Generador de comportamientos de enemigos para videojuegos 2D
Generator of enemy behaviours for 2D videogames

Trabajo de Fin de Grado
Curso 2019–2020

Autor
Daniel Quintero Bernal

Director
Guillermo Jiménez Díaz

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

Generador de comportamientos de enemigos para videojuegos 2D
Generator of enemy behaviours for 2D videogames

Trabajo de Fin de Grado en Desarrollo de Videojuegos
Departamento de Informática

Autor
Daniel Quintero Bernal

Director
Guillermo Jiménez Díaz

Convocatoria: *Junio 2020*

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

26 de junio de 2020

Agradecimientos

En primer lugar, dar las gracias a mis padres, a mi madrina y mis abuelas, por su infinito apoyo, paciencia y cariño. Por darme la oportunidad de hacer esta carrera, por tener fe en mi, aguantar mi cabezonería y recordarme que hay tiempo para todo.

A Laura, por todo lo bueno que me aportas, por haber sido mi compañera durante esta aventura. Gracias por haber estado a mi lado siempre que lo he necesitado, por haberme apoyado en las buenas y en las malas.

A mis amigos, por lo mucho que me han apoyado en esta etapa, por todas las noches en las que nos juntamos, para charlar o echar alguna partida, ayudándome a dejar los problemas a un lado de vez en cuando.

Agradecer también a mi tutor, Guillermo Jiménez Díaz, su confianza depositada en mi, su ayuda y su tiempo invertido en guiarme para sacar adelante este proyecto.

A todas las personas que se ofrecieron voluntarias para probar mi herramienta. Gracias por vuestra curiosidad, vuestro tiempo y comentarios.

Resumen

Generador de comportamientos de enemigos para videojuegos 2D

Hacer un videojuego es un proceso donde se mezclan varias artes para crear una experiencia interactiva: una mezcla de código, arte, sonido y diseño. En cada arte existe una serie de herramientas que ayudan a que el proceso sea más cómodo, rápido, y eficaz. Cuando se quiere hacer un videojuego 2D, se necesita tener claras dos cosas: el objetivo del juego, y los obstáculos que el jugador deberá superar para alcanzarlo. Generalmente, los obstáculos surgen en forma de enemigos, y todo enemigo que quiera suponer un desafío debe contar con una inteligencia artificial.

El objetivo del TFG es proporcionar una herramienta que ayude a generar enemigos, creando un catálogo de componentes que sirva como una herramienta para Unity con la que poder crear y configurar de una manera rápida y sencilla distintos tipos de enemigos para videojuegos en 2D. Esta herramienta debe ser tan accesible que pueda ser utilizada por cualquier persona independientemente del rol que tenga: sirve tanto para prototipar niveles, como para diseñar enemigos funcionales.

Palabras clave

Inteligencia artificial, IA, API, videojuegos, 2D, enemigos, Unity

Abstract

Generator of enemy behaviours for 2D videogames

Making a video game is a process in which several arts are mixed to create an interactive experience: a mixture of code, fine arts, sound and design. In every art, there are several tools that helps during the development, making it easier and faster. When one decides to make a 2D video game, there are two main things that need to be clear: the objective of the game, and the obstacles that the player must overcome to reach it. Generally speaking, the obstacles in 2D video games are the enemies, and any enemy that wishes to be a challenge must have some sort of artificial intelligence.

The main goal of this TFG is to provide a piece of software that helps generate enemies, creating a catalogue of components that serve as a Unity add-on to create and configure easy and in a quickly manner various types of enemies for 2D video games. This tool must be accessible, in order to be used by anyone independently of the role they might have: it helps in level prototyping, as well as enemy designing.

Keywords

Artificial Intelligence, AI, API, video games, 2D, enemies, Unity

Índice

Índice de figuras

Capítulo 1

Introducción

"Carece de nobleza quien no se atreve a alabar a un enemigo."
— John Dryden

1.1. Motivación

A la hora de desarrollar un videojuego, hay varias fases por las que se debe pasar y, sin duda, la más importante es el diseño. Durante esta primera fase se plasmarán las ideas en el papel, y se hacen prototipos sencillos para ver si resulta divertido. Tras analizar el resultado, se repetiría el proceso cambiando los elementos que sean necesarios hasta encontrar algo que resulte entretenido de jugar. Además del objetivo del videojuego, habría que diseñar los obstáculos o desafíos que se le van a introducir al jugador para ponerle a prueba.

Existen multitud de herramientas para mejorar el trabajo de los desarrolladores para muchos campos dentro del desarrollo de videojuegos como el sonido, renderizado y programación gráfica pero existen muy pocas categorías de componentes que sirvan para diseñar enemigos, que un diseñador pueda usar por sí mismo sin tener un conocimiento de programación o sin contar con un programador que tenga que gestionar la creación de componentes para crear estos elementos de juego básicos y necesarios.

Los *NPCs* son entidades que aportan la ilusión de que el juego está "vivo", porque estos personajes sirven para introducir ciertas mecánicas de un videojuego: en lugar de tener un menú con una tienda, es mucho más enriquecedor tener un NPC vendedor, con líneas de diálogo que reflejen su personalidad, que sirva como tienda. Los NPCs enemigos de los videojuegos 2D tienen comportamientos que les aportan un cierto grado de realismo, y sirven como obstáculos para poner a prueba al jugador. Es por ello que surge esta herramienta que está orientada a prototipos de videojuegos, a diseñadores y en general a cualquier persona que quiera hacer un videojuego 2D pero que no necesariamente sepa programar. Aspira a proporcionar todas las facilidades necesarias para cubrir una implementación personalizable de los comportamientos de enemigos en un videojuego 2D, una herramienta que sea sencilla además de potente y que resulte accesible y práctica para que sea utilizada por cualquier persona. Con el uso de esta herramienta se pretende trasladar los diseños de enemigos del papel al juego final sin tener que pasar por la programación, para ahorrar tiempo de desarrollo y que tener enemigos funcionales a la par que personalizables

no se convierta en un bloqueante a la hora de que cualquier persona pueda desarrollar un videojuego 2D.

1.2. Objetivos

El objetivo del proyecto es diseñar y crear una herramienta para ser utilizada en *motor de videojuegos* Unity con el objetivo principal de facilitar un catálogo de componentes en C Sharp para comportamientos de enemigos. La herramienta deberá ser rápida y sencilla de utilizar por cualquier persona que quiera hacer un videojuego 2D, sin importar sus conocimientos de programación. Los componentes de este catálogo se podrán personalizar, añadir con facilidad e incluso mezclar para generar comportamientos nuevos. Será indispensable que cumpla con los principios de *abstracción* y programación orientada a objetos.

Para llevar a cabo el desarrollo de este catálogo de componentes, se estudiarán previamente las técnicas de creación de *NPCs*, así como las herramientas que ofrecen facilidades para crearlos como por ejemplo los motores de videojuegos orientados a desarrolladores sin experiencia en programación. Tras este análisis, se diseñará el catálogo de componentes que se van a implementar, y se justificará el uso de dichos componentes para llevar a cabo implementaciones de enemigos clásicos de videojuegos 2D. Posteriormente, se hará una implementación de dichos componentes dando forma al catálogo y, finalmente, se realizará una evaluación con usuarios de la herramienta en el que se recogerán sus impresiones y comentarios sobre el catálogo de componente, a fin de identificar tanto los puntos positivos como los negativos.

1.3. Plan de trabajo

Para este proyecto se establecerá una metodología ágil de desarrollo desde el primer momento. La metodología a seguir se había estudiado en la asignatura Metodologías Ágiles de Producción, y empleada en muchas asignaturas de la carrera como Proyecto. El tutor de este TFG haría las veces de cliente, ya que se hacían reuniones cada dos semanas en las que le mostraría los avances que iba realizando, escuchaba sus comentarios y críticas al respecto, resolvía con él las dudas que tuviera sobre el desarrollo y finalmente se marcaba la hoja de ruta para las siguientes dos semanas. Este modelo de trabajo se denomina Scrum, y es bastante simple: está basado en reuniones frecuentes, rápidas y diarias para cumplir unos objetivos a corto plazo denominados "sprints". Además de la metodología a implantar se decidió que todo el código que desarrollase iba a estar escrito íntegramente en inglés, ya que el lenguaje no debería suponer una barrera si algún usuario decidiera investigar y estudiar el código.

El catálogo de componentes se ha diseñado de forma incremental a lo largo de los meses de desarrollo. Los capítulos que conforman esta memoria describen una parte del proceso que se va a llevar a cabo para crear este catálogo. En primer lugar se realizará un estudio del Estado del Arte (Capítulo 2), donde se definirá brevemente qué es la Inteligencia Artificial en videojuegos además de mencionar unas técnicas de creación de comportamientos para enemigos. Además, se estudiarán las diversas herramientas de creación de comportamientos, argumentando por qué se han escogido. Así mismo, en ese capítulo se analizarán los *motores de videojuegos* orientados a desarrolladores noveles, de los cuales se van a extraer sus puntos positivos que podrían aportar al catálogo.

Posteriormente, se dedicará el Capítulo 3 a definir y justificar el diseño de la herramienta. En este capítulo de explicará en profundidad el catálogo de componentes, se hablará de sus inspiraciones y se plasmarán los diseños de los componentes, justificando la existencia de estos con ejemplos prácticos de videojuegos 2D. Una vez el catálogo de componentes esté diseñado, y se haya puesto a prueba su utilidad de forma teórica, se procederá a implementar los componentes propuestos.

Como se explicará en el Capítulo 4, este catálogo se habrá de implementar en el *motor de videojuegos* Unity, y se definirá en este mismo capítulo la estructura de los componentes que se irán a desarrollar. Es importante definir la arquitectura básica del catálogo, y explicar tanto la implementación interna de los componentes como la personalización que ofrece cada uno. Estas implementaciones, y sus correspondientes detalles internos, se explicarán de forma detallada en este capítulo de implementación en el que además se mencionarán las posibles formas de combinación de componentes que ofrecerá el catálogo.

Una vez el catálogo se haya implementado, se probará la herramienta con usuarios para valorar su utilidad y extraer posibles mejoras para el proyecto. Como se explica en el Capítulo 5, se redactará un documento diseñando las pruebas según lo que se ha aprendido en la asignatura Usabilidad y Análisis en Videojuegos, respecto a cómo identificar los objetivos de la prueba, saber formular preguntas que me aporten datos coherentes y diseñar las pruebas que los probadores iban a tener que resolver para que sean amenas. Los usuarios que van a probar la herramienta en su estado final tendrán que construir enemigos utilizando este catálogo, por lo que se procurará que los usuarios tengan conocimientos básicos del desarrollo de videojuegos, pero que no hayan tenido contacto previo con la herramienta a fin de que se pueda estudiar tanto la utilidad de la herramienta como la curva de aprendizaje que tiene. Los resultados de las pruebas se recopilarán, se analizarán y se extraerán unas conclusiones de estos resultados, haciendo una crítica tanto de lo que falta como de lo que se podría mejorar.

Finalmente, en el Capítulo 6 se recogerán las conclusiones del proyecto en el que se resumirá el proceso final de desarrollo, así como los puntos positivos y negativos que tiene la herramienta. Posteriormente, se definirá el trabajo futuro que tendría el catálogo y se justificarán las posibles ampliaciones de este catálogo de componentes.

Capítulo 2

Estado de la Cuestión

En este capítulo se va hacer un estudio sobre las técnicas y herramientas utilizadas para la creación de enemigos. Se hará una breve mención a la inteligencia artificial en general y se especificará cómo funciona en videojuegos. Posteriormente se explorarán algunas técnicas de creación de enemigos, argumentando su uso en juegos 2D conocidos. Se mencionarán, además, herramientas actuales que sirven para la creación de *NPCs* y que se utilizan para generar enemigos con las técnicas previamente comentadas. Finalmente, se explorarán los *motores de videojuegos* más sencillos de utilizar, orientados tanto a programadores noveles como a diseñadores, y se extraerán los puntos positivos que servirán de inspiración para crear un catálogo de componentes que resulte sencillo de utilizar.

2.1. Breve introducción a la IA

La inteligencia artificial, abreviado IA, es la inteligencia demostrada por máquinas pero programada por humanos. Se podría resumir la inteligencia artificial como cualquier dispositivo que es capaz de percibir su entorno, en base a lo que conoce y/o a unas reglas limitantes, tomar decisiones que maximicen las probabilidades de alcanzar sus objetivos. Particularmente, en videojuegos es lo que le permite a los *NPCs* simular comportamientos inteligentes, con una cierta lógica, considerando una serie de objetivos que deben de alcanzar y de restricciones que deben cumplir para poder tomar una decisión con coherencia. A día de hoy, la inteligencia artificial es una parte intrínseca de los videojuegos, debido a que mejora la experiencia de juego y crea desafíos al jugador.

Los videojuegos empezaron a cobrar fuerza en la década de los 80, cuando vivieron su edad de oro. Lo que diferenciaba a los videojuegos de esta época con respecto a los predecesores era que podían ser jugados de forma individual, ya que contaban con un modo de un jugador o modo "contra la máquina". Estos juegos contaban con elementos nuevos que hacía que supusiera un reto jugarlos, ya que el jugador tenía que enfrentarse a una inteligencia artificial que le obstaculizara el llegar al final del juego y le supusiera un desafío que superar. Esta inteligencia artificial, por regla general en los juegos de la época, se manifestaban en forma de enemigos.

La inteligencia artificial de estos enemigos, o *NPCs*, se aleja de los algoritmos matemáticos de toma de decisiones, y se centra en otros aspectos del juego como la capacidad de moverse a través de un plano. La búsqueda de una ruta, por ejemplo, es un uso tan

común en la inteligencia artificial que, a día de hoy, se da por hecho que todos los enemigos móviles de un videojuego la tienen, en mayor o menor medida.

Hay muchas técnicas para diseñar e implementar los comportamientos que tendrán los enemigos de cualquier videojuego, pero dos de las más utilizadas son las máquinas de estado finitas y los árboles de comportamiento que se verán en las siguientes subsecciones.

2.1.1. Máquinas de estado finitas

Las máquinas de estado finitas, conocidas por sus siglas anglosajonas FSM, *finite state machine*, son un conjunto de estados abstractos que definen los estados de un sistema. En el caso de las FSM de videojuegos, sería el conjunto de estados que puede tomar una entidad, contando con un único estado activo. Una entidad puede cambiar su estado a otro mediante un *input* determinado, y esas relaciones entre estados son lo que se denominan transiciones. Como describe Bosch (2009), *"Las FSM permiten definir una serie de estados y que cada uno de ellos de un comportamiento distinto a las entidades que los contienen"*. Las entidades, por tanto, tendrán un comportamiento distinto en cada uno de los estados, y que se manifiesta directamente en el videojuego. Mediante ciertos eventos que pueden transcurrir durante una partida, una entidad enemiga puede identificar esos eventos como *input* para cambiar su estado activo, dando así la sensación de que los enemigos son entidades inteligentes.

Diseñar enemigos, como menciona Sabbagh (2015), es "*un arte en si mismo*", ya que el diseñador tiene que saber crear en los enemigos de un videojuego un término medio entre que resulte un desafío para las habilidades del jugador pero dándoles la oportunidad, además de la satisfacción, de derrotarlos. Uno de los primeros juegos con un diseño de enemigos que utilizan máquinas de estado finitas fue el Pac-Man, del japonés Toru Iwatani (1980).

Pac-Man es un videojuego arcade sencillo en apariencia, pero sorprendentemente profundo. El objetivo del juego es conseguir que el protagonista del juego, un círculo amarillo con boca, se coma todos los puntos pequeños de un laberinto para avanzar de nivel. No obstante, hay cuatro fantasmas en este laberinto que harán que termine la partida si atrapan al jugador, por lo que este tendrá que o bien esquivarlos, o bien comerse un objeto especial que le permite comerse a los fantasmas. Como se puede ver en la Figura 2.1, la lógica que se debería seguir para jugar de forma eficiente a Pac-man se puede describir como una máquina de estados finita donde cada estado representa la acción más deseable en función del input.

Comenta Mateas (2005) en su tesis, que *"los comportamientos de los fantasmas son críticos para entender el juego"*. Cuando uno piensa en los fantasmas del Pac-Man, la definición superficial del comportamiento es que se limitan a perseguir al jugador por el laberinto y si el jugador ha obtenido el objeto que le permite comerse a los fantasmas, estos huirán de él. Lo cierto es que los movimientos de los fantasmas no son tan sencillos, ya que tienen unos comportamientos definidos por máquinas de estado finitas, distinguibles entre si y además estos comportamientos de movimiento impactan directamente sobre la experiencia del jugador. Estos comportamientos deben suponer un desafío para el jugador, pero sin ser imposibles de superar, y darle la sensación de que los enemigos siguen unas tácticas definidas.

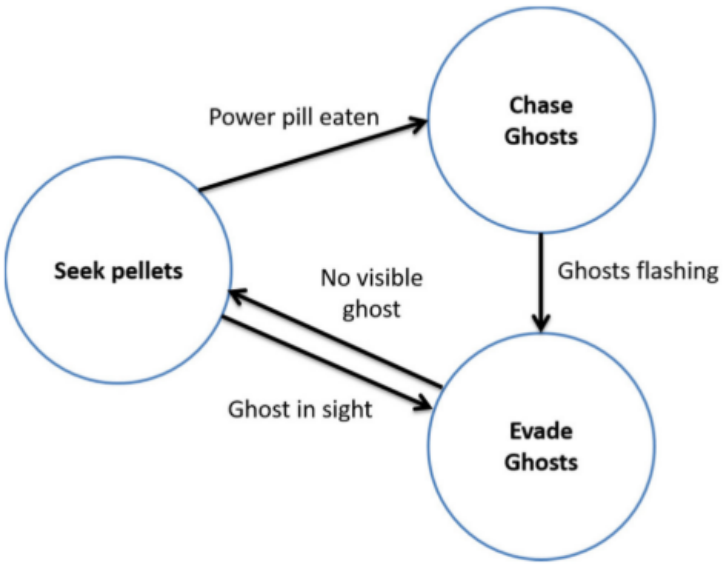


Figura 2.1: FSM de Pac-man, extraído del libro de Yannakakis y Togelius (2018)

Cada fantasma de Pac-Man tiene un comportamiento distinto a los demás, representado en una forma de moverse por el laberinto que sea un indicativo de la estrategia que sigue cada fantasma. Davis (2008) explica que los fantasmas de Pac-Man tienen unos movimientos diferentes porque cada color de enemigo representa un fantasma personalizado con nombre propio, y que la forma de moverse de estos fantasmas es debido a que tienen diferentes "personalidades". Como se puede ver en la Figura 2.2, el juego original de 1980 contaba con 4 tipos de fantasmas distintos, y que cada uno de ellos tenía una *personalidad* (en inglés, *character*) y un *mote* (*nickname*). En primer lugar, el fantasma rojo, *Blinky*, limita su táctica a perseguir a Pac-Man y por ello se le denomina *shadow* en el juego original. En el caso del fantasma rosa, *Pinky*, su tipo de personaje era *speedy*, que sugiere que va más deprisa que el resto. No obstante, todos los fantasmas se mueven a la misma velocidad, por lo que el nombre no se refiere a la velocidad de movimiento sino a la anticipación de los movimientos del jugador ya que intenta rodear a Pac-Man y cortar rutas de escape. El resultado de alternar estos dos tipos de movimiento nos da lugar al tercer fantasma, el fantasma azul, *Inky*, que alterna a intervalos aleatorios entre perseguir directamente a Pac-Man y rodearle mientras que el fantasma amarillo, *Clyde*, se mueve de forma libre por el laberinto, y parece no reparar en el jugador de forma activa ni siquiera cuando este está cerca.

