UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Desarrollo de la inteligencia artificial para un videojuego de estrategia en tiempo real

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

AUTOR/A: Mirete Blanco, Alejandro

Tutor/a: Abad Cerdá, Francisco José

CURSO ACADÉMICO: 2022/2023

Resumen

Este trabajo es una ampliación del proyecto de TFG “Desarrollo de un videojuego multijugador de estrategia en tiempo real con Unity y PUN”. En el trabajo se analizan distintas técnicas de IA comúnmente utilizadas en videojuegos y se selecciona la más adecuada para desarrollar un sistema de IA para un videojuego de estrategia en tiempo real. Tras esto, desde la perspectiva de la ingeniería de software, resuelve el reto de desarrollar e integrar una inteligencia artificial capaz de interactuar con los distintos sistemas del videojuego además del reto de adaptar el software ya existente para el juego multijugador a modo de un jugador.

Palabras clave: IA, Unity, RTS, un jugador.

Abstract

This work is an extension of the TFG project “Desarrollo de un videojuego multijugador de estrategia en tiempo real con Unity y PUN”. In this work different AI techniques commonly used in video games are analyzed and the most suitable one is chosen and used to develop an AI system for a real time strategy video game. This sets out several challenges, on the one hand, the project solves the challenge of developing and integrating an artificial intelligence system capable of interacting with the game systems, on the other hand, the challenge of adapting the existing software for a multiplayer game into a single player game.

Keywords : AI, Unity, RTS, single player.

Agradecimientos

Me gustaría dar las gracias a mi familia y amigos por su apoyo constante, a mi tutor Francisco José Abad Cerdá por su toda su ayuda y paciencia tanto en la elaboración de este trabajo como en el TFG y finalmente a ValgrAI – Valencian Graduate School and Research Network for Artificial Intelligence y a la Generalitat Valenciana por el apoyo económico recibido.

Tabla de contenidos

1. Introducción

1.1 Motivación

Los motivos para querer desarrollar este proyecto no han cambiado respecto a los del trabajo original.

En primer lugar, desde un punto de vista personal, los videojuegos me apasionan desde pequeño, especialmente los de estrategia. Además, decidí estudiar informática motivado por esta misma afición y dedicarme al desarrollo de videojuegos tras terminar los estudios es una opción muy atrayente para mí.

Por otra parte, como ya se comentó en el trabajo anterior, profesionalmente el sector de los videojuegos es uno que no para de crecer y, como en el resto del mundo de la informática, resistente frente a circunstancias económicas adversas. Además, las técnicas aplicadas en el desarrollo de este trabajo no son exclusivas del ámbito de los videojuegos, tienen aplicaciones en otras áreas en las que sea necesario modelar conjuntos de estados o sistemas de toma de decisiones.

1.2 Objetivos

El propósito de este trabajo es el de desarrollar la versión de un solo jugador de un videojuego de estrategia en tiempo real desarrollado anteriormente que únicamente dispone del modo jugador contra jugador a través de la red. Para implementar el modo de juego de un jugador contra la máquina, se debe desarrollar un sistema de inteligencia artificial y, para ello, se plantean los siguientes objetivos:

- Investigar distintos sistemas de inteligencia artificial utilizados en videojuegos, sus ventajas y desventajas y elegir el más apropiado para desarrollar el trabajo.
- Desarrollar un sistema de inteligencia artificial para un videojuego de estrategia en tiempo real.
    - Rediseñar la arquitectura existente **con los menores cambios posibles** para adaptarla para el sistema de IA.
    - Desarrollar dicho sistema creando código mantenible y fácilmente extensible.

1.3 Metodología

En el trabajo anterior, en el que se creó *Castle Strike* como videojuego de estrategia en tiempo real multijugador, se empleó una metodología de trabajo ágil basada en Scrum. Se subdividió el desarrollo en sprints, completando gradualmente el backlog generado para el proyecto. Esta decisión fue motivada porque el proyecto se estaba comenzando desde cero y la necesidad de organizar correctamente un trabajo tan grande. En este trabajo, sin embargo, no se ha considerado necesaria una organización tan estricta y simplemente se han identificado las tareas existentes, ordenado y realizado secuencialmente.

1.4 Estructura del documento

La estructura del trabajo es la siguiente. En primer lugar, en el capítulo 2, se discutirá el estado del arte de la inteligencia artificial en videojuegos y se analizarán y discutirán distintas técnicas comúnmente utilizadas.

En el capítulo 3 se analizará el diseño del proyecto. Se explicará las técnicas de IA utilizadas y porqué han sido escogidas y se mostrará cómo se han incorporado al sistema ya existente y las adaptaciones necesarias para ello.

El capítulo 4 describirá el proceso de desarrollo del proyecto. Se mostrará cómo se han desarrollado los nuevos sistemas para llegar a la solución final, los problemas y dificultades encontrados y finalmente se analizará la implementación de las partes más relevantes.

El capítulo 5 mostrará los resultados de las pruebas con usuarios, comprobando si la aplicación cumple con lo que esperan los potenciales jugadores.

El capítulo 6 resumirá las conclusiones alcanzadas y las lecciones aprendidas durante el proyecto.

1.5 Convenciones

- Los términos del glosario aparecerán como nota a pie de página la primera vez que aparezcan en caso de no estar explicados en el texto.
- Las referencias a las figuras que formen parte del texto estarán marcadas con letra cursiva y la primera letra mayúscula (*Figura 1*).
- Los términos relativos a Unity y a PUN se escribirán con letra normal.
- Los nombres de los videojuegos que se mencionen en la memoria se escribirán con letra cursiva, así como términos propios del área de desarrollo de videojuegos.

2. Estado del Arte

Desarrollar IA para videojuegos consiste, en esencia, en crear la ilusión de inteligencia, en simular comportamientos realistas que engañen al jugador y le hagan creer que está compitiendo contra un adversario humano.

2.1 Máquinas de estado finitas

Las máquinas de estado finitas son un patrón de diseño utilizado para encapsular el funcionamiento de estados discretos en un sistema. Son comúnmente utilizadas en videojuegos para controlar IA, animación y gestionar el estado del juego entre otras tareas[1].

Una máquina de estados finita se utiliza para modelar un sistema. Consiste en un número finito de estados en los que puede estar el sistema y una serie de transiciones entre ellos. Dada una entrada (como, por ejemplo, un evento) se puede provocar una transición de un estado a otro, provocando una acción. Cabe destacar que una máquina de estados finita solo puede estar en un estado[2].

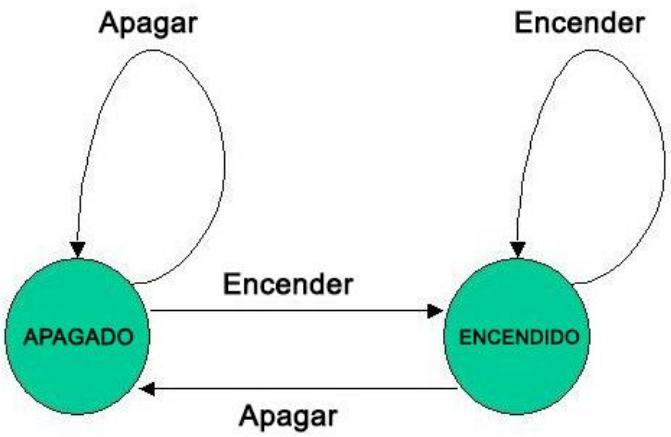


Figura 1 Ejemplo de una máquina de estados que representa el funcionamiento de un interruptor

Las ventajas de las máquinas de estados, y el motivo por el que son tan utilizadas en videojuegos son las siguientes [2]:

- Las máquinas de estados son relativamente simples de desarrollar, el funcionamiento básico es sencillo y la dificultad de implementación radica en lo complejo o simple que sea el comportamiento del agente que la implemente.
- Son fáciles de depurar, ya que la lógica de cada estado funciona de forma independientemente del resto de la máquina de estados, por lo que si se detecta un comportamiento erróneo o equivocado es fácil de aislar el estado en el que se encuentra el fallo.
- Son intuitivas, ya que es fácil dividir el funcionamiento de un agente en estados como, por ejemplo, atacar, patrullar y descansar, y crear reglas para definir y manipular estos estados. Además, también facilitan la comunicación entre programadores y diseñadores ya que no es necesario entender cómo funcionan a nivel técnico para diseñar los distintos estados de la máquina.

Existen numerosos ejemplos de videojuegos que utilizan máquinas de estados finitas para su IA. Un ejemplo famoso, en el videojuego *Pac-man* se usan para definir el comportamiento de los fantasmas, donde cada uno cuenta con dos estados: evadir y perseguir. También se utilizan en los videojuegos RTS para definir el comportamiento de las unidades.

2.1.1 Máquinas de estado jerárquicas

A pesar de las capacidades de las máquinas de estados para definir comportamientos tienen un problema que limita su complejidad: conforme crece el número de acciones en la máquina de estados también crece la dificultad de establecer las reglas para las transiciones. Para mitigar estos problemas se utilizan máquinas de estado jerárquicas. En estas máquinas los estados se dividen en superestados y subestados contenidos en los primeros de forma que solo se pueden ejecutar los subestados del superestado activo. Este diseño permite crear máquinas de estado más grandes y complejas.[3]

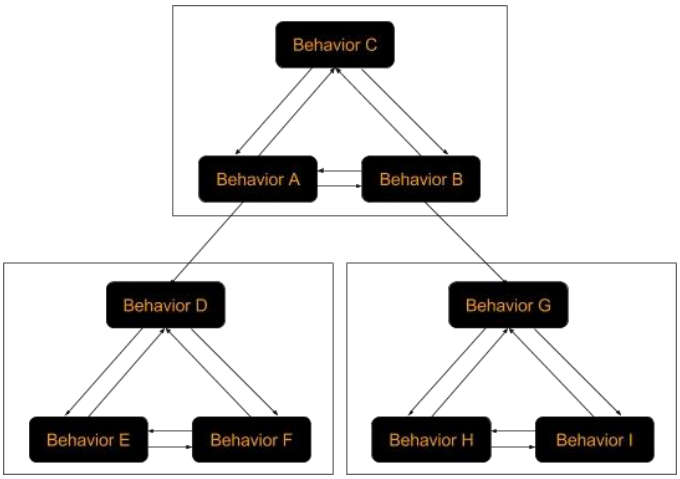


Figura 2 Estructura básica de una máquina de estados jerárquica[3]

La *Figura 2* representa el diseño de una máquina de estados jerárquica con tres superestados y tres subestados en cada uno. Se puede ver que hay transiciones definidas entre los superestados de forma que solo hay uno activo y solo se ejecuten sus subestados. Este tipo de implementación se utiliza cuando la máquina de estados es muy grande o un agente tiene más de una forma distinta de realizar una acción, por ejemplo, en un videojuego FPS[1] la función de atacar se puede realizar con un arma a distancia, cuerpo a cuerpo o lanzando una granada de forma que habría un superestado, ataque y tres subestados: disparar, ataque cuerpo a cuerpo y lanzar granada.

[1] First Person Shooter, juego de disparos en primera persona en castellano.

2.2 Goal Oriented Action Planning

Un agente GOAP es similar a una máquina de estados finita, ya que tiene los mismos componentes, un agente que ejecuta los estados, estados que contienen las acciones que el agente puede realizar y transiciones entre ellos, pero a diferencia de la máquina de estados, las transiciones de estado del agente GOAP no son fijas. Es capaz de, a partir de información del estado del juego y un objetivo que se le dé al agente, planear una secuencia de acciones y ejecutarlas. El planificador crea tantas secuencias de acciones como posibles caminos hayan para satisfacer el objetivo del agente. Tras esto, utilizando el algoritmo A* y los costes asociados a las distintas acciones se determina el mejor plan, que se ejecuta a continuación por una máquina de estados finita. Después de ejecutar una acción se comprueba si la siguiente continúa siendo válida en función del estado del juego, en caso contrario se descartan las acciones restantes en el plan y se crea uno nuevo.[4]

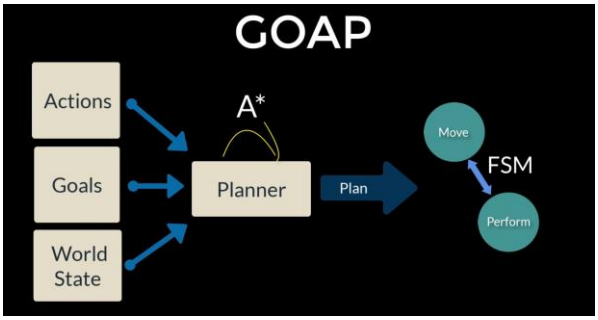


Figura 3 Diagrama ilustrativo del funcionamiento básico de un agente GOAP[4]

El primer videojuego en utilizar esta técnica fue *F.E.A.R*, un videojuego FPS desarrollado en 2005. También se ha utilizado con éxito en el RTS *Empire: Total War* de 2009.[5]

2.3 Utility AI

El funcionamiento de estos agentes está basado en el concepto del utilitarismo, corriente filosófica surgida en el siglo XVIII. La teoría de la utilidad es un marco matemático que permite modelar sistemas de toma de decisiones bajo incertidumbre, y es ampliamente utilizada en videojuegos, robótica y sistemas recomendadores entre otros[7].

Análogamente a un agente GOAP, dónde las acciones tienen asociadas un coste, en la utility AI las acciones tienen asociadas un valor o utilidad. Este valor se calcula a partir de diversos factores, tanto internos como externos al agente, que varían para cada juego e incluso entre distintos agentes dentro del mismo juego. Calcular los valores de utilidad es una tarea subjetiva y es donde se encuentra el mayor desafío a la hora de emplear este tipo de agente. Es un proceso en el que hay que convertir los distintos factores del juego en utilidad. Es común utilizar funciones lineales, cuadráticas y logarítmicas, pero encontrar la función adecuada para calcular la utilidad de cada acción es, como ya se ha dicho, subjetivo y depende de cada juego, los factores que se utilicen para calcular la utilidad y del programador.[6]

Una vez calculada la utilidad existen varias estrategias para elegir la acción a ejecutar. La más simple y directa consiste en seleccionar la acción con el valor de utilidad más alto, pero en algunos juegos esto no tiene por qué ser la mejor opción. Otra solución es asignar los valores de utilidad como pesos, calcular la probabilidad de que una acción sea escogida y escoger

aleatoriamente dependiendo de esos pesos. También, se puede escoger un subconjunto de las acciones con la utilidad más alta y elegir entre estas.

Sin embargo, una solución tan directa puede no ser adecuada para algunos juegos, ya que pueden existir situaciones en las que no se quiere evaluar una acción porque no es posible o no sería adecuado que fuera escogida. Un ejemplo de una situación así es en el videojuego *Los Sims*[2] donde, por ejemplo, si un personaje se está muriendo de hambre, no tiene sentido que trate de satisfacer su necesidad de diversión o de ocio.

