
**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 de comportamientos identificados en este tipo de juegos.

Palabras clave

Inteligencia Artificial, Unity, Enemigo, Maquinas de Estado, 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 behavior patterns identified in this type of games.

Keywords

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

Índice

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	1
1.3. Plan de trabajo	2
2. Estado de la Cuestión	5
2.1. Introducción a la Inteligencia Artificial en videojuegos	5
2.2. Técnicas de toma de decisiones en NPC's	6
2.2.1. Máquinas de estado finitas	8
2.2.2. Árboles de comportamiento	10
2.2.3. Goal-Oriented Action Planning	12
2.3. Análisis de herramientas para la creación de comportamientos inteligentes	14
2.3.1. Behavior Bricks	14
2.3.2. PlayMaker	15
2.4. Motores de videojuegos	17
2.4.1. Unity	17
2.4.2. Unreal Engine	18
2.4.3. Godot	20
2.4.4. GameMaker	21
2.5. Conclusiones	23
3. Diseño del Framework	25
3.1. Análisis de enemigos en videojuegos	26
3.1.1. Hollow Knight	26
3.1.2. Blasphemous	27
3.1.3. Bzzzt	27
3.1.4. Comportamientos relevantes detectados	28
3.2. Conclusiones del análisis y definición del framework	29
3.3. Actuadores	30
3.3.1. Movimiento	30
3.3.2. Spawner	31

3.3.3. Compatibilidad	32
3.4. Sensores	34
3.5. Daño	34
3.6. Máquina de Estados Finita	35
3.6.1. Estado	35
3.6.2. Sincronización con las animaciones	35
3.7. Ejemplos de uso	36
3.7.1. Bouncing Bunny	37
3.7.2. Spinning Rocks	37
3.7.3. Trunk Turret	38
3.7.4. Spline Chicken	39
3.7.5. Skywatch Eagle	41
3.8. Conclusiones del Diseño	42
4. Implementación	45
4.1. Tecnología utilizada	45
4.2. Arquitectura del sistema	46
4.3. Módulo de actuadores	47
4.3.1. Actuator	47
4.3.2. MovementActuator	47
4.3.3. SpawnerActuator	52
4.4. Módulo de sensores	52
4.4.1. Sensor	53
4.4.2. AreaSensor	54
4.4.3. CollisionSensor	54
4.4.4. DistanceSensor	55
4.4.5. TimeSensor	57
4.5. Módulo de daño	57
4.5.1. DamageEmitter	57
4.5.2. DamageSensor	59
4.6. Módulo de Máquina de Estados	60
4.6.1. FSM	60
4.6.2. Transition	60
4.6.3. State	60
4.7. Módulo de Animación	61
4.7.1. AnimatorManager	61
4.8. Módulo del Jugador	63
4.8.1. PlayerMovement	63
4.8.2. PlayerCollisionDetection	64
4.8.3. PlayerJump	65
4.8.4. Life	66
4.8.5. PlayerDistanceAttack	66
4.9. Distribución de la herramienta	67

4.10. Conclusiones del Módulo de Implementación	68
5. Evaluacion Con Usuarios	71
5.1. Objetivos y preguntas de investigación	71
5.2. Diseño de la Evaluación	72
5.2.1. Audiencia Objetivo	73
5.2.2. Duración y Entorno de Realización	73
5.2.3. Descripción de Tareas del Probador	73
5.2.4. Instrucciones Iniciales	74
5.2.5. Comportamiento del Investigador	74
5.2.6. Metodología	74
5.3. Resultados de las pruebas	76
5.3.1. Contexto de las sesiones	76
5.3.2. Datos cuantitativos	77
5.3.3. Datos cualitativos	77
5.4. Análisis de resultados	77
5.4.1. Claridad e intuición de la herramienta	77
5.4.2. Funcionalidad y utilidad práctica	78
5.4.3. Adaptación y extensibilidad	78
5.4.4. Conclusión general	79
6. Conclusiones	81
6.1. Síntesis de resultados	81
6.1.1. Diseño arquitectónico	81
6.1.2. Implementación y entorno	82
6.1.3. Validación con usuarios	82
6.2. Aportaciones y lecciones aprendidas	82
6.3. Trabajo futuro	83
6.4. Motivation	85
6.5. Objectives	85
6.6. Work Plan	86
7. Conclusions	89
7.1. Summary of Results	89
7.1.1. Architectural Design	89
7.1.2. Implementation and Environment	90
7.1.3. User Validation	90
7.2. Contributions and Lessons Learned	90
7.3. Future Work	91
8. Contribuciones Personales	93
8.1. Contribuciones de Cristina Mora Velasco	93
8.1.1. Antecedentes	93
8.1.2. Aportaciones	93

8.2. Contribuciones de Francisco Miguel Galván Muñoz	98
8.2.1. Antecedentes	98
8.2.2. Aportaciones	98

Índice de figuras

1.1.	Diagrama de planificación del desarrollo de la herramienta Enemy Behaviour 2D	4
2.1.	Comportamiento del jugador en <i>Pac-Man</i> , basado en una máquina de estados finita, extraído de Yannakakis y Togelius (2018).	9
2.2.	Ejemplo de maquinas de estado jerárquicas extraido de Borbor (2012).	10
2.3.	Árbol de comportamiento de Spore	12
2.4.	Ejemplo ilustrativo de GOAP donde el estado final es la eliminación de un enemigo.	14
2.5.	Ejemplo de uso de Behavior Bricks	15
2.6.	Ejemplo de uso de Play Maker	16
2.7.	Inspector de Unity, con una serie de componentes	18
2.8.	Blueprints en Unreal Engine	19
2.9.	Sistema VisualScript en Godot	20
2.10.	Drag and Drop en GameMaker	22
3.1.	Easing Function que describe la variación de la posición de un objeto con el movimiento Move To An Object	32
3.2.	Easing Function que describe la variación de la velocidad de un objeto con el movimiento Horizontal	33
3.3.	Ejemplo de estructura general de un enemigo basado en FSM.	36
3.4.	Escena de ejemplo donde observamos al enemigo en cuestión momentos antes de alcanzar la pared.	38
3.5.	Escena de ejemplo donde se aprecian dos enemigos realizando un movimiento circular, cada uno en un sentido diferente.	39
3.6.	Escena de ejemplo donde se aprecia una <i>Trunk Turret</i> disparando.	40
3.7.	Escena de ejemplo donde se aprecia una <i>Spline Chicken</i> siguiendo su trayectoria.	40
3.8.	Escena de ejemplo donde se aprecia a la <i>Skywatch Eagle</i> atacando al jugador.	42
4.1.	AreaSensor utilizado para enemigo que cae al detectar al jugador.	55
4.2.	Medición de distancia a través de la magnitud.	56

4.3. Medición de distancia en el eje X en su lado negativo.	57
4.4. Animator Controller con sus estados y transiciones.	63
4.5. Representación de las cajas de detección de colisiones del jugador . .	65

Índice de tablas

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 sino 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 un framework para el motor de videojuegos *Unity* que simplifique y agilice el proceso de creación de enemigos en juegos plataformas 2D. Este framework define una estructura modular de componentes y comportamientos basada en el análisis de enemigos comunes en este tipo de juegos, con el fin de separar completamente los roles de programación y diseño. De este modo, se facilita que personas sin conocimiento de programación

puedan desempeñar el rol de diseñador de enemigos.

Como exemplificación práctica del framework, se ha desarrollado una herramienta funcional en Unity que permite implementar y utilizar estos componentes de manera visual e intuitiva. La herramienta incluye un catálogo de comportamientos fácil de manejar para cualquier persona, así como un manual de usuario que explica claramente cada componente, su instalación y ejemplos de uso.

Para llevar a cabo este desarrollo, se ha seguido un plan de trabajo estructurado que abarca desde el estudio del entorno y la revisión de enemigos existentes, hasta la implementación de la herramienta, su validación con usuarios y el análisis de los resultados obtenidos.

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.
- Estudio de comportamientos de enemigos: Este estudio nos permitirá tener una base sólida para el diseño de la herramienta por medio de la identificación de similitudes en los comportamientos de distintos enemigos.

- Desarrollo de la herramienta:

- Diseño: En esta etapa, se definirá la arquitectura de la herramienta propuesta, describiendo las técnicas empleadas, esquemas de funcionamiento y organización de elementos principales.

- Implementación de funcionalidades principales: En esta etapa se implementarán las funcionalidades principales de los movimientos básicos incluyendo la integración con sensores y actuadores 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:
 - 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.
 - 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.

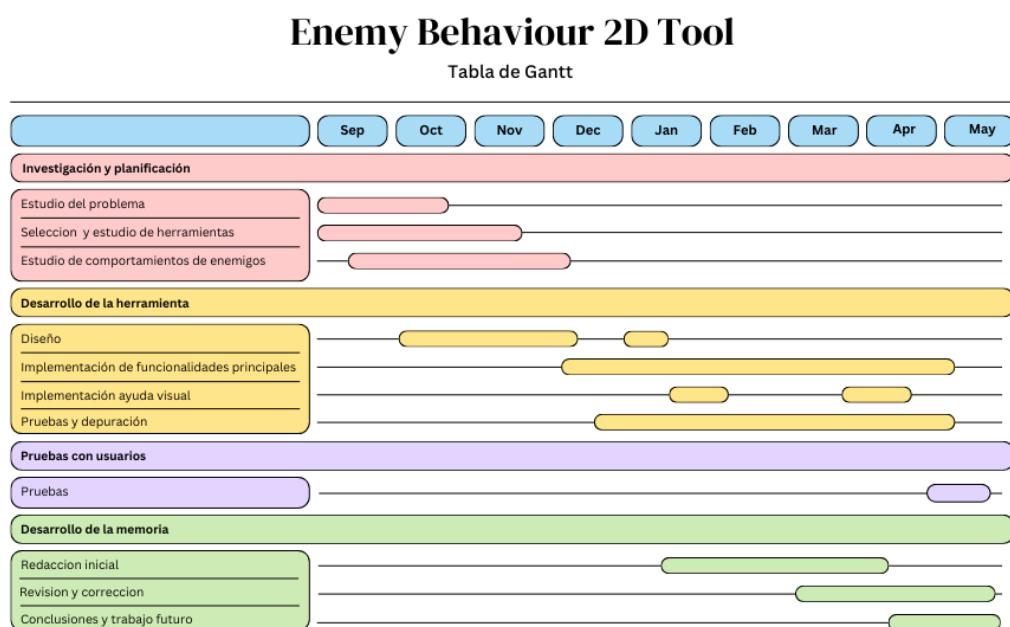


Figura 1.1: Diagrama de planificación del desarrollo de la herramienta Enemy Behaviour 2D

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 2D, ya sea para crear 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 en videojuegos

La Inteligencia Artificial (IA) en videojuegos se refiere a los algoritmos de toma de decisiones que controlan el comportamiento de los personajes dentro del juego, según lo define Bakkes en su tesis doctoral Bakkes (2010).

La IA desempeña un papel crucial, especialmente en entornos 2D donde la utilidad en los enemigos es lo que determina en mayor medida el nivel de dificultad y la jugabilidad del videojuego. Desde los inicios donde se presentaban enemigos con comportamientos simples fácilmente memorizable por los usuarios hasta la actualidad más compleja que logra enemigos “más humanos” y mejora la inmersión. El potencial de la IA y la razón por la que suscita tanto interés, radica en su capacidad de actuar de forma autónoma: no se limita a seguir instrucciones predefinidas, sino que es capaz de tomar decisiones adaptativas en función del contexto.

Por ejemplo, la IA puede adaptarse dinámicamente a las decisiones del jugador, como ocurre con el enemigo principal en *Hope* (2014), donde el comportamiento del Alien se ajusta de forma impredecible para mantener la tensión. También puede utilizarse para generar contenido procedural, como en juegos roguelike tipo *Hades*¹, o para entrenar agentes mediante redes neuronales profundas que imitan el estilo de

¹[https://hades.fandom.com/es/wiki/Hades_\(juego\)](https://hades.fandom.com/es/wiki/Hades_(juego))

conducción de jugadores humanos, como sucede en la saga *Forza Motorsport*².

En definitiva, el uso de la IA en videojuegos responde a una necesidad técnica y creativa: la de crear experiencias de juego más inmersivas, adaptativas, eficientes y realistas. Tal como se expone en Yannakakis y Togelius (2018), la IA no solo permite que el juego “juegue bien”, sino que también posibilita que lo haga de forma convincente y útil para diseñadores y jugadores por igual.

2.2. Técnicas de toma de decisiones en NPC's

En el ámbito del desarrollo de inteligencia artificial para videojuegos, una de las tareas más relevantes es la implementación de sistemas de toma de decisiones para personajes no jugables (NPCs). Estas técnicas permiten dotar a los NPCs de comportamientos coherentes, adaptativos y, en algunos casos, realistas, según el contexto del juego.

A continuación, se presentan tres técnicas (véase la Tabla 2.1) ampliamente utilizadas en esta área:

- Máquinas de estados finitos (FSM): Se basan en un conjunto finito de estados predefinidos, junto con transiciones entre ellos que dependen de condiciones específicas. Son especialmente útiles cuando los comportamientos pueden representarse de forma secuencial o reactiva. Su implementación es sencilla, pero su escalabilidad puede verse limitada en entornos muy complejos. En nuestro proyecto, esta técnica se ajusta de forma natural a la estructura de comportamiento que planteamos, por lo que será la base del sistema de decisión.
- Árboles de comportamiento (Behavior Trees): Proporcionan una estructura jerárquica y modular para organizar comportamientos complejos. Cada nodo del árbol representa una acción o una condición, permitiendo la reutilización de comportamientos y facilitando la depuración. Esta técnica es más flexible que las FSM y se utiliza en muchos videojuegos comerciales debido a su claridad estructural y su adaptabilidad.
- Planificación orientada a objetivos (Goal-Oriented Action Planning, GOAP): A diferencia de las técnicas anteriores, GOAP no se basa en una estructura fija de estados o árboles, sino que genera planes dinámicamente en función de los objetivos del agente y las acciones disponibles. Esto permite una mayor autonomía y adaptabilidad, aunque a costa de una mayor complejidad computacional. Se emplea en juegos donde los NPCs deben tomar decisiones más abiertas y razonadas, como en entornos con múltiples rutas o soluciones posibles.

Estas técnicas representan diferentes niveles de complejidad y flexibilidad, y la elección entre ellas depende del tipo de comportamiento que se desee implementar, así como de los recursos disponibles.

²https://forza.fandom.com/wiki/Forza_Wiki

Tabla 2.1: Comparativa de técnicas de toma de decisiones para NPCs

Técnica	Descripción	Ventajas	Desventajas
Máquinas de estados finitas (FSM)	Sistema basado en estados predefinidos y transiciones condicionadas. Cada estado representa un comportamiento específico del NPC.	Simplicidad, facilidad de implementación, adecuado para comportamientos predecibles.	Escalabilidad limitada, difícil de mantener con muchos estados y transiciones.
Árboles de comportamiento (Behavior Trees)	Estructura jerárquica de nodos que representan condiciones y acciones. Permiten modularidad y reutilización de comportamientos.	Modularidad, claridad, fácil depuración y reutilización de nodos.	Mayor complejidad que FSM, curva de aprendizaje más pronunciada.
GOAP (Goal-Oriented Action Planning)	Técnica basada en planificación dinámica en función de objetivos y acciones disponibles. Genera planes en tiempo real según el entorno.	Alta autonomía, adaptabilidad y comportamiento más realista.	Complejidad de implementación, mayor coste computacional.

2.2.1. Máquinas de estado finitas

En su obra, Ian Millington and John Funge (2016) describen las máquinas de estado finitas (FSM) como una de las herramientas más comunes y efectivas para construir IA en videojuegos. Una FSM se compone de un conjunto finito de estados, donde solo uno está activo en un momento dado, y cada estado contiene un comportamiento asociado así como reglas que determinan las transiciones a otros estados en función de eventos o condiciones.

En el contexto de los videojuegos, las FSM permiten definir con claridad cómo debe comportarse una entidad del juego en distintas situaciones, por ejemplo: caminando, atacando, huyendo o patrullando. La estructura garantiza que solo un estado esté activo a la vez, lo que simplifica la lógica de control y evita conflictos entre comportamientos simultáneos. A menudo, una FSM se representa como un grafo, donde los nodos representan los estados, que realizan las acciones y las aristas las transiciones posibles, activadas por eventos o condiciones del entorno. Esta estructura modular y determinista facilita tanto su diseño como su implementación y depuración.

Como se documenta en el artículo de Gopalakrishnan y Pradeep Gopalakrishnan y Pradeep (2021), 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*³ 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 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.

³https://pacman.fandom.com/es/wiki/Pac-Man_Wiki:Portada

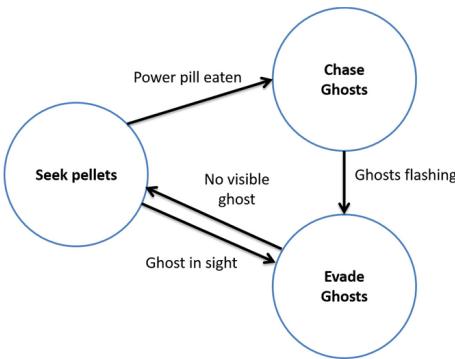


Figura 2.1: Comportamiento del jugador en *Pac-Man*, basado en una máquina de estados finita, extraído de Yannakakis y Togelius (2018).

- 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.

2.2.1.1. Máquinas de Estado Finitas Jerárquicas

La principal desventaja de las FSM es que son muy inflexibles y estáticas y, aunque se puede atenuar mediante la implementación de probabilidades o reglas que no estén tan claras a la hora de hacer las transiciones, siguen aumentando su complejidad progresivamente cuando aumenta su tamaño. Una forma de poder simplificar esa complejidad es dividiendo tareas complejas en otras más sencillas, permitiendo agrupar varias máquinas finitas dentro de otra máquina finita. Las Máquinas de Estados Finitos Jerárquicas (HFSM, por sus siglas en inglés) comparten la misma representación básica que las FSM tradicionales, pero se distinguen por permitir la anidación de estados dentro de otros estados, lo que introduce una estructura jerárquica en su diseño.

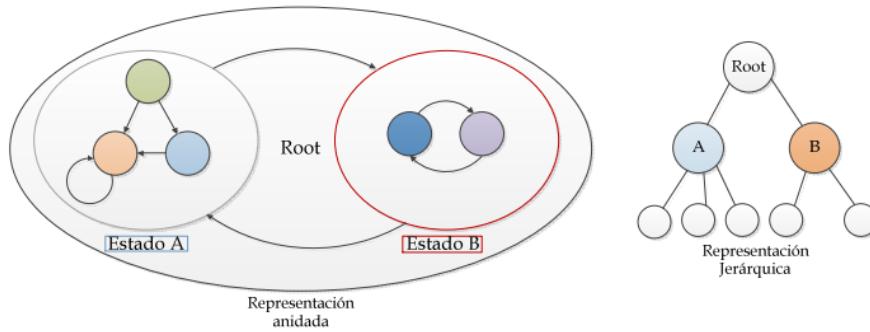


Figura 2.2: Ejemplo de maquinas de estado jerárquicas extraido de Borbor (2012).

2.2.2. Árboles de comportamiento

Un Árbol de Comportamiento (Behavior Tree o BT) es una técnica utilizada para modelar la toma de decisiones en inteligencia artificial, especialmente en videojuegos. Es conceptualmente similar a una Máquina de Estados Finitos, ya que también organiza el comportamiento en unidades que se activan de forma exclusiva (es decir, solo una parte del árbol está “activa” en cada momento). Sin embargo, en lugar de representar estados, los BT están formados por nodos que representan comportamientos, organizados jerárquicamente en forma de árbol. Cada nodo ejecuta una acción o toma una decisión, y el control fluye a través del árbol según unas reglas predefinidas.

Los nodos de un BT pueden dividirse en varias categorías, siendo las principales:

- Nodos hoja (leaf nodes): Acciones concretas que ejecuta el agente, como moverse a un punto” o “atacar al enemigo”.
- Nodos compuestos: Agrupan varios nodos hijos y determinan en qué orden deben ejecutarse (por ejemplo, secuencias o selectores).
- Nodos de control o decoradores: Modifican el comportamiento de los nodos hijos (por ejemplo, repetir un nodo mientras se cumpla una condición).

La principal ventaja respecto a las Máquinas de Estado Finitas 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 probados 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) y *Halo 2* (Gamasutra, 2005) como referencias en el uso de Árboles de comportamiento.

