, el cual simplifica la representación y modificación de máquinas de estado finitas (FSM), haciendo que la creación de comportamientos sea más intuitiva. Por otro lado, Unity ofrece un inspector versátil y de fácil entendimiento, lo que facilita la configuración y manipulación de objetos en el entorno de desarrollo.

En conclusión, el desarrollo de esta herramienta busca simplificar la creación de

enemigos en videojuegos 2D, brindando a los desarrolladores principiantes una forma accesible de implementar inteligencia artificial básica. Al integrar los aspectos positivos de las herramientas y motores estudiados, se espera proporcionar una solución flexible, escalable y fácil de usar, fomentando la creatividad y el aprendizaje en el desarrollo de videojuegos.

Capítulo 3

Descripción del Trabajo

En este capítulo se describe el framework de enemigos creado, mediante el diseño de componentes más sencillos. Primero se describirá el contexto y por tanto la utilidad de la herramienta, aquí se detallaran juegos analizados y sus características en común. Después se explicará que elementos la componen y por último se detallarán algunos ejemplos de uso.

3.1. Contexto

Como se señala en ?, los enemigos bien diseñados son clave para evitar que los niveles queden planos y, en consecuencia, aburridos para el jugador. Un buen diseño de enemigos va más allá de poner un obstáculo en el camino. Aporta dinamismo, construye la atmósfera del juego y hasta cuenta parte de la historia. Un enemigo puede obligarte a pensar una estrategia específica para vencerlo, o incluso tener una personalidad y comportamientos complejos que hacen que el enfrentamiento sea más significativo e inmersivo. En definitiva, diseñar a los antagonistas es una parte fundamental para crear un juego que enganche y deje huella.

3.1.1. Enemigo

Los enemigos son entidades programadas para enfrentarse al jugador y crear desafíos dentro del juego. Su diseño incluye características, comportamientos y habilidades diseñadas para interactuar con las mecánicas del juego y contribuir a la experiencia del jugador.

3.1.2. Análisis de enemigos en videojuegos

El análisis de diversos documentos muestra que la mejor forma de hacer que un enemigo destaque y de un gran potencial al juego es que tenga unos comportamientos únicos. Estos comportamientos han ido evolucionando con el tiempo volviéndose cada vez más sofisticados. Cada enemigo se define mediante una forma única de un

conjunto de componentes bien seleccionados que permiten identificarlo y recordarlo. Con dicha sofisticación, ha aumentado la complejidad del trabajo en el diseño. Para solucionarlo hemos propuesto una herramienta con la composición indicada a continuación.

3.2. Composición

En este trabajo, se ha decidido entender como enemigo a cualquier entidad que pueda repercutir de forma negativa en el jugador, esto significa que no se limita el concepto de enemigo a figuras típicas, como monstruos o soldados hostiles, sino que se amplía su definición a toda entidad que suponga un riesgo, dificultad o amenaza para el progreso o el bienestar del jugador dentro del juego, como pueden ser pinchos, lava o gotas de ácido. Además separamos cada enemigo por comportamientos diferentes, implicando que elementos que clásicamente aparecen en conjunto, como la tubería y la gota de ácido o la bala y el pistolero, serán considerados como dos enemigos distintos.

3.2.1. Máquina de estados finita

La Máquina de Estados Finita (FSM, Finite State Machine) es el núcleo de la lógica que define el comportamiento de los enemigos en nuestro diseño. Cada enemigo tiene su propia FSM, configurada específicamente para representar sus patrones de acciones, reacciones y relaciones en el juego. La FSM organiza el comportamiento de los enemigos mediante estados y transiciones: Los estados agrupan las acciones que el enemigo puede realizar en un momento dado. Las transiciones permiten cambiar de un estado a otro y son activadas por sensores y emisores.

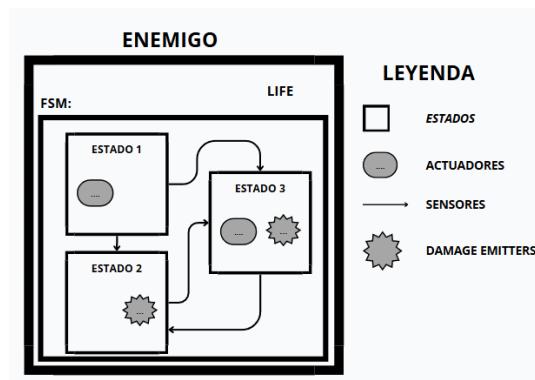


Figura 3.1: Enemigo General

Estos conceptos (estados, acciones, sensores y emisores) se desarrollan con mayor detalle en los apartados siguientes.

3.2.2. Estado

Dentro del marco de la Máquina de Estados Finita (FSM), un estado se define como un conjunto específico de acciones y la potencial activación de mecanismos de percepción (sensores) y de emisión de señales (emisores). Esta conjunción de elementos define de manera integral la conducta observable del enemigo en un momento dado. Las acciones que un estado puede englobar comprenden uno o varios tipos de movimiento que resulten compatibles entre sí, permitiendo una ejecución coordinada de desplazamientos. Adicionalmente, un estado puede tener la capacidad de generar nuevas entidades enemigas a través de un mecanismo de instanciación (spawner), enriqueciendo la dinámica y la complejidad del entorno de juego.

3.2.3. Actuadores (Acciones)

Hace referencia a un conjunto de movimientos y habilidades que definen lo que un enemigo puede hacer en el videojuego. Esto incluye distintos tipos de desplazamientos y la capacidad de crear otros enemigos de forma independiente mediante spawners. Estas habilidades no siempre son compatibles entre sí, teniendo una tabla en la que se indicaran las relaciones entre ellas. Además no es necesario utilizar siempre todas las acciones de forma que a veces un enemigo podrá realizar un tipo de movimiento o usar un spawner de manera exclusiva, mientras que en otras podrá combinar varias habilidades según su diseño y complejidad. Esto permite adaptar las capacidades de los enemigos para diferentes situaciones en el juego.

Spawner

Capacidad que poseen los enemigos para generar otros enemigos independientes, es decir, poder crear nuevas unidades que actúan de forma autónoma dentro del juego.

Los spawners presentan características únicas dentro del sistema de enemigos, ya que su objetivo no es solo enfrentarse al jugador, sino también aumentar el número de amenazas de manera continua o en función de ciertas condiciones. Implementar un spawner requiere establecer ciertas reglas de generación de enemigos, tales como la frecuencia de aparición, el número máximo de enemigos generados o si las unidades generadas son temporales o persistentes.

Movimiento

Podemos definir el término movimiento como el desplazamiento o cambio de posición de un enemigo dentro del juego. Sin embargo, el concepto de movimiento también puede aplicarse en el caso de enemigos que permanecen en una posición fija, haciendo referencia a la ausencia de éste. Los movimientos son fundamentales para definir el comportamiento de los personajes, ya que permiten la interacción con el jugador y el entorno.