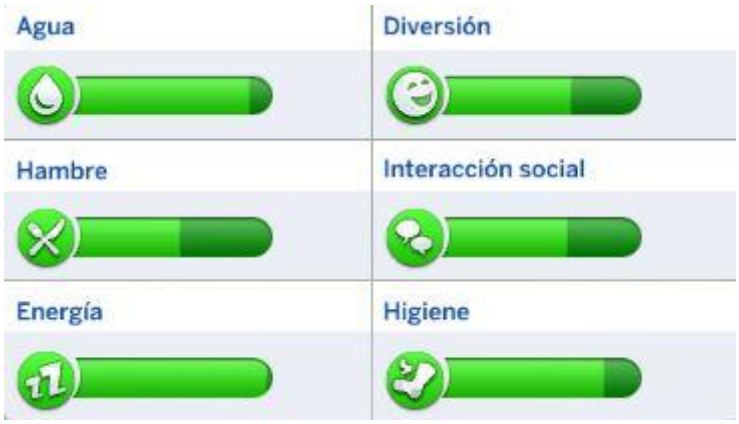


Figura 4 Tabla de necesidades de los Sims

Para solucionar este problema se aplica una técnica llamada *dual utility AI*. En esta técnica todas las acciones se agrupan en categorías y cada categoría recibe un peso. Las categorías con la prioridad más alta se procesan primero, y solamente si no hay acciones válidas en la categoría más alta se procesan acciones de otras categorías.[6]

2.4 Árboles de decisión

Cómo indica el nombre de esta técnica, la lógica que controla el comportamiento del agente está definida en árboles. Los nodos del árbol pueden ser de distintos tipos: de acción, condicionales y compuestos. Los nodos de acción contienen las acciones que el agente puede ejecutar, los condicionales determinan qué rama del árbol debe ejecutarse y los compuestos contienen un conjunto de nodos de acción hijos que deben de ejecutarse juntos, como por ejemplo una acción de disparar y la animación de disparo.[8]

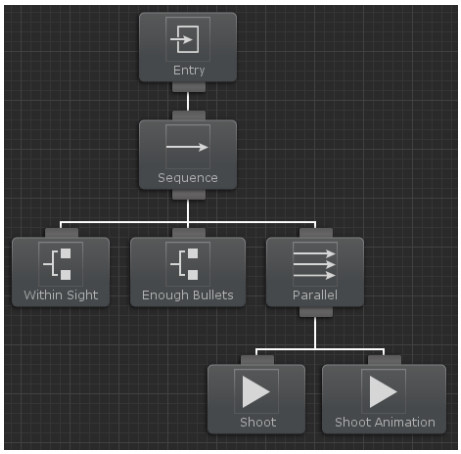


Figura 5 Ejemplo de árbol de decisión [9]

[2] Serie de videojuegos de simulación social desarrollada por Electronic Arts

Puesto que evaluar árboles muy grandes en tiempo de ejecución es muy costoso, generalmente el árbol del agente se divide en subárboles más pequeños que contienen distintas partes del comportamiento y, similarmente a como funcionan las máquinas de estado, se cambia entre los distintos subárboles conforme se necesita.

2.5 Representación del conocimiento en los videojuegos

Los sistemas de inteligencia artificial deben procesar datos para poder tomar decisiones, ya que el comportamiento del agente depende usualmente más de la información que procesa que de la técnica de IA que utiliza [8]. Es por este motivo que es crucial representar los datos necesarios adecuadamente.

Aunque distintos juegos tienen distintos requisitos para su IA, los tipos de datos necesarios pueden, generalmente, clasificarse en las siguientes categorías:

- Información del mundo estática
- Información espacial dinámica
- Información sobre entidades

La información estática representa la estructura del mundo del juego. Es un tipo de información que no cambia o lo hace muy raramente. Contiene los objetos físicos del mundo con los que los agentes pueden interactuar como los recursos en juegos de estrategia. La información de navegación para el path finding, el NavMesh[3], de los agentes también se incluye en esta categoría puesto que sería demasiado costosa de construir en tiempo de ejecución. Existen distintas maneras de representar la información de navegación. La más utilizada incluye las áreas por las que los agentes pueden desplazarse y los obstáculos del mapa, pero dependiendo del juego los agentes pueden necesitar más información. Por ejemplo, un ladrón que busque desplazarse por las sombras necesitará información sobre zonas iluminadas y oscuras.[8]

La información espacial dinámica, como su nombre indica, se utiliza para representar datos espaciales que cambian con el tiempo, incluye elementos como por ejemplo cambios territoriales o el posicionamiento de las unidades en juegos de estrategia. Una forma común de representar esta información es utilizando mapas de influencia que suelen implementarse como un array que toma la información de la posición de las unidades. Un mapa de influencia también puede representar cómo se extienden fenómenos dinámicos por la superficie del mundo del juego como incendios o inundaciones.[8]

Finalmente, la información sobre entidades es aquella que los agentes necesitan conocer sobre las entidades del juego. Es completamente dependiente del tipo de juego y puede variar también entre distintos tipos de agentes. Existen diversas aproximaciones a la forma en la que los agentes pueden adquirir la información dependiendo de las necesidades del juego. Por ejemplo, los agentes pueden tener un sistema sensorial que sea el encargado de determinar qué otras entidades puede ver el agente.[8]

Otra consideración importante es qué datos deben ser privados y qué datos pueden ser conocidos por otros agentes. Dependiendo del tipo de juego, hay información que debe ser

[3] Estructura de datos que describe las superficies caminables del mundo del juego y permite encontrar el camino de una ubicación a otra.

conocida por todos los agentes, información que solo los agentes que presencian un evento deben conocer y finalmente información privada relativa a cada agente. Ejemplos de estos tipos de información pueden ser, las localizaciones de objetos estáticos en el mundo del juego, un guardia avistando a un intruso que conoce la posición del intruso, (mientras que otro guardia puede no conocerla) y el inventario personal de cada agente, respectivamente.[8]

3. Diseño de la solución

3.1 Análisis del problema

Castle Strike es un videojuego que originalmente fue diseñado para el juego multijugador sin tener en cuenta el modo de un jugador. La *Figura 6* muestra la arquitectura que se creó para el juego originalmente.

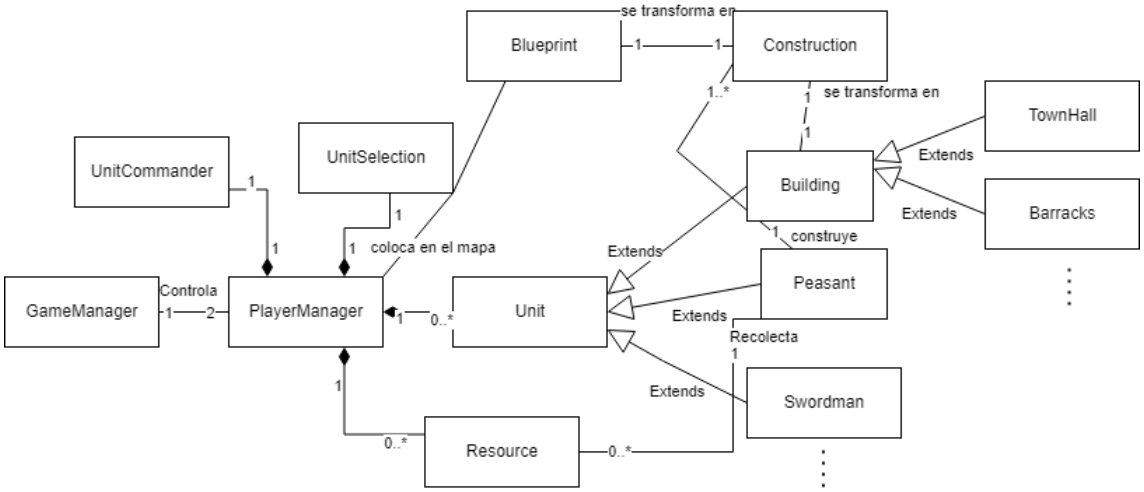


Figura 6 Arquitectura original de Castle Strike

La clase *PlayerManager* es la encargada de recibir y procesar las entradas del jugador. Utiliza las clases *UnitCommander* y *UnitSelection* para interactuar con las unidades y edificios, y contiene la lógica que controla la interfaz de usuario. La clase *GameManager* gestiona la partida, se encarga de cargar a los jugadores en la partida, realiza la preparación inicial de la partida instanciando las unidades y edificios iniciales y de empezar y terminar la partida.

La funcionalidad multijugador está implementada en el juego utilizando PUN (Photon Unity Networking) que es un framework p2p[4]. En Photon todos los gameObjects[5] de la partida deben tener un componente[6] Photon View que identifica un gameObject a través de la red y permite sincronizarlo entre todos los jugadores. Además, para sincronizar acciones entre los ordenadores de los distintos jugadores Photon utiliza llamadas RPC[7]. En el juego la selección de objetivos de las unidades y la función de ataque están implementadas utilizando RPC para garantizar que ocurre lo mismo de forma consistente en los ordenadores de todos los jugadores. Photon incluye un modo offline que permite reutilizar todo el código multijugador en el modo de un jugador y es aprovechando esta funcionalidad que se ha podido desarrollar la IA del videojuego sin apenas cambiar la arquitectura existente, tan solo ampliándola dónde ha sido necesario.

[4] Peer-to-peer o red de pares, es un tipo de red de ordenadores en la que los clientes se conectan directamente entre sí mismos.
[5] Unidad básica para construir las escenas de Unity.
[6] Unidad que dota de funcionalidad a un gameObject.
[7] En computación distribuida, técnica utilizada para ejecutar código de forma remota en una máquina distinta a la que realiza la llamada

3.2 Diseño de la máquina de estados

Aunque el comportamiento de las unidades del juego ya está modelado como una máquina de estados finita, la implementación original no utiliza correctamente el patrón de máquina de estado, resultando en un código difícil de entender, mantener y de extender. Por este motivo se ha rediseñado el funcionamiento de las unidades.

En primer lugar, se han organizado las distintas acciones que puede realizar cada unidad y se han creado tablas de transición de estados para representar la funcionalidad de cada unidad de forma simple y fácil de entender a la hora de programar.

Campesino

Estado actual	Condición	Nuevo estado
Cualquier estado	Target == null	Inactivo
Recolectando	Target == resource && isFull	Depositar recursos
Cualquier estado	Target == building	Construir edificio
Cualquier estado	Target == resource	Recolectar recurso
Cualquier estado	Target == enemy	Atacar enemigo

Soldado

Estado actual	Condición	Nuevo estado
Cualquier estado	Target == null	Inactivo
Cualquier estado	Target == enemy	Atacar enemigo
Inactivo	Enemigo en rango	Atacar enemigo

Arquero

Estado actual	Condición	Nuevo estado
Cualquier estado	Target == null	Inactivo
Cualquier estado	Target == enemy	Atacar enemigo
Inactivo	Enemigo en rango	Atacar enemigo

Caballero

Estado actual	Condición	Nuevo estado
Cualquier estado	Target == null	Inactivo
Cualquier estado	Target == enemy	Atacar enemigo
Inactivo	Enemigo en rango	Atacar enemigo

Catapulta

Estado actual	Condición	Nuevo estado
Cualquier estado	Target == null	Inactivo
Cualquier estado	Target == enemy	Atacar enemigo
Inactivo	Enemigo en rango	Atacar enemigo

Como se puede observar las transiciones de estado son dependientes del objetivo de la unidad y la unidad puede transicionar de cualquier estado a cualquier otro excepto en el caso del estado de recolección del campesino, que solo puede transicionar a depositar recursos una vez tiene su capacidad máxima de recursos. Por otra parte, las unidades de combate comparten todas los mismos estados ya que no cuentan con comportamientos únicos.

Los diagramas de estados resultantes son los siguientes:

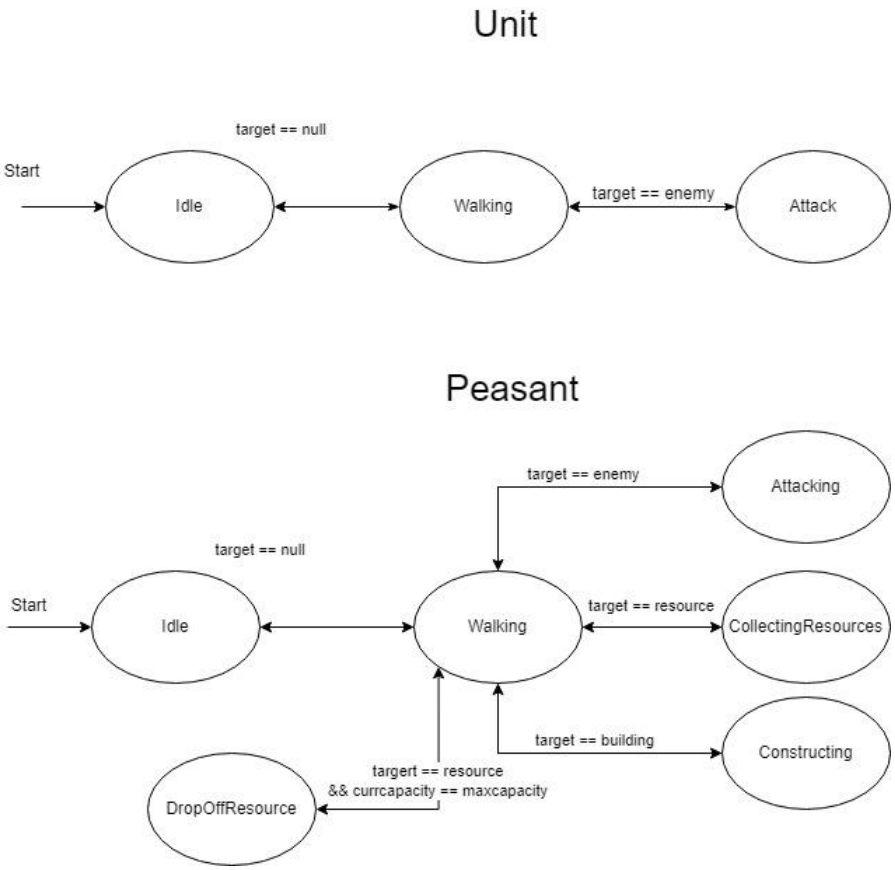


Figura 7 Diagrama de estado del campesino y de las unidades

Nótese el estado *Walking* en el diagrama, que no está incluido en las tablas de transición de estado ya que cada estado lleva integrado su movimiento en caso de necesitarlo, se ha añadido para facilitar la representación en el diagrama.

3.2.1 Arquitectura de la máquina de estados

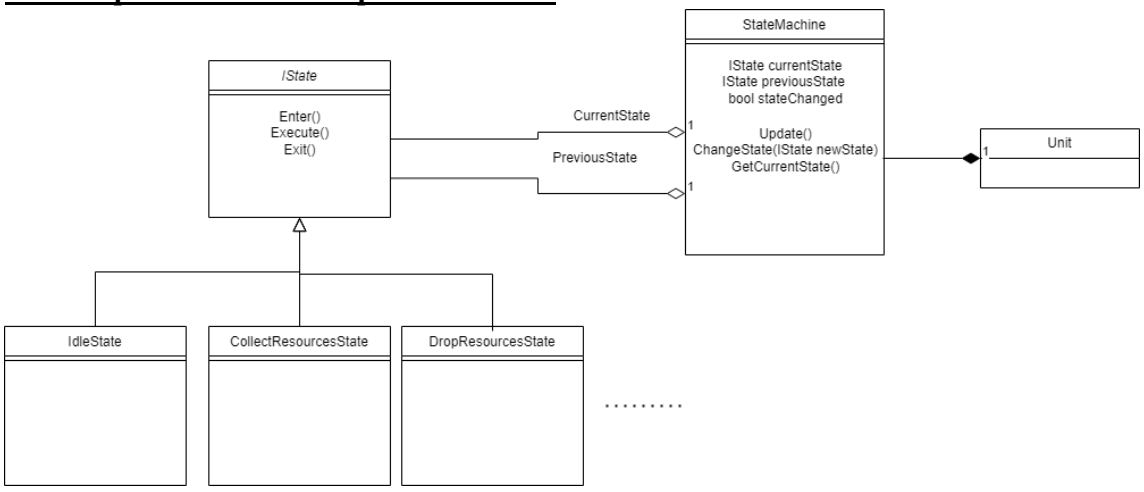


Figura 8 Diseño de la máquina de estados

El diseño de la máquina de estados es simple y reutilizable puesto que lo único que varía es la implementación de los métodos *Enter*, *Execute*, y *Exit* de cada estado. La lógica para realizar las transiciones de estado está contenida en la clase *Unit* ya que, como se ha mencionado al explicar el diseño, las transiciones de estado se realizan en función del objetivo de la unidad. La unidad usa la clase *StateMachine* para ejecutar los distintos estados y para cambiar de estado cuando es necesario.

Los estados implementan la interfaz *IState* que cuenta con los tres métodos ya mencionados. *Enter* contiene lógica para preparar lo que debe hacer la unidad en ese estado, como indicar qué animación se debe reproducir e indicar, como se verá en el siguiente apartado, a la clase *Manager* a la que pertenece la acción que va a realizar. *Execute* contiene la lógica de la acción que debe realizar la unidad en el estado, recolectar recursos en *CollectResourcesState*, descargarlos en *DropResourcesState*, etc. Finalmente, *Exit* cumple la misma función que *Enter* pero a la inversa, desactiva la animación del estado del cual está saliendo la unidad y comunica a la clase *Manager* que ha dejado de realizar cierta acción.

