
**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	7
2.2.2. Árboles de comportamiento	10
2.2.3. Goal-Oriented Action Planning	11
2.3. Análisis de herramientas para la creación de comportamientos inteligentes	13
2.3.1. Behavior Bricks	14
2.3.2. PlayMaker	15
2.4. Motores de videojuegos	16
2.4.1. Unity	16
2.4.2. Unreal Engine	18
2.4.3. Godot	19
2.4.4. GameMaker	21
2.5. Conclusiones	23
3. Diseño del framework	25
3.1. Descripción general del Framework	25
3.1.1. Análisis de enemigos en videojuegos	25
3.2. Composición	26
3.2.1. Identificación de comportamientos comunes	26
3.3. Actuadores	27
3.3.1. Movimiento	28
3.3.2. Spawner	29
3.3.3. Compatibilidad	30
3.4. Sensores	30

3.5.	Daño	31
3.6.	Estado	31
3.7.	Máquina de estados finita	31
3.8.	Ejemplos de uso	32
3.8.1.	Bouncing Bunny	32
3.8.2.	Spinning Rocks	33
3.8.3.	Trunk Torret	34
3.8.4.	Spline Chicken	35
3.8.5.	Skywatch Eagle	36
4.	Implementación	37
4.1.	Tecnología utilizada	37
4.2.	Actuator	38
4.2.1.	MovementActuator	39
4.2.2.	SpawnerActuator	44
4.3.	Sensor	45
4.3.1.	AreaSensor	46
4.3.2.	CollisionSensor	46
4.3.3.	DistanceSensor	47
4.3.4.	TimeSensor	48
4.4.	Damage	49
4.4.1.	DamageEmitter	49
4.4.2.	DamageSensor	51
4.5.	Máquina de Estados Finita	51
4.5.1.	Transition	52
4.5.2.	State	52
4.6.	Animation Manager	52
4.7.	Jugador	54
4.7.1.	PlayerMovement	55
4.7.2.	PlayerCollisionDetection	55
4.7.3.	PlayerJump	56
4.7.4.	Life	57
4.7.5.	PlayerDistanceAttack	57
4.8.	Distribución del Framework	58
4.9.	Conclusiones	58
5.	Evaluacion Con Usuarios	59
5.1.	Objetivos y preguntas de investigación	59
5.1.1.	¿Resulta intuitiva y clara la herramienta para el usuario?	59
5.1.2.	¿El sistema de creación de enemigos demuestra ser funcional?	60
5.2.	Diseño de la Evaluación	61
5.2.1.	Audiencia Objetivo	61
5.2.2.	Duración y Entorno de Realización	61
5.2.3.	Descripción de Tareas del Probador	62

5.2.4. Instrucciones Iniciales	62
5.2.5. Comportamiento del Investigador	62
5.2.6. Diseño de la evaluación	63
5.2.7. Preparación de la Ejecución	65
6. Conclusiones y Trabajo Futuro	67
6.1. Motivation	69
6.2. Objectives	69
6.3. Work Plan	70
Conclusions and Future Work	73
7. Contribuciones Personales	75
7.1. Contribuciones de Cristina Mora Velasco	75
7.1.1. Antecedentes	75
7.1.2. Aportaciones	75
7.2. Contribuciones de Francisco Miguel Galván Muñoz	80
7.2.1. Antecedentes	80
7.2.2. Aportaciones	80
Bibliografía	85

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)	8
2.2. Ejemplo de maquinas de estado jerárquicas extraido de Borbor (2012)	9
2.3. Ejemplo BT, Documentación 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	14
2.6. Ejemplo de uso de Play Maker	16
2.7. Inspector de Unity, con una serie de componentes	17
2.8. Blueprints en Unreal Engine	19
2.9. Sistema VisualScript en Godot	20
2.10. Drag and Drop en GameMaker	22
3.1. Imagen de escenario de <i>Hollow Knight</i> donde aparecen dos Reptacillos	28
3.2. Easing Function que describe la variación de la posición de un objeto con el movimiento Move To An Object	29
3.3. Enemigo General	32
3.4. Escena de ejemplo donde observamos al enemigo en cuestión momentos antes de alcanzar la pared.	33
3.5. Escena de ejemplo donde se aprecian dos enemigos realizando un movimiento circular cada uno en un sentido diferente	34
3.6. Escena de ejemplo donde se aprecia una <i>TrunkTorret</i> disparando.	35
3.7. Escena de ejemplo donde se aprecia una <i>Spline Chicken</i> siguiendo su trayectoria.	35
3.8. Escena de ejemplo donde se aprecia a la <i>Skywatch Eagle</i> atacando al jugador.	36
4.1. Uso de Spline para definir el camino de un enemigo.	42
4.2. Modo Random Area, se aprecia una zona azul (DistanceSensor), un rectángulo gris (zona donde aparecen nuevos puntos) y por último un punto amarillo (waypoint actual).	43

4.3.	AreaSensor utilizado para enemigo que cae al detectar al jugador.	46
4.4.	Medición de distancia a través de la magnitud.	48
4.5.	Medición de distancia en el eje X en su lado negativo.	49
4.6.	Distribución del inspector de clase State	53
4.7.	AnimatorController con sus estados y transiciones.	54
4.8.	Representación de las cajas de detección de colisiones del jugador . .	56

Índice de tablas

2.1. Comparativa de técnicas de toma de decisiones para NPs	7
3.1. Matriz de compatibilidad de movimientos	30

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 una herramienta para el motor de videojuegos *Unity* que simplifique y agilice el proceso de creación de enemigos en juegos plataformas 2D y separe los roles de programación y diseño completamente, permitiendo que no sea necesario el conocimiento en programación para un puesto de diseñador. La herramienta contará con un catálogo de comportamientos fácil de manejar para cualquier persona independientemente de

sus conocimientos de programación, así como de un manual de usuario, que explicará de forma clara cada componente de la herramienta, la instalación y ejemplos de uso.

Para llevar a cabo el desarrollo de nuestra herramienta, hemos organizado un plan de trabajo que permite alcanzar todos los objetivos de forma óptima, pasando desde el estudio del entorno, hasta la evaluación con usuarios y su análisis.

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.

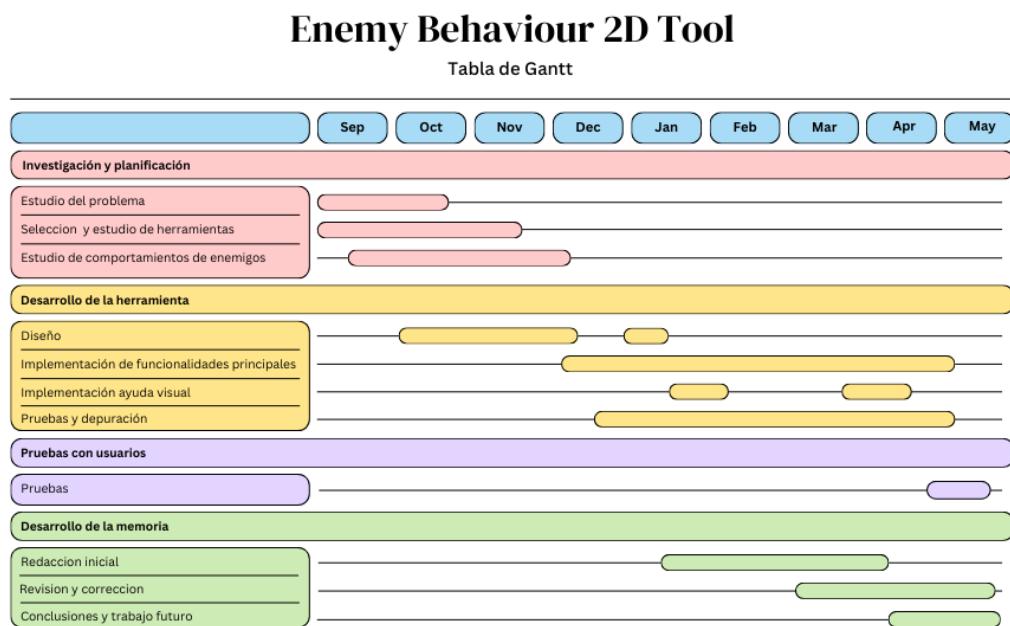


Figura 1.1: Diagrama de planificación del desarrollo de la herramienta Enemy Behaviour 2D

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,

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

así como de los recursos disponibles. En nuestro caso, hemos optado por utilizar máquinas de estados finitas, por su idoneidad en contextos de decisión estructurada y controlada.

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, Millington y Funge 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

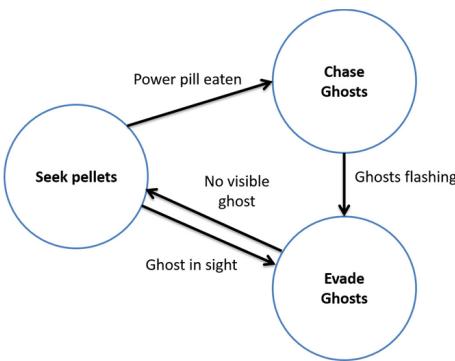


Figura 2.1: Comportamiento del jugador en *Pac-Man*, basado en una máquina de estados finita, extraído de Yannakakis y Togelius (2018).

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

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

comiendo bolitas, comienza a aumentar su velocidad.

