Universidad
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Diseño e implementación de comportamientos para personajes del videojuego '¡Al ladrón!'.

Análisis y comparación de técnicas.

Memoria del Trabajo Fin de Grado

Grado en Ingeniería de Computadores

Autora:

Lucía Fresno Olmeda

Tutores: Miguel Ángel Rodríguez García y María del Soto Montalvo Herranz

Noviembre 2024

Resumen

En este Trabajo de Fin de Grado, se quiere decidir cómo implementar los comportamientos de los enemigos del videojuego '¡Al ladrón!', un juego de plataformas del tipo endless runner. Las tres opciones que se quieren valorar para la programación de estos personajes son las máquinas de estados, los árboles de comportamiento y los sistemas de utilidad. Se han desarrollado tres versiones de un enemigo, usando cada una de estas técnicas. De esta forma, se podrán comparar resultados y el trabajo que requieren cada una.

A lo largo de este documento se detalla el proceso seguido para la implementación de cada una de estas técnicas, desde su diseño, hasta su programación e incorporación en un prototipo del videojuego. Además, se hace una comparación de las tres, valorando cuál de ellas sería más conveniente para los personajes de '¡Al ladrón!'. El desarrollo se ha realizado en el software 'Unity', con ayuda de herramientas como 'Geogebra' para diseñar los diagramas necesarios.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

La industria de los videojuegos ha evolucionado enormemente en las últimas décadas. En este tiempo, se han hecho muchos avances, entre ellos, la inclusión de personajes no jugadores en los mismos. Un personaje no jugador, o NPC (Non-Player Character), por sus siglas en inglés, es aquel controlado por el propio juego.

Dependiendo de su función, se pueden distinguir entre NPCs amistosos y NPCs hostiles (Johansson y Verhagen, 2011). Atendiendo a la clasificación de Johansson y Verhagen, los NPCs amistosos suelen ser personajes que proporcionan ayuda e información al jugador, o le confían misiones. Sus interacciones normalmente están predeterminadas, con varios posibles diálogos pre-programados. Por otra parte, los NPCs hostiles conforman enemigos en el mundo del juego o de las misiones concretas que tiene que superar el jugador. Su funcionamiento es más dinámico, pero es común que siga siendo predecible a lo largo del tiempo.

Los definidos como NPCs hostiles pueden tomar distintas acciones en respuesta al entorno o las acciones del jugador. Estos funcionamientos más complejos se conocen como los **comportamientos** de los personajes. El comportamiento de un NPC es el conjunto de las acciones que puede hacer este y cuales 'decide' hacer en cada momento. En videojuegos, la toma de decisiones permite a los NPCs elegir qué acciones realizar, teniendo en cuenta su estado interno y las percepciones que reciba del entorno (Uludağlı y Oğuz, 2023).

Existen varias formas de modelar un comportamiento, desde árboles de decisión, sencillos de implementar, hasta complejas redes neuronales. Cada técnica tiene sus ventajas e inconvenientes, que se deben tener en cuenta a la hora de elegir cómo implementar un NPC en un juego. Además, es importante estudiar los resultados que se pueden obtener, el coste de su implementación y la eficiencia de esta última, junto con las características concretas del juego al que pertenecen y del equipo de desarrollo encargado de su implementación. Es relevante plantearse estas cuestiones a la hora de elegir cuál de estas técnicas utilizar.

En este Trabajo de Fin de Grado se va a atender la petición de una clienta que quiere decidir de qué forma implementar los NPCs enemigos de su videojuego, '¡Al ladrón!'. Las opciones que se quieren comparar son algunas de las más populares hoy en día: las máquinas de estados finitas (FSM, por sus siglas en inglés), los árboles de comportamiento (conocidos por sus siglas en inglés, BT) y los sistemas de utilidad (en inglés, US).

Para resolver esta cuestión, se va a implementar un prototipo del juego y programar el comportamiento de sus enemigos usando cada una de estas técnicas. De esta forma,

se podrán comparar los resultados obtenidos con cada técnica y analizar el trabajo que conlleve cada una de ellas. Gracias a esto, la cliente podrá decidir cuál de ellas es más conveniente en su caso, atendiendo a las características de su juego y del equipo encargado de desarrollarlo.

A lo largo de este documento se va a exponer el trabajo desarrollado, desde la planificación del proyecto y estudio del diseño del videojuego '¡Al ladrón!', hasta el desarrollo de su prototipo y de cada uno de sus personajes. Se describirán las características principales de las tres técnicas, además de algunos ejemplos de su uso en la industria de los videojuegos. Además, se mostrarán las dificultades que se encuentren a lo largo del proyecto, junto con las soluciones que se hayan empleado para resolverlas.

La realización de este Trabajo de Fin de Grado tiene como objetivo principal atender a la petición de la clienta, pero además servirá para observar las diferencias entre las tres técnicas en un entorno real y no únicamente de forma teórica. Adicionalmente, será una forma de poner en práctica los conocimientos adquiridos durante el Doble Grado de Ingeniería de Computadores + Diseño y Desarrollo de Videojuegos.

Capítulo 2

Estado del arte

En el campo de la informática, se conoce como Agente Inteligente (AI) a "*una entidad que percibe su entorno a través de sensores y actúa de forma autónoma y razonada con la mejor acción posible sobre ese entorno mediante actuadores*"(Alcalá, 2011). Su uso en videojuegos está bastante extendido, especialmente en personajes no jugadores.

En la programación de comportamientos para Agentes Inteligentes en videojuegos, existen una gran variedad de técnicas, desde algunas más antiguas como los árboles de decisión (DT), hasta estructuras complejas como las redes neuronales presentes en el videojuego 'Creatures' (Sweetser y Wiles, 2002).

Para este proyecto son especialmente interesantes tres técnicas: las máquinas de estados finitas (FSM), los árboles de comportamiento (BT) y los sistemas de utilidad (US). Cada una de estas técnicas tiene sus ventajas e inconvenientes. Por ejemplo, las FSM son sencillas y rápidas de producir, pero generan comportamientos más rígidos en los personajes; por otra parte, los US permite comportamientos más flexibles, pero son costosos y complejos de codificar (Sweetser y Wiles, 2002), (D. Garre Carlos y Casas, s.f.), (C. Garre, s.f.-b).

2.1. Máquinas de Estados Finitas

En su libro 'Elements of Robotics' (Ben-Ari, Mondada, Ben-Ari, y Mondada, 2018), Mordechai Ben-Ari y Francesco Mondada, escriben la siguiente definición:

> "*A finite state machine (FSM) consists of a set of states si and a set of transitions between pairs of states si, sj.*"(*Una máquina de estados finita (FSM) está formada por un conjunto de estados y un conjunto de transiciones entre pares de estados*) (Ben-Ari y cols., 2018).

En consecuencia, una máquina de estados finita es un modelo computacional formado por un conjunto de estados y transiciones. Un transición se produce cuando una condición

se cumple y se debe cambiar de estado. Por otra parte, un estado es la configuración de la máquina en un momento concreto, en general, representan acciones o la ausencia de ellas. Por ejemplo, los estados que puede tener una aspiradora son: apagada, encendida y aspirando. Si el usuario pulsa el botón 0N/OFF estando la aspiradora apagada, esta se encenderá; por otra parte, si lo hace con la aspiradora encendida, esta se apagará. Por lo tanto, la salida recibida depende tanto de la entrada, como del estado actual de la máquina.

Las FSM pueden tener un gran número de estados y recibir distintas entradas. Esto permite conseguir una gran variedad de funcionalidades, desde ascensores, hasta usos en programación de personajes en videojuegos.

Para definir el funcionamiento de una máquina en concreto, se utilizan diagramas de estados. En estos se representan todos los posibles estados de la máquina con círculos, y las transiciones con flechas (Ben-Ari y cols., 2018). Siguiendo con el ejemplo anterior, el diagrama de estados para una aspiradora como la descrita sería como el que se puede ver en la Figura 2.1.1.

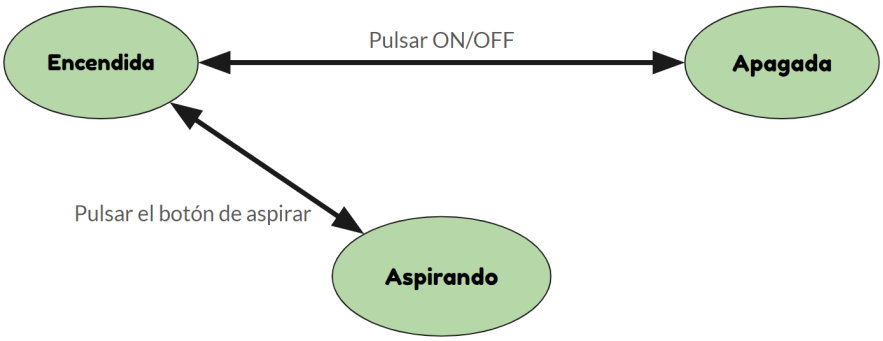


Figura 2.1.1: Diagrama de estados de una aspiradora

La implementación de este funcionamiento se puede hacer de varias formas: usando circuitos lógicos, programando la FSM en algún lenguaje de programación, entre otras. Esta última es la que se utiliza en el mundo de la informática y los videojuegos. Dependiendo de la complejidad de la máquina a programar, se podrá hacer con comprobaciones sencillas del tipo `if...else`, o se utilizarán códigos más avanzados.

Las FSM se han usado en videojuegos como una de las principales herramientas para modelar comportamientos. Sus usos van desde enemigos o contrincantes, como pueden ser los fantasmas en 'Pac-man' (1980) (Birch, 2010); hasta personajes secundarios con poca importancia en el juego, pero que aportan mayor realismo y enriquecen el mundo del juego, por ejemplo, los animales en 'Zelda: Breath of the Wild' (2017). Esta técnica también se usó en los perros de videojuego 'Quake' (1996) y en 'Half-Life' (1998) (AI y Games, 2019a).

Gracias a las FSM se pueden conseguir una gran variedad de comportamientos en videojuegos. Sin embargo, esta no es la única técnica que existe, por lo que es importante conocer sus ventajas y desventajas.

Una gran ventaja de las FSM es que son muy reutilizables: un mismo diseño puede servir para una gran variedad de personajes, incluyendo pequeños cambios. Por ejemplo, un comportamiento de un NPC que se dedica a descansar, comer y trabajar se puede usar

para todo tipo de trabajos: ganaderos, constructores o vendedores, entre otros. Por tanto, una misma FSM se puede reutilizar para varios NPC distintos.

Es bastante fácil diseñar máquinas con comportamientos sencillos y no es necesario tener conocimientos de programación ni informática para hacerlo. Sin embargo, supone un gran reto diseñar comportamientos muy complejos, ya que el número de estados y transiciones puede llegar a ser muy elaborado. Esto mismo se refleja en su implementación: cómoda con máquinas sencillas, pero inviable con máquinas muy complejas, ya que requerirán de un código muy extenso y complicado de mantener y depurar. Esta puede ser su mayor desventaja: es muy difícil y requiere mucho trabajo diseñar e implementar comportamientos complejos.

En conclusión, las máquinas de estados son una técnica muy útil cuando se requiere de comportamientos sencillos, ya que se pueden producir de forma fácil y rápida. Sin embargo, no son muy recomendables cuando se necesiten funcionalidades más complicadas.

2.2. Árboles de Comportamiento

Un árbol de comportamiento es un árbol dirigido con raíz formado por nodos de control de flujo, nodos de ejecución y la raíz (Colledanchise y Ögren, 2018). Estos árboles son estructuras jerárquicas que se utilizan para describir y modelar un comportamiento. Los BT nacen de una necesidad de implementar comportamientos complejos, que no eran viables con las FSM, dado el gran número de estados y transiciones que estos requerirían (Sagredo-Olivenza, Puga, Gómez-Martín, y González-Calero, 2015).

Su origen se encuentra en el mundo de los videojuegos y se popularizaron gracias a su uso en los mismos, uno de los primeros: 'Halo: Combat Evolved' (2001). Desde este momento, se ha usado en muchos otros juegos, algunos de los más populares son 'Far Cry Primal' (2016) o 'Spec Ops: The Line'(2012) ((Thompson, 2018) y (AI y Games, 2019b)). Sin embargo, este no es su único uso, ya que se puede encontrar también en robótica y autómatas (Cai, Li, Huang, y Yang, 2021).

Como se ha mencionado anteriormente, los BT son estructuras jerárquicas, formadas por nodos que se unen entre sí formando un árbol. Los nodos pueden ser padre y/o hijos de otros nodos, excepto el nodo raíz, que es el primero del árbol y únicamente es padre. En árboles de comportamiento se pueden encontrar tres tipos de nodos:

- Los **nodos hoja**: son los que se encuentran al final de cada rama, y pueden representar acciones o condiciones. El resultado que devuelven será éxito o fracaso.
- Los **nodos composición o de flujo**: tienen nodos hijos y definen el flujo en ellos. Existen una gran variedad de nodos composición y existen una serie de símbolos para definirse a cada uno de ellos. Los más comunes son: el nodo **secuencia** (representado con una flecha), que ejecuta todos sus nodos hijos de izquierda a derecha, hasta terminar o que uno falle; y el nodo **selector** (identificado por '?'), que ejecuta sus hijos en ese mismo orden hasta que uno de ellos termine con éxito.
- Los **nodos decoradores**: se suelen utilizar para indicar alguna funcionalidad un poco más compleja o hacer cambios en el resultado de algún nodo hijo. Los más comunes son el **inverter**, que invierte el resultado de un nodo hoja; o el **loop**, que ejecuta en bucle los nodos que se encuentren después de este.

Cada uno de estos tipos se representa con una forma geométrica distinta. Además, se suele usar algún tipo de diferenciación de color para distinguir los nodos de condición de los de acción y de los demás. Algunos ejemplos se pueden ver en la Figura 2.2.1 (C. Garre, s.f.-a).

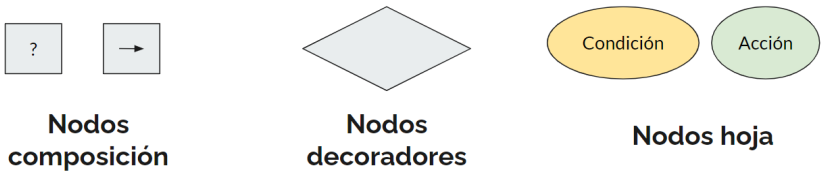


Figura 2.2.1: Tipos de nodos

La estructura de un BT es la siguiente: siempre comienzan desde un nodo **raíz**, que puede ser un nodo composición o uno decorador. Partiendo de este nodo, los nodos del árbol se van ejecutando hacia abajo y de izquierda a derecha. La ejecución del árbol seguirá hasta que el nodo raíz reciba un resultado: **éxito** o **fracaso** (Narratech, 2019). Este resultado vendrá dado por algún nodo hoja, que representará una condición o una acción. Qué se considera éxito y qué fracaso es algo propio de cada árbol, y de la lógica e implementación del mismo. De cualquier manera, este resultado será transmitido a su nodo padre. Dependiendo del diseño del BT y de los nodos composición y decoradores que haya, este resultado llegará hasta la raíz o implicará otro tipo de acción.

Para entender mejor este funcionamiento, se va explicar con un ejemplo típico: un personaje cuyo objetivo es abrir una puerta. El árbol de comportamiento del mismo sería el que se muestra en la Figura 2.2.2. Este árbol comienza con un nodo selector, que resultará en éxito cuando uno de sus hijos devuelva éxito, o fracaso, si todos ellos fallan.

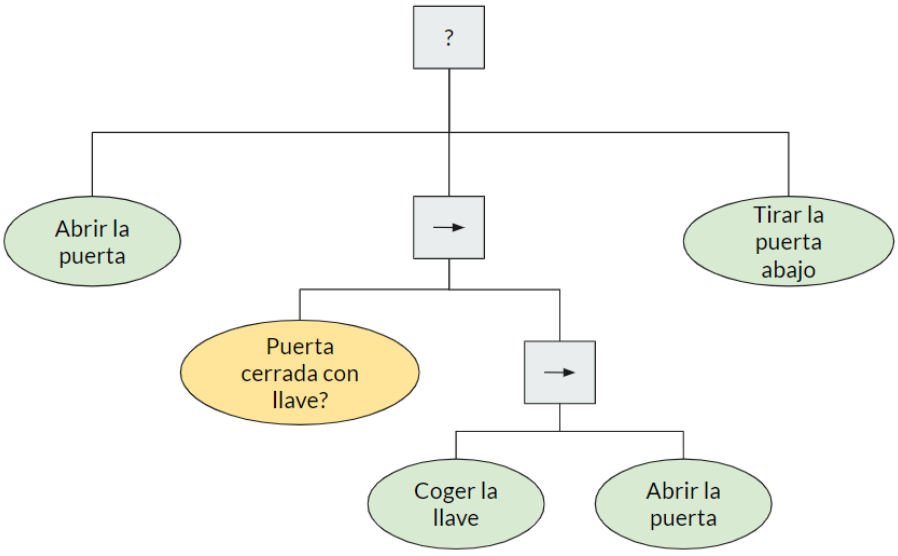


Figura 2.2.2: Árbol de comportamiento para abrir una puerta

Poniendo de ejemplo el árbol de la Figura 2.2.2, si el personaje consigue abrir la puerta directamente, devolverá éxito y la ejecución del árbol terminará, ya que el selector también actúa como nodo raíz. Por otro lado, si el personaje no consigue simplemente

abrir la puerta, comprobará si está cerrada con llave o bloqueada de alguna otra forma. En caso de que esté cerrada con llave, el personaje cogerá la llave y abrirá la puerta. Si alguna de estas acciones fallase, el nodo secuencia devolvería fallo, y el personaje intentará tirar la puerta abajo.

Como se puede ver en el ejemplo, el orden en el que se colocan los nodos en el árbol es decisivo. Esta es una de las mayores dificultades a la hora de diseñar un BT. No obstante, también es un punto positivo, ya que obliga al diseñador a tener en cuenta todas las posibles opciones y encontrar posibles errores en los momentos iniciales del desarrollo.

Los árboles de comportamiento pueden usarse para diseñar funcionalidades muy distintas entre sí. Una de sus grandes ventajas frente a las FSM es que se pueden modelar comportamientos bastante complejos y que su diseño e implementación siga siendo asequible (AI y Games, 2019b). Otro punto a favor de los árboles de comportamiento es que existe mucha información y documentación de los mismos, ya que es una herramienta muy popular en videojuegos y otros ámbitos. Por esto mismo, existen una gran cantidad de herramientas y ejemplos gratuitos que ayudan a los desarrolladores y les permiten hacer buenos diseños.

No obstante, también tienen desventajas: su diseño requiere de más tiempo que el de una FSM, ya que, como se ha mencionado antes, hay que tener muy en cuenta el orden en el que se accede a cada nodo para poder representar el comportamiento deseado correctamente. Aún teniendo mucha documentación disponible, el diseño de un BT se basa muchas veces en la prueba y error, ya que es fácil pasar por alto algún nodo que pueda ser necesario.

Con todo esto, los árboles de comportamiento son una herramienta muy popular, en especial en el mundo de los videojuegos y la programación de NPCs, aunque en muchas ocasiones esta técnica se combina con otras. Por ejemplo, incluyendo una FSM en un nodo de un árbol, o hacerlo al contrario (Sagredo-Olivenza y cols., 2015).

2.3. Sistemas de Utilidad

Los sistemas de utilidad, o sistemas de toma de decisiones basados en la utilidad (US), a veces incluidos en un grupo de técnicas conocidas como Goal-Oriented Behaviour, cuya traducción sería motivadas por su objetivo o meta. Estos sistemas basan su comportamiento en decidir entre una serie de acciones, dependiendo de cuál sea su meta y cuánto ayuda cada una de esas acciones a conseguirla (Millington, 2019).