3.3 Diseño de la Utility AI

Para desarrollar la IA contra la que se enfrentará el jugador se ha utilizado la técnica de Utility AI por los siguientes motivos:

- Es similar a la máquina de estados, pero reemplazando los estados por acciones y las transiciones por funciones de utilidad.
- Al ejecutar las acciones una a una no tiene la complejidad de navegar árboles y de tener que construirlos en tiempo de ejecución como sí ocurre en los agentes GOAP o los árboles de decisión.
- La capacidad de expresar los factores de utilidad en lenguaje natural hace que resulten intuitivas y fáciles de implementar.
- Las funciones de utilidad no realizan computaciones complejas lo que hace muy escalable la cantidad de acciones que el agente es capaz de ejecutar.
- Agrupar las acciones similares con la técnica de Dual Utility AI permite obtener un mejor comportamiento.

El funcionamiento del agente es como sigue: en primer lugar, se calcula la utilidad de cada categoría de acciones y de las acciones que no tienen categoría. En caso de ser una acción sin categoría la elegida se ejecuta y se repite el proceso. En caso de ser una categoría la escogida se calcula la utilidad de las acciones de esa categoría y se escoge la acción con la mayor utilidad. Si hubiese dos o más acciones con la misma utilidad se selecciona al azar entre las acciones empatadas. Tras ejecutar la acción se repite el proceso hasta que termina la partida. Una posible ejecución podría ser: el agente evalúa las acciones y categorías resultando en que la categoría de recolección tenga la utilidad más alta, entonces se llama a *ExecuteAction* de la categoría, que evalúa la utilidad de las tres acciones que contiene y selecciona la que tenga mayor, se llama a *ExecuteAction* de la acción escogida, se ejecuta y se repite el proceso.

Acciones sin categoría:

Acción	Variables de puntuación	Función de utilidad
Reclutar campesino	Número de campesinos	1.5 – (0.05*campesinos)
	Hay recursos suficientes	-1000
Asignar campesino a construcción	Hay construcciones pendientes y no hay 5 campesinos en cada construcción pendiente	1000
Defenderse	Unidades enemigas dentro del área de defensa	1000

La acción de reclutar campesinos no está agrupada con el resto de las acciones de reclutamiento porque al no ser una unidad de combate su función de utilidad se calcula de forma diferente. Cabe destacar que la fórmula empieza en un valor de 1.5, esto es para asegurar que la IA recluta una gran cantidad de campesinos para evitar que económicamente el jugador pueda dejar atrás a la IA fácilmente. La acción de asignar campesinos a la construcción tampoco está agrupada en ninguna categoría puesto que no encaja con el resto de acciones. Su función de utilidad es muy simple, por cada edificio en construcción trata de asignar al menos a 5 campesinos a cada una.

La acción de defenderse no está agrupada con las acciones de ataque porque podría ocurrir que la IA no evalúe con la mayor utilidad las acciones de ataque si se ve atacada y entonces que no reaccione, por este motivo si la IA está bajo ataque reacciona dando máxima prioridad a la acción de defensa independientemente de la evaluación de las acciones de ataque. Para calcular si la IA está bajo ataque monitoriza una distancia mínima de seguridad y en el momento en que unidades enemigas cruzan ese umbral se considera que se está siendo atacado.

Acciones de recolección

Acción	Factores de utilidad	Función de utilidad
Recolectar madera	Número de campesinos asignados	1 – (0.05*campesinos asignados)
Recolectar oro	Número de campesinos asignados	1 – (0.05*campesinos asignados)
	Tiene cuartel o arquería	-1000
Recolectar piedra	Número de campesinos asignados	1 – (0.05*campesinos asignados)

Las acciones de recolección tratan de repartir equitativamente a los campesinos entre los recursos disponibles excepto en el caso de no tener cuartel o arquería, en esa situación la IA no recolecta oro porque no podría utilizarlo.

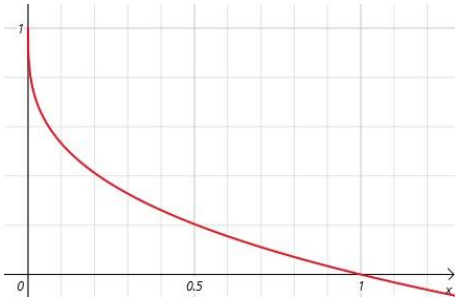
Acciones de construcción

Acción	Factores de utilidad	Función de utilidad
Construir granja	Número de granjas	$1 - x^2$
	Hay espacio de construcción	-1000
	Tiene recursos	-1000
Construir cuartel	Número de cuarteles	$1 - x^{0.33}$
	Hay espacio de construcción	-1000
	Tiene recursos	-1000
Construir arquería	Número de arquerías	$1 - x^{0.33}$
	Hay espacio de construcción	-1000
	Tiene recursos	-1000

Dónde x = $\frac{edificios\ del\ tipo}{espacios\ de\ construcción}$

Las acciones de construir edificios dependen primeramente de si hay espacio de construcción disponible y recursos suficientes para construir los edificios. La función de utilidad se calcula en función del número de edificios de ese tipo y del espacio de construcción disponible restante. Las acciones de construir cuartel y arquería utilizan una curva cuadrática invertida para que al principio su utilidad decaiga más rápidamente para asegurar que la IA construye un número adecuado de granjas. Las siguientes gráficas muestran las curvas respectivas de las acciones:

Gráfica 1 Función de utilidad de construir granja



Gráfica 2 Función de utilidad de construir cuartel y arquería

Acciones de reclutamiento

Acción	Factores de utilidad	Función de utilidad
Reclutar soldado	Tiene recursos	-1000
	Coste de recursos	$\frac{\sum_1^n x_i}{n}$
	Poder militar	
	Tiempo de reclutamiento	
Reclutar caballero	Tiene recursos	-1000
	Coste de recursos	$\frac{\sum_1^n x_i}{n}$
	Poder militar	
	Tiempo de reclutamiento	
Reclutar Arquero	Tiene recursos	-1000
	Coste de recursos	$\frac{\sum_1^n x_i}{n}$
	Poder militar	
	Tiempo de reclutamiento	
Reclutar Catapulta	Tiene recursos	-1000
	Coste de recursos	$\frac{\sum_1^n x_i}{n}$
	Poder militar	
	Tiempo de reclutamiento	

Donde $\sum_1^n x_i$ representa el sumatorio de los factores de utilidad, siendo estos coste de recursos y poder militar para balances de fuerza positivos y poder militar y tiempo de reclutamiento para balances negativos.

Las acciones de reclutamiento dependen del balance de fuerza entre el jugador y la IA. El balance se calcula con la diferencia entre las unidades del jugador y de la IA. Cada unidad tiene un parámetro que representa su fuerza y la suma de la fuerza de todas las unidades es la fuerza total del jugador o de la IA. Si el balance de fuerza es positivo en favor a la IA, esta calcula la utilidad de las unidades en función de los recursos que cuesta frente al total actual y respecto al poder militar de la unidad frente a la suma del poder de todas las unidades.

Si la IA tiene balance de fuerza negativo calcula la utilidad en función del tiempo de reclutamiento de la unidad frente a la que tiene el tiempo mayor y del poder militar de la unidad.

Acciones de ataque

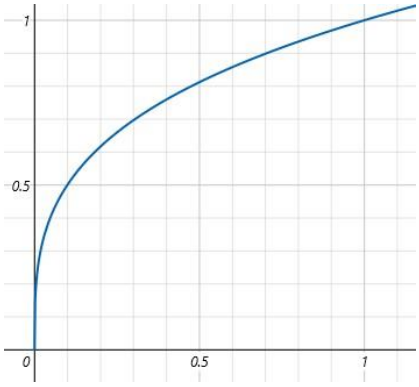
Acción	Factores de utilidad	Función de utilidad
Atacar cuarteles	Número de cuarteles enemigos	$\frac{\sum_1^n x_i}{n}$
	Distancia del cuartel más cercano	
Atacar arquerías	Número de arquerías enemigas	$\frac{\sum_1^n x_i}{n}$
	Distancia de la arquería más cercana	
Atacar granjas	Número de granjas enemigas	$\frac{\sum_1^n x_i}{n}$
	Distancia de la granja más cercana	
Atacar unidades	Número de unidades enemigas	$x^{0.3}$
Atacar ayuntamiento	Balance de poder > 20	1

Donde $\sum_1^n x_i$ representa el sumatorio de los factores de utilidad

A la hora de diseñar las estrategias de ataque se ha tenido que decidir si la IA es capaz de ver o no lo que posee el jugador. Dado que no hay ningún tipo de niebla de guerra u obstáculo que impida ver lo que posee no tiene sentido ocultar esta información a la IA por lo que cada vez que toma la decisión de realizar una acción de ataque puede consultar lo que posee el jugador para decidir el objetivo. Su funcionamiento es simple: cuando se decide una estrategia la IA envía todas las unidades que no estén en combate a atacar el objetivo que se ha decidido.

La utilidad de las acciones que atacan edificios se calcula obteniendo el valor medio de dos factores de utilidad. El primero es el número de edificios de ese tipo en proporción al total de edificios del enemigo y el segundo es la distancia del edificio más cercano del tipo con relación a la distancia mínima de seguridad explicada en la acción de defensa.

Finalmente, la acción de atacar unidades utiliza una curva cuadrática invertida como la que se muestra en la siguiente gráfica para que la IA priorice acabar con las unidades enemigas antes de tratar de destruir la base.

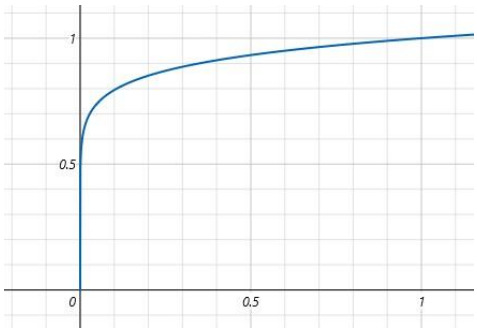


Gráfica 3 Función de utilidad de atacar unidades

Evaluación de la utilidad de las categorías

Categoría	Factores de utilidad	Función de utilidad
Recolección	Número de trabajadores inactivos	$x^{0.1}$
Construcción	Número de espacios de construcción disponibles	$1 - \frac{1}{1 + e^{(-x*7.5)+6}}$
	Tiene recursos suficientes	-1000
Reclutamiento	Balance de poder > 10	1 - x
	-10 <= Balance de poder <= 10	$1 - \frac{1}{1 + e^{(-x*12)+6}}$
	Balance de poder < -10	1
Ataque	Balance de poder > 0 Número de unidades > 5	$\frac{1}{1 + e^{(-x*30)+6}}$

La utilidad de la categoría de recolección se calcula en función de la proporción de campesinos inactivos frente al total de campesinos y tiene un exponente muy pequeño para que, aunque haya una cantidad pequeña de campesinos inactivos, asignarles una tarea con una utilidad alta. ya que un campesino que no hace nada es un campesino inútil. La gráfica siguiente muestra la curva de la función de utilidad:



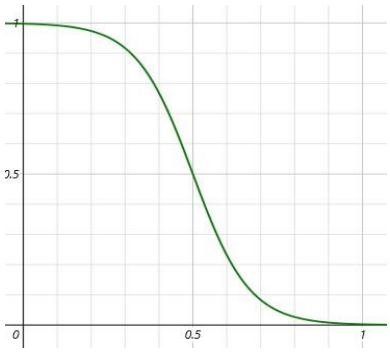
Gráfica 4 Función de utilidad de la categoría recolección

La categoría de construcción en primer lugar valora que haya recursos suficientes para construir, al menos, el edificio más barato y, tras esa verificación, la utilidad se calcula utilizando una curva logística donde x es la proporción de espacio de construcción libre, se multiplica el exponente por 7.5 para asegurar que cuando hay una cantidad de espacio libre superior al 50% del total la construcción tiene una utilidad elevada y el +6 al exponente para asegurar que el resultado es un número positivo.

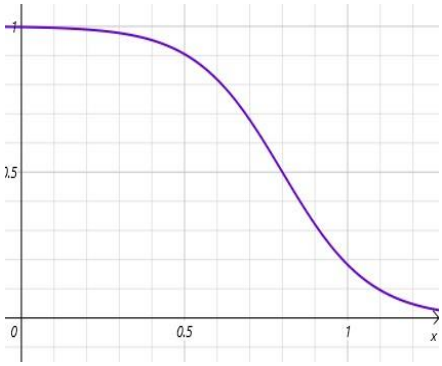
La categoría de acciones de reclutamiento tiene tres funciones distintas para calcular la utilidad y se escogen en función del balance de poder actual. Si este es mayor que 10, la utilidad se calcula en función de la proporción de unidades frente a campesinos. Si se encuentra entre 10 y -10 utiliza una curva logística con exponente multiplicado por 12 que provoca que la curva sea más o menos equilibrada donde un valor de x=0.5 devuelve y=0.5. Finalmente, si es menor que -10 se fija a uno para que la IA trate de evitar estar en una gran desventaja.

La categoría de ataque primero valora que la IA tenga un balance de poder positivo y un mínimo de 5 unidades libres, ya que hacer ataques con muy pocas unidades no tiene demasiado sentido. Tras eso, utiliza una curva logística donde x es la proporción de poder militar de la IA frente al jugador, y se multiplica el exponente por 30 para que la utilidad crezca rápidamente con una ventaja pequeña para tratar de crear una IA no demasiado cauta.

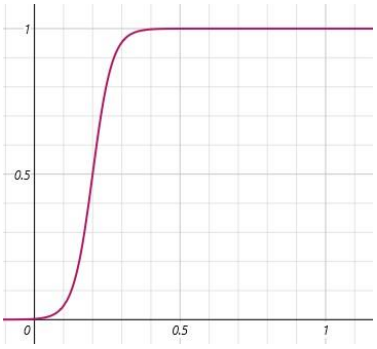
Las siguientes gráficas muestran las curvas respectivas de las funciones de construcción, reclutamiento y ataque. Se puede observar claramente cómo el distinto exponente afecta a las funciones.



Gráfica 5 Función de utilidad de la categoría reclutamiento



Gráfica 6 Función de utilidad de la categoría construcción



Gráfica 7 Función de utilidad de la categoría ataque

3.3.1 Arquitectura de la IA

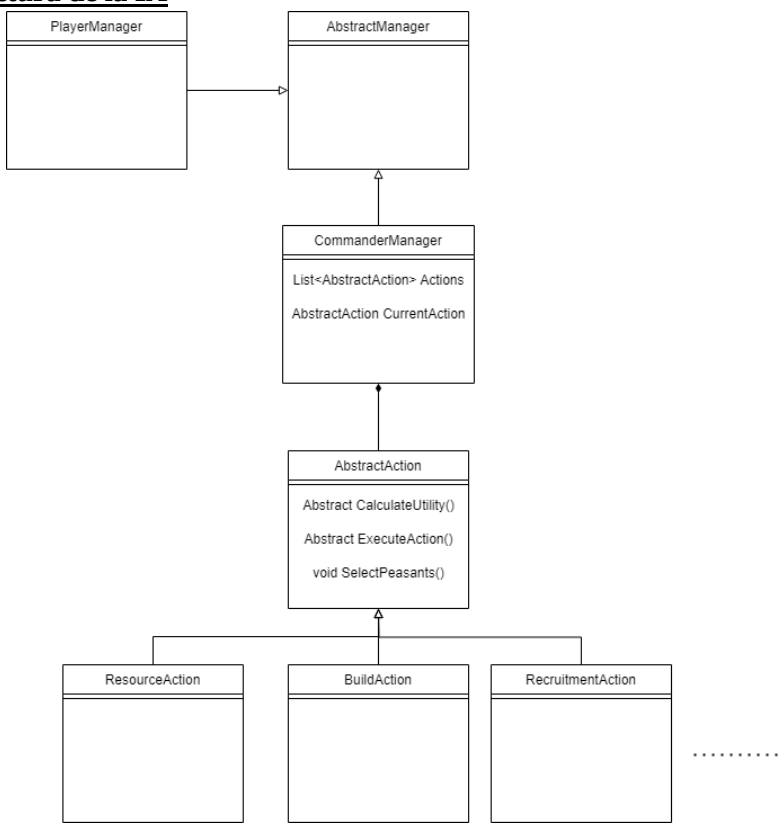


Figura 9 Diseño de la IA

La *Figura 9* muestra la arquitectura de la Utility AI del juego. En el apartado 3.1 se ha mencionado que, en la implementación original la clase *PlayerManager* contiene la información sobre unidades y edificios del jugador. Para poder reutilizar la mayor parte de esta clase en el diseño de la IA, se ha refactorizado y se ha extraído toda la lógica referente a la gestión de unidades y edificios a la nueva clase *AbstractManager*. Además, se ha ampliado para poder ver desde la clase qué tareas desempeña cada unidad de manera que contiene referencias a los campesinos por tipo de recurso que recolectan, a los campesinos que están construyendo un

edificio, a los que están inactivos, etc. En la implementación original de *Castle Strike* no era necesaria está lógica ya que los jugadores podían ver fácilmente lo que están haciendo sus unidades. Sin embargo, para el agente es necesaria para determinar a qué unidades asignar qué tareas, y además permite también mejorar la experiencia del jugador ya que al saber que tarea realiza cada unidad se puede, por ejemplo, notificar al jugador cuando tiene campesinos inactivos.