- Pinky: como su nombre indica es el fantasma de color rosa. En japonés se llama *Machibuse*, el que tiende emboscadas. Pinky es el interceptor del juego por lo que va a tratar de cortar el camino del jugador. Es un fantasma relativamente rápido, por lo que calculará constantemente hacia donde se dirige el jugador para usar su velocidad para adelantarse y cortar el paso.
- Inky: el fantasma azul es el más impredecible de todos, deambula tranquilo por el laberinto hasta que está cerca del jugador y entonces, lo persigue.
- Clyde: el fantasma naranja y el más tranquilo de todos. Suele ser el último en salir de la casa de los fantasmas y no intentará atrapar al jugador en ningún caso.

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

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.

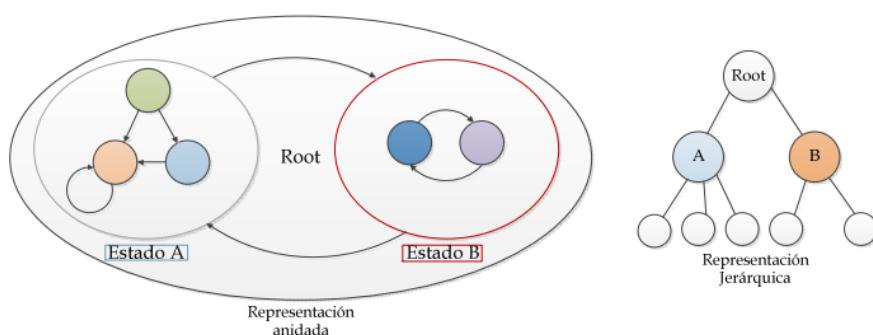


Figura 2.2: Ejemplo de maquinas de estado jerárquicas extraido de Borbor (2012).

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* (Wikipedia contributors) 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* (Arts, n.d.). *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) y 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.

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.

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

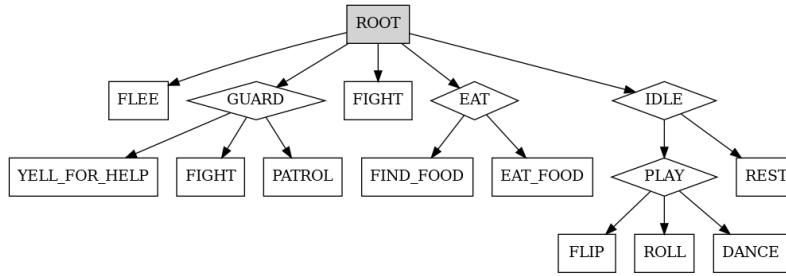


Figura 2.3: Ejemplo BT, Documentación Spore

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 tendrá 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*⁵. 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 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 como calcular su relevancia actual y sabe cuando ha sido alcanzado.

Aunque conceptualmente son similares, hay una diferencia clave entre los objetivos en *NOLF 2* y GOAP y es que en el primero cada objetivo tiene un plan

⁵<https://www.gdcvault.com/play/1013282/Three-States-and-a-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.

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.

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

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

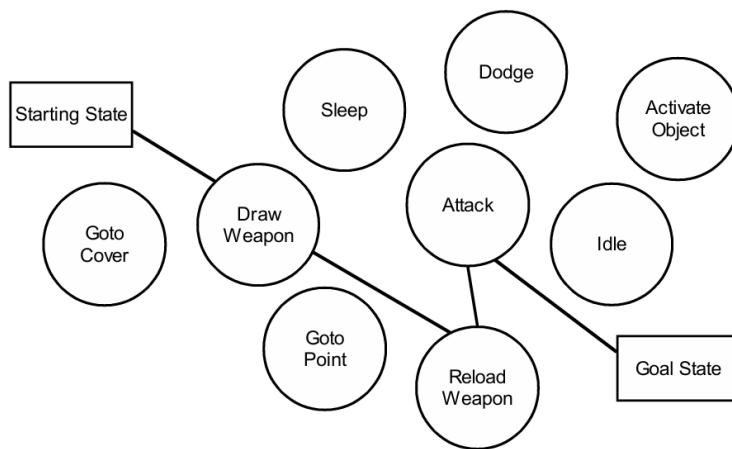


Figura 2.4: Ejemplo ilustrativo de GOAP donde el estado final es la eliminación de un enemigo.

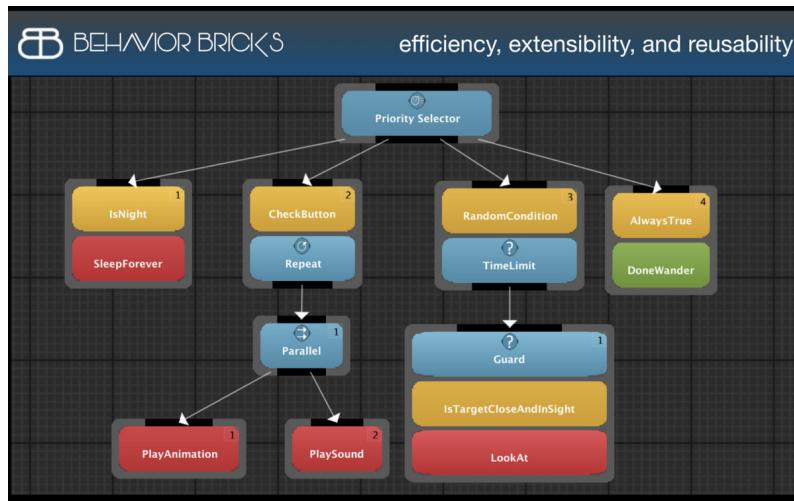


Figura 2.5: Ejemplo de uso de Behavior Bricks

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 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) en el contexto del Congreso de Software Educativo y de Conocimiento (COSECIVI).

2.3.2. PlayMaker

PlayMaker⁶ 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 ejecutar *Unity*. A partir de ahí, el usuario puede agregar más estados y definir eventos que permiten cambiar entre ellos según ciertas condiciones.

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

Gracias a su flexibilidad y facilidad de uso, PlayMaker es ideal para diseñar IA,

⁶<https://hutonggames.com/>

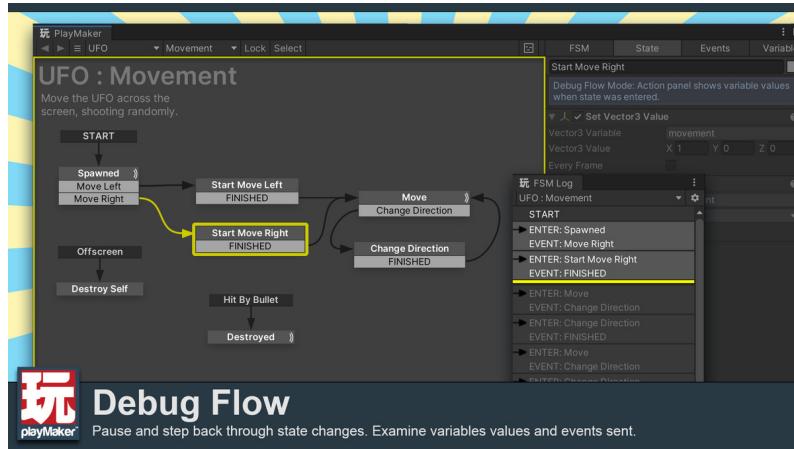


Figura 2.6: Ejemplo de uso de Play Maker

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*⁷, *Firewatch*⁸ la aventura narrativa del estudio norteamericano Campo Santo o el juego de plataforma de los creadores de *Limbo*, *INSIDE*⁹.

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 ajustan 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*¹⁰ es un motor de videojuegos desarrollado por *Unity Technologies* que se ha convertido en una de las herramientas más utilizadas en la industria del desa-

⁷<https://www.hollowknight.com/>

⁸<https://www.firewatchgame.com/>

⁹https://inside.fandom.com/wiki/Inside_Wiki

¹⁰<https://unity.com>

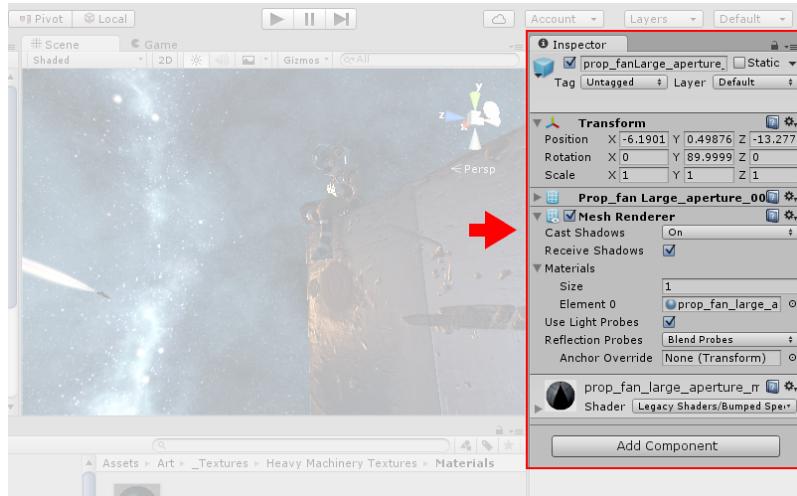


Figura 2.7: Inspector de Unity, con una serie de componentes

rrollo de videojuegos. Su versatilidad y facilidad de uso han permitido la creación de títulos de gran éxito como *Hollow Knight*¹¹, *Cuphead*¹² y *Genshin Impact*¹³. 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*¹⁴ 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

¹¹<https://www.hollowknight.com>

¹²<https://cupheadgame.com>

¹³<https://genshin.hoyoverse.com/es>

¹⁴<https://docs.unity3d.com/2019.3/Documentation/Manual/VisualScripting.html>

sistema de renderizado *Universal Render Pipeline (URP)* permite optimizar los gráficos para múltiples plataformas, mientras que el *High Definition Render Pipeline (HDRP)* está diseñado para juegos con gráficos de alta calidad en PC y consolas de última generación.