Estos mecanismos de implementación de comportamientos se basan en la Teoría de la Utilidad Esperada. Esta teoría de toma de decisiones está basada en la hipótesis del matemático Daniel Bernoulli, que pensaba que las decisiones de las personas no estaban motivadas únicamente por el resultado de las mismas, sino también considerando sus preferencias (Jensen, 1967). A raíz de este pensamiento nace la Teoría de la Utilidad, usada en una gran variedad de ámbitos, desde la economía y negocios hasta la informática.

Esta teoría dice que cada individuo le asigna a cada opción posible en una decisión, una '**utilidad**'. Esta utilidad viene dada por las preferencias de cada individuo en una situación concreta (Academy, 2012). Por ejemplo: una persona debe decidir entre comprar dos productos iguales en dos tiendas que ofrecen exactamente el mismo servicio, uno más barato que el otro. Lo normal es que este individuo elija el producto barato, ya que perderá

menos dinero. Pero, esto podría cambiar si el producto barato tarda tres días más en llegar a su casa. Aquí entran las preferencias de cada persona: si para ese individuo tiene más valor el tiempo o el dinero. La 'utilidad' de cada una de estas decisiones para cada individuo se puede representar con una **función de utilidad**, en la que se representan los factores a tener en cuenta y la importancia que tiene cada uno de ellos para cada individuo, en este ejemplo serían el tiempo y el dinero (Graham, 2014).

Los US implementados en videojuegos se basan en esta teoría, de forma que el agente tendrá varias posibles acciones que tomar y decidirá qué hacer en base a la utilidad de cada una en cada momento. El diseñador del US será el responsable de determinar la función de utilidad que seguirá un NPC para tomar sus decisiones, además de definir qué elementos son los que influyen en cada una de ellas (C. Garre, s.f.-b). A estos elementos se les llama **factores de decisión**, y debe poder calcularse a partir de datos objetivos en el juego. Por ejemplo, un factor de decisión a la hora de elegir entre las acciones 'comer' o 'dar un paseo', podría ser el hambre del personaje. El dato objetivo del juego que permite saber si el personaje tiene hambre es: el tiempo que lleve sin comer. La relación entre el tiempo sin comer y el hambre se puede representar con un crecimiento lineal: a más tiempo sin comer, más hambre. La función o curva que representa esta relación es lo que se conoce como **curva de utilidad**. Se pueden usar todo tipo de funciones para representar estas curvas, las más comunes son: lineales, constantes, cuadráticas o exponenciales. La única restricción a la hora de hacerlo es tener en cuenta que el resultado final debe estar normalizado (tener un valor entre 0 y 1), para que se pueda comparar de una forma equitativa con otros factores (AI y Games, 2021).

En resumen, cada factor de utilidad se puede obtener gracias a la curva de utilidad que represente su relación con un dato objetivo del juego. Ese factor, junto con otros, se tendrán en cuenta en una función de utilidad para calcular la utilidad total de cada una de las acciones. Cada factor estará ponderado con un 'peso', un número que marca la importancia que tiene cada factor a la hora de calcular la utilidad. El personaje 'decidirá' qué acción hacer teniendo en cuenta la utilidad total de los mismos (TooLoo, 2021).

Por lo general, se suele seleccionar la opción con mayor utilidad, no obstante, esto no ocurre siempre así. Existen distintas estrategias a la hora de elegir qué acción realizar. Las más populares son: elegir la que tiene mayor utilidad total, elegir una acción aleatoria de entre las que tengan mayor utilidad o cambiar de acción solo si su utilidad supera un cierto umbral (C. Garre, s.f.-b). Todas las estrategias son válidas y es tarea del diseñador decidir cuál usar.

Los US se han usado en juegos como por ejemplo, 'Total War: Three Kingdoms' (2019) o 'Dragon Age: Inquisition' (2014) (AI y Games, 2021). Uno de los mejores y más populares ejemplos para el uso de sistemas de utilidad en videojuegos son 'Los Sims'. Su funcionamiento se describe brevemente en el artículo de Yoann Bourse (Bourse, 2012), en el que se explica que la felicidad de un Sim depende de 8 factores (ver Figura 2.3.1): el hambre, la comodidad, la diversión, la interacción social, la higiene, la vejiga, la energía y el entorno del Sim.

Figura 2.3.1: Necesidades de los Sims

En 'Los Sims', la utilidad que tiene cada necesidad en la felicidad se calcula con una función (o curva). Por ejemplo, un poco de hambre no produce un gran efecto negativo en la felicidad, pero mucha hambre sí lo hace. En su artículo, Yoann Bourse muestra las curvas de utilidad de estas necesidades respecto a la felicidad del Sim (ver Figura 2.3.2). Dependiendo de estos valores, el Sim decidirá qué acción hacer para poder aliviar la necesidad que más afecte a su felicidad en ese momento (Bourse, 2012).

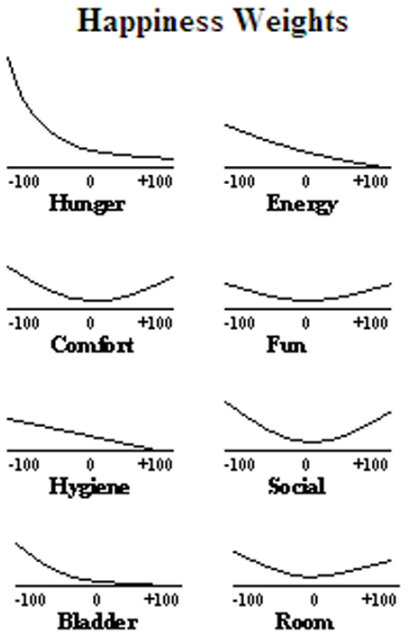


Figura 2.3.2: Curvas de utilidad de las necesidades de los Sims

Como se puede ver, los sistemas de utilidad son una técnica compleja, tanto a la hora de diseñar como de implementar comportamientos. Su uso conlleva mucho tiempo y esfuerzo, además, es conveniente tener unos conocimientos mínimos sobre funciones matemáticas. Estas son algunas de sus mayores desventajas. Sin embargo, tienen la gran ventaja de conseguir modelar comportamientos muy complejos y bastante cercanos a los de un individuo real.

En resumen, las tres técnicas más comunes en la implementación de comportamientos en videojuegos son las máquinas de estados, los árboles de comportamiento y los sistemas de utilidad. En algunas ocasiones incluso se pueden combinar entre ellas. Por ejemplo, modelando una FSM con árboles de comportamiento en uno de sus estados.

Para decidir qué técnica es más conveniente en cada caso concreto, los equipos de trabajo pueden tener en cuenta algunos de las siguientes cuestiones:

- Las ventajas y desventajas de cada técnica.
- La formación y conocimientos del equipo de diseño y desarrollo.
- El objetivo y tipo de juego.
- La importancia que tienen los NPCs en el juego, y si se busca un comportamiento realista en ellos.
- Las propias preferencias de los desarrolladores.

Atendiendo a estos factores, se podrá elegir la técnica más útil en cada caso, con la que conseguir los resultados esperados de forma más eficiente.

Capítulo 3

Objetivo y especificaciones del cliente

El objetivo final de este proyecto es atender la petición de la cliente de crear un prototipo de su juego y programar los enemigos del mismo con cada una de las tres técnicas mencionadas anteriormente, de forma que pueda decidir cuál de ellas será más conveniente. No obstante, antes de comenzar el desarrollo, es necesario entender el diseño del juego, el cuál ha sido proporcionado por el cliente en un Game Design Document (abreviado como GDD).

Algunos de los elementos más relevantes de este documento son: el género y sinopsis del juego, sus mecánicas, sus interfaces y sus personajes. En el GDD se describen estos de la siguiente manera:

Género: '¡Al ladrón!' pertenece al subgénero endless runner, también conocido como infinite runner (corredor infinito).

En este subgénero, perteneciente al género de videojuegos de plataformas, los jugadores avanzan de forma infinita por un camino, evitando obstáculos. El objetivo en estos videojuegos es aguantar el máximo tiempo posible en la partida. Este tipo de juegos no cuenta con una condición de victoria, por lo que la meta de los jugadores siempre será batir sus propias marcas, o las de otros jugadores. Sin embargo, siempre tienen una condición de fin de partida muy clara, en general, haber chocado con varios obstáculos.

Sinopsis y contenido: '¡Al ladrón!' es un videojuego infinite runner con una estética cartoon/infantil. En este juego, el usuario controla a un niño que está en un bosque recogiendo frutos. Al inicio de la partida, suenan las

campanas de un pueblo próximo y el niño se da cuenta de que es tarde y debe volver a casa, por lo que emprende su camino.

En su camino, varios animales intentarán robarle los frutos de su cesto. Para evitar esto, el niño debe estar atento y girarse si oye ruidos raros, espantando así a los animales. En el momento en que el jugador siga avanzando, los animales volverán a acechar su cesta. Además, debe tener cuidado de no tropezarse con ninguna roca, ya que se le caerá una pieza de fruta si lo hace.

Su objetivo será llegar lo más lejos posible sin que le roben. Si el jugador pierde tres frutos, terminará su partida.

Mecánicas: '¡Al ladrón!' es un videojuego en 3D con cámara en tercera persona. Sus controles son muy típicos: arrastrar el dedo a izquierda o derecha para moverse de un lado a otro; y un toque en la pantalla para mirar a los animales.

El sistema de puntuación es sencillo: se ganan 10 puntos por cada 5 segundos que el jugador pasa en la partida. Además, pasado un tiempo en la partida, empezarán a aparecer objetos que otorguen puntos en el camino y el jugador podrá cogerlos pasando por encima de ellos.

El sistema de guardado aparecerá en futuras versiones del juego, cuando este esté disponible para jugar online y entrar en rankings con otros jugadores. Sin embargo, para esta primera versión no habrá funcionalidades de guardado y cargado de partida, ya que es un juego con partidas cortas e individuales.

Por último, es interesante conocer en detalle la jugabilidad de '¡Al ladrón!': una vez termina la cinemática inicial, el jugador se moverá automáticamente hacia delante por un camino. Este camino tendrá varios 'carriles', y el jugador podrá pasar de uno a otro moviéndose hacia los lados, para poder evitar los obstáculos que aparezcan en ellos. Al mismo tiempo, el jugador debe escuchar con atención y, si oye el ruido de algún animal cerca, debe girarse para espantarlo y que no le robe la fruta.

El jugador contará con una cantidad limitada de 'energía', que se gasta cuando se gira para vigilar a los animales y se va recargando con el tiempo. Si pierde toda su energía, los animales le alcanzarán y le robarán fruta.

A medida que vaya avanzando la partida, el personaje se moverá más rápido, lo que dificultará el camino al jugador.

Interfaces: En la primera versión del juego se pueden encontrar interfaces en cuatro pantallas, todas ellas con un estilo cartoon desenfadado, en las que predominan los colores verde y amarillo. En la Figura 3.0.1 se muestra el diagrama de flujo de la aplicación, además de un pequeño boceto de las distintas pantallas.

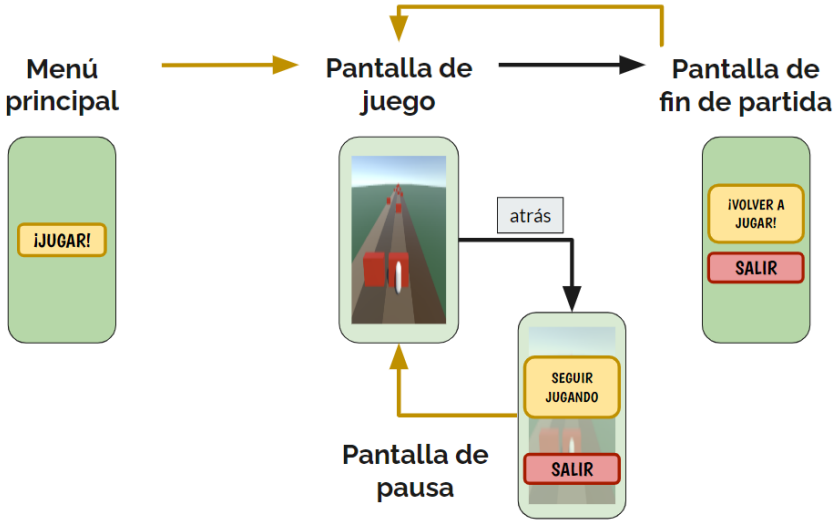


Figura 3.0.1: Flujo de pantallas del videojuego '¡Al ladrón!'

Las interfaces de '¡Al ladrón!' son:

- Pantalla de menú principal: es desde la cuál el jugador puede empezar una partida.
- HUD durante la partida: en esta interfaz el jugador podrá visualizar su puntuación y las 'vidas' que le quedan.
- Pantalla de pausa: interfaz en la cual el jugador tendrá la opción de seguir jugando o de salir del juego.

- Pantalla de fin de partida: es en la que se muestra la puntuación de la partida y se da la opción de de jugar otra vez o salir.

Personajes: '¡Al ladrón!' tiene a un único personaje principal: el niño al cuál controla el jugador. Para el primer prototipo no se ha desarrollado en profundidad su aspecto, pero sí se han reflejado en la siguiente Tabla 3.1 algunos aspectos claves del mismo.

Nombre	El personaje no tiene nombre propio.
Sexo	Masculino.
Edad	8 años.
Cualidades	Es un niño inocente, bueno y obediente. También es curioso, pero algo miedoso.
Trasfondo	El niño estaba en un bosque cerca de la casa de sus abuelos recogiendo frutos, para después hacer pasteles con ellos. Cuando oyó las campanas de la iglesia del pueblo sonar, se dio cuenta de que era tarde y decidió poner rumbo a casa. Es en este momento en el que comienza el juego.
Rol	Es el personaje que controla al jugador. Su único objetivo es que no le roben la fruta los animales, por lo que se dedica a huir de ellos.

Tabla 3.1: Ficha de personaje del niño

Enemigos: '¡Al ladrón!' cuenta con tres enemigos: los animales que intentan robarle fruta al niño. Estos tres personajes forman el 'Club del antifaz negro', famosos por robar frutos y objetos de valor a hurtadillas. Su papel en el juego es sencillo: perseguirán al jugador hasta que estén muy cerca de él y puedan robarle o hasta que este se gire a vigilarles, momento en el que se esconderán o huirán.

Estos tres NPCs conforman la principal dificultad del juego (Fresno, 2024).

Por otra parte, se hizo una breve entrevista para entender bien qué resultado se buscaba. Tras esta, se ha concluido que la finalidad del prototipo a desarrollar es únicamente observar el comportamiento de los NPCs enemigos, por lo que no es necesario invertir

tiempo en detalles artísticos o mecánicas secundarias. Por último, se han definido una serie de características mínimas que debe cumplir en el prototipo:

Característica 1. Debe ser una aplicación para PC, por lo que el diseño hecho originalmente para móvil, debe adaptarse a este medio.

Característica 2. Debe tener al menos tres estados, con sus pantallas correspondientes: un menú de inicio, uno de pausa y el estado de juego. Estos deben tener unos diseños similares a los mostrados en la Figura 3.0.1.

Característica 3. Debe poder observarse en todo momento el funcionamiento de los NPCs enemigos. Se debe diseñar e implementar una interfaz que muestre su estado actualizado.

Característica 4. Debe tener el funcionamiento básico de un juego endless runner: el jugador avanzará automáticamente por un camino con varios carriles y obstáculos que tendrá que esquivar moviéndose a los lados.

Característica 5. Debe tener la mecánica de girarse a vigilar o espantar a los animales para que estos se alejen del jugador.

Para poder definir claramente los objetivos y requisitos del proyecto, se ha modelado junto a la cliente un diagrama de casos de uso de la aplicación (ver Figura 3.0.2).

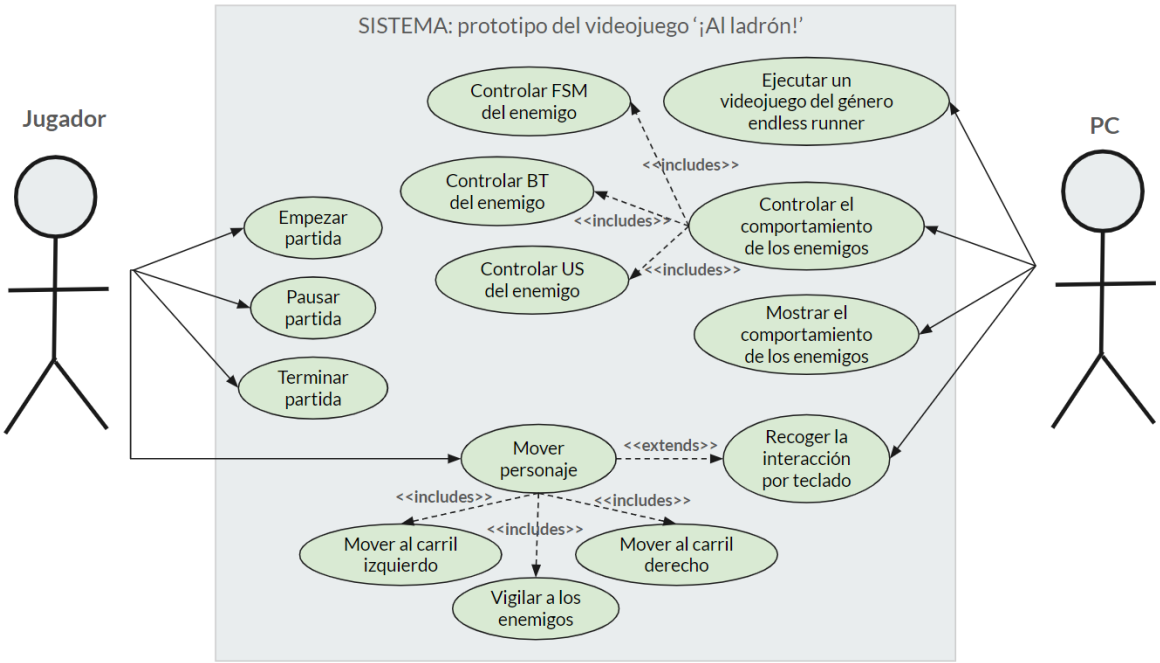


Figura 3.0.2: Diagrama de casos de uso del prototipo.

Con toda esta información, se pueden establecer un objetivo principal y varios objetivos específicos en este proyecto. El **objetivo principal** de este proyecto es comparar tres de las distintas técnicas de implementación de comportamientos en videojuegos, para decidir cuál de ellas sería la más conveniente a la hora de programar los NPCs del videojuego '¡Al ladrón!'. Más concretamente, se van a cumplir los siguientes **objetivos específicos**:

Obj 1. Implementar un prototipo del juego '¡Al ladrón!' que cumpla los requisitos mínimos acordados con la cliente para poder probar en este los comportamientos de los personajes.

Obj 2. Diseñar e implementar tres algoritmos de comportamientos

Obj 3. Evaluar y comparar las distintas técnicas desarrolladas para el caso concreto.

Para conseguir estos objetivos, se han planteado una serie de requisitos funcionales (abreviados como RF) y no funcionales (indicados como RNF) para el proyecto. En la Tabla 3.2 se muestran todos ellos y se indica a qué objetivo están asociados.

A lo largo de este trabajo se presenta el proceso llevado a cabo para conseguir los objetivos anteriores, además de los resultados obtenidos para los mismos.