Un ejemplo no tan conocido de uso de Árboles de comportamiento en la industria del videojuego es *Spore* (Electronic Arts, 2008). *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 Árboles de comportamiento.

La gran diferencia en cómo Spore utiliza los Árboles de Comportamiento frente a juegos como Halo 2 es que separa el concepto de decider del de behavior. En Halo 2, los árboles están compuestos por behaviors (comportamientos que pueden ser grupales o individuales) e impulsos (transiciones entre comportamientos basadas en prioridades). Esta estructura, aunque funcional, tiende a generar problemas de escalabilidad, como:

- Dificultad para entender y mantener el árbol.
- Duplicación innecesaria de comportamientos.
- Poca reutilización del código.
- Aumento del riesgo de errores al introducir nuevas acciones o decisiones.

Para resolver estos problemas, el equipo de Maxis decidió separar completamente los nodos de decisión (deciders) de los nodos de acción (behaviors). Los deciders actúan como controladores que eligen qué behavior ejecutar en función del contexto, mientras que los behaviors se mantienen como bloques de acción reutilizables. Esta estructura modular y jerárquica no solo mejora la claridad del árbol, sino que también facilita su escalabilidad y mantenimiento en proyectos complejos como Spore.

La Figura 2.3 es un ejemplo de un BT sacado de las documentación⁴ que hay publicada del juego. En este árbol, los nodos se dividen en dos tipos principales:

- Nodos de decisión (deciders): Representados con forma de rombo (como *GUARD*, *EAT*, *IDLE*), son los encargados de seleccionar cuál de sus nodos hijos ejecutar en función del contexto o condiciones actuales. Evalúan a sus hijos en orden y seleccionan el primero que sea válido.
- Nodos de acción (behaviors): Representados con rectángulos (como *FIGHT*, *PATROL*, *YELL_FOR_HELP*), son los encargados de ejecutar una acción concreta dentro del mundo del juego. Estas acciones son atómicas y no contienen lógica de decisión propia.

El nodo raíz (*ROOT*) inicia la ejecución del árbol. El flujo de control comienza en este nodo y se propaga hacia abajo evaluando nodos hijos de izquierda a derecha. Cuando un decider es activado, se encarga de decidir cuál de sus hijos (otros deciders o behaviors) debe activarse en ese momento. Si ninguno de los hijos es adecuado, se retorna al nodo anterior, lo que permite reconsiderar otras opciones disponibles. Este enfoque permite que el comportamiento sea flexible y jerárquico.

⁴https://chrisshecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs

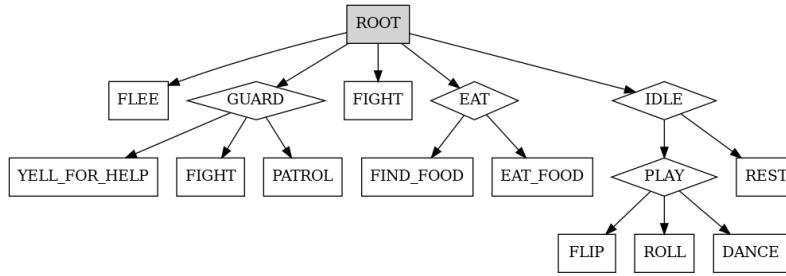


Figura 2.3: Ilustración de un árbol de comportamiento tomado del diseño del videojuego *Spore*, utilizado para modelar decisiones complejas en NPC's.⁵

Gracias a esta organización, se logra modularidad y escalabilidad: por ejemplo, el decider *IDLE* encapsula comportamientos distintos como *REST* o *PLAY*, y este último a su vez contiene subacciones como *FLIP*, *ROLL* o *DANCE*. Esta jerarquía hace posible reutilizar comportamientos en distintos contextos y facilita el mantenimiento del árbol.

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 para alcanzar el objetivo designado. Esta técnica fue desarrollado en el *MIT* por *Orkin (2004)* 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 será 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, 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 Máquina de Estado Finita, pero en GOAP las acciones y metas no van de la mano, sino 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*⁶. El propio Jeff Orkin explica que en el videojuego se quería llegar a una complejidad en la IA a la que las Máquinas de Estado Finitas 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 es perseguido por un enemigo, el enemigo puede dinámicamente

⁵Fuente: Documentación de diseño de Spore https://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs

⁶<https://www.gdcvault.com/play/1013282/Three-States-and-a-Plan>

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.

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*, como se menciona en Orkin (2004), 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 cómo 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.
- 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 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 2.4. 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 al mundo desde un estado en el que el enemigo está vivo hasta otro en el que este está muerto.

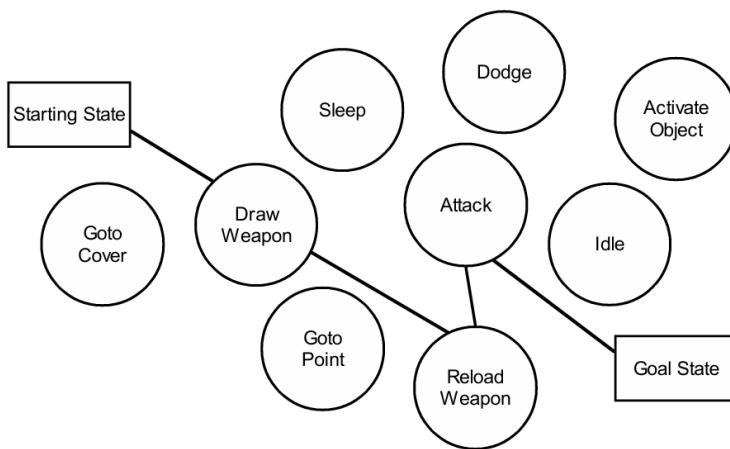


Figura 2.4: Ejemplo ilustrativo de GOAP donde el estado final es la eliminación de un enemigo.

Este proceso se aborda como un *pathfinding*, la búsqueda de un camino válido que nos lleve desde el estado inicial al estado final. El planificador tiene que encontrar un plan válido y no siempre este es el esperado por el usuario.

2.3. Análisis de herramientas para la creación de comportamientos inteligentes

Tras abordar técnicas usadas para la toma de decisiones, se ha visto que en ocasiones la complejidad de crear esos algoritmos no es del todo trivial. Por ello, surge la necesidad de crear herramientas que ayudasen a los desarrolladores a crear Inteligencias Artificiales de toma de decisiones, concretamente centrada en la creación de NPCs. Teniendo en cuenta que la creación de IA 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 que ayudan a dicha creación como ejemplo.

2.3.1. Behavior Bricks

Behavior Bricks es una herramienta de scripting visual diseñada para el motor de videojuegos *Unity*, orientada a facilitar la implementación de comportamientos complejos en entornos interactivos. Esta herramienta permite a los usuarios modelar tanto Máquinas de Estados Finitos (FSM, por sus siglas en inglés) como Árboles de Comportamiento (BT), utilizando una interfaz visual intuitiva (véase la Figura 2.5).

Uno de los principales objetivos de *Behavior Bricks* es fomentar la colaboración efectiva entre diseñadores y programadores, superando las barreras que suelen surgir entre estos perfiles durante el desarrollo de videojuegos. Su enfoque visual

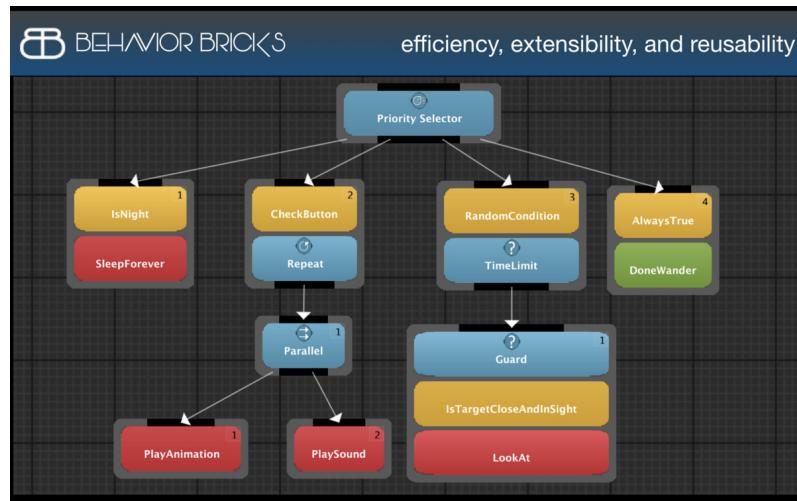


Figura 2.5: Ejemplo de uso de Behavior Bricks

reduce la necesidad de codificación directa, facilitando la comunicación de ideas y la implementación de comportamientos complejos sin depender exclusivamente de programación textual.

Otro aspecto destacado de *Behavior Bricks* es su diseño modular. Cada componente puede ser modificado de forma independiente, lo que promueve la reutilización de elementos en distintos proyectos. Esta modularidad permite, además, que un estado o comportamiento pueda agrupar internamente otros estados o comportamientos, favoreciendo la creación de estructuras jerárquicas y reutilizables.

Desde el punto de vista del rendimiento, *Behavior Bricks* ha sido optimizado para minimizar su impacto en la ejecución del juego. Emplea estrategias eficientes de gestión de memoria, incluyendo el uso compartido de datos entre distintos estados o comportamientos cuando es posible, con el fin de ahorrar recursos computacionales.

Esta herramienta se basa en un modelo integrador de Máquinas de Estados y Árboles de Comportamiento, lo que proporciona flexibilidad y potencia expresiva al sistema. Este enfoque ha sido presentado en la literatura académica por Sagredo-Olivenza et al. (2016).

2.3.2. PlayMaker

PlayMaker⁷ es un editor visual de FSM para Unity 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 2.6, seguido de un estado predeterminado llamado *State 1*, al cual se transiciona al eje-

⁷<https://hutonggames.com/>

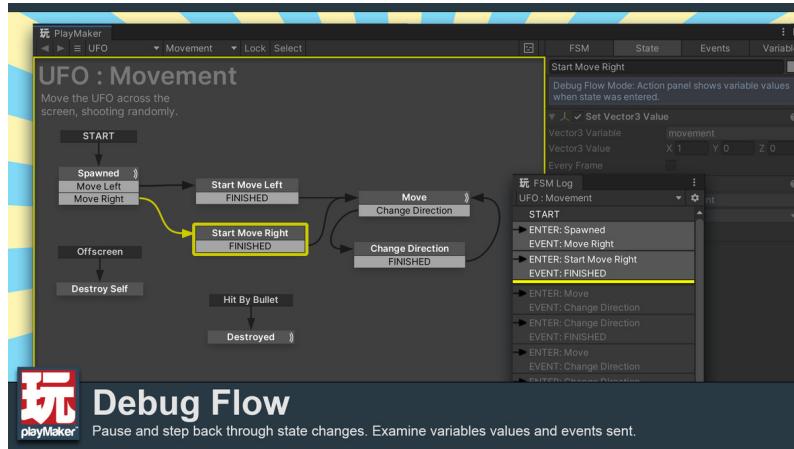


Figura 2.6: Ejemplo de uso de Play Maker

cutar *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 IA, 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*⁸, *Firewatch*⁹ la aventura narrativa del estudio norteamericano Campo Santo o el juego de plataforma de los creadores de *Limbo*, *INSIDE*¹⁰.

⁸<https://www.hollowknight.com/>

⁹<https://www.firewatchgame.com/>

¹⁰https://inside.fandom.com/wiki/Inside_Wiki

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, para así, razonar justificadamente cual se ajusta más a las necesidades de este proyecto. Todos los motores escogidos tienen gran renombre en la industria ofreciendo gran variedad y distintas filosofías de diseño.

2.4.1. Unity

*Unity*¹¹ 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*¹², *Cuphead*¹³ y *Genshin Impact*¹⁴. La versión más actual del motor, *Unity 6.0*, incorpora mejoras en su sistema de renderizado, herramientas avanzadas de optimización y un motor de físicas más eficiente.

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 lógica de juego 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, son usados para implementar componentes. Estos componentes se adjuntan a los objetos (*GameObjects*) dentro del editor (Figura 2.7) y pueden controlar aspectos como la física, la inteligencia artificial y las interacciones del jugador.

Por otro lado, el sistema de programación visual *Bolt*¹⁵ 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*

¹¹<https://unity.com>

¹²<https://www.hollowknight.com>

¹³<https://cupheadgame.com>

¹⁴<https://genshin.hoyoverse.com/es>

¹⁵<https://docs.unity3d.com/2019.3/Documentation/Manual/VisualScripting.html>

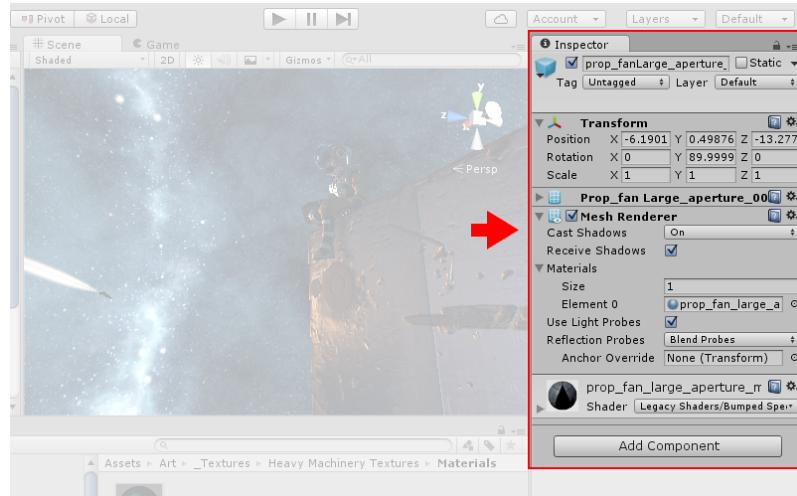


Figura 2.7: Inspector de Unity, con una serie de componentes

(*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 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*¹⁶, desarrollado por *Epic Games*¹⁷, es uno de los motores de videojuegos más potentes y utilizados en la industria de los videojuegos en títulos como

¹⁶<https://www.unrealengine.com/es-ES>

¹⁷<https://www.epicgames.com/site/es-ES/home>

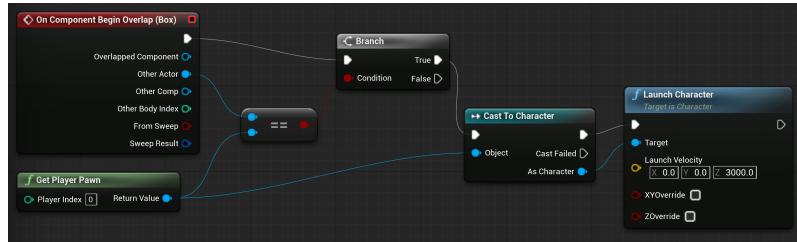


Figura 2.8: Blueprints en Unreal Engine

la saga *Hellblade*¹⁸ o el éxito mundial *Fortnite*¹⁹ 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 2.8) 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 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.

¹⁸https://thehellblade.fandom.com/wiki/Hellblade:_Senua%27s_Sacrifice

¹⁹<https://www.fortnite.com/?lang=es-ES>

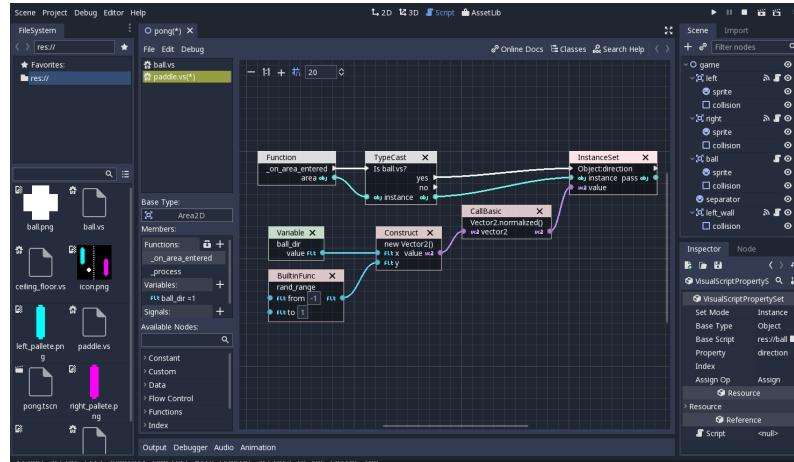


Figura 2.9: Sistema VisualScript en Godot

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

Godot²⁰, 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 llamado *VisualScript* (Figura 2.9), 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 extensión²¹.

²⁰<https://godotengine.org/>

²¹https://docs.godotengine.org/es/3.5/tutorials/scripting/visual_script/index.html

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 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. Al no requerir licencias, lo convierte en una excelente opción para proyectos indie para estudios de pequeño y mediano tamaño. Aunque tradicionalmente ha sido asociado con el desarrollo indie como el roguelite *Brotato*²². Su adopción ha crecido considerablemente siendo usado por títulos de mayor proyección como *Marvel Snap*²³. 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*²⁴ 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 la creación de títulos exitosos como *Undertale*²⁵, *Hotline Miami*²⁶ o *Katana ZERO*²⁷.

A diferencia de otros motores, GameMaker ofrece dos formas principales de desarrollo: un sistema de programación visual basado en eventos llamado *Drag and*

²²https://brotato.wiki.spellsandguns.com/Brotato_Wiki

²³<https://www.marvelsnap.com/home>

²⁴<https://gamemaker.io/en>

²⁵<https://undertale.com>

²⁶https://store.steampowered.com/app/219150/Hotline_Miami/

²⁷<https://katanazero.com>

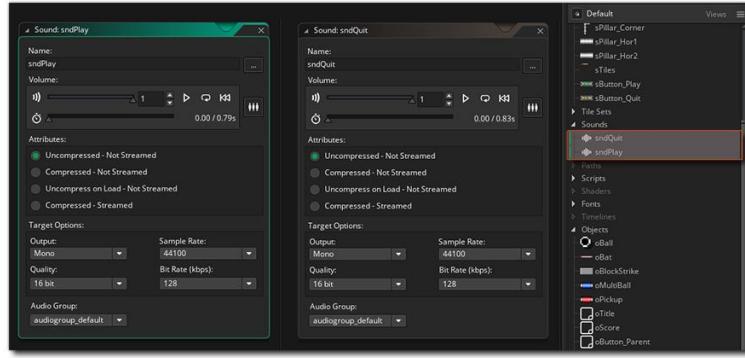


Figura 2.10: Drag and Drop en GameMaker

Drop 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 2.10), 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 rapidez 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

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 toma de decisiones en NPC's 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 nuestro caso, hemos optado por utilizar máquinas de estados finitas, por su idoneidad en contextos de decisión estructurada y controlada.

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

Diseño del Framework

El objetivo principal de este Trabajo de Fin de Grado es facilitar el diseño de enemigos en videojuegos de plataformas en 2D, permitiendo que diseñadores sin conocimientos de programación puedan crear comportamientos complejos de manera visual e intuitiva. Para ello, se ha desarrollado una herramienta sobre el motor *Unity* que implementa un framework modular basado en componentes de comportamiento reutilizables.

Este capítulo describe el diseño conceptual de ese framework. En primer lugar, se analiza el contexto en el que se enmarca la herramienta y su utilidad dentro del desarrollo de videojuegos. Para ello, se presentan varios títulos de referencia y se identifican patrones comunes en el diseño de enemigos. A continuación, se explica cómo se estructura el framework, qué elementos lo componen y qué principios han guiado su desarrollo. Finalmente, se muestran ejemplos de aplicación que ilustran su uso en distintos tipos de enemigos.

Como se señala en Patashnik (2014), 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 e incluso contribuye a la narrativa. Un enemigo puede requerir estrategias específicas para vencerlo o tener comportamientos complejos que enriquecen la experiencia del jugador.

El framework propuesto busca facilitar la creación de este tipo de enemigos inteligentes en 2D mediante una colección de comportamientos básicos y combinables. Está orientado a diseñadores, estudiantes o desarrolladores que deseen construir enemigos sin necesidad de escribir código, fomentando así una separación clara entre los roles de diseño y programación.

3.1. Análisis de enemigos en videojuegos

El análisis de diversos documentos muestra que la mejor forma de hacer que un enemigo destaque y aporte valor al juego es dotarlo de comportamientos únicos. Cada enemigo se define por una combinación específica de estos comportamientos, que determinan lo que puede o no puede hacer, y permiten diferenciarlo de otros. A lo largo del tiempo, estos comportamientos se han vuelto más sofisticados, lo que ha aumentado también la complejidad del trabajo de diseño. Para abordar esta dificultad, se ha propuesto una herramienta basada en comportamientos sencillos y precisos, que reduce la carga de trabajo sin sacrificar expresividad.