Las acciones que pertenecen a categorías *ResourceAction*, *RecruitmentAction*, etc. se instancian individualmente por cada acción de la categoría, es decir, existe una *ResourceAction* para la recolección de oro, una para la madera y otra para la piedra, cada una con su propia implementación de *ExecuteAction* y de *CalculateUtility* en caso de tener una función de utilidad distinta de otras acciones de su categoría.

3.4 Vista final del sistema

Tras los cambios explicados en los apartados anteriores, la arquitectura final resultante es la que se muestra en la *Figura 10*:

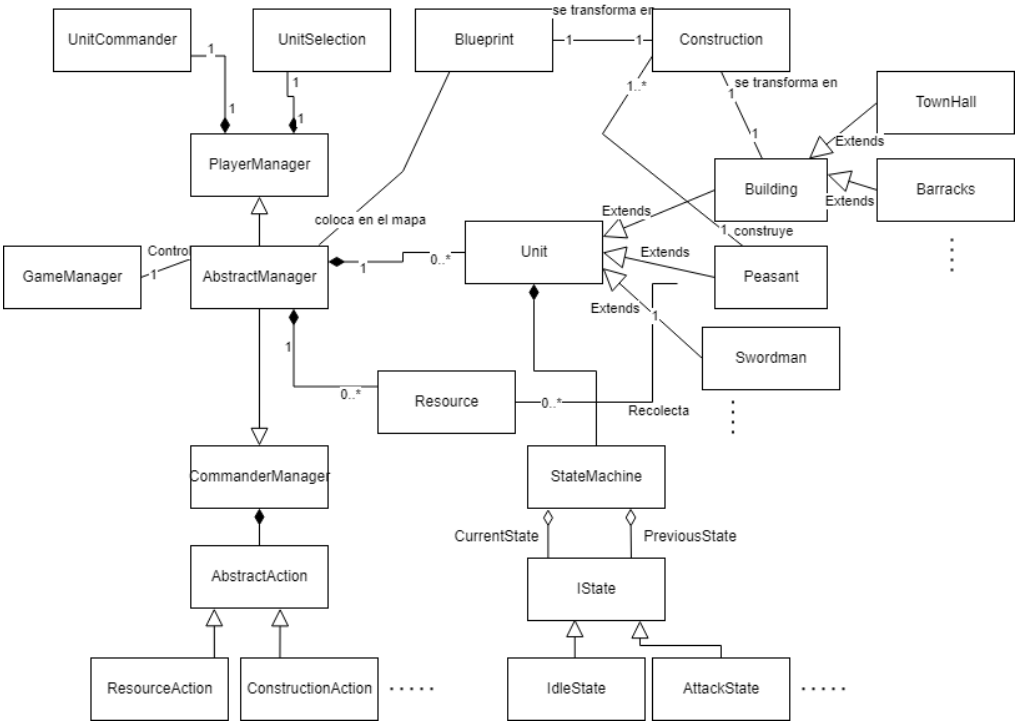


Figura 10 Arquitectura final de Castle Strike

Como se puede observar, se ha cumplido el objetivo inicial de ampliar el juego para poder jugar en modo de un jugador sin tener que realizar grandes modificaciones en la arquitectura ya existente de forma que pueda utilizarse la misma funcionalidad en ambos modos de juego.

Cabe detallar que los objetos de clase *Building* y sus derivados no utilizan la máquina de estados ya que no tienen ningún comportamiento autónomo y sólo actúan cuando se les ordena reclutar unidades, o en el caso de la granja produciendo alimento de forma constante. Sin embargo, si en el futuro se expanden los tipos de edificios se podrían añadir edificios que sí actúen de manera independiente como, por ejemplo, una torre de arqueros que dispare a los enemigos automáticamente o una muralla cuya puerta se abra cuando hay unidades aliadas cerca.

4. Desarrollo de la solución

4.1 Preparación del proyecto

Para el desarrollo original del proyecto se utilizó la versión de Unity 2020.3.11f1 LTS porque era la última versión LTS disponible en el momento. Para este trabajo se ha utilizado la versión 2022.3.4f1 LTS que es actualmente la última versión LTS con la que cuenta Unity. El motivo principal ha sido para poder utilizar la versión 1.0.0 del profiler[8] de Unity que se encuentra disponible a partir de la versión 2022.2. A la hora de desarrollar cualquier sistema de inteligencia artificial, el mayor factor limitante, siempre es el hardware por eso es imperativo utilizar una herramienta de profiling para medir el rendimiento de la IA y realizar optimizaciones en caso de ser necesario.

Actualizar la versión de Unity es muy sencillo, tan solo hay que instalar la versión del editor deseada y ejecutar el proyecto en esa versión del editor, como se muestra en la siguiente figura:

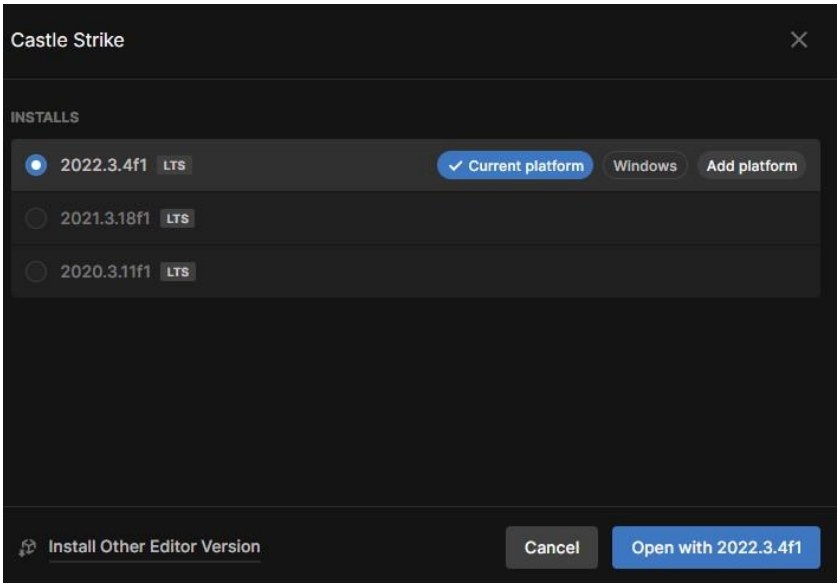


Figura 11 Selección de versión en Unity

Tras escoger la versión deseada Unity se encarga de realizar todos los cambios necesarios en el proyecto para poder utilizar la nueva versión del editor. Tras esto puede comenzar el desarrollo.

4.2 Desarrollo

En este apartado se expondrán los aspectos más interesantes del desarrollo, así como las dificultades encontradas.

4.2.1 Modificaciones para determinar equipos en el modo de un jugador

Es práctica común en Unity utilizar el sistema de tags[9] que permite identificar a los distintos gameObjects por su rol en el juego. Sin embargo, por defecto Unity solo permite asignar un tag por gameObject lo cual es muy limitante si se necesita tener subcategorías como que un edificio tenga el tag Building para identificarlo como edificio y el tag Barracks para saber que son unos cuarteles. Para solucionar este problema se desarrolló el componente CustomTag para añadir a

[8] Herramienta de análisis de costes utilizada para medir el rendimiento de una aplicación
[9] Referencia o etiqueta asignable a un gameObject para identificarlo durante la partida.

los gameObjects todos los tags que se desee y de nuevo, se ha ampliado el componente en este trabajo para utilizarlo para identificar las unidades que pertenecen al jugador y las que pertenecen a la IA.

La primera modificación realizada al respecto en el proyecto es al método *IsMine* de la clase *Unit*, como se puede presuponer por su nombre, su función es la de comprobar a quién pertenece la unidad en la que se llama. En el proyecto original el método devolvía el valor de la llamada *photonView.IsMine*. En el modo multijugador esto proporciona la funcionalidad deseada ya que cada jugar es dueño del componente photonView[10] de cada objeto instanciado en su ordenador. En el modo de un jugador esto implica que todos los gameObjects devolvían *true* al llamar a este método puesto que todas las photonView pertenecen a un único jugador. La modificación realizada hace que si se está en modo offline en lugar de devolver el valor de *PhotonView.IsMine* utilice el componente CustomTag de la unidad para comprobar si la unidad tiene el mismo tag que quién está comprobando si la unidad le pertenece. El método final es el siguiente:

```
public bool IsMine(string tag)
 {
     if (PhotonNetwork.OfflineMode)
     {
         return GetComponent<CustomTag>().HasTag(tag);
     }

     return photonView.IsMine;
 }
```

Figura 12 Método IsMine de la clase Unit

Con esta pequeña modificación el método tiene el funcionamiento correcto en el modo de un jugador y como se puede ver por la arquitectura del juego, este método permite identificar a todas las unidades y edificios del juego.

Una modificación parecida se ha realizado en las clase *Construction* que representa los cimientos de un edificio y construcción y en la clase *Blueprint* que representa la plantilla que se utiliza para colocar los edificios en el mundo del juego. La clase *Construction* también utiliza el componente photonView para sincronizarse en el modo multijugador por lo que para identificar a su dueño utilizaba un método *IsMine* igual al de la clase *Unit* y se ha modificado de la misma forma. La clase *Blueprint* no tiene un componente photonView porque no es necesario sincronizarla en el modo multijugador ya que solo el jugador que está construyendo cierto edificio puede ver la plantilla de este. Por otro lado, en el modo de un jugador sí que es necesario identificar al dueño de la *Blueprint* por lo que se le ha añadido también el componente CustomTag pero a diferencia de la clase *Unit* y *Construction* como no permanece en el mundo más que unos segundos hasta que se transforma en el edificio a construir, no requiere de un método *IsMine*, tan solo necesita conocer si pertenece al *AbstractManager* del jugador o de la IA. Para esto se le ha añadido una variable *AbstractManager* que contiene la referencia a su dueño y el método *GetPlayer* que le permite encontrarlo como se muestra en la *Figura 13*.

[10] Componente de PUN utilizado para sincronizar los gameObjects a través de la red.

```
private AbstractManager owner;

…

 private void GetPlayer()
    {
        if (PhotonNetwork.OfflineMode)
        {
            AbstractManager[] managers =
(AbstractManager[])FindObjectsOfType(typeof(AbstractManager));

            foreach (AbstractManager manager in managers)
            {
                if (GetComponent<CustomTag>().HasTag(manager.tag))
                {
                    owner = manager;
                    return;
                }
            }
        }

        GameObject[] playerManagers =
GameObject.FindGameObjectsWithTag("Player");
        foreach (GameObject player in playerManagers)
        {
            if (player.GetPhotonView().IsMine)
            {
                this.owner = player.GetComponent<PlayerManager>();
                return;
            }
        }
    }
```

Figura 13 Modificaciones a la clase Blueprint

Se puede observar que a pesar de que se ha explicado que la clase *Blueprint* no requiere sincronizarse en el modo multijugador, el método también asigna al dueño de la *Blueprint* en modo multijugador. Esto es para evitar una excepción que se produciría al llamar al método *Build* de la clase en el modo multijugador, que también se ha sido modificado como se verá más adelante.

Se han explicado las modificaciones realizadas para saber a qué equipo pertenece cada gameObject de la partida, pero todavía falta la parte más importante: las modificaciones realizadas para añadir el tag correcto a cada equipo. Primero, hay que entender el proceso de instanciación que sigue cada objeto de la partida: las unidades son creadas por los distintos edificios de reclutamiento y los edificios siguen la siguiente cadena para su construcción: *Blueprint*, *Construction* y finalmente la *Building* que se desea construir. La forma en la que se ha implementado esta funcionalidad es que cada gameObject es responsable de asignar el tag correcto a los objetos que se encarga de instanciar. Por lo tanto, los edificios de reclutamiento asignan el tag a las unidades que reclutan y la *Blueprint* asigna el tag a su *Construction* que luego lo asigna a su *Building*. El código modificado que asigna los tags es el que se muestra a continuación:

En primer lugar, el ya mencionado método *Build* de la clase *Blueprint*

```
public void Build()
    {
        GameObject newBuilding = PrefabManager.NetworkInstantiate(building,
transform.position, Quaternion.Euler(-90, 0, 0));
        newBuilding.GetComponent<CustomTag>().AddTag(owner.tag);
        Destroy(gameObject);
    }
```

Figura 14 Método Build de la clase Blueprint

El funcionamiento es muy simple: instancia el edificio, le asigna el tag del dueño de la *Blueprint* y después desaparece.

En la clase *Construction* este código se encuentra dentro del método *Update*:

```
if (health >= building.GetComponent<Building>().buildingStats.health)
        {
            GameObject finishedBuilding =
PrefabManager.NetworkInstantiate(building, transform.position,
transform.rotation);
            finishedBuilding.GetComponent<CustomTag>().AddTag(owner.tag);
            owner.RemoveFromSpecificList(gameObject,
ListType.BuildingQueue);
            if( photonView.IsMine) PhotonNetwork.Destroy(this.gameObject);
        }
```

Figura 15 Fragmento del método Update de la clase Construction

Cuando el edificio termina de construirse, instancia el edificio terminado, le asigna el tag y desaparece. Hay que destacar que aquí se utiliza *PhotonNetwork.Destroy*, porque sí que es un objeto que debe sincronizarse en el modo multijugador, pero gracias al modo offline de Photon el mismo código funciona también en el modo de un jugador.

En la clase *Building* la funcionalidad está contenida en la corutina encargada del reclutamiento de unidades. Una corutina es una función que tiene la capacidad de pausar su ejecución sin afectar a la ejecución del resto del juego. Se utiliza para regular la ejecución de funciones que deben de ejecutarse en un tiempo predeterminado o con cierta periodicidad, en este caso de uso la duración de la corutina es igual al tiempo de reclutamiento de la unidad.

Finalmente, la última modificación relativa a los equipos se encuentra en la clase *GameManager*, en la que, al instanciar las unidades y edificios iniciales de cada jugador, les asigna el tag adecuado. El resultado final es un componente CustomTag que contiene el tag de su subcategoría junto con el tag de su dueño como se muestra en la *Figura 16*.

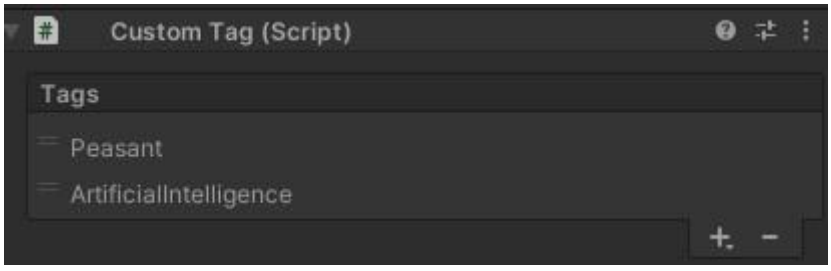


Figura 16 CustomTag de un campesino de la IA

4.2.2 Refactorizaciones realizadas

Como se ha mencionado a lo largo del apartado 3 se han realizado diversas refactorizaciones para añadir la funcionalidad de la máquina de estados y de la Utility AI al juego.