Si se compara con otros motores como *Unreal Engine*, del cual se hablará más adelante, *Unity* destaca por su flexibilidad y menor consumo de recursos. Mientras que *Unreal Engine* es ampliamente reconocido por su calidad gráfica superior, *Unity* ofrece un entorno más ligero y optimizado, lo que lo convierte en una mejor opción 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*¹⁵, desarrollado por *Epic Games*¹⁶, es uno de los motores de videojuegos más potentes y utilizados en la industria de los videojuegos en títulos como la saga *Hellblade*¹⁷ o el éxito mundial *Fortnite*¹⁸ 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

¹⁵<https://www.unrealengine.com/es-ES>

¹⁶<https://www.epicgames.com/site/es-ES/home>

¹⁷https://thehellblade.fandom.com/wiki/Hellblade:_Senua%27s_Sacrifice

¹⁸<https://www.fortnite.com/?lang=es-ES>

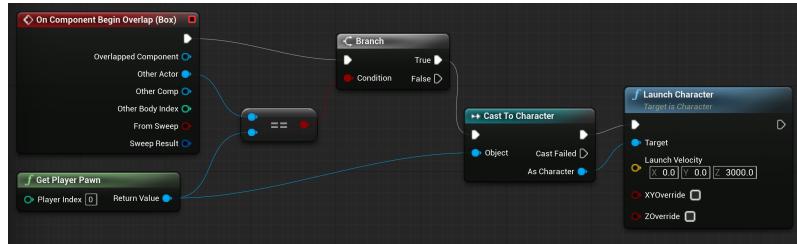


Figura 2.8: Blueprints en Unreal Engine

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.

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

Godot¹⁹, 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

¹⁹<https://godotengine.org/>

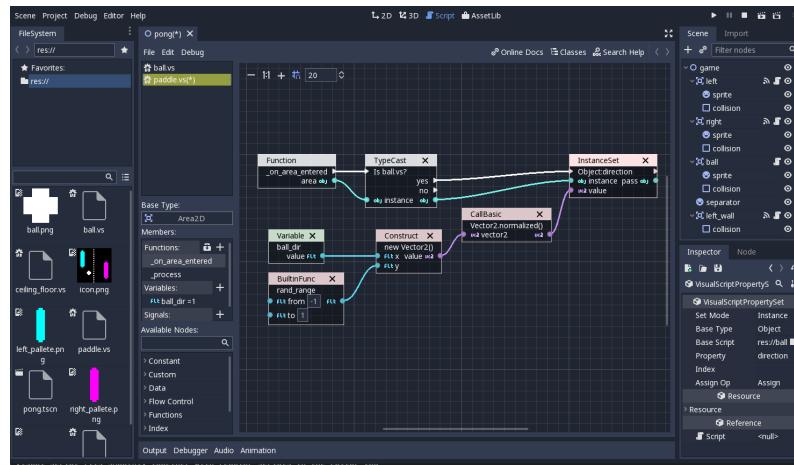


Figura 2.9: Sistema VisualScript en Godot

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 extensión²⁰.

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*²¹. Su adopción ha crecido considerablemente siendo usado por títulos de mayor proyección como *Marvel Snap*²². También es especialmente potente para aquellos

²⁰https://docs.godotengine.org/es/3.5/tutorials/scripting/visual_script/index.html

²¹https://brotato.wiki.spellsandguns.com/Brotato_Wiki

²²<https://www.marvelsnap.com/home>

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*²³ 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*²⁴, *Hotline Miami*²⁵ o *Katana ZERO*²⁶.

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

²³<https://gamemaker.io/en>

²⁴<https://undertale.com>

²⁵https://store.steampowered.com/app/219150/Hotline_Miami/

²⁶<https://katanazero.com>

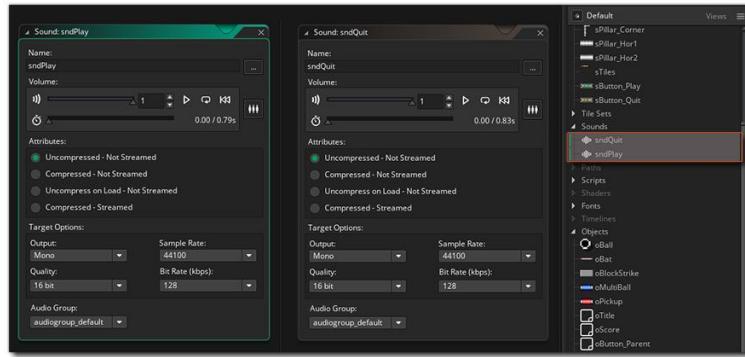


Figura 2.10: Drag and Drop en GameMaker

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

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

Capítulo 3

Diseño del framework

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

3.1. Descripción general del Framework

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 y hasta cuenta parte de la historia. Un enemigo puede obligarte a pensar una estrategia específica para vencerlo, o incluso tener una personalidad y comportamientos complejos que hacen que el enfrentamiento sea más significativo e inmersivo. En definitiva, diseñar a los antagonistas es una parte fundamental para crear un juego que enganche y deje huella.

El framework se ha diseñado con el objetivo de facilitar la creación de enemigos inteligentes en 2D por medio de comportamientos básicos. Esta orientado a diseñadores, estudiantes o desarrolladores que quieran diseñar comportamientos de enemigos. Con la particularidad de que no es necesario ningún conocimiento de programación.

3.1.1. Análisis de enemigos en videojuegos

El análisis de diversos documentos muestra que la mejor forma de hacer que un enemigo destaque y de un gran potencial al juego es que tenga unos comportamientos únicos. Cada enemigo se define mediante una combinación específica de estos. Determinando así lo que puede y no puede hacer y permiten diferenciarlo de otros tipos de enemigos. Estos comportamientos han ido evolucionando con el tiempo volviéndose cada vez más sofisticados. Con dicha sofisticación, ha aumentado la

complejidad del trabajo en el diseño. Para solucionarlo hemos propuesto una herramienta con comportamientos sencillos y precisos que ayudarán a reducir la carga de trabajo.

3.2. Composición

En este trabajo, se ha decidido entender como enemigo a cualquier entidad que pueda repercutir de forma negativa en el jugador, esto significa que no se limita el concepto de enemigo a figuras típicas, como monstruos o soldados hostiles, sino que se amplía su definición a toda entidad que suponga un riesgo, dificultad o amenaza para el progreso o el bienestar del jugador dentro del juego, como pueden ser pinchos o lava. Además separamos cada elemento en función de su comportamiento, implicando que elementos que clásicamente se consideran un único enemigo por aparecer juntos, como la tubería y la gota de ácido o la bala y el pistolero, se tratan aquí como entidades diferentes. Esta decisión se basa en que, desde el punto de vista de la lógica de comportamiento, actúan como elementos autónomos y con reglas distintas.

3.2.1. Identificación de comportamientos comunes

Para poder empezar con el desarrollo de la herramienta es necesario analizar distintos videojuegos existentes que tengan características comunes al objetivo de la herramienta, en este caso, que sean en dos dimensiones y plataformas. Para el análisis se estudiaron tres videojuegos. *Hollow Knight* y *Blasphemous* fueron los primeros en analizarse, se hizo un estudio del comportamiento individual de cada enemigo, separando jefes y enemigos comunes y centrándonos en estos últimos. El tercer juego fue *Bzzzt*, que sirvió para verificar que con los comportamientos ya identificados se podría realizar el comportamiento de los enemigos que aparecían con mayor frecuencia en él.

3.2.1.1. 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.

3.2.1.2. 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

¹<https://www.teamcherry.com.au/>

²<https://thegamekitchen.com/>

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

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.

3.2.1.3. 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.

3.2.1.4. Comportamientos relevantes detectados

El análisis de los juegos permitió la identificación de los siguientes patrones:

- *Patrulla*: Este comportamiento se caracteriza por caminar a un ritmo constante en el eje horizontal, cambiando de dirección al chocar. Se identificó en los tres juegos y en distintos enemigos dentro de cada uno de ellos (p.ej., Reptacillo del Hollow Knight (Figura 3.1)).
- *Torreta*: Enemigos que lanzan proyectiles de forma periódica (p.ej., Vengamoscas antes de moverse).
- *Embate lineal*: Consiste en una vez detectado al jugador modificar significativamente la velocidad, aumentándola para atacarle. (p.ej., Cáscara errante).
- *Aparición del suelo*: Son enemigos que parece que están saliendo del suelo, no atacan, solo salen y se esconden (p.ej., Goam o pinchos).
- *Persecución*: Son enemigos que persiguen al jugador hasta chocarse con él. La persecución se puede dar de distintas formas, únicamente en un eje o si el enemigo vuela, en forma de revolotéo donde no se acerca directamente si no que va deambulando.
- *Rotación*: Estos describen un movimiento circular alrededor de un punto (p.ej. Bolas circulares de Bzzzt).
- *Seguimiento de camino*: Enemigos que siguen un camino predeterminado y no lo abandonan independientemente del jugador (p.ej. Balas de Bzzzt).