A continuación se muestran todos los movimientos implementados, junto con una breve descripción.

- **Horizontal Actuator:** Deslaza el objeto horizontalmente, hacia la izquierda o derecha.
- **Vertical Actuator:** Desplaza el objeto verticalmente, hacia arriba o abajo.
- **Directional Actuator:** Mueve el objeto en una dirección definida por un ángulo o hacia el jugador.
- **Circular Actuator:** Hace que el objeto siga un movimiento circular alrededor de un punto.
- **Move to a Point Actuator:** Dirige el objeto hacia puntos aleatorios dentro de un área o puntos predefinidos.
- **Move to an Object Actuator:** Desplaza el objeto hacia un punto en movimiento.
- **Spline Follower Actuator:** Permite al objeto seguir una trayectoria definida por un spline.

Compatibilidad

La versatilidad de la herramienta se potencia al permitir la combinación de diversos actuadores simultáneamente. En este sentido, un enemigo podría perfectamente desplazarse horizontalmente mientras activa un spawner para generar secuaces. Esta sinergia entre actuadores (movimientos y spawners) enriquece la complejidad del comportamiento y abre un abanico de posibilidades creativas para los diseñadores. No obstante, es crucial reconocer que no todas las combinaciones de acciones resultan coherentes o deseables desde la perspectiva del diseño del juego. Por ejemplo, intentar realizar un movimiento vertical mientras se está definido un movimiento circular podría generar comportamientos visuales o de jugabilidad no intencionados. Para clarificar estas interdependencias y asegurar la coherencia en la configuración de los enemigos, se ha elaborado la tabla **tabla 3.1** de compatibilidad. Esta tabla actuará como una guía visual e intuitiva, indicando qué combinaciones de movimientos y la activación de spawners son factibles y recomendadas, permitiendo a los diseñadores construir enemigos con comportamientos complejos pero lógicos y bien definidos.

	Movimiento Horizontal	Movimiento Vertical	Quieto	Movimiento hacia un punto	Circular/Rotación	Péndulo
Movimiento Horizontal						
Movimiento Vertical						
Quieto						
Movimiento hacia un punto						
Circular/Rotación						
Péndulo						

Tabla 3.1: Matriz de compatibilidad de movimientos

3.2.4. Sensores y Emisores

Los términos sensores y emisores definen los mecanismos mediante los cuales los personajes interactúan con su entorno y entre sí. Podemos definir el término sensor como el elemento necesario para medir variables exteriores y enviar la información al enemigo.

Definimos Emisores como anónimo de sensor, es decir, es aquel elemento que envía información del enemigo al exterior.

- **Area Sensor:** Detecta si un objeto entra o sale en una zona de detección.
- **Collision Sensor:** Detecta colisiones físicas con objetos de capas específicas.
- **Distance Sensor:** Detecta si un objeto está dentro o fuera de una distancia específica.
- **Time Sensor:** Detecta cuando ha transcurrido un tiempo determinado.
- **Damage Sensor:** Detecta si el objeto ha recibido daño de cualquier fuente.
- **Damage Emitter (Instant):** Aplica daño una sola vez al contacto con otro objeto.
- **Damage Emitter (Permanence):** Aplica daño mientras se permanece en contacto con el objeto.
- **Damage Emitter (Residual):** Aplica daño inicial y luego daño adicional durante un tiempo tras el contacto.

3.2.5. Daño

Aunque ya ha aparecido el concepto de daño en los sensores y emisores creados, es importante definir con precisión su significado así como las diversas formas de daño existen.

Es relevante mencionar que al hablar de daño, el concepto de enemigo pierde relevancia y se aborda más con respecto a las representaciones físicas de volúmenes de colisión.

Se define daño como la consecuencia negativa que recibe una entidad con respecto a su vida. Esto se produce como la consecuencia de que un volumen de colisión que emite daño colisione contra otro volumen de colisión que recibe daño.

Existen diferentes tipos de daño y parámetros específicos que determinan cómo los volúmenes de colisión reciben, emiten y procesan daño. La primera consideración que hay que tener en cuenta es que el daño no es bidireccional, lo que implica que un volumen que recibe daño no tiene porqué emitir daño. Para reflejar esta diferenciación, se pueden distinguir tres tipos de cajas:

- **Caja de Colisión Recibir Daño:** Áreas en las que el enemigo es vulnerable.
- **Caja de Colisión Emitir Daño:** Áreas desde las que el enemigo infinge daño.

- **Caja de Colisión No Daño:** Zonas invulnerables del enemigo que no interactúan con el sistema de daño.

Aunque en teoría distinguimos esos tres tipos, a la hora de implementar son cosas distintas. Las cajas de no daño son áreas sin ningún elemento añadido. Las que reciben daño llevan el componente llamado Damage Sensor que detecta y gestiona el daño. Las que dan daño tienen un Damage Emitter que se encarga de hacer daño a otros.

Además, se puede distinguir entre tres tipos de daños:

- **Instantáneo:** Aplica daño una única vez el daño al tener contacto.
- **De Permanencia:** Infinge daño continuo mientras haya contacto, definiendo la cantidad y la frecuencia.
- **Residual:** Aplica daño inicial y luego daño periódico tras el contacto, definiendo cantidades, frecuencia y número de aplicaciones.

Capítulo 4

Implementación

En el Capítulo 3 se abordó la descripción de la herramienta sin entrar en detalles de como funcionaba esta por debajo, una visión general de lo que se iba a ofrecer, como la organización de los componentes, los tipos de daño o los diferentes tipos de actuadores. En este capítulo se va a tratar en profundidad la implementación de estos componentes, hablando de cómo funcionan, cómo se pueden personalizar y como los distintos componentes interactúan entre ellos.

4.1. Tecnología utilizada

Este proyecto ha sido desarrollado íntegramente en Unity, mencionado en el Capítulo 2 de este trabajo.

Para la construcción de esta herramienta, se seleccionó la versión 2022.3.18f1(LTS) de Unity. Por consiguiente, no podemos garantizar su correcto funcionamiento en versiones previas. No obstante, la previsión de la herramienta es ofrecer el correcto funcionamiento en versiones posteriores, salvo modificaciones significativas en la API fundamental de Unity.

Unity es una herramienta muy versátil, la cual se adapta muy bien a un gran rango de aplicaciones diferentes entre sí, desde entornos simples en dos dimensiones hasta entornos mucho más complejos en tres dimensiones, incluso en realidad virtual o realidad aumentada. Unity surge de la idea de acercar el desarrollo de videojuegos a segmentos de la población que se podrían ver abrumados por la necesidad de entender de programación para realizar sus proyectos ya sean estos profesionales o amateurs. Este motor de videojuegos ofrece soporte para varios lenguajes de programación a través de su sistema de *plugins*, de manera nativa Unity nos ofrece los lenguajes C# y Javascript como principales lenguajes de programación de *scripts*.