En este contexto, se entiende por enemigo cualquier entidad que represente una amenaza o dificultad para el jugador, afectando negativamente su progreso dentro del juego. Esto incluye no solo enemigos clásicos como soldados o monstruos, sino también elementos del entorno como pinchos, lava u obstáculos móviles. Asimismo, se ha optado por segmentar los enemigos según sus comportamientos, tratando como entidades separadas aquellos elementos que tradicionalmente se considerarían un único enemigo por su presentación conjunta. Por ejemplo, una tubería y una gota de ácido, o un pistolero y sus balas, se modelan de forma independiente en nuestro framework, ya que sus lógicas de comportamiento son autónomas y distintas.

Para fundamentar el diseño de esta herramienta, se ha realizado un análisis de los enemigos distintos videojuegos de plataformas en 2D. Los títulos seleccionados fueron *Hollow Knight*, *Blasphemous* y *Bzzzt*. En los dos primeros se examinó el comportamiento individual de enemigos comunes (excluyendo jefes), mientras que el tercero se utilizó como caso de validación para comprobar que los comportamientos identificados eran suficientes para modelar los enemigos más frecuentes del juego.

Estos enemigos han sido seleccionados por su simplicidad estructural y su potencial para ser descompuestos en comportamientos básicos reutilizables dentro del framework.

3.1.1. Hollow Knight

Hollow Knight es un *Metroidvania* en 2D desarrollado y auto publicado por el estudio australiano *Team Cherry*¹. Su versión inicial para ordenador se lanzó en 2017. El jugador controla al *Caballero*, que explora un reino subterráneo de insectos abandonado. Derrotando enemigos por medio de distintas habilidades que se van desbloqueando, se descubre el mundo y los secretos que este oculta.

A continuación se describen algunos de los enemigos estudiados:

- **Reptacillo:** Enemigo básico que patrulla una zona caminando en línea recta y cambiando de dirección al colisionar con un obstáculo.

¹<https://www.teamcherry.com.au/>

- **Cáscara errante:** Merodea pacíficamente hasta que detecta al jugador. Una vez detectado, realiza una embestida rápida en línea recta.
- **Vengamosca:** Enemigo volador que permanece revoloteando en una zona determinada y que, en caso de detectar al jugador si este se acerca lo suficiente, lo perseguirá.
- **Gruzzer:** Enemigo volador que cambia de dirección únicamente al golpearse con paredes y objetos.

3.1.2. Blasphemous

Desarrollado por el estudio sevillano *The Game Kitchen*² y publicado por Team17, *Blasphemous*³ llegó a Windows, Nintendo Switch, PlayStation 4 y Xbox One el 10 de septiembre de 2019. El jugador encarna al *Penitente* el único superviviente en la tierra de Cvstodi. Atrapado en un ciclo de penitencia de muerte y resurrección, el Penitente tendrá que liberar al mundo del destino que le espera.

Como enemigos analizados en este caso encontramos:

- **Enraged Pilgrim:** Enemigo terrestre que permanece inmóvil hasta que el jugador entra en un rango de detección determinado. Una vez activado, comienza a caminar lentamente hacia el jugador. Cuando se encuentra a corta distancia, ejecuta un ataque cuerpo a cuerpo utilizando un cuchillo.
- **Spikes:** Elemento estático del entorno que actúa como obstáculo letal. Consiste en una superficie con pinchos que, al entrar en contacto con el jugador, provoca su derrota inmediata.
- **Shooting Trap:** Trampa fija que lanza proyectiles a través de una abertura frontal. Dispara de forma continua en una única dirección, con una frecuencia constante y sin necesidad de detección del jugador.
- **Acolyte:** Enemigo terrestre que patrulla una zona y, al detectar al jugador, se detiene brevemente para luego realizar una embestida en su dirección.

3.1.3. Bzzzt

*Bzzzt*⁴ es un plataformas de acción de estética retro desarrollado casi íntegramente por el checo Karel Matějka y publicado por Cinemax. Se estrenó para PC en noviembre de 2023 y para Nintendo Switch en septiembre de 2024. El jugador, en el papel de un robot, debe frustrar los planes del villano Badbert y rescatar a sus creadores, los doctores Emily y Norbert, atravesando 52 niveles llenos de trampas y enemigos de estilo arcade.

En este caso los enemigos son:

²<https://thegamekitchen.com/>

³<https://thegamekitchen.com/blasphemous/>

⁴<https://store.steampowered.com/app/1293170/BZZZT/>

- **Saw Drone:** Enemigo volador que se desplaza siguiendo un patrón circular o semicircular. Causa daño al contacto y no reacciona a la presencia del jugador.
- **Cannon Turret:** Trampa fija que dispara proyectiles a intervalos regulares en una dirección determinada. No detecta al jugador, pero su ritmo constante obliga al jugador a calcular el momento adecuado para pasar.
- **Walking Bot:** Enemigo terrestre que patrulla horizontalmente entre dos límites. Cambia de dirección al chocar con obstáculos o al llegar al borde de la plataforma.
- **Chasing Bot:** Enemigo que, al detectar al jugador en un cierto rango, cambia su patrón de patrulla por una persecución directa. Su velocidad puede aumentar durante la persecución.
- **Spike Block:** Obstáculo estático que actúa como una trampa pasiva. Causa daño letal al jugador si entra en contacto con él. Puede estar en el suelo, techo o paredes.

3.1.4. Comportamientos relevantes detectados

El análisis de los enemigos presentes en los tres videojuegos permitió identificar una serie de patrones de comportamiento comunes entre ellos:

- *Patrulla:* Movimiento horizontal constante entre dos puntos, cambiando de dirección al chocar con un obstáculo o al alcanzar un borde. Este comportamiento está presente en enemigos como el Reptacillo (*Hollow Knight*), el Acolyte (*Blasphemous*) y el Walking Bot (*Bzzzt*).
- *Embestida lineal:* Una vez que detectan al jugador, interrumpen su comportamiento actual y se lanzan rápidamente en línea recta para atacarlo. Se observa en enemigos como la Cáscara errante (*Hollow Knight*) y el Acolyte (*Blasphemous*).
- *Persecución:* Enemigos que, al detectar al jugador, comienzan a seguirlo directamente, adaptando su trayectoria en tiempo real. Este comportamiento se encuentra en la Vengamosca (*Hollow Knight*) y el Chasing Bot (*Bzzzt*).
- *Torreta:* Disparo continuo o periódico de proyectiles en una dirección fija, sin depender de la posición del jugador. Representado por la Shooting Trap (*Blasphemous*) y la Cannon Turret (*Bzzzt*).
- *Rotación fija:* Movimiento circular o semicircular alrededor de un punto o eje, sin modificar el patrón según la presencia del jugador. Ejemplificado por el Saw Drone (*Bzzzt*).
- *Trampa estática letal:* Elementos del entorno que causan la derrota del jugador al contacto directo, sin movimiento ni detección. Este patrón se encuentra en los Spikes (*Blasphemous*) y el Spike Block (*Bzzzt*).

3.2. Conclusiones del análisis y definición del framework

El estudio de los enemigos presentes en los videojuegos *Hollow Knight*, *Blasphemous* y *Bzzzt* permitió identificar una serie de patrones recurrentes en su comportamiento. Estos patrones han sido agrupados en categorías comunes, lo que permite abstraer el comportamiento de cada enemigo en términos funcionales.

A partir de esta observación, se ha definido un framework modular que permite construir enemigos a partir de tres componentes fundamentales:

- **Sensores:** Son los mecanismos que permiten a un enemigo percibir información de su entorno. Definen condiciones de activación para ciertos comportamientos.
- **Actuadores:** Son las acciones que un enemigo puede ejecutar en un momento determinado. Incluyen movimientos, ataques y generación de nuevos enemigos, entre otros.
- **Sistema de daño:** Determina cómo un enemigo recibe y procesa el daño, incluyendo la interacción con elementos ofensivos del entorno (como ataques del jugador), invulnerabilidad temporal, reacciones visuales o físicas, y la gestión de su salud.

Esta separación en sensores, actuadores y daño permite una construcción flexible y reutilizable de comportamientos. En lugar de diseñar enemigos de forma específica para cada situación, es posible combinar estos elementos de forma modular para crear una gran variedad de comportamientos complejos.

En esta sección, se presentan los principales **actuadores** identificados a partir del análisis anterior, que constituyen la base para definir el comportamiento activo de los enemigos. Posteriormente, se describirán los **sensores**, encargados de activar estos actuadores según las condiciones del entorno. A continuación, se detalla el sistema de **daño**, necesario para gestionar las interacciones ofensivas entre el jugador y los enemigos y sus consecuencias.

Por último, una vez definidos los sensores, actuadores y el sistema de daño como elementos fundamentales del comportamiento enemigo, se introduce el concepto de **estado**, entendido como una combinación coherente de acciones que un enemigo puede ejecutar en un momento determinado, encapsuladas en los actuadores. Esta noción permite estructurar y organizar el comportamiento de forma modular. A partir de ello, se construye la Máquina de Estados Finita (FSM), el componente central de la lógica de control del enemigo. Cada enemigo contará con su propia FSM, la cual gestiona sus diferentes estados y las transiciones entre ellos en función de las señales recibidas por los sensores. Esta arquitectura permite representar comportamientos complejos de manera flexible, escalable y reutilizable dentro del framework.

propuesto.

3.3. Actuadores

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 (spawners). Estas habilidades no siempre son compatibles entre sí, por lo que es necesario especificar las relaciones de compatibilidad mediante una tabla.

En nuestro diseño, distinguimos dos tipos principales de actuadores: de movimiento, que definen cómo se desplazan los enemigos por el entorno, y los spawners, que les permiten generar otras entidades dentro del juego.

3.3.1. 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 diseñados, junto con una breve descripción.

- **Horizontal:** Desplaza al enemigo horizontalmente, hacia la izquierda o derecha.
- **Vertical:** Desplaza al enemigo verticalmente, hacia arriba o abajo.
- **Directional:** Mueve al enemigo en una dirección definida por un ángulo, siendo el valor cero un movimiento hacia la derecha e incremental en el sentido antihorario.
- **Circular:** Hace que el enemigo siga un movimiento circular alrededor de un punto, describiendo una circunferencia cerrada si el ángulo es igual a 360 y, en caso de ser menor, actuando como péndulo.
- **Move To A Point:** Dirige al enemigo hacia puntos concretos no actualizables. Se puede configurar por medio de una lista, donde se indican los puntos concretos a recorrer o por medio de un área, donde se elegirá aleatoriamente un punto y al llegar a él, se escogerá otro.
- **Move To An Object:** Desplaza al enemigo hacia una entidad que puede estar en movimiento.

- **Spline Follower** : Permite al enemigo seguir una trayectoria definida por una curva suave que se genera a partir de un conjunto de puntos de control, denominada *spline*. Una spline es una curva que conecta estos puntos de control mediante segmentos continuos y suavizados. Cada punto puede incluir manejadores (handlers) para controlar la tangente de entrada y salida, lo que permite ajustar con precisión la forma de la curva. Esta técnica proporciona trayectorias fluidas y naturales, ideales para movimientos complejos o decorativos.

Todos los tipos de movimiento pueden configurarse para ejecutarse de forma acelerada. Para ello, se emplea una función de aceleración/suavizado (*Easing Function*), una función que define cómo varían en un tiempo definido tanto la velocidad como la posición del enemigo.

Dependiendo del tipo de movimiento, esta función puede aplicarse de dos formas:

- En movimientos como **Horizontal**, **Vertical**, **Directional** o **Circular** (Figura 3.2), la *Easing Function* afecta principalmente a la velocidad, modificándola teniendo en cuenta una velocidad objetivo y un tiempo para llegar a ella.
- En movimientos como **Move To A Point** y **Move To An Object** (Figura 3.1), la *Easing Function* influye directamente en la interpolación de la posición, alterando cómo el enemigo se desplaza entre el origen y el destino en un tiempo determinado.

Este enfoque permite controlar el dinamismo del movimiento, proporcionando una mayor expresividad y naturalidad en el comportamiento de los enemigos. Por ello, se incluye un catálogo⁵ de distintas *Easing Functions* que permite variar la forma en que se definen y perciben los movimientos.

La influencia de las *Easing Functions* en cada tipo de movimiento, así como las imágenes utilizadas para ilustrarlas, se explicará con mayor profundidad en el apartado de Implementación (Capítulo 4).

3.3.2. Spawner

Spawner es la capacidad que poseen algunos enemigos para generar otras unidades enemigas de forma independiente, actuando como entidades que crean nuevas amenazas dentro del juego. Esta funcionalidad añade dinamismo y complejidad a la jugabilidad, permitiendo, por ejemplo, que una torreta dispare proyectiles o que un enemigo invoque aliados en mitad del combate.

Para diseñar un sistema de generación versátil, es necesario establecer un conjunto de reglas claras que definan cómo y cuándo debe producirse esta generación. A continuación, se plantean estas reglas como preguntas clave de diseño:

- **¿Con qué frecuencia se generan nuevas unidades?**
Cada entidad generadora establece un intervalo de espera entre una generación

⁵<https://gist.github.com/cjddmut/d789b9eb78216998e95c>

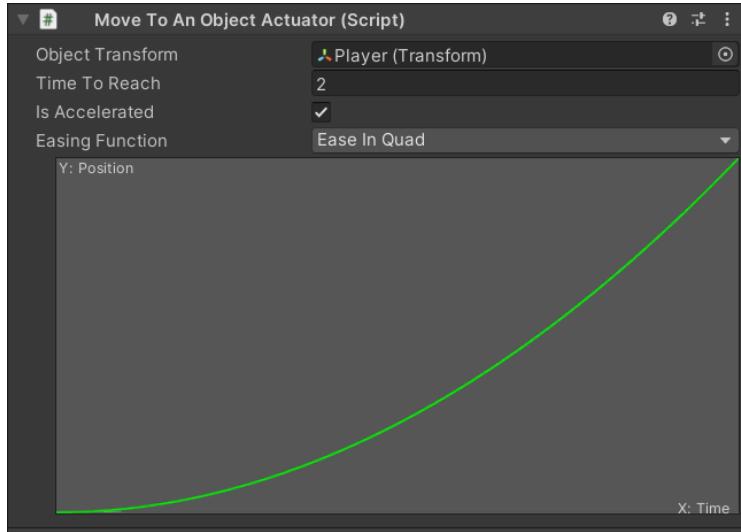


Figura 3.1: Easing Function que describe la variación de la posición de un objeto con el movimiento Move To An Object

y la siguiente. Este intervalo define el ritmo con el que aparecen las nuevas unidades, afectando directamente al nivel de presión sobre el jugador.

- **¿Cuántas unidades se pueden generar?**

El sistema permite limitar la cantidad total de unidades generadas o, si se desea, habilitar una generación ilimitada. Esta decisión influye tanto en el balance como en la duración del desafío que representa el generador.

- **¿Dónde aparecen las unidades generadas?**

Cada entidad generadora puede estar asociada a uno o varios puntos del escenario. En cada ciclo de generación, una nueva unidad aparece en estos puntos previamente definidos, lo que permite controlar espacialmente el comportamiento del sistema.

- **¿Cómo se produce la generación de forma visual o interactiva?**

El sistema de generación puede integrarse con las animaciones de la entidad generadora. Esto permite sincronizar la aparición de nuevas unidades con acciones visuales específicas (como una animación de ataque), aportando coherencia visual y reforzando la inmersión del jugador.

El comportamiento del generador se rige por estas reglas, evaluando de forma continua si las condiciones necesarias se cumplen. Solo entonces se permite la creación de nuevas unidades, asegurando que el sistema se mantenga bajo control y alineado con la experiencia de juego deseada.

3.3.3. Compatibilidad

La versatilidad de la herramienta se potencia al permitir la combinación de diversos actuadores simultáneamente. En este sentido, un enemigo podría perfectamente

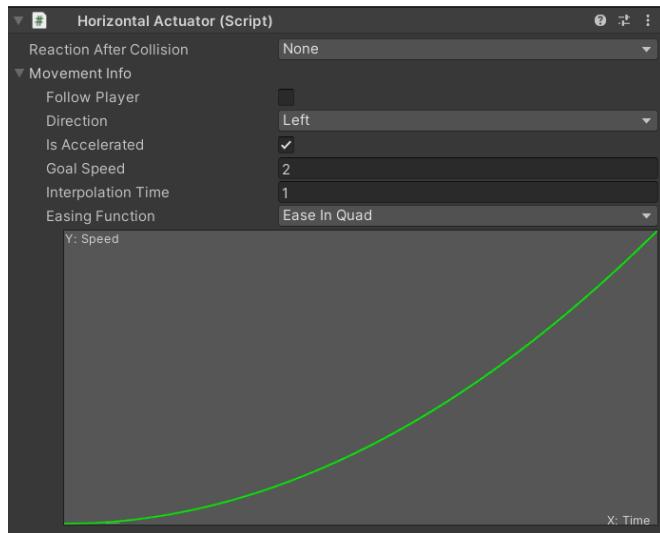


Figura 3.2: Easing Function que describe la variación de la velocidad de un objeto con el movimiento Horizontal

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 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.

Tabla 3.1: Matriz de compatibilidad de movimientos

	Horizontal	Vertical	Directional	Circular	Move To A Point	Move To An Object	Spline Follower	Spawner
Horizontal	Grey							
Vertical	Green	Red						
Directional	Red	Red	Grey					
Circular				Red				
Move To A Point					Red			
Move To An Object						Red		
Spline Follower	Red						Green	
Spawner	Green							Green

Leyenda de colores: Verde = combinación compatible; Rojo = combinación no compatible; Gris = combinación redundante.

3.4. Sensores

El término sensor se define como los mecanismos mediante los cuales los personajes interactúan con su entorno y entre sí. Podemos definir el término sensor como el elemento que sirve para que el enemigo conozca el entorno y reciba información de los cambios que se producen en el mismo. Son los responsables de activar las transiciones entre estados, es decir, se encargan de cambiar de un comportamiento predefinido a otro. A continuación se enumerarán los diferentes sensores disponibles en la herramienta.

- **Area:** Detecta si un objeto entra o sale en una zona de detección.
- **Collision:** Detecta colisiones físicas con otros objetos.
- **Distance:** Evalúa si un objeto se encuentra dentro o fuera de una distancia específica con respecto a otro. Esta condición puede interpretarse de dos maneras distintas según su configuración:
 - *Distancia absoluta:* Se mide la distancia total, sin importar la dirección. Es útil para detectar proximidad general.
 - *Distancia por eje:* Permite verificar la distancia solo en un eje determinado (por ejemplo, solo en el eje X), ignorando la separación en el Y. Esta modalidad resulta útil para detectar alineamientos específicos (como estar “a la misma altura” o “en la misma línea horizontal”).
- **Time:** Detecta cuando ha transcurrido un tiempo determinado.

3.5. Daño

Se define daño como la consecuencia negativa que recibe una entidad con respecto a su vida. Esto se produce como la consecuencia de una colisión entre dos entidades del juego, una que hace daño y otra que lo recibe.

Existen diferentes tipos de daño y parámetros específicos que determinan cómo las entidades 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 por qué emitir daño. Para reflejar esta diferenciación se han creado los siguientes dos componentes:

- **Damage Sensor:** Detecta si el objeto ha recibido daño.
- **Damage Emitter:** Efectúa daño.

Además, se puede distinguir entre tres tipos de daños:

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

3.6. Máquina de Estados Finita

La Máquina de Estados Finita (FSM, por sus siglas en inglés *Finite State Machine*) constituye el núcleo de la lógica de comportamiento de los enemigos en el framework propuesto. Su función principal es organizar y controlar el flujo de acciones que puede realizar un enemigo a lo largo del tiempo, de forma estructurada, flexible y escalable. Cada enemigo cuenta con su propia instancia de FSM.

La FSM se compone de un estado inicial, el cual encapsula una configuración concreta de comportamiento, así como sus posibles **transiciones** hacia otros estados. Esta estructura permite modelar de forma clara y modular la lógica interna de cada enemigo.

3.6.1. Estado

Un estado representa una situación concreta en la que se encuentra el enemigo, definida por un conjunto específico de actuadores que se ejecutan mientras permanece en dicho estado. Además, cada estado incluye sus propias transiciones y puede incorporar emisores de daño.

En particular, cada estado puede incluir uno o varios emisores de daño, que son los encargados de infligir daño a otras entidades del juego durante la ejecución del estado. Esto permite, por ejemplo, que un enemigo cause daño únicamente cuando se encuentra atacando, o que un área peligrosa esté activa solo durante determinados comportamientos.