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).

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



Figura 3.1: Imagen de escenario de *Hollow Knight* donde aparecen dos Reptacillos

Estas habilidades no siempre son compatibles entre sí, teniendo una tabla en la que se indicaran las relaciones entre ellas. Además no es necesario utilizar siempre todas las acciones de forma que a veces un enemigo podrá realizar un tipo de movimiento o usar un spawner de manera exclusiva, mientras que en otras podrá combinar varias habilidades según su diseño y complejidad. Esto permite adaptar las capacidades de los enemigos para diferentes situaciones en el juego.

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.

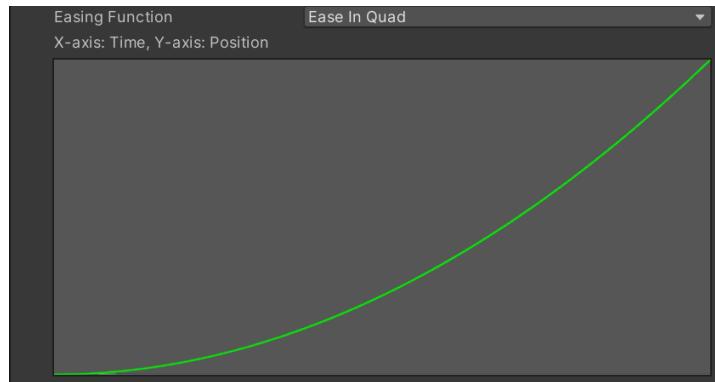


Figura 3.2: Easing Function que describe la variación de la posición de un objeto con el movimiento **Move To An Object**

- **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 (spline).

Todos los tipos de movimiento pueden configurarse para ejecutarse de forma acelerada. Para ello, se emplea una *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**, 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 3.2), 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 manera en la que influyen las *EasingFunction* en cada tipo de movimiento será explicada con más profundidad en el apartado de Implementación (Capítulo 4).

3.3.2. Spawner

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

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

juego. Un ejemplo serían las torretas, que crean balas.

Implementar un spawner requiere establecer ciertas reglas de generación de enemigos, tales como la frecuencia de aparición o el número máximo de enemigos generados.

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 desplazarse horizontalmente mientras activa un spawner para generar secuaces. Esta sinergia entre actuadores (movimientos y spawners) enriquece la complejidad del comportamiento y abre un abanico de posibilidades creativas para los diseñadores.

No obstante, es crucial reconocer que no todas las combinaciones de acciones resultan coherentes o deseables desde la perspectiva del diseño del juego. Por ejemplo, intentar realizar un movimiento vertical mientras se está definido un movimiento circular podría generar comportamientos visuales o de jugabilidad no intencionados.

Para clarificar estas interdependencias y asegurar la coherencia en la configuración de los enemigos, se ha elaborado la tabla **tabla 3.1** de compatibilidad. Esta tabla actuará como una guía visual e intuitiva, indicando qué combinaciones de movimientos y la activación de spawners son factibles y recomendadas, permitiendo a los diseñadores construir enemigos con comportamientos complejos pero lógicos y bien definidos.

Tabla 3.1: Matriz de compatibilidad de movimientos

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

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 y emisores disponibles en la herramienta.

- **Área:** Detecta si un objeto entra o sale en una zona de detección.

- **Collision:** Detecta colisiones físicas con otros objetos.
- **Distance:** Detecta si un objeto está dentro o fuera de una distancia específica.
- **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. Estado

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

3.7. Máquina de estados finita

La Máquina de Estados Finita (FSM, Finite State Machine) es el núcleo de la lógica que define el comportamiento de los enemigos en nuestro diseño. Cada enemigo tiene su propia FSM, configurada específicamente para representar sus patrones de

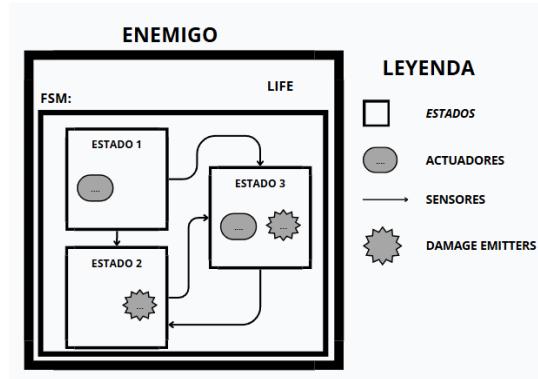


Figura 3.3: Enemigo General

acciones, reacciones y relaciones en el juego. La FSM organiza el comportamiento de los enemigos mediante estados y transiciones: Los estados, como ya se ha explicado, agrupan las acciones que el enemigo puede realizar en un momento dado. Las transiciones permiten cambiar de un estado a otro y son activadas por sensores y emisores.

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

3.8. Ejemplos de uso

Con esta separación de actuadores, sensores y daño la creación de enemigos se vuelve mucho más sencilla. A continuación se presentan unos ejemplos construidos mediante la combinación de los componentes de la herramienta sin la necesidad de escribir código, estos ejemplos están incluidos en la propia herramienta.

Se mencionarán algunos componentes que aún no han sido introducidos, estos serán explicados en el Capítulo 4.

Todos los enemigos mencionados incluyen los siguientes componentes:

- Un **Animator Controller** propio, configurado de manera que las animaciones se correspondan con la lógica de comportamiento de cada enemigo.
- Un componente de tipo **FSM** (Finite State Machine).
- Al menos un componente de tipo **State**. Siempre que se mencione que un enemigo incorpora un **Actuator**, **Sensor** o **DamageEmitter**, estos estarán referenciados dentro de una clase **State** específica. En caso de no especificarse ninguna, se asumirá que el enemigo dispone de un único estado.

3.8.1. Bouncing Bunny

En la Figura 3.4 se puede ver un enemigo sencillo utilizado habitualmente al comienzo de los juegos de plataformas en dos dimensiones. El enemigo se desplaza

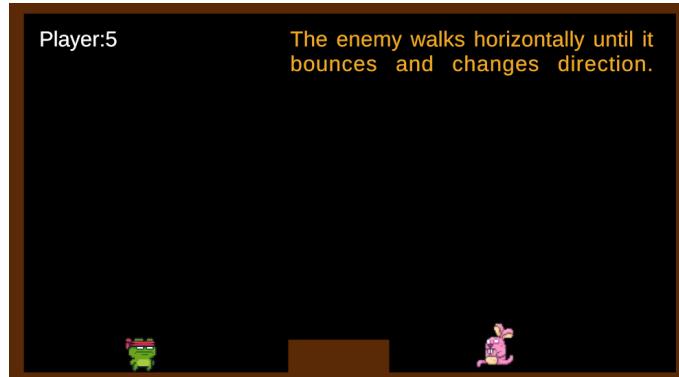


Figura 3.4: Escena de ejemplo donde observamos al enemigo en cuestión momentos antes de alcanzar la pared.

de izquierda a derecha describiendo un movimiento horizontal, hasta colisionar con una pared, momento en el cual cambia de dirección.

Para construir este comportamiento, configuramos un único estado el cual tiene un **Horizontal Actuator** configurado para que rebote en caso de colisionar con las paredes y con una velocidad constante y un **Damage Emitter** configurado para realizar daño instantáneo, concretamente una única unidad.

También necesitaremos añadir un componente **Life** configurado para que el enemigo tenga un punto de vida.

3.8.2. Spinning Rocks

Con respecto al enemigo mostrado en la Figura 3.5, este realiza un movimiento circular alrededor de un punto central. Aunque en apariencia pueda parecer una única entidad, en realidad cada bola actúa como un enemigo independiente que ejecuta su propio movimiento circular.

Este tipo de enemigo no puede ser detenido ni eliminado en ningún momento, por lo que no tendrá componente relacionado con la vida, lo que incrementa significativamente su nivel de peligrosidad. Su presencia constante y la imposibilidad de neutralizarlo lo convierten en una amenaza persistente dentro del escenario de juego.

Para construirlo necesitaremos crear tres objetos con los sprites mostrados en la figura con un **Circular Actuator** con los parámetros necesarios para que produzca un giro completo de 360° . También necesitará un **Damage Emitter** configurado para realizar daño instantáneo, concretamente una única unidad.

Cada objeto necesitará asignar un punto sobre el que rotar, por lo que se necesitará crear otro objeto que sirva como centro de la rotación.

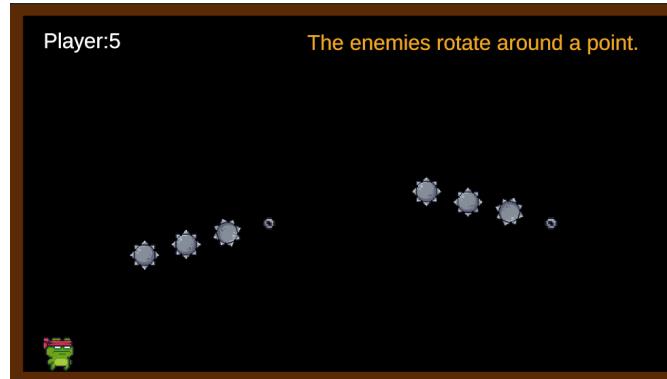


Figura 3.5: Escena de ejemplo donde se aprecian dos enemigos realizando un movimiento circular cada uno en un sentido diferente

3.8.3. Trunk Torret

Con respecto al enemigo mostrado en la Figura 3.6, este permanece completamente estático durante toda su existencia en el escenario. A pesar de su falta de movilidad, representa una amenaza constante al disparar proyectiles de manera periódica cada dos segundos, lo que obliga al jugador a mantenerse en movimiento para esquivarlos.