Cada uno de estos *NPCs* tenía una forma distinta de moverse por el laberinto, y esas distinciones de movimiento eran lo que definía la táctica para atacar al jugador. Todos en conjunto representaban el obstáculo al que se tenía que enfrentar el jugador para avanzar por los laberintos y conseguir la mejor puntuación.

2.1.2. Árboles de comportamiento

Los videojuegos de los años 80 dejaron claro que los enemigos ya no esperaban a que llegase su turno para pasar a la acción, sino que eran personajes que estaban en constante movimiento, siempre alerta, diseñando tácticas y estrategias en base a los movimientos del jugador. Posteriormente, los enemigos empezaron a ser capaces de modular sus tácticas

Figura 2.2: Fantasmas de Pac-Man, 1980

para que sean más o menos efectivas, ofreciendo así diferentes niveles de dificultad. Este avance en los comportamientos de los *NPCs* fue posible gracias a los árboles de comportamiento.

Un árbol de comportamientos es un árbol de nodos, organizados según una jerarquía, que controlan el flujo de decisiones de una entidad con inteligencia artificial. Como explica Simpson (2014) en su artículo, en los extremos del árbol se sitúan los comandos de acción que controlan la entidad, y en las ramas hay varios tipos de nodos de utilidad, que deciden por qué camino descenderá la toma de decisión, con el fin de que el comando final sea la acción o secuencia de acciones que mejor de adaptan a la situación. Los árboles de comportamiento tienen dos grandes ventajas: La *abstracción* de la implementación y la *modularidad* de los comportamientos.

Un ejemplo perfecto para ejemplificar los árboles de comportamiento es el videojuego Hollow Knight (Figura 2.3) creado por Team Cherry en el año 2017 y financiado mediante apoyo popular en una exitosa campaña de *kickstarter*. Es un videojuego de acción lateral donde manejas a un personaje que viaja por unos detallados mapas dibujados a mano y se enfrentará a una cantidad enorme de enemigos, muy bien construidos y variados. Igual que en Pac-man, los enemigos siguen tácticas tanto de movimiento como de ataque que varían en función del enemigo. Estas tácticas son fácilmente distinguibles por el jugador, ya sea porque las ha visto previamente o porque el aspecto del enemigo te aporta información visual sobre esos patrones.

Hay enemigos que sólo atacan en determinadas condiciones: si el jugador tiene poca vida, está muy cerca o si está siendo atacado por otro enemigo. De la misma manera, hay enemigos que al ser atacados deciden huir, retirándose para volver a intentar el ataque momentos después. Por ejemplo, en la Figura 2.3 se observa un tipo de enemigo que vuela manteniendo las distancias con el jugador, y es precisamente en función de esa distancia

Figura 2.3: Hollow Knight (Team Cherry, 2017)

lo que define el modo de ataque: si está lejos, lanzará proyectiles en línea recta hacia el jugador mientras que si está lo suficientemente cerca cargará hacia él[1]. El juego cuenta con entidades más complicadas de derrotar que son los jefes. Los jefes son enemigos más poderosos que generalmente el jugador debe derrotar si quiere avanzar en el videojuego. Lo que caracteriza a un jefe de un enemigo normal es que no tiene una única forma de atacar al jugador, sino que va incorporando ataques más complicados a medida que el jugador le hiere, incluso es capaz de cambiar de comportamiento en mitad de la batalla.

2.2. Herramientas para la creación de NPCs

Como se ha comentado anteriormente, las técnicas de creación de *NPCs* han ido evolucionando a lo largo del tiempo. Debido a ello, surgió la necesidad de crear herramientas que ayudasen a los desarrolladores a crear enemigos complejos. Se han escogido dos herramientas relativamente populares, que proporcionan interfaces gráficas muy sencillas.

2.2.1. Playmaker

El equipo, Team Cherry, reveló en una entrada de su *kickstarter*[2] que los árboles de comportamientos de Hollow Knight han sido creados con una herramienta de *scripting visual* llamada Playmaker, por Hutong Games[3].

Playmaker es una herramienta muy versátil, orientada a artistas y diseñadores para permitirles desarrollar videojuegos sin necesidad de código. Su interfaz es muy visual, similar al diseño en papel, pero es capaz de extraer el potencial que tiene Unity con un amplio catálogo de acciones. Las acciones son eventos comunes de un videojuego como, por ejemplo, hacer zoom con la cámara. En circunstancias normales estas acciones tendrían que estar programadas a mano, por lo que contar con una herramienta de este tipo supone un ahorro de tiempo para personas que no saben programar. De la misma manera, esta

[1] `https://hollowknight.fandom.com/wiki/Lance_Sentry`
[2] `http://teamcherry.com.au/tag/ai/`
[3] `https://hutonggames.com/`

herramienta sirve de ayuda para las personas que sí sepan programar porque es fácilmente extensible con *scripts* propios, por lo que se podría utilizar con los componentes de mi TFG.

Playmaker está disponible en la *asset store* de Unity, una tienda virtual de Unity donde se pueden comprar distintos elementos para el desarrollo de videojuegos, por un precio razonable. No obstante, en el caso de las empresas más pequeñas o con pocos recursos, ¿Acaso existe una *API* de creación de *NPCs* que solucione este problema, y que permita a un equipo crear enemigos funcionales con un gasto mínimo?

2.2.2. Behaviour Bricks

Al buscar sobre herramientas comerciales para crear *NPCs* en general, y enemigos en particular, aparece un artículo muy interesante sobre un modelo integrador de máquinas de estados y árboles de comportamiento para videojuegos creado por Sagredo-Olivenza et al. (2016). La herramienta se llama Behavior Bricks [4], sirve para hacer árboles de comportamiento para videojuegos de forma visual con un editor propio, pensado para el *motor de videojuegos* Unity. Es una herramienta gratuita que se puede encontrar en la *asset store*, la tienda virtual de Unity. Esta herramienta fue desarrollada por Padaone Games, empresa asociada a la Universidad Complutense de Madrid.

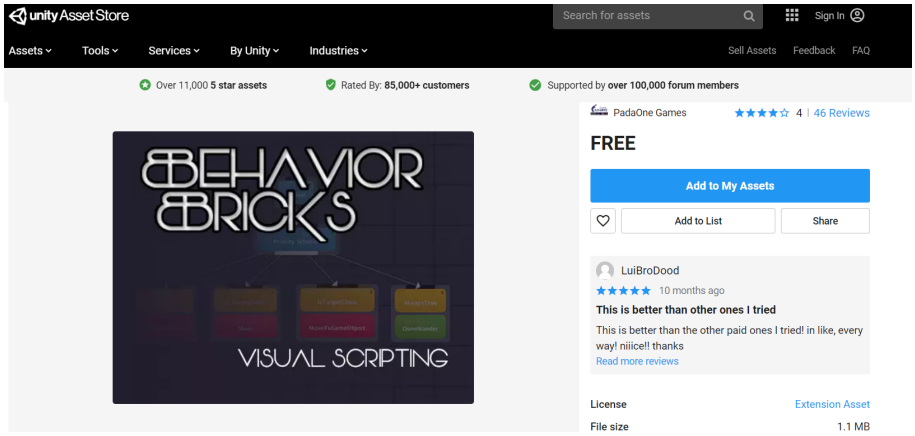


Figura 2.4: Behaviour Bricks, Padaone Games

Behaviour Bricks es una herramienta de *scripting visual* y por tanto presenta muchas similitudes con Playmaker: ambas herramientas aspiran a promover la colaboración entre diseñadores y programadores. No obstante, Behaviour Bricks pone un énfasis extra en el rendimiento de la aplicación y en la gestión de memoria. Gracias a su optimización se garantiza que el uso de esta herramienta no suponga un lastre en el rendimiento del juego final.

[4] `http://bb.padaonegames.com/doku.php`

2.3. Motores de videojuegos

Se ha hablado de las herramientas externas, fácilmente integrables en el *motor de videojuegos* Unity y que existen para crear *NPCs* inteligentes, pero lo cierto es que en determinados motores de videojuegos estas herramientas forman parte del núcleo principal. A continuación, se van a analizar dos motores de videojuegos caracterizados por ofrecer diversas ayudas para hacer videojuegos a usuarios que no necesariamente tengan conocimientos avanzados de programación.

2.3.1. Construct 3

Un ejemplo de *motor de videojuegos* orientado a personas que no necesariamente sepan programar es Construct 3, creado por Scirra [5]. Esta herramienta destaca porque está orientada a videojuegos en 2D, es sencilla de usar y es muy visual. A pesar de que se puede crear código propio con este motor, la idea de la herramienta es que mediante unas interfaces gráficas se puedan construir comportamientos para *NPCs*.

Construct 3 cuenta con un elemento llamado *hojas de evento* (en inglés, event sheets), que son similares a los archivos fuente de los lenguajes de programación convencionales. Cada hoja tiene una lista de eventos, que contienen condicionales o sensores, y una vez esos sensores detectan una condición preestablecida, se activarán una serie de acciones asociadas. Con este sistema, es posible hacer videojuegos sofisticados sencillamente arrastrando y soltando los elementos en la escena, añadiendo los eventos necesarios y sin necesidad de aprender un lenguaje de programación. Por supuesto, en Construct 3 existen objetos al igual que en otros *motores de videojuegos*, y con los comportamientos adecuados se pueden convertir en agentes inteligentes que sirvan de enemigos en videojuegos.

Figura 2.5: Editor Construct 3, 2020

Lo más positivo de Construct 3, que he intentado replicar en la herramienta, es lo rápido y sencillo que es hacer un videojuego de tal manera que cualquier persona puede aprender a usarla en poco tiempo. Existe una curva de aprendizaje en este *motor de videojuegos*, por supuesto, pero es muy leve y llevadera. Para desarrolladores noveles, lo más atractivo

[5] `https://www.scirra.com/`

de la herramienta es que ya existe un catálogo de componentes previo (Figura 2.6), así como una serie de sensores que están muy bien construidos, con un nivel de *abstracción* que te permite tener un juego construido sin que el usuario se preocupe de cómo funciona por dentro, sacando así un videojuego con poco esfuerzo y en muy poco tiempo.

Si se compara con Unity, el motor de videojuegos empleado en este TFG, la conclusión es que Unity es mucho más potente pero para realizar cualquier comportamiento que se desee es necesario entrar de forma obligatoria a hacer la programación a mano, lo que puede resultar complicado y frustrante a desarrolladores noveles. Por ello, a pesar de que Unity es el motor de videojuegos que finalmente he escogido por diversos motivos que trato en el Capítulo 4, se querían integrar los valores que hacen que Construct 3 sea tan accesible.

Figura 2.6: Movimientos en Construct 3, 2020

Con Construct 3, es posible crear enemigos de forma sencilla ya que nos proporciona funcionalidades básicas como un movimiento en linea recta hacia delante (*bala*) o la posibilidad de modificar la orientación del objeto (*girar*). Mezclando esos dos elementos constantemente se puede obtener un enemigo que se lance a por el jugador girando previamente hacia su posición. Además, si hace eso mismo cada cierto tiempo se obtendría un enemigo que te persigue, algo con un poco de profundidad y se puede realizar sin programar una línea de código. Algunos de los movimientos pre-programados son movimientos simples, no obstante que también cuenta con movimientos físicos, movimientos en cuadrículas o incluso proporciona las herramientas para crearlo como una combinación de movimientos (*personalizado*). Sin embargo Unity carece de una herramienta de este estilo, y es lo que este TFG intentará suplir mediante un catálogo de componentes de movimiento similar para poder hacer juegos en 2D con enemigos funcionales.

2.3.2. Game Maker

Entre los *motores de videojuegos* sencillos de utilizar por casi cualquier persona y que nos permiten crear *NPCs* destaca el motor Game Maker[6] que, al igual Construct 3, se especializa en videojuegos 2D y se caracteriza por ser muy fácil de utilizar, pudiendo hacer videojuegos mediante un sistema visual de arrastrar y soltar, sin tener que programar. Además de este sistema, ofrece programación de *scripts* con un lenguaje propio conocido como lenguaje *Game Maker* para apelar a los usuarios más avanzados, o que prefieran aprender a programar.

Como herramienta para crear *NPCs*, Game Maker tiene funcionalidades para hacer objetos de forma gráfica, con un sistema de nodos que permite aplicar componentes a objetos

[6] `https://www.yoyogames.com/gamemaker`

en la escena de forma muy cómoda. Al igual que sucede con Construct 3, Game Maker cuenta con un sistema de eventos que se activarán de forma automática bajo una serie de condiciones también pre-programadas. Es un *motor de videojuegos* sencillo de aprender a usar, pero no es tan sencillo porque no aporta tantos componentes de movimiento pre-diseñados sino que cuenta con un editor de código donde se pueden programar scripts en un lenguaje propio llamado *Game Maker Language*.

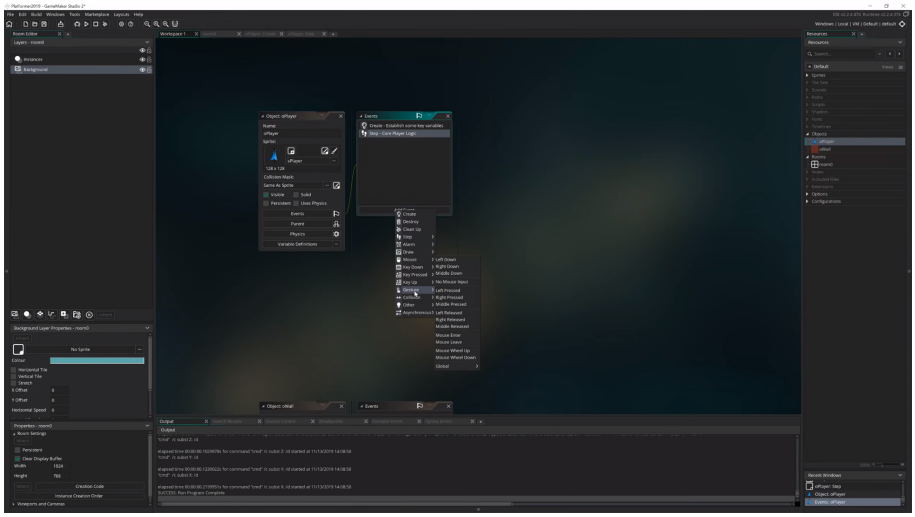


Figura 2.7: Game Maker, 2020

Mientras que el *motor de videojuegos* Game Maker cuenta con muchas facilidades a la hora de crear videojuegos en dos dimensiones, el hecho de que permita crear scripts tiene la ventaja de que el usuario puede crear algo que se adapte a sus necesidades concretas, y aunque este usuario no sepa programar existen muchos tutoriales sobre Game Maker y su particular lenguaje de programación. Es cierto que en Construct 3 se tarda muy poco en tener un personaje moviéndose en 8 direcciones que sirva para un *NPC* de un videojuego con vista de pájaro, mientras que en Game Maker ese mismo control del movimiento habría que programarlo, pero el principal problema que tiene Construct 3 es que todos los videojuegos parecen iguales, mientras que los videojuegos publicados con el motor Game Maker tienen una variedad muy notable, y son más populares por lo general. Unity es más complicado que Game Maker, y esto puede resultar abrumador, pero es precisamente en un motor como este donde hace mucha más falta una herramienta que ayude a los desarrolladores noveles, pero que además tenga un potente editor de *scripting* donde los desarrolladores más veteranos puedan modificar la herramienta, o ampliarla a su gusto.

Un ejemplo de juego actual con *NPCs* creados en este *motor* es el título Undertale[7], creado por Toby Fox en el año 2015. Este videojuego se aleja un poco de los juegos de avance lateral que se han comentado previamente, porque este es un juego RPG donde cada enfrentamiento luchas contra un enemigo en una pantalla por turnos, donde en tu turno puedes elegir una acción y en el turno del enemigo te ataca con un ataque de estilo *bullet hell*, que consisten en una serie de proyectiles que aparecerán en la pantalla. Durante un combate, el jugador controla un pequeño corazón rojo, confinado en una región en mitad de la pantalla, y deberá esquivar los proyectiles que aparecen en la pantalla en el turno del enemigo. Estos proyectiles rara vez son balas que viajan en línea recta, sino que siguen unos patrones sencillos de movimiento e infligen daño al jugador si entran en colisión con

[7] `https://undertale.com/`

este. Los movimientos de los proyectiles pueden ser laterales, en caída libre o avanzar de forma sinodal que son formas de moverse muy clásicas de los videojuegos laterales en 2D.

Figura 2.8: Undertale (Toby Fox, 2015)

2.4. Conclusiones del análisis

A modo de resumen, se recopilarán los puntos positivos tanto de las herramientas de creación de *NPCs* como de los *motores de videojuegos* orientados a desarrolladores noveles que se han comentado. Este proyecto, como se ha explicado previamente, consistirá en un catálogo de componentes de movimiento y surge con la idea de ser una herramienta para hacer enemigos sencillos y personalizables. Este catálogo deberá ser modular y, por tanto, fácilmente ampliable con comportamientos nuevos por lo que se podría integrar su contenido con alguna técnica de creación de comportamientos complejos, ya sea con FSM o con árboles de comportamiento que se han estudiado al principio de este capítulo.

En cuanto a los puntos positivos de los motores de videojuegos analizados, se destaca de Construct 3 la facilidad que ofrece al usuario para añadir componentes de movimiento a cualquier objeto gracias a su ventana emergente. Además, Construct 3 cuenta con un catálogo propio de componentes y de componentes capaces de percibir el entorno para activar "eventos", con lo que se añade una capa de personalización a los enemigos que sería muy positivo para el catálogo de componentes que se va a desarrollar. Del análisis acerca de Game Maker han resultado especialmente interesantes las técnicas de *scripting visual*, que hacen que resulte sencillo utilizar una herramienta aunque no se tenga un conocimiento avanzado de la misma, si esta emplea ayudas visuales que, por ejemplo, representen el movimiento que va a seguir un enemigo antes de que ocurra.

Capítulo 3

Diseño del catálogo de componentes

En este capítulo se realizará el diseño del catálogo de componentes que se van a desarrollar. Antes de empezar a diseñar este catálogo de componentes primero hay que entender la utilizad que debería tener. Para ello, en primer lugar se explorará de dónde surge la idea de hacer un catálogo de comportamientos de enemigos, se explicará qué tipos de atributos se van a diseñar y, finalmente, se justificará que este catálogo es una herramienta para diseñar enemigos de videojuegos 2D, construyendo enemigos del videojuego Cave Story de forma conceptual utilizando los comportamientos descritos.

3.1. Trabajo previo sobre comportamientos de NPCs

La idea principal de este catálogo de componentes es contar con una herramienta sencilla y accesible, que cuente con un amplio catálogo de componentes para el *motor de videojuegos* Unity. Como se ha tratado al final del Capítulo 2, Unity puede resultar más abrumador de aprender a utilizar por desarrolladores noveles, que pueden preferir otros *motores* más sencillos de utilizar como Construct 3 o Game Maker. Esta herramienta apela precisamente a acercar Unity a cualquier persona que quiera crear un videojuego en 2D, para que cualquier persona pueda construir enemigos para sus videojuegos con este catálogo de componentes. De la misma manera, los desarrolladores más expertos pueden utilizar el catálogo de base para construir inteligencia artificial más compleja al combinar los componentes en árboles de comportamiento.

La inspiración principal de hacer este catálogo sencillo de componentes surge de un interesante artículo publicado en la página Gamasutra, y escrito por Bright (2014). En su artículo se recoge una lista de atributos que son muy comunes en los juegos retro, títulos clásicos de las consolas de los años ochenta, que van desde patrones de movimiento a las distintas habilidades de ataque que tenían los enemigos de la época.