Unity cuenta con una interfaz de usuario muy gráfica la cual resulta muy intuitiva dentro de su complejidad. También es importante mencionar la personalización de su interfaz otorgando al usuario la capacidad de distribuir en pantalla las distintas ventanas que componen la interfaz de usuario de la manera más cómoda posible.

Otro elemento muy importante de Unity es el sistema *drag and drop* el cual nos

permite construir las escenas de juego de manera muy sencilla, moviendo los objetos en la escena haciendo click y arrastrándolos y también permite asignar scripts a las entidades de juego de la misma manera. Con respecto a la curva de aprendizaje de Unity esta no es muy pronunciada ya que desde Unity como empresa se toman muy en serio el tener una documentación clara y accesible, así como habilitar foros y tutoriales para que sea la propia comunidad de usuarios la que se ayuda a sí misma.

El motivo por el que se escoge Unity sobre los demás motores de videojuegos es esa accesibilidad que ha sido mencionada anteriormente que hace que resulte sencillo de utilizar por cualquier persona. Otro motivo de peso por el que hemos considerado Unity como nuestra opción principal de motor de videojuegos es la cantidad de usuarios que lo usan día a día lo que lo convierte en una muy buena plataforma para poder poner a prueba nuestra herramienta a través de pruebas de usuario para su posterior uso para el gran público. Esa popularidad da ciertas garantías de que la herramienta será usada y servirá para confeccionar un mínimo de proyectos. El sistema *drag and drop* facilita mucho el uso de nuestra herramienta y la comprensión de la misma. Se hará uso de otros elementos de Unity para llevar a cabo esta herramienta como el motor de físicas 2D o las herramientas que tiene el motor para ayudar al usuario a debuggear como puede ser Gizmos. Unity funciona mediante una arquitectura por componentes, lo que es muy útil para que el programa sea modular, que pueda ser dividido en piezas más pequeñas y que estas piezas sean independientes.

(COMENTARIO: FALTAN FOTOS EN ESTE APARTADO.)

A continuación se va a hacer una descripción específica de cada uno de los apartados implementados.

4.2. FSM

El comportamiento de la entidad estará encapsulado en una FSM, cuya única funcionalidad es la de gestionar el estado actual, comprobar que no ha habido ningún cambio de estado y, en caso de haberlo, manejar el cambio. Esta comprobación se hará al finalizar la actualización del estado actual (en el `LateUpdate`) para así evitar problemas con cambiar de estado en medio de un bucle sin terminar.

Valores de configuración

- (State) **Initial State**: Estado inicial de la FSM.

4.2.1. Transition

Esta clase representa una transición de estado. Está compuesta por un sensor y un estado objetivo.

Si el sensor se activa, la entidad cambiará automáticamente al estado especificado.

Valores de configuración

- (Sensor) **Sensor**: Sensor encargado de detectar el evento que hará que el cambio de estado se dé.
- (State) **Target State**: Estado de destino de la transición.

4.2.2. State

La clase *State* representa un estado de comportamiento de un enemigo. Para ello gestiona dos elementos fundamentales: *Actuators* y *Sensors*.

Valores de configuración

- (List<Actuator>) **Actuator List**: Lista de Actuadores que son actualizados en cada bucle y representan acciones, se abordarán a continuación.
- (List<SensorStatePair>) **Transition List**: Lista de Transiciones que pueden ser activadas, lo que supone un cambio de estado al estado objetivo.
- (List<DamageEmitter>) **Damage Emitter in State**: Lista de DamageEmitter activos en el estado y que son los encargados de reportar daño.
- (bool) **Debug State**: Booleano utilizado para indicar si se quiere que los Actuators en *Actuator List* y Sensores en *Transition List* muestren información a través del Gizmos.

Cuando se produce un cambio de estado, todos los actuadores y sensores se detienen, y los sensores se desuscriben de todas las transiciones a los que estuvieran vinculados.

4.3. Actuator

Como se ha mencionado anteriormente, un actuator es el script encargado de ejecutar una acción, por lo que para cada tipo de acción existirá un actuator que la represente. La clase *Actuator* representa la clase base de la que heredarán todos los tipos de actuadores. Esta clase abstracta contiene métodos para crear, destruir y actualizar cada actuador, así como un booleano que indica si el Actuador tiene la depuración activa o no.

4.3.1. MovementActuator

La clase *Movement Actuator* que hereda de *Actuator* se usa para todos aquellos actuadores que realizan una acción de movimiento.

Valores de configuración

- (bool) **Is Accelerated**: bool que indica si el movimiento que se va a realizar tiene una velocidad constante (valor a false), o que por el contrario, la velocidad va a ir variando (valor a true).
- (EasingFunction.Ease) **Easing Function**: Función que define el movimiento.

4.3.1.1. HorizontalActuator

Horizontal actuator es un actuador que hereda de *Movement Actuator* y permite mover un objeto horizontalmente, ya sea a la izquierda o a la derecha, con diferentes configuraciones de velocidad y comportamientos tras una colisión.

Valores de configuración

- (LayerMask) **Layers To Collide**: Máscara de capas que indica cuáles son las que utilizamos para colisionar con el objeto.
- (enum) **Direction**: Dirección inicial del movimiento. Puede ser *Left* o *Right*.
- (enum) **On Collision Reaction**: Reacción que va a tener el objeto al colisionar. Puede ser *None* (sin reacción), *Bounce* (rebota cambiando la dirección), o *Destroy* (se destruye al colisionar).
- (float) **Speed**: Velocidad del movimiento.
- (float) **Goal Speed**: Velocidad final del movimiento si este es acelerado.
- (float) **Interpolation Time**: Duración en segundos que va desde la velocidad inicial a *Goal Speed*.
- (bool) **Throw**: Indica si el movimiento es un lanzamiento, es decir, si está a true, se lanzará inicialmente con una velocidad inicial y, en caso contrario, se aplicará constantemente una fuerza.
- (bool) **Follow Player**: si es *true*, la dirección del movimiento se ajusta automáticamente para acercarse al jugador.

4.3.1.2. VerticalActuator

Vertical actuator es un actuador que hereda de *Movement Actuator* y permite mover un objeto verticalmente, ya sea arriba o abajo, con diferentes configuraciones de velocidad y comportamientos tras una colisión.

Valores de configuración

- (LayerMask) **Layers To Collide**: Máscara de capas que indica cuáles son las que utilizamos para colisionar con el objeto.
- (enum) **Direction**: Dirección inicial del movimiento. Puede ser *Up* o *Down*.
- (enum) **On Collision Reaction**: Reacción que va a tener el objeto al colisionar. Puede ser *None* (sin reacción), *Bounce* (rebota cambiando la dirección), o *Destroy* (se destruye al colisionar).