3.6.1.1. Transiciones

Cada estado contiene sus propias transiciones, que determinan a qué otros estados puede cambiar el enemigo y bajo qué condiciones. Estas condiciones están definidas por los sensores, que detectan información relevante del entorno.

Cuando se cumple la condición de una transición, el estado actual cede el control al nuevo estado indicado, permitiendo una evolución fluida y dinámica del comportamiento del enemigo.

3.6.2. Sincronización con las animaciones

El comportamiento de un enemigo no solo debe ser funcional, sino también comprensible y coherente para el jugador. Por ello, los actuadores incluidos en cada estado pueden modificar la representación visual del enemigo mediante animaciones. Esta sincronización garantiza que lo que el enemigo está haciendo internamente, también se refleje externamente de forma clara.

Cada vez que un enemigo realiza una acción específica de un actuador —por ejemplo, *spawnnear* una entidad, o pasar de deambular en una zona a perseguir al

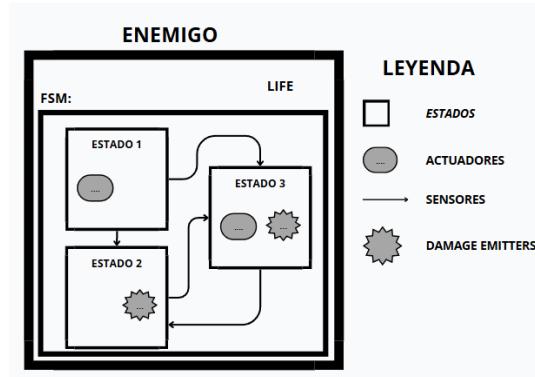


Figura 3.3: Ejemplo de estructura general de un enemigo basado en FSM.

jugador— su animación cambia para representar visualmente esa transición. Así, el jugador puede anticipar acciones, reconocer patrones de comportamiento y reaccionar en consecuencia.

Además, ciertos eventos clave como recibir daño o morir, activan animaciones específicas que refuerzan su impacto y aportan una mayor riqueza expresiva al sistema. Estas animaciones no son decorativas: forman parte esencial de la comunicación entre el enemigo y el jugador.

Este enfoque integrador entre comportamiento y apariencia permite diseñar enemigos no solo eficaces a nivel de juego, sino también consistentes desde el punto de vista estético y narrativo.

Se darán detalles de como funciona esta integración en la sección 4.7.

3.7. Ejemplos de uso

La división en actuadores, sensores, módulos de daño y estados permite construir enemigos complejos de forma modular, reutilizable y visualmente comprensible, sin necesidad de escribir código. A través de la combinación de estos elementos básicos, es posible definir distintos tipos de enemigos, cada uno con su propio comportamiento, nivel de amenaza e impacto sobre el jugador.

En esta sección se presentan ejemplos representativos de enemigos diseñados mediante la combinación de estos componentes. Estos ejemplos no son únicos, pero ilustran cómo el sistema permite definir comportamientos diversos de forma coherente y escalable.

Cabe destacar que estos enemigos están inspirados directamente en los patrones y casos identificados durante el análisis previo. En dicha fase se observaron distintas tipologías de enemigos en juegos de plataformas —desde amenazas constantes en movimiento, hasta enemigos estáticos con ataques periódicos o comportamientos

reactivos— y aquí se muestra cómo tales comportamientos pueden reproducirse y estructurarse utilizando el sistema propuesto. El objetivo es demostrar que el marco de diseño desarrollado no solo es teóricamente sólido, sino también aplicable de forma práctica a casos reales observados en juegos comerciales.

Aunque algunos términos aún no han sido desarrollados, todos serán introducidos más adelante en el capítulo de implementación.

Cada enemigo combina los siguientes elementos de diseño:

- Una lógica basada en una máquina de estados, que permite cambiar su comportamiento (si es necesario) en función del entorno del enemigo.
- Acciones específicas definidas por actuadores.
- Un sistema visual coherente mediante animaciones asociadas al estado del enemigo.
- Un sistema de daño que define cómo y cuándo el enemigo puede hacer daño o recibarlo.

3.7.1. Bouncing Bunny

El *Bouncing Bunny* (Figura 3.4) representa un ejemplo clásico de enemigo patrullero, inspirado directamente en enemigos como el Reptacillo (*Hollow Knight*) o el Walking Bot (*Bzzzt*), ambos caracterizados por un movimiento horizontal entre dos límites definidos del escenario.

Este enemigo recorre continuamente un tramo en línea recta, desplazándose de izquierda a derecha a velocidad constante. Al llegar a la pared invierte automáticamente su dirección, generando un patrón de patrullaje sencillo pero efectivo.

Su principal función dentro del diseño del nivel es la de introducir una amenaza predecible en zonas iniciales, obligando al jugador a aprender los tiempos de movimiento y a coordinar sus saltos para evitar el contacto. Esta previsibilidad lo convierte en un enemigo ideal para fases de aprendizaje, donde el jugador comienza a familiarizarse con las mecánicas básicas del juego.

A nivel de comportamiento, su peligro reside en el daño que infinge al contacto, aunque su resistencia es mínima, por lo que puede ser eliminado fácilmente. Esta dualidad contribuye a establecer un ritmo de juego dinámico sin suponer un obstáculo insalvable para el jugador principiante.

3.7.2. Spinning Rocks

El enemigo *Spinning Rocks* (Figura 3.5) representa un ejemplo claro del patrón de rotación fija identificado en el análisis previo, similar al comportamiento del Saw

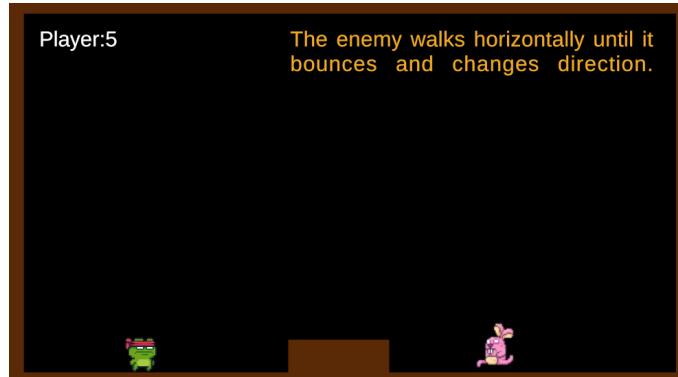


Figura 3.4: Escena de ejemplo donde observamos al enemigo en cuestión momentos antes de alcanzar la pared.

Drone en *Bzzzt*. Su función principal es la de constituir una amenaza constante basada en un movimiento predecible pero difícil de evitar, especialmente en espacios reducidos o plataformas móviles.

Cada unidad de este enemigo gira de forma continua alrededor de un punto central, describiendo una trayectoria circular completa. Aunque visualmente pueda interpretarse como una estructura compuesta, en realidad se trata de varias entidades independientes que orbitan de manera sincronizada, generando un entorno hostil a su alrededor.

La amenaza que presenta este tipo de enemigo no reside en su capacidad de persecución o ataque dirigido, sino en su mera presencia en el escenario. Al no poder ser destruido ni detenido, obliga al jugador a anticipar el patrón de movimiento y planificar cuidadosamente el momento para cruzar o interactuar con el entorno. Esta característica lo convierte en un excelente recurso para secciones de plataformas que requieren precisión y *timing*.

A nivel de diseño, su uso está orientado a crear zonas de paso peligrosas que introducen tensión sostenida. Su predecibilidad lo hace justo, pero su invulnerabilidad lo vuelve implacable, lo que refuerza su valor como obstáculo en segmentos avanzados o de alto riesgo dentro del nivel.

3.7.3. Trunk Turret

La *Trunk Turret* (Figura 3.6) es un claro ejemplo del patrón de comportamiento identificado como *torreta* durante el análisis de enemigos, tal como se observa en la *Shooting Trap* de *Blasphemous* o la *Cannon Turret* de *Bzzzt*. Se trata de un enemigo completamente estático cuya principal amenaza reside en la emisión periódica de proyectiles, los cuales obligan al jugador a mantenerse en constante movimiento y adoptar un enfoque reactivo basado en la evasión.

A diferencia de otros enemigos móviles o interactivos, la *Trunk Turret* no inflige

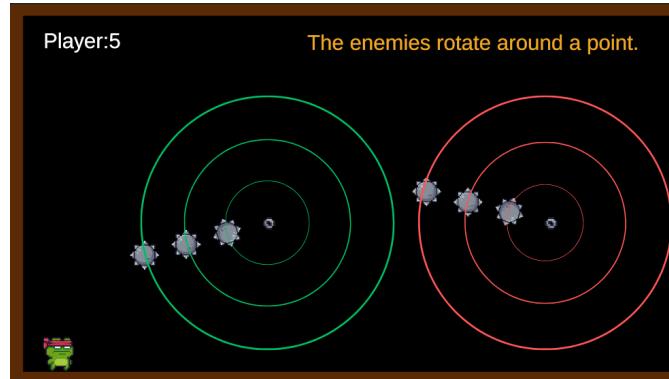


Figura 3.5: Escena de ejemplo donde se aprecian dos enemigos realizando un movimiento circular, cada uno en un sentido diferente.

daño directamente ni puede ser destruida. Su condición de invulnerabilidad la convierte en un elemento de peligro permanente en el escenario, generando zonas de paso peligrosas donde el jugador debe coordinar sus desplazamientos con el ritmo de disparo para evitar ser alcanzado.

Este tipo de enemigo funciona particularmente bien en zonas de transición o plataformas estrechas, donde el margen de error es reducido. Además, su previsibilidad rítmica aporta una capa de estrategia, ya que el jugador puede anticipar el disparo y actuar en consecuencia, generando una tensión constante sin recurrir a un comportamiento impredecible.

3.7.3.1. Bullet

El proyectil emitido por la *Trunk Turret* se comporta como una amenaza secundaria pero letal. Se desplaza horizontalmente a velocidad constante y se destruye al colisionar con cualquier superficie o con el jugador, a quien infinge daño inmediato. Esta dinámica complementa el carácter del enemigo principal, reforzando su papel como generador de obstáculos periódicos que condicionan el movimiento del jugador.

La inclusión del proyectil permite ampliar la funcionalidad del enemigo sin necesidad de dotarlo de movilidad o interacción directa, lo que demuestra la versatilidad del sistema basado en componentes. Tanto la torre como la bala funcionan de forma independiente pero coordinada, lo que permite su reutilización en otros contextos o combinaciones.

3.7.4. Spline Chicken

La *Spline Chicken* (Figura 3.7) ejemplifica una variación del patrullaje, previamente identificado. Sin embargo, a diferencia de aquellos enemigos que se desplazan en línea recta, esta gallina se mueve siguiendo una trayectoria definida,

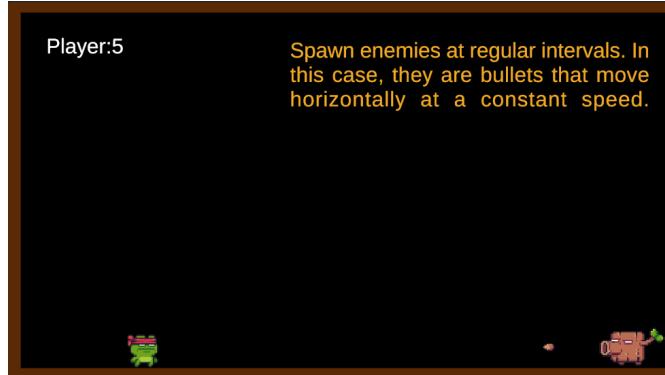


Figura 3.6: Escena de ejemplo donde se aprecia una *Trunk Turret* disparando.

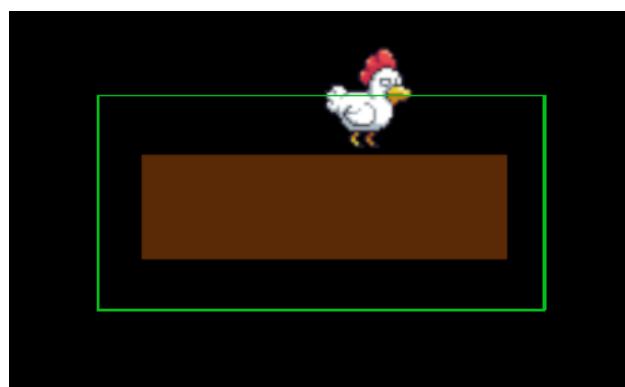


Figura 3.7: Escena de ejemplo donde se aprecia una *Spline Chicken* siguiendo su trayectoria.

generando un movimiento continuo y cíclico alrededor de una plataforma concreta.

Este comportamiento introduce una dinámica distinta en el diseño del nivel. La trayectoria predefinida obliga al jugador a observar y anticipar los patrones de movimiento con mayor precisión, ya que las curvas pueden generar ángulos muertos o zonas de paso más breves. Aunque su diseño visual puede transmitir un tono humorístico, la *Spline Chicken* representa una amenaza persistente: no puede ser eliminada ni detenida, y provoca daño inmediato al contacto.

Su presencia funciona como elemento de presión en zonas de transición o salto, donde el jugador debe sincronizar su avance con el ritmo del movimiento del enemigo. Al igual que otros enemigos invulnerables, cumple una función más cercana a la de una trampa móvil, generando tensión constante sin requerir interacción directa con el combate.

Este diseño demuestra cómo el marco de componentes propuesto permite extender comportamientos clásicos (como el patrullaje) a variantes más sofisticadas y visualmente interesantes, sin necesidad de redefinir estructuras complejas.

3.7.5. Skywatch Eagle

La *Skywatch Eagle* (Figura 3.8) es un ejemplo de enemigo con comportamiento reactivo, construido a partir de una máquina de estados. Su diseño combina movilidad aérea libre y un cambio abrupto hacia una acción ofensiva directa, lo que la convierte en una amenaza dinámica que exige atención constante por parte del jugador.

Este enemigo patrulla el cielo en un área determinada, recorriendo distintos puntos de forma aleatoria. Al detectar al jugador en su campo de visión, interrumpe su patrón y se lanza en picado hacia él. Este cambio de comportamiento genera un contraste entre una amenaza inicialmente pasiva y una reacción agresiva, lo que introduce tensión en el ritmo de juego.

Desde el punto de vista del diseño, la *Skywatch Eagle* responde a dos intenciones clave: por un lado, introducir un enemigo volador que no se limita a un movimiento cíclico, sino que reacciona activamente a la presencia del jugador; y por otro, obligar al jugador a planificar su avance con más cuidado, ya que un movimiento en falso puede activar un ataque difícil de esquivar.

El comportamiento de este enemigo está compuesto por dos estados diferenciados:

- **Estado 1 – Patrullaje aéreo:** En este estado, el águila se desplaza de forma libre dentro de un área específica del escenario. La trayectoria entre puntos es fluida, y su movimiento tiene una función más disuasoria que ofensiva. Durante esta fase, observa el entorno en busca del jugador.
 - *Acción:* Movimiento entre posiciones aleatorias en el espacio aéreo.
 - *Condición de transición:* Si el jugador se acerca lo suficiente, la enemiga cambia al estado de ataque.
- **Estado 2 – Picado de ataque:** Al entrar en este estado, la *Skywatch Eagle* modifica radicalmente su comportamiento, lanzándose en picado hacia el jugador a gran velocidad. Durante este movimiento, su capacidad de infligir daño se activa. Tras el impacto, regresa al estado de patrullaje.
 - *Acción:* Desplazamiento rápido hacia la última posición detectada del jugador.
 - *Condición de transición:* Al producirse el impacto con el jugador, retorna al estado de patrullaje.

Este comportamiento se inspira directamente en patrones detectados en enemigos como la Vengamosca de *Hollow Knight*, que combina movilidad aérea con un cambio de comportamiento agresivo al detectar al jugador. Esta dinámica se refleja también en la *Skywatch Eagle*, cuya transición desde un patrullaje pasivo a un ataque en picado introduce una amenaza repentina y exige una respuesta rápida por parte

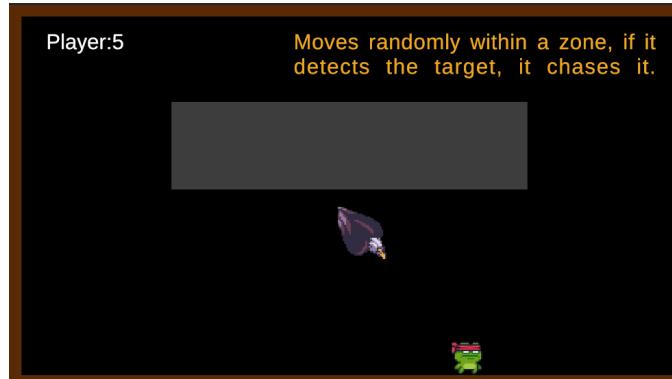


Figura 3.8: Escena de ejemplo donde se aprecia a la *Skywatch Eagle* atacando al jugador.

del jugador. La inclusión de un estado de patrullaje previo refuerza el impacto del ataque y permite anticiparse, fomentando una experiencia de juego más estratégica.

3.8. Conclusiones del Diseño

En este capítulo, tras el análisis de enemigos en juegos de referencia como *Hollow Knight*, *Blasphemous* y *Bzzzt*, se ha definido la arquitectura conceptual del framework para la creación de enemigos en 2D. Dicha definición se organiza en cuatro bloques fundamentales:

- **Sensores**, responsables de detectar condiciones del entorno (presencia del jugador, colisiones, distancia, daño recibido).
- **Actuadores**, encargados de ejecutar acciones (movimientos lineales, circulares o spline, generación de unidades).
- **Módulo de daño**, que distingue entre emisión y recepción de daño, regulando cómo se infinge o recibe daño.
- **Máquina de Estados Finita**, que organiza los sensores y actuadores en estados coherentes y controla las transiciones entre ellos.

Asimismo, se ha incluido el mecanismo de sincronización con animaciones, que asocia cada estado de la FSM con la representación visual correspondiente, garantizando coherencia entre la lógica de IA y la experiencia del jugador.

Los ejemplos de diseño han validado la capacidad del framework para reproducir y combinar los patrones detectados en juegos comerciales, demostrando:

- Modularidad en la composición de comportamientos.
- Escalabilidad en la complejidad de los enemigos.
- Claridad visual y diseño accesible para no programadores.

Con esta base conceptual completada, en el siguiente capítulo se desarrollará la implementación práctica de todos estos componentes en Unity para materializar el diseño teórico en una herramienta funcional.

Capítulo 4

Implementación

En el Capítulo 3 se abordó la descripción de la herramienta sin entrar en detalles de como estaba implementada, 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.

La implementación de la herramienta puede encontrarse en el siguiente enlace: <https://github.com/CristinaMora/EnemyBehaviourFramework-2D.git>.

4.1. Tecnología utilizada

En este capítulo se describe el desarrollo de una herramienta diseñada para facilitar la creación y configuración de enemigos en un entorno 2D dentro del motor Unity. El objetivo principal es que pueda ser utilizada por diseñadores sin necesidad de escribir ni una sola línea de código, haciendo uso exclusivamente del editor y de una interfaz visual basada en prefabs y componentes reutilizables. La herramienta se presenta como un plugin modular, intuitivo y fácilmente integrable en distintos proyectos.

Para su desarrollo se ha utilizado el motor Unity, introducido en el Capítulo 2, concretamente en su versión 2022.3.18f1 (LTS). No se garantiza compatibilidad con versiones anteriores, aunque se espera que funcione correctamente en versiones posteriores, salvo cambios relevantes en la API del motor.

Unity ha sido elegido frente a otros motores de videojuegos por su accesibilidad, su curva de aprendizaje relativamente baja y su amplia comunidad. Estas características lo convierten en una opción ideal para que cualquier usuario, sin profundos conocimientos técnicos, pueda sacar provecho de sus funcionalidades.

A lo largo del desarrollo se emplean diversos recursos del propio motor, como

el sistema de físicas 2D o herramientas de depuración como Gizmos. Además, la arquitectura basada en componentes de Unity favorece la modularidad del sistema, permitiendo dividir el comportamiento de los enemigos en partes independientes y reutilizables.

A continuación se presentará la arquitectura escogida y los principales módulos de implementación.

4.2. Arquitectura del sistema

La arquitectura de la herramienta está organizada en módulos funcionales que encapsulan distintas responsabilidades dentro del sistema. Esta organización facilita la extensibilidad, el mantenimiento y la reutilización de los distintos elementos.