Su análisis en el campo ha resultado muy esclarecedor a la hora de entender qué se debería esperar de un componente tan pequeño, lo que haría cada componente así como posibles combinaciones que se podrían hacer. Como se ve en la Figura 3.1, la lista de movimientos que propone Bright (2014) consiste en un nombre explicativo, una breve descripción que permita visualizar el movimiento acompañado con un ejemplo de un juego popular. En su artículo no sólo analiza comportamientos de movimiento sino que también explora las habilidades y formas de ataque que podrían tener los enemigos.

Movement Attributes

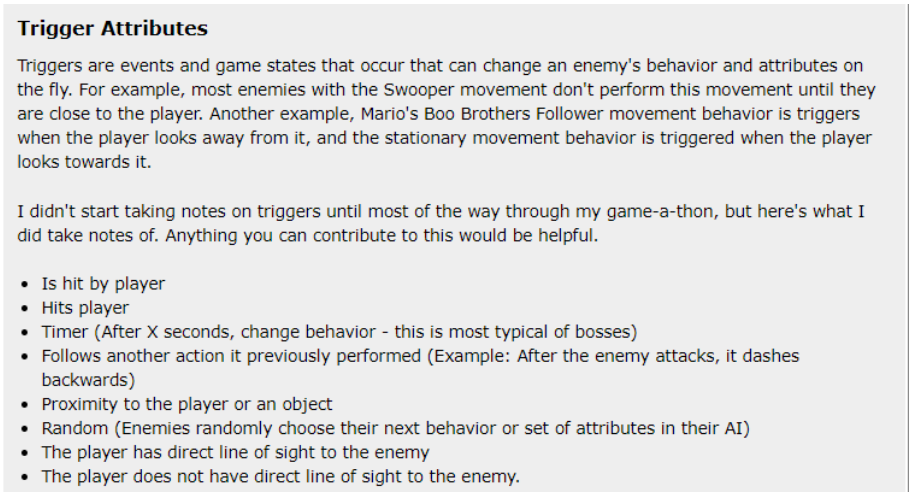
Stationary	**The enemy does not move at all.**
Walker	**The enemy walks or runs along the ground.** Example: Super Mario's Goombas
Riser	**The enemy can increase its height** (often, can rise from nothing). Examples: Super Mario's Piranha Plants and Castlevania's Mud Man
Ducker	**The enemy can reduce its height** (including, melting into the floor). Example: Super Mario's Piranha Plants
Faller	**The enemy falls from the ceiling onto the ground**. Usually these enemies are drops of something, like acid. Some games have slimes that do this.
Jumper	**The enemy can bounce or jump**. (some jump forward, some jump straight up and down). Examples: Donkey Kong's Springs, Super Mario 2's Tweeter, Super Mario 2's Ninji
Floater	**The enemy can float, fly, or levitate**. Example: Castlevania's Bats
Sticky	**The enemy sticks to walls and ceilings**. Example: Super Mario 2's Spark
Waver	**The enemy floats in a sine wave pattern**. Example: Castlevania's Medusa Head
Rotator	**The enemy rotates around a fixed point**. Sometimes, the fixed point moves, and can move according to any movement attribute in this list. Also, the rotation direction may change. Example: Super Mario 3's Rotodisc, These jetpack enemeis from Sunsoft's Batman (notice that the point which they rotate around is the player)
Swinger	**The enemy swings from a fixed point**. Example: Castlevania's swinging blades
Pacer	**The enemy changes direction** in response to a trigger (like reaching the edge of a platform). Example: Super Mario's Red Koopas
Follower	**The enemy follows the player** (Often used in top-down games). Example: Zelda 3's Hard Hat Beetles
Roamer	**The enemy changes direction completely randomly**. Example: Legend of Zelda's Octoroks
Liner	**The enemy moves in a straight line directly to a spot on the screen**. Forgot to record the enemies I saw doing this, but usually they move from one spot to another in straight lines, sometimes randomly, other times, trying to 'slice' through the player.
Teleporter	**The enemy can teleport from one location to another**. Example: Zelda's Wizrobes
Dasher	**The enemy dashes in a direction, faster than its normal movement speed**. Example: Zelda's Rope Snakes
Ponger	**The enemy ignores gravity and physics, and bounces off walls in straight lines**. Example: Zelda 2's "Bubbles"
Geobound	**The enemy is physically stuck to the geometry of the level**, sometimes appears as level geometry. Examples: Megaman's Spikes, Super Mario's Piranha Plants, CastleVania's White Dragon
Tethered	**The enemy is tethered to the level's geometry by a chain or a rope**. Example:

Figura 3.1: Artículo de Bright (2014), Atributos de movimiento

Además de las listas de comportamientos de movimiento y de ataque, Bright (2014) propone una serie de atributos de distinta naturaleza que son los *Trigger Attributes*, que se pueden definir como atributos de tipo sensor y que se recogen en la Figura 3.2. El autor los define como *"eventos y estados de juego que se puedan dar, que pueden cambiar el comportamiento de un enemigo así como sus atributos en ejecución"*. Esto quiere decir que un enemigo que posea un sensor sabrá detectar variaciones en su entorno y actuar en consecuencia. Gracias a estos eventos el enemigo es capaz de modificar su estado, por lo que su comportamiento se podría definir con una FSM, que se comentó en el Capítulo 2.

3.2. Análisis de los NPCs de Cave Story

El primer paso a la hora de diseñar esta herramienta fue realizar un estudio de los enemigos de Cave Story, jugando al videojuego para enfrentar a los enemigos y así estudiarlos, tratando de entender sus movimientos y definirlos en función de uno o varios atributos de movimiento que aparecen en la lista de Bright (2014) de la Figura 3.1.



Trigger Attributes

Triggers are events and game states that occur that can change an enemy's behavior and attributes on the fly. For example, most enemies with the Swooper movement don't perform this movement until they are close to the player. Another example, Mario's Boo Brothers Follower movement behavior is triggers when the player looks away from it, and the stationary movement behavior is triggered when the player looks towards it.

I didn't start taking notes on triggers until most of the way through my game-a-thon, but here's what I did take notes of. Anything you can contribute to this would be helpful.

- Is hit by player
- Hits player
- Timer (After X seconds, change behavior - this is most typical of bosses)
- Follows another action it previously performed (Example: After the enemy attacks, it dashes backwards)
- Proximity to the player or an object
- Random (Enemies randomly choose their next behavior or set of attributes in their AI)
- The player has direct line of sight to the enemy
- The player does not have direct line of sight to the enemy.

Figura 3.2: Artículo de Bright (2014), Atributos de tipo sensor

Cave Story, *la historia de la cueva*, en español, es un videojuego de aventuras 2D desarrollado por un sólo desarrollador, el japonés Daisuke Amaya, conocido con el apodo de "Pixel". El juego original fue publicado de forma totalmente gratuita, algo sorprendente teniendo en cuenta la inusual cantidad de contenido que aporta, y está disponible en la página web oficial [1] o en su página tributo más popular [2] donde además hay traducciones del juego hechas por la comunidad. Es considerado uno de los mejores juegos *indie* por su magistral diseño, historia, e impacto en la industria de los videojuegos. A pesar de ser un juego desarrollado por una sola persona cuenta con una variedad muy notable de enemigos, cada uno de ellos con comportamientos únicos o combinaciones de comportamientos sencillos.

Un aspecto fundamental de los enemigos en los videojuegos 2D es que su movimiento debe ser sencillo de entender, aportando información en un vistazo acerca de cómo puede atacar dicho enemigo. Xu (2016) explora en su artículo el complejo diseño del juego, mencionando que "*cuantos más enemigos encuentres, más fácil resultará derrotarlos*"; además de que "*algunos enemigos tienen pistas visuales para ayudarte*". Por ejemplo, la rana de la Figura 3.3 se puede intuir que saltará por el aspecto visual que tiene, dando pistas de cómo se va a mover. Estas pequeñas pistas visuales son lo que permite al jugador tener una idea de cómo derrotar a cada enemigo.

3.2.1. Enemigo Rana

Los enemigos de Cave Story son enemigos sencillos en apariencia, pero con mucha diversidad de comportamientos, formas de moverse y de atacar al jugador. Algunos enemigos, como la rana[3] que se ve en la imagen 3.3, avanza hacia el jugador dando saltos cada cierto tiempo. Gráficamente, tiene sentido su movimiento ya que hay una coherencia entre su aspecto visual y su comportamiento. A pesar de su simplicidad, ese movimiento no es trivial: es una combinación de un movimiento hacia arriba con un movimiento lateral hacia la posición del jugador, por lo que además de moverse tiene que conocer dónde se encuentra el jugador en el momento de efectuar el salto. Según los componentes de movimiento de

[1] `https://studiopixel.jp/`
[2] `https://www.cavestory.org/`
[3] `https://cavestory.fandom.com/wiki/Frog`

la Figura 3.1, el movimiento de este enemigo se podría construir como un Jumper si se considera que, según la descripción que nos propone Bright (2014) pueden saltar en línea recta o hacia delante en una dirección. Este enemigo, además, tiene una condición de activación que se corresponde con uno de los atributos sensor que aparecen en la Figura 3.2, concretamente con el sensor de "proximidad con el jugador u otro objeto" porque la rana no salta hacia el jugador a no ser que el jugador esté cerca.

Figura 3.3: Enemigo Rana en Cave Story

3.2.2. Enemigo murciélago

En la Figura 3.4 se puede ver un enemigo sencillo, un tipo de murciélago [4] que aparece en el primer nivel del videojuego Cave Story. Este enemigo vuela moviéndose verticalmente entre dos puntos fijos: el punto más alto del movimiento y el punto más bajo. Este enemigo se detiene un breve lapso de tiempo en los extremos, y debido a que es un enemigo de los primeros niveles no se mueve de forma brusca, ni ataca de forma activa al jugador.

Para construir este comportamiento con los atributos de la Figura 3.1, se utilizaría el comportamiento Waver porque el movimiento de este personaje describe un movimiento sinusoidal vertical. Este enemigo no cuenta con una condición de activación, sino que se moverán aunque el jugador no se encuentre inmediatamente al lado de los murciélagos.

No obstante, no todos los enemigos de tipo murciélago son tan sencillos. A medida que el jugador avanza en el videojuego se encontrará con ciertos tipos de enemigos que, aunque visualmente sean parecidos tienen movimientos más complejos e incluso incorporan formas de atacar de forma activa al jugador. Un ejemplo de ello son los murciélagos que se encuentran en la imagen 3.3, ocultos debajo de la rana. Estos murciélagos se diferencian por el color, ligeramente más oscuro que los de la Figura 3.4, y que en lugar de flotar permanecen quietos hasta que detectan que el jugador ha pasado por debajo de ellos. Cuando eso sucede, abandonan su posición tras un breve retardo, y se lanzan en picado para atacar al jugador. En la Figura 3.5, se puede apreciar al murciélago precipitarse hacia el jugador porque este está situado justo debajo. Al igual que la rana, estos enemigos dependen de información del entorno para poder llevar a cabo su funcionalidad completa, es necesario que cuenten

[4] `https://cavestory.fandom.com/wiki/Bat`

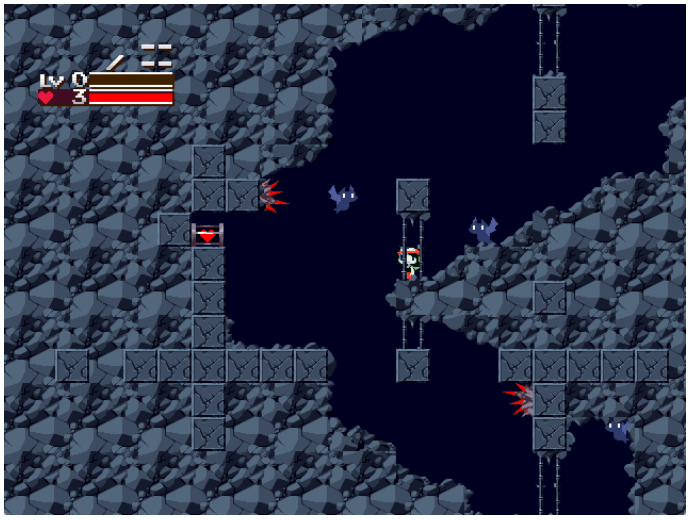


Figura 3.4: Enemigo Murciélago en Cave Story

con un elemento sensorial que responda a la pregunta "¿El jugador ha pasado por debajo de mi posición?". Este enemigo se construiría mediante un comportamiento de tipo Faller, acorde a la lista de la Figura 3.1, porque cae desde el techo al suelo pero tiene la condición de sensor de "línea directa de visión con el enemigo", que aparece en la Figura 3.2.

Figura 3.5: Enemigos Murciélago y Critter en Cave Story

3.2.3. Critter

Además de los murciélagos negros, se puede ver otro tipo de enemigo en la Figura 3.5 que se asemeja a una bola verde que aparece en el aire encima del jugador. Estos enemigos se denominan Critters [5] y, al igual que la rana, avanzan saltando hacia la posición del jugador. No obstante, estos enemigos no caen inmediatamente sino que al llegar a su punto más alto comienzan a "volar" hacia el jugador manteniéndose unos instantes en el aire antes de caer. Este movimiento sugiere que es una combinación de componentes: por un lado, se requeriría el componente Jumper que ya se ha visto previamente pero por otro lado, se le debería añadir un componente Floater para que flote antes de caer. No obstante, no sería necesario añadir un componente Faller de la lista de la Figura 3.1 ya que la caída

[5] `https://cavestory.fandom.com/wiki/Critter`

tras el salto es una parte intrínseca del componente Jumper que describe Bright (2014) en su artículo. Estos enemigos no saltan todo el rato, sino que cuando detectan que el jugador está cerca se activan y comenzarán a moverse hacia el jugador pero si este se aleja lo suficiente estos enemigos volverán a su estado inactivo.

3.2.4. Behemoth

Los Behemoth [6] son criaturas parecidas a un elefante que aparecen en la Figura 3.6. Estos enemigos avanzan en línea recta y cambian el sentido cuando se encuentran con una pared u otro obstáculo que les impide seguir avanzando. No son unidades hostiles, y no atacan al jugador a no ser que este les dispare primero, en ese caso cambian de color marfil al rojo y cambia su comportamiento. Cuando este enemigo está enfurecido, en lugar de moverse lateralmente intentará cargar hacia el jugador a mayor velocidad, pero seguirá sin ser capaz de superar obstáculos del entorno.

Figura 3.6: Enemigo Behemoth en Cave Story

Para crear a este enemigo con los atributos de Bright (2014), se puede emplear un comportamiento de tipo Pacer, dado que este comportamiento se describe como un movimiento que cambia de dirección, aunque el ejemplo que se utiliza en la descripción (Figura 3.1) está mas orientado a no caerse de las plataformas. Debido a esa especificación, en lugar de un comportamiento Pacer sería más correcto definirlo como un comportamiento de tipo Ponger. Este enemigo tiene un evento que modifica su comportamiento que no figura en la lista de la Figura 3.2 de Bright (2014) y ese cambio de estado se da cuando la vida de la unidad baja de cierto porcentaje sin ser derrotado. Cuando esto sucede, se modifican los parámetros de los componentes previos para hacerlos más rápidos o hacer que vayan a por un objetivo particular, convirtiendo al enemigo en una amenaza mayor.

3.3. Diseño de Componentes

Teniendo en cuenta lo que se ha analizado acerca del artículo de Bright (2014), y utilizando su estudio como inspiración, se procede a diseñar los componentes de este catálogo

[6] `https://cavestory.fandom.com/wiki/Behemoth`

enfocado a enemigos 2D y finalmente se construirán enemigos del videojuego Cave Story comentados en las subsecciones anteriores con los componentes diseñados.

Los componentes con los que va a contar este catálogo se podrían agrupar en dos apartados fundamentales: comportamientos de movimiento, que además deberán ser combinables, y atributos sensoriales que puedan activar o desactivar estos componentes de movimiento.

Estos componentes debían ser sencillos, concisos, y diseñados de tal manera que cada uno realice una sola acción. A continuación, se definirán brevemente cada uno de los apartados de este catálogo y se resumirán los componentes diseñados para cada uno de ellos. En las listas que resumen el diseño final se incluirá el nombre del componente además de una breve descripción de lo que se podría esperar de cada uno de los componentes para ayudar a visualizar el comportamiento que aportarán a la entidad. Los componentes de este catálogo se explicarán con más profundidad en el Capítulo 4, detallando cómo han sido implementados para que formen parte del catálogo y listando los atributos que el usuario puede modificar para personalizar el movimiento.

3.3.1. Componentes de Movimiento

Los componentes de movimiento sirven para darles atributos de movimiento específicos a las entidades que posean el componente. Es recomendable que cada componente de movimiento tenga una serie de atributos de ese comportamiento para personalizarlo. Estos atributos deberán ser sencillamente números y otras opciones asociadas al comportamiento, que sea simple a la par que permita la personalización individual de cada uno de los componentes. Además, estos componentes deberían ser sencillos y se deberían poder combinar para crear comportamientos nuevos. El diseño final de los componentes de movimiento es el siguiente:

- **Floater**: Flota entre dos puntos, horizontal o verticalmente.
- **Faller**: Cae en picado hacia abajo
- **Jumper**: Salta periódicamente hacia arriba en línea recta.
- **Bullet**: Avanza en línea recta en una dirección.
- **Forward Jumper**: Salta hacia delante en la dirección del jugador.
- **Liner**: Avanza hacia un objetivo en línea recta.
- **Follower**: Persigue a un objetivo.
- **Pacer**: Avanza por una plataforma sin caerse. Al llegar al borde, cambia de sentido.
- **Bumper**: Avanza en línea horizontal y, si detecta un obstáculo cambia de sentido.
- **Orbit**: Orbita alrededor de un punto dado.
- **Wander**: Se mueve aleatoriamente dentro de un área.

Además de los componentes de movimiento, este catálogo de componentes debería contar con atributos de tipo sensor que reaccionen ante determinados eventos y activen o desactiven los componentes de movimiento para darle profundidad a la herramienta.

3.3.2. Componentes sensoriales

Los sensores dan al jugador la sensación de que la entidad enemiga está viva y reacciona ante lo que percibe en el mundo. La mayoría de los enemigos modifica su comportamiento cuando el jugador se acerca a menos de una determinada distancia, por lo que integran sensores de proximidad. Lo que se busca es el efecto de que el enemigo se encuentra alerta debido a que considera al jugador como una amenaza, y que supone un comportamiento perfectamente natural. No obstante, un sensor de proximidad también cumple una función de ahorro de recursos, porque es ineficiente tener cosas en movimiento que el jugador no va a poder apreciar.

- **Area Sensor**: Define un área donde se detectará si ha entrado el jugador.
- **Line Sensor**: Define la línea de visión del enemigo.
- **Range Sensor**: Mide la proximidad con el jugador.

Estos componentes de movimiento y sensoriales son los que conforman la herramienta, son los elementos que constituyen este catálogo de componentes con los que es posible crear enemigos para un videojuego 2D. No se puede negar que este trabajo ha tomado como principal inspiración el artículo de Bright (2014) que se comentaba previamente y, por tanto, existen algunas similitudes con las tablas que aparecen en las Figuras 3.1 y 3.2 pero este catálogo también cuenta con comportamientos modificados y algunos de ellos son totalmente nuevos.

3.4. Diseño de NPCs usando este catálogo

Una vez se han definido los comportamientos que va a tener nuestra herramienta, se va a definir con ella los movimientos de los enemigos descritos en las secciones anteriores, construyéndolos utilizando estos componentes que se han diseñado. La rana de la Figura 3.3 se podría definir utilizando los comportamientos descritos como una entidad con comportamiento Forward Jumper. Debido a que, conceptualmente, Jumper sólo salta hacia arriba, sería necesario un movimiento lateral si se quisiera que avance hacia delante. Con este comportamiento, y un sensor de proximidad de tipo Area o Range Sensor se conseguiría que este enemigo salte hacia el jugador pero que sólo lo haga cuando el jugador se acerca demasiado.