- (float) **Speed**: Velocidad del movimiento.
- (float) **Goal Speed**: Velocidad final del movimiento si este es acelerado.
- (float) **Interpolation Time**: Duración en segundos que va desde la velocidad inicial a *Goal Speed*.
- (bool) **Throw**: Indica si el movimiento es un lanzamiento, es decir, si está a true, se lanzará inicialmente con una velocidad inicial y, en caso contrario, se aplicará constantemente una fuerza.
- (bool) **Follow Player**: si es *true*, la dirección del movimiento se ajusta automáticamente para acercarse al jugador.

4.3.1.3. DirectionalActuator

La clase *Directional Actuator* que hereda de *Movement Actuator* se usa para describir un movimiento en función de un ángulo y una velocidad.

Valores de configuración

- (LayerMask) **Layers To Collide**: Capas con las que puede colisionar el objeto y activar reacciones.
- (float) **Speed**: Velocidad del movimiento.
- (float) **Goal Speed**: Velocidad final del movimiento si este es acelerado.
- (float) **Interpolation Time**: Duración en segundos que va desde la velocidad inicial a *Goal Speed*.
- (float) **Angle**: Ángulo de dirección del movimiento, en grados.
- (bool) **Throw**: Indica si el movimiento es un lanzamiento, es decir, si está a true, se lanzará inicialmente con una velocidad inicial y, en caso contrario, se aplicará constantemente una fuerza.
- (enum) **On Collision Reaction**: Reacción que va a tener el objeto al colisionar. Puede ser *None* (sin reacción), *Bounce* (rebota cambiando la dirección), o *Destroy* (se destruye al colisionar).
- (bool) **Aim Player**: Si es *true*, el objeto calculará automáticamente el ángulo inicial para moverse en dirección al jugador.

4.3.1.4. CircularActuator

La clase *Circular Actuator* que hereda de *Movement Actuator* se usa para describir un movimiento circular alrededor de un punto.

Valores de configuración

- (float) **Angular Speed**: Velocidad de giro en grados por segundo.

- (Transform) **Rotation Point Position**: Punto central de la circunferencia que describe el objeto.
- (float) **Max Angle**: Ángulo máximo de giro. Si el ángulo es 360° entonces describirá una circunferencia completa, si no, hará un movimiento en forma de péndulo con los grados indicados.
- (float) **Angular Acceleration**: Aceleración angular.
- (float) **Goal Angular Speed**: Velocidad angular que se quiere alcanzar si el objeto es acelerado.
- (bool) **Can Rotate**: si es *true*, el objeto rota sobre sí mismo siguiendo la trayectoria.
- (float) **Interpolation Time**: Tiempo en segundos que tarda desde la velocidad inicial hasta la velocidad final, *Goal Speed*.
- (bool) **Point Player**: si es *true*, el objeto ajusta su trayectoria para orientarse hacia el jugador.

4.3.1.5. SplineActuator

La clase *Spline Actuator* que hereda de *Movement Actuator* se usa para describir un movimiento mediante curvas Splines de Unity. El Actuador sigue la curva pudiendo girar el objeto a su vez.

Valores de configuración

- (float) **Speed**: Velocidad del movimiento.
- (float) **Goal Speed**: Velocidad final del movimiento si este es acelerado.
- (float) **Interpolation Time**: Duración en segundos que va desde la velocidad inicial a *Goal Speed*.
- (SplineContainer) **Spline Container**: Spline que define la trayectoria que seguirá el objeto.
- (enum) **Teleport To Closest Point**: Define cómo se ajusta el objeto a la spline al iniciarse el movimiento. Puede tener dos valores:
 - **Enemy**: el objeto se teletransporta a la spline.
 - **Spline**: la spline se desplaza para alinearse con la posición actual del objeto.

4.3.1.6. MoveToAPointActuator

La clase *Move to a Point Actuator* que hereda de *Movement Actuator* se usa para mover el objeto en dirección a un punto no actualizable. Puede ser a un punto concreto y seguir una lista de ellos o a puntos aleatorios dentro de un área.

Valores de configuración

- (UsageWay) **Usage Way**: Define si se va a seguir una ruta por puntos (*Waypoint*) o si se escogen puntos aleatoriamente dentro de una zona (*RandomArea*).
- (List<WaypointData>) **Waypoints Data**: Lista de puntos que el objeto debe seguir. Cada punto incluye el tiempo que se tarda en llegar, si se desea aceleración, si se debe detener al llegar, duración de la parada y función de aceleración si es necesaria.
- (Collider2D) **Random Area**: Zona dentro de la cual se moverá el objeto si está configurado como *RandomArea*.
- (bool) **Is a Cicle**: si es true, al finalizar los puntos volverá al primero y el movimiento se repetirá.
- (bool) **All Waypoints Have The Same Data**: si es true, todos los puntos usarán la misma configuración.

4.3.1.7. MoveToAnObjectActuator

La clase *Move to an Object Actuator* que hereda de *Movement Actuator* se usa para mover el objeto en dirección a un punto actualizable, esto implica que si el punto se mueve, el objeto actualizará la trayectoria a la nueva posición.

Valores de configuración

- (UsageWay) **Usage Way**: Define si se va a seguir una ruta por puntos (*Waypoint*) o si se escogen puntos aleatoriamente dentro de una zona (*RandomArea*).
- (List<WaypointData>) **Waypoints Data**: Lista de puntos que el objeto debe seguir. Cada punto incluye el tiempo que se tarda en llegar, si se desea aceleración, si se debe detener al llegar, duración de la parada y función de aceleración si es necesaria.
- (Collider2D) **Random Area**: Zona dentro de la cual se moverá el objeto si está configurado como *RandomArea*.
- (bool) **Is a Cicle**: si es true, al finalizar los puntos volverá al primero y el movimiento se repetirá.
- (bool) **All Waypoints Have The Same Data**: si es true, todos los puntos usarán la misma configuración.

4.3.2. SpawnerActuator

La clase *SpawnerActuator* hereda de *Actuator*. Este actuador se usa para poder generar nuevos enemigos, pudiendo generar infinitos enemigos o un número definido de ellos cada X tiempo en un lugar predefinido.

Valores de configuración

- (class) **Spawn Info**: Clase auxiliar serializable que almacena un prefab (`GameObject`) a instanciar y un punto de aparición (`Transform`) donde se colocará dicho objeto.
- (float) **Spawn Interval**: Intervalo de tiempo en segundos que tarda el objeto en volver a generar otro enemigo.
- (bool) **Infinite Enemies**: Indica si se generarán enemigos indefinidamente. Si es verdadero, se crearán continuamente nuevos enemigos cada X tiempo indicado por *spawnInterval* y si es falso, el número de spawns estará limitado por **Number of Times To Spawn**.
- (int) **Number of Times To Spawn**: Número total de veces que se permitirá hacer spawn, si no es infinito.
- (List<SpawnInfo>) **Spawn List**: Lista de elementos de tipo **Spawn Info**. Cada elemento contiene la información necesaria para instanciar un enemigo. Esta lista está diseñada para poder crear distintos tipos de enemigos o en varios sitios a la vez, dando así más flexibilidad.