Los principales módulos de implementación incluidos en esta arquitectura son:

- **Módulo de actuadores:** Ejecutan acciones concretas, como moverse, disparar o generar nuevos enemigos.
- **Módulo de sensores:** Detectan eventos, como la presencia del jugador o colisiones.
- **Módulo de daño:** Permite que los enemigos reciban y hagan daño.
- **Módulo de Máquina de Estados:** Define el comportamiento que un enemigo adopta en un momento determinado.
- **Módulo de animaciones:** Coordina las animaciones asociadas a el estado actual del enemigo.
- **Módulo del jugador:** Módulo que sirve para contextualizar y ver las respuesta de los enemigos a las acciones del jugador.

Además, se incluyen una serie de *prefabs* preconfigurados, diseñados para acelerar la creación de enemigos con comportamientos comunes. En Unity, un *prefab* es una plantilla reutilizable que agrupa un conjunto de componentes, configuraciones y objetos en una única entidad. Esto permite instanciar múltiples copias consistentes de un mismo objeto manteniendo su configuración original. Gracias a esta funcionalidad, los *prefabs* permiten a los diseñadores iterar rápidamente sin necesidad de modificar el código, fomentando así la reutilización y facilitando el prototipado.

En resumen, se ha creado una herramienta extensible, basada en módulos bien definidos y en la arquitectura por componentes de Unity. Permite crear comportamientos complejos de forma visual, lo que facilita su uso por diseñadores sin conocimientos técnicos avanzados.

En las siguientes secciones se describirá en detalle cada uno de los módulos que componen esta arquitectura.

4.3. Módulo de actuadores

El módulo de actuadores es el encargado de englobar todo el comportamiento de los actuadores en la herramienta. Para una explicación detallada del diseño de este módulo, puede consultarse la Sección 3.3, donde se aborda el sistema de actuadores en su conjunto.

4.3.1. Actuator

La clase abstracta `Actuator` define la interfaz común para todos los actuadores del módulo. Hereda de `MonoBehaviour`, permitiendo así su uso como componente en objetos de Unity. Su principal propósito es establecer una estructura homogénea que garantice que todos los actuadores puedan ser gestionados, activados, actualizados y destruidos de forma coherente y centralizada.

`Actuator` declara tres métodos abstractos que deben ser implementados por todas las clases derivadas:

- **StartActuator()**: Método encargado de inicializar el actuador. Se invoca al comienzo de su ciclo de vida.
- **UpdateActuator()**: Llamado una vez por frame, permite actualizar el comportamiento del actuador mientras esté activo.
- **DestroyActuator()**: Método encargado de finalizar y limpiar los recursos asociados al actuador cuando ya no es necesario.

Además, la clase contiene una variable protegida que se puede activar o desactivar mediante el método público `SetDebug(bool debug)`. Gracias a este diseño, se consigue una arquitectura flexible, extensible y desacoplada que facilita la incorporación de nuevos actuadores en el sistema con un esfuerzo mínimo, manteniendo siempre una interfaz común y predecible.

4.3.2. MovementActuator

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

Parámetros de configuración

- (bool) **Is Accelerated**: 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 cómo varía el movimiento en el tiempo. Este parámetro solo se requiere si el movimiento es acelerado.

4.3.2.1. 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.

Parámetros de configuración

- (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).
- (LayerMask) **Layers To Collide**: En caso de querer algún tipo de reacción, especificar qué 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*.
- (float) **Speed**: Velocidad del movimiento.
- (float) **Goal Speed**: Velocidad final del movimiento si éste 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, se lanzará inicialmente con una velocidad inicial y, en caso contrario, se aplicará constantemente una fuerza.
- (bool) **Follow Player**: Determina si la dirección del movimiento se ajusta automáticamente para acercarse al jugador.

4.3.2.2. VerticalActuator

`Vertical Actuator` permite mover un objeto verticalmente, ya sea arriba o abajo, con diferentes configuraciones de velocidad y comportamientos tras una colisión.

Parámetros de configuración

- (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).
- (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*.
- (float) **Speed**: Velocidad del movimiento.
- (float) **Goal Speed**: Velocidad final del movimiento si éste 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, se lanzará inicialmente con una velocidad inicial y, en caso contrario, se aplicará constantemente una fuerza.
- (bool) **Follow Player**: Determina si la dirección del movimiento se ajusta automáticamente para acercarse al jugador.

4.3.2.3. DirectionalActuator

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

Parámetros 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, siendo 0^0 un movimiento hacia la derecha. Los grados se suman en sentido antihorario.
- (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**: Determina si el objeto calculará automáticamente el ángulo inicial para moverse en dirección al jugador.

4.3.2.4. CircularActuator

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

Parámetros 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**: Determina si 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**: Determina si el objeto se orienta automáticamente hacia el jugador.

4.3.2.5. SplineFollowerActuator

La clase `SplineFollowerActuator` se usa para describir un movimiento mediante curvas Splines de Unity. El Actuador sigue la curva pudiendo girar el objeto a su vez.

Parámetros 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**: Objeto que define una trayectoria basada en una spline, como la descrita en la Subsección 3.3.1. Este componente permite establecer una serie de puntos de control conectados por curvas suaves, editables directamente en el editor de Unity. Además, el `SplineContainer` ofrece funciones para acceder a la curva en tiempo real, interpolar posiciones a lo largo del recorrido y orientar el objeto dinámicamente durante su desplazamiento.
- (enum) **Teleport To Closest Point**: Define cómo se ajusta el objeto a la spline al iniciarse el movimiento. Puede tener dos valores:
 - **Move Enemy To Spline**: El enemigo se teletransporta a la spline.
 - **Move Spline To Enemy**: La spline se desplaza para alinearse con la posición actual del objeto.

4.3.2.6. MoveToAPointActuator

La clase `Move To A Point 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 (descripción completa en Sección 3.3.1). Dado que la clase tiene dos maneras de funcionar y dos configuraciones distintas, primero se abordarán los parámetros de configuración comunes entre ambas y luego los parámetros específicos de cada configuración.

Parámetros de configuración comunes

- (enum) **Mode**: Define si se va a seguir una ruta por puntos (*Waypoint*) o si se escogen puntos aleatoriamente dentro de una zona (*RandomArea*).

Parámetros de configuración para Waypoint

- (bool) **Loop**: Determina si al finalizar los puntos volverá al primero y el movimiento se repetirá.
- (bool) **All Waypoints Have The Same Data**: Determina si todos los puntos usarán la misma configuración.
- (List<`WaypointData`>) **Waypoints Data**: Lista de puntos que el objeto debe seguir. En este caso el struct `WaypointData` está compuesto por:
 - (Transform) **Waypoint Transform**: Posición a la que se debe de llegar.
 - (float) **Time To Reach**: Tiempo estimado para alcanzar la posición deseada.
 - (bool) **Is Accelerated**: Si se requiere aceleración.
 - (EasingFunction.Ease) **Easing Function**: Función de aceleración que describe como varía la posición del objeto con respecto al tiempo, en caso de tratarse de un movimiento acelerado.
 - (bool) **Should Stop**: Indica si debe detenerse al llegar.
 - (float) **Stop Duration**: Tiempo que debe durar la parada en caso de existir.

Parámetros de configuración para RandomArea

- (Collider2D) **Random Area**: Zona dentro de la cual se moverá el objeto si está configurado como *RandomArea*.
- (float) **Time Between Random Points**: Tiempo necesario, en segundos, para ir de un punto aleatorio a otro.

4.3.2.7. MoveToAnObjectActuator

La clase `Move To An Object 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.

Parámetros de configuración

- (`Transform`) **Object Transform**: Posición del objeto destino. No se necesita la referencia al objeto en sí, ya que lo único que se va a utilizar de éste es su posición.
- (`float`) **Time To Reach**: Tiempo necesario, en segundos, para llegar a la posición destino.

4.3.3. SpawnerActuator

La clase `SpawnerActuator` se usa para poder generar nuevos enemigos, pudiendo generar infinitos enemigos o un número definido de ellos cada cierto tiempo en un lugar predefinido.

Parámetros de configuración

- (`float`) **Spawn Interval**: Intervalo de tiempo en segundos que tarda el objeto en volver a generar.
- (`bool`) **Infinite Enemies**: Indica si se generarán enemigos indefinidamente. Si es verdadero, se crearán continuamente nuevos enemigos cada cierto tiempo indicado por *Spawn Interval*. 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 Points**: Lista de elementos de tipo `Spawn Info`, clase compuesta por:
 - (`GameObject`) **Prefab To Spawn**: Objeto a instanciar.
 - (`Transform`) **Spawn Point**: Punto de aparición del objeto.

Esta lista está diseñada para poder crear distintos tipos de enemigos o en varios sitios a la vez, dando así más flexibilidad.

4.4. Módulo de sensores

El sistema de sensores constituye un módulo independiente encargado de detectar condiciones específicas del entorno durante la ejecución del juego. Este módulo proporciona una interfaz común para diferentes tipos de sensores, permitiendo su

integración flexible en otros componentes de la herramienta, como las transiciones de estado o los comportamientos de los enemigos.

Desde el punto de vista estructural, el módulo está diseñado siguiendo una arquitectura basada en el patrón observador (observer), lo que permite a otros componentes suscribirse a los eventos generados por los sensores sin necesidad de establecer dependencias directas.

Para más detalles del diseño del módulo visitar la Sección 3.4.

4.4.1. Sensor

La clase **Sensor** actúa como clase base para todos los sensores de la herramienta. Implementa una arquitectura basada en el patrón observador (observer), en la que cualquier componente interesado puede registrarse para ser notificado cuando el sensor se active.

Cada sensor contiene una variable de tipo **Action<Sensor>** que alberga una lista de funciones (callbacks) a ejecutar cuando se detecta una condición específica. Estas funciones pueden ser proporcionadas por otros componentes, como por ejemplo una transición que deba activarse tras la detección.

Para garantizar una interfaz unificada, las funciones que se registren como oyentes deben aceptar un único parámetro de tipo **Sensor**. Esto permite que, al activar el sensor, se le pueda pasar una referencia a sí mismo, facilitando que el componente suscrito acceda a su información contextual si lo necesita.

Este enfoque permite una arquitectura desacoplada y extensible, donde los sensores no necesitan conocer directamente qué componentes responderán a su activación, simplemente notifican a todos los suscriptores registrados.

Además, la implementación del sistema de eventos incluye un contador de suscriptores, que permite llevar un control sobre cuántos componentes están actualmente registrados. Esta medida ayuda a evitar errores en tiempo de ejecución relacionados con notificaciones no deseadas o fugas de memoria.

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 gestionan el estado interno del sensor. Una de las variables indica si el sensor se encuentra activo o inactivo, mientras que otra se utiliza para facilitar la depuración, mostrando información útil sobre su estado actual.

Parámetros de configuración

- (float) **Start Detecting Time**: Tiempo que necesitará el sensor para encenderse al entrar en el estado que lo alberga. Si el sensor no está encendido se considera que está apagado y su funcionalidad quedará suspendida hasta que esté encendido.

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

4.4.2. AreaSensor

AreaSensor representa un tipo de sensor espacial que detecta la presencia de un objeto objetivo dentro de una zona delimitada (Figura 4.1). Está pensado para funcionar con zonas de activación, por lo que requiere que el objeto que lo contiene tenga un componente **Collider2D**.

La detección se realiza mediante los métodos *OnTriggerEnter/Stay/Exit2D* 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 encenderse el sensor. Esto evita perder eventos relevantes si la detección no estaba habilitada previamente.

Parámetros de configuración

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

4.4.3. CollisionSensor

CollisionSensor es el sensor encargado de detectar colisiones entre el objeto que lo contiene y un conjunto específico de objetos definidos mediante una **LayerMask** de *Unity*. Al igual que el **AreaSensor**, este sensor requiere obligatoriamente la presencia de un componente **Collider2D** en el objeto. Para garantizarlo, se utiliza la anotación correspondiente en *Unity* que fuerza su inclusión automáticamente.

La detección de colisiones se realiza mediante los métodos *OnCollisionEnter2D* y *OnCollisionStay2D*. Este último es necesario para cubrir casos en los que la colisión ya se está produciendo cuando el sensor se enciende: si se utilizara únicamente *OnCollisionEnter2D*, dichas situaciones no serían detectadas. Para que la colisión sea válida, es importante que ninguno de los dos objetos involucrados debe ser trigger.

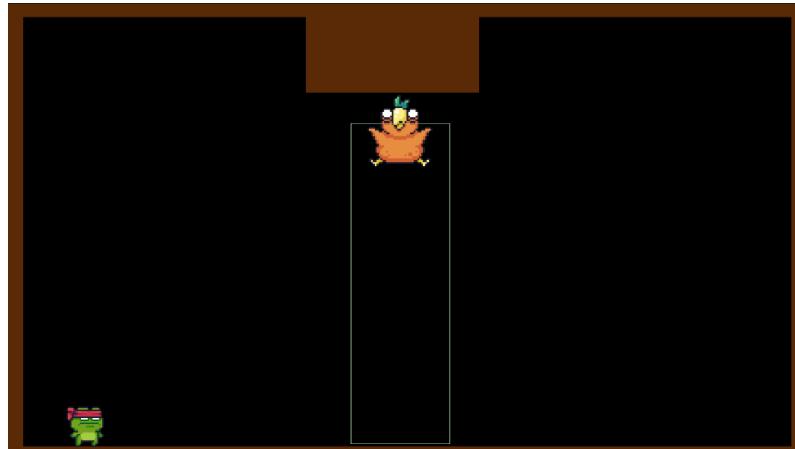


Figura 4.1: AreaSensor utilizado para enemigo que cae al detectar al jugador.

Parámetros de configuración

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

4.4.4. DistanceSensor

DistanceSensor es el sensor utilizado para medir la distancia entre dos puntos. La distancia se medirá tomando de referencia las posiciones de sendos objetos, uno de ellos el que posee este componente 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*.

Parámetros 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*, sus usos y las variables específicas necesarias en cada caso.

- *Magnitude*



Figura 4.2: Medición de distancia a través de la magnitud.

Configuración utilizada cuando se quiere medir la distancia como magnitud, cuya representación gráfica es un círculo, como se muestra en la Figura 4.2. Dependiendo del valor de la variable *Detection Condition*, el sensor se activará si el objeto objetivo está dentro o fuera del círculo.

Parámetros de configuración

- (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.

En la Figura 4.3 vemos un ejemplo de como luce la medición en el eje X.

Parámetros 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, en el lado positivo o en el lado negativo.

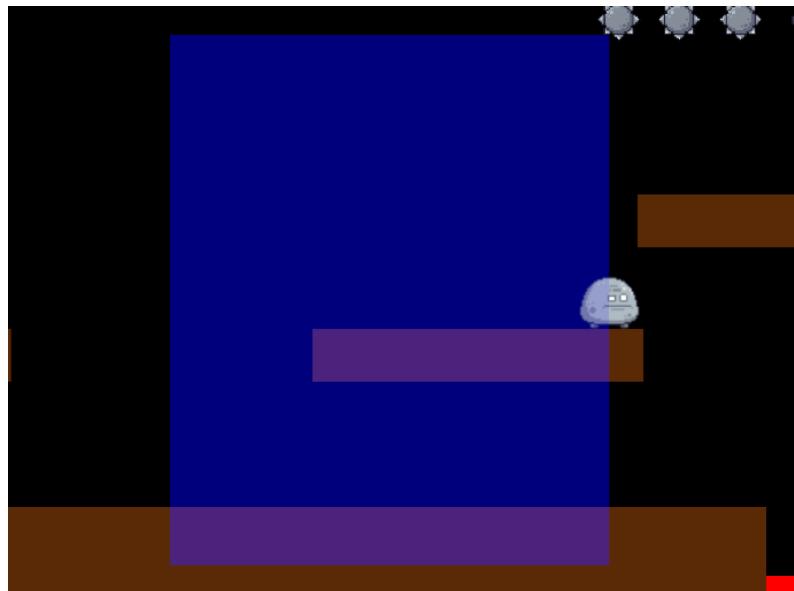


Figura 4.3: Medición de distancia en el eje X en su lado negativo.

4.4.5. TimeSensor

Sensor encargado de activarse cuando pasa un tiempo determinado desde su activación. En caso de ajustar que el sensor necesitará un tiempo para encenderse, primero se procederá a medir tal tiempo y, cuando este llegue a su fin, se medirá el tiempo de activación propio del sensor.

Parámetros de configuración

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

4.5. Módulo de daño

El módulo de daño es el encargado de gestionar todas las interacciones relacionadas con la aplicación y recepción de daño entre entidades del juego. Su diseño (Sección 3.5) sigue una arquitectura basada en componentes desacoplados, que permite separar claramente los elementos que emiten daño de aquellos que lo reciben.

Este módulo se compone principalmente de dos clases clave: `DamageEmitter` y `DamageSensor`.

4.5.1. DamageEmitter

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

Este componente no implementa funcionalidad adicional en su propio componente, sino que actúa como una señal para el `DamageSensor`, que será explicado en el

siguiente apartado. 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:

Parámetros 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.

- *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.

Parámetros 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 cierto tiempo. Este daño se producirá mientras dure la colisión o, en caso de que sea un trigger, la superposición.

Parámetros 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.

Parámetros 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.

4.5.2. 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`.

La funcionalidad de este sensor es doble. Además de activar transiciones, se utiliza para gestionar la lógica del componente `Life` (explicado en la Sección 4.8.4). 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`.

Dado que `Life` es un componente utilizado para contextualizar la herramienta, pero no necesariamente el único componente de vida posible, `DamageSensor` puede utilizarse para crear nuevos componentes de vida con distintas características, separando así la funcionalidad de ambos componentes.

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.

Parámetros 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.

4.6. Módulo de Máquina de Estados

El módulo de Máquina de Estados es el encargado de gestionar el comportamiento dinámico de una entidad. Está basado en una Máquina de Estados Finita (FSM, por sus siglas en inglés), en la que el comportamiento se estructura mediante un conjunto de **States** y transiciones (**Transitions**) entre ellos. Este módulo permite definir de forma clara y modular los distintos estados que una entidad puede tener, así como las condiciones necesarias para cambiar de uno a otro.

Para detalles del diseño visitar la Sección 3.6.

Para una mejor organización y escalabilidad, este módulo se compone de las siguientes clases: **FSM**, **State** y **Transition**.

4.6.1. FSM

La clase **FSM** representa la máquina de estados finita propiamente dicha. Se encarga de mantener el estado actual de la entidad, actualizarlo, y gestionar los posibles cambios de estado de manera segura. La comprobación de transiciones se realiza en el método **LateUpdate**, una vez finalizadas todas las actualizaciones del estado, para evitar cambios prematuros durante el ciclo de actualización.

Parámetros de configuración

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

4.6.2. 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.

Parámetros de configuración

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

4.6.3. State

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

Parámetros de configuración

- (**List<Actuator>**) **Actuator List**: Lista de **Actuators** que son actualizados en cada bucle y representan acciones.

- (`List<Transition>`) **Transition List**: Lista de Transitions que pueden ser activadas.
- (`List<DamageEmitter>`) **Damage Emitter in State**: Lista de DamageEmitters activos en el estado
- (`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.

4.7. Módulo de Animación

El módulo de animación se encarga de coordinar la representación visual del comportamiento de los enemigos, sincronizando sus animaciones con su lógica interna. Este módulo permite adaptar dinámicamente las animaciones en función de factores como la velocidad de movimiento, la dirección, o eventos del juego como recibir daño o morir.

El sistema se basa en el uso del componente **Animator** de Unity, que a su vez está asociado a un **Animator Controller**. Este controlador define una Máquina de Estados Finita (FSM) visual, donde cada estado representa una animación concreta (por ejemplo: caminar, aparecer, atacar, morir) y las transiciones entre estos estados se activan mediante parámetros. Estos parámetros pueden ser booleanos, numéricos, o de tipo *trigger*, y son gestionados por el módulo en tiempo de ejecución.

Este módulo proporciona una interfaz común para vincular las animaciones con otros módulos, como los actuadores de movimiento o el módulo de daño, de forma desacoplada y escalable.

Para más detalle del diseño de la lógica de animaciones consultar Sección 3.6.2.

4.7.1. AnimatorManager

La clase **AnimatorManager** encapsula toda la lógica necesaria para actualizar los parámetros del **Animator** de forma dinámica. Su objetivo es reflejar correctamente el comportamiento del enemigo en la animación visual que se muestra al jugador.

Parámetros de configuración

- (`bool`) **Can Flip X**: Indica si el sprite puede voltearse horizontalmente.
- (`bool`) **Can Flip Y**: Indica si el sprite puede voltearse verticalmente.
- (`SpriteRenderer`) **Sprite Renderer**: Componente de Unity encargado de representar visualmente un sprite en pantalla. Es el responsable de renderizar la imagen 2D asociada al objeto en la escena.