Obj 1			
RF-1.1	El prototipo debe ser una aplicación para PC.	RF-1.6	Tener diseños de interfaces para pantallas horizontales.
RF-1.2	Tener al menos tres estados: menú de inicio, pausa y estado de juego.	RF-1.7	Control por teclado y ratón de toda la aplicación.
RF-1.3	Poder observarse en todo momento el funcionamiento de los NPCs enemigos.	RF-1.8	Tener un diseño de pantalla por estado, con una estética similar a la que se muestra en la Figura 3.0.1.
RF-1.4	Tener el funcionamiento básico de un videojuego del género endless runner: el jugador avanzará automáticamente por un camino con varios carriles y obstáculos que tendrá que esquivar moviéndose a los lados.	RF-1.9	Tener una interfaz durante el juego que muestre el estado de los personajes en todo momento, formada por una vista cenital con la que se pueda ver su avance y un cuadro de texto que indique su estado.
RF-1.5	Tener la mecánica de girarse a espantar a los animales para que estos se alejen del jugador.	RF-1.10	Crear un escenario 3D formado por geometrías básicas que sirva para representar el escenario y a los personajes.
RNF-1.1	Las acciones del usuario se deben ver reflejadas rápidamente en el prototipo (en menos de 2 segundos).	RNF-1.2	Tener una interfaz intuitiva para facilitar su uso.
Obj 2			
RF-2.1	Diseñar e implementar el comportamiento del enemigo utilizando una FSM.	RF-2.3	Diseñar e implementar el comportamiento del enemigo utilizando un US.
RF-2.2	Diseñar e implementar el comportamiento del enemigo utilizando un BT.	RNF-2.1	Los enemigos deben tardar menos de 3 segundos en reaccionar a las acciones del jugador y al entorno del juego.
Obj 3			
RF-3.1	Comparar las tres técnicas de implementación de comportamientos para este caso concreto.	RNF-3.1	La comparación de las tres técnicas debe presentarse de una forma clara y fácil de comprender.

Tabla 3.2: Requisitos planteados para el proyecto

Capítulo 4

Metodología

> "*Un modelo de desarrollo de SW determina el orden en el que se llevan a cabo las actividades del proceso de desarrollo de SW, es decir, es el procedimiento que se sigue durante el proceso*"(Ojeda y Fuentes, 2012).

Se conocen como metodologías de desarrollo software a las distintas formas de afrontar un proyecto, planificarlo y llevarlo a cabo. Una de las clasificaciones más importantes es la diferenciación entre metodologías tradiciones y metodologías ágiles.

Las **metodologías tradicionales** organizan el trabajo entendiendo el proyecto como un total, sin divisiones, de forma que los requerimientos del proyecto se definen al principio y no se hacen apenas cambios en los mismos (Montero, Cevallos, y Cuesta, 2018). En los proyectos en los que el trabajo se desarrolla de forma tradicional, se suele dar una gran importancia a la documentación y no se recibe casi feedback del cliente. Además, las pruebas y evaluación del producto se realizan al terminar el mismo. Esta situación hace que el trabajo sea más rígido y los problemas puedan ser más difíciles de solucionar, ya que se detectan en fases muy avanzadas de trabajo.

La metodología más popular de este grupo es el desarrollo en cascada, que propone cinco fases: análisis de requerimientos, diseño, implementación, verificación y mantenimiento; que se realizan en ese orden y de manera secuencial (Montero y cols., 2018). Su nombre viene dado por su representación gráfica (Figura 4.0.1).

Por otra parte, las **metodologías ágiles** dan mayor importancia a la retroalimentación del cliente y a solucionar los problemas que puedan surgir, ya sea a nivel de diseño o de desarrollo, en fases tempranas del desarrollo (Canós, Letelier, y Penadés, 2003). La metodologías de esta familia son muy útiles en proyectos cuyos requisitos puedan sufrir cambios durante el desarrollo, ya sean por avances tecnológicos o por petición del cliente. Además, una de sus características principales es que se divide el proyecto a realizar en tareas más pequeños, de forma que sea más fácil afrontarlas y revisar su progreso. Son especialmente populares en proyectos de desarrollo software, ya que sus características pueden cambiar muy rápido debido a avances tecnológicos y suele ser interesante recibir retroalimentación del cliente a lo largo del desarrollo.

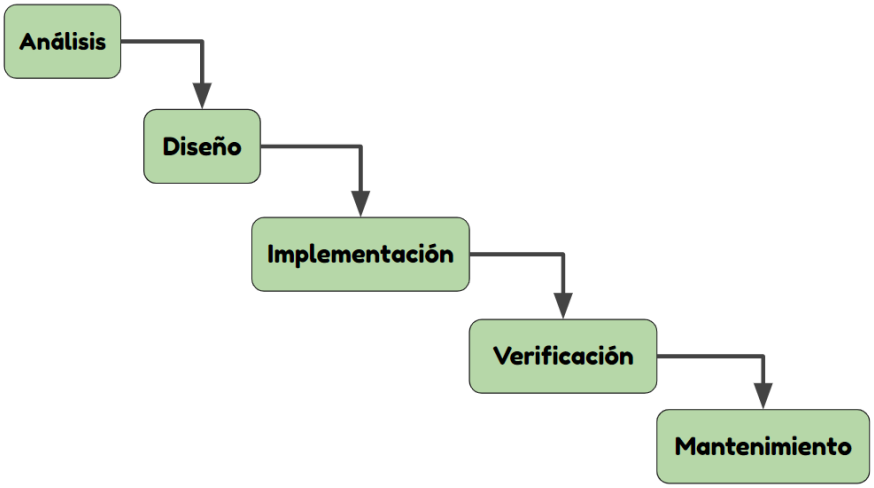


Figura 4.0.1: Fases del desarrollo en cascada

Una de las metodologías ágiles más conocidas es **Scrum**. Los equipos que la utilizan están formados por pocas personas, entre ellas un Scrum Master que se encarga de que todo el equipo conozca bien el funcionamiento de esta metodología y la aplique.

El desarrollo del proyecto se organiza en varios **sprints**, también llamados **iteraciones**. Un sprint es un corto periodo de tiempo en el que se completan algunas de las tareas del proyecto. Al inicio de cada sprint se debe definir las tareas que se van a realizar y el tiempo que va a durar este. Lo ideal es que un sprint dure alrededor de cuatro semanas, pero esto puede variar dependiendo de las necesidades de cada equipo y cada proyecto. Durante este tiempo, el equipo tiene reuniones diarias en las que comentar los avances y dificultades que hayan podido encontrar, y al final de este, se revisarán las tareas realizadas y el trabajo del equipo durante el sprint, y se planificará el alcance de la siguiente iteración (Schwaber y Sutherland, 2013). Además, al terminar cada iteración se presentarán al cliente los avances realizados, para recibir feedback y hacer algún cambio, si fuera necesario.

Después de analizar las características más relevantes de las metodologías de desarrollo software más utilizadas en la actualidad, se ha decidido seleccionar una metodología ágil. A continuación se presentan las motivaciones:

- El proyecto, por su naturaleza, es fácil de dividir en varias tareas más pequeñas: una por cada una de las técnicas que se van a utilizar para la implementación del comportamiento.
- El hecho de visualizar el proyecto como varias tareas de menor tamaño, facilita su planificación y afrontar el trabajo a realizar.
- La persona encargada del proyecto está muy familiarizada con estas metodologías, por lo que le será sencillo llevarla a cabo.
- Hacer una pequeña revisión de lo realizado cada vez que se finalice una tarea puede ser muy útil para encontrar posibles fallos y estimar el tiempo necesario para terminar el proyecto.

- Seguir una metodología en cascada podría implicar errores al final del desarrollo, momento en el que ya es muy costoso corregirlos.
- Teniendo en cuenta que la persona encargada del trabajo aún no es profesional en el campo, puede ser necesario modificar algunos de los requerimientos del proyecto atendiendo a las posibles dificultades que se encuentren durante el desarrollo. Esta situación es contraria a lo definido en metodologías tradicionales, para las que se plantean los requerimientos al inicio y no deben sufrir cambios.
- La fase de mantenimiento, propia de la metodología en cascada, no tiene sentido en este trabajo, ya que el proyecto a desarrollar es un estudio puntual, no una aplicación software que deba perdurar.

Dado que el equipo de desarrollo está formado únicamente por una persona, no tendría sentido usar la metodología Scrum como tal, ya que todos los roles definidos en esta recaen sobre la misma persona. Sin embargo, sí se van a utilizar algunas herramientas propias de la metodología: se va a dividir el desarrollo en varios sprints, de forma que al final de cada uno de estos se pueda presentar un prototipo funcional. Además, al inicio de cada sprint se revisarán los avances que se hayan hecho en el anterior para poder ajustar mejor su alcance, por ejemplo, definiendo mejor los tiempos de trabajo.

Para planificar las tareas que formarán parte de cada sprint se va a hacer uso de la herramienta 'Trello', en la cuál se creará un tablero con varias columnas: una para cada sprint, una '*To Do*', otra '*Doing*' y una última '*Done*', para poder ver claramente el trabajo realizado y lo que falte por realizar. Esta forma de representación se conoce como **Tablero Kanban** (ver Figura 4.0.2), y es muy usada en proyecto ágiles (Yacelga y Cabrera, 2022).

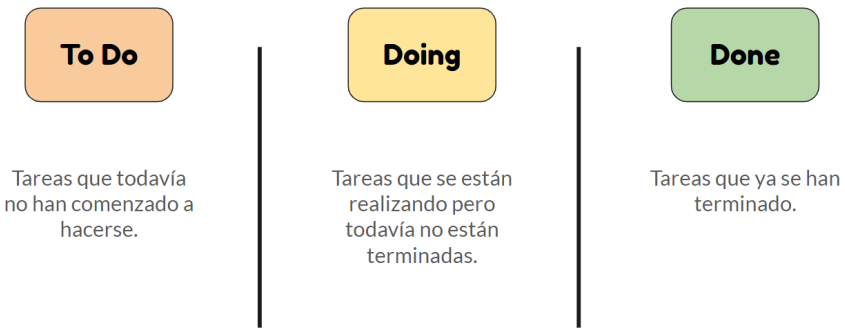


Figura 4.0.2: Ejemplo de tablero Kanban

En resumen, para el desarrollo de este proyecto se va a mantener un marco de desarrollo ágil, dividiendo el desarrollo en varios sprints y se irán revisando sus resultados. Además, se usará un tablero Kanban que ayude a visualizar mejor el trabajo realizado y pendiente.

Capítulo 5

Descripción informática

En el siguiente capítulo se va a detallar el trabajo realizado para este proyecto y las tecnologías usadas en el mismo.

5.1. Solución propuesta

A fin de cumplir los objetivos descritos anteriormente (sección 3), se han detallado una serie de tareas a realizar. Siguiendo la metodología descrita (sección 4), se ha dividido el trabajo total del proyecto en tareas más pequeñas. Estas tareas se han organizado en distintos sprints, intentando que la carga de trabajo en cada uno de ellos fuese similar. Además, se ha hecho una pequeña estimación de tiempo para cada sprint, sin olvidar que esta puede cambiar en las revisiones de cada uno de ellos. Por último, se ha creado un tablero Kanban en el que monitorizar el avance del trabajo en cada sprint.

Teniendo esto en mente, el desarrollo del proyecto se ha dividido en cuatro sprints, todos ellos de una duración estimada de unos 7 días.

- **Sprint 1**
  Este primer sprint se centra en la implementación de un prototipo en el que probar el funcionamiento de los NPCs enemigos. A lo largo de esta primera iteración se deben hacer dos tareas: establecer las características que tendrá este prototipo, partiendo de la información provista por el GDD, y de los requisitos acordados con la cliente (ver 3), y desarrollar el prototipo a partir de estas. Al finalizar este sprint quedará resuelto el primer objetivo específico:

  > **Obj 1**
  > Implementar un prototipo del juego '¡Al ladrón!' que cumpla los requisitos mínimos acordados con la cliente.

  Además de los requisitos RF-1.1, RF-1.2, RF-1.3, RF-1.4, RF-1.5, RF-1.6, RF-1.7, RF-1.8, RF-1.9, RF-1.10, RNF-1.1 y RNF-1.2, todos ellos detallados en la Tabla 3.2.

- **Sprint 2**
  Se centra en el diseño e implementación del primer comportamiento, utilizando una máquina de estados. Este sprint corresponde con el requisito funcional RF-2.1, pero también se tendrá en cuenta lo establecido en el requisito no funcional RNF-2.1.

- **Sprint 3**
  El segundo comportamiento a implementar será con un sistema de utilidad. Este sprint es algo más largo que el anterior, ya que se espera que el desarrollo de este comportamiento lleve más tiempo por su dificultad. En este sprint se completará el requerimiento RF-2.3, atendiendo, igual que en el sprint anterior, a lo establecido en el requisito RNF-2.1.

- **Sprint 4**
  En el último sprint, se lleva a cabo el comportamiento restante, usando un árbol de comportamiento. Se ha decidido dejar esta técnica para el final porque se espera que sea la más fácil de implementar, por lo que llevará menos tiempo y se puede realizar junto a ella la última tarea: evaluar y comparar todas las técnicas usadas. Estas dos tareas corresponden a los dos subobjetivos restantes:

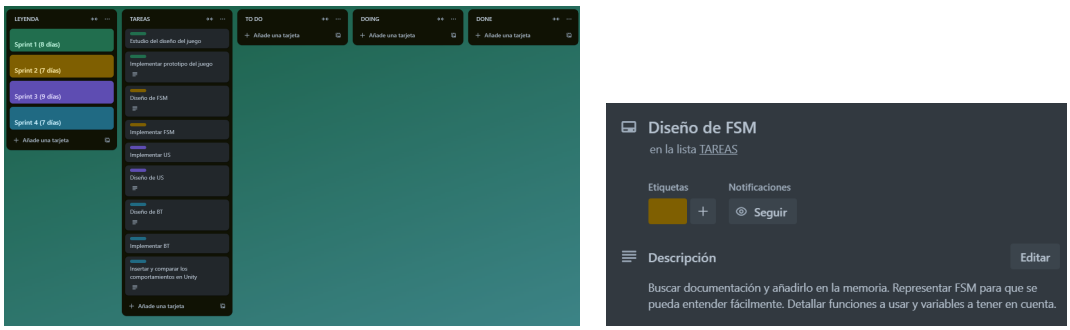
  Obj 2
  Diseñar e implementar tres algoritmos de comportamientos.
  Obj 3
  Evaluar y comparar las distintas técnicas desarrolladas para el caso concreto.

  A su vez, en este sprint se completarán los últimos requerimientos: RF-2.2, RNF-2.1, RF-3.1 y RNF-3.1.

Al finalizar cada sprint se hará una revisión del mismo, que ayudará a mejorar los fallos de organización que se puedan haber cometido a la largo del trabajo, de cara a próximas tareas; y a ajustar las estimaciones de tiempo de los siguientes, si caso de que fuese necesario. En este revisión, además, se mostrarán los resultados al cliente para recibir su feedback.

Todo lo mencionado se ve reflejado en un tablero creado en la herramienta Trello, un software para organización de proyectos. Este software permite crear tableros personalizados con las columnas y tarjetas necesarias. Para este proyecto, se ha creado: una columna con una leyenda de colores para diferenciar las tareas que pertenecen a cada sprint; una columna con todas las tareas a realizar; y las columnas típicas de un tablero Kanban. Se muestra este tablero en la Figura 5.1.1a.

En algunas de las tarjetas del tablero se han añadido detalles de la tarea a realizar (ver Figura 5.1.1b).



(a) Tablero Kanban (b) Tarjeta con descripción de la tarea

Figura 5.1.1: Elementos del Tablero Kanban

5.2. Sprints

En esta sección se mostrarán los detalles de cada sprint y su resultado. Se describirá el trabajo llevado a cabo en cada una de las iteraciones y se valorarán los fallos que se hayan podido tener, ya sea en el trabajo o en la planificación.

5.2.1. Sprint 1

Durante el primer sprint del proyecto se han planificado las tareas relacionadas con la creación del prototipo del juego.

Descripción del prototipo

La primera tarea de este sprint consiste en describir las características que debe tener el prototipo que se va a desarrollar, considerando los requisitos establecidos (ver Tabla 3.2), la información aportada por el documento de diseño del juego (Fresno, 2024) y los casos de uso definidos en la Figura 3.0.2.

En primer lugar, se van a definir algunas adaptaciones del diseño original, poniendo atención a que el objetivo de este proyecto está centrado en la programación de los NPCs, y no en la del juego.

La primera de ellas: en lugar de desarrollar el juego para dispositivos móviles, será para PC, como se acordó con la cliente, y como se especifica en el requerimiento RF-1.1. Esto implica algunos cambios en los controles y las interfaces.

Por su parte, las interfaces del juego serán adaptadas para pantallas horizontales. Además, se ha decidido eliminar la interfaz de fin de partida, de forma que únicamente existirán los estados mínimos recogidos en el segundo requisito funcional y que permitirán los casos de uso de 'Empezar partida', 'Pausar partida' y 'Terminar partida'.

Por lo tanto, se va a desarrollar una aplicación con tres estados, correspondientes a tres pantallas: un menú principal, una pantalla de juego y un menú de pausa. De forma que se elimina la pantalla de fin de partida del juego original. El flujo de pantallas, por lo tanto, queda como se puede ver en la Figura 5.2.1.

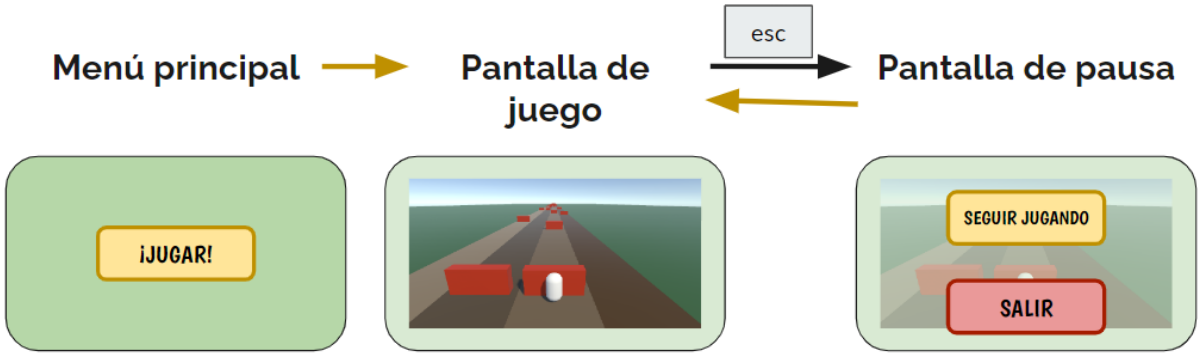


Figura 5.2.1: Flujo de pantallas del prototipo a desarrollar

Con estos cambios, se completarán los requisitos RF-1.6, RF-1.2, RF-1.8 y RNF-1.2.

Otro cambio que se incluirá, es el motivado por los requerimientos RF-1.3 y RF-1.9: se van a añadir un par de elementos nuevos a la pantalla de juego. En primer lugar, una segunda cámara que ofrezca una visión cenital que irá siguiendo al jugador. Gracias a esta cámara se podrá tener todo el tiempo a la vista a los personajes y al jugador, ya que con la cámara 'de juego' no se deberían ver a los NPCs hasta que estos no estén muy cerca. El otro elemento es un pequeño cuadro de texto que informe de qué acción están realizando los NPCs en todo momento. El objetivo de estos dos elementos es poder observar claramente el funcionamiento de los personajes, comprobar que no haya fallos y compararlos entre ellos, además de hacer posible el caso de uso 'Mostrar el comportamiento de los enemigos'.

Por último, teniendo en cuenta los requisitos RF-1.4, RF-1.5, RF-1.7, RF-1.10 y RNF-1.1, correspondientes a los casos de uso 'Ejecutar un videojuego del género endless runner', 'Recoger la interacción por teclado', 'Mover al personaje' y los incluidos en este, se han decidido dejar algunas mecánicas fuera del prototipo. Además de las adaptaciones mencionadas para los requisitos anteriores, el prototipo tendrá las siguientes características:

1. Un escenario y personajes modelados con formas básicas, para que se puedan distinguir bien unos de otros, pero sin ahondar en detalles.

2. Una jugabilidad básica: un escenario compuesto por cuatro carriles por los que el jugador podrá moverse y donde se encontrará obstáculos. El jugador avanzará cada vez más deprisa, subiendo la dificultad del juego.

   Las únicas acciones que podrá realizar el jugador son: moverse de un carril a otro y girarse para 'vigilar' a los animales enemigos. Cuando decida hacerlo, tendrá que estar unos segundos quieto, para poder observar las reacciones de los NPCs. Después de transcurrir este tiempo, seguirá su camino.