4.4. Sensors and Emitters

Para que exista una comunicación entre la entidad y su entorno, esta necesita poder recibir y enviar información. Con este propósito se diseñan los sensores y los emisores.

4.4.1. Sensor

La clase *Sensor* es de la que heredarán los sensores utilizados en la herramienta. Esta clase contiene variables como el evento que almacena las funciones que se deberán llamar en caso de que el sensor sea activado, como puede ser la función que activará la transición posteriormente.

Cualquier clase que herede de *Sensor* tendrá la posibilidad de modificar tres funciones relativas a la lógica del sensor:

- **StartSensor**: Función que se encarga de activar el sensor para que se pueda comenzar a captar información y, en caso de querer un tiempo de espera al inicio de la activación, se crea el *timer* correspondiente.

- **UpdateSensor:** Función llamada en cada bucle y encargada de actualizar el tiempo que queda por esperar en caso de que el *timer* no haya acabado.
- **StopSensor:** Función que desactiva el sensor.

Estas funciones son las encargadas de gestionar las variables de control.

En caso de que se quiera agregar lógica extra a alguna de estas funciones para un sensor en específico, se puede sobrescribir estos métodos en la clase del sensor correspondiente con la condición de que es indispensable que se llame al método de la clase base. (**COMENTARIO: ¿FOTO DEL CÓDIGO?**)

También se incluye un modo debug opcional, activable mediante el método **SetDebug(bool)**, método que será llamado desde State y tomará el valor de la variable *Debug State* del propio estado.

La funcionalidad de los sensores se ha implementado de manera que si se quiere suscribir desde un componente a un evento de cualquier sensor se podrá hacer, siempre y cuando la función que será llamada en caso de activarse el sensor reciba un objeto de tipo *Sensor*.

La implementación del evento incluye un contador de suscriptores que permite llevar un control interno sobre cuántos componentes están actualmente registrados para recibir notificaciones del sensor. Esta es una práctica útil para evitar errores en tiempo de ejecución.

Es importante tener en cuenta que toda suscripción a un sensor debe ir acompañada, en algún momento, de su correspondiente desuscripción. Un caso típico es realizar esta desuscripción cuando la entidad es destruida, utilizando el método **OnDestroy**.

Valores de configuración

- (float) **Start Detecting Time:** Tiempo que necesitará el sensor para activarse al entrar en el estado que lo alberga.

A continuación se enumerarán y explicarán los tipos de sensores incluidos en la herramienta.

4.4.1.1. AreaSensor

AreaSensor representa un tipo de sensor espacial que detecta la presencia de un objeto objetivo dentro de una zona delimitada. Está pensado para funcionar con zonas de activación, por lo que requiere que el objeto que lo contiene tenga un componente *Collider2D*. Esta condición se asegura mediante el atributo **[RequireComponent(typeof(Collider2D))]** que obliga a Unity a añadir automáticamente ese componente si no está presente.

La detección se realiza mediante los métodos **OnTriggerEnter2D** y **OnTriggerStay2D** provistos por *Unity*. El primero detecta la entrada del objetivo en la zona, mientras

que el segundo permite capturar situaciones en las que el objetivo ya se encuentra dentro del área al momento de activarse el sensor. Esto evita perder eventos relevantes si la detección no estaba habilitada previamente.

Al usarse métodos como `OnTriggerEnter2D` y `OnTriggerStay2D` se activará automáticamente la flag que convierte un `Collider2D` en un trigger.

Valores de configuración

- (`GameObject`) **Target**: Objeto que se quiere detectar dentro de la zona delimitada por el `Collider2D`.

4.4.1.2. CollisionSensor

`CollisionSensor` es el sensor encargado de detectar la colisión de un objeto con un grupo determinado de objetos.

Al igual que en `AreaSensor`, se necesitará un `Collider2D` para detectar colisiones y se obligará a Unity a añadirlo de la misma manera.

Para acotar con qué objetos se activa el sensor y con cuáles no, se utilizarán las `LayerMask` de `Unity` y se detectarán las colisiones con `OnCollisionEnter2D` y `OnCollisionStay2D`, por lo que, para reportar una colisión, ninguno de los dos objetos debe ser trigger.

Valores de configuración

- (`LayerMask`) **Layers To Collide**: Máscara de capas físicas que, en caso de colisión, activarán el sensor.

4.4.1.3. DamageSensor

`DamageSensor` es un tipo de sensor que detecta las colisiones y entradas en el área de un objeto que emite daño. Este sensor está diseñado para funcionar tanto con colisiones como con triggers. Requiere que el objeto al que se le asigne tenga un componente `Collider2D`, lo cual se asegura automáticamente gracias al atributo `[RequireComponent(typeof(Collider2D))]`.

La funcionalidad de este sensor es doble. Además de activar transiciones, se utiliza para gestionar la lógica del componente `Life`. En caso de colisión, y bajo ciertas condiciones, como colisionar con un objeto que tenga el componente `DamageEmitter`, el sensor se activa. Luego, desde el componente `Life`, se gestiona qué hacer en función de las características del `DamageEmitter`.

El sensor puede ser configurado para que se active desde el inicio mediante el atributo `Active From Start`, lo que permite que el sensor sea útil en caso de que no se quiera que el sensor esté implicado en ninguna transición, pero sí en el sistema

de gestión de salud.

El sensor detecta las entradas y salidas de objetos mediante los métodos `OnTriggerEnter2D`, `OnTriggerExit2D`, `OnCollisionEnter2D` y `OnCollisionExit2D`. Este control será necesario para gestionar los distintos tipos de daños que serán abordados más adelante.

Valores de configuración

- (bool) **Active From Start**: Si es verdadero, el *DamageSensor* no tendrá por qué ser incluido en ninguna transición para que este se considere activo.

4.4.1.4. DistanceSensor

DistanceSensor es el sensor utilizado para medir la distancia entre dos puntos. La distancia se medirá tomando de referencia los *Transform* de sendos objetos, uno de ellos el que posee este script y el otro el objetivo de la medición.

El funcionamiento del sensor dependerá del valor asignado a la variable *Distance Type*, la cual es de un tipo enumerado *TypeOfDistance* que especifica la manera en que se calculará la distancia. Según el valor de esta variable, se requerirán distintos parámetros o configuraciones, aunque varios valores de configuración seguirán siendo necesarios independientemente de *Distance Type*.

Valores de configuración comunes

- (enum) **Distance Type**: Tipo enumerado que determinará de qué manera se mide la distancia y qué variables se necesitarán para medirla.
- (enum) **Detection Condition**: Tipo enumerado que determina si el sensor se activa cuando el objetivo está dentro de esa distancia o cuando está fuera de la misma.
- (GameObject) **Target**: Entidad con la que se mide la distancia.