El cambio más grande ha sido en el *Update* de la clase *Peasant* puesto que de todas las unidades es la que tiene la máquina de estados más compleja. A continuación, se va a mostrar el antes y después de los cambios realizados.

```
if (target != null)
        {
            float distance = (transform.position - destination).magnitude;
            if (distance > unitStats.attackRange)
            {
                MoveToTarget();
            }
            else if (!CheckIfTargetEnemy())
            { //Attack
                collectingResources = false;
                dropOff = null;
                if (Time.time >= performAction)
                {
                    anim.SetBool("isAttacking", true);
                    anim.SetBool("isMoving", false);
                    Attack();
                    performAction = Time.time + unitStats.attackSpeed;
                }
            }
            else if (target.CompareTag("Resource") && (currCapacity + 5) <=
maxCapacity)
            { //Collect resources
                if (Time.time >= performAction)
                {
                    currentResource = target.gameObject;
                    anim.SetBool("isCollecting", true);
                    anim.SetBool("isMoving", false);
                    CollectResources();
                    performAction = Time.time + unitStats.attackSpeed;
                    collectingResources = true;
                }

            }
            else if ((collectingResources && currCapacity == maxCapacity) ||
(!collectingResources && currCapacity > 0))
            { //Drop off resources
                if (dropOff == null) { SetDropOff(); }
                navAgent.isStopped = false;
                //navAgent.stoppingDistance = unitStats.attackRange + 2f;
                if (dropOff != null)
                {
                    target = dropOff;
                    OffSetDestination();
                    navAgent.SetDestination(destination);
                    float dropDistance = (transform.position -
destination).magnitude;
                    if (dropDistance <= unitStats.attackRange + 5)
DropResources();
                }
            }
            else if (target.CompareTag("Construction") && distance <=
unitStats.attackRange)
            {   //Build
```

```
                transform.LookAt(target);
                collectingResources = false;
                if (Time.time >= performAction)
                {
                    anim.SetBool("isBuilding", true);
                    anim.SetBool("isMoving", false);
                    navAgent.isStopped = true;
                    Construct();
                    performAction = Time.time + unitStats.attackSpeed;

                }
            }
        }

        else if (target == null)
        {
            attack = false;
            collectingResources = false;
            dropOff = null;
            anim.SetBool("isAttacking", false);
            anim.SetBool("isBuilding", false);
        }
```

Figura 17 Update de la clase Peasant antiguo

En la *Figura 17* podemos ver el código del método *Update* de la clase *Peasant* antes de la implementación del patrón de la máquina de estados. Es un método demasiado grande, difícil de entender, difícil de depurar y de extender, es decir, es mal código. A continuación, en la *Figura 18* se muestra el resultado de la refactorización resultante tras implementar la máquina de estados.

```
if (target != null)
        {
            if (target.CompareTag("Resource") && (currCapacity + 5) <=
maxCapacity)
            {
                if (stateMachine.GetCurrentState() != gatherResourcesState)
                    stateMachine.ChangeState(gatherResourcesState);
            }

            else if ((collectingResources && currCapacity == maxCapacity) ||
                (currCapacity > 0 && target.CompareTag("Townhall")) &&
target.GetComponent<Unit>().IsMine(owner.tag))
            {
                if (stateMachine.GetCurrentState() != dropResourcesState)
                    stateMachine.ChangeState(dropResourcesState);
            }
            else if (target.CompareTag("Construction") &&
target.GetComponent<Construction>().IsMine(GetComponent<CustomTag>().GetAtIn
dex(1)))
            {
                if (stateMachine.GetCurrentState() != buildState)
                    stateMachine.ChangeState(buildState);
            }
            else if
(!target.GetComponent<Unit>().IsMine(GetComponent<CustomTag>().GetAtIndex(1)
))
            {
                if (stateMachine.GetCurrentState() != attackState)
                {
                    stateMachine.ChangeState(attackState);
```

```
                }
            }

        }
        else if (target == null && stateMachine.GetCurrentState() !=
idleState) stateMachine.ChangeState(idleState);

        stateMachine.Update(this);
```

Figura 18 Update de la clase Peasant tras la refactorización

Cómo podemos comprobar, el tamaño del método se ha reducido significativamente ya que la lógica de cada acción del campesino ahora está contenida en la clase de su propio estado y el método solo es responsable de gestionar las transiciones de estado.

Las siguientes refactorizaciones son las realizadas en la clase *PlayerManager* que como se explicó en el apartado 3, es la que contiene las referencias a las unidades y edificios y procesa las entradas del jugador. Los cambios realizados han consistido en extraer toda la lógica relacionada con la gestión de unidades y edificios a la nueva clase *AbstractManager*. Además, se ha ampliado la clase para que pueda distinguir entre las unidades según la tarea que realizan, para que la IA pueda gestionar sus unidades adecuadamente. En la *Figura 19* a continuación se muestra todas las tareas que distingue la clase *AbstractManager*:

```
[Header("Unit states")]
    [SerializeField]
    protected List<GameObject> peasants;
    [SerializeField]
    protected List<GameObject> idlePeasants;
    [SerializeField]
    protected List<GameObject> selectedPeasants;
    [SerializeField]
    protected List<GameObject> builderPeasants;
    [SerializeField]
    protected List<GameObject> goldPeasants;
    [SerializeField]
    protected List<GameObject> woodPeasants;
    [SerializeField]
    protected List<GameObject> stonePeasants;
    [SerializeField]
    protected List<GameObject> idleTroops;
    [SerializeField]
    protected List<GameObject> fightingTroops;

    [Header("Buildings")]
    [SerializeField]
    protected GameObject Townhall;
    [SerializeField]
    protected List<GameObject> barracks;
    [SerializeField]
    protected List<GameObject> archeryRange;
    [SerializeField]
    protected List<GameObject> farm;
    [SerializeField]
    protected List<GameObject> buildingQueue;
```

Figura 19 Fragmento de la clase AbstractManager

Las listas mostradas en la figura contienen a los campesinos según la tarea que realizan según indica el nombre de las variables excepto la lista peasants que contiene el total de todos los

campesinos que se poseen y selectedPeasants, que contiene a los campesinos que la IA ha seleccionado para asignar a una acción. Continuando, idleTroops y fightingTroops contienen a las unidades de combate en función de si están combatiendo o no y las listas de los edificios contienen los distintos edificios para asignarles las acciones de reclutamiento. Finalmente, buildingQueue contiene los edificios que están siendo construidos para poder asignarles constructores.

Todas estas listas necesitan ser accedidas y modificadas a lo largo de toda la partida para que la IA pueda realizar acciones por lo tanto hacen falta métodos para poder modificar las listas. En lugar de generar un método *Add*, *Remove* y *Get* para cada lista se han creado una serie de métodos genéricos que permiten acceder a la lista que se desee modificar con facilidad. La *Figura 20* muestra esta implementación parcialmente. En primer lugar, el *Enum ListType* y el método *GetListByType* nos permiten acceder a la lista deseada fácilmente y con los métodos *AddToSpecificList* y *RemoveFromSpecificList* podemos modificar las listas.

```
public void AddToSpecificList(GameObject objectToAdd, ListType listType)
    {
        List<GameObject> targetList = GetListByType(listType);
        if (targetList != null)
        {
            targetList.Add(objectToAdd);
        }
    }

public enum ListType

{
    Peasants,
    IdlePeasants,
    BuilderPeasants,
    GoldPeasants,
    WoodPeasants,
    StonePeasants,
    IdleTroops
    FightingTroops,
    Barracks,
    ArcherRange,
    Farm,
    SelectedPeasants,
    BuildingQueue }
```

Figura 20 Métodos para gestionar las listas de AbstractManager

Además, para aligerar la clase *CommanderManager* y hacer la comunicación bidireccional entre ambos tipos de agentes, son las propias unidades las encargadas de indicar que tarea están realizando, como se puede ver en la *Figura 21* en el método *Enter* de la clase *BuildState.*

```
public void Enter(Unit owner)
    {
        owner.GetComponent<Animator>().SetBool("isBuilding", true);
        owner.GetComponent<Animator>().SetBool("isMoving", false);

owner.GetComponent<Peasant>().GetManager().RemoveFromAllLists(owner);

owner.GetComponent<Peasant>().GetManager().AddToSpecificList(owner.gameObjec
t,ListType.BuilderPeasants);
    }
```

Figura 21 Método Enter del estadoBuildState

Cómo se explicó en el apartado 3, cada unidad es responsable de indicar al *AbstractManager* la acción que está desempeñando. El método *RemoveFromAllLists* es otro método de la clase *AbstractManager* cuya función es eliminar de todas las listas de acciones a los campesinos cuando inician una tarea para asegurar que solo están asignados en la tarea correspondiente.

4.2.3 Inercia y acciones periódicas

Un problema común de la Utility AI es la inercia. Este problema se da cuando se ejecuta una acción que tarda en alterar el estado del juego y que por lo tanto puede provocar que el agente la evalúe muchas veces consecutivas con el mismo valor de utilidad y se quede atascado repitiendo la misma acción una y otra vez sin provocar cambios. Para solucionarlo, a este tipo de acciones se les asigna un tiempo de reutilización de forma que una vez ejecutada la acción que provoca inercia no se pueda volver a ejecutar hasta pasado un determinado periodo de tiempo. Esto da la oportunidad de que se actualice el estado del mundo, permitiendo que la IA reevalúe la utilidad de la acción adecuadamente. Como se ha explicado en el apartado anterior, las corutinas son funciones de Unity cuya ejecución puede detener y se utilizan para realizar acciones prolongadas en el tiempo o periódicas y han sido la elección para implementar este tiempo de reutilización a las acciones.

```
protected IEnumerator SetCooldown()
{
    isInCooldown = true;
    yield return new WaitForSeconds(coolDown);
    isInCooldown = false;
}
```

Figura 22 Corutina SetCooldown

En la *Figura 22* se muestra la corutina que establece el tiempo de reutilización de las acciones: simplemente hay que llamarla al final de la ejecución de la acción. Las acciones que implementan el tiempo de reutilización son las de ataque y las de reclutamiento, ya que son las dos acciones que no tienen un efecto inmediato sobre el mundo.

En el apartado 3 se ha explicado que la IA debe ser capaz de ver todo lo que posee el jugador, Los sensores del agente, son los encargados de hacer consultas al mundo para comprobar lo que ocurre en él, y en concreto el que tiene la IA que se encarga de detectar si está siendo atacada para poder defenderse. Si la lógica de este sensor se encontrara en alguno de los métodos *Update* de Unity significa que se llamaría mínimo una vez por frame. En un juego a 60FPS eso serían 60 veces por segundo y, al ser una función que tiene comprobar si alguna unidad enemiga está atacando conforme aumenta el número de unidades, las comprobaciones que tiene que realizar cada vez aumentan. Si además se llama tantas veces por estar en el método *Update*, podría provocar serios problemas de rendimiento que irían empeorando conforme avanza la partida. Por este motivo esta tarea debe realizarse con una corutina que se ejecute periódicamente. En la *Figura 23* se muestra la corutina encargada de detectar si la IA está siendo atacada o no. La función *ProximityCheck* simplemente determina si hay enemigos a menos de una distancia de seguridad predeterminada.

```
private IEnumerator CheckEnemies()
    {
        for (; ; )
        {
            isUnderAttack = ProximityCheck();
            yield return sensorSweep;
        }
    }
```

Figura 23 Corutina CheckEnemies

La palabra clave `yield` indica a Unity dónde se suspende la ejecución de la corutina mientras que `sensorSweep` es un objeto de tipo `WaitForSeconds` como el que se muestra en la *Figura 22* pero por motivos de eficiencia dado que el tiempo de espera no varía, en esta corutina es mejor que esté guardado en una variable en lugar de instanciar uno nuevo cada vez.

5. Pruebas

Las pruebas realizadas en el proyecto han tenido dos frentes. En primer lugar, se ha utilizado el profiler de Unity, como se explicó en el apartado 4.1, para medir el rendimiento de la aplicación en funcionamiento y verificar que la IA funciona eficientemente.

En segundo lugar, se ha publicado el juego en la página itch.io que es una plataforma de distribución digital destinada a videojuegos independientes, que son aquellos desarrollados por estudios desarrollo pequeños sin grandes empresas financiando los proyectos. En la página se ha indicado que *Castle Strike* es un prototipo desarrollado como trabajo de fin de máster universitario y se ha añadido una encuesta para evaluar la satisfacción de los jugadores. Además, se ha utilizado Unity Analytics para conocer la proporción de jugadores que responden a la encuesta y la duración media de las sesiones de juego.

5.1 Resultados del Profiler

A la hora de utilizar el profiler se realizaron pruebas tanto en el modo juego del editor de Unity como en una build del juego configurada para ser utilizada con el profiler.

El profiler tiene dos modos de uso: se puede utilizar integrado en el editor, dentro del proceso de este, o se puede utilizar por separado ejecutándolo en un proceso separado. Se ha utilizado el segundo método, ya que ejecutar el profiler dentro del editor de Unity puede afectar al rendimiento.

Otro factor a tener en cuenta es el ordenador en el que se realizan las pruebas, ya que un hardware potente proporcionará un mejor rendimiento y podría enmascarar problemas. Por este motivo los estudios de videojuegos prueban los juegos en diversas combinaciones de hardware para encontrar los requisitos mínimos con los que se puede jugar y tener una experiencia aceptable. Para este trabajo no es posible y todas las pruebas se han realizado en un único ordenador con las siguientes características:

- Sistema operativo: Windows 11 Pro 64-bit (10.0, Build 22621) (22621.ni_release.220506-1250)
- Placa base: X570 Pro4
- CPU: AMD Ryzen 7 3700X 8-Core Processor (16 CPUs), ~3.6GHz
- GPU: AMD Radeon RX 6800 XT (32683 MB)
- Memoria: 32768MB RAM
- DirectX Version: DirectX 12

5.1.1 Pruebas en el editor

Unity recomienda realizar profiling directamente en las builds, ya que el editor puede afectar al rendimiento del juego por compartir proceso con el juego. Sin embargo, el profiler proporciona también información sobre los recursos que está utilizando el editor para no confundir una bajada de rendimiento causada por este con una mala optimización del juego.

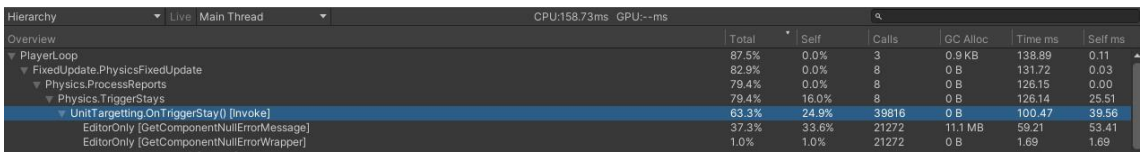


Figura 24 Captura del profiler de Unity durante la primera prueba

En la primera ejecución del juego utilizando el profiler, conforme avanzaba la partida el rendimiento iba reduciéndose cada vez más, hasta que llegó un punto en el que era injugable. Como se puede ver en la *Figura 24*, el causante era la función *UnitTargetting.OnTriggerStay*. Esta función es parte de la clase *MonoBehaviour* de Unity y se llama como parte de las actualizaciones del sistema de físicas de Unity que ocurren 50 veces por segundo por defecto. Además, la función se llama en cada actualización de física una vez por cada collider que esté en contacto con el trigger que esté invocando la función, y como se puede ver en la *figura*, se estaban realizando 39816 llamadas a la función por frame y estaban ocupando un 63,3% del tiempo de la CPU y causando que se necesitasen 158.73ms para computar cada frame, provocando que el juego estuviese funcionando a 6 fps. Al inspeccionar la llamada podemos ver que la función devolvía un error que pone *EditorOnly*, que efectivamente, ejecutando el juego en una build este problema no ocurría. Sin embargo, esto no significa que la función no pudiese provocar problemas de rendimiento también en las builds, ya que 20 unidades en rango unas de otras provocarían 20.000 llamadas por segundo a la función y rutinariamente, las partidas contienen bastante más que 20 unidades, por lo que se optó por reimplementar la funcionalidad que utilizaba la función *OnTriggerStay*. Las unidades pueden 'ver' en un radio a su alrededor y si una unidad enemiga se acerca demasiado, la atacan automáticamente. Esta es la funcionalidad que se cambió. Para reemplazarla se creó una corutina *VisionUpdate* que utiliza *Physics.OverlapSphere* que devuelve, similarmente a *OnTriggerStay*, los colliders en contacto con la esfera que genera la función, pero permitiendo filtrar utilizando las layerMask de Unity para ignorar los colliders que no sean unidades. Además, sólo siendo llamada cuando se ejecuta la corutina por lo que se puede modificar la frecuencia con la que se ejecuta la función. Al ejecutar cada unidad su propia corutina independientemente de las demás en lugar de ejecutarse todas a la vez, se evita que puedan producirse tantas llamadas simultaneas como estaba ocurriendo.