   Los controles del prototipo serán los siguientes:

   - Flechas izquierda y derecha, o teclas 'A' y 'D': se usarán para mover al jugador lateralmente, de un carril a otro.
   - 'Espacio': el jugador mirará a los animales.
   - En las pantallas de menú, clic con el ratón para pulsar los distintos botones.

3. Unos personajes - enemigos, cuyo comportamiento consiste en perseguir al jugador, robarle cuando esté lo suficientemente cerca y esconderse si este se gira a vigilar. Estos comportamientos se implementarán en futuros sprints del proyecto.

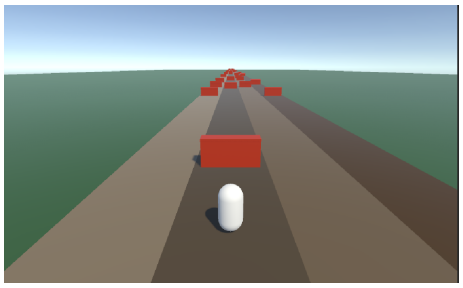
Implementación del prototipo

Para conseguir las características listadas, se ha usado el motor de videojuegos 'Unity' (https://unity.com/es), un software gratuito especializado en la creación de videojuegos. En primer lugar, se creó una escena vacía y varios paneles de UI en los que se han diseñado los menús (principal y de pausa) y la interfaz requerida para observar el estado de los NPCs. Además, se hará uso de un script 'GameManager', para controlar los distintos estados que puede haber en el juego. Estos estados son:

- **Menu State**: estado en el que se encuentra el juego si el jugador está en el menú principal.

- **Game State**: ocurre mientras el jugador está jugando la partida.
- **Pause State**: estado de pausa del juego, donde el jugador puede decidir si seguir jugando o si salir.

(a) Menú principal



(b) Interfaz en partida

(c) Menú de pausa

Figura 5.2.2: Pantallas e interfaces del prototipo

Para navegar entre una interfaz y otra, se ha creado el script 'UIManager', que se encarga de recibir la interacción con los botones y hacer los cambios correspondientes en el estado del juego. Además, se han creado métodos para poder activar el panel necesario en cada estado desde el GameManager. Esto último se hace con un `switch` (líneas 10-32) mostrado en el bloque de código 5.1.

```
1   public enum GameState {
2       MenuState,
3       GameState,
4       PauseState
5   }
6
7   public void UpdateGameState(GameState newState) {
8       State = newState;
9
10      switch(newState) {
11          case GameState.MenuState:
12              Time.timeScale = 0f;
13              UIManager.Instance.MenuActive(true);
14              UIManager.Instance.EndMenuActive(false);
15              break;
16
17          case GameState.GameState:
18              Time.timeScale = 1.0f;
19              player.ResetPlayer();
20              UIManager.Instance.MenuActive(false);
21              UIManager.Instance.EndMenuActive(false);
```

```
                break;

            case GameState.PauseState:
                Time.timeScale = 0f;
                UIManager.Instance.MenuActive(false);
                UIManager.Instance.EndMenuActive(true);
                break;

            default:
                break;
        }
    }
}
```

Listado de código 5.1: Gestión de estados e interfaces desde el GameManager

Gracias a esto, los jugadores podrán empezar, pausar y terminar una partida, por lo que se dan por cubiertos esos casos de uso y los requisitos correspondientes (RF-1.2, RF-1.6, RF-1.8 y RNF-1.2).

Por otra parte, se creó un escenario con objetos 3D de Unity, como indica el requisito RF-1.10. Se han usado varios planos para delimitar los cuatro carriles por los que se puede mover el jugador, cubos como obstáculos y una cápsula blanca para representar al jugador.

La interacción del jugador se ha codificado en el script PlayerMovement. Este script recoge las acciones en el teclado del jugador y se mueve su personaje a izquierda o derecha, siempre que queden carriles en esa dirección. Además, desde el inicio de la partida el jugador se moverá hacia delante, aumentando su velocidad progresivamente, de forma que la dificultad aumente. En resumen, se ha implementado el funcionamiento básico de un juego endless runner, como se indicaba en el requisito RF-1.4.

Por último, se ha implementado la acción de vigilar a los animales cuando el jugador pulse la tecla 'espacio'. En esta acción se ve involucrado tanto el movimiento del personaje, que se parará en el sitio durante un segundo, como el movimiento de la cámara que se alejará para que el jugador tenga más visibilidad, como se puede ver en la Figura 5.2.3.

Este funcionamiento se ha conseguido gracias a una corrutina en el script de PlayerMovement, cuya función es parar el reloj interno del juego mientras el jugador esté vigilando (ver listado de código 5.2).

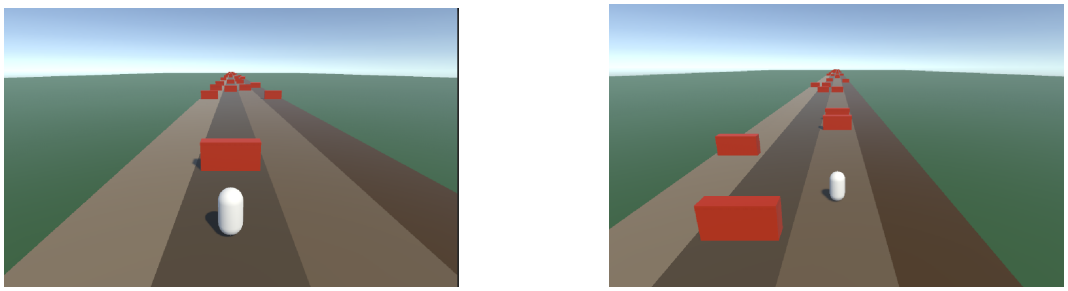
```
    private IEnumerator LookBack() {
        lookingBack = true;
        float t = Time.realtimeSinceStartup;
        Time.timeScale = 0f;

        while(Time.realtimeSinceStartup - t < 1.0f){
            yield return 0;
        }

        Time.timeScale = 1.0f;
        Debug.Log("player finished looking back");
        lookingBack = false;
        yield break;
    }
```

Listado de código 5.2: Corrutina para la acción de girarse a vigilar del jugador.

Por último, se ha incluido un script en la cámara para que realice el movimiento que se puede ver en las imágenes de la Figura 5.2.3. De esta forma, quedan resuelto el requisito RF-1.5, además de todos los casos de uso relacionados con el jugador (ver diagrama 3.0.2).



(a) Vista general (b) Vista vigilando

Figura 5.2.3: Movimiento de la cámara durante la partida

Revisión del Sprint 1

Como se mencionó anteriormente en la metodología (sección 4), al final de cada sprint se realiza una pequeña revisión del mismo, en el que valorar si se estimaron bien los tiempos de trabajo y si el trabajo entregado al final del sprint concuerda con lo esperado.

Se ha comprobado que la estimación de tiempos realizada en un inicio para este sprint era muy optimista, ya que ha llevado más tiempo de lo esperado. Para mejorar esto en futuros sprints y poder ajustarse mejor a la duración real de las tareas, se han añadido tres días de trabajo a cada uno de ellos.

En cuanto al trabajo a realizar, en este sprint se han estudiado y adaptado las características más importantes del juego en un prototipo. Para ello, se han cumplido todos los requisitos acordados con la cliente relacionados con este objetivo excepto el RNF-4 (ver Tabla 3.2). Esta decisión se consultó con la clienta y se acordó aplazar este requisito al siguiente sprint, ya que no se podría probar el funcionamiento de la interfaz diseñada para vigilar a los enemigos, si no había ningún enemigo aún implementado. Se realizará esta interfaz como parte de la tarea 'Implementación de la FSM', en la que se implementará el personaje y una forma de probar su funcionamiento. Por lo demás, el resto de requisitos y las tareas asociadas a ellos se han superado de forma exitosa.

Antes de dar por finalizado este sprint, se tuvo una reunión con la cliente para recibir feedback. En esta, el único cambio que se propuso fue que la acción de vigilar del jugador durase más tiempo para poder ver claramente el comportamiento de los enemigos cuando estos estén implementados. Para ello, simplemente se cambió el tiempo de espera en la corrutina a tres segundos.

Una vez terminadas las tareas definidas en un inicio y tras los cambios hechos, se da por finalizado el primer sprint y se colocan las tareas del nuevo sprint en el tablero Kanban (Figura 5.2.4).

5.2.2. Sprint 2

El segundo sprint se centra en completar los requisitos RF-2.1 (teniendo en cuenta lo establecido en el requisito RNF-2.1) y RF-1.9, detallados en la Tabla 3.2. Para conseguirlo,

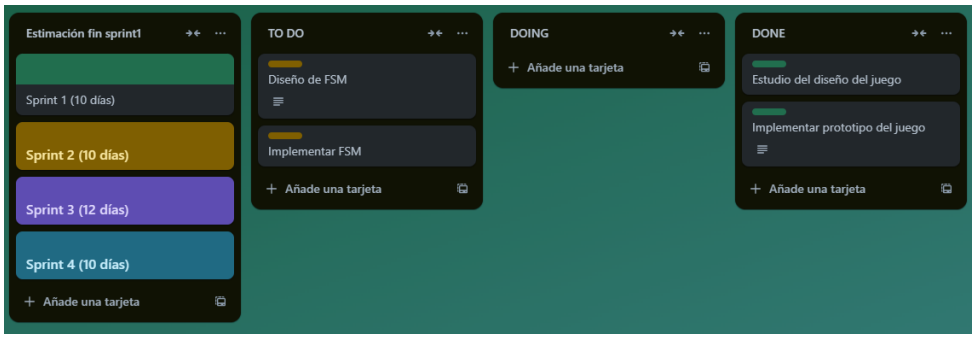


Figura 5.2.4: Tablero Kanban al final del Sprint 1

se han planificado dos tareas a realizar en un tiempo estimado de diez días de trabajo: el diseño de una FSM para el comportamiento de los NPCs, y su implementación en un personaje en el prototipo.

El comportamiento desarrollado con una FSM es el primero de los tres en implementarse. Es por esto que, antes de comenzar a trabajar en su programación como tal, es importante plantear el diseño de los scripts que conformarán los comportamientos de los NPCs.

Siguiendo el paradigma de la Programación Orientada a Objetos, se van a desarrollar los distintos comportamientos partiendo de una base común: una clase con las características básicas de estos personajes. Cada personaje heredará de esta base y especificará su comportamiento a partir de esta. El diagrama de clases de la Figura 5.2.5 refleja esto, además de algunos de los atributos y métodos que tendrá esta clase.

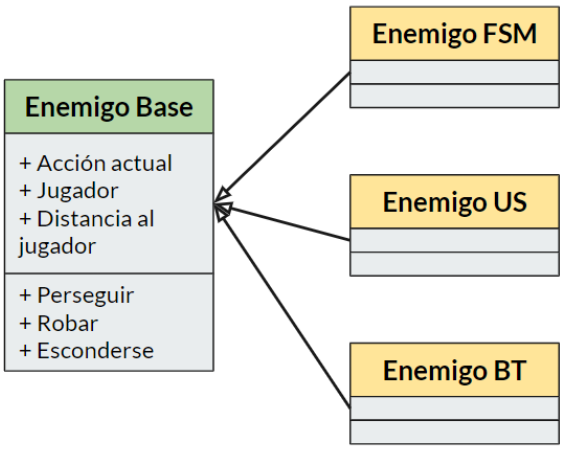


Figura 5.2.5: Diagrama de clases que muestra la herencia entre las clases de los distintos enemigos.

Las máquinas de estados finitas tienen la gran ventaja de que son fáciles de diseñar e implementar para comportamientos sencillos, como es este caso. Por este motivo, se espera que esta técnica sea de las más adecuadas para implementar el comportamiento de los enemigos de este juego.

Diseño de la FSM

Antes de empezar a programar es necesario definir claramente los estados y transiciones de la máquina de estados, y sabiendo que el funcionamiento de los NPCs consiste en perseguir al jugador, robarle si está cerca y esconderse si se gira a vigilarles, se puede definir una FSM con únicamente tres estados:

- **Perseguir**: es el estado en el que se encontrará la mayoría del tiempo el personaje. Este estado consiste en seguir al jugador, a medida que se acorta la distancia con el mismo.
- **Robar**: se llegará a este estado si el animal consigue acercarse lo suficiente al jugador. En este momento, el NPC se acercará rápidamente al jugador y le robará. Según el diseño del juego, esto haría que al jugador le robarán una fruta, perdiendo un punto de vida.
- **Esconderse**: si el jugador se gira a vigilar a los animales, estos intentarán esconderse.

Esta FSM se ha representado en el diagrama de la Figura 5.2.6. Como se puede observar, la FSM está formada por tres estados y varias percepciones que harán que el personaje cambie de un estado a otro.

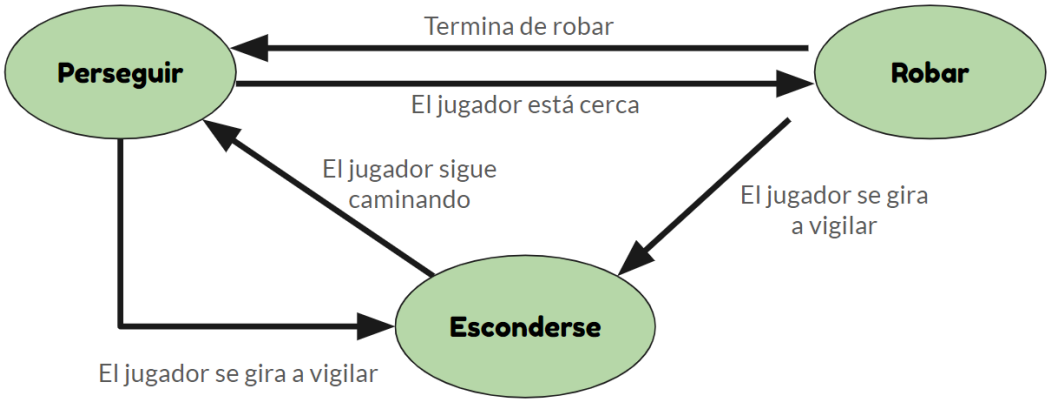


Figura 5.2.6: Comportamiento diseñado con FSM

Implementación de la FSM

Partiendo del planteamiento de la Figura 5.2.5, se ha creado un script (llamado NPCBase) en el que establecer una base común para todos los enemigos. Esta base se compone de: una variable en la que guardar la acción que esté realizando el NPC en ese momento; referencias al jugador y variables para saber a qué distancia se encuentra de éste; y los métodos que modelan las acciones de perseguir, robar y esconderse. Estos últimos deben ser iguales en los tres personajes, de forma que la comparación entre los tres métodos sea justa.

Además, la clase NPCBase hereda de 'Monobehaviour', de forma que los enemigos puedan utilizar las funciones 'Start' y 'Update' para especificar su funcionamiento. Estas

funciones permiten programar acciones para el momento en el que el personaje empiece a ejecutarse, y para instrucciones que deban ejecutarse cada frame. En resumen, el código de la base para todos los enemigos es el mostrado en el bloque 5.3.

```
    public abstract class NPCBase: MonoBehaviour
{
    public NPCState npcCurrent;

    //Player
    public GameObject player;
    public PlayerMovement playerMovement;
    public Vector3 distanceToPlayer0;
    public Vector3 distanceToPlayer;

    public abstract void Start();
    public abstract void Update();

    //ACTIONS (all NPCs will do the same actions)
    public void Chase(){
        npcCurrent = NPCState.Chase;

        //enemy gets progrisevely closer to the player
        distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.01f));
        transform.position = player.transform.position + distanceToPlayer;
        return;
    }

    public void Steal(){
        npcCurrent = NPCState.Steal;

        //finished stealing, comes back to initial distance
        //changes to CHASE state
        if (Mathf.Abs(distanceToPlayer.z) <= 0.2f)
        {
            distanceToPlayer = distanceToPlayer0;
            transform.position = player.transform.position +
   distanceToPlayer;
            npcCurrent = NPCState.Chase;
        }

        //steal
        distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.1f));
        transform.position = player.transform.position + distanceToPlayer;
        return;
    }

    public void Hide(){
        npcCurrent = NPCState.Hide;

        //this enemy will be represented by a bird, so their way to hide is
     fly away
        distanceToPlayer = distanceToPlayer + (new Vector3(0.05f, 0.3f, 0.2
   f));
        transform.position = player.transform.position + distanceToPlayer;
    }
}

public enum NPCState
```

```
{
    Chase,
    Steal,
    Hide
}
```

Listado de código 5.3: Clase base para todos los NPCs

Como se puede ver en el Listado de código (ver 5.3), la programación de las acciones es sencilla:

- Perseguir (*Chase*): consiste en acortar progresivamente la distancia con el jugador.
- Robar (*Steal*): consiste en acercarse más rápido al jugador y volver a la distancia inicial una vez le ha 'tocado'. De esta forma, se puede entender que el enemigo ha robado al jugador y ha salido corriendo.
- Esconderse (*Hide*): consiste en que el personaje salga 'volando'. Se ha decidido hacer esto para simular la animación de uno de los enemigos del juego: el mosquero cardenal. De esta forma, se puede ver claramente en la simulación la acción que está realizando el NPC.

Heredando de esta clase base, se ha creado un script específico para el comportamiento modelado con FSM. En primer lugar, el personaje toma la referencia del jugador para poder guardar la distancia inicial que tiene con él (ver 5.4). Esto servirá para que el personaje vuelva a colocarse a esta distancia después de haber robado o si el jugador le ha espantado. Además, se define su estado inicial en 'Perseguir'.

```
    public override void Start()
    {
        distanceToPlayer0 = transform.position - player.transform.position;
        distanceToPlayer = distanceToPlayer0;
        playerMovement = player.GetComponent<PlayerMovement>();

        npcCurrent = NPCState.Chase;
    }
```

Listado de código 5.4: Primeras instrucciones que ejecuta el script antes de comenzar con el funcionamiento de la FSM.

Una vez hecho esto, comienza la ejecución de la FSM propiamente dicha. Para implementarla, se han programado una serie de funciones que hagan las comprobaciones necesarias en cada momento. En estos métodos, además, si la comprobación indica que se debe hacer la transición, se actualizará la variable que guarda el estado actual. Dado que algunas transiciones de la FSM se ven causadas por lo mismo, estas se pueden representar únicamente con los tres métodos mostrados en el listado de código 5.5.

```
    //checks if  the player is close enough to steal from them.
    //changes to steal state if they are
    bool checkPlayerClose()
    {
        if (Mathf.Abs(distanceToPlayer.z) <= 4.5f)
        {
            npcCurrent = NPCState.Steal;
            return true;
```

```
        }
        return false;
    }

    //check if the player is looking towards the enemies
    //changes to hide state
    bool checkPlayerLooking()
    {
        if (playerMovement.lookingBack)
        {
            distanceToPlayer = distanceToPlayer0;
            npcCurrent = NPCState.Hide;
            return true;
        }
        return false;
    }

    //check if player stopped looking towards the enemies
    //changes the state to chase
    bool checkPlayerStoppedLooking()
    {
        if (!playerMovement.lookingBack)
        {
            distanceToPlayer = distanceToPlayer0;
            transform.position = player.transform.position +
   distanceToPlayer;
            npcCurrent = NPCState.Chase;
            return true;
        }
        return false;
    }
```

Listado de código 5.5: Métodos con comprobaciones que motivan las transiciones de la FSM.