A continuación se especificarán los valores que puede tomar el enumerado *TypeOfDistance* y las variables específicas necesarias en cada caso.

Valores de *TypeOfDistance*

- *Magnitude*

Configuración utilizada cuando se quiere medir la distancia como magnitud. La representación gráfica de esta forma de medir la distancia es un círculo, como se muestra en la Figura 4.1.

Dependiendo del valor de la variable *Detection Condition*, el sensor se activará si el objeto objetivo está dentro o fuera del círculo.

Valores de configuración



Figura 4.1: Medición de distancia a través de la magnitud.

- (float) **Detection Distance**: Distancia necesaria para que el sensor se active.
- *Single Axis*

Con el tipo de medida *Single Axis* se mide la distancia entre ambos objetos, pero solo en uno de los dos ejes: X o Y.

Valores de configuración

- (float) **Detection Distance**: Distancia necesaria para que el sensor se active.
- (enum) **Axis**: Da opciones para escoger cuál de los dos ejes se va a medir, si X o Y.
- (enum) **Detection Sides**: Permite elegir si se quiere medir la distancia a ambos lados del eje o solo por uno de ellos.
- (enum) **Part**: En caso de que se haya determinado que solo se quiere medir la distancia en uno de los dos lados del eje elegido, se deberá especificar cuál de los dos es. Por ejemplo, si queremos medir la distancia en el eje X, nos permite medir cuando el objetivo entra en el rango determinado por *Detection Distance* por la izquierda o por la derecha.

4.4.1.5. TimeSensor

Sensor encargado de activarse cuando pasa un tiempo determinado desde su activación.

En caso de ajustar que el sensor necesitará un tiempo de activación, primero se procederá a medir tal tiempo y, cuando este llegue a su fin, se medirá el tiempo de activación propio del sensor.

Valores de configuración

- (float) **Detection Time**: Tiempo necesario, medido en segundos, para que el sensor se active.

4.4.2. DamageEmitter

Para que una entidad pueda emitir daño, debe tener adjunto el componente *DamageEmitter*.

Este componente no implementa funcionalidad adicional en su propio script, sino que actúa como una señal para el *DamageSensor*. La detección de daño por parte del sensor depende de que el objeto con el que colisiona tenga este componente adjunto.

La manera en la que *DamageEmitter* reporta daño dependerá de un tipo enumerado llamado *DamageType*.

Como valores de configuración comunes para todas las configuraciones disponibles encontramos:

Valores de configuración comunes

- (bool) **Active From Start**: En caso de que esté desactivado, no se reportará daño a no ser que el *DamageEmitter* esté incluido en un estado.
- (enum) **Damage Type**: Diferentes formas de producir daño a las entidades que tengan *DamageSensor*.

A continuación se concretarán los valores que puede tomar el enumerado *DamageType*, cómo funciona cada tipo de daño y qué otros valores de configuración serán necesarios.

Valores de *DamageType*

- *Instant* Tipo de daño que se aplica una vez producido el contacto y que no se va a volver a aplicar hasta que ese contacto no haya finalizado y se dé uno nuevo.

Valores de configuración

- (bool) **Destroy After Doing Damage**: Booleano que determina si el objeto es destruido al reportar daño.
- (bool) **Instant Kill**: Determina si la entidad que reporta daño elimina a su objetivo instantáneamente.

- (float) **Damage Amount**: En caso de no eliminar instantáneamente al objetivo, se especificará la cantidad de daño que se hará.
- *Permanence* El daño de permanencia es aquel que produce constantemente una entidad cada X tiempo. Este daño se producirá mientras dure la colisión o, en caso de que sea un trigger, la superposición.

Valores de configuración

- (float) **Damage Amount**: Cantidad de daño administrada cada vez.
- (float) **Damage Cooldown**: Tiempo necesario para administrar daño de nuevo, medido en segundos.
- *Residual* El daño residual es aquel que produce una cantidad de daño instantáneo y, tras esto, produce un número determinado de aplicaciones de otra cantidad de daño cada cierto tiempo.

Valores de configuración

- (bool) **Destroy After Doing Damage**: Booleano que determina si el objeto es destruido al reportar el daño instantáneo.
- (float) **Instant Damage Amount**: Cantidad de daño administrada cuando se produce el contacto.
- (float) **Residual Damage Amount**: Cantidad de daño administrada por cada aplicación de daño residual.
- (float) **Damage Cooldown**: Tiempo entre aplicaciones de daño residual, medido en segundos.
- (int) **Number Of Applications**: Número de veces que se aplicará el daño residual.

4.5. Animations

La clase *AnimatorManager* se encarga de gestionar las animaciones de cada enemigo, en función del movimiento, es decir, de los actuadores que estén activos.

Valores de configuración

- (bool) **Can Flip X**: Indica si el sprite puede voltearse horizontalmente.
- (bool) **Can Flip Y**: Indica si el sprite puede voltearse verticalmente.
- (SpriteRender) **Sprite Render**: Referencia al Sprite Render del propio objeto.

Durante la ejecución, se actualizan continuamente los parámetros *XSpeed*, *YSpeed* y *RotationSpeed* del animator a partir de la velocidad del *Rigidbody2D*, lo que permite que las animaciones reflejen con precisión el movimiento real del objeto.

Además, si el enemigo cambia de dirección, los sprites rotarán siempre que se hayan activado los booleanos previamente mencionados.

Cuando el enemigo recibe daño o muere, se activan los triggers *Damage* y *Die* respectivamente, lo que lanza las animaciones correspondientes. En el caso de la muerte, el objeto se destruye automáticamente una vez finaliza la animación.

Por otro lado, la clase permite indicar si el enemigo está siguiendo a otro objeto con el parámetro *Follow* o saber si la FSM ha cambiado de estado con la función *ChangeState*.

Finalmente, al invocar eventos de aparición, se activa el trigger *Spawn*.

Parámetros requeridos en el Animator

- Triggers: *Die*, *Damage*, *Spawn*, *ChangeState*
- Bools: *Left*, *Right*, *Up*, *Down*, *Follow*, *Rotating*
- Floats: *XSpeed*, *YSpeed*, *RotationSpeed*

Para facilitar la gestión de los estados y de los parámetros del Animator, se ha creado el controlador *Controller*, que pretende servir de base para la gestión de las animaciones.

4.5.1. Movimiento del jugador

Como ayuda a la implementación, se han creado una serie de scripts auxiliares que dan funcionalidades básicas a todos los juegos y que, por tanto, eran esenciales para una implementación lógica. Esta sección y las siguientes estarán enfocadas en la explicación de estos elementos.

A la hora de implementar el movimiento del jugador usamos de base la implementación usada por el canal de YouTube *Mix and Jam* en su interpretación del videojuego *Celeste*¹.