Unity utiliza un sistema de animaciones basado en un componente llamado **Animator**, que se asocia a un **Animator Controller**. Este controlador define una Máquina de Estados Finita (FSM) de animaciones, donde cada estado representa una animación concreta (como caminar, correr o morir) y las transiciones entre estados se controlan mediante parámetros. Estos parámetros pueden ser de distintos tipos, como booleanos, flotantes, enteros o *triggers* (disparadores), y se actualizan dinámicamente desde el código para reflejar el comportamiento del objeto en tiempo real.

4.7.1.1. Parámetros requeridos en el Animator

Para centralizar la gestión de las animaciones, se ha creado un controlador personalizado (Figura 4.4), del tipo **Animator Controller**. Este controlador permite una integración fluida entre la lógica del comportamiento del enemigo y su representación visual, facilitando el mantenimiento y la ampliación del sistema.

El **Animator Controller** funciona como una máquina de estados de animación, donde cada estado representa una acción visual (como caminar, aparecer, recibir daño o morir), y las transiciones entre estos estados están condicionadas por parámetros que se actualizan dinámicamente desde el código.

Durante la ejecución del juego, se monitoriza constantemente la velocidad del objeto mediante su componente **Rigidbody2D**. Esta información se utiliza para actualizar tres parámetros del **Animator**: *XSpeed*, *YSpeed* y *RotationSpeed*. Estos valores permiten adaptar en tiempo real la animación del enemigo: por ejemplo, mostrar una animación de correr hacia la derecha cuando la velocidad en el eje X es positiva. Además, si el enemigo cambia de dirección (por ejemplo, de izquierda a derecha), su sprite puede rotarse automáticamente para mantener la coherencia visual, siempre que esta opción haya sido habilitada mediante parámetros booleanos.

Por otro lado, el sistema de animaciones responde a eventos clave del juego mediante parámetros adicionales que se activan desde otros módulos del sistema:

- El parámetro *Follow* (bool) indica si el enemigo debe ajustar su animación al seguir a otro objeto, generalmente el jugador.
- La función *ChangeState* modifica el estado de la FSM (máquina de estados finita), provocando potenciales cambios de animación.
- El trigger *Spawn* se activa al instanciar el enemigo, permitiendo reproducir una animación de aparición o ejecutar lógica específica.
- Los triggers *Damage* y *Die* se activan al recibir daño o morir, lanzando las animaciones correspondientes. En el caso de muerte, el objeto se destruye al finalizar la animación.

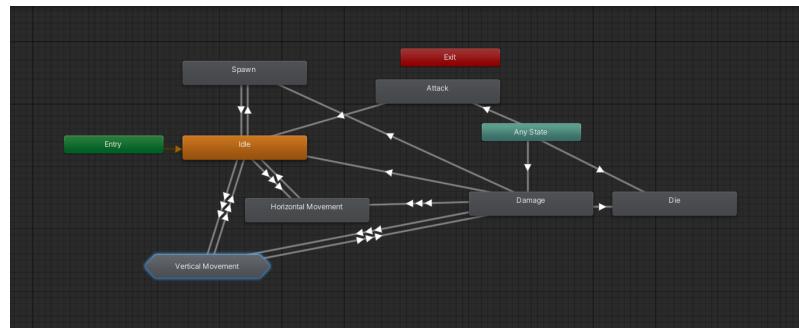


Figura 4.4: Animator Controller con sus estados y transiciones.

En resumen, los parámetros utilizados por el `Animator Controller` se agrupan en tres tipos principales:

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

Gracias a esta estructura, se consigue una sincronización precisa entre el comportamiento interno del enemigo y su aspecto visual, ofreciendo una experiencia más inmersiva y coherente para el jugador.

4.8. Módulo del Jugador

El **Módulo del Jugador** encapsula toda la lógica de movimiento, interacción y control del personaje principal. Este módulo proporciona los sistemas fundamentales para el desplazamiento, salto, detección de colisiones, salud y ataques del jugador. Su implementación busca replicar una experiencia precisa y responsiva, inspirada en estándares de juegos de plataformas modernos.

Para lograr este comportamiento se ha tomado como referencia la implementación del canal de YouTube *Mix and Jam* en su análisis del videojuego *Celeste*¹, adaptando sus principios para integrarlos dentro de un sistema modular y reutilizable.

Este módulo está compuesto por varios componentes especializados que, combinados, ofrecen una experiencia de control intuitiva y sólida. En las siguientes subsecciones se detallan estos componentes, sus responsabilidades y parámetros de configuración.

4.8.1. 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.

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

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

- (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.

4.8.2. PlayerCollisionDetection

Este componente se encarga de detectar las colisiones del jugador. Para ello, se ajustan tres cajas de colisión (Figura 4.5): una en cada lado y otra para detectar el contacto con el suelo. Las cajas no detectan realmente colisiones, sino que comprobarán 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.

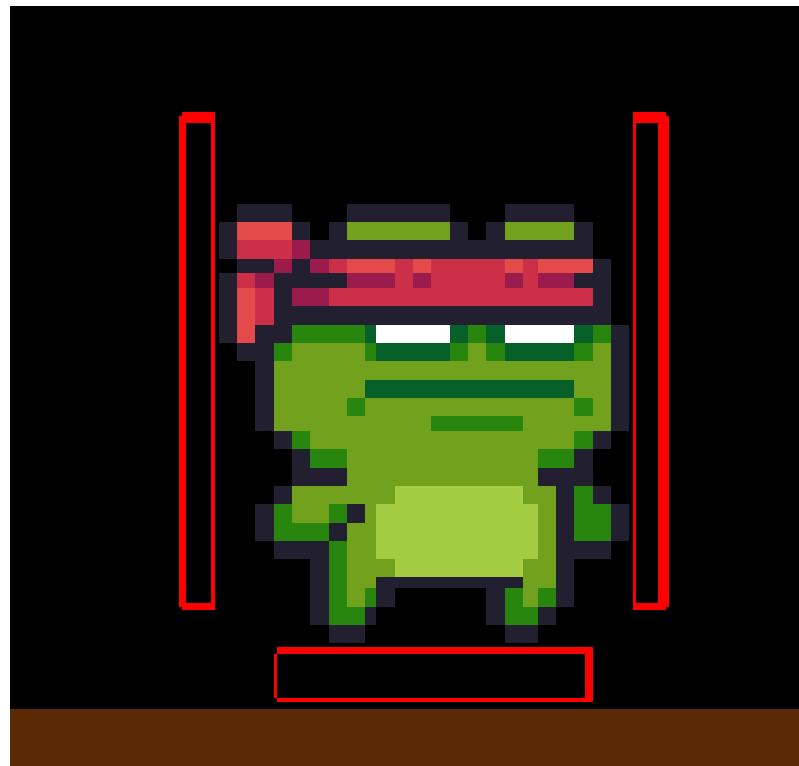


Figura 4.5: Representación de las cajas de detección de colisiones del jugador

4.8.3. PlayerJump

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 componente 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.

Parámetros 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.

4.8.4. 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` 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.

Parámetros 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.

4.8.5. 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, por lo que se proporciona con la herramienta un prefab llamado `PlayerBullet` que ya cumple con esta característica), 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.

Parámetros 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.

4.9. Distribución de la herramienta

El Framework será distribuido a través de un archivo `.unitypackage` descargable desde el repositorio de la herramienta.

Una vez descargado, el usuario podrá acceder a todo el contenido distribuido, que se detalla a continuación:

- **Animations**: Carpeta donde se encuentran tanto los *sprites* usados como los `AnimatorController` creados para los enemigos de ejemplo.
- **Materials**: En este directorio se encuentran materiales usados en la herramienta y el `Physic Material 2D` usado en algunos enemigos para mejorar su interacción con el jugador.
- **Prefabs**: En la carpeta de *Prefabs* se encuentran todos los *prefabs* utilizados en la herramienta. También se encuentra el prefab *BaseEnemy* el cual es usado en el manual que será mencionado luego.
- **Scenes**: Al abrir la carpeta el usuario verá otra llamada *EnemySamples* donde se encuentran todas las escenas con enemigos de ejemplo. También encontrará *BaseScene* que se trata de la plantilla de escena usada en los ejemplos. Por último encontramos la escena *ExampleLevel* la cual es un nivel de prueba en el que se encuentran varios enemigos en el mismo nivel.
- **Scripts**: Por último encontramos la carpeta *Scripts* en la cual encontramos:
 - **Actuators**: Directorio en el cual se pueden encontrar todos los componentes relacionados con los `Actuators`.
 - **Animations**: Aquí se encuentra el componente `AnimatorManager`.
 - **Editor**: En esta carpeta están todos los componentes relacionados con los editores, todos ellos heredan de la clase de *Unity* llamada `Editor`.
 - **FSM**: Carpeta en la que se encuentran los componentes `State` y `FSM`.
 - **Player Behaviour**: Lugar donde se encuentran todos los componentes que tienen que ver con el funcionamiento del jugador.
 - **SensorsAndEmitter**: Carpeta en la que están todos los componentes que heredan de `Sensor` (incluyéndolo) y `DamageEmitter`.
 - **Otros componentes**: También se pueden encontrar los componentes: `EasingFunctions`, `Life` y `Timer`.

Junto con el Framework, se incluye un manual de uso: que contiene imágenes adicionales y está orientado a diseñadores de videojuegos. También se proporcionan una serie de tutoriales para reproducir algunos de los enemigos incluidos en el Framework.

4.10. Conclusiones del Módulo de Implementación

En este capítulo se ha detallado la construcción práctica de la herramienta, materializando el diseño teórico en un plugin de Unity modular, escalable y orientado a usuarios no programadores. A continuación, se resumen los principales logros y aprendizajes:

- **Tecnología y arquitectura:** La elección de Unity 2022.3.18f1 LTS como motor base ha permitido aprovechar plenamente su sistema de componentes, prefabs y **MonoBehaviour**, garantizando compatibilidad futura y un entorno conocido para diseñadores y desarrolladores.
- **Módulo de actuadores:** Se implementó la clase abstracta **Actuator** y sus subtipos (**MovementActuator**, **SpawnerActuator**, etc.), permitiendo definir acciones diversas (movimiento con easing, generación de entidades) de forma coherente y reutilizable.
- **Módulo de sensores:** La jerarquía de **Sensor** (área, colisión, distancia y tiempo) facilita la detección de condiciones de juego mediante un patrón observador. Cada sensor notifica a las transiciones asociadas sin acoplamiento directo, potenciando la extensibilidad.
- **Módulo de daño:** Se separaron los roles de **DamageEmitter** y **DamageSensor**, regulando la emisión y recepción de daño de forma independiente y permitiendo integraciones limpias con sistemas de vida o efectos especiales.
- **Módulo de Máquina de Estados Finita:** La introducción de la clase **FSM** junto a **State** y **Transition** ha proporcionado un orquestador central para combinar sensores y actuadores en comportamientos coherentes, con cambios de estado controlados en **LateUpdate** para evitar inconsistencias en la lógica.
- **Módulo de animación:** Con **AnimatorManager** se ha logrado sincronizar los parámetros de movimiento, rotación y eventos (*Spawn*, *Damage*, *Die*, *ChangeState*) con un **Animator Controller**, garantizando coherencia visual entre la lógica de IA y las animaciones.
- **Módulo del jugador:** Se proporcionaron componentes auxiliares para movimiento (**PlayerMovement**), detección de colisiones (**PlayerCollisionDetection**), salto mejorado (**PlayerJump**), gestión de vida (**Life**) y ataque a distancia (**PlayerDistanceAttack**), cubriendo necesidades comunes de juego 2D y demostrando la flexibilidad del framework.

- **Distribución:** El empaquetado como plugin de Unity y la inclusión de un manual visual y de código de ejemplo facilitan su adopción en proyectos reales sin requerir modificaciones internas, cumpliendo el objetivo de accesibilidad para diseñadores.

En conjunto, el módulo de implementación ha validado la viabilidad del diseño conceptual: se ha construido una herramienta que, mediante una estructura modular y basada en patrones de diseño claros, permite configurar comportamientos de enemigos complejos sin escribir código, integrándose de forma natural en el flujo de trabajo de Unity. La implementación ha cubierto todos los aspectos planteados (movimiento, sensores, daño, estados y animaciones) y ha sentado las bases para futuras mejoras en extensibilidad y usabilidad.

Capítulo 5

Evaluacion Con Usuarios

Una vez finalizada la fase de implementación de la herramienta, se consideró fundamental validar su utilidad y usabilidad a través de pruebas con usuarios. Estas pruebas permiten obtener una primera impresión del sistema desde el punto de vista de quienes no han participado en el desarrollo, y proporcionan información clave sobre posibles mejoras o dificultades de uso.

Para ello, se diseñó un protocolo de evaluación estructurado que incluye objetivos claros, preguntas de investigación específicas y una guía detallada para la realización de las pruebas. El objetivo principal de esta fase es identificar brechas entre el diseño previsto y la experiencia real del usuario, así como recoger sugerencias que puedan orientar futuras iteraciones de la herramienta.

En las siguientes secciones se describen los objetivos y preguntas de investigación, el guion de evaluación utilizado, el desarrollo de las sesiones de prueba y el análisis de los resultados obtenidos.

5.1. Objetivos y preguntas de investigación

En el contexto de esta evaluación, se ha definido un objetivo general: comprender la experiencia de uso de los usuarios al interactuar por primera vez con la herramienta. Este objetivo engloba aspectos clave como la facilidad de uso, la claridad de la interfaz y la capacidad del sistema para guiar al usuario en la configuración de comportamientos sin necesidad de escribir código.

A partir de este objetivo general, se han formulado dos líneas de evaluación principales, cada una con sus correspondientes preguntas de investigación, que servirán para guiar tanto la observación como la recogida de datos cualitativos.

1. Evaluar la claridad e intuición de la herramienta. Esta línea se centra en determinar si los usuarios perciben la herramienta como fácil de entender y utilizar, incluso sin conocimientos técnicos. Lo más importante es que la interfaz y los flujos de trabajo resulten naturales, sin ambigüedades ni puntos de

bloqueo.

1. ¿El usuario entiende fácilmente cómo añadir funcionalidad al enemigo?

Esta pregunta pretende verificar si el usuario comprende el funcionamiento base de los enemigos, incluyendo el uso de la máquina de estados, la creación de nuevos estados, la configuración de transiciones, actuadores y Damage Emitters.

2. ¿Las opciones de comportamiento ofrecidas son comprendidas en su totalidad?

Se analiza si la funcionalidad de los distintos actuadores es clara para el usuario y si las descripciones disponibles en la herramienta son suficientes para usarlos correctamente.

3. ¿La interfaz comunica correctamente el propósito de cada estado y transición?

Evaluamos si el usuario puede interpretar visualmente en qué estado se encuentra el enemigo en cada momento y cómo se espera que se comporte.

4. ¿La documentación aclara y detalla la configuración de las funcionalidades?

Se analiza si el material de apoyo resulta suficiente y útil para ayudar al usuario a entender la herramienta sin necesidad de asistencia externa.

2. Validar la funcionalidad y utilidad práctica del sistema. Esta línea tiene como objetivo verificar que la herramienta no solo es comprensible, sino que también es útil y funcional en el flujo de trabajo de diseño de videojuegos.

1. ¿La herramienta agiliza y simplifica el proceso de diseño de enemigos?

Se estudia si los usuarios sienten que la herramienta mejora la eficiencia de su proceso creativo respecto a métodos más tradicionales.

2. ¿El comportamiento de los enemigos generados se corresponde con lo esperado en el videojuego?

Evaluamos si lo que el usuario configura en el editor se traduce correctamente al comportamiento observado en el entorno de juego.

3. ¿La herramienta permite la adaptación y expansión sencilla de los enemigos?

Esta pregunta busca determinar si es posible modificar o ampliar los enemigos existentes sin generar confusión ni complejidad innecesaria.

5.2. Diseño de la Evaluación

En esta sección se detalla la planificación y la estructura de las pruebas con usuarios.

5.2.1. Audiencia Objetivo

- Edad: mayor de 18 años
- Genero: No relevante
- Extracto Sociocultural: El público objetivo se centra en perfiles que tengan un extracto sociocultural relacionado con el diseño de videojuegos. Esto incluye:
 - Estudiantes actuales o pasados de diseño de videojuegos.
 - Profesionales que se dediquen o tengan interés en la creación de enemigos.
 - Personas que sin poseer estudios específicos demuestren tener interés en la creación de entidades y sus comportamientos.
- Habilidades Requeridas: Se espera que los participantes en la evaluación cuenten con conocimientos previos en diseño de personajes, conceptos básicos del funcionamiento de Unity y comprensión básica de maquinas de estado.

5.2.2. Duración y Entorno de Realización

Cada sesión de prueba se extenderá durante unos 60 minutos. Este lapso lo dedicaremos primero a una breve charla introductoria sobre la prueba. Luego, se pedirá que se explore el manual y realicen los ejemplos prácticos. Finalmente se realizará una entrevista que proporcionará feedback adicional.

Las sesiones se realizarán en entornos controlados y tranquilos donde el usuario no tenga distracciones. Se dispondrá un ordenador con la herramienta ya instalada y lista para su uso, así como teclado y ratón estándar. Al haber simultaneidad de pruebas con varios usuarios, para evitar la influencia entre ellos se procurará situarlos separados entre ellos y en caso de que no fuese posible, pedirles la mínima interacción posible entre ellos.

5.2.3. Descripción de Tareas del Probador

El probador deberá realizar las siguientes actividades:

- Lectura del manual (sin ejemplos de uso). Estimación: 10 minutos.
- Ejecución de ejemplos de uso. Estimación: 30 minutos.
- Creación libre de un enemigo. Estimación 10 minutos
- Realización del cuestionario. Estimación 5 minutos.
- Realización de la entrevista. Estimación 5 minutos.

Las tareas se presentarán de forma secuencial, permitiendo al usuario familiarizarse gradualmente con las funcionalidades de la herramienta.

5.2.4. Instrucciones Iniciales

Antes de comenzar las pruebas se agradecerá la participación de los usuarios. Se dejará claro al inicio que no es una crítica hacia ellos y que solo estamos evaluando la funcionalidad de la herramienta y en ningún caso juzgándoles, además, se les indicará las diferentes partes que consta la evaluación.

5.2.5. Comportamiento del Investigador

Durante toda la sesión de pruebas, el investigador actuará como observador imparcial y facilitador. Su presencia tendrá como objetivo principal garantizar que los participantes comprendan las instrucciones generales y ofrecer soporte logístico en caso de que surjan problemas técnicos o situaciones inesperadas. No se ofrecerá ayuda directa sobre el uso de la herramienta salvo que una situación impida completamente continuar con la evaluación, ya que se busca observar cómo los usuarios interactúan con la interfaz sin intervenciones externas.

El investigador evitará cualquier influencia en las decisiones o acciones del usuario, no corrigiendo ni orientando sobre cómo utilizar la herramienta. Anotará observaciones relevantes sobre el comportamiento del usuario, como expresiones de duda, errores recurrentes, pasos omitidos o dificultades generales en la navegación.

Al finalizar la sesión, el investigador facilitará la entrevista, siguiendo una estructura abierta pero guiada que permita profundizar en las impresiones subjetivas de cada usuario.

5.2.6. Metodología

La evaluación ha sido diseñada utilizando una metodología mixta, combinando técnicas cuantitativas y cualitativas con el objetivo de obtener una visión completa del desempeño de la herramienta, tanto en términos de usabilidad como de experiencia percibida por los usuarios.

5.2.6.1. Enfoque cualitativo

El enfoque cualitativo busca recoger impresiones, percepciones y comportamientos de los usuarios durante la interacción con la herramienta, a través de técnicas como la observación directa y la entrevista semiestructurada.

Observación Durante la realización de las tareas, el investigador observará de forma no intrusiva el comportamiento de los participantes. Se tomarán notas sobre:

- Dudas frecuentes o repetitivas.
- Tiempos estimados de realización por tarea.

- Momentos de frustración, vacilación o errores.
- Fluidez general en la interacción con la interfaz.

Esta observación permitirá identificar posibles problemas de usabilidad no detectables únicamente a través del cuestionario.

Puesta en común Tras la finalización del cuestionario SUS, se llevará a cabo una entrevista semiestructurada con cada participante. Esta dinámica de cierre busca profundizar en la experiencia del usuario, identificar áreas de mejora no previstas y recoger sugerencias espontáneas.