Gracias a estas transiciones y a los métodos programados para cada acción en la clase base (NPCBase), el funcionamiento de la FSM ha sido sencillo de definir. Este funcionamiento se incluye en la función `Update`, ya que debe ir actualizándose constantemente y haciendo las comprobaciones correspondientes. Para modelar la FSM, se utiliza una estructura `switch`, que distinguirá según el estado actual del personaje. En cada uno de los estados, se hacen las comprobaciones propias de ese estado: si alguna de ellas devuelve 'true', se terminará la ejecución del estado, de lo contrario, se ejecuta la acción pertinente. Por lo tanto, el funcionamiento de la FSM queda como se muestra en el código 5.6.

```
    public override void Update()
    {
        if(GameManager.Instance.State != GameState.GameState) return;

        switch(npcCurrent)
        {
            case NPCState.Chase:
                //check if we have to change state
                if(checkPlayerClose() || checkPlayerLooking()) break;

                Chase();
                break;

```

```
            case NPCState.Steal:
                //check if we have to hide
                if(checkPlayerLooking()) break;

                Steal();
                break;

            case NPCState.Hide:
                //check if we can chase
                if(checkPlayerStoppedLooking()) break;

                Hide();
                break;

            default:
                break;
        }
    }
```

Listado de código 5.6: Programación de la FSM.

Una vez creados estos scripts, se ha actualizado el diagrama de clases creado inicialmente con los nombres finales de los atributos y las funciones utilizadas en las transiciones de la FSM. Con todo esto, el diagrama queda como se muestra en la Figura 5.2.7.

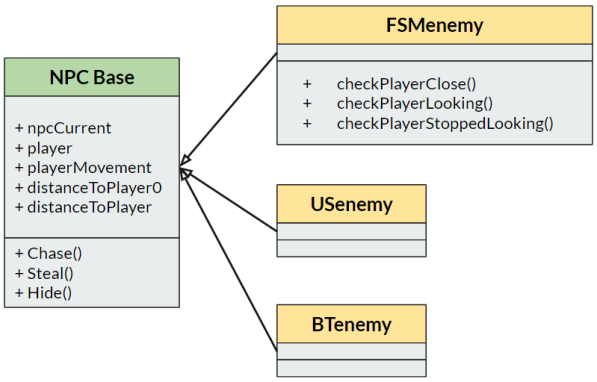
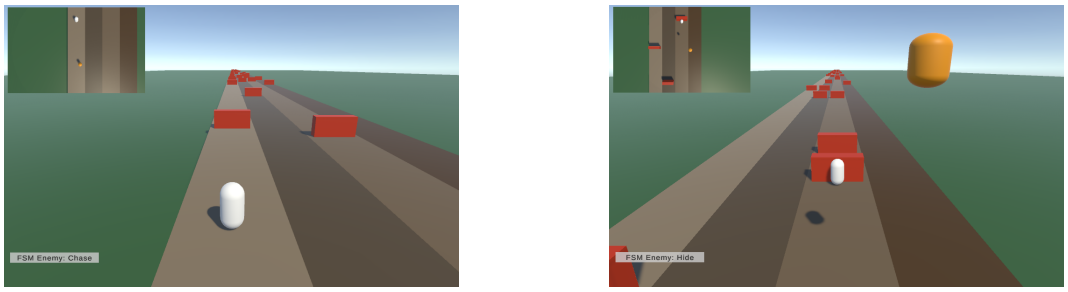


Figura 5.2.7: Diagrama de clases de los enemigos actualizado tras la implementación de la FSM.

Por último, se ha creado una cápsula naranja en Unity que representará este personaje, y se le ha añadido el script. Se ha creado una cámara cenital en la escena y un cuadro de texto cuyo objetivo es mostrar el estado del enemigo en todo momento, de forma que quede cubierto el requisito RNF-4. Se ha hecho una referencia a esta cápsula en el código de la UI, de forma que aparezca su estado actual en pantalla durante el juego (Figura 5.2.8).

Revisión del sprint 2

Para finalizar este sprint, se ha revisado el trabajo realizado y la organización que se había previsto del mismo. En esta segunda iteración se han completado dos tareas: diseño de la FSM e implementación de la misma. Gracias a estas se han concluido los



(a) FSM persiguiendo (b) FSM escondiéndose

Figura 5.2.8: Estados de la FSM en el juego

requisitos RF-2.1 y RF-1.9, dando por terminado así el primer objetivo (ver Tabla 3.2). Sin embargo, en este sprint se ha encontrado un pequeño fallo en la organización. Se ha considerado demasiado amplia la tarea de implementación, ya que esta se podría haber dividido en dos:

- Implementación de una clase base para los NPCs.
- Implementación de la FSM.

Esto no ha supuesto un problema en el desarrollo del proyecto, pero es interesante tenerlo en cuenta para ocasiones futuras. No obstante, este pequeño fallo no ha causado retrasos en la planificación, por lo que esta seguirá sin cambios.

Por último, se mostraron los avances a la cliente y se recibió feeback positivo, sin necesidad de ningún cambio. Así mismo, se ha actualizado el tablero Kanban, como se ve en la Figura 5.2.9.

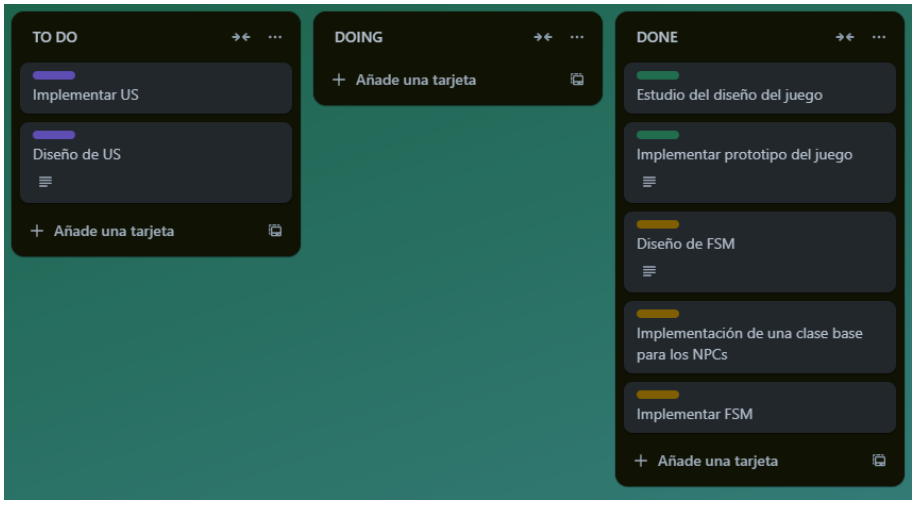


Figura 5.2.9: Tablero Kanban al final del Sprint 2

5.2.3. Sprint 3

Durante el tercer sprint del proyecto se va a centrar en la creación del comportamiento usando un sistema de utilidad. Al final de este, se habrá resuelto el requisito funcional

RF-2.3 (ver Tabla 3.2), correspondiente al caso de uso 'Controlar US del enemigo' (ver Figura 3.0.2). De las tres técnicas, esta es la más compleja, tanto en su diseño como en su implementación. Por este motivo, es probable que se necesiten varias versiones del diseño del comportamiento este enemigo, el cuál se irá refinando con prueba y error. Por este motivo, se estimó que este sprint tendría una duración de doce días, siendo esta la iteración más larga del proyecto.

Diseño del US - primera versión

Como se explica en la sección 2.3, un sistema de utilidad cuenta con una serie de posibles acciones entre las que elegirá una, dependiendo de cuál de ellas le aporte una mayor utilidad. La utilidad de cada una de las acciones se calculará con una suma ponderada de los factores de decisión de cada una de ellas. Por su parte, la utilidad de estos factores se calculará gracias a una curva que los relacione con alguna variable del juego.

Este personaje debe decidir entre las tres acciones que pueden realizar los enemigos de '¡Al ladrón!': perseguir al jugador, robarle o esconderse. Para decidir cuál de ellas hacer, se va a tener en cuenta su distancia al jugador y si este está vigilando o no. De la distancia con el jugador se obtienen tres curvas de utilidad, todas ellas con valores acotados entre 0 y 1:

- La que representa la **facilidad para robar**: cuánta menos distancia haya con el jugador, mayor será esta. Esta curva decrece de una forma muy brusca, ya que se quiere que el enemigo únicamente realice esta acción si está muy cerca del jugador. Se representa esta curva en la Figura 5.2.10 y con la función 5.2.1.

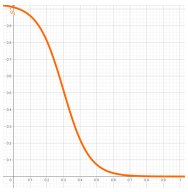


Figura 5.2.10: Curva de utilidad de la facilidad para robar en relación con la distancia a la que se encuentra el jugador.

$$f(x) = 1{,}03/(1 + e^{13*(x-0{,}3)}) \tag{5.2.1}$$

- La curva que muestra modula la **seguridad** que siente el personaje. Cuanta más distancia haya con el jugador, más seguro y tranquilo se sentirá el enemigo. Esta curva se ha modelado con una función lineal creciente, que está representada con la función 5.2.2 y Figura 5.2.11. Como se puede ver, esta función cambia de una forma mucho más suave y constante que la anterior, ya que no se busca un cambio brusco en esta característica.

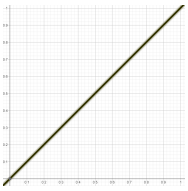


Figura 5.2.11: Curva de utilidad de la seguridad del personaje dependiendo de la distancia a la que se encuentra el jugador.

$$f(x) = x \tag{5.2.2}$$

- La última curva relacionada con la distancia del enemigo con el jugador es la que representa la **inseguridad** del NPC. Esta curva representa lo contrario a la anterior, pero es un poco distinta, ya que no es una función constante (función 5.2.3 y Figura 5.2.12).

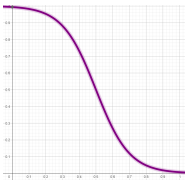


Figura 5.2.12: Curva de utilidad de la inseguridad del personaje según su distancia con el jugador.

$$f(x) = 1/(1 + e^{10*(x-0,5)}) \tag{5.2.3}$$

Todas las curvas anteriores se han modelado con el software 'Geogebra' (https://www.geogebra.org/, y se han limitado al intervalo [0, 1]. Por último, se ha creado manualmente una función que representa el **miedo** que tendrá el personaje en relación con si el jugador está vigilando o no. Esta función se muestra en la Figura 5.2.13 y, como se puede ver, tendrá valor 0 si el jugador no está vigilando, y 1 si sí lo está haciendo.

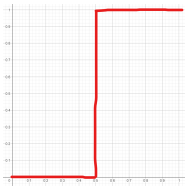


Figura 5.2.13: Curva de utilidad del miedo del personaje dependiendo de si el jugador está vigilando o no.

Una vez definidas las curvas de utilidad del sistema, se deben relacionar estas con cada una de las acciones del personaje. Para conseguir el comportamiento deseado se han definido las siguientes relaciones:

- La acción de **robar** depende únicamente de la **facilidad para robar** del personaje. De forma que, si el personaje está muy cerca, la utilidad de esta acción será muy alta.

- La acción de **perseguir** está relacionada con la **seguridad** del personaje, muy alta si está lejos del jugador, y el **miedo**. Con estas dos curvas se busca que el personaje persiga al jugador hasta que esté lo suficientemente cerca para robar, donde la otra curva ofrecerá mayor utilidad. Además, si el jugador se gira a vigilar pero el NPC está muy lejos, podrá seguir avanzando un poco, hasta que la acción de huir aporte una mayor utilidad.
- La acción de **esconderse** se ve condicionada por la **inseguridad** y el **miedo** del personaje, de forma que, si el jugador se gira a vigilar y el NPC está cerca, esta acción será la que más utilidad tenga.

Este comportamiento se ha representado con el diagrama de la Figura 5.2.14.

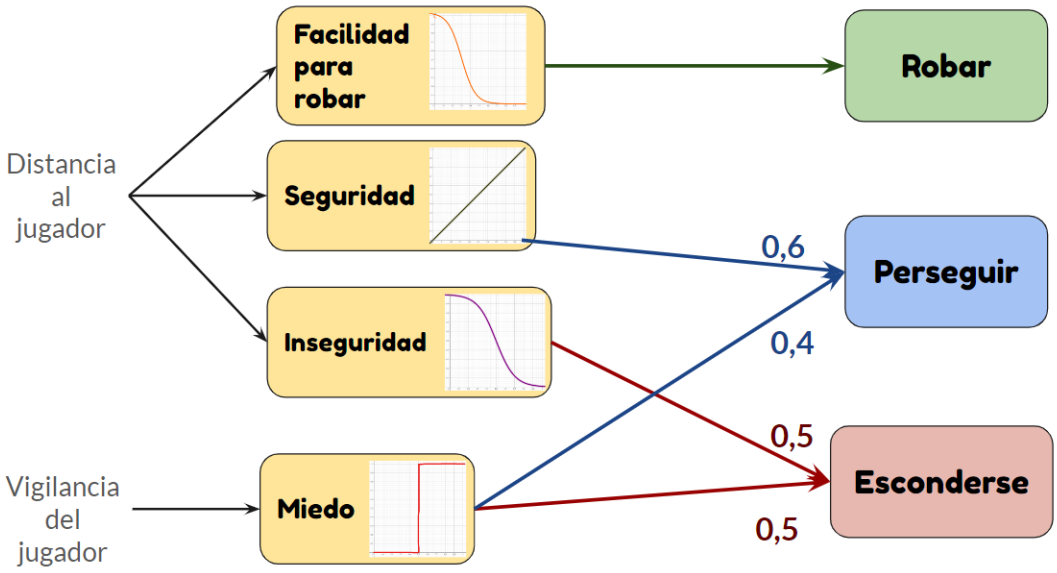


Figura 5.2.14: Sistema de Utilidad del personaje - primera versión

Implementación del US - primera versión

A partir del diseño de la Figura 5.2.14, se ha programado el comportamiento en el script 'USenemy', que hereda de la clase base creada en el primer sprint (NPCBase). Para la programación del sistema de utilidad, en primer lugar se han definido unas variables que servirán para calcular y poder comparar la utilidad de cada acción.

Como se puede ver en el bloque de código 5.7, se ha creado una variable por cada curva de utilidad definida en el diagrama excepto para la de facilidad para robar, ya que al ser la única función que afecta a esta acción, se calculará directamente en la variable de la utilidad final.

```
1     //utility of each action
2     [SerializeField] double stealingUtility = 0;
3     [SerializeField] double chasingUtility = 0;
4     [SerializeField] double hidingingUtility = 0;
5
6     //utility curves
7     private double security;
8     private double insecurity;
```

```
9       private double fear;
```

Listado de código 5.7: Variables que se usarán en el cálculo de la utilidad de las acciones.

Una vez definidas estas variables, se ha programado el US en la función `Update`. Aquí se ha definido la función de cada curva para que se calculen en cada frame. Con estos valores, se ha hecho el cálculo de la utilidad de cada acción, cuyo valor se ha limitado entre 0 y 1 con la función `Math.Clamp(valor, min, max)`. Por último, se llama a un método cuya tarea es comparar los valores de las tres utilidades y ejecutar la acción correspondiente. Todo esto se muestra en el listado de código 5.8.

```
1       public override void Update()
2       {
3           if(GameManager.Instance.State != GameState.GameState) return;
4
5           //curves
6           security = (distanceToPlayer.z / distanceToPlayer0.z);
7           insecurity = 1/(1 + Math.Exp(-10*((distanceToPlayer.z /
    distanceToPlayer0.z) - 0.5)));
8           fear = playerMovement.lookingBack ? 1 : 0;
9
10
11          //utility functions
12          //the results are clamped between 0 and 1
13          stealingUtility = Math.Clamp(1.03 / (1 + Math.Exp(13*(
    distanceToPlayer.z / distanceToPlayer0.z) - 0.3)), 0, 1);   //this is
    also the curve for the factor ease to setal
14          chasingUtility = Math.Clamp(0.6*security + 0.4*fear, 0, 1);
15          hidingingUtility = Math.Clamp(0.5*insecurity + 0.5*fear, 0, 1);
16
17          ChooseAction(stealingUtility, chasingUtility, hidingingUtility);
18      }
19
20      void ChooseAction(double stealUt, double chaseUt, double hideUt) {
21
22          if((chaseUt >= stealUt) && (chaseUt >= hideUt)) {
23              Chase();
24          } else if((stealUt > chaseUt) && (stealUt >= hideUt)) {
25              Steal();
26          } else {
27              Hide();
28
29              //before finishing this action, enemy goes back to initial
    position
30              distanceToPlayer = distanceToPlayer0;
31              transform.position = player.transform.position +
    distanceToPlayer;
32          }
33          return;
34      }
```

Listado de código 5.8: Métodos que modelan el funcionamiento del US.

Para finalizar esta implementación, se ha creado un elemento nuevo en la escena: una cápsula de color morado. Este objeto representa al enemigo y tendrá como componente el script USenemy. Además, en la interfaz de la partida se ha insertado la referencia a este enemigo y se muestra su estado en pantalla, como se puede ver en la Figura 5.2.15.

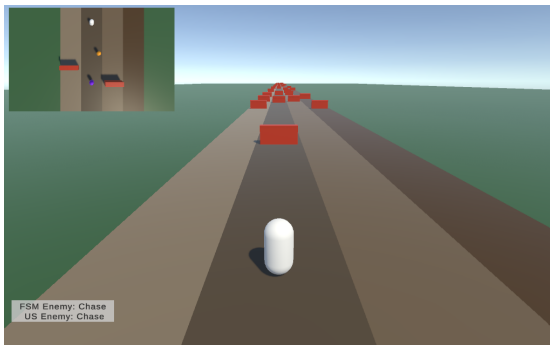


Figura 5.2.15: Implementación del NPC usando un sistema de utilidad

Como se advirtió antes de comenzar a diseñar este personaje, los US requieren mucha prueba y error hasta llegar al diseño final. Es por esto que se dedicó bastante tiempo a comprobar que el NPC estaba actuando como se esperaba. Sin embargo, se encontró un error: el personaje nunca se llega a acercar lo suficiente como para robar. Observando atentamente, se puede ver que cuando el personaje se acerca a una determinada distancia, corre a esconderse en lugar de seguir avanzando hacia el jugador, por lo que se deduce que el problema se encuentra en las curvas o las ponderaciones de la **seguridad** y la **inseguridad** del personaje. Siguiendo este hilo, se va a proponer un segundo diseño y se va a implementar.

Diseño del US - segunda versión

Para solucionar el problema de la primera versión, se ha optado por cambiar las ponderaciones de la acción de esconderse para que cobre más importancia el hecho de que el jugador esté vigilando que la distancia hasta él. Con este cambio, el diagrama queda como se muestra en la Figura 5.2.16.

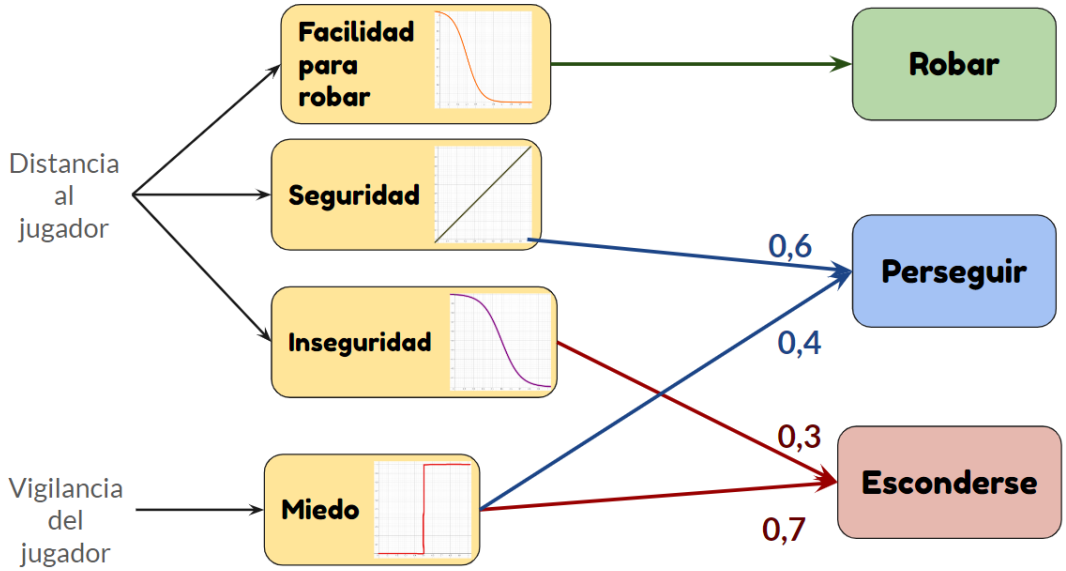


Figura 5.2.16: Sistema de Utilidad del personaje - segunda versión

Implementación del US - segunda versión

El cambio introducido respecto de la primera versión únicamente afecta a una línea en el script del personaje (ver 5.9).

```
1    hidingingUtility = Math.Clamp(0.3*insecurity + 0.7*fear, 0, 1);
```

Listado de código 5.9: Cambio introducido en la segunda versión del US.