Este enemigo es completamente invulnerable a cualquier tipo de ataque y tampoco hace daño directamente, por lo que no cuenta con componentes asociados a vida o daño, tanto recibido como realizado. Esta característica aumenta su peligrosidad, ya que su presencia no puede ser eliminada y el jugador debe enfrentarse a él únicamente mediante la evasión.

Para implementarlo, se le asignará un componente de tipo **Spawner Actuator**, configurado para emitir un proyectil cada dos segundos que aparece desde un punto de su boca.

3.8.3.1. Bullet

El proyectil generado por la *Trunk Turret* se desplaza horizontalmente hacia la izquierda con un movimiento lineal. Al entrar en contacto con el jugador, produce daño instantáneo. En caso de colisionar con el entorno o con el propio jugador, el proyectil se destruirá automáticamente.

Para construir este comportamiento, se utilizará un objeto , al que se le asignará un **Horizontal Actuator** configurado para el movimiento en dirección izquierda y que reaccione al contacto destruyéndose, así como un **Damage Emitter** capaz de infligir una unidad de daño al contacto.

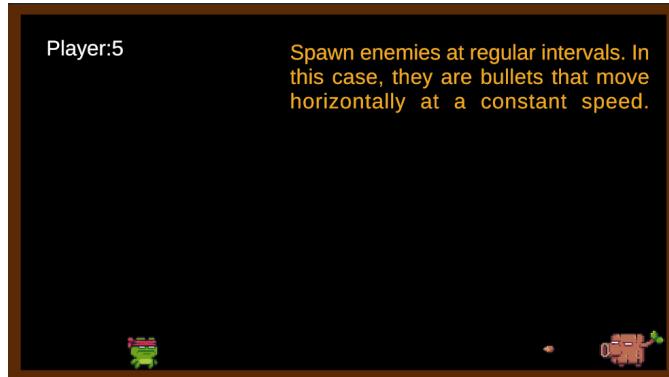


Figura 3.6: Escena de ejemplo donde se aprecia una *TrunkTorret* disparando.

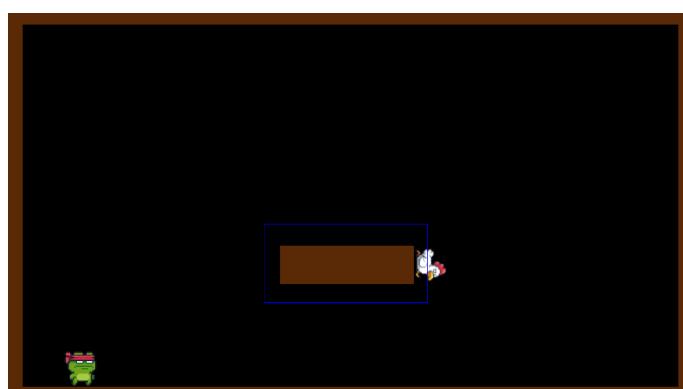


Figura 3.7: Escena de ejemplo donde se aprecia una *Spline Chicken* siguiendo su trayectoria.

3.8.4. Spline Chicken

Con respecto al enemigo mostrado en la Figura 3.7, esta gallina realiza un movimiento continuo siguiendo una trayectoria curva predefinida (spline) que rodea una plataforma. Este desplazamiento suave y cerrado permite que patrulle una zona concreta, dificultando el paso del jugador y añadiendo presión durante momentos clave del recorrido.

Aunque su apariencia pueda resultar cómica o inofensiva, este enemigo representa una amenaza real, ya que infinge daño al contacto. Además, no puede ser eliminado ni detenido, lo que lo convierte en un obstáculo constante que obliga al jugador a calcular cuidadosamente el momento en el que atravesar su recorrido.

Para su implementación, se instanciará un objeto con el sprite correspondiente, al que se le añadirá un componente de tipo **Spline Follower**, configurado para recorrer una trayectoria cerrada en bucle. Además, se incluirá un **Damage Emitter** que provocará la eliminación instantánea del jugador.

Tendrá un componente **Life** con cinco puntos de salud.

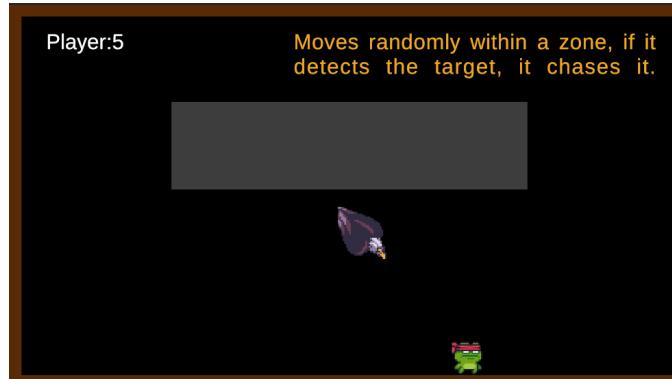


Figura 3.8: Escena de ejemplo donde se aprecia a la *Skywatch Eagle* atacando al jugador.

3.8.5. Skywatch Eagle

Con respecto al enemigo mostrado en la Figura 3.8, esta águila deambula libremente por una zona delimitada del escenario, describiendo un patrón de patrullaje aéreo. Su comportamiento cambia de forma agresiva en cuanto detecta la presencia del jugador dentro de su campo de visión, momento en el cual se lanza en picado a gran velocidad con la intención de impactarlo.

Este enemigo combina movilidad impredecible y comportamiento reactivo, lo que lo convierte en una amenaza considerable. Su capacidad para alterar su trayectoria bruscamente en función de la posición del jugador obliga a mantenerse en constante movimiento y a anticipar su patrón de ataque.

Para construir este comportamiento, se utilizarán un componentes de tipo **State** extra:

- Uno de ellos será el comportamiento de patrulla, estado en el que el enemigo se desplaza aleatoriamente por una zona aérea mediante un **Move To A Point Actuator** configurado para moverse a través de puntos generados aleatoriamente en la zona gris de la Figura 3.8. En este estado también se encuentra la transición al estado de ataque, a través de un **Distance Sensor** que mide la distancia entre ambos.
- En este estado, el águila se mueve rápidamente hacia la posición del jugador usando un **Move To Object Actuator**. Cuando el daño es realizado, vuelve a su estado inicial a través de un **Collision Sensor** configurado para activarse al contacto con el jugador.

Adicionalmente, contará con un **Damage Emitter** que infinge una unidad de daño si colisiona con el jugador durante el picado y exclusivamente durante el picado.

Capítulo 4

Implementación

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

La implementación de la herramienta puede ser encontrada en el siguiente [enlace](#).

4.1. Tecnología utilizada

Este proyecto ha sido desarrollado íntegramente en Unity, mencionado en el Capítulo 2 de este trabajo, concretamente la versión 2022.3.18f1(LTS) del motor. Por consiguiente, no podemos garantizar su correcto funcionamiento en versiones previas. No obstante, la previsión de la herramienta es ofrecer el correcto funcionamiento en versiones posteriores, salvo modificaciones significativas en la API fundamental de Unity.

El motivo por el que se escoge Unity sobre los demás motores de videojuegos es esa accesibilidad que ha sido mencionada anteriormente que hace que resulte sencillo de utilizar por cualquier persona.

Se hará uso de otros elementos de Unity para llevar a cabo esta herramienta como el motor de físicas 2D o las herramientas que tiene el motor para ayudar al usuario a debuggear como puede ser Gizmos. Unity funciona mediante una arquitectura por componentes, lo que es muy útil para que el programa sea modular, que pueda ser dividido en piezas más pequeñas y que estas piezas sean independientes.

La herramienta desarrollada en este proyecto tiene como objetivo facilitar la creación y configuración de enemigos en un entorno 2D dentro de Unity. Está diseñada como un plugin modular que permite a diseñadores y desarrolladores configurar

enemigos sin necesidad de escribir código, mediante un sistema visual e intuitivo basado en prefabs y componentes reutilizables.

En el núcleo de esta arquitectura se encuentra un sistema basado en una Máquina de Estados Finita (FSM). Cada enemigo se define mediante una FSM que gestiona su comportamiento, transiciones entre estados y reacciones ante el entorno. Los estados están compuestos por actuadores (actuators), que ejecutan acciones (como movimiento, ataque o generación de enemigos), y sensores, que detectan condiciones o eventos del entorno y disparan transiciones entre estados.

Los elementos que conforman esta arquitectura se organizan en varios niveles:

- Estados: Definen el comportamiento que el enemigo adopta en un momento determinado.
- Sensores: Detectan eventos del entorno, como la presencia del jugador o colisiones.
- Actuadores: Ejecutan acciones concretas, como moverse, disparar o generar nuevos enemigos.
- Emisores de daño: Permiten que los enemigos infljan daño a otras entidades.

La herramienta incluye también una serie de prefabs preconfigurados que facilitan el uso del sistema y permiten una rápida iteración por parte del diseñador. Estos prefabs agrupan configuraciones comunes de enemigos con distintos comportamientos, reduciendo el tiempo de prototipado y aumentando la reutilización del código.

En resumen, lo que se ha creado es una herramienta extensible compuesta por scripts modulares, basada en la arquitectura de componentes de Unity, que permite crear comportamientos complejos mediante una configuración visual, facilitando así su adopción por parte de usuarios con conocimientos limitados de programación.