Posteriormente se trataba uno de los enemigos iniciales, los murciélagos de la Figura 3.4. Este enemigo se construiría con un comportamiento Floater, para que se mueva verticalmente entre dos puntos. Gracias a la combinación de comportamientos que permite este catálogo, si a ese movimiento constante entre dos puntos se le añadiera un movimiento en un eje, con un comportamiento Bullet, se conseguiría un enemigo que avance de forma sinusoidal en la dirección que se desee, describiendo un movimiento parecido al Waver que propone Bright (2014) (Figura 3.1).

Se mencionaban también a los murciélagos oscuros de la Figura 3.5, que se lanzan a por el jugador al tener "contacto visual" con el mismo. Este enemigo se construiría con un componente Faller que se active cuando el componente sensorial Line Sensor detecte al jugador en una línea vertical hacia debajo. Por otra parte, el Critter que también aparece en la Figura 3.5, resultaría una mezcla de Forward Jumper combinado con un movimiento

Faller, debido a que el comportamiento Faller es capaz de controlar cuando debe la entidad precipitarse al suelo. Además, es necesario un sensor de proximidad para saber cuándo el jugador está demasiado cerca para empezar el movimiento, simulando el comportamiento del juego original.

Finalmente, se definió el comportamiento del Behemoth, el enemigo con aspecto de elefante de la Figura 3.6. Utilizando los componentes del catálogo que se ha diseñado, se puede definir este comportamiento como un comportamiento de tipo Bumper debido a sus cambios de sentido al colisionar con las paredes. Un enemigo que posea un comportamiento de este tipo no necesita un sensor que le informe del evento de colisión con una pared porque se considera una parte intrínseca del comportamiento y, por tanto, no es responsabilidad del diseñador crear esa particularidad del movimiento mediante sensores sino que se gestionará de forma interna.

En resumen, en los videojuegos 2D resulta casi indispensable tener enemigos, y para que haya enemigos tiene que existir una serie de componentes que les de vida. Además, a medida que avanza el juego es muy recomendable que sean más complejos, por lo que tienen que incluir más comportamientos y combinarlos. Tener enemigos con dificultad progresiva es un obstáculo que hay que afrontar si se quiere hacer un videojuego retro o *neo-retro* y, como se ha mostrado de forma teórica gracias a este catálogo de movimientos resulta sencillo hacer enemigos funcionales y personalizables para cualquier videojuego 2D de avance lateral.

Capítulo 4

Implementación

En el Capítulo 3 se comentaba de dónde surge la herramienta, se definieron los comportamientos que iba a ofrecer el catálogo de componentes y, finalmente, se definieron unos enemigos comunes. En este capítulo se va a tratar en profundidad la implementación de los componentes diseñados, entrando en detalles acerca de las formas de personalizar cada componente y las distintas combinaciones de comportamientos que ofrece este catálogo.

4.1. Tecnología utilizada

Este proyecto se ha desarrollado de forma íntegra con el *motor de videojuegos* Unity, una herramienta multiplataforma creada por la empresa *Unity technologies* en el año 2005. La versión escogida para desarrollar la herramienta es la 2019.2.1f, y por tanto no se garantiza que la herramienta funcione en una versión anterior, pero si debería funcionar sin problemas en versiones posteriores.

Unity es una herramienta muy versátil, moderna, y potente que es capaz de renderizar desde entornos simples en dos dimensiones hasta entornos de gran tamaño en tres dimensiones, pasando por la realidad virtual y la realidad aumentada. Este *motor de videojuegos* surge con la idea de hacer el desarrollo de videojuegos más accesible a todo tipo de desarrolladores, tanto profesionales en la industria como amateurs autodidactas, ofreciendo soporte para varios lenguajes de programación en forma de *plugins* a pesar de contar con el lenguaje C sharp y Javascript como principales lenguajes de programación de *scripts*.

Unity resulta muy atractivo de utilizar porque cuenta con muchas facilidades a la hora de desarrollar aplicaciones de todo tipo, no solamente sirve para videojuegos sino también es muy utilizado en simuladores, para cine o para arquitectura. Unity cuenta con una interfaz de usuario muy gráfica, intuitiva dentro de su complejidad, y fácilmente personalizable al gusto de cada uno. Además, este *motor de videojuegos* tiene integrado un sistema de arrastrar y soltar para construir las escenas de juego, mover a voluntad los objetos dentro de ella y asignar *scripts* a las entidades de juego para darles vida. Este sistema ya aparecía en los *motores de videojuegos* que se han comentado en el Capítulo 2, y es lo que permite que Unity sea empleado por un gran número de personas. La curva de aprendizaje no es muy pronunciada ya que los creadores han proporcionado una documentación muy extensa, tutoriales para aprender a utilizar el motor así como unos foros donde la comunidad puede preguntar, y responder dudas a otros usuarios de la comunidad.

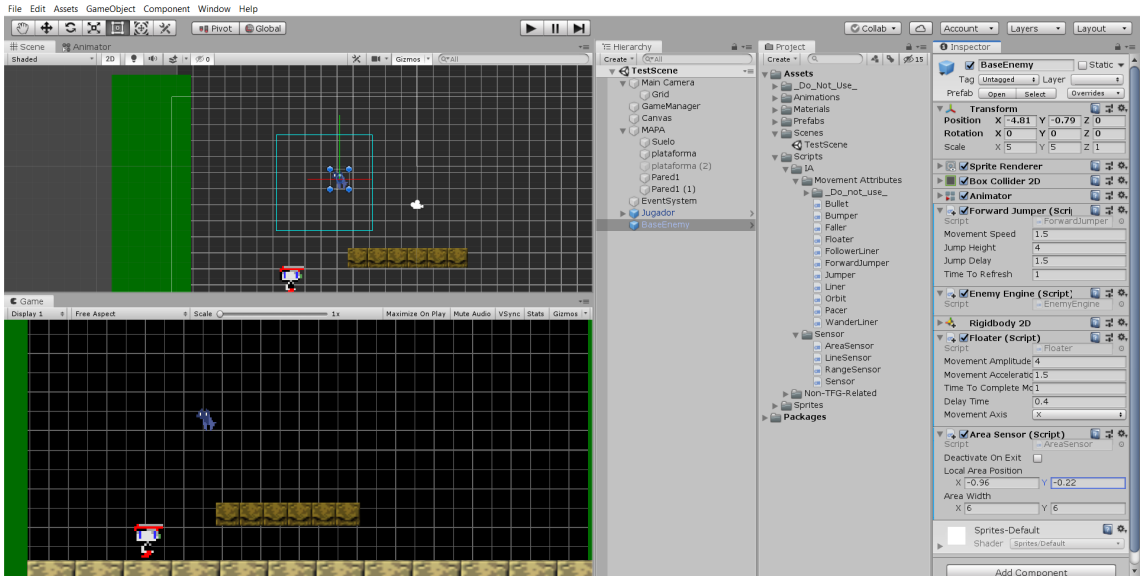


Figura 4.1: Interfaz de Unity *(2 by 3)*, 2020

El motivo por el que se escogió Unity frente a otros *motores de videojuegos* es precisamente por la accesibilidad y lo sencillo que resulta de utilizar por cualquier persona. La herramienta que se ha desarrollado se fundamenta en Unity porque a diario la utilizan muchas personas de distintos niveles de conocimiento y la idea del catálogo de enemigos es que llegue al mayor número de personas posibles, especialmente a gente que no necesariamente sepa programar o no tenga los recursos necesarios para construir toda la inteligencia artificial de los enemigos desde cero.

El sistema de arrastrar y soltar con el que cuenta el motor es muy útil para el uso de la herramienta porque hace que añadir y combinar los diferentes *scripts* de movimiento sea una tarea sencilla, al alcance de cualquier usuario del motor con un mínimo conocimiento. Además, Unity cuenta con una gran cantidad de elementos que han ayudado a hacer la herramienta, como por ejemplo su motor de físicas 2D integrado o las herramientas que tiene para ayudar visualmente al usuario a hacerse una idea de lo que va a pasar al añadir un *script* específico gracias a sus *gizmos*.

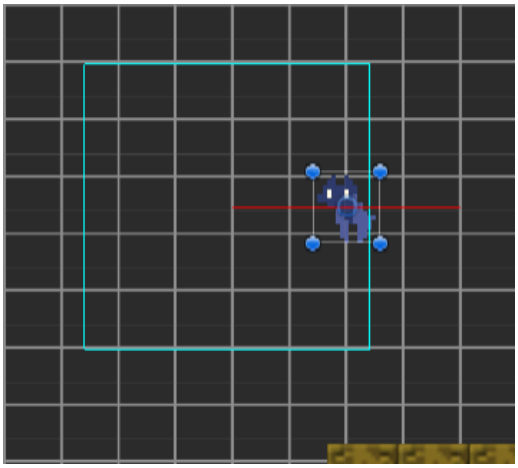


Figura 4.2: Ejemplo de gizmos

Otro punto a favor para utilizar este *motor de videojuegos* frente a otros motores previamente comentados es que Unity funciona con una arquitectura de código por componentes lo que lo hace perfecto para integrar esta herramienta. El estilo de arquitectura basado en componentes pone énfasis en la descomposición de un sistema en elementos más pequeños de código para fomentar la *modularidad* del programa. Los componentes programados deben ser funcionales y presentar una interfaz que sirva para *abstraer* la implementación privada de la parte personalizable de un componente, que será pública. De esta manera, es posible definir en Unity un apartado de componentes personalizado (Figura 4.3) donde guardar los componentes propios de la herramienta de tal manera que quede totalmente aislado del código fuente que hace que funcione, además de que así están en un lugar que resulta intuitivo de encontrar.

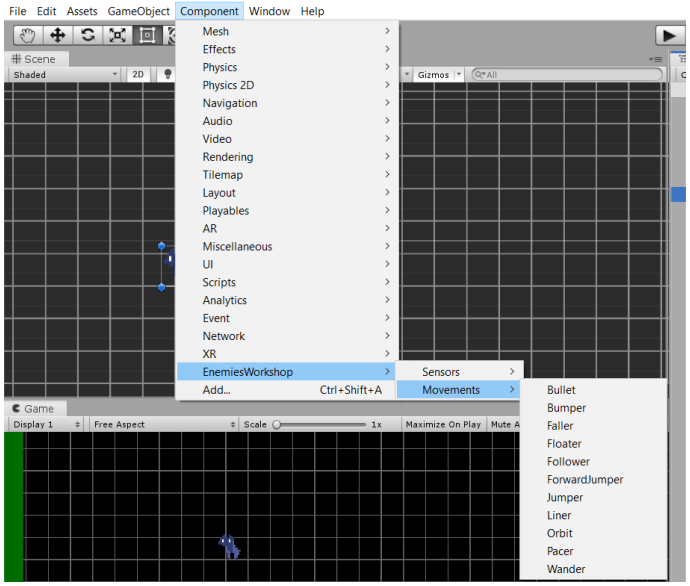


Figura 4.3: Interfaz de la herramienta

4.2. Infraestructura básica

Para llevar a cabo el diseño de este catálogo de componentes e implementar los comportamientos de movimiento y atributos sensoriales era indispensable poner la herramienta en el contexto de un videojuego 2D. Utilizando una serie de componentes y objetos se creó una simulación sencilla de un nivel donde había un enemigo, una plataforma y el jugador.

La clase Player Controller que sirve de controlador del jugador. Esta clase es la encargada de mover a la entidad jugador o hacer que salte cuando se invocan los métodos determinados, pero por si sola no es capaz de interpretar el input del jugador sino que sólo contiene los métodos que implementan las acciones de movimiento posibles para el jugador. La clase *PlayerMovement* es una clase que permite al usuario controlar el movimiento del jugador debido a que recoge el input del usuario, y utiliza los métodos definidos en Player Controller con la acción deseada. Permite, además, modificar ciertos valores como la velocidad de movimiento o la potencia del salto. Como no es una parte intrínseca de este catálogo he creado el código siguiendo un tutorial de un usuario de la comunidad de Unity, Brackeys [1].

[1] `https://www.youtube.com/watch?v=dwcT-Dch0bA`

La clase Camera Controller de la Figura 4.4 es una clase simple que, añadida a un objeto cámara hace que este objeto que mueva para mantener a una entidad objetivo siempre visible. El objeto al que la cámara debe seguir es personalizable, pero para esta implementación se ha establecido que el jugador sea el objeto al que siga la cámara.

Se ha implementado también una clase *Health* que gestiona la vida de la unidad a la que se le aplicase, y que permite eliminar entidades de la escena cuando su vida llegue a 0. Esto se crea para probar un componente de ataque sencillo que es el daño por contacto. La clase *Contact Damage* controla la colisión de la entidad con el entorno, y gestiona que si la entidad que colisiona con el jugador era un enemigo, le enviara un mensaje indicándole al componente *Health* del jugador que ha sido dañado como se ve en la Figura 4.4. Para el desarrollo del catálogo se ha decidido hacer más énfasis en los comportamientos de movimiento que en los posibles comportamientos de ataque, por lo que estas clases han sido relegadas a un segundo plano.

La clase Enemy Engine que aparece en la Figura 4.4 sirve como gestor de información importante de cada enemigo. Es una clase que se añade automáticamente al añadir cualquier comportamiento de movimiento y que sirve para guardar una referencia de la posición del jugador, constantemente actualizada y que puede ser requerida por cualquier componente de movimiento mediante la clase Movement Behaviour. La clase Enemy Engine es capaz de buscar las capas concretas del suelo, o del jugador, dentro de las capas físicas del *motor* y devolver una referencia. Mediante el uso de estas capas, por ejemplo, los enemigos saben si han colisionado con el suelo y los sensores son capaces de detectar al jugador para activar los componentes. Actualmente las capas que sabe detectar esta clase son las capas físicas "*Player*" para el jugador, y "*Ground*" para el suelo, porque son capas físicas muy comunes en cualquier videojuego.

Finalmente, en la Figura 4.4 aparecen dos módulos con los componentes de movimiento y de tipo sensor. Se crearán dos jerarquías correspondientes, con los componentes implementados en cada una de ellas. Cada jerarquía tiene como raíz una clase que les aporta funcionalidades comunes, y de la que heredarán los componentes. Esto se tratará en detalle en las siguientes secciones.

4.3. Componentes de Movimiento

Los componentes de movimiento son *scripts* que, al aplicárselos a una entidad, generan un tipo de comportamiento. Los componentes están hechos cuidando la *abstracción*, teniendo en cuenta que el usuario final no tiene por qué saber cómo están hechos los *scripts*. Funcionan de tal manera que no haga falta añadir ni configurar nada que no sean los atributos públicos que cada componente ofrece, por lo que basta con añadir el componente y ejecutar la aplicación para que la entidad empieze a moverse.

Como podemos ver en la Figura 4.5, todos los componentes de movimiento son hijos de la clase Movement Behaviour. Esta es una clase común a todos los comportamientos de movimiento, lo que quiere decir que cualquier clase que defina un movimiento es una clase *hija* de esta clase. Mediante esta herencia, las clases hijas son capaces de obtener atributos y métodos de las clases padre. Concretamente, la clase Movement Behaviour declara un método que heredan todos los componentes de movimiento, por el cual cada componente puede devolver su vector de movimiento si estuviera actuando por si mismo pero sin mover a la entidad. Esto es muy útil para las clases que combinan varios componentes de forma interna, como es el caso del Forward Jumper que se diseñó en el Capítulo 3.

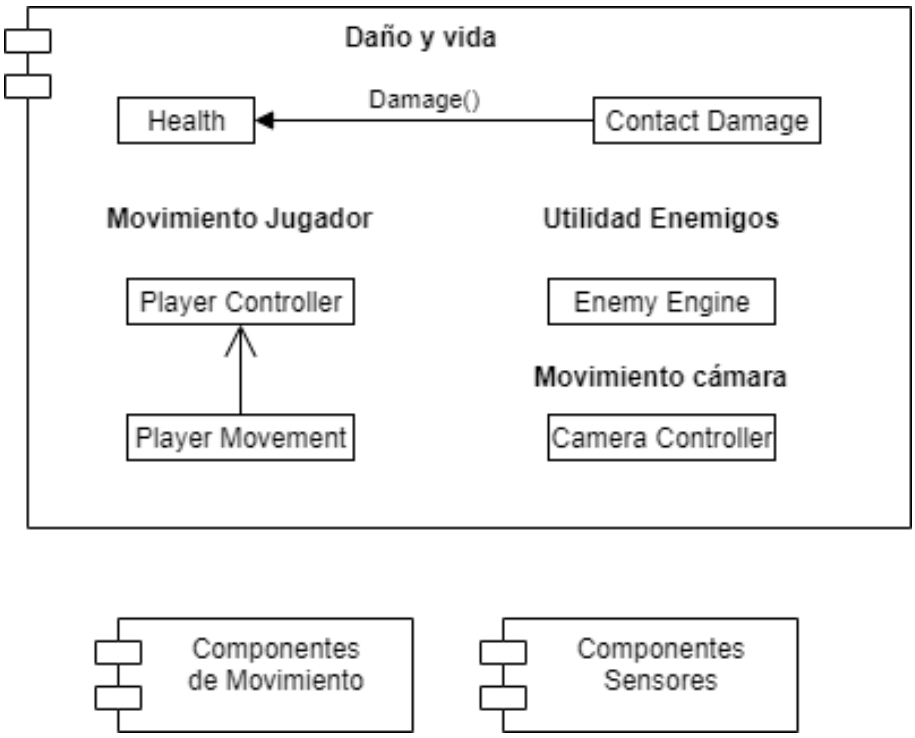


Figura 4.4: Esquema general

La clase Movement Behaviour requiere un componente de tipo Enemy Engine para funcionar, de esta manera existe la garantía de que la entidad cuenta con esta clase porque al añadir cualquier componente de movimiento, se comprobará si la entidad tiene un Enemy Engine y en caso de no tenerlo lo añadirá automáticamente. Guardando la referencia al Enemy Engine, las clases de movimiento pueden preguntar por la posición del jugador o por una capa física concreta como se ha explicado previamente.

La estructura que siguen los componentes de Unity viene determinada por una serie de funciones que son llamadas automáticamente en lo que es llamado el "ciclo de vida" del *script*. Los componentes de movimiento desarrollados para la herramienta, por tanto, siguen la misma estructura. Al iniciar la aplicación, se llamará al método *Start* que construye la parte interna del componente, gestionando las dependencias y montando el comportamiento interno con los valores públicos. No obstante, si el enemigo cuenta con algún tipo de sensor no se llamará a ese método al iniciarse sino cuando el sensor los active al detectar al jugador. Cada vez que se ejecuta un ciclo de juego, lo que se conoce como *frame*, se llama a un método llamado *update* que actualiza la lógica de cada componente de movimiento activo. Es en este método cuando los componentes llaman a sus funciones privadas de gestión de movimientos, y dependiendo de la estructura del enemigo moverán la entidad, o devolverán el movimiento generado a otro comportamiento si este se lo pidiera.