4.5.1.1. PlayerMovement

La clase *PlayerMovement* actúa como el controlador del jugador. Su función principal es gestionar la entrada del usuario y, en consecuencia, mover al personaje. Además, si el jugador se encuentra en el suelo y se presiona la tecla de salto, la clase se encarga de ejecutar el salto. Asimismo, maneja una situación particular en la que el jugador queda suspendido en el aire mientras se mueve hacia una pared. Si este caso no se contempla, la fuerza ejercida en el eje X puede anular la del eje Y, haciendo que el personaje quede inmóvil en una posición poco natural. Para evitar este comportamiento, si el jugador está en el aire y colisiona lateralmente con una superficie, se le fuerza a deslizarse a lo largo de esta con una velocidad constante. Esta clase permite modificar ciertos valores como la velocidad de movimiento, la potencia de salto o la velocidad con la que el jugador se desliza por las superficies anteriormente mencionadas.

Valores de configuración

¹https://www.youtube.com/watch?v=STyY26a_dPY

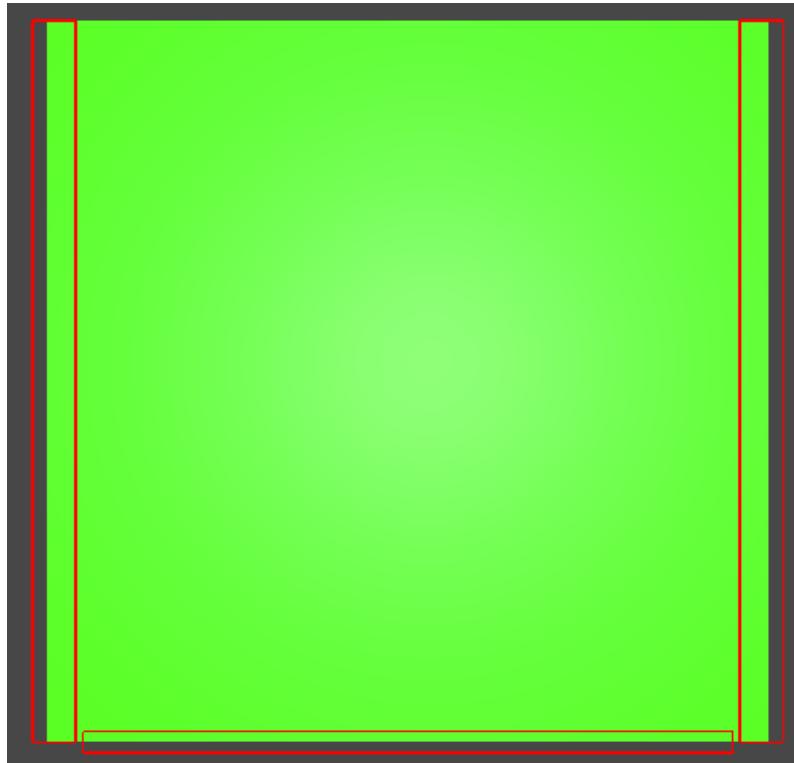


Figura 4.2: Representación de las cajas de detección de colisiones del jugador

- (float) **Speed**: Velocidad constante a la que se moverá el jugador.
- (float) **Jump Force**: Fuerza aplicada al saltar
- (float) **Slide Speed**: Velocidad aplicada en el eje Y cuando el jugador está en el aire y colisiona lateralmente con una superficie.

4.5.1.2. PlayerCollisionDetection

Este script se encarga de detectar las colisiones del jugador. Para ello, se ajustan tres cajas de colisión (Figura 4.2): una en cada lado y otra para detectar el contacto con el suelo. Las cajas no detectan realmente colisiones, sino que comprueban si estas se superponen con alguna entidad de las capas especificadas con el método `Physics2D.OverlapBox()`.

PlayerCollisionDetection será usado por *PlayerMovement* para gestionar acciones como determinar cuándo el jugador debe deslizarse por una superficie o cuándo puede saltar.

Valores de configuración

- (bool) **Debug Boxes**: En caso de que esta variable sea true, las cajas definidas por los campos que serán presentados a continuación serán representadas en pantalla con color rojo.

- (LayerMask) **Detection Layers**: Capas que serán tomadas en cuenta para detectar si el jugador está en el suelo o en contacto con una pared cuando está en el aire. En este caso se querrá especificar la capa que alberga a los objetos estáticos que conforman el mundo, por ejemplo *World*.
- (Vector2) **Bottom/Right/Left Size**: Tamaño de las cajas.
- (Vector2) **Bottom/Right/Left Offsets**: Valores usados para reposicionar las cajas para que estén en el lugar considerado por el usuario, idealmente en los bordes del objeto.

4.5.1.3. PlayerBetterJumping

Para que el salto se ajuste más al estándar de los juegos de plataformas 2D, este script mejora la sensación de salto para que el personaje caiga más rápido, evitando una sensación de flotación. Además, permite realizar saltos más pequeños si el jugador suelta el botón antes.

Para lograrlo, el script ajusta dos multiplicadores: uno que incrementa la gravedad cuando el jugador está cayendo y otro que reduce la altura del salto cuando este es interrumpido antes de tiempo.

Valores de configuración

- (float) **Fall Multiplier**: Multiplicador que aumenta la gravedad cuando el personaje está cayendo, hace que el personaje caiga más rápido, dándole más peso y realismo al salto.
- (float) **Low Jump Multiplier**: Multiplicador que aumenta la gravedad cuando el jugador suelta el botón de salto antes de llegar al punto más alto del salto, haciendo que se puedan hacer saltos cortos, dando más control del movimiento al jugador.

4.5.2. Life

Se ha implementado una clase *Life* que gestiona los puntos de salud del objeto al que está adjunto y que se encarga de eliminar el objeto en el caso de que su vida llegue a cero. El componente pedirá al usuario un valor inicial para los puntos de vida del objeto y un valor máximo al que puede llegar la vida, para que esta sea limitada en caso de que se quiera implementar un sistema de recuperación de vida.

Este componente está estrechamente ligado al *DamageSensor*, el cual será mencionado más adelante y que detecta si ha habido una colisión con un objeto que aplique daño. Esta relación es necesaria ya que, para que se detecte el daño que decrementa la salud del objeto, se necesita este sensor, por lo que al añadir un componente *Life* se añadirá este sensor automáticamente.

Life diferencia entre enemigos y el jugador; en caso de que el componente esté adjunto al jugador, se deberá marcar en la opción *EntityType* y el componente requerirá que se le dé la referencia a un texto del *Canvas* donde se escribirá la vida actual del jugador.