De nuevo, se han realizado varias pruebas para validar el comportamiento implementado. Esta vez se han encontrado dos fallos que mejorar:

1. El enemigo no roba al jugador hasta que no está muy cerca de éste.
2. En el momento en que el personaje va a esconderse, su distancia con el jugador aumenta considerablemente. Este comportamiento hace que la seguridad también lo haga, por lo que la acción de esconderse no se llega a ejecutar correctamente y no se aprecia bien qué le sucede al personaje. Además, en este momento los valores de la utilidad de ambas acciones es muy similar, por lo que el personaje se queda unos instantes alternando muy rápido entre las dos, y por lo tanto, no haciendo ninguna.

Para intentar solucionar estos problemas, se van a hacer las siguientes modificaciones en la siguiente versión:

1. Cambiar la curva de utilidad de la facilidad para robar, para que llegue a valores altos de utilidad cuando se encuentre más lejos del jugador.
2. Modificar en 0.05 la ponderación de la acción de perseguir, para que la seguridad tenga una mayor importancia en esta.
3. Implementar una pequeña **inercia** en el US (C. Garre, s.f.-b). La inercia evitará el momento de alternancia entre las acciones. Esta se implementará con mediciones de tiempo, que obliguen a que cada acción que se realice dure como mínimo un segundo.

Diseño del US - tercera versión

Una vez más, se han hecho los cambios requeridos en el diseño. El primero de ellos ha sido sustituir la curva de la facilidad para robar, por la modelada por la función 5.2.4, representada en la siguiente Figura 5.2.17.

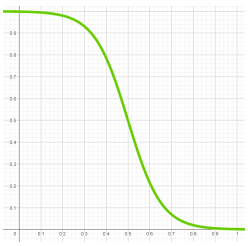


Figura 5.2.17: Curva de utilidad que relaciona la facilidad para robar con la distancia con el jugador.

$$f(x) = 1/(1 + e^{x-0,4}) \tag{5.2.4}$$

Incluyendo también el cambio en las ponderaciones para la acción de perseguir, el diagrama de la tercera versión queda como se muestra en la Figura 5.2.18.

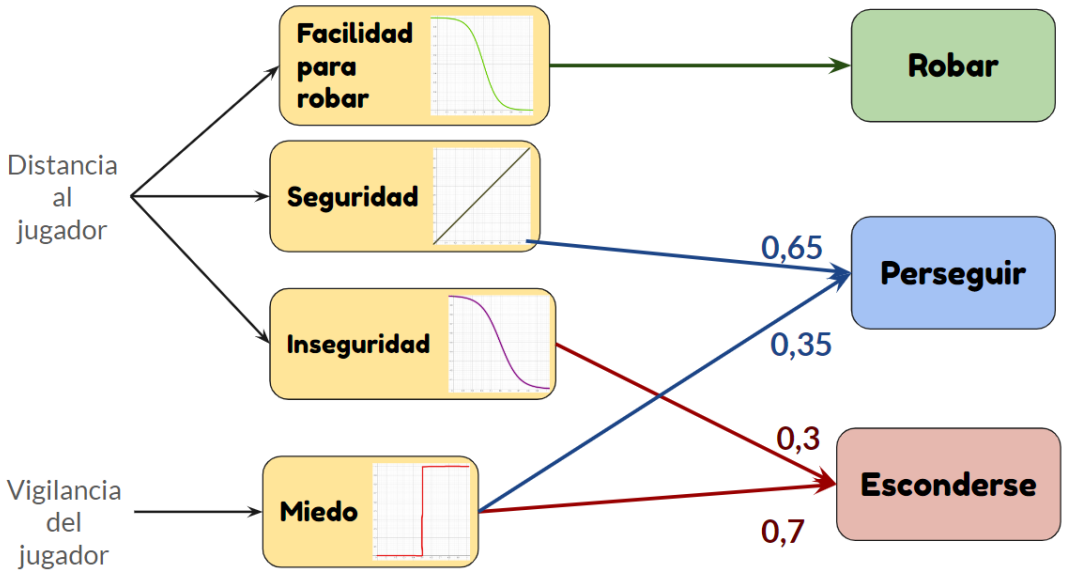


Figura 5.2.18: Sistema de Utilidad del personaje - tercera versión

Implementación del US - tercera versión

En primer lugar, se han hecho los cambios requeridos en las funciones de robar y perseguir, como se ve en el bloque de código 5.10.

```
1    stealingUtility = Math.Clamp(1 / (1 + Math.Exp(13*(distanceToPlayer.z /
     distanceToPlayer0.z) - 0.4)), 0, 1);
2    chasingUtility = Math.Clamp(0.65*security + 0.35*fear, 0, 1);
```

Listado de código 5.10: Cambios hechos en las funciones para la tercera versión del US.

Una vez hecho esto, se ha afrontado el problema de implementar inercia en el US. Para ello, se han hecho cambios en el método `ChooseAction()` (ver código 5.11), en el que se elige y ejecuta la acción con mayor utilidad. En este se ha añadido una comprobación, de forma que deban transcurrir como mínimo 0.2 segundos entre una acción y la siguiente. Para ello, se ha creado una variable que guarda el instante de tiempo en el que se cambió de acción por última vez. Además, se guarda en una variable la última acción que se ha realizado, de forma que estas dos variables nuevas solo se actualizarán si el personaje ha cambiado de acción tras el tiempo de inercia. En caso de que no haya pasado este tiempo, se ejecutará una estructura `switch` para que el NPC ejecute la acción que había elegido la última vez.

Por último, se han hecho algunas correcciones en el código: tras esconderse, el NPC sólo volverá a la posición inicial antes de cambiar de acción, y no antes; y para todas las mediciones de distancia con el jugador se usará la magnitud del vector, y no únicamente su coordenada z. Con todos estos cambios, la versión final del comportamiento se puede ver en el bloque de código 5.11.

```
    public class USenemy : NPCBase
{
    //utility of each action
    [SerializeField] double stealingUtility = 0;
    [SerializeField] double chasingUtility = 0;
    [SerializeField] double hidingingUtility = 0;

    //utility curves
    private double security;
    private double insecurity;
    private double fear;

    //inertia
    float lastChanged;
    NPCState previousState;

    public override void Start()
    {
        //set a start position and store its distance from the player
        distanceToPlayer0 = transform.position - player.transform.position;
        distanceToPlayer = distanceToPlayer0;
        playerMovement = player.GetComponent<PlayerMovement>();

        npcCurrent = NPCState.Chase;
        previousState = NPCState.Chase;
        lastChanged = Time.time;
    }

    public override void Update()
    {
        if(GameManager.Instance.State != GameState.GameState) return;

        //curves
        security = (distanceToPlayer.magnitude / distanceToPlayer0.
    magnitude);
        insecurity = 1/(1 + Math.Exp(-10*((distanceToPlayer.magnitude /
    distanceToPlayer0.magnitude) - 0.5)));
        fear = playerMovement.lookingBack ? 1 : 0;

        //utility functions
        //the results are clamped between 0 and 1
        stealingUtility = Math.Clamp(1 / (1 + Math.Exp(13*(distanceToPlayer
    .magnitude / distanceToPlayer0.magnitude) - 0.4)), 0, 1);   //this is
    also the curve for the factor ease to steal
        chasingUtility = Math.Clamp(0.65*security + 0.35*fear, 0, 1);
        hidingingUtility = Math.Clamp(0.3*insecurity + 0.7*fear, 0, 1);

        ChooseAction(stealingUtility, chasingUtility, hidingingUtility);
    }

    void ChooseAction(double stealUt, double chaseUt, double hideUt) {
        if(Time.time - lastChanged >= 0.2f) {
            if((chaseUt >= stealUt) && (chaseUt >= hideUt)) {
                Chase();
            } else if((stealUt > chaseUt) && (stealUt >= hideUt)) {
                Steal();
            } else {
                Hide();
```

```
55              }
56              if(previousState != npcCurrent) {
57                  if (previousState == NPCState.Hide)
58                  {
59                      //before finishing this action, enemy goes back to
    initial position
60                      distanceToPlayer = distanceToPlayer0;
61                      transform.position = player.transform.position +
    distanceToPlayer;
62                  }
63                  previousState = npcCurrent;
64                  lastChanged = Time.time;
65              }
66          } else {
67              switch(npcCurrent) {
68                  case NPCState.Chase:
69                      Chase();
70                      break;
71                  case NPCState.Steal:
72                      Steal();
73                      break;
74                  case NPCState.Hide:
75                      Hide();
76                      break;
77                  default: break;
78              }
79          }
80          return;
81      }
82  }
```

Listado de código 5.11: Implementación final del comportamiento usando un Sistema de Utilidad.

Los atributos y métodos que han sido necesarios para especificar el funcionamiento del enemigo, además, se han reflejado en el diagrama de clases creado en el segundo sprint. Quedando el diagrama como se muestra en la Figura 5.2.19.

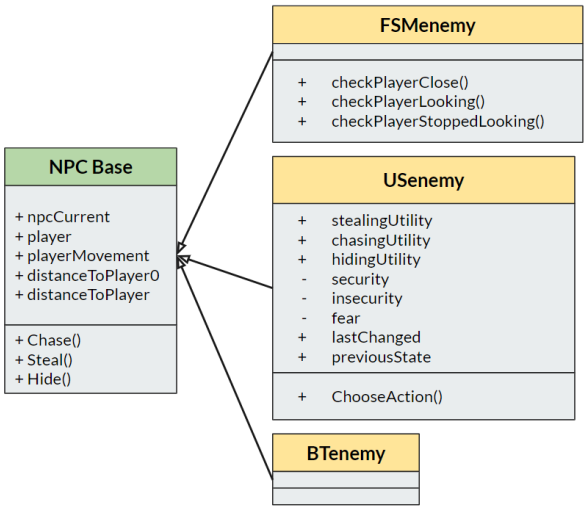


Figura 5.2.19: Diagrama de clases de los enemigos actualizado tras la implementación del US.

Revisión del sprint 3

En la revisión del sprint se tuvo una reunión con cliente y se le mostraron el funcionamiento final del personaje, además de las distintas pruebas que se hicieron para llegar hasta esta versión. La retroalimentación de la cliente fue buena, ya que no vio ningún problema en el resultado final presentado. Por lo tanto, se dio por finalizado el requerimiento RF-2.3, definido al inicio del proyecto.

Por otra parte, en lo relacionado con la planificación del proyecto, este sprint ha llevado algunos días más de lo esperado: quince días. Por este motivo, se va a modificar la duración del siguiente sprint, ya que se espera que lleve más de diez días. Este último sprint tendrá una duración estimada de trece días y dará comienzo una vez se muevan las tareas correspondientes en el tablero Kanban queda como se muestra en la Figura 5.2.20.

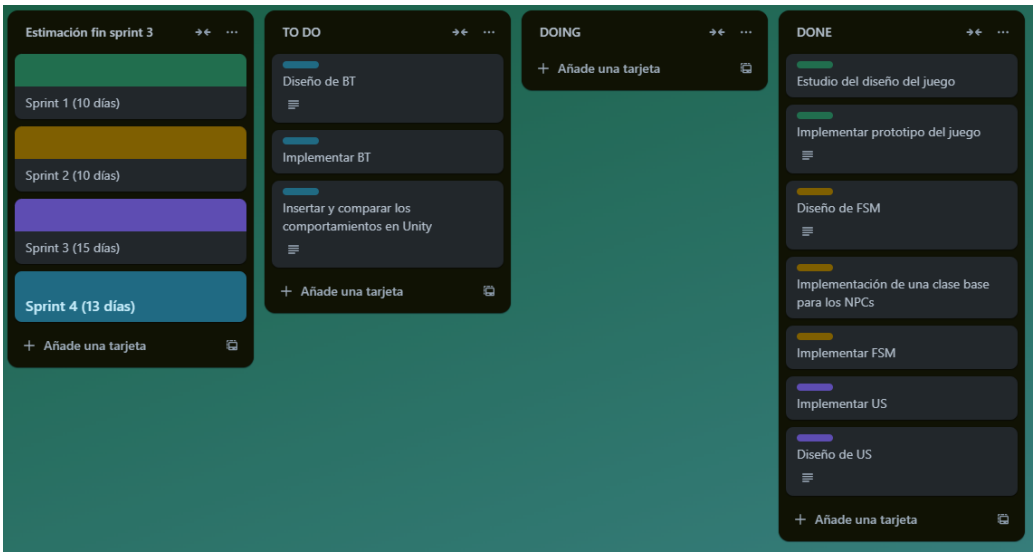


Figura 5.2.20: Tablero Kanban al final del Sprint 3

5.2.4. Sprint 4

Para terminar la implementación de los personajes, en este sprint se va a diseñar y programar el último de los comportamientos del juego, usando árboles de comportamiento, y se hará una comparación de las tres técnicas usadas en el proyecto. Así, se completarán los últimos objetivos específicos:

Objetivo 2

Diseñar e implementar tres algoritmos de comportamientos.

Objetivo 3

Evaluar y comparar las distintas técnicas desarrolladas para el caso concreto.

Una vez terminado el trabajo planificado para esta iteración, se dará por finalizado el proyecto, ya que se habrán completado los objetivos planteados para el mismo, los

requisitos restantes, RF-2.2, RNF-2.1, RF-3.1 y RNF-3.1 (ver Tabla 3.2) y los casos de uso planteados (ver 3.0.2).

Diseño del BT

La mayor dificultad a la hora de diseñar este árbol de comportamiento era decidir en qué orden ejecutar las tres acciones del personaje: esconderse, robar y perseguir. Igual que en los anteriores, se ha decidido que si el jugador se gira a vigilar en cualquier momento, el personaje se esconderá. Es por eso que esta comprobación se encuentra la primera. Se ha usado un nodo secuencia en ella, de forma que si el jugador no está vigilando, no se realizará la acción de esconderse.

En caso de que el jugador no esté vigilando, se comprobará si el jugador está lo suficientemente cerca para robarle, y en caso contrario, se pasará a la acción de perseguir.

El comportamiento descrito hasta ahora se ve reflejado en la Figura 5.2.21.

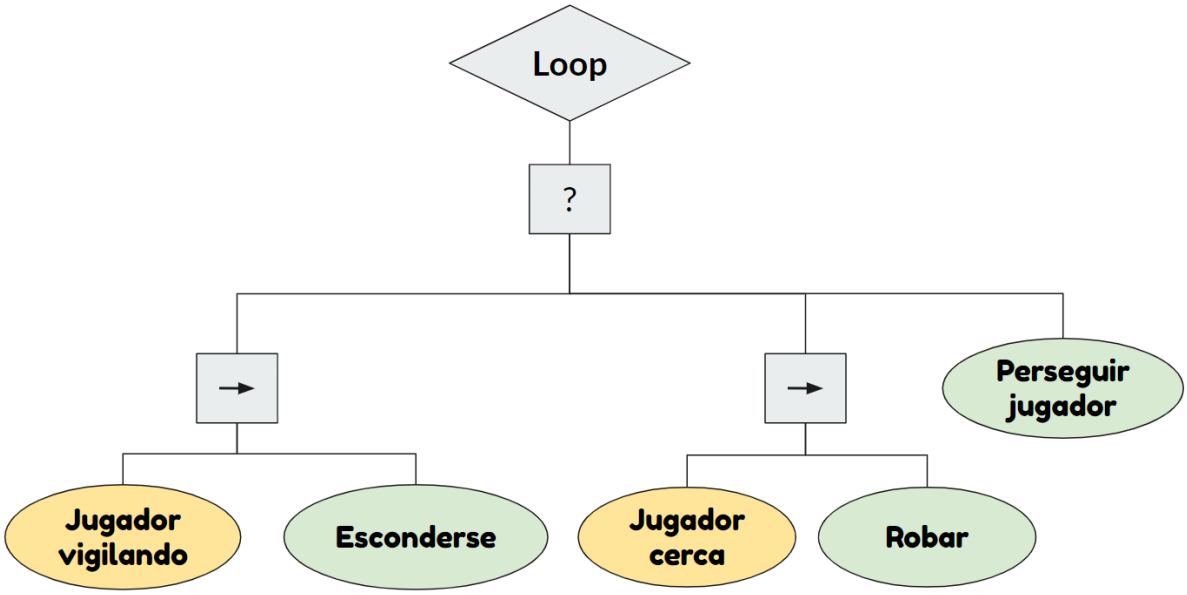


Figura 5.2.21: Comportamiento diseñado con BT

Implementación del BT

Para la implementación de este comportamiento, se ha hecho uso de la herramienta 'Panda BT' (http://www.pandabehaviour.com/), un asset de Unity que permite programar de una forma sencilla el árbol de comportamiento y visualizar a tiempo real el funcionamiento del mismo. Siguiendo la documentación de este framework, se ha creado el árbol de comportamiento haciendo uso de dos archivos:

- **BT-BTenemy.BT.txt**: Un archivo de texto (ver bloque de código 5.12) en el que se modela la estructura del árbol como tal, indicando los distintos niveles que se encuentren en el mismo, los nodos composición y las condiciones y acciones. Estas últimas deben tener los nombres concretos de los métodos que las modelan, que se desarrollan en otro archivo.

- **BTenemy.cs**: Un script (ver bloque de código 5.13) que hereda de la clase NPCBase y donde se implementan cada una de las acciones y condiciones que debe ejecutar el personaje. Además, al igual que en los anteriores NPCs, se guarda su estado actual en una variable, de forma que se pueda observar en todo momento.

El contenido de estos archivos es el siguiente:

```
tree("Root")
    tree("level1")

tree("level1")
  fallback
    sequence
      checkPlayerLooking
      HideBT
    sequence
      checkPlayerClose
      StealBT
    ChaseBT
```

Listado de código 5.12: Archivo de Panda BT en el que se define el árbol de comportamiento.

```
    public class BTenemy : NPCBase
{
    //Using PandaBT to create the BT logic
    //this script is used to define the actions that each node must do.
    PandaBehaviour pandaBT;

    public override void Start()
    {
        pandaBT = GetComponent<PandaBehaviour>();
        distanceToPlayer0 = transform.position - player.transform.position;
        distanceToPlayer = distanceToPlayer0;
        playerMovement = player.GetComponent<PlayerMovement>();
    }

    public override void Update()
    {
        pandaBT.Tick();
        pandaBT.Reset();
    }

    //HIDE
    [Task]
    bool checkPlayerLooking() { return playerMovement.lookingBack;  }

    [Task]
    void HideBT(){
        if(GameManager.Instance.State != GameState.GameState)  return;

        npcCurrent = NPCState.Hide;

        distanceToPlayer = distanceToPlayer + (new Vector3(0.05f, 0.3f, 0.2
    f));
        transform.position = player.transform.position + distanceToPlayer;
    }

```

```
35      //STEAL
36      [Task]
37      bool checkPlayerClose() {   return Mathf.Abs(distanceToPlayer.z) <= 4.5
        f;   }
38
39      [Task]
40      void StealBT(){
41          if(GameManager.Instance.State != GameState.GameState)  return;
42
43          npcCurrent = NPCState.Steal;
44
45          //finished stealing, comes back to initial distance
46          //changes to CHASE state
47          if (Mathf.Abs(distanceToPlayer.z) <= 0.2f)
48          {
49              distanceToPlayer = distanceToPlayer0;
50              transform.position = player.transform.position +
        distanceToPlayer;
51              npcCurrent = NPCState.Chase;
52          }
53
54          //steal
55          distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.1f));
56          transform.position = player.transform.position + distanceToPlayer;
57          return;
58      }
59
60      //CHASE
61      [Task]
62      void ChaseBT(){
63          if(GameManager.Instance.State != GameState.GameState)  return;
64
65          //to ensure the enemy gets back to initial place after hiding
66          if(npcCurrent == NPCState.Hide) {
67              distanceToPlayer = distanceToPlayer0;
68              transform.position = player.transform.position +
        distanceToPlayer;
69          }
70
71          npcCurrent = NPCState.Chase;
72
73          //enemy gets progrisevely closer to the player
74          distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.01f));
75          transform.position = player.transform.position + distanceToPlayer;
76          return;
77      }
78
79  }
```

Listado de código 5.13: Script en el que se modelan las acciones que realiza el NPC.