4.3.1. Floater

El componente Floater se caracteriza porque desplaza a la entidad entre dos puntos a lo largo de un eje, el horizontal o el vertical, siguiendo un patrón fijo de movimiento. El enemigo que tenga este comportamiento se moverá con una velocidad calculada en función de la distancia hacia el punto de destino y el tiempo total en completar el movimiento,

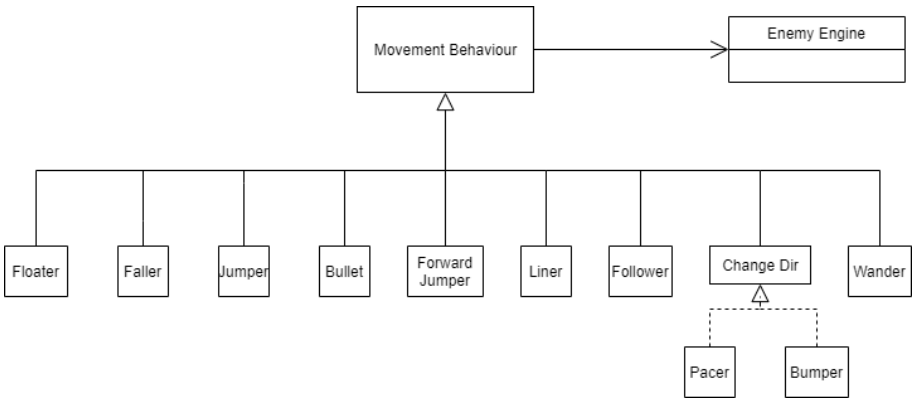


Figura 4.5: Esquema de los componentes de movimiento

de tal manera que va frenando a medida que se aproxima a los puntos finales. Además del tiempo total que tarda en completar el movimiento es posible definir un tiempo extra de retardo tras completar un sector del movimiento. Por tanto, si se desea la entidad se detendrá una cantidad fija de tiempo en esos límites antes de reanudar el movimiento.

Se desplaza, por tanto, entre dos puntos en el espacio que se definen como las dos mitades desde el punto del objeto, que deberá situarse en el punto medio de la ruta. Como esta condición puede resultar compleja de entender, se ha creado una ayuda visual para este componente como se ve en la Figura 4.6. Esta ayuda visual representa la amplitud del movimiento que va a seguir, trazando el recorrido total que va a efectuar la unidad y, además, se actualiza si se modifica el valor.

Atributos de configuración

- (Float) Movement Amplitude: Amplitud de onda. Distancia total que quieres cubrir en unidades de Unity.
- (Float) Time to Complete Movement: Tiempo que tarda en llegar al extremo al que se deba dirigir.
- (Float) Delay Time: Tiempo que espera en los límites del movimiento.

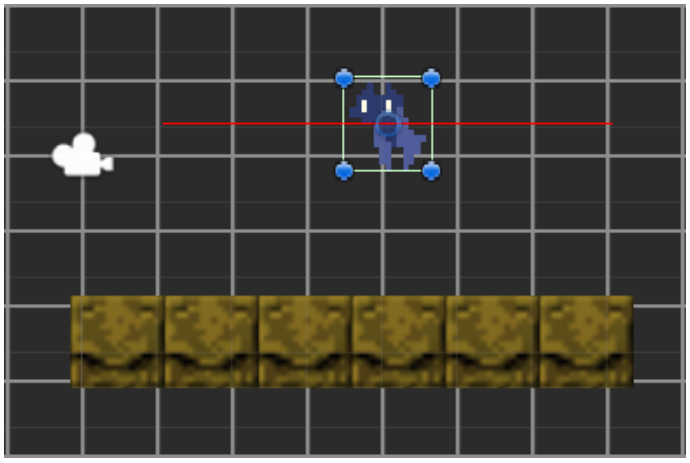


Figura 4.6: Componente Floater

4.3.2. Faller

Un enemigo con el componente Faller tiene un movimiento vertical hacia abajo, impulsado por la masa del enemigo y por la "escala de gravedad" que se le aplicará. Si se desea, el efecto de caída libre no se activará inmediatamente, existe una opción para definir un retardo en la caída porque si cayera inmediatamente el jugador podría no ver al enemigo a tiempo y no tener tiempo a reaccionar. Una vez pasado ese tiempo, se dejará caer el objeto aplicándole una fuerza hacia abajo sin modificar el peso de la entidad.

No se debe confundir este comportamiento de caída libre con un comportamientos de habilidad de "lanzar elementos", porque se entiende el comportamiento Faller como que es la entidad la que se lanza con su propio cuerpo en un movimiento descendiente en lugar de generar objetos que después puedan caen por gravedad, utilizando el mismo comportamiento Faller.

Atributos de configuración

- (Int) Gravity: Fuerza hacia abajo que se le aplicará a la entidad, resultando en la velocidad con la que cae.
- (float) Time before fall: Tiempo, en segundos, que espera antes de caer.

4.3.3. Jumper

El comportamiento Jumper hace que la entidad salte, impulsándola hacia arriba con una altura máxima personalizable. El salto está programado mediante físicas: se le aplica un incremento de velocidad hacia arriba con una fuerza inicial calculada internamente con la altura máxima indicada en el editor. Se añadió posteriormente, ajustándome a los comentarios de los probadores, una variable que controla el tiempo que tarda la entidad en llegar al punto más alto.

Mediante la variable de la altura máxima, el diseñador puede hacerse a la idea de lo alto que va a saltar la entidad con los gizmos que posee el componente. Como se ve en la Figura 4.7, la línea indica tanto el punto más alto como el recorrido que va a hacer el enemigo. Además el Jumper cuenta con un atributo que controla el retardo que existe entre saltos, y que empezará a contarse cuando la entidad *Jumper* haya entrado en contacto con el suelo. Si el punto más bajo de la entidad *Jumper* no está en contacto directo con el suelo no saltará.

Atributos de configuración

- (Float) Jump Height: Lo mucho que salta la entidad.
- (Float) Jump Time: Tiempo que tarda la entidad en llegar al punto más alto.
- (Float) Jump Delay: Tiempo que la entidad espera antes de saltar de nuevo.

4.3.4. Bullet

El comportamiento Bullet permite a una entidad avanzar en línea recta en una dirección. La dirección del movimiento viene dada por el vector Bullet Direction que, como

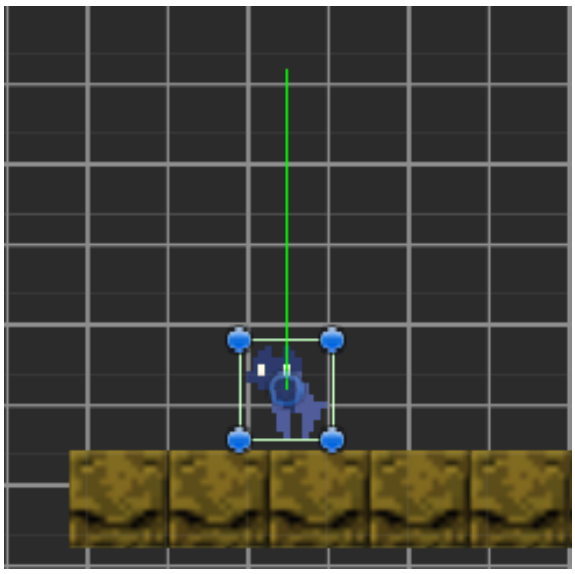


Figura 4.7: Componente Jumper

se ve en la Figura 4.8 se representa como una línea en la dirección elegida. La línea está normalizada por claridad, para no llenar la pantalla de gizmos, pero el movimiento será en línea recta de forma indefinida. Es posible seleccionar si se desea que el enemigo colisione con las paredes o que, por otro lado, las atraviese. Por último, se puede personalizar la velocidad a la que se mueve la entidad que posea este comportamiento.

Atributos de configuración

- (Bool) Collide with walls: Determina si colisiona o no con el entorno.
- (Vector 2) Bullet direction: Vector que define la dirección del movimiento.
- (Float) Bullet speed: Velocidad a la que se mueve el objeto en unidades de Unity por segundo.

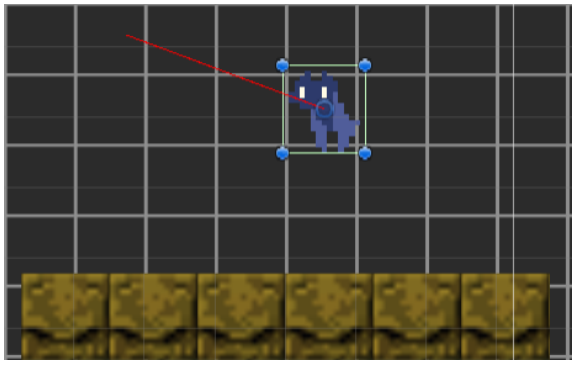


Figura 4.8: Componente Bullet

4.3.5. Forward Jumper

Este comportamiento es el resultado de combinar un *Jumper* con un *Bullet* de forma interna. Tiene todos los atributos con los que luego se construyen los componentes necesa-

rios, sin que el diseñador tenga que montar la estructura en el editor. La combinación de Jumper con Bullet permite hacer un movimiento particular: la entidad salta en una dirección diagonal, dando la sensación de que salta avanzando hacia delante. El salto siempre se orienta hacia el jugador, y la entidad espera una breve cantidad de tiempo entre saltos, que al igual que pasaba con el *Jumper* sólo empezará a contar el tiempo cuando la parte más baja de la entidad esté en contacto con el suelo.

Atributos de configuración

- Cuenta con los tributos de un Jumper y un Bullet.
- (Float) Delay between jumps: Establece el retardo entre saltos, en segundos.

4.3.6. Liner

El comportamiento Liner se desplaza a la entidad a una velocidad definida hasta alcanzar un objetivo, orientando previamente su dirección hacia el mismo. La velocidad de movimiento o de aceleración vendrán definidas por el diseñador. Es importante destacar que el enemigo no se va a desplazar en esa dirección constantemente, como un Bullet, sino que va a moverse hacia un punto fijo en el espacio en línea recta y una vez llegue a ese punto el movimiento se detendrá. Además, se le ha dado al componente Liner la funcionalidad de rotar el objeto para orientarse hacia el objetivo, cosa que no sucede con el componente Bullet.

Originalmente este comportamiento viajaba hacia el jugador, no obstante siguiendo con los comentarios extraídos en las pruebas el diseñador tiene la opción de seleccionar un objetivo personalizado. En caso de que el diseñador no establezca un objetivo se establecerá de objetivo del movimiento al jugador por defecto.

Atributos de configuración

- (Game Object) Target: Objetivo del movimiento. En caso de ser vacío al empezar la partida, se asignará el jugador.
- (Float) Time to reach target: Tiempo en segundos que le tomará al entity llegar al punto objetivo.
- (Bool) Rotate towards target: Si está activo, el Liner rotará su sprite para orientarlo hacia el punto objetivo.
- (Enum) Tipo movimiento: Enumerado que determina si el tipo de movimiento es continuo o acelerado. En función de esta decisión se usará un gestor de movimiento u otro

4.3.7. Follower

El componente Follower permite al enemigo actualizar la posición de objetivo cada cierto tiempo, proporcionando así un nuevo punto objetivo al movimiento Liner interno y evitando así que se detenga. Con esta sencilla modificación se consigue que un enemigo persiga constantemente a su objetivo. De la misma manera que sucedía con el comportamiento Liner, si el usuario no asigna un objetivo a este componente se establecerá el

jugador por defecto. Además, el usuario puede modificar el tiempo de refresco o renovación de la posición del jugador para que sea más o menos constante. Por ejemplo, con un valor bajo se consigue que la entidad se mueva de forma fluida hacia su objetivo mientras que con un valor más alto el efecto se asemeja más a que el enemigo carga hacia la posición del jugador.

Atributos de configuración

- Como es una modificación del atributo Liner, cuenta con todos sus atributos.
- (Float) Time to Refresh: Tiempo que el componente espera antes de actualizar la posición.

4.3.8. Wander

El comportamiento Wander hace que la entidad deambule sin rumbo fijo dentro de un área determinada. Este componente tiene de forma interna un Liner, pero no permite que se le asigne ningún objetivo. En su lugar, cada cierto tiempo configura su movimiento Liner interno con una nueva posición generada dentro de área circular de este componente. El usuario puede establecer tanto las dimensiones de ese área como el tiempo que tardará la entidad en elegir un nuevo objetivo.

Atributos de configuración

- Como es una modificación del atributo Liner, cuenta con todos sus atributos salvo el target.
- (Float) Area Radius: Radio de la esfera en la que se moverá.
- (Float) Time to Refresh: Tiempo que el componente espera antes de actualizar la posición.

4.3.9. Pacer

El comportamiento Pacer hace que la entidad avance en línea recta, en la dirección hacia la que mire, a lo largo de una superficie. Cuando este componente detecta que delante de la entidad no hay suelo firme sobre el que caminar, el componente cambiará de sentido rotando a la entidad volviendo sobre sus pasos. Debido a esto, este comportamiento se suele utilizar en plataformas elevadas.

Para este movimiento, el componente cuenta con un sensor interno que consiste en un rayo físico que se lanza hacia debajo desde una posición ligeramente más adelantada que la de la entidad. Mientras que este sensor detecte que hay suelo delante la entidad seguirá avanzando pero si, por el contrario, el sensor no detectase suelo el enemigo se daría la vuelta.

Atributos de configuración

- (Float) Movement speed: Velocidad en unidades de Unity por segundo a la que se mueve la entidad en línea recta.

4.3.10. Bumper

Una entidad con un componente Bumper se mueve linealmente en la dirección en la que mire. Cuando este componente detecta que ha colisionado con un obstáculo, cambiará de sentido.

El componente Bumper cuenta con un sensor interno situado en la parte delantera del objeto, con una longitud personalizable. Este sensor se activará cuando colisione con un obstáculo, cambiando el sentido del movimiento y haciendo que la entidad vuelva sobre sus propios pasos. Al contrario de lo que sucedía en el comportamiento Pacer este componente sí permite a la entidad caer desde una plataforma e incluso precipitarse al vacío.

Atributos de configuración

- (Float) Movement speed: Velocidad en unidades de Unity por segundo a la que se mueve la entidad.
- (Float) Detection range: Rango en unidades de Unity en la que se va a revisar si se ha colisionado con un obstáculo.

4.3.11. Orbit

Un enemigo con el componente Orbit girará en torno a un punto fijo con un sentido personalizable. Este giro se efectúa cuando el enemigo se encuentra en el extremo del área del círculo, cuya posición y radio se puede modificar fácilmente en el editor como se muestra en la Figura 4.9. Si el enemigo se encuentra más alejado del círculo, se moverá hacia él mientras gira, de nuevo en el sentido elegido, hasta entrar en el extremo del área del circulo. La velocidad a la que el enemigo va a moverse hacia la órbita, en el caso de se sitúe al objeto fuera del área, viene dada por la velocidad de atracción. Por otra parte, la velocidad dentro de la propia órbita viene determinada por la velocidad de rotación.

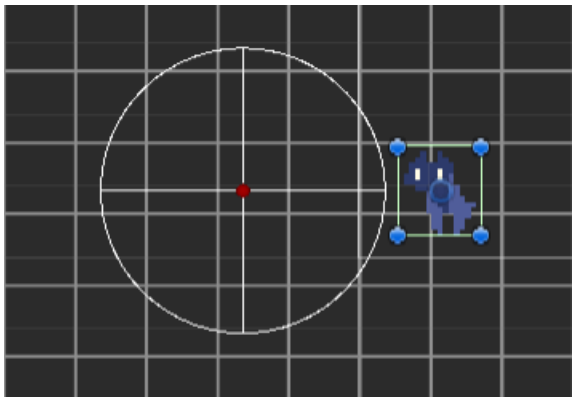


Figura 4.9: Componente Orbit

Atributos de configuración

- (Vector2) Orbitational center: Centro del movimiento orbital
- (Float) Radius: Radio de la esfera sobre la que se orbitará.
- (Float) Attraction speed: Velocidad a la que la entidad se aproxima a la órbita

- (Float) Rotation speed: Velocidad a la que la entidad se mueve por la órbita.
- (Bool) Rotate clockwise: Determina si el objeto girará en el sentido horario, o anti-horario.
- (Bool) Collide with terrain: Determina si el objeto colisiona con su entorno o no.

4.3.12. Combinación de componentes

La mayoría de componentes se mueven de forma física, mediante el uso de fuerzas, y ello les permite combinarse. Algunos comportamientos funcionan muy bien juntos de forma natural, mientras que otros componentes pueden trabajar juntos pero con una serie de limitaciones, teniendo un componente que los coordine. No obstante, algunas combinaciones pueden no tener sentido desde el punto de vista del diseño de un enemigo, porque dan lugar a movimientos poco naturales u óptimos, o que sencillamente no se pueden combinar.

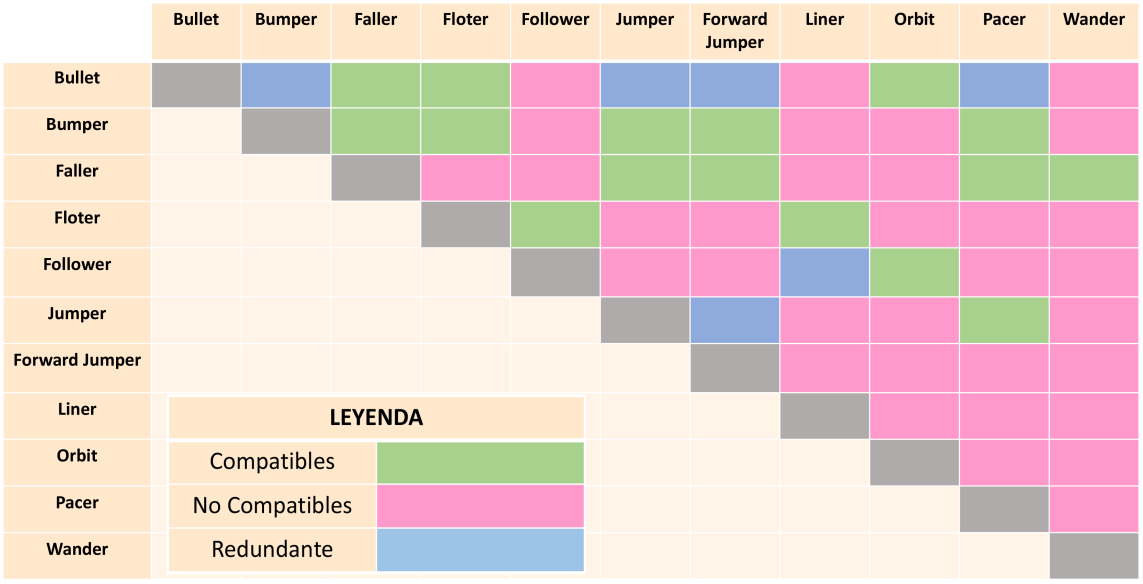


Figura 4.10: Combinaciones de componentes posibles

Combinar comportamientos simples es muy útil para crear la ilusión de que los enemigos se mueves de forma realista, sofisticada y de una manera razonable. Pero existen unos límites: hay componentes que, sencillamente, no encajan bien juntos. Por ejemplo, si se tuvieran dos componentes en un enemigo y uno dictase que quiere avanzar en una dirección, y el otro componente hiciera lo mismo en la dirección opuesta el resultado sería que uno ejercería más fuerza que el otro o que sus fuerzas, de tener la misma magnitud, se acabarían anulando. Sea lo que fuere, el comportamiento resultaría tosco y poco eficiente si actuasen a la vez.

4.4. Sensores

La clase Sensor es una clase de la que heredan todos los sensores de esta herramienta, como se aprecia en la Figura 4.11. Esta herencia supone que todos los sensores obtienen

los atributos y métodos de esta clase padre, por tanto la clase sensor contiene la parte común de todos los sensores, junto con todas las funcionalidades iniciales. Cuenta con un *flag* para desactivar los componentes a la salida del sensor, lo que quiere decir es que la lista de componentes que se activan cuando el sensor detecta al jugador pasarán de nuevo a ser desactivados cuando el sensor activo deje de detectar al jugador. Es posible que dos sensores coexistan en un objeto, pero si se quiere desactivar los componentes esta opción tiene que estar activa en cada uno de los sensores, en el caso contrario no se desactivarán porque uno de esos sensores lo mantiene activo. Estas características las gestiona la clase sensor, pero una clase sensor por si misma no hace nada porque no tiene una forma de recoger eventos del entorno.