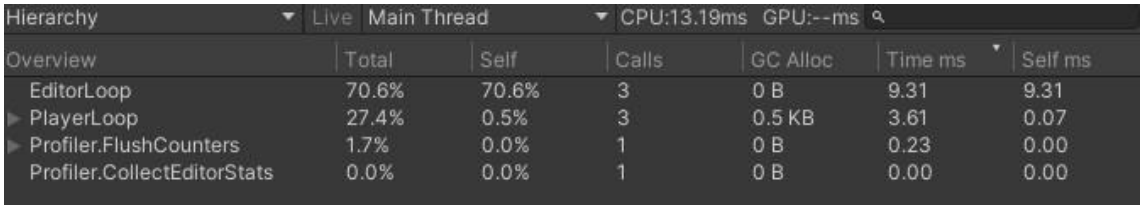


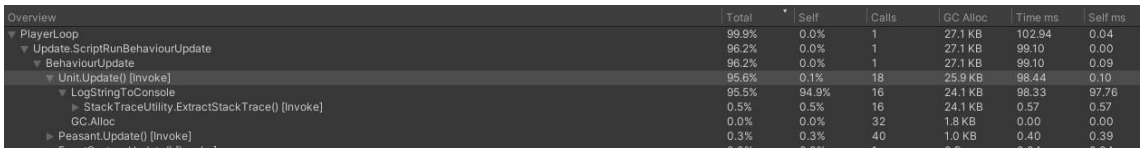
Figura 25 Captura del profiler tras cambiar la función

Como se observa en la *Figura 25* que muestra los resultados de ejecutar el profiler tras cambiar la función, podemos ver que el *PlayerLoop*, que contiene las llamadas a la actualización de físicas, está utilizando el 27,4% del tiempo de CPU, que equivale a 3.61ms y es el *EditorLoop*, el proceso del editor, el que está consumiendo la mayoría de recursos y aun así tenemos un tiempo por frame de 13.19ms, que equivale a 75 fps.

Una vez solucionado este problema se creó una build que se analizó con el editor.

5.1.2 Pruebas en la build

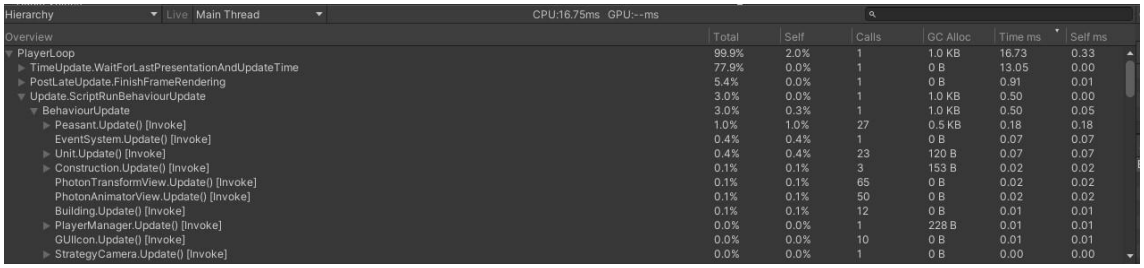
Las pruebas en la build revelaron un problema distinto de rendimiento



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	99.9%	0.0%	1	27.1 KB	102.94	0.04
Update.ScriptRunBehaviourUpdate	96.2%	0.0%	1	27.1 KB	99.10	0.00
BehaviourUpdate	96.2%	0.0%	1	27.1 KB	99.10	0.09
Unit.Update() [Invoke]	95.6%	0.1%	18	25.9 KB	98.44	0.10
LogStringToConsole	95.5%	94.9%	16	24.1 KB	98.33	97.76
StackTraceUtility.ExtractStackTrace() [Invoke]	0.5%	0.5%	16	24.1 KB	0.57	0.57
GC.Alloc	0.0%	0.0%	32	1.8 KB	0.00	0.00
Peasant.Update() [Invoke]	0.3%	0.3%	40	1.0 KB	0.40	0.39

Figura 26 Captura del profiler en una build

Como se puede ver en la *Figura 26* el problema lo provocó un debug statement en el bucle *Update* de las unidades. La solución fue simplemente eliminarlo, ya que ya no era necesario.



Hierarchy | Live | Main Thread | CPU:16.75ms GPU:--ms

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	99.9%	2.0%	1	1.0 KB	16.73	0.33
TimeUpdate.WaitForLastPresentationAndUpdateTime	77.9%	0.0%	1	0 B	13.05	0.00
PostLateUpdate.FinishFrameRendering	5.4%	0.0%	1	0 B	0.91	0.01
Update.ScriptRunBehaviourUpdate	3.0%	0.0%	1	1.0 KB	0.50	0.00
BehaviourUpdate	3.0%	0.3%	1	1.0 KB	0.50	0.05
Peasant.Update() [Invoke]	1.0%	1.0%	27	0.5 KB	0.18	0.18
EventSystem.Update() [Invoke]	0.4%	0.4%	1	0 B	0.07	0.07
Unit.Update() [Invoke]	0.4%	0.4%	23	120 B	0.07	0.07
Construction.Update() [Invoke]	0.1%	0.1%	3	153 B	0.02	0.02
PhotonTransformView.Update() [Invoke]	0.1%	0.1%	65	0 B	0.02	0.02
PhotonAnimatorView.Update() [Invoke]	0.1%	0.1%	50	0 B	0.02	0.02
Building.Update() [Invoke]	0.1%	0.1%	12	0 B	0.01	0.01
PlayerManager.Update() [Invoke]	0.0%	0.0%	1	228 B	0.01	0.01
GUIIcon.Update() [Invoke]	0.0%	0.0%	10	0 B	0.01	0.01
StrategyCamera.Update() [Invoke]	0.0%	0.0%	1	0 B	0.00	0.00

Figura 27 Captura del profiler tras eliminar el debug statement

En la *Figura 27* se puede ver cómo el problema ha desaparecido tras eliminar el debug statement.

Los resultados del análisis con el profiler han sido sorprendentes ya que no ha habido problema alguno ni con las máquinas de estado ni con la IA. Ambos han tenido un muy buen rendimiento incluso en partidas con muchas unidades. En la *Figura 27*, en el momento de tomar la captura, se puede ver que había 27 campesinos y 23 unidades que utilizan 0.25ms de tiempo de CPU, lo que representa un muy buen rendimiento. Por otra parte, el script de la IA está tan abajo en la traza del profiler que ni se muestra en la imagen. La posible causa de estos buenos resultados es que debido a experiencia previa desarrollando bots de web scraping, tenía ya conocimientos sobre la importancia de limitar la cantidad de acciones que pueden tomar agentes que actúan automáticamente y se tuvo ya en cuenta desde la fase de diseño.

Figura 28 Situación de una partida con muchas unidades

5.2 Análisis de la encuesta y Unity Analytics

Para crear la encuesta se han utilizado las preguntas tipo de *Fundamental Components of the Gameplay Experience: Analysing Immersion* [11]. Se han escogido las preguntas que se han considerado que encajaban con el trabajo y se ha añadido una categoría extra de preguntas relativas a las expectativas de los jugadores con respecto al sistema de IA del videojuego.

5.2.1 Analytics

Además de las Unity Analytics podemos utilizar las analytics de itch.io para obtener una visión más completa del alcance del juego. El juego ha estado disponible para su descarga del 6 a 8 de septiembre, en este intervalo de tiempo la página ha recibido 48 visitas y el juego 17 descargas como se muestra en la *Figura 29*.

Figura 29 Analytics de itch.io

A continuación, en Unity podemos ver que ha habido un total de 15 jugadores, lo cual es correcto para el número de descargas ya que dos de esas 17 fueron para comprobar que el juego se había subido correctamente a la página.

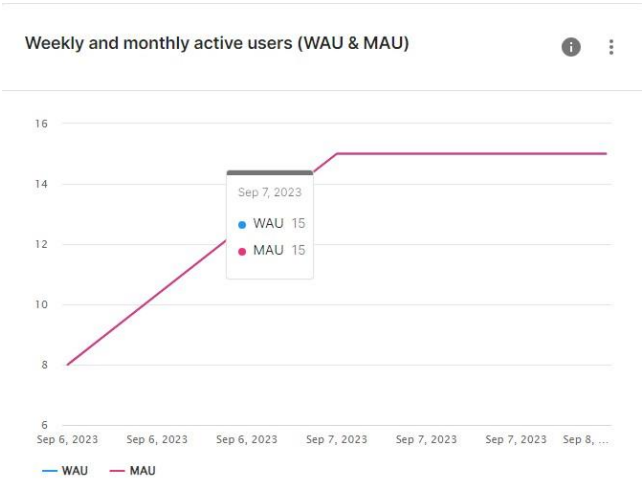


Figura 30 Número de jugadores de Castle Strike

Finalmente, en la *Figura 31* podemos observar la duración media de las partidas.

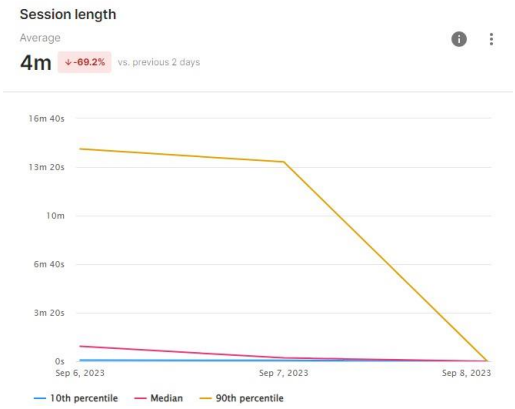


Figura 31 Duración media de las partidas

La duración media de las partidas es de alrededor de cuatro minutos. Esto se explica porque, juzgando por las partidas observadas, en las primeras partidas de los jugadores, mientras aprendían a jugar, eran derrotados rápidamente por la IA. Tras un par de partidas de aprendizaje eran capaces de resistir más y ser capaces de ganar la partida, resultando en varias partidas cortas por jugador y una partida final más larga.

5.2.2 Encuesta

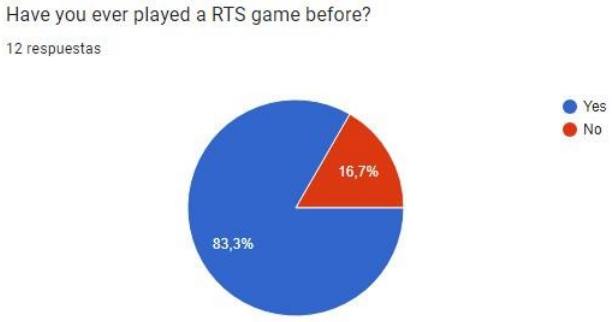
La encuesta consta de 19 preguntas agrupadas por secciones

En primer lugar, dos preguntas introductorias para conocer la experiencia de los jugadores con los videojuegos RTS. La siguiente sección contiene preguntas sobre usabilidad, la tercera sobre disfrute del juego, y a continuación, se encuentran las secciones de gratificación personal, estética visual y comportamiento de la IA.

Las respuestas a las preguntas usan la Escala de Likert están acotadas entre 1 y 5 dónde uno es completamente en desacuerdo y 5 completamente de acuerdo, y los encuestados deben responder según cuánto se identifiquen con la afirmación de la pregunta.

Introducción

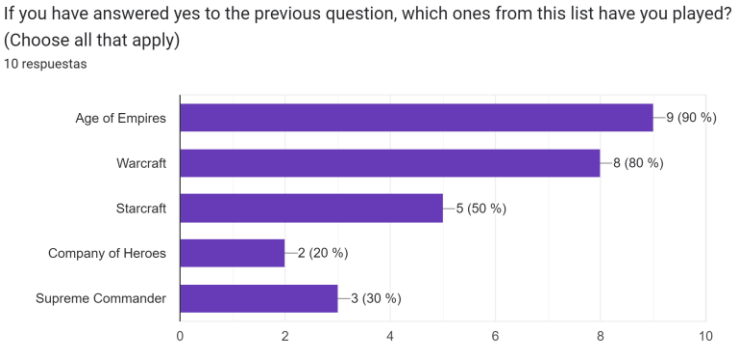
Pregunta 1: ¿Habías jugado a un juego RTS antes?



De los doce encuestados para dos era su primera experiencia con videojuegos RTS. Es importante contar tanto con jugadores con experiencia como sin experiencia para poder conocer las expectativas de ambos grupos de jugadores.

Pregunta 2: Si has respondido que sí a la pregunta anterior ¿Qué juegos de esta lista has jugado?

En esta pregunta se proponen distintas sagas famosas de videojuegos RTS para poder conocer las expectativas del grupo de jugadores con experiencia.

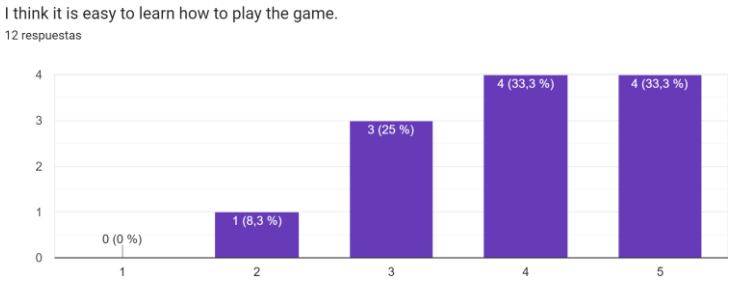


Las respuestas son las esperadas y podemos ver como la mayoría de encuestados han jugado a *Age of Empires* y *Warcraft* que son dos de las sagas de videojuegos RTS más exitosas mientras que el resto de opciones tienen menos respuestas, relativamente acordes a la popularidad de las sagas.

Usabilidad

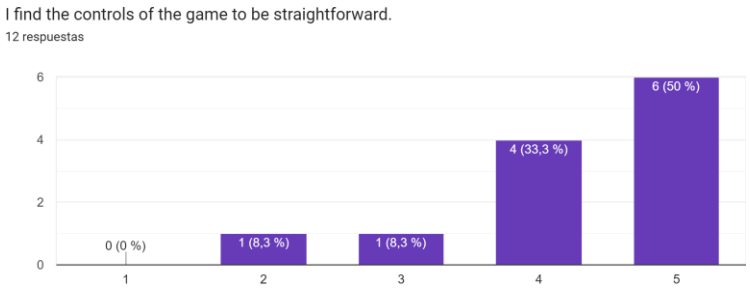
Las preguntas de usabilidad son relativas a la facilidad de uso de la interfaz, los controles y la información que el jugador recibe y que necesita para tomar decisiones dentro del juego.

Pregunta 3: Creo que es fácil de aprender a jugar al juego



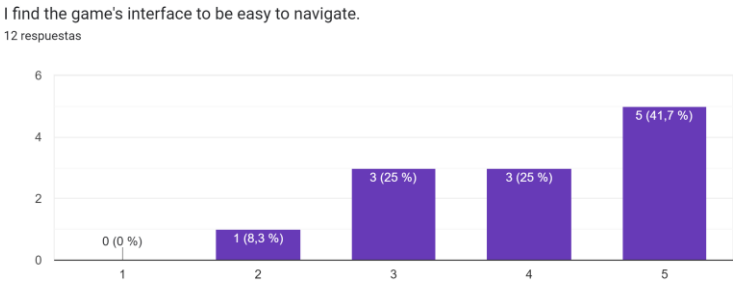
La mayoría de las respuestas son positivas, lo cual indica que el juego expone de manera clara y precisa lo que los jugadores pueden hacer y cómo pueden hacerlo.