Como se puede ver, en el script se define cada condición y cada acción con la etiqueta `[Task]`. Esto permite a PandaBT saber que son esos los métodos que se referencian desde el archivo de texto. Por otra parte, se puede observar que los métodos utilizados para las acciones no son los heredados desde NPCBase, sino que se han implementado de nuevo. Esta nueva implementación se debe a que PandaBT obliga a que sean métodos implementados en el script, por lo que no es posible utilizar los ya existentes. Se ha

actualizado el diagrama de clase anterior (ver Figura 5.2.19), de forma que ahora el enemigo BT implemente los métodos correspondientes a cada acción. El nuevo diagrama tras este cambio se puede ver en la Figura 5.2.22.

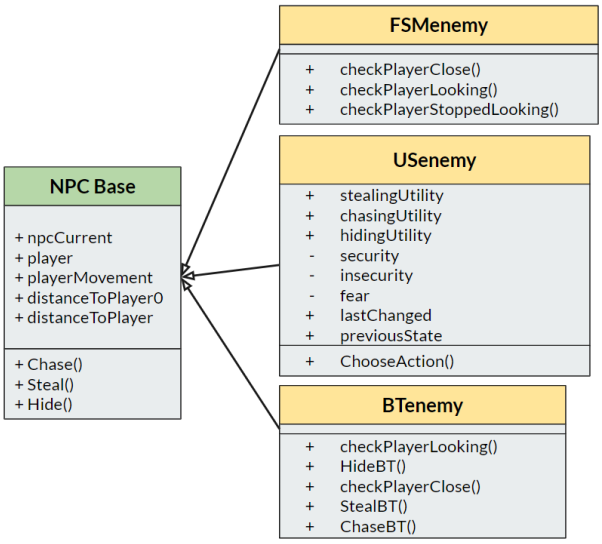


Figura 5.2.22: Diagrama de clases de los enemigos actualizado tras la implementación

Para terminar la implementación del personaje, se ha creado un objeto en la escena. En este caso una cápsula de color verde a la que se le se han añadido en esta un componente para cada uno de los archivos, como se puede observar en la Figura 5.2.23.

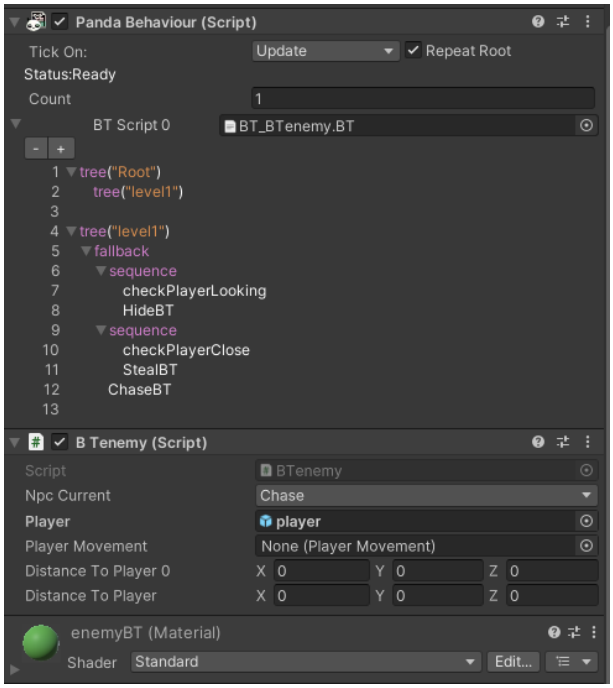


Figura 5.2.23: Componentes añadidos al GameObject del enemigo implementado con BT.

Por último, se ha añadido en el UIManager la referencia a este personaje, de forma que se vea su estado en pantalla durante el juego. Una vez hecho esto, todos los personajes se pueden ver en el juego y comparar su comportamiento (ver Figura 5.2.24).

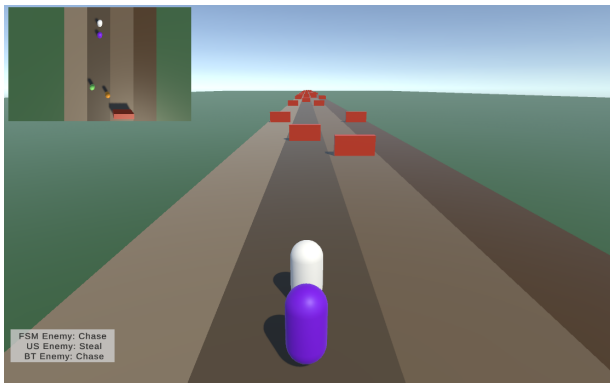


Figura 5.2.24: Prototipo de '¡Al ladrón!' con todos los NPCs implementados.

Comparación de comportamientos

Antes de dar por finalizado el trabajo, se van a comparar las distintas técnicas usadas y decidir cuál de ellas será más conveniente en este proyecto concreto. Para ello, además de evaluar el resultado final, se van a listar algunas ventajas, desventajas y dificultades que se han encontrado durante el desarrollo.

En cuanto al resultado final, se pueden observar algunas diferencias entre los tres personajes, la principal: su velocidad al perseguir al jugador es distinta. Los tres NPCs utilizan los mismos métodos para ejecutar las acciones, por lo que esta diferencia no debería depender de la velocidad que tienen programada para esta acción, sino de la rapidez de cada una de estas técnicas para evaluar qué acción realizar y llevarla a cabo. Cuanto más rápido decida el personaje qué acción hacer, antes la realizará y podrá decidir cuál será su siguiente acción. Para reafirmar esto, se han añadido a la escena varios enemigos de cada tipo. De esta forma, se ha confirmado que la velocidad a la que se mueven los personajes al perseguir al jugador depende la técnica utilizado en ellos.

Por lo tanto, se puede afirmar que la técnica que evalúa más rápidamente que acción realizar son los árboles de comportamiento, con una gran diferencia frente a las otras dos técnicas. Las FSM y los US están muy a la par, sin embargo, los US tardan más en decidirse por robar, por lo que esto los retrasa a la larga. Estas diferencias eran de esperar, debido a varios motivos:

- Las comprobaciones que realiza el BT son sencillas y lo único que debe hacer con estas es devolver un resultado 'true' o 'false'. Por lo tanto, su ejecución es rápida. Adicionalmente, los BT no deben llamar a ningún método externo para ejecutar las acciones, ya que están implementadas en el propio BT.

- En las FSM, cada función de comprobación, también debe hacerse cargo de la transición entre estados. Si se hace una comprobación y esta tiene resultado 'true', se debe ejecutar la transición de estado antes que ninguna acción. Además, el estado de perseguir cuenta con dos comprobaciones (no solo una), lo que requiere más tiempo.

- Los US necesitan hacer varios cálculos matemáticos para decidir qué acción hacer, por lo que era de esperar que su ejecución fuese más lenta que la de un BT.

No obstante, el funcionamiento de los tres personajes es correcto y esta pequeña diferencia se podría corregir ajustando la velocidad de cada personaje.

En lo relacionado al desarrollo, se pueden encontrar varios pros y contras de cada técnica. En las Tablas 5.1 y **??** se mencionan algunos de ellos.

	Pros	Contras
FSM	Conceptualmente, son fáciles de entender y de traspasar la descripción del comportamiento, al diagrama de estados. Ha sido fácil de diseñar e implementar. Existe mucha documentación y una gran cantidad de ejemplos y tutoriales para su implementación en videojuegos.	Tarda más tiempo que otras técnicas en hacer las comprobaciones para las transiciones. Modela comportamientos bastante rígidos. Con esta técnica se ha conseguido un personaje poco natural y con un comportamiento poco fluido, en comparación con las demás.
US	Se pueden conseguir comportamientos más realistas y complejos, en los que los personajes tengan mucha libertad. En este caso, el personaje puede decidir no esconderse si está muy lejos del jugador. Este es un comportamiento más realista que el de sus compañeros. A pesar de tener comprobaciones más complejas (con varias operaciones matemáticas), su velocidad de decisión es similar a la de una FSM.	Es más complicado conseguir comportamientos que sigan unas pautas concretas. Han hecho falta varias versiones para llegar al comportamiento deseado. Su funcionamiento es bastante complejo y requiere de bastantes conocimientos técnicos, por ejemplo para definir las curvas de utilidad o programar la inercia en la evaluación de las acciones. Requiere un mayor tiempo de trabajo, ya que es fácil caer en errores en el diseño, y tener que probar con varias versiones. De las tres, es la técnica de la que ha sido más difícil encontrar información y ejemplos en videojuegos.
BT	Existe mucha documentación y herramientas gratuitas para implementarlos en videojuegos. Gracias a herramientas como PandaBT, su implementación es bastante sencilla. Además, su diseño es mucho más sencillo que el de un US. Se consiguen muy buenos resultados en comportamientos pautados. De las tres técnicas ha sido la que hace las decisiones más rápido.	Su diseño puede ser complejo, ya que hay que pensar mucho en el orden en el que se realiza cada comprobación y cada tarea. Requiere conocer los distintos tipos de nodos que existen. Es difícil conseguir comportamientos más naturales o con mayor libertad.

Tabla 5.1: Ventajas y desventajas en el desarrollo de comportamientos con las distintas técnicas.

No obstante, estas ventajas y desventajas son subjetivas y se ven influenciadas por los conocimientos de la desarrolladora. Siempre será el equipo de desarrollo el que deba decidir qué técnica es más acorde a sus conocimientos y su proyecto.

Revisión del sprint 4

Para finalizar el proyecto, se va a hacer la última revisión de sprint, en la que se mostrará el resultado final a la cliente y se analizará la planificación general a lo largo de todo el proyecto.

Después de mostrarle al cliente el último comportamiento desarrollado, y las ventajas y desventajas de cada técnica, ha concluido que la técnica más adecuada para su juego '¡Al ladrón!' es un árbol de comportamiento. Un BT ofrece buenos resultados para el tipo de juego que se quiere desarrollar. Además, permite añadir complejidad al comportamiento desarrollado, cosa que sería muy complicada usando FSMs; y no requiere de tanto tiempo de desarrollo como un US. Por esto mismo, cuando empiece el desarrollo del juego, la implementación de los comportamientos de los NPCs enemigos se harán con BT.

Al concluir este sprint, se han completado los dos objetivos y requisitos restantes. En consecuencia, se da por finalizado el proyecto.

Para futuros proyectos, es interesante analizar de forma general la planificación del proyecto. En la Figura 5.2.25 se muestra la estimación inicial de cada sprint (en días), las estimaciones que se han realizado en las distintas revisiones de sprints y la duración real de cada uno (en las tarjetas marcadas con una parte negra).

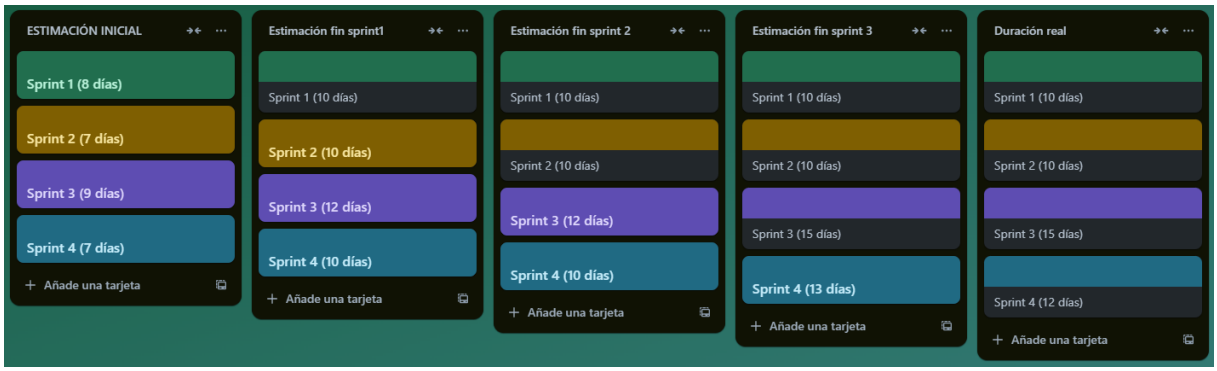


Figura 5.2.25: Comparación de las distintas estimaciones de tiempo durante el proyecto.

Como se puede ver, la estimación inicial (antes de comenzar con el desarrollo) fue muy optimista. Sin embargo, las correcciones que se han ido haciendo a lo largo del desarrollo han llevado a estimaciones bastante buenas.

Capítulo 6

Conclusiones

A lo largo de este documento se ha presentado una problemática: una cliente quiere decidir si es mejor implementar los enemigos de su videojuego usando FSMs, BTs o USs. Para atender a esta petición, se ha planteado desarrollar el comportamiento de estos personajes usando cada una de las técnicas propuestas. Además, se ha realizado un prototipo de su juego para poder probar en este los NPCs. De esta forma, se han podido comparar las tres técnicas, atendiendo tanto al resultado de cada una de ellas como a la dificultad y trabajo que suponen.

Es importante plantearse estas cuestiones e invertir un poco de tiempo en hacer una elección de este estilo, que ahorrará mucho tiempo de trabajo innecesario al equipo de desarrollo. Una mala elección hará que la implementación de los personajes lleven más tiempo, ya sea por errores en el diseño o la programación, por la necesidad de emplear tiempo en estudiar el funcionamiento de la técnica elegida, o por no conseguir los resultados esperados. Esta pérdida de tiempo se traduce en una pérdida de dinero para la empresa involucrada. Sin embargo, plantear esta cuestión al inicio del proyecto hará que los desarrolladores puedan trabajar de una forma más eficiente y conseguir buenos resultados.

En este proyecto concreto, la cliente ha podido ver las ventajas y desventajas que presentan cada una, y habiendo estado informada de todo el desarrollo, ha decidido que lo más conveniente para su proyecto es hacer uso de **árboles de comportamiento**. Estos permiten una implementación rápida, ya que su diseño y programación se pueden hacer de forma sencilla, siempre que se conozcan algunos conceptos básicos. Este motivo les hace una buena opción para desarrolladores con poca experiencia, como es su caso. Además, permiten conseguir muy buenos resultados, especialmente en juegos en los que se busquen comportamientos bastante controlados, como es el caso de los enemigos de '¡Al ladrón!'.

Para llegar que la clienta pudiese llegar a esta conclusión, se han ido cumpliendo todos los objetivos planteados al inicio del proyecto. En primer lugar, se desarrolló un prototipo del videojuego con las funcionalidades básicas de un juego endless runner y añadiendo algunas características específicas de '¡Al ladrón!'. Este prototipo cumple los requisitos funcionales acordados con la cliente (ver Tabla 3.2) y servirá como escenario para probar el funcionamiento de los distintos personajes. Una vez hecho esto, el trabajo se basó en implementar el comportamiento de los enemigos con cada una de las técnicas: FSM, US y BT, en ese orden. De cada una de estas se realizó su diseño y su implementación e inclusión en el prototipo. De esta forma, al terminar los cuatro sprints se habían completado todos los requisitos del proyecto y se dio por finalizado.

6.1. Futuros trabajos

Habiendo finalizado este proyecto, se plantean una serie de futuros trabajos que podrían llevarse a cabo, en caso de que la clienta estuviera de acuerdo.

Siguiendo la línea de trabajo establecida para este proyecto, sería interesante desarrollar comportamientos más complejos para los personajes. Estos comportamientos podrían ser extensiones o adaptaciones de los ya desarrollados, o comportamientos completamente nuevos. Gracias a ellos, se podrían comparar de nuevo las tres técnicas y decidir cuál sería más conveniente en ese caso. Es posible que las distinciones entre las tres técnicas sean notables en este caso y es interesante conocer sus diferencias y la dificultad a la hora de implementar cada una.

Además, crear comportamientos más complejos puede añadir mayor riqueza al juego. Por ejemplo, podría ser interesante que los personajes tengan varias formas de esconderse: ocultándose detrás de un obstáculo o huyendo si no tienen ninguno cerca. Otro cambio que añadiría dinamismo al juego sería hacer que los personajes avancen más rápido o más despacio mientras se acercan. Por ejemplo, si el jugador se acaba de girar a vigilarles, se sentirían más inseguros y avanzarían más despacio, pero si ya ha pasado un tiempo, empiezan a acercarse más deprisa. Estas adaptaciones servirán tanto como un reto personal para mejorar en la programación de comportamientos para personajes, como una herramienta para entender mejor el funcionamiento de cada uno.

Referencias

Academy, S. (2012). Utility theory. *U*.

AI, y Games. (2019a). *The AI of Half-Life: Finite State Machines | AI 101.* Descargado de https://youtu.be/JyF0oyarz4U

AI, y Games. (2019b). *Behaviour Trees: The Cornerstone of Modern Game AI | AI 101.* Descargado de https://youtu.be/6VBCXvfNlCM

AI, y Games. (2021). *How utility ai helps npcs decide what to do next | ai 101.* Descargado de https://youtu.be/p3Jbp2cZg3Q

Alavi, M., Aliaga, S., y Murga, M. (2016). Máquinas de estado finito. *Revista de Investigación Estudiantil Iluminate*, *8*, 41.

Alcalá, J. (2011). Inteligencia artificial en videojuegos. *Ciclo de conferencias Game Spirit*, *2*.

Ben-Ari, M., Mondada, F., Ben-Ari, M., y Mondada, F. (2018). Finite state machines. *Elements of Robotics*, 55–61.

Birch, C. (2010, December). Gameinternals - understanding pac-man ghost behavior. *https://gameinternals.com/understanding-pac-man-ghost-behavior*.

Bourse, Y. (2012). Artificial intelligence in the sims series. *URL: https://team. inria. fr/imagine/files/2014/10/sims-slides. pdf (44 pages)*.

Cai, Z., Li, M., Huang, W., y Yang, W. (2021). Bt expansion: a sound and complete algorithm for behavior planning of intelligent robots with behavior trees. En *Proceedings of the aaai conference on artificial intelligence* (Vol. 35, pp. 6058–6065).

Canós, J. H., Letelier, P., y Penadés, M. C. (2003). Metodologías ágiles en el desarrollo de software. *Universidad Politécnica de Valencia, Valencia*, 1–8.

Colledanchise, M., y Ögren, P. (2018). *Behavior trees in robotics and ai: An introduction.* CRC Press.

Fresno, L. (2024). *'¡Al ladrón!' Diseño, modelado y animación de una galería de personajes para un juego 3D.* (Trabajo de Fin de Grado. Grado en Diseño y Desarrollo de Videojuegos. Universidad Rey Juan Carlos.)

Garre, C. (s.f.-a). Toma de Decisiones II. Behaviour Trees. *Apuntes de la asignatura de Comportamientos de Personajes.*

Garre, C. (s.f.-b). Toma de Decisiones III. Sistemas de Utilidad. *Apuntes de la asignatura de Comportamientos de Personajes.*

Garre, D., Carlos y Casas. (s.f.). Toma de Decisiones I. Máquinas de Estados - Finite State Machines (FSM). *Apuntes de la asignatura de Comportamientos de Personajes.*

Graham, D. . (2014). An introduction to utility theory. En *Game ai pro 360: Guide to architecture* (pp. 67–80). CRC Press.