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

4.2. Actuator

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

4.2.1. 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.

4.2.1.1. HorizontalActuator

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

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 que 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.

4.2.1.2. 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.

4.2.1.3. 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° 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.

4.2.1.4. 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.

4.2.1.5. 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.

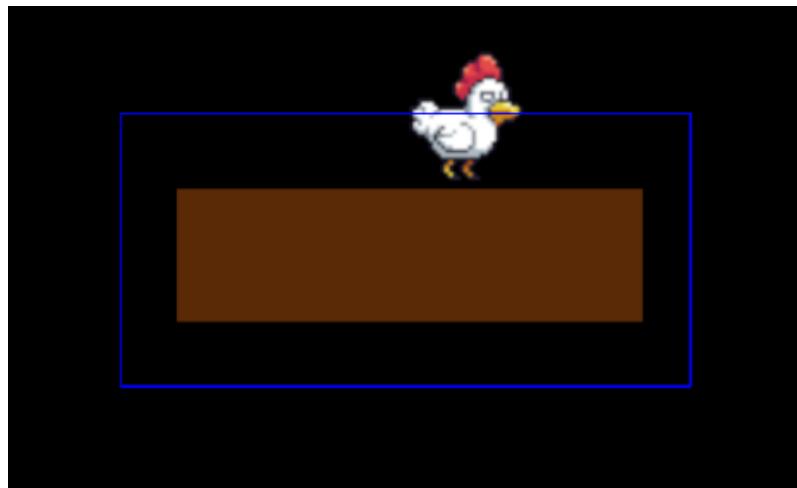


Figura 4.1: Uso de Spline para definir el camino de un enemigo.

- (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 mediante una curva spline (Figura 4.1), la cual será seguida por el objeto durante su movimiento. Este componente permite establecer una serie de puntos de control (control points) conectados por curvas suaves, generando así un recorrido continuo. Cada punto puede tener handles (manejadores) para controlar la tangente de entrada y salida, permitiendo ajustar la forma de la curva con precisión.
Unity proporciona herramientas visuales para editar la forma de la curva directamente en el editor, y el componente *SplineContainer* permite acceder a su representación en tiempo real, interpolar posiciones a lo largo del recorrido, o determinar la orientación del objeto mientras lo recorre.
- (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.

4.2.1.6. 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 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

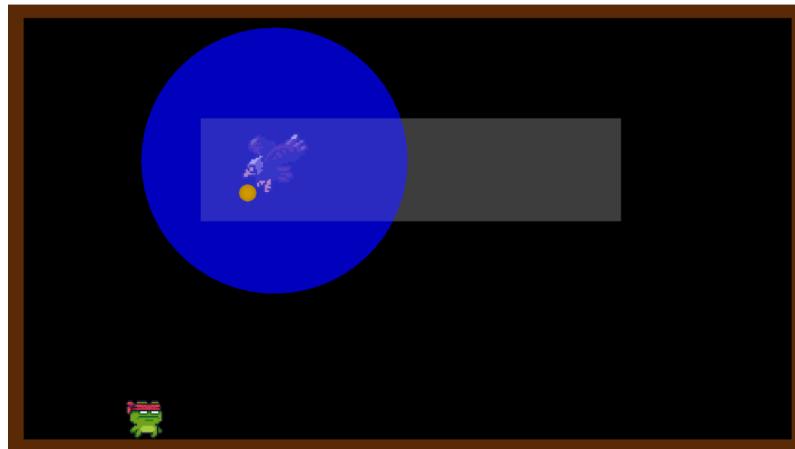


Figura 4.2: Modo Random Area, se aprecia una zona azul (DistanceSensor), un rectángulo gris (zona donde aparecen nuevos puntos) y por último un punto amarillo (waypoint actual).

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.

4.2.1.7. 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.

4.2.2. SpawnerActuator

La clase `SpawnerActuator` se usa para poder generar nuevos enemigos, pudiendo generar infinitos enemigos o un número definido de ellos cada X 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 X 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.

4.3. 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 activarse 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.

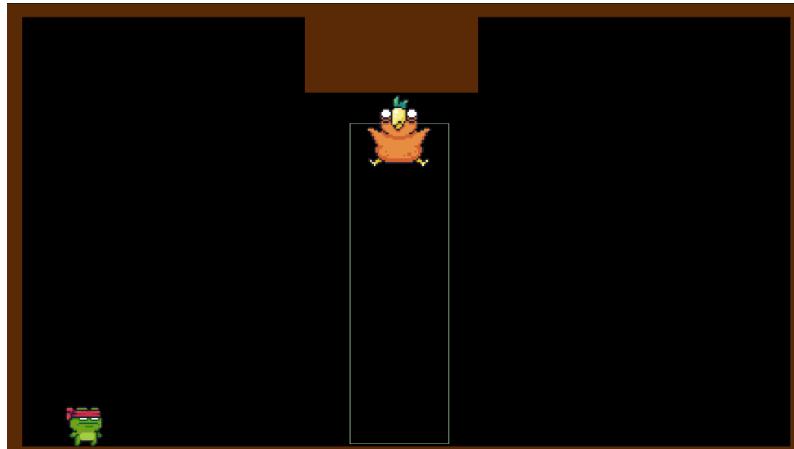


Figura 4.3: AreaSensor utilizado para enemigo que cae al detectar al jugador.

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

4.3.1. AreaSensor

AreaSensor representa un tipo de sensor espacial que detecta la presencia de un objeto objetivo dentro de una zona delimitada (Figura 4.3). 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**.

4.3.2. 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.

Parámetros de configuración

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

4.3.3. 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.

Valores de *TypeOfDistance*

- *Magnitude*

Configuración utilizada cuando se quiere medir la distancia como magnitud. La representación gráfica de esta forma de medir la distancia es un círculo, como se muestra en la Figura 4.4.

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.



Figura 4.4: Medición de distancia a través de la magnitud.

- *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 4.5 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.

4.3.4. 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.

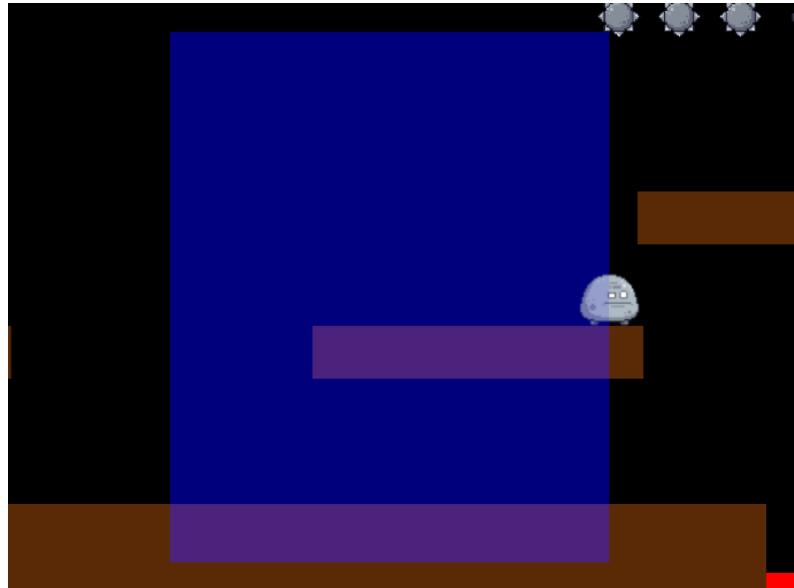


Figura 4.5: Medición de distancia en el eje X en su lado negativo.

4.4. Damage

En esta sección se abordará el como se gestiona todo lo relacionado con el daño a través de, principalmente, dos componentes: `DamageEmitter` y `DamageSensor`.

4.4.1. 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.

rios.

Valores de *DamageType*

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

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 X 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.

4.4.2. 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`. 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.

4.5. Máquina de Estados Finita

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

Valores de configuración

- (State) **Initial State**: Estado inicial de la Máquina de Estados Finita.

4.5.1. Transition

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

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

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.

4.5.2. 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 DamageEmitter 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.

En la Figura 4.6 vemos como se distribuyen los parámetros en el inspector.

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.

4.6. Animation Manager

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

Parámetros de configuración

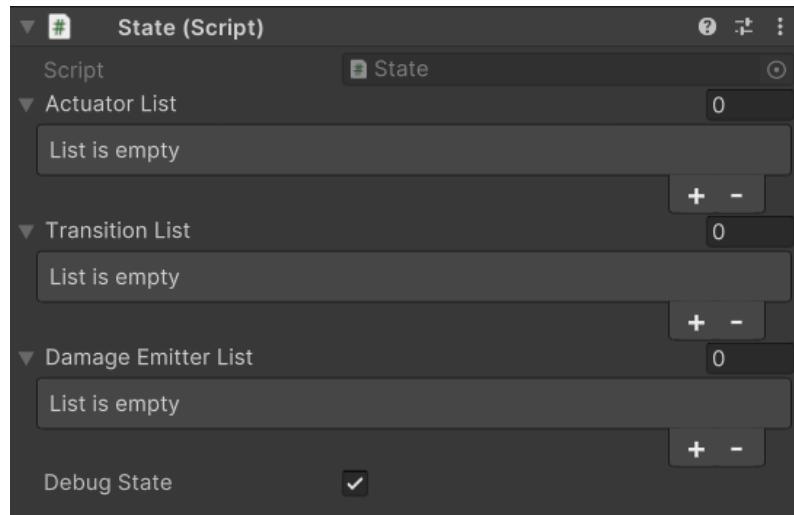


Figura 4.6: Distribución del inspector de clase **State**.