Pregunta 4: Los controles del juego son directos



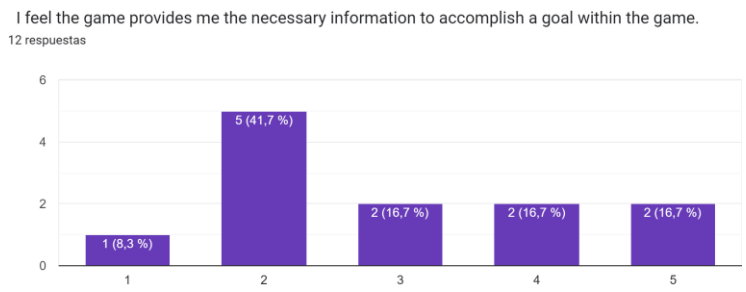
La mayoría de las respuestas a esta pregunta son también positivas y en conjunción con la pregunta anterior nos permiten corroborar que los controles son adecuados para un videojuego RTS. Las dos respuestas más negativas, probablemente, pueden atribuirse a los jugadores sin experiencia en videojuegos RTS por no tener costumbre del tipo de controles de este tipo de juego.

Pregunta 5: La interfaz del juego es fácil de navegar



La mayoría de las respuestas son positivas por lo que se puede concluir que la interfaz del juego está bien diseñada y los jugadores no tienen problemas a la hora de hacer lo que quieren dentro del juego.

Pregunta 6: El juego me provee la información necesaria para conseguir objetivos dentro del juego

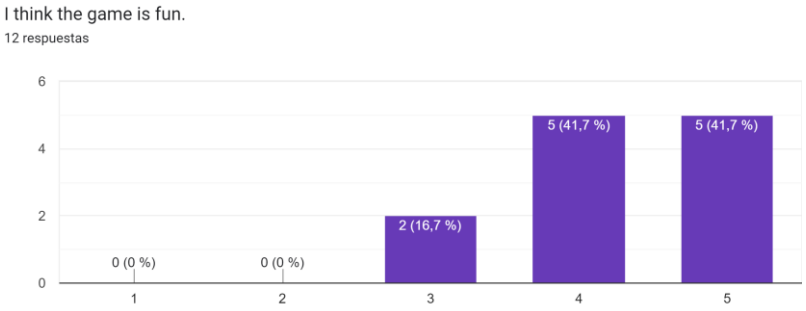


En esta pregunta, aunque las respuestas están repartidas la mayoría son negativas. Esto significa que a pesar de que la interfaz es fácil de navegar no provee suficiente información que los jugadores esperan en los videojuegos RTS.

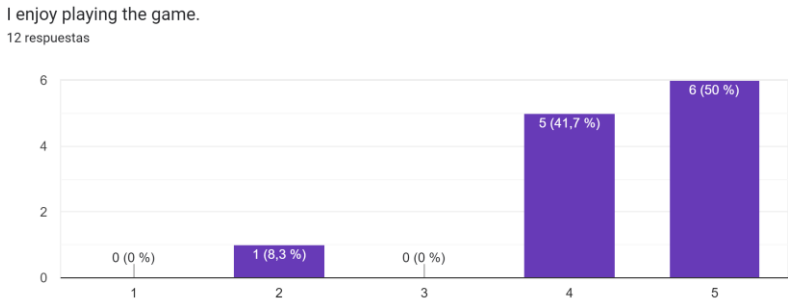
Divertimiento

Esta sección es relativa al disfrute del juego obtenido por los jugadores.

Pregunta 7: Creo que el juego es divertido

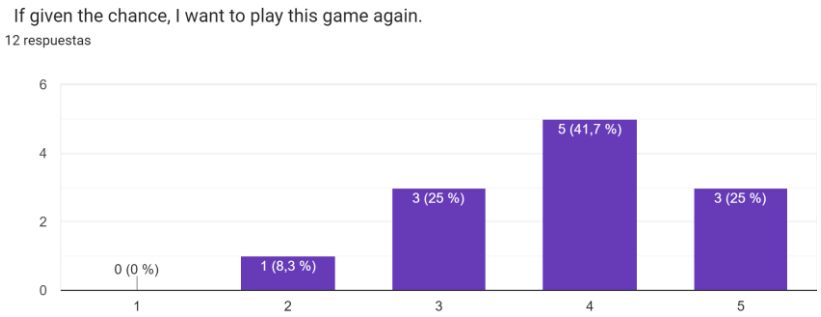


Pregunta 8: Me divierto jugando al juego



La séptima y octava preguntas están relacionadas y podemos ver que los encuestados consideran que el juego es divertido y se han divertido jugando, por lo que se puede concluir que el juego es divertido.

Pregunta 9: Dada la oportunidad, quiero volver a jugar al juego

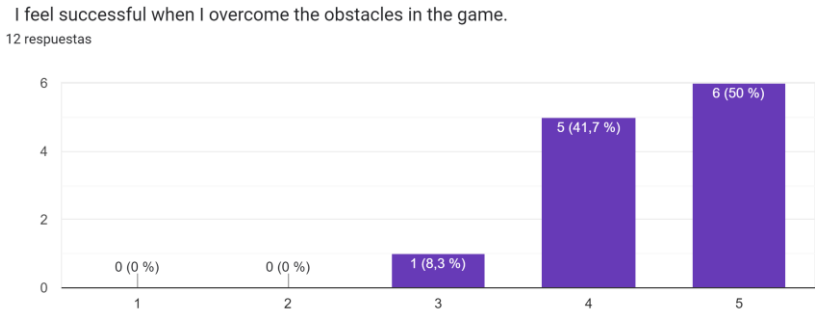


La mayoría de respuestas son positivas, lo cual da pie a la posibilidad de trabajos futuros que amplíen y completen el proyecto para crear una experiencia completa.

Gratificación personal

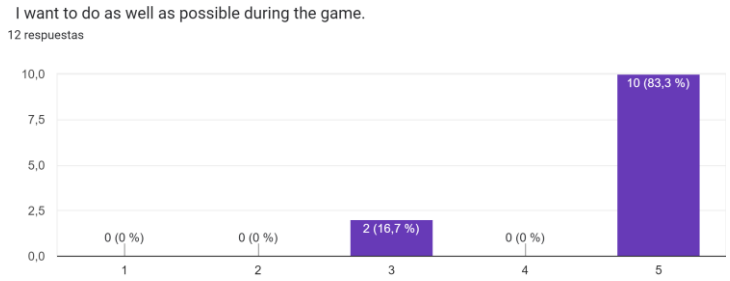
Estas preguntas son para conocer el grado de realización que han sentido los jugadores en el juego.

Pregunta 10: Me siento exitoso cuando supero obstáculos dentro del juego

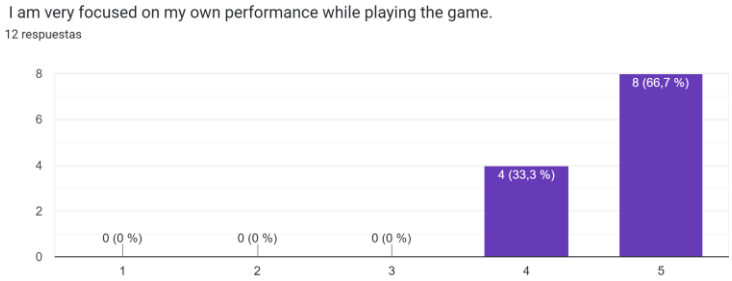


Los resultados de la pregunta son positivos, oscilando entre el 4 y el 5, por lo que se puede concluir que, aunque simple, el objetivo del juego y la IA proporcionan un reto que es capaz de motivar a los jugadores.

Pregunta 11: Quiero hacerlo lo mejor posible mientras juego



Pregunta 12: Estoy muy centrado en mi propio rendimiento mientras juego

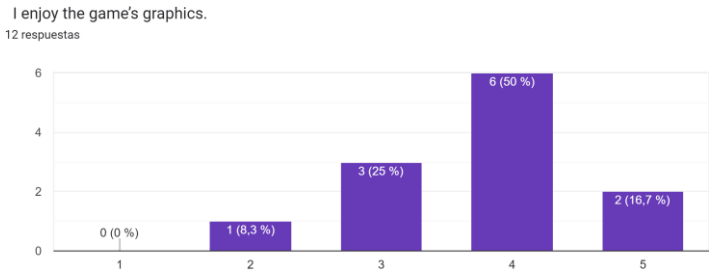


Viendo los resultados de estas dos preguntas juntos una conclusión que se puede extraer es que el juego es capaz de sumergir a los jugadores para se esfuercen en hacerlo lo mejor posible, además de hacerles conscientes de lo que hacen bien o mal.

Estética visual

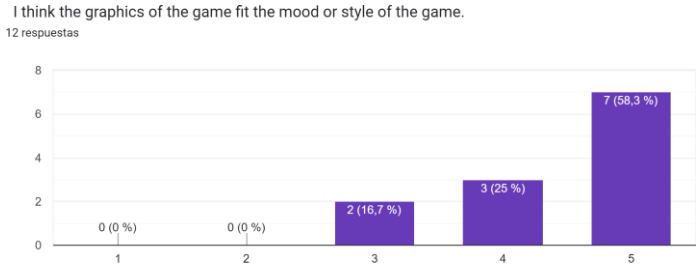
En este aparto se pregunta a los encuestados sobre su percepción de la estética y gráficos del juego.

Pregunta 13: Disfruto de los gráficos del juego



A pesar de no ser el objetivo del trabajo y por ello no estar trabajado el apartado visual del juego, a los jugadores les agrada generalmente el aspecto visual del juego.

Pregunta 14: Creo que los gráficos del juego encajan con el ambiente o estilo del juego

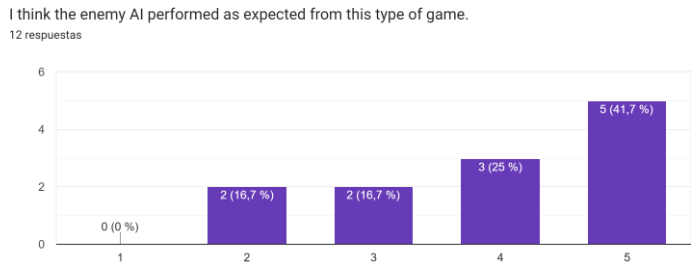


Lo que esta pregunta quiere significar es si el estilo gráfico es acorde al tipo de juego que se presenta, no sería lógico por ejemplo que un juego de miedo o de terror tuviera gráficos de dibujos animados, por lo que podemos concluir que a los jugadores les parece que encaja el estilo visual del juego con su jugabilidad.

Rendimiento y comportamiento de la IA

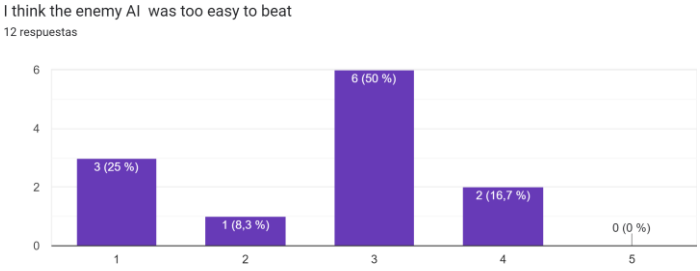
Esta sección no es parte de *Fundamental Components of the Gameplay Experience: Analysing Immersion*, de donde se han extraído las preguntas para la encuesta. Dado que el objetivo del proyecto era desarrollar una inteligencia artificial se ha considerado relevante preguntar a los encuestados sobre la misma.

Pregunta 15: Creo que la IA ha funcionado como se espera de este tipo de juego

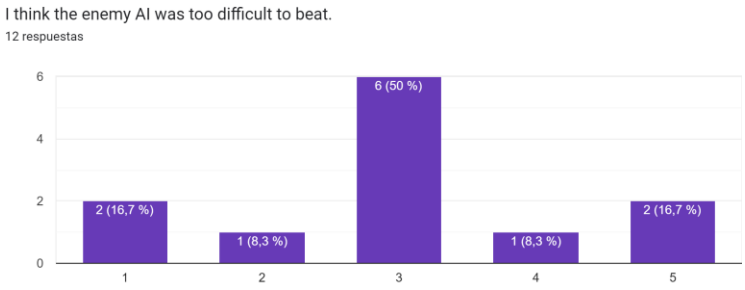


Aunque con algo de dispersión, la mayoría de respuestas son positivas por lo que se puede considerar que la IA rendido adecuadamente

Pregunta 16: Creo que la IA ha sido demasiado fácil de derrotar

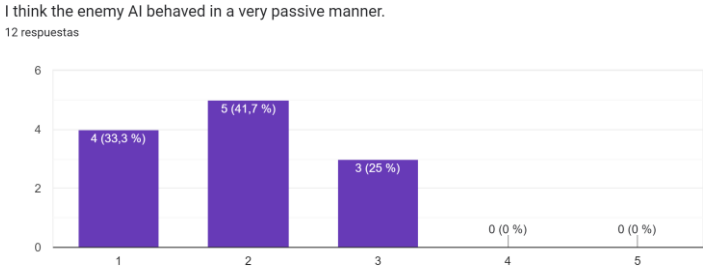


Pregunta 17: Creo que la IA ha sido demasiado difícil de derrotar

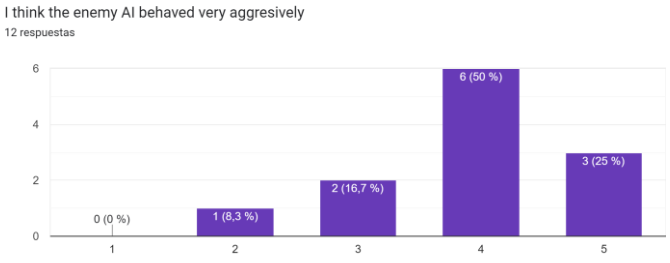


El objetivo de estas preguntas es determinar la dificultad del juego, aunque parezca que se contradicen, que un jugador no encuentre el juego fácil no significa que lo considere difícil y viceversa, por lo que la mayoría de respuestas se encuentren centradas es un buen resultado. Significa que la dificultad de la IA es más o menos la adecuada, aunque podría ajustarse mejor su comportamiento viendo la dispersión de las respuestas.

Pregunta 18: Creo que la IA se ha comportado de una forma muy pasiva



Pregunta 19: Creo que la IA se ha comportado muy agresivamente



En el apartado 3 se explicó que la función de utilidad de la categoría de ataque estaba pensada para que la IA no tuviese un comportamiento excesivamente cauto y juzgando las respuestas de estas dos últimas preguntas se puede considerar que el comportamiento de la IA es el deseado.

6. Conclusiones

Los resultados del proyecto son, sin duda, satisfactorios y se han alcanzado los objetivos propuestos y a parte se ha realizado también uno de los trabajos futuros mencionados en el trabajo original que era desarrollar un modo de un jugador para el juego.

Gracias a este trabajo se han reforzado los conocimientos de diseño de software adquiridos durante la carrera y el máster, y también ha permitido poner en práctica conocimientos adquiridos en las asignaturas de Ingeniería de software experimental por la parte investigadora del trabajo para encontrar la técnica de IA que mejor se adapta al videojuego.

6.1 Relación del trabajo con los estudios cursados

En el trabajo se han podido aplicar conocimientos adquiridos en el máster y además ha permitido reforzar conceptos estudiados durante la carrera.

En primer lugar, a la hora de investigar las distintas técnicas de IA para decidir cuál aplicar, lo aprendido sobre ingeniería de software experimental ha permitido encontrar y buscar información de mayor calidad. Se han empleado técnicas de medición de costes y profiling, estudiadas en el máster y que han permitido encontrar y solucionar problemas de la aplicación. También cabe destacar el desarrollo del proyecto del coche autónomo que ha ayudado en gran medida a entender el funcionamiento de la técnica de la utility AI. Durante el desarrollo de este trabajo se han ido encontrando similitudes entre ambos sistemas y en mi opinión son bastante más parecidos de lo que podría parecer a simple vista. Resumiendo, ambos son sistemas que toman información del ambiente y de si mismos, y en función de ciertos parámetros de funcionamiento y de esta información toman decisiones sobre cómo actuar.

También se han reforzado distintos conceptos del diseño de software aplicando tanto conocimientos adquiridos en el grado como los conocimientos de modelado conceptual del máster que han permitido rediseñar la arquitectura del juego para incluir los sistemas de IA, preservando toda la funcionalidad ya existente y permitiendo reutilizar el código desarrollado para el multijugador con modificaciones mínimas.