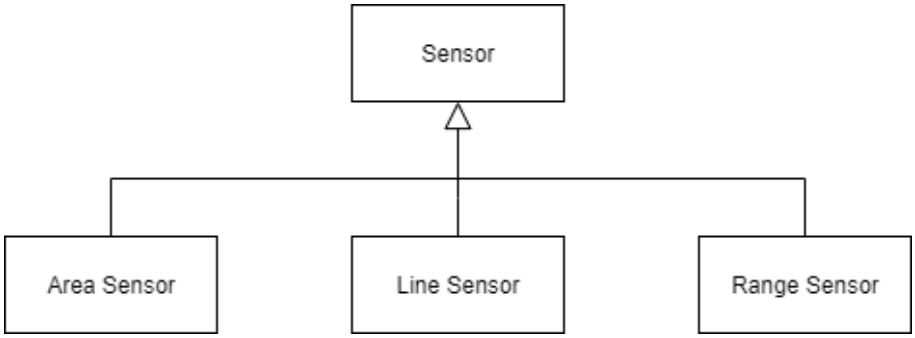


Figura 4.11: Esquema de los sensores

Los sensores son elementos que proporcionan a las entidades una capacidad sensorial con el entorno. Son los elementos que detectan ciertas condiciones en el mapa, y son capaces de activar componentes cuando detectan a su objetivo. Los componentes que se activarán en caso de detectar al jugador serán todos los componentes de movimiento que tenga la entidad, que previamente se habrán desactivado al iniciar la aplicación. Además, en los enemigos que se mueven por físicas se desactivará el efecto de la gravedad sobre los mismos para evitar que se caigan por su propio peso al inicial la aplicación y tener sus componentes de movimiento desactivados.

Al inicio de la aplicación, los sensores recorren la lista de componentes del objeto al que están asociados y, automáticamente, los desactivan. Estos componentes son añadidos a una lista aparte para activarlos si el sensor es activado, de forma que cuando el sensor haya detectado al jugador sea capaz de recordar los componentes que ha desactivado y volver a iniciarlos para activar sus comportamientos.

Los sensores tienen formas distintas de detectar al jugador, pero todas tienen en común la habilidad de activar componentes cuando lo detectan. Es posible marcar una opción en los sensores para que en el caso de que el sensor esté activo, si dejan de detectar al jugador desactiven los componentes de movimiento que previamente fueron activados. Es posible colocar más de un sensor en una entidad, debido a que los sensores sólo anulan los comportamientos de movimiento. Además, cada sensor cuenta con una lista privada de movimientos a activar por lo que no existe una dependencia entre sensores: simplemente se activarán los componentes más de una vez. No obstante, si se desea desactivar los componentes cuando el jugador deje de ser percibido es indispensable marcar la opción en todos los sensores. En el caso contrario, a pesar de que los sensores dejen de percibir al jugador, los componentes seguirán activos.

4.4.1. Range Sensor

Un range sensor, o sensor de rango, detecta la proximidad con el objetivo y activa los componentes listados si el objetivo está dentro de un umbral de proximidad. El sensor de rango funciona teniendo en cuenta la diferencia de posiciones que hay entre la entidad y su objetivo, siendo posible seleccionar un eje particular, o que sea basado en distancia simple, como se ve en la Figura 4.12.

Atributos Range Sensor

- (Enum) Tipo de Sensor: Selecciona el eje que se tendrá en cuenta para medir la distancia o si, por el contrario, mide la distancia general.
- (Float) Detection range: Rango de detección del sensor.

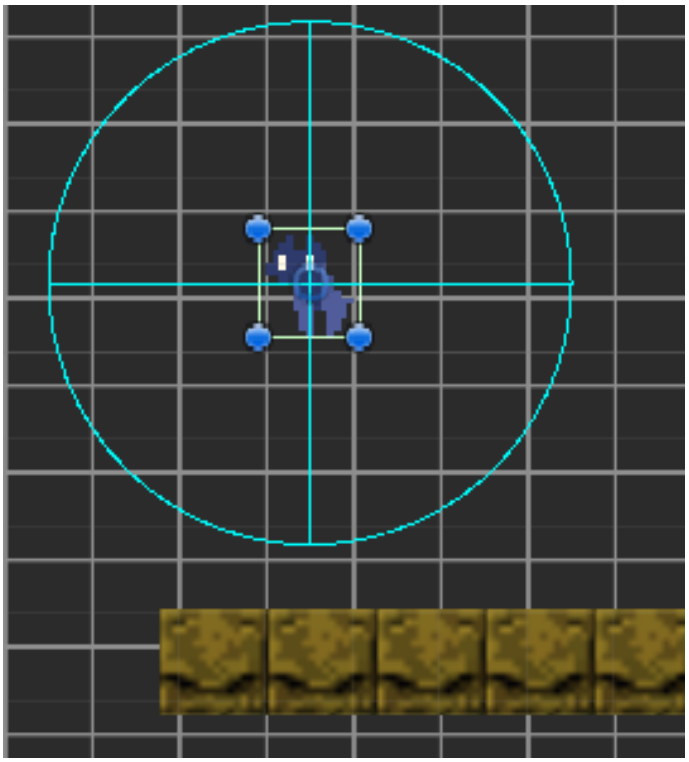


Figura 4.12: Sensor de rango

4.4.2. Line Sensor

Un line sensor, o sensor de línea de visión, lanza un rayo físico constante en una dirección y con una magnitud personalizables. Como se ve en la Figura 4.13, se dibuja una línea en la escena representada por la dirección y el tamaño del vector elegidos. Cuando se inicia la escena, la línea visual se transforma en una línea física generada por el sensor y por tanto sólo responde ante colisiones con capas físicas. Para que funcione correctamente, es indispensable que el jugador tenga una capa física asociada, y es recomendable que sea una exclusiva para este. Como se explicó en el apartado de arquitectura básica, la clase Enemy Engine es la que gestiona las capas de colisión y, por tanto, este componente preguntará a la clase Enemy Engine que tenga la entidad por la capa física correspondiente.

Atributos Line Sensor

- (Bool) See through walls: Determina si el sensor de línea de visión puede ver a través de las paredes o no.
- (Vector 2) Line direction: Define la dirección que tendrá el rayo.
- (Float) Line distance: Distancia de la línea de visión.

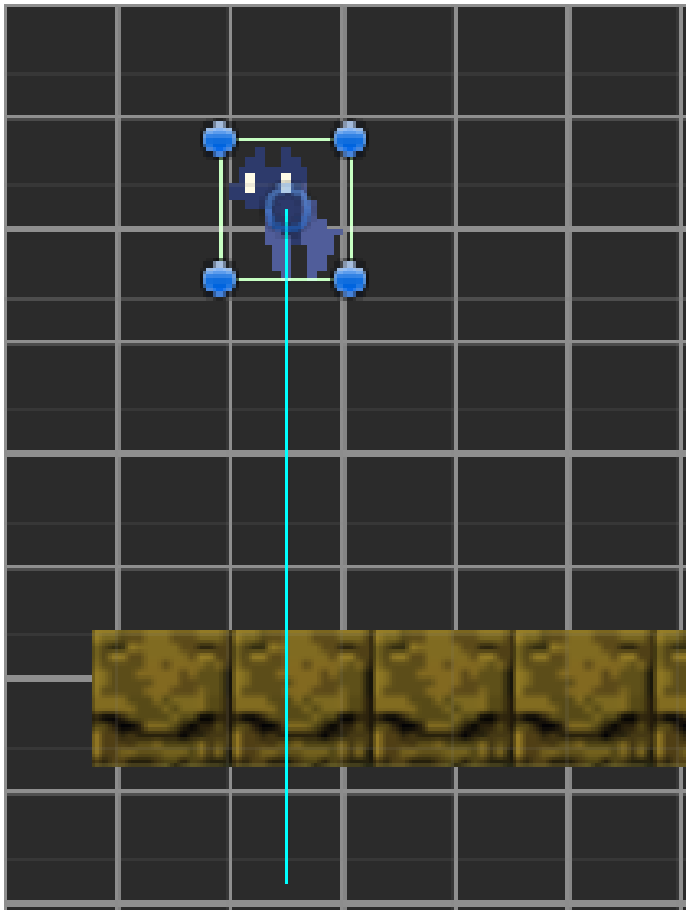


Figura 4.13: Sensor de línea de visión

4.4.3. Area Sensor

Un área sensor, o sensor de área, es un espacio de dimensiones personalizables que sabe detectar si el jugador está en ella. Cuando el jugador entra en estas zonas invisibles se activan los componentes de movimiento asociados al jugador. De la misma manera, si se desea desactivar los componentes y el jugador abandona la zona se desactivarán todos los atributos de movimiento.

Como se puede apreciar en la Figura 4.14, el área de detección se puede colocar a voluntad gracias al vector de posición local, que sitúa el área tomando como referencia a la entidad. A pesar de que el área aparezca cuadrada, el tamaño es fácilmente personalizable gracias al vector de tamaño. Como sucede en todos los gizmos, los cambios se ven reflejados según se efectúan.

Atributos Area Sensor

- (Vector 2) Local Area Position: Determina la posición local del área, que por defecto es la posición de la entidad.
- (Vector 2) Area Size: Determina el ancho y el alto del área.

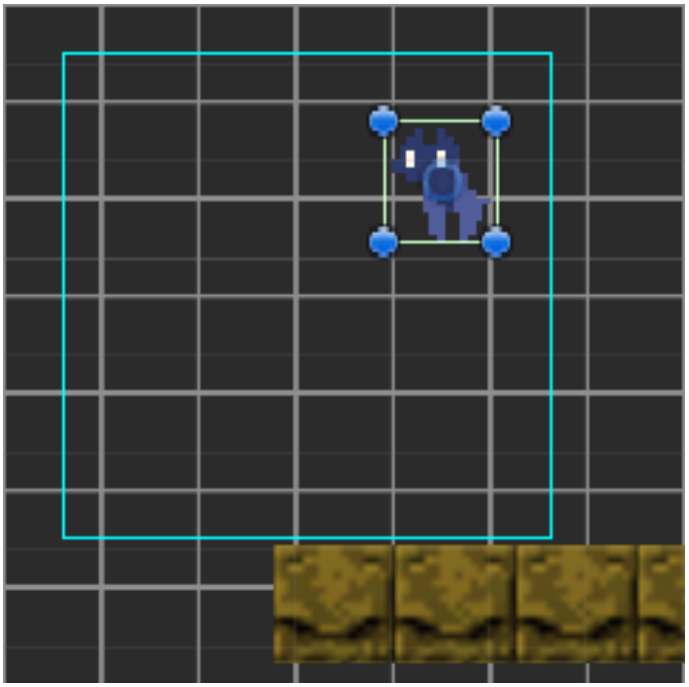


Figura 4.14: Sensor de área

Capítulo 5

Evaluación con usuarios

En las fases finales del desarrollo se llevaron a cabo pruebas con usuarios que no habían tenido contacto previo con la herramienta. Una vez el catálogo estuvo implementado, se desarrolló un plan de pruebas para probar si el diseño de la herramienta planteado es correcto, e identificar posibles cambios y mejoras. En este capítulo se recogen los objetivos de las pruebas, se analizarán los resultados obtenidos y se sacarán conclusiones al respecto.

5.1. Objetivos de las pruebas

Previo a las pruebas con usuarios, se definieron una serie de objetivos para la evaluación. Estos objetivos, descritos a continuación, están diseñados para cubrir los fundamentos principales del catálogo de componentes y orientar las pruebas para que respondan a estas cuestiones.

Objetivos de las pruebas

- Valorar la usabilidad del estado actual de la herramienta como catálogo para diseñar enemigos.
- Evaluar si el nivel de configuración de los componentes que constituyen el catálogo es correcto.
- Averiguar si faltan componentes o si, por el contrario, algún componente se podría retirar del catálogo.

Para definir estos objetivos, es importante recordar la fase de diseño del Capítulo 3 donde se analizaron los enemigos del videojuego Cave Story, se definieron los componentes de este catálogo y, de forma teórica, se demostró su funcionamiento creando enemigos con el catálogo. El objetivo de la herramienta es que cualquier persona pueda construir enemigos de forma sencilla, por ello las pruebas se orientaron a replicar comportamientos de enemigos clásicos con los componentes y sensores implementados.

5.2. Organización de las pruebas

Debido a la pandemia mundial que tuvo lugar durante la publicación de esta memoria debido al virus SARS-CoV-2, conocido como *Coronavirus*, la planificación del trabajo y las pruebas con usuarios de vieron modificadas. Inicialmente se iba a probar la herramienta de forma presencial y posiblemente con un número más elevado de probadores. No obstante, las pruebas se han realizado de forma online, mediante una videollamada individual con cada probador. Las pruebas tenían una duración de unos treinta y cinco a cuarenta minutos.

En la fase previa, tras una introducción, se le explicaba al probador brevemente los componentes que va a manipular, usando la guía visual de referencia con los comportamientos que deberían esperar y garantizarle asistencia durante la prueba para resolver dudas o errores que pudieran surgir. La guía visual puede consultarse en el Apéndice B, y consta de dos páginas. Para evitar sobrecargar al probador al principio, les recomendaba centrar su atención en la primera página (Figura B.1) en la que se ilustran los tipos de movimientos básicos que iban a usar al principio, y con eso delante se les explicaba brevemente los componentes así como se les solucionaba alguna duda que pudieran tener. Cuando el probador había realizado entre tres o cuatro comportamientos básicos, pasaba a introducirles las combinaciones de los mismos. Eran combinaciones sencillas, siguiendo la tabla de la Figura 4.10, y no se les avisaba de que tenían que combinar dos sencillos explícitamente, para ver si se entendía que algunos comportamientos eran combinables. Finalmente, se les introducía sensores a los comportamientos a replicar. Era entonces cuando se les dirigía a la segunda página, en la Figura B.2 y, de nuevo, se les explicaba brevemente o les resolvía cualquier duda que tuvieran al respecto. Para la explicación inicial no hace falta entrar en detalle, puesto que la guía visual aporta el conocimiento necesario para empezar a utilizar la herramienta. Esto se realizaba mientras abrían el proyecto y se instalaba el paquete vacío con todo lo necesario, así se aprovechaba el tiempo.

Durante las pruebas se pedía a los probadores una serie de comportamientos, entre siete y ocho, para que los replicaran con la herramienta. No se le mencionaban los nombres de los componentes, sino que las tareas se presentaban en forma de descripciones de enemigos clásicos de videojuegos 2D. Con la descripción de lo que tiene que hacer en enemigo, y los datos que proporciona la guía, el objetivo era que el probador crease un comportamiento aproximado. Por lo general, existe una forma rápida y directa de obtener ese resultado que es usando el componente correcto, no obstante había veces que el comportamiento era muy aproximado como para considerar que la tarea había resultado errónea. Por ejemplo, uno de los probadores decidió utilizar un comportamiento Floater (Figura 4.6) horizontal para cubrir una plataforma sin caerse, lo que según el diseño original se puede resolver con un Pacer. En este caso, ambos comportamientos cumplen con la descripción propuesta de que el enemigo debe recorrer la plataforma, teniendo en cuenta que al llegar al extremo ha de darse la vuelta así que se daba por válido.

Durante la prueba, se les respondía a las dudas que tuvieran y, para que los probadores no sintieran que estaban siendo examinados, se les dejaba experimentar con la herramienta como quisieran aunque eso supusiera una desviación en las pruebas. Debido al reducido número de probadores, y a que las pruebas se hacían de forma online, las pruebas se diseñaron a mano y se personalizaron en función de los grupos componentes que funcionaban bien juntos. No obstante, todas las pruebas con usuarios siguieron el mismo patrón: al principio, se describían componentes sencillos, luego se introducía una combinación de algunos componentes previamente vistos y finalmente introducían comportamientos de enemigos con sensores. Al principio de la prueba, se elegía un grupo de descripciones de comportamientos

para la prueba de forma aleatoria entre todas las pruebas diseñadas que no se hubieran utilizado. No obstante, si durante el transcurso de las pruebas con el usuario iban fluidas, se hacían menos pruebas de componentes sencillos y daba el salto a cosas más complejas mientras que en caso contrario incidía más en los comportamientos para intentar identificar los problemas de base en la herramienta.

Para extraer los resultados se ha utilizado un cuestionario sencillo, de apenas 3 minutos, en el que se recoge lo más importante de cara a generar unas gráficas que respondan a las preguntas principales definidas en los objetivos. Las preguntas eran de escala numérica, donde la escala medía el nivel de acuerdo con la frase de la pregunta. Estaba valorado de tal manera que el 1 significaba un desacuerdo absoluto con la afirmación planteada, mientras que un 5 significaba un acuerdo absoluto con la afirmación. El cuestionario era totalmente anónimo, y antes de mandar el formulario se les pedía que dejasen de compartir pantalla para no ver lo que respondían. Por supuesto, todos los probadores realizaron el mismo cuestionario por lo que los datos reflejan la opinión y crítica de los probadores de forma conjunta.

5.3. Resultados de las pruebas

Los experimentos con la herramienta se realizaron a lo largo de una semana, y todos los usuarios utilizaron el mismo paquete de Unity con el mismo catálogo de componentes. Lo único que se modificaba entre versiones del paquete era la corrección de los errores que pudieran tener algunos de los componentes y que entorpecían el correcto desarrollo de las pruebas.

En total, este experimento ha contado con diez probadores que se han ofrecido voluntariamente a probar la herramienta. Para buscar probadores se utilizó la red social Twitter, y el director de este TFG, por su parte, difundió la herramienta por los foros de las asignaturas que imparte. Al contar con probadores que vengan de redes sociales se ha observado como algo positivo que la disposición a hacer pruebas y la curiosidad por la herramienta surge de ellos. A pesar de no tener un gran número de pruebas se han obtenido unas opiniones más variadas que si hubiera seguido el plan de pruebas que se pensó inicialmente con un gran número de alumnos del grado, donde los comentarios sobre la herramienta hubieran sido mucho más homogéneos.

Los probadores que se han ofrecido a probar esta herramienta tienen distintos niveles de conocimiento en el desarrollo de videojuegos y por tanto se pueden englobar en dos conjuntos:

- Noveles: Este grupo está compuesto por 2 alumnos de primer año del Grado en Desarrollo de Videojuegos en la Universidad Complutense, estudiantes que están aprendiendo los fundamentos de la programación y el desarrollo de videojuegos por lo que suponen un público objetivo de la herramienta. Tienen un conocimiento aproximado acerca de la programación, pero son un grupo que no tiene experiencia real aún. Se incluye en este grupo a una diseñadora del Grado en Diseño, por la facultad de Bellas Artes de la Universidad Complutense que, a pesar de no contar con conocimientos de programación, ha utilizado Unity en diversas asignaturas. Este grupo de 3 personas supone un público objetivo porque este catálogo debería resultar sencillo de utilizar por usuarios sin conocimientos de programación.
- Veteranos: He podido contar con 5 personas de últimos años del Grado en Desarrollo

de Videojuegos y que, por tanto, tienen más conocimiento y experiencia en el campo del desarrollo de videojuegos. Se incluyen en este grupo, además, 2 personas veteranas de la industria, con varios juegos lanzados al mercado y experiencia trabajando con herramientas para el desarrollo de videojuegos. Es interesante contrastar las opiniones de este grupo con las opiniones del conjunto de probadores noveles para entender qué se esperaría de este catálogo desde un punto de vista más profesional.

Las opiniones de este segundo grupo, más mayoritario, son especialmente interesantes porque son personas que cuentan con una experiencia mucho mayor en el desarrollo de videojuegos y por tanto, pueden criticar el proyecto no desde un punto de vista académico sino como una herramienta que pueda salir al mercado, valorando la usabilidad y el nivel de acabado como personas que usan herramientas similares en su día a día.