Isaac. (s.f.). *Máquinas de estado finito ¿Qué son? ¿Para qué sirven? — profesionalreview.com.* https://www.profesionalreview.com/2022/11/26/maquinas-de-estado-finito-que-son-para-que-sirven/. ([Accessed 01-09-2024])

Jensen, N. E. (1967). An introduction to bernoullian utility theory: I. utility functions. *The Swedish journal of economics*, 163–183.

Johansson, M., y Verhagen, H. (2011). "where is my mind"-the evolution of npcs in online worlds. En *Proceedings of the 3rd international conference on agents and artificial intelligence - volume 2: Icaart,* (p. 359-364). SciTePress. doi: 10.5220/0003306903590364

Maida, E. G., y Pacienzia, J. (2015). Metodologías de desarrollo de software. *Tesis de Licenciatura en Sistemas y Computación. Facultad de Química e Ingeniería "Fray Rogelio Bacon"*.

Millington, I. (2019). *Ai for games*. CRC Press.

Montero, B. M., Cevallos, H. V., y Cuesta, J. D. (2018). Metodologías ágiles frente a las tradicionales en el proceso de desarrollo de software. *Espirales revista multidisciplinaria de investigación*, *2*(17).

Narratech, F. P. L. (2019). Árbol de comportamiento – narratech. *https://narratech.com/es/inteligencia-artificial-para-videojuegos/decision/arbol-de-comportamiento/*.

Ojeda, J. C., y Fuentes, M. d. C. G. (2012). Taxonomía de los modelos y metodologías de desarrollo de software más utilizados. *Universidades*(52), 37–47.

Rivas, C. I., Corona, V. P., Gutiérrez, J. F., y Hernández, L. (2015). Metodologías actuales de desarrollo de software. *Revista de Tecnología e Innovación*, *2*(5), 980–986.

Sagredo-Olivenza, I., Puga, G. F., Gómez-Martín, M. A., y González-Calero, P. A. (2015). Implementación de nodos consulta en árboles de comportamiento. En *Cosecivi* (pp. 144–155).

Schwaber, K., y Sutherland, J. (2013). La guía de scrum. *Scrumguides. Org*, *1*, 21.

Sweetser, P., y Wiles, J. (2002). Current ai in games: A review. *Australian Journal of Intelligent Information Processing Systems*, *8*(1), 24–42.

Thompson, T. (2018). Primal instinct | companion ai in far cry primal. *https://www.gamedeveloper.com/design/primal-instinct-companion-ai-in-far-cry-primal/*.

TooLoo. (2021). *Utility ai in unity - part 1 - introduction.* Descargado de https://youtu.be/ejKrvhusU1I

Uludağlı, M. Ç., y Oğuz, K. (2023). Non-player character decision-making in computer games. *Artificial Intelligence Review*, *56*(12), 14159–14191.

Yacelga, A. R. L., y Cabrera, M. A. C. (2022). Uso de tableros kanban como apoyo para el desarrollo de las metodologías ágiles. *Universidad y Sociedad*, *14*(S2), 208–214.

Apéndice A

Detalles del proyecto en Unity

En este apéndice se va a describir brevemente la estructura del proyecto y se van a adjuntar todo el código desarrollado para el mismo. El proyecto de Unity en el que se ha implementado el prototipo y los distintos personajes se puede encontrar en el siguiente repositorio de GitHub https://github.com/lfresno/AL-LADRON-comportamientos, en la carpeta con el nombre 'Al ladrón'.

A.1. Estructura del proyecto

Para poder abrir el proyecto es necesario tener instalado el software Unity. Se debe crear un nuevo proyecto 3D e importar el paquete con el nombre 'AlLadron-UnityPackage' (Assets>Import Package>Custom>Seleccionar Archivo). Una vez hecho esto, se actualizarán los archivos del proyecto. En la carpeta 'Assets' se encuentran los directorios más relevantes para este proyecto:

- Carpeta '**Prefabs**': contiene los prefabs de los tres personajes, el jugador y los obstáculos. En Unity, un prefab es una especie de objeto prefabricado, que se puede añadir directamente a la escena y contará con unas características concretas. En este caso, los prefabs de los enemigos tienen todos los componentes necesarios para que funcionen al insertarlos en la escena.
- Carpeta '**Scenes**': únicamente tiene una escena llamada 'GameScene', en la que se ha desarrollado el prototipo.
- Carpeta '**Scripts**': donde se encuentran todos los archivos de código utilizados en el proyecto.

En primer lugar, para poder probar el prototipo es necesario abrir la escena 'GameScene'. Al hacer esto se cargará la escena con el juego y se podrá ejecutar y probarlo. Como se puede ver, la escena contiene el escenario, formado por el suelo, los cuatro carriles del camino y los obstáculos; las dos cámaras y los paneles para las distintas interfaces; y el jugador y los tres enemigos. Por otra parte, se pueden explorar los scripts del proyecto. En esta carpeta se pueden encontrar:

- Los scripts utilizados para le control del jugador: 'PlayerMovement' y 'Player'.

- Los archivos que controlan el movimiento de cada una de las cámaras: 'CameraController', de la cámara principal, y 'TopCameraController', de la cámara cenital.
- Los scripts 'GameManager' y 'UIManager', cuya función es controlar el flujo principal del juego, sus estados y el movimiento ellos.
- Una carpeta con el nombre 'NPCs', que contiene los scripts correspondientes a los tres enemigos:
  - 'NPCBase': el script que se usó de base para los comportamientos.
  - 'FSMenemy': el código que modela el comportamiento del enemigo desarrollado con una máquina de estados.
  - 'USenemy': el archivo que se usa en el enemigo implementado con un sistema de utilidad.
  - 'BT-BTenemy' y 'BTenemy': los archivos que contienen la estructura del árbol de comportamiento y la implementación de los métodos del mismo, respectivamente.

A.2. Scripts del proyecto

A continuación se incluyen los scripts de mayor interés del proyecto..

```
    public class PlayerMovement : MonoBehaviour
{
    private CharacterController controller;

    [SerializeField]
    private float playerSpeed = 2.0f;
    public float speedIncrement = 1.0f;
    private float lastIncrement;
    private int playerTrack = 0;    //there are 4 tracks (0-3) in which the player can choose to run
    public bool lookingBack = false;

    private void Start()
    {
        controller = gameObject.AddComponent<CharacterController>();
        lastIncrement = Time.time;
    }

    void Update()
    {
        if(GameManager.Instance.State != GameState.GameState) return;

        //if player reaches the end of the created path, they will be redirected to the beggining
        //this is only used in the prototype to test NPCs, but not in the final game
        if(gameObject.transform.position.z >= 2022.2) {
            controller.Move(new Vector3(0, 0, -1000.0f));
        }


```

```
29      if(Input.GetKeyDown(KeyCode.Space)) {
30          //if space is pressed, the player will look back to ckeck on
    the thieves.
31          //in this moment, the player will stop moving and the camera
    will move so that the player can check on the animals (they will try to
    hide)
32
33          //this will throw a coroutine
34          speedIncrement -= 0.03f;
35          StartCoroutine(LookBack());
36      }
37      else if (!lookingBack)  //when the player is not looking back, they
     can move along the tracks
38      {
39          //speed is incremented every 0.5 seconds
40          //the increment in speed is greater at the beginning of the
    game, so that it gets difficult fast, but not too much
41          if (Time.time - lastIncrement > 0.02f)
42          {
43              if (Time.time < 2.0f)
44              {
45                  speedIncrement += 0.1f;
46              }
47              else
48              {
49                  speedIncrement += 0.07f;
50              }
51
52              lastIncrement = Time.time;
53          }
54
55          //player will have a constant movement forward and will only be
     able to move left or right
56          //if the player uses left or right keys, they will move to the
    corresponding track, if possible
57          if (Input.GetKeyDown(KeyCode.RightArrow) || Input.GetKeyDown(
    KeyCode.D))
58          {
59              if (playerTrack < 3)
60              {
61                  controller.Move(new Vector3(5.0f, 0, 0));
62                  playerTrack++;
63              }
64
65          }
66          else if (Input.GetKeyDown(KeyCode.LeftArrow) || Input.
    GetKeyDown(KeyCode.A))
67          {
68              if (playerTrack > 0)
69              {
70                  controller.Move(new Vector3(-5.0f, 0, 0));
71                  playerTrack--;
72              }
73
74          }
75
76      }
77
```

```
        Vector3 move = new Vector3(0, 0, playerSpeed + speedIncrement);

        controller.Move(move * Time.deltaTime);

        if (move != Vector3.zero)
        {
            gameObject.transform.forward = move;
        }
    }

    private IEnumerator LookBack() {
        lookingBack = true;
        float t = Time.realtimeSinceStartup;
        Time.timeScale = 0f;

        while(Time.realtimeSinceStartup - t < 3.0f){
            yield return 0;
        }

        Time.timeScale = 1.0f;
        lookingBack = false;
        yield break;
    }

}
```

```
    public class CameraController : MonoBehaviour
{
    public GameObject player;
    public PlayerMovement playerMovement;
    private Vector3 distanceToPlayer;
    //private bool lookingBack = false;

    void Start()
    {
        distanceToPlayer = transform.position - player.transform.position;
        playerMovement = player.GetComponent<PlayerMovement>();
    }

    void Update()
    {
        if(GameManager.Instance.State != GameState.GameState) return;

        if (Input.GetKeyDown(KeyCode.Space) || playerMovement.lookingBack)
        {
            //if player presses Space, the camera will move a little
   further so that they can see the animals following
            transform.position = player.transform.position +
   distanceToPlayer + new Vector3(0, 2, -8);
            //StartCoroutine(LookBack());
        }
        else if (!playerMovement.lookingBack)
        {
            //camera will follow player
            transform.position = player.transform.position +
   distanceToPlayer;
        }
    }
```

```
30
31 }
```

```
1     public class GameManager : MonoBehaviour
2 {
3
4     public static GameManager Instance;
5     public GameState State;
6     private Player player;
7
8     void Awake()  {
9         Instance = this;
10    }
11
12    // Start is called before the first frame update
13    void Start()
14    {
15        UpdateGameState(GameState.MenuState);
16        player = GameObject.FindGameObjectWithTag("Player").GetComponent<
      Player>();
17    }
18
19    // Update is called once per frame
20    void Update()
21    {
22        if(Input.GetKeyDown(KeyCode.Escape)){
23            UpdateGameState(GameState.PauseState);
24        }
25    }
26
27    public void UpdateGameState(GameState newState) {
28        State = newState;
29
30        switch(newState) {
31            case GameState.MenuState:
32                Time.timeScale = 0f;
33                UIManager.Instance.MenuActive(true);
34                UIManager.Instance.EndMenuActive(false);
35                break;
36
37            case GameState.GameState:
38                Time.timeScale = 1.0f;
39                player.ResetPlayer();
40                UIManager.Instance.MenuActive(false);
41                UIManager.Instance.EndMenuActive(false);
42                break;
43
44            case GameState.PauseState:
45                Time.timeScale = 0f;
46                UIManager.Instance.MenuActive(false);
47                UIManager.Instance.EndMenuActive(true);
48                break;
49
50            default:
51                break;
52        }
53    }
54 }
```

```

public enum GameState {
    MenuState,
    GameState,
    PauseState
}
```

```
    public abstract class NPCBase: MonoBehaviour
{
    public NPCState npcCurrent;

    //Player
    public GameObject player;
    public PlayerMovement playerMovement;
    public Vector3 distanceToPlayer0;
    public Vector3 distanceToPlayer;


    public abstract void Start();
    public abstract void Update();


    //ACTIONS (all NPCs will do the same actions)
    public void Chase(){

        npcCurrent = NPCState.Chase;

        //enemy gets progrisevely closer to the player
        distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.01f));
        transform.position = player.transform.position + distanceToPlayer;
        return;
    }

    public void Steal(){

        npcCurrent = NPCState.Steal;

        //finished stealing, comes back to initial distance
        //changes to CHASE state
        if (Mathf.Abs(distanceToPlayer.z) <= 0.2f)
        {
            distanceToPlayer = distanceToPlayer0;
            transform.position = player.transform.position +
   distanceToPlayer;
            npcCurrent = NPCState.Chase;
        }

        //steal
        distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.1f));
        transform.position = player.transform.position + distanceToPlayer;
        return;
    }

    public void Hide(){

        npcCurrent = NPCState.Hide;

        //this enemy will be represented by a bird, so their way to hide is
```

```
    fly away
        distanceToPlayer = distanceToPlayer + (new Vector3(0.05f, 0.3f, 0.2
    f));
        transform.position = player.transform.position + distanceToPlayer;

    }

}

public enum NPCState
{
    Chase,
    Steal,
    Hide
}
```

```
    public class FSMenemy : NPCBase
{

    public override void Start()
    {
        //set a start position and store its distance from the player
        distanceToPlayer0 = transform.position - player.transform.position;
        distanceToPlayer = distanceToPlayer0;
        playerMovement = player.GetComponent<PlayerMovement>();

        npcCurrent = NPCState.Chase;
    }

    public override void Update()
    {
        if(GameManager.Instance.State != GameState.GameState) return;

        switch(npcCurrent)
        {
            case NPCState.Chase:
                //check if we have to change state
                if(checkPlayerClose() || checkPlayerLooking()) break;

                Chase();
                break;

            case NPCState.Steal:
                //check if we have to hide
                if(checkPlayerLooking()) break;

                Steal();
                break;

            case NPCState.Hide:
                //check if we can chase
                if(checkPlayerStoppedLooking()) break;

                Hide();
                break;

            default:
                break;
```

```
        }

    }

    //PERCEPTIONS (will cause transitions between states )

    //checks if  the player is close enough to steal from them.
    //changes to steal state if they are
    bool checkPlayerClose()
    {
        if (Mathf.Abs(distanceToPlayer.z) <= 4.5f)
        {
            npcCurrent = NPCState.Steal;
            return true;
        }
        return false;
    }

    //check if the player is looking towards the enemies
    //changes to hide state
    bool checkPlayerLooking()
    {
        if (playerMovement.lookingBack)
        {
            distanceToPlayer = distanceToPlayer0;
            npcCurrent = NPCState.Hide;
            return true;
        }
        return false;
    }

    //check if player stopped looking towards the enemies
    //changes the state to chase
    bool checkPlayerStoppedLooking()
    {
        if (!playerMovement.lookingBack)
        {
            distanceToPlayer = distanceToPlayer0;
            transform.position = player.transform.position +
   distanceToPlayer;
            npcCurrent = NPCState.Chase;
            return true;
        }
        return false;
    }

}
```

```
    public class USenemy : NPCBase
{
    //utility of each action
    [SerializeField] double stealingUtility = 0;
    [SerializeField] double chasingUtility = 0;
    [SerializeField] double hidingingUtility = 0;

    //utility curves
    private double security;
    private double insecurity;
```

```
11      private double fear;
12
13      //inertia
14      float lastChanged;
15      NPCState previousState;
16
17      public override void Start()
18      {
19          //set a start position and store its distance from the player
20          distanceToPlayer0 = transform.position - player.transform.position;
21          distanceToPlayer = distanceToPlayer0;
22          playerMovement = player.GetComponent<PlayerMovement>();
23
24          npcCurrent = NPCState.Chase;
25          previousState = NPCState.Chase;
26          lastChanged = Time.time;
27      }
28
29      public override void Update()
30      {
31          if(GameManager.Instance.State != GameState.GameState) return;
32
33          //curves
34          security = (distanceToPlayer.magnitude / distanceToPlayer0.
    magnitude);
35          insecurity = 1/(1 + Math.Exp(-10*((distanceToPlayer.magnitude /
    distanceToPlayer0.magnitude) - 0.5)));
36          fear = playerMovement.lookingBack ? 1 : 0;
37
38
39          //utility functions
40          //the results are clamped between 0 and 1
41          stealingUtility = Math.Clamp(1 / (1 + Math.Exp(13*(distanceToPlayer
    .magnitude / distanceToPlayer0.magnitude) - 0.4)), 0, 1);   //this is
    also the curve for the factor ease to steal
42          chasingUtility = Math.Clamp(0.65*security + 0.35*fear, 0, 1);
43          hidingingUtility = Math.Clamp(0.3*insecurity + 0.7*fear, 0, 1);
44
45          ChooseAction(stealingUtility, chasingUtility, hidingingUtility);
46      }
47
48      void ChooseAction(double stealUt, double chaseUt, double hideUt) {
49
50          if(Time.time - lastChanged >= 0.2f) {
51
52              if((chaseUt >= stealUt) && (chaseUt >= hideUt)) {
53                  Chase();
54              } else if((stealUt > chaseUt) && (stealUt >= hideUt)) {
55                  Steal();
56              } else {
57                  Hide();
58              }
59
60              if(previousState != npcCurrent) {
61                  if (previousState == NPCState.Hide)
62                  {
63                      //before finishing this action, enemy goes back to
    initial position
```

```
                    distanceToPlayer = distanceToPlayer0;
                    transform.position = player.transform.position +
    distanceToPlayer;
                }
                previousState = npcCurrent;
                lastChanged = Time.time;
            }
        } else {

            switch(npcCurrent) {
                case NPCState.Chase:
                    Chase();
                    break;

                case NPCState.Steal:
                    Steal();
                    break;

                case NPCState.Hide:
                    Hide();
                    break;

                default: break;
            }
        }

        return;
    }
}
```

```
    tree("Root")
  tree("level1")

    tree("level1")
      fallback
        sequence
          checkPlayerLooking
          HideBT
        sequence
          checkPlayerClose
          StealBT
        ChaseBT
```

```
    public class BTenemy : NPCBase
{
    //Using PandaBT to create the BT logic
    //this script is used to define the actions that each node must do.
    PandaBehaviour pandaBT;


    public override void Start()
    {
        pandaBT = GetComponent<PandaBehaviour>();

        distanceToPlayer0 = transform.position - player.transform.position;
        distanceToPlayer = distanceToPlayer0;
        playerMovement = player.GetComponent<PlayerMovement>();
    }

```

```
17      public override void Update()
18      {
19          pandaBT.Tick();
20          pandaBT.Reset();
21      }
22
23
24      //HIDE
25      [Task]
26      bool checkPlayerLooking() {
27          return playerMovement.lookingBack;
28      }
29
30      [Task]
31      void HideBT(){
32          if(GameManager.Instance.State != GameState.GameState)  return;
33
34          npcCurrent = NPCState.Hide;
35
36          distanceToPlayer = distanceToPlayer + (new Vector3(0.05f, 0.3f, 0.2
    f));
37          transform.position = player.transform.position + distanceToPlayer;
38      }
39
40      //STEAL
41      [Task]
42      bool checkPlayerClose() {
43          return Mathf.Abs(distanceToPlayer.z) <= 4.5f;
44      }
45
46
47      [Task]
48      void StealBT(){
49          if(GameManager.Instance.State != GameState.GameState)  return;
50
51          npcCurrent = NPCState.Steal;
52
53          //finished stealing, comes back to initial distance
54          //changes to CHASE state
55          if (Mathf.Abs(distanceToPlayer.z) <= 0.2f)
56          {
57              distanceToPlayer = distanceToPlayer0;
58              transform.position = player.transform.position +
    distanceToPlayer;
59              npcCurrent = NPCState.Chase;
60          }
61
62          //steal
63          distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.1f));
64          transform.position = player.transform.position + distanceToPlayer;
65          return;
66      }
67
68      //CHASE
69      [Task]
70      void ChaseBT(){
71          if(GameManager.Instance.State != GameState.GameState)  return;
72
```

```
73          //to ensure the enemy gets back to initial place after hiding
74          if(npcCurrent == NPCState.Hide) {
75              distanceToPlayer = distanceToPlayer0;
76              transform.position = player.transform.position +
    distanceToPlayer;
77          }
78
79          npcCurrent = NPCState.Chase;
80
81          //enemy gets progrisevely closer to the player
82          distanceToPlayer = distanceToPlayer + (new Vector3(0, 0, 0.01f));
83          transform.position = player.transform.position + distanceToPlayer;
84          return;
85      }
86
87  }
```