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

Durante la ejecución del juego, se monitoriza constantemente la velocidad del objeto mediante su componente **Rigidbody2D**. Esta información se utiliza para actualizar tres parámetros del **AnimatorManager**: *XSpeed*, *YSpeed* y *RotationSpeed*. Estos parámetros permiten que el sistema de animaciones adapte en tiempo real la apariencia del personaje según su movimiento: por ejemplo, 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, pasa de moverse a la izquierda a hacerlo a la derecha), su sprite puede rotarse automáticamente para mantener la coherencia visual. Este comportamiento solo se aplica si se han activado previamente unos booleanos de configuración que permiten dicha rotación.

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

Por otro lado, se han definido varios parámetros y funciones que permiten controlar su comportamiento:

- El parámetro *Follow* (de tipo bool) indica si el enemigo debe seguir a otro objeto, normalmente el jugador.
- La función *ChangeState* permite modificar el estado actual de la máquina de estados finita (FSM) del enemigo. Este cambio puede provocar una transición de animación o modificar su comportamiento.
- El trigger *Spawn* se activa al instanciar el enemigo en la escena. Permite reproducir una animación de aparición o ejecutar lógica específica asociada a su entrada en el juego.

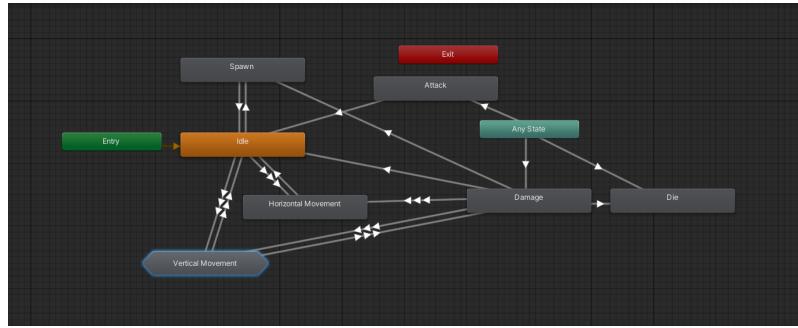


Figura 4.7: AnimatorController con sus estados y transiciones.

Parámetros requeridos en el Animator Para sincronizar el comportamiento del enemigo con sus animaciones, se ha configurado un **Animator Controller** que utiliza los siguientes parámetros:

- **Triggers:** *Die, Damage, Spawn, ChangeState*. Se activan para lanzar animaciones específicas cuando ocurren eventos clave (por ejemplo, recibir daño o morir).
- **Bools:** *Left, Right, Up, Down, Follow, Rotating*. Indican direcciones, estados o comportamientos que influyen en las transiciones de animación.
- **FLOATS:** *XSpeed, YSpeed, RotationSpeed*. Permiten animar al enemigo en función de su velocidad de movimiento o rotación.

Para centralizar la gestión de las animaciones, se ha creado un controlador personalizado (Figura 4.7), del tipo **Animator Controller**. Este actúa como una máquina de estados de animación, donde cada estado representa una animación (como caminar, aparecer, recibir daño, etc.), y las transiciones entre estados se controlan mediante los parámetros antes mencionados.

Este enfoque facilita la integración entre la lógica del enemigo (programación) y su representación visual (animación).

4.7. Jugador

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

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

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

4.7.1. PlayerMovement

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

Valores de configuración

- (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.

4.7.2. PlayerCollisionDetection

Este script se encarga de detectar las colisiones del jugador. Para ello, se ajustan tres cajas de colisión (Figura 4.8): una en cada lado y otra para detectar el contacto con el suelo. Las cajas no detectan realmente colisiones, sino que comprueban si estas se superponen con alguna entidad de las capas especificadas con el método `Physics2D.OverlapBox()`.

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

Valores de configuración

- (bool) **Debug Boxes**: En caso de que esta variable sea true, las cajas definidas por los campos que serán presentados a continuación serán representadas en pantalla con color rojo.
- (LayerMask) **Detection Layers**: Capas que serán tomadas en cuenta para detectar si el jugador está en el suelo o en contacto con una pared cuando está en el aire. En este caso se querrá especificar la capa que alberga a los objetos estáticos que conforman el mundo, por ejemplo *World*.
- (Vector2) **Bottom/Right/Left Size**: Tamaño de las cajas.

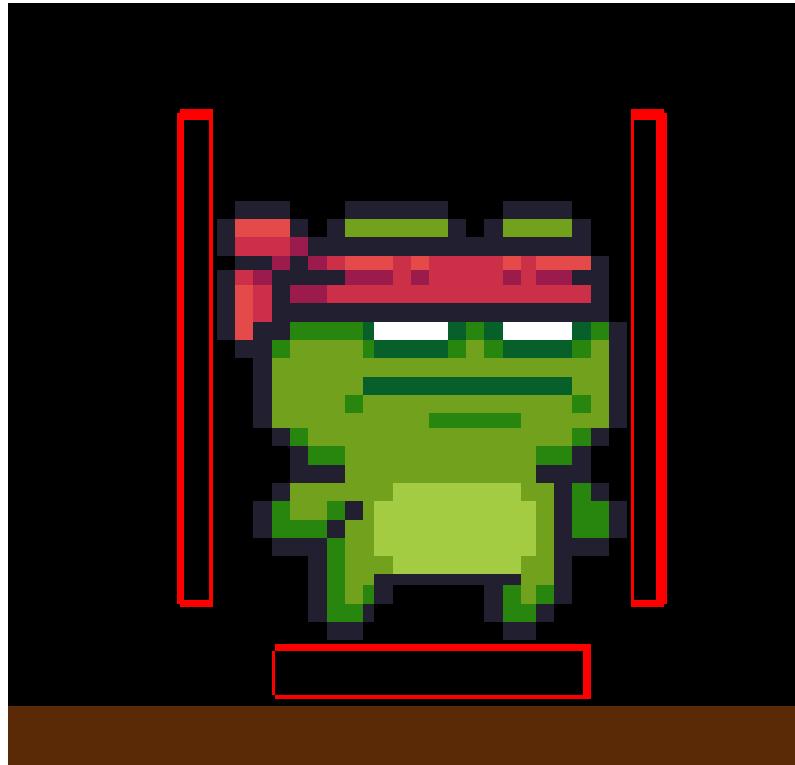


Figura 4.8: Representación de las cajas de detección de colisiones del jugador

- (**Vector2**) **Bottom/Right/Left Offsets**: Valores usados para reposicionar las cajas para que estén en el lugar considerado por el usuario, idealmente en los bordes del objeto.

4.7.3. 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.

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

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

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.

4.7.5. 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.

4.8. Distribución del Framework

4.9. Conclusiones

Capítulo 5

Evaluacion Con Usuarios

Tras las fases de desarrollo de la herramienta se llevarán a cabo las pruebas con usuarios que nos mostrarán los datos de un primer contacto con usuarios.

Para eso hemos creado un guión de evaluación que fijará unos objetivos y unas preguntas de investigación, así como marcar las pautas para la realización de las pruebas.

Después de la realización de las pruebas se procederá a hacer una análisis que muestre las posibles brechas encontradas, basándonos en la experiencia de los usuarios.

5.1. Objetivos y preguntas de investigación

Los objetivos y preguntas de investigación tienen como finalidad tratar de comprender la experiencia de los usuarios reales al interactuar con nuestra herramienta por primera vez. Buscamos identificar cualquier punto donde puedan surgir dificultades, confusión o áreas donde la herramienta podría ser más efectiva. Para guiar esta exploración, hemos definido los siguientes objetivos y preguntas de investigación.

5.1.1. ¿Resulta intuitiva y clara la herramienta para el usuario?

Lo más importante de la herramienta es que sea clara para los usuarios que van a usarla, ya que sin eso, aunque la herramienta tenga un buen funcionamiento, la utilidad de esta sigue siendo nula.

Nuestro primer objetivo se centra en evaluar si los usuarios perciben la interfaz y los flujos de trabajo de manera natural y sin ambigüedades.

Preguntas de investigación relacionadas con el objetivo:

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

Esta pregunta de investigación pretende comprobar, si el usuario es capaz de comprender el funcionamiento base de los enemigos, es decir, si comprende como utilizar la máquina de estados o como se añaden estados nuevos, transiciones, Damage Emitters y actuadores.

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

Aquí, nuestro interés reside en la claridad con la que se presentan y se entienden los distintos actuadores disponibles. Buscamos identificar cualquier punto de confusión en su aplicación, analizando si la funcionalidad y las explicaciones proporcionadas para cada uno son suficientes para su correcta utilización.

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

La pregunta de investigación pretende conocer si el diseñador es capaz de saber en todo momento en que estado está el enemigo y por tanto que comportamiento cabría esperar.

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

Buscamos evaluar si la documentación adjunta proporciona la información necesaria para que el usuario pueda configurar correctamente los enemigos, aprovechando al máximo las funcionalidades implementadas.

5.1.2. ¿El sistema de creación de enemigos demuestra ser funcional?

Este objetivo busca validar la operatividad y la utilidad de la herramienta.

Preguntas de investigación relacionadas con el objetivo:

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

Esta pregunta se enfoca en la eficiencia que aporta la herramienta al flujo de trabajo del diseñador de videojuegos. ¿Facilita la creación de enemigos?