Respecto al nivel de utilidad a nivel teórico, en la pregunta del cuestionario acerca de lo útil que resulta la herramienta se aprecia que la impresión general acerca de la idea de hacer un catálogo de comportamientos llama mucho la atención. Como se indica en la Figura 5.1, un 90 % (9 de 10) de los probadores coinciden en que la herramienta funciona como catálogo de componentes, con un 70 % calificándolo con la mejor puntuación, lo que deja en manifiesto el enorme potencial que tiene este proyecto como herramienta para crear movimientos enemigos en dos dimensiones, y augura un buen futuro a la herramienta si se ampliase su utilidad.

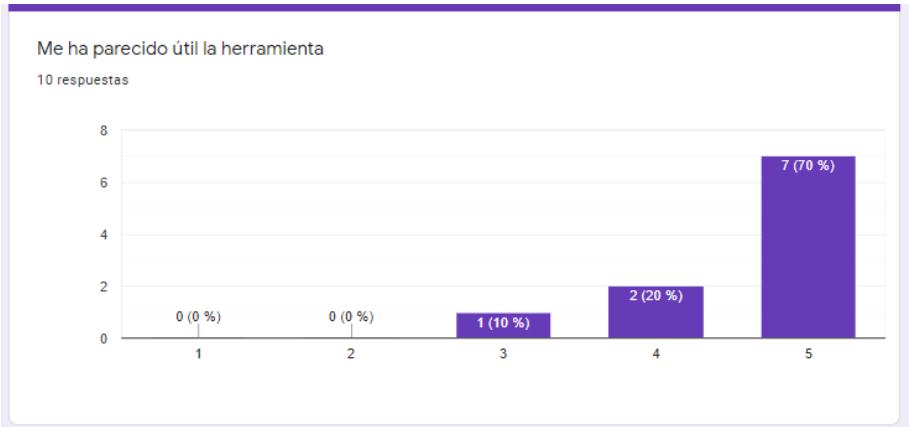


Figura 5.1: Utilidad de la herramienta

Un objetivo de las pruebas era evaluar si la herramienta es accesible, es decir, que sea rápida y sencilla de utilizar por cualquier persona independientemente de su nivel de experiencia con ella. Idealmente, una persona que prueba la herramienta por primera vez y que, por tanto, no conozca nada sobre ella pueda aprender a usarla y construir su primer enemigo rápidamente. Para ello se desarrolló la guía visual (Figuras B.1 y B.2), con la idea de que sirviera como documentación tanto para aprender a utilizar la herramienta como para refrescar su contenido.

La herramienta resulta sencilla de aprender a usar, pero no todo lo que hubiera resultado deseable. En este caso, la interpretación que se le da a los resultados tan variados, como se aprecia en la Figura 5.2, acerca de por qué no ha resultado tan sencilla de utilizar se debe a los estándares de cada perfil de probador. Un factor determinante es que la guía visual resultaba útil, pero era un arma de doble filo porque si generaba confusión inicial, la primera impresión de la herramienta podría resultar frustrante. La crítica elaborada a la guía visual está al final del Apéndice B.

Durante las pruebas, se observó que los alumnos de primer año del grado utilizaban la herramienta de forma correcta una vez veían los componentes funcionar en ejecución lo cual no es negativo, ya que no se considera que el ensayo y el error sea algo a evitar al utilizar esta herramienta. No obstante, cuando se les presentaban las primeras descripciones, el proceso de saber qué comportamiento utilizar era más lento, y la gran mayoría fallaba en su primer intento. Esto pasaba también con alumnos de los últimos años del grado, pero en menor medida debido a la experiencia previa y a que contaban con una formación en programación. Para los probadores veteranos de la industria, la usabilidad de la herramienta no cumplía con los estándares que ellos utilizarían en el día a día, y me inspiraron a hacer modificaciones como la integración con Unity que se ve en la Figura 4.3.

A pesar de la diversidad de opiniones que observamos en la Figura 5.2 acerca de lo fácil que es empezar a utilizar la herramienta, que se podría calificar como buena, pero no sobresaliente, se extrae de la Figura 5.3 un punto positivo que es que a pesar de que la herramienta pueda tener una dificultad de entrada más alta que lo previsto, la herramienta no genera frustración cuando se utiliza. Un 60 % de los encuestados afirma que están en desacuerdo con la afirmación de que la herramienta les parece frustrante, pero un 30 % han sentido algo de frustración al utilizarla, presumiblemente por esas primeras interacciones con la herramienta.

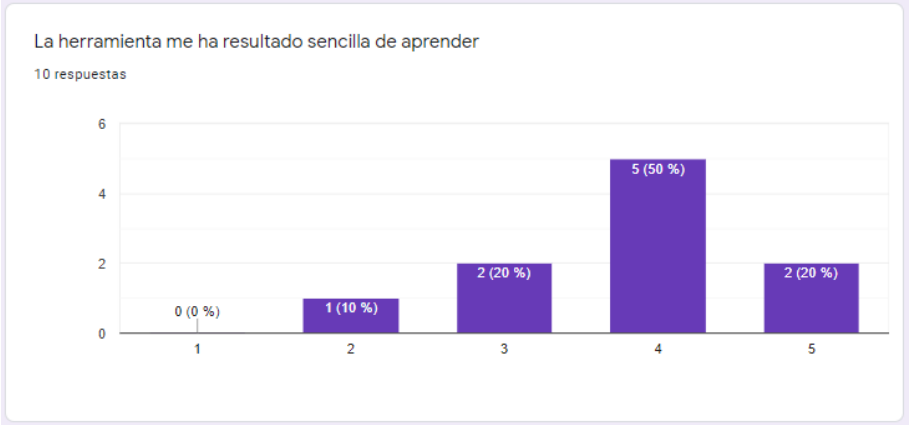


Figura 5.2: Sencillez de la herramienta

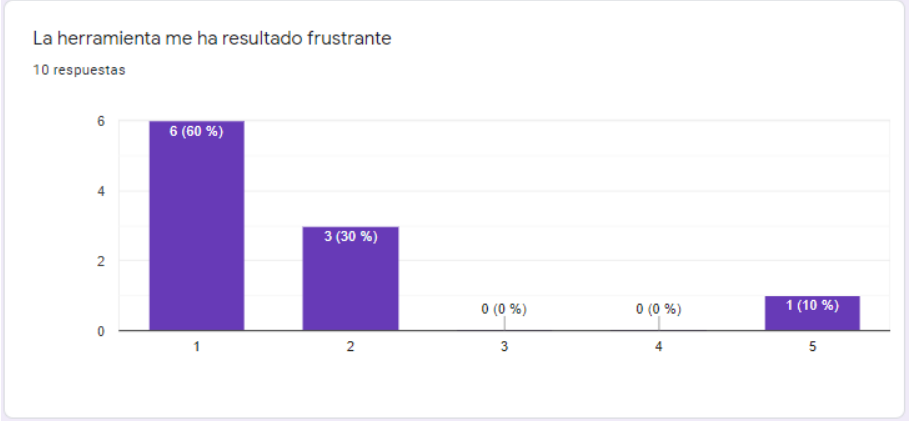


Figura 5.3: Frustración generada por la herramienta

Otro aspecto problemático que perjudica la usabilidad de la herramienta son los nom-

bres de los componentes. Los nombres utilizados no son casualidad, sino que siguen un estándar inspirado en el artículo de Bright (2014) acerca de los componentes sencillos para enemigos en dos dimensiones, que se trata al inicio de este mismo Capítulo. Son nombres en inglés, siguiendo con los estándares comentados en el Capítulo 1 de que la herramienta tenía que ser íntegramente en inglés, pero no considero que eso haya sido un factor tan importante.

Como se aprecia en la Figura 5.4, los nombres que dado a cada uno de los componentes no son auto explicativos. Por ejemplo, uno de los nombre que han generado más confusión es Bullet. El componente Bullet se mueve en línea recta en una dirección simulando el movimiento de una bala, pero era muy confundido con el componente Liner por su asociación a "línea", a pesar de que Liner es un componente de movimiento lineal hacia el jugador. De nuevo, la guía visual supone un arma de doble filo porque, aunque es rápida de consultar, no se incluyen los atributos que tiene cada componente y aunque eso suponga una documentación más extensa ayudará a entender mejor de qué es capaz cada componente. Los probadores más veteranos, tanto los del perfil de industria como los de último año del grado, tienen a prestar más atención a los atributos públicos de ha herramienta, pero otro elevado porcentaje de probadores ignoran los atributos.

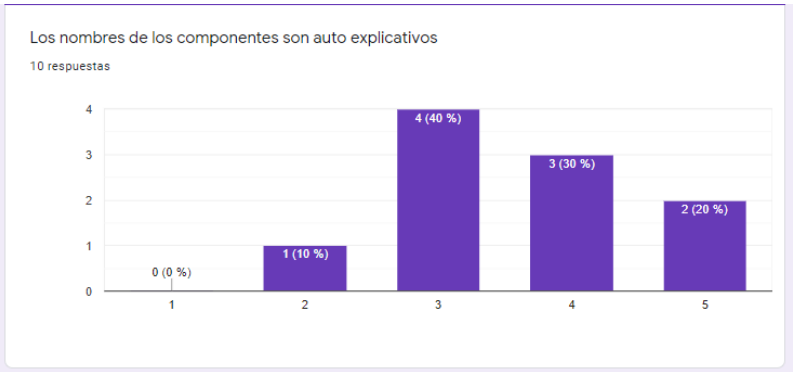


Figura 5.4: Claridad de los nombres

Figura 5.5: Personalización de los atributos

Un aspecto más positivo sobre los componentes ha sido que el nivel de personalización que presentaban los componentes en general ha sido bastante positivo. Como se aprecia en la Figura 5.5 con un 50 % de los probadores clasificándolo como adecuado. No obstante, no se puede ignorar que un 30 % de los probadores lo han considerado mejorable. Durante las

pruebas, observé que algunos de los probadores más veteranos echaban en falta un nivel de personalización más avanzada en algunos componentes.

Cuando los probadores más veteranos manipulaban los comportamientos de movimiento que implicaban conocer la posición del jugador, encontraba que había usuarios que echaban en falta poder definir un objetivo del movimiento distinto al jugador. Por ejemplo, cuando utilizaban el componente Follower que persigue al jugador se planteaban la opción de que el enemigo pudiera perseguir otro objeto, asignable desde el editor. Cuando se les preguntaba acerca de esta cuestión a los probadores más noveles, mencionaban que no lo echaban tanto en falta.

Otro punto de la herramienta que podría mejorarse, en relación a la Figura 5.5 es la configuración de los sensores. Actualmente, los sensores son un elemento muy simplificado y automático, tanto que se encarga de recoger manualmente todos los componentes de movimiento asignados al objeto y, por defecto, los desactiva hasta que el sensor detecte al jugador. Un probador de primer año del grado me planteó la cuestión de si podía crear un objeto que se mueva al principio de la ejecución y que, si el sensor se activa, incorpore un movimiento extra a su ruta. Con la implementación actual no sería difícil crear ese sistema, pero sería recomendable contar con el editor visual antes de integrar esa funcionalidad.

Lo que se extrae de la Figura 5.5 es que aún es preciso afinar los atributos públicos de los componentes, intentando apelar tanto a los usuarios más conformistas como a los usuarios que desean un nivel de personalización mas elevado, propio de una herramienta más sofisticada.

En resumen, la idea del catálogo de componentes cumple con sus objetivos principales de ser una herramienta sencilla e utilizar y capaz de construir enemigos personalizables rápidamente. Se podría calificar esta primera implementación como positiva, pero es importante corregir los nombres de los componentes así como crear una documentación mejor al margen de la Guía Visual (Apéndice B) que ayude a solventar las dudas que surjan al utilizar la herramienta por primera vez. Finalmente, sería interesante añadir una ventana más visual a la herramienta, para separar el catálogo del editor y así tener un mayor control sobre cómo se presenta la información.

Capítulo 6

Conclusiones y Trabajo Futuro

El objetivo del proyecto es diseñar y crear una herramienta para ser utilizada en *motor de videojuegos* Unity con el objetivo principal de facilitar un catálogo de componentes en C Sharp para comportamientos de enemigos. La herramienta deberá ser rápida y sencilla de utilizar por cualquier persona que quiera hacer un videojuego 2D, sin importar sus conocimientos de programación. Los componentes de este catálogo se podrán personalizar, añadir con facilidad e incluso mezclar para generar comportamientos nuevos. Será indispensable que cumpla con los principios de *abstracción* y programación orientada a objetos.

Para crear este catálogo de componentes, el primer paso fue diseñarlo. Como se vio en el Capítulo 3, utilizando el estudio de Bright (2014) y viendo la utilidad que éste tenía a la hora de diseñar comportamientos de forma teórica se diseñó la herramienta como una implementación práctica que cumpliera el mismo fin: crear enemigos para un videojuego 2D. La implementación de los componentes supuso tener que crear tanto los comportamientos de movimiento y componentes sensor, como la estructura interna que gestione todas las dependencias de cada componente, de tal manera que el usuario sólo tenga que manipular el componente del catálogo sin preocuparse por el resto. Durante el desarrollo de esta implementación se han desarrollado once componentes de movimiento, tres componentes sensoriales y una serie de clases internas que se explican en profundidad en el Capítulo 4.

Las pruebas realizadas con usuarios nos muestran unos datos positivos para el catálogo de componentes. Los datos son homogéneos a pesar de la diversidad de perfiles y niveles de experiencia, que ponen en manifiesto el enorme potencial de esta herramienta. Se podría afirmar que el diseño de este catálogo de componentes es satisfactorio, y que es capaz de cubrir una necesidad de la comunidad de desarrolladores retro. No obstante, aún es necesario pulir ciertos factores que pueden resultar frustrantes para el usuario final, como por ejemplo modificar los nombres de los componentes, añadir nuevos componentes que enriquezcan el catálogo y desarrollar una documentación clara y concisa sobre los componentes.

6.1. Trabajo futuro

Durante las pruebas con usuarios, que duraron apenas una semana, se puso en la balanza el trabajo de varios meses. En cada prueba de 40 minutos obtenía comentarios que aportaban ideas para meses de desarrollo, que ampliarían la herramienta y la harían mucho más sencilla de utilizar. A continuación, se deja constancia de las que se han considerado

más determinantes para el futuro de la herramienta.

La herramienta actualmente se integra con el *motor de videojuegos* Unity, pero podría ser mejor haciendo un editor propio. Una idea recurrente es que la herramienta podría contar con una ventana emergente que permita seleccionar de forma más gráfica los componentes que se desea añadir, como sucede con Construct 3. De esta manera, se reforzaría la accesibilidad de la herramienta, ya que en esa ventana de selección tendrías una mayor libertad para crear enemigos que estando integrada en Unity. Otra ventaja de tener una ventana emergente es que se puede diseñar libremente, y añadir funcionalidades como controlar que el jugador no añada dos componentes que no sean compatibles para evitar confusiones.

Urge cambiar los nombres actuales de los componentes, llegando a un consenso de nombres que representen mejor los movimientos esperados. También sería importante revisar los atributos públicos de los comportamientos, con elementos que supongan una capa extra de complejidad. Por ejemplo, es correcto que un enemigo vaya por defecto hacia el jugador, pero debería existir una opción que al marcarla despliegue dinámicamente unas opciones más avanzadas, como ajustar con una precisión extra el movimiento o añadir opciones nuevas a determinados comportamientos.

Finalmente, se ha demostrado que el proyecto funciona a nivel de diseño en su estado actual. No obstante, gracias a su organización por componentes es fácilmente ampliable con nuevos comportamientos de movimiento, además de la posibilidad de incluir componentes de combate, ataques típicos de enemigos en videojuegos 2D tales como disparos, lanzamiento de objetos o creación de nuevas entidades que funcionen con esta misma herramienta. De la misma manera, se podría integrar este catálogo en una herramienta de máquinas de estado finitas o árboles de comportamiento, descritas en el Capítulo 2, para emplear los componentes de este catálogo en comportamientos más complejos.

Chapter 7

Introduction

7.1. Motivation

This chapter contains the translation of Chapter 1.

When one decides to develop a game, there are certain steps that you must go through and, without a doubt, the most important one is the design phase. During this phase ideas are written on paper, and then simple prototypes are made to see if it's fun to play. Afterward, analyse, and repeat the process changing whatever is necessary to find something entertaining to play. Apart from the game's main objective, one must design the obstacles or challenges that will be introduced to the player to test them.

There are several tools that can help developers and improve their work. These exist to aid developers in many fields such as sound processing, rendering and also graphic programming but there is yet to exist one catalogue of components that helps design some basic enemies, none that a designer can use by themselves with no programming knowledge, or without the aid of a programmer that deals with the programming to create such basic game elements as simple enemies that provides an obstacle for the player to achieve the game's objective.

Video game's *NPCs* are entities that make believe that the game is "alive", since this characters are generally used to introduce mechanics to the game in a more humanizing way: Instead of having a plain menu with a shop, you have an NPC that looks like a merchant and sells you in-game objects, with dialogues that enrich the game. Enemy NPCs in 2D video games usually have behaviours that provides then with a certain degree of realism and serve as obstacles to test the player. For the given reasons, this tool is made for prototyping video games, for the use of developers or anyone that wishes to make a 2D video game but that does not necessarily knows how to code. This tool aims to provide all the facilities that should be necessary in order to cover a game's enemies with a customizable enemies for 2D video games, a tool that it is powerful but simple to use, easy to pick up and useful so that it can be used by anyone. With this tool, the process of making ones enemies designs into a game without having to code one single line of code, as to save development time, end up having functional as well as customizable enemies without programming being an obstacle, so that everyone can make a 2D video game.

7.2. Objectives

The main objective of this project is to design and create a tool to be used in the Unity *game engine* with the main purpose of providing a catalogue of C sharp components for making enemy behaviours. This tool should be easy to pick up for any kind of person that wishes to make a 2D video-game, no matter their programming experience. The components of this catalogue shall be customizable, easily addable and even mixable together to create brand new behaviours. Thus, to achieve that goal the components must follow the principles of *code abstraction*, as well as object oriented programming.

To create this catalogue of components, the techniques of *NPC* creation were studied, as well as the tools that offer easy ways of implementing them such as game engines that are oriented to developers with little to no experience in programming. After this analysis, the component catalogue will be designed and justified by creating certain enemies with the designed components. Afterwards, these components will be implemented into a practical version and then, finally, the catalogue will be tested with users in testing sessions that should be useful to collect data and commentaries about this catalogue, as to identify what works and what does not.

7.3. Works plan

To develop this project, we established an agile methodology from the start, during the first few meetings. The chosen methodology was studied in the subject Metodologías Ágiles de Producción, and used in quite a few other subjects such as Proyecto. My director would take the role of client, and we would have meetings every two weeks in order to show him my progress, listen to his feedback and criticism, resolve any doubts I might have and finally define the route of action for the following two weeks. This methodology is called Scrum, and it's quite simple: Scrum is based on frequent reunions, which should be quick ones in order to define objectives in short term commonly called "sprints". The scrum methodology relies on the auto-managed organisation to deal with the unpredictability of the development of any application, and manage before it gets out of control with said short "sprints". When a "sprint" ends, the team must make and analysis of it and schedule the objectives for the next one. Also, it was decided that all the code that is going to be produced shall be entirely in English, so that the language does not become a barrier in case any user decides to read through the code and study it.

The component catalogue is designed in a incremental manner during the development months. The chapters that make this memory describe each part of the process that is going to be followed for this implementation. First of all, a study of the State of Art (Chapter 2) will be made in which techniques of enemy behaviour creation will be explained. Furthermore, we will explore the tools that are commonly used that implement these techniques, as well as, the *game engines* that are suited for novel developers, extracting the positive aspects of them to be implemented in this tool.

After this study, the 3rd Chapter will be dedicated to define and justify the design of this tool. In this chapter the design will be explained, it is going to talk about the main inspirations for this project and the component's design will be listed, justifying the existence of these components by theorically making enemies with them. After this is explained and properly justified, then these components can be implemented.