A continuación, se muestran algunas de las preguntas que servirán como guía para esta conversación:

- ¿Hubo algo que os sorprendiera o no esperabais durante el uso de la herramienta?
- ¿Qué parte del proceso de creación de enemigos os pareció más interesante o satisfactoria?
- ¿En qué momentos sentisteis que no sabíais muy bien qué hacer o cómo proceder?
- ¿Cambiaríais algo de la forma en la que se explican los elementos como estados, transiciones o actuadores?
- ¿Creéis que la herramienta permite expresar bien ideas de diseño complejas?
- ¿Consideráis que podríais usar esta herramienta para un proyecto real o profesional? ¿Por qué?
- ¿Hay alguna funcionalidad que echasteis de menos mientras la usabais?
- ¿Os surgieron ideas o sugerencias que podrían mejorar la herramienta?

5.2.6.2. Enfoque cuantitativo

El enfoque cuantitativo está centrado en recoger datos objetivos y medibles a través de un cuestionario estandarizado. En este caso, se ha utilizado el cuestionario SUS (System Usability Scale), desarrollado por Brooke (1996), con el fin de evaluar la percepción de usabilidad del sistema de forma estructurada y comparable.

Cuestionario SUS Una vez completadas las tareas, los participantes responderán al cuestionario SUS (System Usability Scale), propuesto por Brooke (1996). Este instrumento se compone de diez afirmaciones que deben valorarse mediante una escala de Likert de cinco puntos, donde 1 representa "totalmente en desacuerdo" y 5 representa "totalmente de acuerdo".

El cuestionario está diseñado para ofrecer una puntuación global de usabilidad del sistema. Sin embargo, en el contexto de esta evaluación, también se analizarán

de manera individual algunas de las respuestas, ya que ciertas afirmaciones están directamente relacionadas con las preguntas de investigación planteadas previamente. De esta forma, se podrá extraer información más específica sobre aspectos concretos de la herramienta, como su facilidad de aprendizaje, consistencia o integración funcional.

Las afirmaciones que componen el cuestionario son las siguientes:

1. Creo que me gustaría usar este sistema con frecuencia.
2. Encontré el sistema innecesariamente complejo.
3. Pensé que el sistema era fácil de usar.
4. Creo que necesitaría la ayuda de una persona técnica para poder usar este sistema.
5. Encontré que las diversas funciones de este sistema estaban bien integradas.
6. Pensé que había demasiada inconsistencia en este sistema.
7. Imaginaría que la mayoría de la gente aprendería a usar este sistema muy rápidamente.
8. Encontré el sistema muy engorroso de usar.
9. Me sentí muy confiado al usar el sistema.
10. Necesité aprender muchas cosas antes de poder usar este sistema.

Este cuestionario permitirá calcular una puntuación global de usabilidad, útil para comparar esta herramienta con otras o evaluar mejoras a lo largo del tiempo.

5.3. Resultados de las pruebas

5.3.1. Contexto de las sesiones

Las pruebas se llevaron a cabo en una única sesión de aproximadamente una hora de duración. Participaron siete voluntarios, todos ellos estudiantes del Máster en Diseño de Videojuegos o del Máster en Desarrollo de Videojuegos de la UCM, con conocimientos previos de diseño y, en algunos casos, de programación. Todas las evaluaciones se realizaron sobre la misma versión del proyecto; pese a que se utilizaron distintas versiones de Unity, no se detectaron problemas técnicos derivados de ello.

5.3.2. Datos cuantitativos

Para medir la usabilidad de la herramienta se empleó el cuestionario SUS (System Usability Scale) de Brooke (1996), que proporciona una puntuación de 0 a 100. La puntuación obtenida fue de $\bar{X} = 75,7$ (desviación estándar $\sigma = 16,1$), lo que supera el umbral de 68 puntos considerado “por encima de la media” en la literatura de usabilidad. Aunque el tamaño de la muestra es limitado, este resultado indica una percepción global positiva de la herramienta.

5.3.3. Datos cualitativos

Mediante observación directa y entrevista semiestructurada se recogieron impresiones de los usuarios:

- Dudas frecuentes sobre la existencia de tooltips (ningún participante los localizó espontáneamente).
- Confusión entre el estado de la máquina de estados y el estado de animación.
- Recomendaciones de gestionar las animaciones desde el Inspector en lugar de la pestaña *Animator*.
- Sugerencia de incluir vídeos tutoriales para ilustrar la configuración de comportamientos.
- Valoración positiva de la gran variedad de comportamientos y de la modularidad del sistema.
- Necesidad de un feedback visual más claro en casos como la aplicación de daño y herramientas de depuración extendidas.

5.4. Análisis de resultados

A continuación se analiza cómo estos datos responden a las preguntas de investigación formuladas en la Sección 5.1, combinando los tres métodos empleados (cuestionario SUS, observación y entrevista).

5.4.1. Claridad e intuición de la herramienta

Objetivo: Evaluar si la interfaz y los flujos de trabajo resultan naturales y sin ambigüedades.

- *Cuestionario SUS:*

- Los ítems 3 (*Pensé que el sistema era fácil de usar*) y 7 (*Imaginaría que la mayoría de la gente aprendería a usar este sistema muy rápidamente*) reflejan que más del 85 % de los usuarios seleccionaron valores altos (4 o 5), lo que indica una percepción general positiva respecto a la facilidad de uso y aprendizaje inicial.

- *Observación:*

- Se observaron tiempos adecuados de realización de las tareas, aunque algunos usuarios mostraron dudas concretas a la hora de añadir estados, transiciones o elementos como actuadores y emisores de daño.

- *Entrevista:*

- Ninguno de los participantes hizo uso de los tooltips integrados, lo cual se debió a que no sabían que existían. En las entrevistas manifestaron que no los habían detectado visualmente. Se concluye que una posible mejora sería mencionar explícitamente su uso en el manual, ya que modificar su diseño gráfico podría sobrecargar visualmente la herramienta.

5.4.2. Funcionalidad y utilidad práctica

Objetivo: Verificar si la herramienta agiliza y simplifica el proceso de diseño de enemigos, y si el comportamiento de los enemigos generados se corresponde con lo esperado dentro del videojuego.

- *Cuestionario SUS:*

- Este aspecto se refleja indirectamente en los ítems 1 (*Creo que me gustaría usar este sistema con frecuencia*) y 3 (*Pensé que el sistema era fácil de usar*). En ambos casos, la mayoría de los usuarios seleccionaron valores de 4 o 5, lo que sugiere una buena percepción de utilidad y eficiencia de la herramienta a la hora de diseñar enemigos.

- *Observación:*

- Durante las pruebas, se comprobó que los enemigos configurados se comportaban correctamente según los ejemplos incluidos en el manual. Sin embargo, algunos usuarios mostraron dificultades para comprender completamente el funcionamiento subyacente, lo que ralentizó ligeramente la ejecución de ciertas tareas.

- *Entrevista:*

- Los participantes sugirieron la incorporación de vídeos explicativos como complemento al manual actual. Consideraron que una guía visual podría facilitar el entendimiento del proceso de configuración y ayudar a relacionar mejor la lógica de edición con el comportamiento final observado en la escena.

5.4.3. Adaptación y extensibilidad

Objetivo: Determinar si es sencillo modificar o ampliar los comportamientos de los enemigos sin causar confusión.

- *Cuestionario SUS:*

- Ítem 10 (“Necesité aprender muchas cosas antes de usarlo”) registró principalmente valores bajos (1–2), indicando curva de aprendizaje baja.

- *Observación:*

- Los usuarios probaron con éxito variaciones de parámetros sin guía externa.

- *Entrevista:*

- La modularidad recibió elogios, aunque se planteó mejorar la visualización de las conexiones entre estados y actuadores.

En conjunto, el análisis muestra que la herramienta cumple su propósito de ofrecer una interfaz accesible y eficiente, si bien puede beneficiarse de mejoras en la visibilidad de ayudas contextuales (tooltips, tutoriales) y en la representación visual de las relaciones internas para reducir la curva de aprendizaje en configuraciones más complejas.

5.4.4. Conclusión general

La evaluación preliminar de la herramienta sugiere que la usabilidad es generalmente aceptable en su versión actual. Se perciben fortalezas en la claridad de la interfaz, la integración de funcionalidades y una baja percepción de complejidad. Sin embargo, es importante señalar que la muestra reducida de usuarios impide que estos datos sean estadísticamente significativos o generalizables.

Las observaciones y respuestas cualitativas de este pequeño grupo indican posibles áreas de mejora en la herramienta. A pesar de estas sugerencias, parece cumplir los objetivos definidos a un nivel básico para los usuarios que la han probado.

Capítulo 6

Conclusiones

A lo largo de este Trabajo de Fin de Grado se ha afrontado el reto de dotar a diseñadores de videojuegos 2D de una herramienta completa y fácil de usar para la configuración de enemigos, eliminando la necesidad de escribir código. Partiendo de un estudio exhaustivo del estado del arte —que incluyó soluciones basadas en máquinas de estados finitas, sistemas de scripting visual como PlayMaker y Blueprints, y entornos visuales de nodos— se identificaron las buenas prácticas de accesibilidad y modularidad que servirían de base al framework.

6.1. Síntesis de resultados

6.1.1. Diseño arquitectónico

Se definieron cinco módulos principales:

- **Sensor:** Con subtipos para detección de área, colisiones, distancia y tiempo. Este módulo adopta el patrón *observer* para notificar eventos de forma desacoplada, facilitando la creación de transiciones dinámicas sin acoplar lógica de alto nivel.
- **Actuator:** Clases abstractas que representan acciones (movimiento y generación de entidades). Gracias a la herencia y a parámetros configurables, cada actuador implementa su propia lógica de inicialización, actualización y destrucción.
- **Máquina de Estados Finita (FSM):** El módulo orquesta transiciones seguras entre estados, evitando inconsistencias y permitiendo definir comportamientos complejos mediante la combinación de sensores y actuadores.
- **Animación:** A través de la clase `AnimatorManager` y un `Animator Controller` personalizado, se sincronizan parámetros de velocidad, dirección y eventos (spawning, daño, muerte) con animaciones, integrando la lógica de IA con la representación visual.

- **Daño:** Separación clara entre `DamageEmitter` y `DamageSensor`, lo que permite aplicar y recibir daño de forma modular y disparar efectos secundarios (animaciones, eventos de UI) sin dependencias circulares.

Esta arquitectura por componentes garantiza extensibilidad, alta cohesión y bajo acoplamiento, habilitando nuevos módulos o variantes sin alterar el núcleo del framework.

6.1.2. Implementación y entorno

La puesta en práctica se realizó como un paquete de Unity 2022.3.18f1 LTS, que incluye:

- *Depuración:* Gizmos en escena, tooltips contextuales y opciones de modo debugging que facilitan la comprensión del flujo de datos.
- *Ejemplos de uso:* Escenas demostrativas que combinan distintos sensores y actuadores, validando la integridad de los estados y las animaciones.
- *Componentes para jugador:* Clases de movimiento avanzado, detección de colisiones, saltos mejorados, gestión de vida y ataque a distancia, para verificar la aplicabilidad del framework en un caso de uso real.

6.1.3. Validación con usuarios

La evaluación con siete participantes (estudiantes de los Másteres en Diseño y en Desarrollo de Videojuegos de la UCM) incluyó:

- **Método cuantitativo:** Cuestionario SUS con puntuación media de 75,7 ($\sigma = 16,1$), por encima del estándar de 68, indicando buena usabilidad.
- **Observación:** Identificación de cuellos de botella al añadir nuevos estados y poca visibilidad de tooltips.
- **Entrevistas semiestructuradas:** Feedback sobre la necesidad de tutoriales en vídeo, aclaración de documentación y sugerencias para mejorar la visualización de conexiones entre nodos.

Aunque la muestra fue limitada a una única sesión, los datos cualitativos y cuantitativos coinciden en destacar la claridad de los flujos de trabajo y la potencia de la modularidad, al tiempo que señalan áreas de mejora en la ayuda contextual.

6.2. Aportaciones y lecciones aprendidas

- **Accesibilidad para no programadores:** Se demuestra que un sistema de configuración visual, bien estructurado, puede rivalizar en potencia con soluciones basadas en código, permitiendo a diseñadores centrarse en la jugabilidad.

- **Importancia de la documentación interactiva:** La evaluación reveló que la documentación extensa no sustituye a guías prácticas (vídeos, tutoriales in-editor) para acelerar la curva de aprendizaje.
- **Modularidad como clave de mantenimiento:** Diseñar por módulos claros facilita la evolución del framework y la integración de nuevas funcionalidades sin provocar regresiones.
- **Sincronización lógica–visual:** Integrar FSM, sensores, actuadores y animaciones bajo un mismo pipeline asegura coherencia entre la lógica de IA y la experiencia perceptual del jugador.

6.3. Trabajo futuro

Para consolidar y ampliar lo desarrollado, se proponen las siguientes líneas de acción:

1. Mejora de la ayuda contextual y UX:

- Rediseñar e insertar tooltips más visibles y parametrizables.
- Crear tutoriales interactivos en el propio editor de Unity.

2. Ampliación de sensores y emisores:

- Añadir un sensor de sonido y emisores acústicos para comportamientos basados en estímulos sonoros.
- Incorporar sensores basados en proximidad temporal o condiciones múltiples (combinación de distancia y colisión).

3. Navegación y AI avanzada:

- Integrar algoritmos de *steering* para mayor complejidad en los movimientos.
- Desarrollar un sistema de memoria de interacciones y comunicación entre agentes para comportamientos cooperativos.

4. Interfaz gráfica de edición:

- Crear una ventana de editor con drag-and-drop de nodos (estados, transiciones) y paneles de propiedades.
- Implementar vistas filtrables y agrupaciones lógicas para gestionar proyectos con multitud de estados.

5. Mantenimiento y escalabilidad:

- Establecer un plan de actualizaciones periódicas frente a nuevas versiones de Unity y cambios en la API.

- Publicar el framework como paquete oficial de Unity para facilitar su adopción y contribución comunitaria.

En conjunto, este TFG ha sentado las bases para un entorno de desarrollo de IA y comportamientos de enemigos 2D accesible, potente y escalable, ofreciendo tanto a diseñadores como programadores una herramienta que mejora la eficiencia creativa y la coherencia técnica de sus proyectos.

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

6.4. 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.

6.5. Objectives

The main objective of this project is the design and development of a framework for the *Unity* game engine that simplifies and streamlines the process of creating enemies in 2D platformer games. This framework defines a modular structure of components and behaviors based on the analysis of common enemy types in such games, with the goal of completely separating the roles of programming and design. In this way, it enables individuals without programming knowledge to take on the role of enemy designer.

As a practical example of the framework, a functional tool has been developed in *Unity* that allows these components to be implemented and used in a visual and

intuitive manner. The tool includes a catalog of behaviors that is easy to use for anyone, regardless of their programming experience, as well as a user manual that clearly explains each component, installation steps, and usage examples.

To carry out this development, a structured work plan has been followed, ranging from an analysis of the development environment and a review of existing enemies, to the implementation of the tool, its validation with users, and analysis of the results obtained.

6.6. 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.

Chapter 7

Conclusions

Throughout this Final Degree Project, we have tackled the challenge of providing 2D game designers with a comprehensive and user-friendly tool for configuring enemy behaviors, eliminating the need to write code. Starting from an exhaustive review of the state of the art—including solutions based on finite state machines, visual scripting systems such as PlayMaker and Unreal’s Blueprints, and node-based visual editors—we identified the best practices in accessibility and modularity that would underpin our framework.

7.1. Summary of Results

7.1.1. Architectural Design

Five main modules were defined:

- **Sensor:** Subtypes for area detection, collisions, distance and time. This module adopts the *observer* pattern to broadcast events in a decoupled manner, simplifying the creation of dynamic transitions without embedding high-level logic.
- **Actuator:** Abstract classes representing actions (movement and entity spawning). Thanks to inheritance and configurable parameters, each actuator implements its own initialization, update, and destruction logic.
- **Finite State Machine (FSM):** Orchestrates safe transitions between states, preventing inconsistencies and enabling complex behaviors through the combination of sensors and actuators.
- **Animation:** Via the `AnimatorManager` class and a custom `Animator Controller`, parameters for speed, direction, and events (spawn, damage, death) are synchronized with animations, seamlessly linking AI logic to visual representation.
- **Damage:** Clear separation between `DamageEmitter` and `DamageSensor`, allowing damage application and reception in a modular fashion and triggering side-effects (animations, UI events) without circular dependencies.

This component-based architecture ensures extensibility, high cohesion, and low coupling, enabling new modules or variants to be introduced without altering the framework's core.

7.1.2. Implementation and Environment

The framework was implemented as a Unity 2022.3.18f1 LTS package, which includes:

- *Debugging tools*: Scene gizmos, contextual tooltips, and debug modes that help visualize data flow.
- *Usage examples*: Demonstration scenes combining various sensors and actuators to validate state integrity and animation correctness.
- *Player components*: Advanced movement, collision detection, enhanced jumping, health management, and ranged attack scripts to verify the framework's applicability in a complete scenario.

7.1.3. User Validation

Seven participants (students from the UCM's Game Design and Game Development master's programs) evaluated the tool via:

- **Quantitative method**: SUS questionnaire yielding an average score of 75.7 ($\sigma = 16.1$), above the usability benchmark of 68.
- **Observation**: Identification of bottlenecks when adding new states and low tooltip visibility.
- **Semi-structured interviews**: Feedback on the need for in-editor video tutorials, improved documentation clarity, and better visualization of node connections.

Despite the limited sample and single session, qualitative and quantitative data consistently highlight clear workflows and strong modularity, while pointing out areas for improving contextual help.

7.2. Contributions and Lessons Learned

- **Accessibility for non-programmers**: A well-structured visual configuration system can match the power of code-based solutions, allowing designers to focus on gameplay rather than syntax.
- **Importance of interactive documentation**: Extensive written manuals are no substitute for practical guides (videos, in-editor tutorials) in speeding up the learning curve.

- **Modularity as a maintenance cornerstone:** Designing clear, self-contained modules enables the framework to evolve and integrate new features without regressions.
- **Logic–visual synchronization:** Integrating FSM, sensors, actuators, and animations in a unified pipeline ensures coherence between AI logic and the user’s visual experience.

7.3. Future Work

To consolidate and extend this development, the following action lines are proposed:

1. Enhanced contextual help and UX:

- Redesign and surface more prominent, configurable tooltips.
- Create interactive tutorials embedded within the Unity editor.

2. Expanded sensors and emitters:

- Add a sound sensor and acoustic emitters for behavior based on auditory stimuli.
- Incorporate multi-condition sensors (e.g., combining distance and collision triggers).

3. Advanced navigation and AI:

- Integrate *steering* algorithms and obstacle avoidance.
- Develop a memory system for past interactions and inter-agent communication for cooperative behaviors.

4. Graphical editing interface:

- Implement a drag-and-drop node editor (states, transitions) with property panels.
- Provide filterable views and logical groupings to manage large state graphs.

5. Maintenance and scalability:

- Establish a periodic update plan to maintain compatibility with new Unity versions and API changes.
- Publish the framework as an official Unity package to encourage community adoption and contributions.

Overall, this project lays a solid foundation for an accessible, powerful, and scalable 2D enemy behavior framework, equipping both designers and programmers with tools that enhance creative efficiency and technical coherence in game development.

Capítulo 8

Contribuciones Personales

En esta sección se describen las aportaciones individuales de cada autor al desarrollo de la herramienta y la elaboración de la memoria. Aunque en la mayoría de los casos ambos autores han participado en conjunto, provocando que una misma tarea esté en ambos autores a la misma vez.

8.1. Contribuciones de Cristina Mora Velasco

8.1.1. Antecedentes

En el verano anterior al inicio del proyecto, comencé una investigación exhaustiva enfocada en cómo desarrollar herramientas efectivas, poniendo especial atención en facilitar las tareas de los diseñadores. El objetivo principal era hacer que las funcionalidades fueran lo más intuitivas y visuales posibles, reduciendo al máximo la necesidad de memorizar procedimientos complejos o comandos específicos. Esto es debido a que aunque pueda parecer algo sumamente sencillo, usan muchas herramientas distintas y tienen que tener grandes cantidades de información en la cabeza, haciendo que sea tedioso el tener que recordar funcionalidades que pueden ser reconocidas de manera más rápida y sencilla de manera visual. Adicionalmente, contaba ya con conocimientos previos sobre el funcionamiento y aplicación de las máquinas de estados. Este tema introducido por primera vez en la asignatura Fundamentos de Computadores y reforzado de forma práctica a lo largo de toda la carrera.