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

En este punto, la atención se centra en la fidelidad con la que la lógica definida en la herramienta se manifiesta en el entorno de juego. ¿Actúan los enemigos según lo previsto? ¿Se activan sus acciones en las circunstancias correctas? Identificar cualquier desviación entre el diseño y la ejecución es crucial.

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

Queremos evaluar si es posible modificar o ampliar las capacidades de los enemigos creados mediante la adición de nuevos comportamientos o la alteración de parámetros existentes, sin generar 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 máquinas 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.

Guion de la sesión:

"Buenos días a todos y muchísimas gracias por la colaboración en estas pruebas de evaluación. Como ya sabéis, vais a probar una herramienta que sirve para diseñar enemigos en dos dimensiones en Unity. En los ordenadores tenéis ya un proyecto con la herramienta abierta, así como un manual de usuario detallado en el escritorio. Por favor leed el manual y haced los ejemplos prácticos que se describen dentro de él. Una vez familiarizados con la herramienta y habiendo terminado los ejemplos prácticos, usadla creando el enemigo que se os ocurra. Por último se os abrirá un breve cuestionario, para su cumplimentación y terminaremos con una breve entrevista."

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. Diseño de la evaluación

La evaluación ha sido diseñada utilizando una metodología mixta, combinando técnicas cuantitativas y cualitativas para obtener una visión completa del desempeño de la herramienta.

5.2.6.1. Observación

Durante la realización de las tareas, el investigador observará el comportamiento de los participantes de forma no intrusiva. 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 detectados únicamente a través del cuestionario o la entrevista.

5.2.6.2. Cuestionario

Una vez completadas las tareas anteriores, para conocer la opinión de los usuarios sobre la facilidad de uso del sistema que hemos desarrollado, decidimos emplear el cuestionario SUS (System Usability Scale) de Brooke (1996).

El SUS consta de diez preguntas en forma de afirmación. Los participantes deben indicar su grado de acuerdo o desacuerdo por medio de una escala Likert de cinco puntos, donde 1 representa totalmente en desacuerdo y 5 representa totalmente de acuerdo. Los ítems que componen el cuestionario son los 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.

5.2.6.3. Puesta en común

Tras la finalización del cuestionario SUS, se procederá a realizar una entrevista semiestructurada con los participantes. Esta dinámica de cierre tiene como objetivo recoger información cualitativa que complementa los datos obtenidos mediante la observación y el cuestionario. Se busca comprender en profundidad la experiencia de uso, las percepciones del usuario y detectar posibles áreas de mejora no contempladas anteriormente.

La puesta en común se guiará por un conjunto de preguntas abiertas, que permitirán a los usuarios expresar con libertad sus impresiones y sugerencias. El investigador podrá adaptar o profundizar en algunas preguntas en función de las respuestas obtenidas, permitiendo así una conversación fluida y natural.

A continuación, se muestran las preguntas principales que se utilizarán como guía:

- ¿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?

5.2.7. Preparación de la Ejecución

Antes de llevar a cabo las pruebas con usuarios, se realizarán las siguientes acciones de preparación:

- Preparación de la herramienta (abierta y funcional).
- Dejar abiertos en los ordenadores que se usen para las pruebas la herramienta y el manual.
- Comprobar el uso de las grabadoras de voz para la entrevista.
- Tener el guión al alcance para revisar los puntos durante la ejecución.

Capítulo 6

Conclusiones y Trabajo Futuro

Conclusiones finales

Esta herramienta ha sido enfocada en todo momento al diseño de enemigos en 2D. La propuesta surge como respuesta a la necesidad de dotar a diseñadores de una solución práctica y accesible que les permita configurar comportamientos complejos sin requerir conocimientos avanzados en programación.

A través de una arquitectura basada en máquinas de estado finitas, y complementada con un conjunto de sensores, actuadores y emisores, se ha logrado una solución flexible, accesible y con capacidad de escalabilidad.

Durante el desarrollo, se han cubierto aspectos esenciales como el movimiento, el control de daño, la generación de entidades, el tratamiento de colisiones y la vinculación con animaciones. Además, se ha incluido un manual orientado a usuarios no técnicos, lo que refuerza el propósito de accesibilidad de la herramienta.

Trabajo Futuro

De cara a futuras iteraciones, se han identificado distintas áreas de expansión que permitirán dar a la herramienta más funcionalidades y por tanto versatilidad:

- **Mantenimiento evolutivo:** Será necesario realizar revisiones periódicas que aseguren la compatibilidad con futuras versiones del motor Unity, en especial ante posibles modificaciones en su API base.
- **Implementación de sensores de sonido:** Se planteó incorporar un sensor de sonido capaz de detectar estímulos auditivos, lo que ampliaría las posibilidades de interacción de los enemigos dentro del juego.
- **Emisores de sonido:** Complementarios a los sensores acústicos, estos componentes permitirían emitir señales auditivas que activen comportamientos en otras entidades cercanas, posibilitando nuevas dinámicas de juego.

- **Comportamientos de Steering y Avoidance:** Se propone integrar técnicas de evasión y navegación consciente del entorno, que otorguen a los enemigos capacidad para evitar obstáculos y desplazarse de manera más realista.
- **Memoria de interacciones previas:** Una funcionalidad avanzada consistiría en permitir a los enemigos almacenar información sobre acciones pasadas del jugador, adaptando sus respuestas en función de patrones reconocidos.
- **Coordinación entre enemigos:** Otra línea de mejora es permitir la comunicación y cooperación entre múltiples enemigos, generando patrones de comportamiento colectivo, tales como persecuciones en grupo o ataques sincronizados.
- **Desarrollo de una interfaz gráfica externa:** A largo plazo, se sugiere diseñar una interfaz visual independiente que facilite la selección y configuración de componentes mediante una ventana específica, optimizando así la experiencia del usuario.

Estas líneas de trabajo reflejan el potencial del sistema como herramienta activa dentro del flujo de desarrollo de videojuegos 2D. La estructura modular y la orientación hacia la escalabilidad permitirán incorporar estas mejoras de forma progresiva, garantizando su utilidad en proyectos reales de diseño de enemigos complejos.

Introduction

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

6.1. Motivation

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

6.2. Objectives

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

6.3. Work Plan

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

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

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

Conclusions and Future Work

Final Conclusions

This tool has been consistently focused on the design of 2D enemies. The proposal arises as a response to the need to provide designers with a practical and accessible solution that allows them to configure complex behaviors without requiring advanced programming knowledge.

Through an architecture based on finite state machines, and complemented by a set of sensors, actuators, and emitters, a flexible, accessible, and scalable solution has been achieved.

During development, essential aspects such as movement, damage control, entity generation, collision handling, and animation linking have been covered. In addition, a manual aimed at non-technical users has been included, which reinforces the accessibility purpose of the tool.

Future Work

Looking towards future iterations, several areas of expansion have been identified that will allow the tool to have more functionalities and therefore versatility:

- **Evolutionary Maintenance:** Periodic reviews will be necessary to ensure compatibility with future versions of the Unity engine, especially in the face of possible modifications to its base API.
- **Implementation of Sound Sensors:** Incorporating a sound sensor capable of detecting auditory stimuli was proposed, which would expand the interaction possibilities of enemies within the game.
- **Sound Emitters:** Complementary to acoustic sensors, these components would allow the emission of auditory signals that trigger behaviors in other nearby entities, enabling new game dynamics.
- **Steering and Avoidance Behaviors:** Integrating evasion techniques and conscious environment navigation is proposed, which would give enemies the ability to avoid obstacles and move more realistically.

- **Memory of Previous Interactions:** An advanced functionality would consist of allowing enemies to store information about past player actions, adapting their responses based on recognized patterns.
- **Coordination Between Enemies:** Another line of improvement is to allow communication and cooperation between multiple enemies, generating collective behavior patterns, such as group chases or synchronized attacks.
- **Development of an External Graphical Interface:** In the long term, it is suggested to design an independent visual interface that facilitates the selection and configuration of components through a specific window, thus optimizing the user experience.

These lines of work reflect the potential of the system as an active tool within the 2D video game development workflow. The modular structure and the orientation towards scalability will allow these improvements to be incorporated progressively, guaranteeing its usefulness in real projects for the design of complex enemies.

Capítulo **7**

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.

7.1. Contribuciones de Cristina Mora Velasco

7.1.1. 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.

7.1.2. Aportaciones

7.1.2.1. 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.

7.1.2.2. 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.

7.1.2.3. 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.

7.1.2.4. 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.

7.2. Contribuciones de Francisco Miguel Galván Muñoz

7.2.1. 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.

7.2.2. Aportaciones

7.2.2.1. 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.

7.2.2.2. 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.

7.2.2.3. 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.

7.2.2.4. 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

ARTS, E. Spore. n.d. Disponible en <https://www.ea.com/es-es/games/spore> (último acceso, January, 2025).

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).

FAN, G. Rules of the game: Five techniques from quite inventive designers. <https://youtu.be/d8QAVGeEj-U?t=1676>, 2016. Video en YouTube, último acceso: mayo de 2025.

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, January, 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).

- MILLINGTON, I. y FUNGE, J. *Artificial Intelligence for Games*. ???? 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: mayo de 2025.
- WIKIPEDIA CONTRIBUTORS. Bioshock. <https://es.wikipedia.org/wiki/BioShock>, ????
- 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.