As will be explained in the 4rd Chapter, the implementation of this catalogue is going to be made in the Unity *game engine*, and this chapter will define the structure of the components that will be implemented. It is important to define the basic architecture of this catalogue, and to explain both the internal implementation as well as the personalization that each component offers. These implementations and the internal details will be covered in this implementation chapter, in which some forms of component combination will be explained.

Once that the catalogue is implemented, it will be tested with tester users as to value its usefulness and extract some possible improvements for this project. As is explained in the 5th Chapter, a document regarding the design of the trials will be designed, taking into consideration what was learnt in the subject Usabilidad y Análisis en Videojuegos, regarding how to identify the objectives of the testing, and how to make questions that provide information about these objectives, as well as design some light tests that the testers must solve. The users needed to create enemies with this catalogue of components, so a basic knowledge of game development was desirable, but these users must not have used the tool beforehand as to know how easy this catalogue is to use as a first timer. These result are to be collected, analysed and some conclusions will be extracted in order to make a critic of what need changing and the functionalities that are missing.

Finally, the 8th Chapter gathers the conclusions of this project in which the final implementation process will be resumed, and it will analyse the positive and the negative aspects of this catalogue of components. Afterwards, it will define the future work that can be done for this catalogue and it will justify the possible changes and extensions that could be made.

Chapter 8

Conclusions and Future Work

This chapter contains the translation of Chapter 6.

The main objective of this project is to design and create a tool to be used in the Unity *game engine* with the main purpose of providing a catalog of C sharp components for making enemy behaviours. This tool should be easy to pick up for any kind of person that wishes to make a 2D video-game, no matter their programming experience. The components of this catalog shall be customizable, easily addable and even mixable together to create brand new behaviours. Thus, to achieve that goal the components must follow the principles of *code abstraction*, as well as object oriented programming.

To make this catalog of components, the first step is to design it. As was explained in the 3rd Chapter, using Bright (2014)'s article and considering the general utility of this study when designing enemy behaviours, the tool was designed as an implementation that serves the same purpose: to create enemies for 2D video games. The implementation of these components means that both the movement and sensor components needs to be implemented, as well as the internal structure that manages all the dependencies in such a way that the user only needs to handle each component without worrying about the rest. During the development of this implementation, eleven movement components, three sensors and several internal classes that are explained in Chapter 4 were created.

The testing with final users reveals some positive data for this catalog of components. The data is homogeneous although given the variety of profiles and levels of experience, which shows the potential of this tool. The design of this catalogue is satisfactory, and it helps solve a necessity for the retro developers community. Nonetheless, there are some aspects of this tool that needs polishing as they might be frustrating for the final user, for example the names of the components needs to be changed, new components could be added that could enrich this catalog and create a clear and straightforward documentation about the components.

8.1. Future work

During the testing with users, that hardly lasted a week, the work of various months was put to test. Every session, consisting of 40 minutes each, gave me feedback and ideas for another several months, improving the tool and making it easier to use. In the following

paragraphs, we will go through the ones I've considered more important for the future of this tool.

This tool is integrated in the *game engine* Unity, but it could be even better by adding a custom editor. A recurring, yet interesting idea is that the tool should be an emergent window in which you can select the components that you wish to add, as in Construct 3. That way, we can strengthen the accessibility of this tool, because in said selection window one could have much more freedom to create enemies rather than the tool being integrated in Unity. Another positive point of having an emergent window is that you can customise it, and add functionalities such as a way to control that the player does not add two components that does not work well together in order to avoid unnecessary confusion.

It is mandatory that the current naming of the components is reworked, creating a new group of games that suits better the expected movements. Furthermore, the public attributes of the behaviours needs some tuning, adding elements that make an extra complexity layer. For instance, it is correct that a follower enemy follow the player by default but there should exist an option that, when toggled on, displays more complex options such as adjust some extra movement parameters or add new options to certain components.

Finally, the project works design-wise in the current implementation. However, thanks to its organisation based on components the project is easily extendable with new movement behaviours, as well as the opportunity to add attack components like classic 2D attacks such as projectile shots, throw objects around or even spawning new enemies that may work in their own ways with this tool. There is certainly room for improvement with this project, and an excellent starting point is the Bright (2014) article, described in the 2nd Chapter that has a huge catalogue that can be integrated in this project.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

Bosch, J. Introducción a las máquinas de estado finito. *Geeks.ms*, Disponible en https://geeks.ms/jbosch/2009/12/16/ia-introduccin-a-las-mquinas-de-estado-finito-finite-state-machines-fsm-parte-i-de-ii/.

Bright, G. Build a bad guy workshop. *Gamasutra*, Disponible en https://www.gamasutra.com/blogs/GarretBright/20140422/215978/.

Davis, A. Blinky, inky, pinky, and clyde: A small onomastic study (2008). *Destructoid*, Disponible en https://www.destructoid.com/blinky-inky-pinky-and-clyde-a-small-onomastic-study-108669.phtml.

Mateas, M. Expressive ai: Games and artificial intelligence (2005). Disponible en https://web.archive.org/web/20120514225846/http://www.lcc.gatech.edu/~mateas/publications/MateasDIGRA2003.pdf.

Sabbagh, M. How to design formidable and unforgettable video game enemies (2015). *Destructoid*, Disponible en https://michelsabbagh.wordpress.com/2015/10/21/how-to-design-formidable-and-unforgettable-video-game-enemies/.

Sagredo-Olivenza, I., Gómez-Mart'in, M. A. y González-Calero, P. A. Un modelo integrador de maquinas de estados y arboles de comportamiento para videojuegos. Disponible en http://bb.padaonegames.com/doku.php.

Simpson, C. Behavior trees for ai: How they work. *Gamasutra*, Disponible en https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php.

Xu, D. The careful design of cave story (2016). *Gamasutra*, Disponible en https://www.gamasutra.com/blogs/DavidXu/20160209/265428/The_Careful_Design_of_Cave_Story.php.

Yannakakis, G. N. y Togelius, J. *Artificial Intelligence and Games (2018)*. Springer, Disponible en https://books.google.es/books?id=HK1MDwAAQBAJ&.

Apéndice A

Diccionario de términos

Con el fin de mantener el cuerpo del documento centrado en la herramienta, se empleará esta sección para explicar la terminología concreta del desarrollo de videojuegos que se emplea a lo largo del TFG.

Api: El nombre de API surge de las siglas de *application programming interface*, lo que podríamos traducir como la interfaz de una aplicación de programación. Define los tipos de llamadas o solicitudes que se pueden hacer, cómo hacerlas, qué formatos de datos se deben utilizar, qué esperar de las llamadas y las convenciones particulares que se deban seguir en el uso de una biblioteca, para ser utilizada por otro software como una capa de abstracción. Generalmente, se utilizan en las bibliotecas de programación.

Motor de videojuegos: Un motor de videojuegos no es más que un entorno donde el usuario cuenta con un conjunto de herramientas, que típicamente incluye renderizado 2D o 3D, un motor de físicas, gestión de memoria, y especialmente la capacidad de añadir scripts con código propio. Este TFG está basado en el motor de videojuegos **Unity**. Este generador de comportamientos se fundamenta con código creado mediante scripting, utilizando una arquitectura por componentes y cuidando tanto los principios de la programación orientada a objetos como la abstracción.

Script: Un script es un programa simple que, por lo general, se almacena en un archivo de texto plano y realizar distintas tareas. Unity se caracteriza por tener una arquitectura basada en **componentes**. Este tipo de arquitectura de software se enfoca en la separación de un código complejo en scripts sencillos y autosuficientes, que proporcionan una interfaz pública, con atributos modificables, y una utilidad a un objeto.

Los scripts se utilizan en los videojuegos para definir mediante interfaces una serie de acciones y atributos que posee, lo que respondería a la pregunta de "¿qué sabe hacer una entidad que tenga este comportamiento?". Por ejemplo, un objeto que tenga el componente **Jumper** asociado tomará como propio el código y lo incorporará como parte de ese objeto, de tal manera que un objeto con el componente **Jumper** sabrá hacer lo que esté programado en ese script, y podrá saltar.

Abstracción: El principio de abstracción se define como la existencia de una "barrera de abstracción"que separa las características específicas de un objeto, de tal manera que esconda las implementaciones internas y sólo exhiba los detalles necesarios para manipular la entidad en cuestión. En el caso de este TFG, la barrera de abstracción la marcan los atributos públicos que muestro al usuario, y en una menor instancia los métodos públicos que pueden ser llamados desde otros objetos pero que el usuario no va a ver. La parte oculta sería la implementación interna, el conjunto de métodos y atributos públicos que hacen que la entidad funcione pero que el usuario no tiene por qué saber cómo funciona.

Modularidad: El principio de modularidad se define como la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. En el caso de este TFG, cada componente de movimiento debe poder funcionar al ser añadido a un enemigo, sin tener dependencias con ningún otro componente de movimiento.

Npc: Obtiene las siglas del inglés *non-playable character*, que significa personajes no jugables. Esto quiere decir que un NPC es cualquier personaje que no esté controlado por un jugador. En el caso particular de los videojuegos, se entiende que un NPC está controlado por el ordenador mediante el uso de algoritmia o comportamientos predeterminados, lo que no implica necesariamente que posean inteligencia artificial.

Apéndice B

Guía visual de pruebas

B.1. La guía visual

La guía se ha llevado a cabo en la herramienta **Canva**, un software y sitio web de herramientas de diseño gráfico creada por la empresa Canva, Inc fundada por Melanie Perkins en 2012. Está basado en un formato de arrastrar y soltar que resulta muy sencillo de utilizar para usuarios sin un conocimiento del diseño gráfico, pero que proporciona unos resultados muy profesionales.

La idea de crear la guía visual como base fundamental del desarrollo de las pruebas surge debido a que el público objetivo de esta herramienta son diseñadores y/o gente que quiere algo sencillo y rápido. La herramienta necesitaba documentación pero a pesar de tener descripciones completas de cada uno de los componentes, estaban en formato texto. Si el diseñador que quiera utilizar la herramienta tuviera que indagar en la documentación cada vez que quiera aprender a usar algún componente, puede resultar algo tedioso. Por ello, la documentación tenía que ser gráfica, simple, y que de un sólo vistazo te sirva para aprender, o recordar, como funciona la herramienta. Ambas cosas tenían que ir a la par: si la herramienta es ligera y simple, la documentación debía ser visual y sencilla.

B.1.1. Crítica a la guía visual

Una parte fundamental de las pruebas fue instar a que los usuarios pensaran en voz alta, y señalar elementos bloqueantes o que le generan confusión para poder mejorar la herramienta. Algunas de las críticas fueron a parar a la guía visual, y me gustaría que quedase constancia de ellas como evaluación del material de pruebas.

Un punto que hay que reforzar de la memoria es la falta de texto. La guía visual se apoya mucho en ilustraciones (figura B.1) para expresar ideas, pero las ilustraciones han acabado por tener demasiada carga. Entre mi formación no está saber ilustrar, por lo que pedí consejo a una compañera del Grado del Diseño. Y es gracias a sus consejos por los que la guía no es un fracaso, pero tiene flaquezas. Además de las ilustraciones, urge añadir unas explicaciones en unas pocas líneas sobre el comportamiento general que refuercen la guía.

Además, incluir los atributos podría tener resultados positivos porque frecuentemente son ignorados en Unity.

Además, considero que se pierde información al no incluir los atributos porque un componente puede tener varias formas distintas de actuar, cambiando los valores, y es bueno que quede reflejado para que el diseñador no tenga que averiguarlo por su cuenta. Por ejemplo, el Follower con unos valores muy específicos tiene el efecto de cargar hacia el jugador cada cierto tiempo, en lugar de avanzar constantemente, así que sería bueno ponerlo en la guía como variaciones que ofrecen los comportamientos, enriqueciendo aún más la herramienta.

En conclusión, la guía necesita más texto y más ideas para hacer enemigos. Los dibujos quedan muy abiertos a interpretación, y no cumplen del todo con la función de informar a una persona. Sirven para refrescar los conocimientos, la guía como concepto es muy útil para recordar los comportamientos pero no es suficiente para aprender a usar la herramienta de cero.

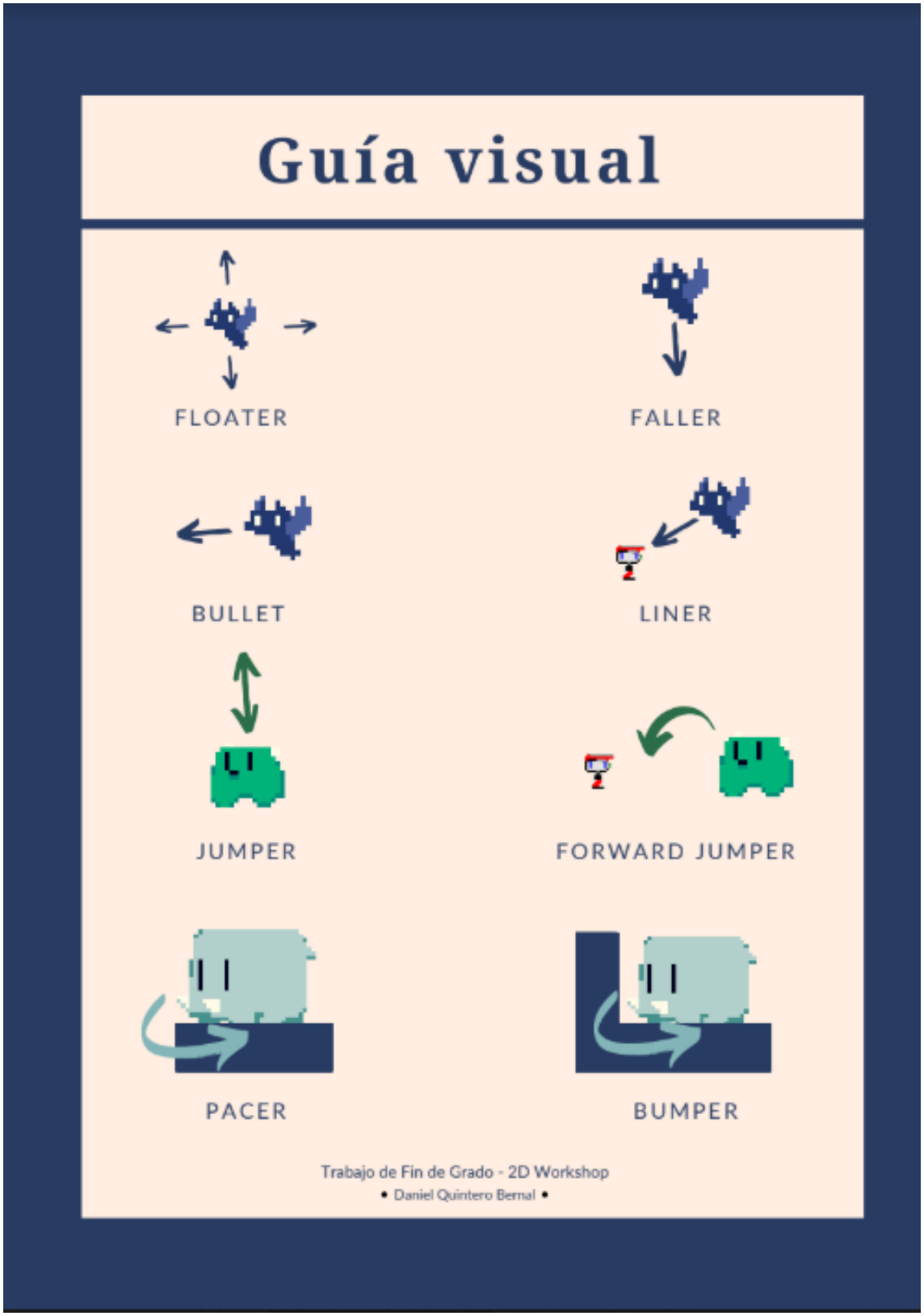


Figura B.1: Primera página

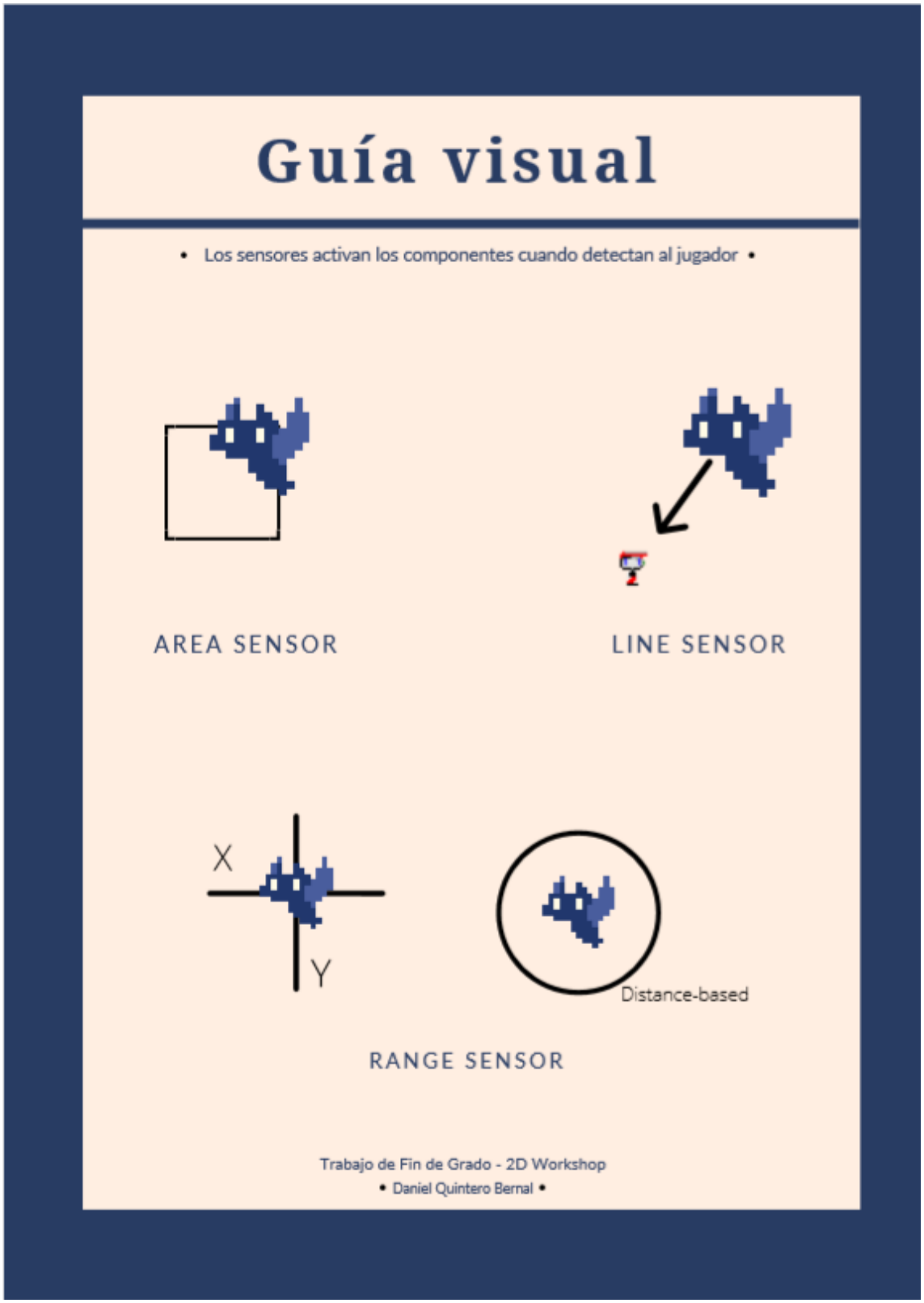


Figura B.2: Segunda página