Valores de configuración

- (enum) **Entity Type**: Enumerado usado para diferenciar entre jugador y enemigos.
- (float) **Initial Life**: Cantidad de vida con la que inicia la entidad.
- (float) **Max Life**: Cantidad de vida máxima a la que puede llegar la entidad.
- (string) **Text Name**: En caso de ser el jugador, se pedirá un texto que precederá a los puntos de vida del jugador.
- (TextMeshProUGUI) **Life Text**: Objeto del *Canvas* usado para representar los puntos de vida actuales del jugador.

4.5.3. PlayerDistanceAttack

Componente encargado de representar un posible ataque a distancia del jugador. El ataque se activará con el clic izquierdo del ratón, lo que instanciará el prefab *Bullet Prefab* en la posición del jugador (será importante que el objeto instanciado no pueda colisionar con el jugador), el cual deberá albergar el comportamiento de una bala. La dirección que tomará la bala será aquella en la que se encuentre el cursor del ratón en el momento del clic.

Para evitar que el jugador pueda disparar sin restricción, se ha introducido un tiempo de espera entre disparos.

Valores de configuración

- (GameObject) **Bullet Prefab**: Objeto que se instancia en la posición del jugador al hacer clic.
- (float) **Shooting Cooldown**: Tiempo necesario (en segundos) antes de poder disparar nuevamente.

Capítulo **5**

Evaluacion Con Usuarios

5.1. Tecnología utilizada

5.2. Actuators

Hablar de los actuators

5.2.1. Movement

Movement contenido a ver si sale guay

5.2.1.1. AAAA

aaaaaa

5.2.2. Spawner

Spawner

5.3. Sensors and emitters

Contenido

5.3.1. Sensors

mas contenido

5.3.2. Emitters

muchisimo mas

Capítulo 6

Conclusiones y Trabajo Futuro

Conclusiones del trabajo y líneas de trabajo futuro.

Antes de la entrega de actas de cada convocatoria, en el plazo que se indica en el calendario de los trabajos de fin de grado, el estudiante entregará en el Campus Virtual la versión final de la memoria en PDF.

(COMENTARIO: EN TRABAJO FUTURO PODEMOS PONER QUE EN CASO DE QUE SE CAMBIE LA API BÁSICA NECESITARÍA ALGO DE MANTENIMIENTO BÁSICO.)

Introduction

“Los seres humanos no nacen para siempre el día en que sus madres los alumbran, sino que la vida los obliga a parirse a sí mismos una y otra vez”
— Gabriel García Márquez

6.1. Motivation

Over the years, video games have undergone a remarkable evolution, transforming into more complex entities. In parallel, enemies have followed a similar trajectory. Within the specific context of two-dimensional platformer videogames, enemies are more than just an obstacle for the player; they are key to showcasing the essence of the game. Designing enemies, especially in the aforementioned type of video games, is an increasingly complex task. It is not limited to giving them a certain appearance but requires them to possess unique behaviors and characteristics. Consequently, the person responsible for this task must have certain multidisciplinary knowledge (art, design, programming, ...). In recent years, tools aimed at significantly simplifying the workflow of designers have emerged. However, a limited proportion of these focus specifically on this workspace. The purpose of these tools lies in facilitating the work of designers, allowing them, even without programming proficiency, the ability to generate enemies with complete functionalities.

6.2. Objectives

The main objective of this work is the design and development of a tool in *C#* for the *Unity#* game engine, which simplifies and streamlines the process of creating enemies in 2D platformer games and completely separates the roles of programming and design, allowing that programming knowledge is not necessary for a designer position. The tool will feature an easy-to-manage catalog of behaviors for anyone, regardless of their programming knowledge. It will also include a user manual that clearly explains each component of the tool, its installation, and usage examples. To carry out the development of our tool, we will research similar tools and analyze the behavior of enemies in this type of game, taking renowned titles, which will be presented later, as a reference.

6.3. Work Plan

To carry out this work, the Agile Scrum methodology has been followed. This methodology allows the creation of a workflow focused on iteration and continuous improvement, ensuring efficient development progress and possible adaptations to problems detected during the process. The work will be divided into four blocks: research and planning, memory development, tool development, and user testing. Each block will in turn be divided into subsections explained below.

- Research and planning:
 - Problem study: This initial phase will involve a study of the state of the art, focusing on the role of enemies in video games, their importance in gameplay, and the different techniques used for their design and behavior.
 - Tool selection and study: This phase will involve a comparative analysis of different techniques and game engines, evaluating their advantages and disadvantages, as well as a study of their operation and architectures.
 - Tool design: In this stage, the architecture of the proposed tool will be defined, describing the techniques used, operation schemes, and organization of main elements.
- Memory development:
 - Initial drafting: In this phase of the work, the initial drafting of the contents will proceed, covering all the points specified in the index.
 - Review and correction: Once the initial drafting is completed, the necessary corrections will be made after thoroughly reviewing the document.
 - Conclusions and future work: After finalizing the developments and user tests, the conclusions obtained based on the results will be written, and the possible steps to follow in the future will be detailed.
- Tool development:
 - Implementation of main functionalities: In this stage, the main functionalities of basic movements will be implemented, including integration with sensors and emitters, allowing interaction between them.
 - Implementation of visual aids: Visual aids will be developed to serve as references for designers, including graphic elements that facilitate the understanding of behaviors.
 - Testing and debugging: An iterative testing process will be carried out to ensure the functionality of the tool, correcting errors detected during its implementation.
- User testing:

- First phase of testing: Tests will be carried out with users who have not tried the tool before, following a test plan specified in the user evaluation section. The tests will focus on: detecting possible errors in the main functionalities, validating functionality, and evaluating usability and clarity.
- Second phase of testing: After implementing improvements based on feedback from the first test, a second verification test will be carried out.
- Correction and results: After analyzing the results of each test phase, the errors and difficulties encountered will be documented. Following this, the necessary corrections will be implemented to improve the results.

Conclusions and Future Work

Conclusions and future lines of work. This chapter contains the translation of Chapter 6.

Contribuciones Personales

En caso de trabajos no unipersonales, cada participante indicará en la memoria su contribución al proyecto con una extensión de al menos dos páginas por cada uno de los participantes.

En caso de trabajo unipersonal, elimina esta página en el fichero `TFGTeXiS.tex` (comenta o borra la línea `\include{Capitulos/ContribucionesPersonales}`).

Estudiante 1

Al menos dos páginas con las contribuciones del estudiante 1.

Estudiante 2

Al menos dos páginas con las contribuciones del estudiante 2. En caso de que haya más estudiantes, copia y pega una de estas secciones.

Apéndice A

Título del Apéndice A

Los apéndices son secciones al final del documento en las que se agrega texto con el objetivo de ampliar los contenidos del documento principal.

Apéndice **B**

Título del Apéndice B

Se pueden añadir los apéndices que se consideren oportunos.

Qué suerte tengo de tener algo que hace que decir adiós sea tan difícil.

Alan Alexander Milne

Este trabajo fin de grado no es solo un proyecto, es el reflejo de años de esfuerzo, ilusión y crecimiento.