8.1.2. Aportaciones

8.1.2.1. Investigación

La investigación empezó en septiembre, tratando de recabar información relevante. Para eso primero realicé un análisis en profundidad de los distintos tipos de enemigos que existen en el Hollow Knight separando entre enemigos base y jefes finales. Centrándome en los primeros ya que son los que aparecen con más frecuencia. Analice su comportamiento creando descripciones y máquinas de estado sobre su comportamiento. permitió identificar patrones de comportamientos similares entre

diferentes enemigos a nivel visual, donde en realidad la implementación es la misma. Esto se puede ver mucho mejor en el juego bzzzt que también analicé donde un mismo comportamiento se puede ver hasta en cinco enemigos diferentes. Todo este análisis permitió no solo encontrar elementos comunes como ya he mencionado, sino también encontrar los comportamientos más repetidos. Todo esto hizo que tuviera una estructura clara en la cabeza que permitiese ser escalada de forma sencilla y que tuviera todos los comportamientos identificados.

En paralelo a este trabajo realicé la lectura de diferentes trabajos realizados como Generador de comportamientos de enemigos para videojuegos 2D de Daniel Quintero o Herramientas De Diseño Basado En Bocetos Del Comportamiento En Videojuegos de Jorge Antonio Magallanes Borbor. Además de los artículos de investigación también use conocimientos de conferencias de la GDC que permitió tener un conocimiento general sobre el entorno de la herramienta, saber que estaba ya hecho y de qué forma, pudiendo analizarlos encontrando puntos débiles, fortalezas e información relevante para nuestra estructura.

8.1.2.2. Confección de la herramienta

- **Movimiento Vertical:** Desarrollo del movimiento vertical, con movimiento constante y acelerado, incluyendo colisiones, velocidad, lanzamiento...
- **Movimiento Horizontal:** Desarrollo del movimiento horizontal, con movimiento constante y acelerado, incluyendo colisiones, velocidad, lanzamiento...
- **Movimiento circular:** Desarrollo del movimiento circular y solución de problemas relacionados con el cálculo de posiciones alrededor de un punto de rotación.
- **Movimiento por splines:** Desarrollo del movimiento por splines creando una integración del sistema de splines de Unity para permitir enemigos que sigan trayectorias curvas predefinidas.
- **Actuador Spawner:** Desarrollo del componente encargado de generar enemigos u objetos, incluyendo control por intervalos, posición y número de repeticiones.
- **Simplificaciones en Move to a Point:** Revisé el componente de movimiento hacia puntos fijos para mejorar su configuración y permitir comportamientos uniformes o personalizados por waypoint.
- **Collision Sensor:** Desarrollo del sensor encargado de las colisiones.
- **Sensor base:** Implementación de la lógica base del Sensor. Programación del sensor de tiempo y su clase base, permitiendo activar transiciones tras un periodo específico.
- **Sensor de distancia:** Implementación del sensor que detecta la proximidad a un objetivo.

- **Sensores de daño:** Creación de distintos comportamientos que producían daño. Más adelante fueron unificados en uno solo simplificando su uso e integración.
- **Sensor de Tiempo:** Implementación del sensor, que envía un mensaje tras un tiempo especificado.
- **Temporizador:** Creación de la clase base Timer que se utiliza para el SpawnerEnemy y para el Sensor de Tiempo.
- **Movimiento inicial del jugador:** Desarrollo de una versión básica del sistema de control del jugador, que más adelante fue mejorado e implementado por mi compañero.
- **Gestión de animaciones:** Implementación del sistema Animator Manager para conectar los estados de la FSM con las animaciones del enemigo, incluyendo control de dirección (flip) y vinculación con controladores.
- **Controladores de animaciones:** Creación de una máquina de estados base con todos los estados más comunes que se dan en los enemigos y que para usarla solo hace falta cambiar los clips de esta.
- **Sistema de vida:** Implementación del componente Life, que gestiona la salud tanto del jugador como de los enemigos, enlazado a sensores de daño.
- **Dibujado de elementos:** Desarrollo de elementos visuales auxiliares dentro de los componentes para facilitar la visualización de trayectorias, áreas de detección y puntos de aparición.
- **Editores personalizados:** Programación de editores en Unity para la configuración intuitiva de componentes.
- **FSM y State base:** Implementación de la lógica base de las máquinas de estados y de los estados individuales.
- **Limitación en la lista de actuadores y creación de transiciones:** Desarrollo de una validación para evitar que un mismo tipo de actuador se añada más de una vez a un mismo estado, evitando errores de ejecución. Además de la implementación de la lista de transiciones.
- **Subscripción a eventos:** Diseño de un sistema de eventos que permite que un sensor mande información.
- **Creación de escenas de enemigos:** Elaboración de escenas de ejemplo que muestran enemigos funcionales usando las distintas combinaciones de sensores, actuadores y estados.
- **Comentarios explicativos:** Añadí comentarios extensivos en el código y en las escenas para facilitar la comprensión del comportamiento de cada enemigo.

- **Tooltips personalizados:** Implementación de descripciones emergentes (tooltips), facilitando el uso de la herramienta sin necesidad de consultar el manual constantemente.
- **Creación del proyecto:** Configuración inicial del proyecto en Unity y sistema de carpetas.
- **Exportación de la herramienta:** Preparación de la herramienta como un paquete Unity exportable, con estructura organizada y funcionalidad probada. Aunque esto lo inicie yo, mi compañero hizo la versión final.
- **Importación de assets:** Integración de los assets visuales del proyecto y adecuación de sus propiedades para su uso en el framework.
- **Revisión general del código:** Revisión exhaustiva de todo el código fuente para garantizar la limpieza y consistencia.

8.1.2.3. Parte de la memoria

organización en orden de aparición en la memoria:

- **Corrección del resumen y de los agradecimientos:** Revisión completa del resumen y agradecimientos.
- **Abstract:** Redacción de el resumen introductorio en inglés y las *keywords*.
- **Introducción:** Redacción completa de los apartados de motivación, objetivos y plan de trabajo, donde se justifica la necesidad de la herramienta y se establece una organización de trabajo.
- **Organización del estado de la cuestión:** Elección de técnicas, herramientas y motores que se iban a explicar, explicadas por mi compañero.
- **Corrección general del estado de la cuestión:** Revisión de los contenidos.
- **Descripción del trabajo:** Explicación detallada de las decisiones de diseño e implementación, reflejando tanto los aspectos técnicos como la justificación de decisiones.
- **Implementación:** Redacción de los actuadores y Animations, es decir el 4.3 y el 4.5
- **Evaluación con usuarios:** Redacción de la sección dedicada a la evaluación, que incluye:
 - Introducción.
 - Objetivos y preguntas de investigación.
 - Audiencia objetivo.
 - Duración y Entrono de Realización.

- Descripción de Tareas de probador.
 - Instrucciones iniciales.
 - Cuestionario SUS.
- **Corrección general de la memoria:** Lectura final completa del documento, detectando errores y posibles mejoras en redacción.
 - **Conclusiones y Trabajo Futuro:** Redacción de la sección dedicada añas conclusiones obtenidas y el trabajo que se realizará en un futuro.

8.1.2.4. Manual

Me encargué íntegramente de la redacción y organización del manual de usuario del framework, tanto en su versión en español como en inglés. Teniendo en mente siempre que tenía que ser accesible para los diseñadores y por tanto no tener ningún tecnicismo en él. El manual pretende ser una parte fundamental de la herramienta que permita comprender a cualquier usuario el funcionamiento de la herramienta y conocer todos sus componentes.

La estructura general del manual fue diseñada con el objetivo de guiar al usuario desde la comprensión conceptual de la herramienta hasta su aplicación práctica. Está organizado en los siguientes bloques principales:

- Introducción breve: Contexto de la herramienta, conocimientos que se asumen y estructura del documento.
- Funcionalidad de la herramienta y del manual: Objetivo de la herramienta y del manual.
- Público objetivo: A quién va dirigida la herramienta.
- Requisitos e instalación: Descripción por pasos para la instalación. Así como la especificación de recursos necesarios para esta.
- Contenido del paquete: Organización de los archivos que se imprimen, explicando la utilidad de cada carpeta.
- Componentes Detallados: Es el bloque más extenso del manual y el más técnico. Aquí documenté de forma minuciosa todos los elementos del framework. Dividido en secciones: Actuadores, Sensores, Máquinas de estado, animaciones y Vida.
- Ejemplos prácticos: Diseñé y documenté cinco ejemplos representativos que abarcan desde enemigos muy simples (como pinchos) hasta comportamientos más complejos.
- Preguntas frecuentes y solución a posibles errores: tabla de errores comunes detectados durante el testeo y sus posibles soluciones.

- Glosario: Redacté un glosario técnico con definiciones breves y claras de todos los conceptos importantes para facilitar la comprensión a personas no expertas en programación o desarrollo de videojuegos.
- Soporte: Incluí la información de contacto para que los usuarios pudieran resolver dudas o proponer mejoras.

8.2. Contribuciones de Francisco Miguel Galván Muñoz

8.2.1. Antecedentes

Antes de comenzar este proyecto, ya tenía una base sólida sobre Máquinas de Estados gracias a asignaturas como *Fundamentos de Computadores* e *Inteligencia Artificial*, donde implementamos por primera vez una en Unity. En esa práctica, la arquitectura propuesta resultó poco escalable y compleja, especialmente para modelar comportamientos básicos como patrullar, perseguir o atacar, lo que motivó mi interés por buscar soluciones más eficientes.

Aunque siempre me había interesado el diseño de enemigos, no lo había explorado teóricamente hasta este proyecto. Durante su desarrollo, investigamos patrones comunes en enemigos de videojuegos 2D, lo que nos permitió identificar comportamientos reutilizables y orientar nuestra herramienta hacia una arquitectura modular y accesible para desarrolladores.

8.2.2. Aportaciones

8.2.2.1. Investigación

Cuando comenzamos con la etapa de investigación, a mediados de septiembre, tanto Cristina como yo seleccionamos varios videojuegos con el objetivo de analizar el comportamiento de sus enemigos más representativos. En mi caso, elegí *Blasphemous* y adopté una dinámica basada en la observación directa: mientras jugaba, capturaba imágenes cada vez que aparecía un enemigo nuevo y realizaba un análisis detallado de su comportamiento. Esta dinámica la mantuve durante las primeras tres horas de juego, lo cual me permitió identificar paralelismos entre distintos tipos de enemigos, especialmente en lo referente a la forma en que se activaban las transiciones entre estados y cómo variaban sus comportamientos en función del estado en el que se encontraban.

Paralelamente a esta labor práctica, llevé a cabo una investigación teórica consultando artículos académicos (*papers*) y conferencias de la GDC (Game Developers Conference), como la de Fan (2016), con el fin de comprender cómo se aborda el diseño de enemigos a nivel profesional. Esta información fue de gran utilidad para

dar forma a los fundamentos de nuestra herramienta.

Una vez finalizada la fase de recopilación de información por ambas partes, celebramos varias reuniones para poner en común los hallazgos, identificar similitudes entre nuestros análisis y definir una base conceptual común. Gracias a este trabajo colaborativo, pudimos orientar adecuadamente la arquitectura de la herramienta, y comenzamos con el diseño e implementación de los primeros sensores y actuadores que formarían parte del sistema.

8.2.2.2. Confección de la herramienta

- **Actuator:** Diseño e implementación de clase padre **Actuator** y relación con sus clases hijas.
- **MovementActuator:** Desarrollo de nuevo nivel de abstracción de **Actuator** para todas las clases que representen movimientos (se excluye la clase encargada de crear enemigos). Integración de *Easing Functions* en todos los movimientos excepto el movimiento basado en splines.
- **VerticalActuator:** Implementación del desplazamiento vertical con movimiento constante, resolución de errores en la aceleración. Se incluye el manejo de colisiones, control de velocidad y comportamiento de lanzamiento.
- **HorizontalActuator:** Implementación del desplazamiento horizontal con movimiento constante, solución a problemas relacionados con la aceleración. También se contemplan las colisiones, la velocidad y el comportamiento de lanzamiento.
- **CircularActuator:** Implementación del movimiento circular, incluyendo su variante en modo péndulo, con soporte para aceleración mediante *Easing Functions* y visualización de trayectoria para depuración.
- **SpawnerActuator:** Corrección de errores y mantenimiento de la clase.
- **MoveToAPointActuator:** Realización de la implementación íntegra del componente.
- **MoveToAnObjectActuator:** Realización de la implementación íntegra del componente.
- **DirectionalActuator:** Implementación de la clase surgida de la necesidad de crear proyectiles que fueran hacia el jugador al instanciarse.
- **Sensor:** Implementación de la lógica base de la clase padre **Sensor**, que sirve como estructura común para los distintos tipos de sensores.
- **CollisionSensor:** Desarrollo del sensor encargado de detectar colisiones. Se resolvió un bug que impedía detectar colisiones persistentes si el sensor se activaba tras el impacto.

- **DistanceSensor:** Implementación del sensor de distancia, incluyendo visualización para depuración y capacidad para identificar si un objeto se encuentra dentro o fuera del rango especificado.
- **DamageSensor:** Desarrollo inicial y posterior refactorización de toda la lógica del daño, ya que originalmente se utilizaba el sensor tanto para recibir como para emitir daño, lo cual motivó su separación en componentes específicos.
- **DamageEmitter:** Creación de la clase encargada de gestionar la emisión de daño. Esta clase contiene toda la información relevante que se consulta cuando un objeto que infinge daño interactúa con otro que puede recibirla.
- **Life:** Creación del sistema de vida que gestiona la salud de todas las entidades de la herramienta usando un **DamageSensor**.
- **TimeSensor:** Solución de error donde se utilizaba una única instancia de **Timer** para dos tiempos diferentes.
- **PlayerMovement:** Modificación de la clase proporcionada por *Mix And Jam* eliminando lógica que no nos interesaba.
- **PlayerDistanceAttack:** Creación de componente para ataque a distancia del jugador.
- **PlayerCollisionDetection:** Modificación de la clase proporcionada por *Mix And Jam* añadiendo la opción de visualizar las cajas de colisión que detectan los eventos de posibilidad de salto (caja inferior) o de deslizamiento por las paredes (cajas laterales).
- **Solución de errores de animaciones:** Surgió un error donde al eliminar un estado o transición del **Animator Controller** del que derivan todos los demás, *Unity* no hacía la limpieza de éstos bien dejando basura. Para ello había que borrar manualmente esos bloques de código sobrantes.
- **Editores personalizados:** Implementación de la lógica necesaria para que la información requerida en el inspecto de *Unity* fuera acorde a los valores que se tengan en todo momento.
- **FSM:** Creación de la clase que representa la Máquina de Estados y lógica necesaria.
- **State:** Lógica de la clase excepto transiciones entre estados.
- **Creación de escenas enemigos:** Creación y ajuste de escenas de enemigos de prueba para que fueran lo más cercanos a lo que se ve en un videojuego comercial posible. También realicé un pequeño nivel de prueba donde se combinaban una serie de enemigos.
- **Documentación del código:** Documentación del código para su posible corrección y para que su mantenimiento sea más sencillo.

- **Tooltips personalizados:** Confección de *tooltips* para que el uso de la herramienta fuera más sencillo.
- **Exportación de la herramienta:** Preparación para que la herramienta pueda ser importada desde un repositorio de *GitHub*.
- **Revisión general del código:** Revisión exhaustiva de todo el código fuente para garantizar la limpieza y consistencia.

8.2.2.3. Implicación en la memoria

En cuanto a la redacción de la memoria, estuve involucrado activamente en todo el proceso de elaboración. A continuación, detallo mi participación en cada uno de los capítulos:

- **Resumen:** Redacción completa del resumen en ambos idiomas, así como de los agradecimientos.
- **Introducción:** Revisión general de todos los apartados del capítulo.
- **Estado de la Cuestión:** Redacción completa del capítulo.
- **Diseño del Framework:** Redacción inicial parcial del capítulo durante la fase de investigación, con posterior revisión de las modificaciones realizadas por mi compañera. Además, a partir de los comentarios del tutor, redacté varios contenidos adicionales, entre ellos, los ejemplos de uso de la herramienta.
- **Implementación:** Escritura íntegra del capítulo, a excepción de los apartados 4.2 y 4.6, que posteriormente actualicé en base a las observaciones del tutor.
- **Evaluación con Usuarios:** Revisión completa del capítulo y redacción de los apartados 5.2.5 y 5.2.6, excluyendo la sección dedicada al cuestionario SUS.
- **Conclusiones y Trabajo Futuro:** Revisión general del capítulo.
- **Corrección general de la memoria:** Lectura exhaustiva e iterativa del documento para detectar errores y proponer mejoras de redacción.
- **Últimas modificaciones:** Revisión final de la memoria y últimas modificaciones propuestas por el tutor.

8.2.2.4. Manual

Mi participación en la elaboración del manual fue de apoyo al principio y con el tiempo acabé encargándome de:

- **Glosario:** Realización del glosario.
- **Actualización de contenidos:** Cada cambio que se hacía en el Framework debía verse reflejado en el manual, por lo que me encargué de ello.

- **Revisión general de contenidos:** Revisión de los contenidos del manual y búsqueda de erratas o contenido que pudiera ser explicado de manera más entendible para un diseñador.
- **Actualización del manual:** Tras las pruebas de usuario, se hicieron una serie de cambios en el manual producto al *feedback* recibido.

Todo estas aportaciones se dieron en ambas versiones, español e inglés.

Bibliografía

*Y así, del mucho leer y del poco dormir, se
le secó el celebro de manera que vino a
perder el juicio.
(modificar en Cascaras\bibliografia.tex)*

Miguel de Cervantes Saavedra

2K GAMES. Bioshock. <https://2k.com/games/bioshock/bioshock-1/>, 2007.

BAKKES, S. *Rapid adaptation of video game AI*. Tesis Doctoral, Tilburg University, Países Bajos, 2010.

BORBOR, J. A. M. Herramientas De Diseño Basado En Bocetos Del Comportamiento En Videojuegos. 2012. Disponible en <https://docta.ucm.es/rest/api/core/bitstreams/f460a74b-5e6c-4bbb-8089-e6ac6a6667fe/content> (último acceso, May, 2025).

ELECTRONIC ARTS. Spore. 2008. Disponible en <https://www.ea.com/es-es/games/spore> (último acceso, May, 2025).

FAN, G. Rules of the Game: Five Techniques from Quite Inventive Designers. 2016. Disponible en <https://youtu.be/d8QAVGeEj-U> (último acceso, May, 2025). Minuto 27:56.

GAMASUTRA. GDC 2005 Proceeding: Handling Complexity in the *Halo 2* AI. 2005. Disponible en <https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai> (último acceso, May, 2025).

GOPALAKRISHNAN, K. y PRADEEP, R. Finite state machine in game development. 2021. Disponible en <https://www.ijarsct.co.in/Paper2062.pdf> (último acceso, January, 2025).

HOPE, A. The perfect organism: The AI of Alien: Isolation. 2014. Disponible en <https://www.gamedeveloper.com/design/the-perfect-organism-the-ai-of-alien-isolation> (último acceso, January, 2025).

- IAN MILLINGTON AND JOHN FUNGE. *Artificial Intelligence for Games*. 2016. Disponible en <https://theswissbay.ch/pdf/Gentoomen%20Library/Game%20Development/Programming/Artificial%20Intelligence%20for%20Games.pdf> (último acceso, January, 2025).
- ORKIN, J. Applying goal-oriented action planning to games. 2004. Disponible en https://web.archive.org/web/20230912173044/https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf (último acceso, January, 2025).
- PATASHNIK, O. Build a Bad Guy Workshop. 2014. Disponible en <https://www.gamedeveloper.com/design/build-a-bad-guy-workshop---designing-enemies-for-retro-games> (último acceso, January, 2025).
- SAGREDO-OLIVENZA, I., GÓMEZ-MARTÍN, M. A. y GONZÁLEZ-CALERO, P. A. Un modelo integrador de máquinas de estados y árboles de comportamiento para videojuegos. *Actas del Congreso de Software Educativo y de Conocimiento (COSECVI)*, 2016. Disponible en <https://gaia.fdi.ucm.es/sites/cosecivi14/es/papers/27.pdf> (último acceso, May, 2025).
- YANNAKAKIS, G. N. y TOGELIUS, J. *Artificial Intelligence and Games*. Springer, 2018. ISBN 978-3-319-63518-7. Disponible en <https://doi.org/10.1007/978-3-319-63519-4> (último acceso, January, 2025).

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.