Este proyecto ha continuado construyendo sobre los conocimientos de Unity adquiridos en la carrera y en el desarrollo del trabajo previo a este. Se ha aprendido a utilizar herramientas avanzadas de Unity, como por ejemplo el profiler, y a aplicar técnicas de análisis de rendimiento en Unity. Se han aprendido técnicas para poder implementar acciones periódicas o que requieren de mucho tiempo computacional eficientemente como lo son las corutinas, o utilizar la clase *Time* de Unity para crear funciones que se ejecutan cada una cantidad de frames determinada.

Finalmente, gracias a la realización del proyecto se ha aprendido sobre distintas técnicas de IA y su funcionamiento, ventajas y desventajas y distintos casos de uso en el ámbito de los videojuegos en los que pueden aplicarse. Particularmente, se ha adquirido un gran conocimiento sobre las máquinas de estados finitos y la utility AI, que sin duda serán útiles en futuros proyectos.

6.2 Trabajos futuros

El proyecto actualmente cuenta con modo de un jugador y multijugador. Sin embargo, no es un producto completo, existen muchas mejoras y ampliaciones posibles. A continuación, se enumeran algunas:

- Mejorar la UI con la información obtenida de la encuesta
- Mejorar la IA, mejorar su funcionamiento en algunas acciones como ataque y construcción, haciéndolas más complejas
- Añadir música y sonidos
- Añadir distintos mapas con distintos modos de juego
- Aumentar el número de jugadores por partida

En términos generales todas estas mejoras permitirían llegar a obtener un proyecto completo. Tengo los assets y conocimientos necesarios para poder hacerlo y planeo hacerlo poco a poco a lo largo del tiempo. Una posible consideración sería empezar el proyecto de cero aplicando desde el principio todos los conocimientos aprendidos durante el desarrollo del TFG y del TFM para poder rediseñar y mejorar funcionalidades del juego.

Un videojuego al final es un tipo de software cuyo objetivo principal es entretener al usuario y ser divertido, o provocar distintas reacciones emocionales. Podrían considerarse una forma de arte, puesto que combinan muchas disciplinas artísticas como el dibujo, la animación y la música con la tecnología. Además, son tanto social como económicamente muy importantes en todo el mundo. Hoy en día se ve desde a futbolistas famosos jugando a videojuegos tras partidos de fútbol como a pilotos de F1 emitiendo en Twitch y creando sus propios equipos de e-sports. Los videojuegos se han convertido en la última década en un fenómeno global con un gran impacto en la sociedad. No hay que olvidar también otro tema muy importante como es el de la accesibilidad. Desarrollar productos, ya sea software o cualquier otro producto, y en este área los videojuegos son sin ninguna duda pioneros e innovadores como por ejemplo el videojuego Forza Motorsport que es un videojuego de carreras que cuenta con un sistema de asistencia para permitir a personas ciegas jugar [12]. Este impulso por desarrollar nuevos y mejores métodos de accesibilidad es en mi opinión una gran iniciativa que debería de replicarse en muchas otras industrias.

Para concluir, este trabajo ha aumentado mis conocimientos sobre la informática en diversas áreas como el diseño, la programación y la verificación de software, todas vitales para desarrollar buen software y ha reforzado mi convicción de que convertirme en ingeniero de software fue la decisión correcta. Además, quiero seguir construyendo sobre la experiencia acumulada en estos dos trabajos y aumentar mis conocimientos de diseño y desarrollo de videojuegos y aprender sobre otras áreas que desconozco como la animación o los efectos VFX.

7. Referencias

[1] GRAHAM David. *A reusable, light-weight finite-state machine* [en línea] [fecha de consulta 1 agosto de 2023]. Disponible en: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter12_A_Reusable_Light-Weight_Finite-State_Machine.pdf

[2] BUCKLAND Mat. *Programming game AI by example*. 1ª ed. Summit Avenue, Texas, 2005, pp. 43-44. ISBN 978-1-55622-078-4

[3] RASMUSSEN Jakob. *Are behavior trees a thing of the past?* [en línea] [fecha de consulta 15 agosto de 2023]. Disponible en: https://www.gamedeveloper.com/programming/are-behavior-trees-a-thing-of-the-past-#close-modal

[4] *An Introduction to GOAP* [en línea] [fecha de consulta 10 julio de 2023]. Disponible en: https://learn.unity.com/tutorial/an-introduction-to-goap?uv=2021.3&projectId=5e0bc1a5edbc2a035d136397#

[5] ORKIN Jeff. *Goal-Oriented Action Planning (GOAP)* [en línea] [fecha de consulta 14 agosto de 2023]. Disponible en: https://alumni.media.mit.edu/~jorkin/goap.html

[6] GRAHAM David. *An Introduction to Utility Theory* [en línea] [fecha de consulta 14 agosto de 2023]. Disponible en: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf

[7] BANSAL Vasu. *Utility Theory in Artificial Intelligence* [en línea] [fecha de consulta 14 agosto de 2023]. Disponible en: https://www.scaler.com/topics/utility-theory-in-artificial-intelligence/

[8] BREWER Daniel. *Knowledge is Power, an Overview of AI Knowledge Representation in Games* [en línea] [fecha de consulta 16 agosto de 2023]. Disponible en: http://www.gameaipro.com/GameAIProOnlineEdition2021/GameAIProOnlineEdition2021_Chapter04_Knowledge_is_Power_an_Overview_of_AI_Knowledge_Representation_in_Games.pdf

[9] *What is a behaviour tree* [en línea] [fecha de consulta 16 agosto de 2023]. Disponible en: https://opsive.com/support/documentation/behavior-designer/what-is-a-behavior-tree/

[10] BUCKLAND Mat. *Programming game AI by example*. 1ª ed. Summit Avenue, Texas, 2005, pp. 47. ISBN 978-1-55622-078-4

[11] ERMI Laura, MÄYRÄ Frans. *Fundamental Components of the Gameplay Experience: Analysing Immersion* [en línea] [fecha de consulta 6 septiembre de 2023]. Disponible en: https://www.researchgate.net/publication/221217389_Fundamental_Components_of_the_Gameplay_Experience_Analysing_Immersion

[12] *Forza Motorsport – Blind Driving Assists* [en línea] [fecha de consulta 6 septiembre de 2023]. Disponible en: https://youtu.be/T7aVUqmQ_Sc?si=83eGQSNPPyXoYAYg

8. Anexos

8.1 Game Design Document

Análisis del Juego

Castle Strike es un videojuego de estrategia 3D del género RTS en el cual se tiene que reclutar un ejército para derrotar al resto de jugadores enemigos.

El videojuego consistirá en un modo de un jugador uno contra uno contra la IA y de un modo multijugador de 2 a 4 jugadores que lucharán por equipos o individualmente con el objetivo de destruir la base enemiga.

Declaración de objetivos

Utiliza tu genio estratégico para sorprender y vencer a tus enemigos empleando tácticas y estrategias superiores a las suyas.

Género y extensión del juego

RTS, multijugador. El juego se compondrá de varios mapas donde jugar las partidas.

Plataformas

Microsoft Windows, macOS, Linux.

Público Objetivo

Jugadores a partir de 12 años de edad a los que les guste el género de la estrategia.

PEGI

El juego está orientado a una clasificación de PEGI 12 debido a que existen muestras de violencia de una naturaleza gráfica hacia los personajes, pero sin ser parecido a la vida real.

Jugabilidad

Resumen de la Jugabilidad

Se trata de un videojuego perteneciente al género de la estrategia en tiempo real y estará disponible únicamente para PC.

Castle Strike será un videojuego multijugador en el que la inteligencia y capacidad de decisión del jugador serán su principal herramienta.

Experiencia del Jugador

Empiezas en un menú minimalista desde el que se accede al menú multijugador donde se mostrará un listado de las partidas existentes o se podrá crear una nueva a la que podrán unirse otros jugadores, tras configurar una partida se empezará y cada jugador controlará su base y sus tropas.

Directrices de Jugabilidad

El juego se enfocará en la construcción de bases y el combate por destruir las del resto de jugadores por lo que se deberá diseñar el juego teniendo en cuenta el equilibrio entre las distintas tropas para que sea la habilidad de los jugadores la que les de la victoria.

El juego deberá incentivar la creatividad y la experimentación con distintas estrategias para permitir que no haya dos partidas iguales ni la existencia de una única estrategia ganadora.

Objetivos del Juego y Recompensas

Recompensas	*Obstáculos*	*Dificultad de los niveles*
Ganar partidas y subir de nivel de habilidad para enfrentarse a mejores jugadores.	El resto de los jugadores	La dificultad dependerá del propio nivel del jugador y de la habilidad de su contrincante ya que el que sea el mejor estratega vencerá.

Mecánicas del Juego

Atributos de las unidades	
Campesino	- Construir: los campesinos son la única unidad capaz de construir nuevos edificios para la base. - Recolección de recursos: oro, madera, piedra y comida son los recursos que los campesinos recolectarán. - Milicia: en caso de necesidad los campesinos pueden transformarse temporalmente en milicianos, más débiles que soldados normales, para luchar contra las tropas enemigas.
Espadachín	- Muro de escudo: el espadachín avanza con su escudo alzado, reduciendo su velocidad de movimiento y aumentando su defensa. -
Arquero	- Salva: los arqueros concentran sus disparos en una zona para crear una zona por donde resulta imposible avanzar sin sufrir graves bajas.
Caballero	- Carga: los caballeros aceleran a gran velocidad con sus monturas y embisten a los soldados enemigos infligiendo grandes daños.
Catapulta	- Asedio: las catapultas causan daño aumentado a las estructuras.

Batalla	Modo de juego único multijugador.
Conquista	Ganará el jugador que destruya el edificio principal del enemigo.
Victoria total	Ganará el jugador que destruya todos los edificios de la base enemiga.

Diseño de Niveles

El videojuego se compondrá de un conjunto de mapas de estilo RTS con localizaciones fijas y distintos puntos de recursos por los que los jugadores deberán competir. Los mapas tendrán distintos tamaños en función del número de jugadores de la partida.

Cámara

La cámara en Castle Strike será una cámara clásica de RTS de estilo isométrico con zoom para acercarla y alejarla.

Inteligencia Artificial

Las unidades contarán con una IA para pathfinding así como con una IA básica implementada mediante máquinas de estado para poder realizar acciones automáticas como la recolección de recursos, cuando se agote un nodo buscar otro del recurso al que estaba asignado el campesino, o atacar automáticamente a enemigos cercanos u ordenar defender una posición.

Para el modo de un jugador se desarrollará un sistema de dual utility AI que contará con las siguientes categorías de acciones y acciones:

Categorías

- Reclutar Campesino
- Asignar campesino a construcción
- Defenderse
- Acciones de recolección
- Acciones de construcción
- Acciones de reclutamiento
- Acciones de ataque

Acciones:

- Recoger recurso, una acción por tipo de recursos
- Construir edificio, una acción por tipo de edificio
- Reclutar unidad, una acción por tipo de unidad
- Atacar enemigo, una acción por tipo de edificio, una para atacar unidades enemigas y otra para atacar el ayuntamiento enemigo.

Las acciones de reclutar campesino, asignar campesino a construcción y defenderse no son categorías, pero no forman parte de ninguna categoría porque son acciones muy importantes como para que pudiera ocurrir que no se ejecuten cuando son necesarias porque su categoría no se ha evaluado con la máxima utilidad.

Esquema de controles

Acción	Teclado
Selección de unidades	Click izquierdo del ratón sobre la unidad
Selección múltiple de unidades	Mantener shift + click sobre las unidades o Mantener click izquierdo y arrastrar el ratón sobre las unidades
Movimiento	Click derecho sobre un punto válido del mapa
Movimiento múltiple	Mantener shift + Click derecho sobre los puntos a los que se quiera mover las unidades
Establecer un punto de reunión	Seleccionar un edificio de producción de unidades y hacer click derecho en un punto válido del mapa
Crear grupo de control	Con las unidades deseadas seleccionadas pulsar control + número
Menú	Escape

Estética del Juego e Interfaz de Usuario

Estética de Niveles

El juego será luminoso y tendrá una estética cartoon que encaje con el diseño de las unidades.

Estética de Personajes

Tanto las unidades como estructuras serán las contenidas en el Toony Tiny RTS set obtenido de la Unity Asset Store.

Representación general del paquete de Assets.

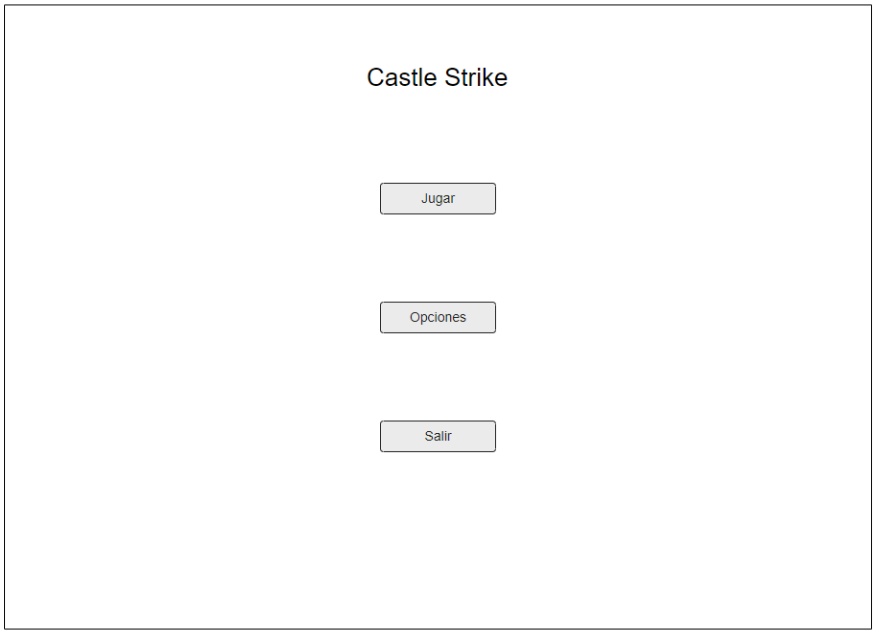
Diseño de audio

El juego cuenta con música diferente para los menús y para las partidas que ayuda a la inmersión y a caracterizar el juego, también hay efectos de sonido para las interacciones de los menús como en la transición entre estos como las interacciones con los elementos del menú de opciones, a su vez también hay efectos de sonido para las acciones de las unidades como talar árboles o construir. El volumen tanto de la música como de los efectos puede ser regulado, independientemente el uno del otro, con los sliders del menú de opciones.

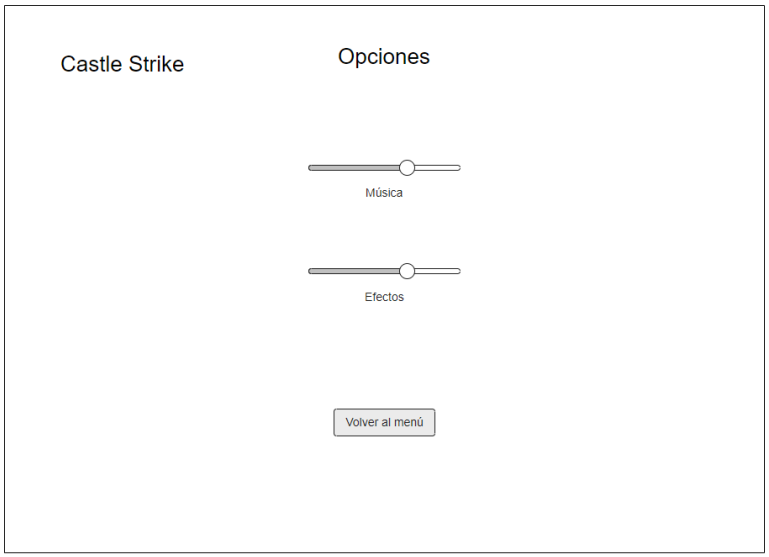
Interfaz de Usuario

Como solo hay un modo de juego, el menú principal permitirá al jugador ir al menú de jugar, cambiar las opciones (ajustes de sonido) o salir del juego.

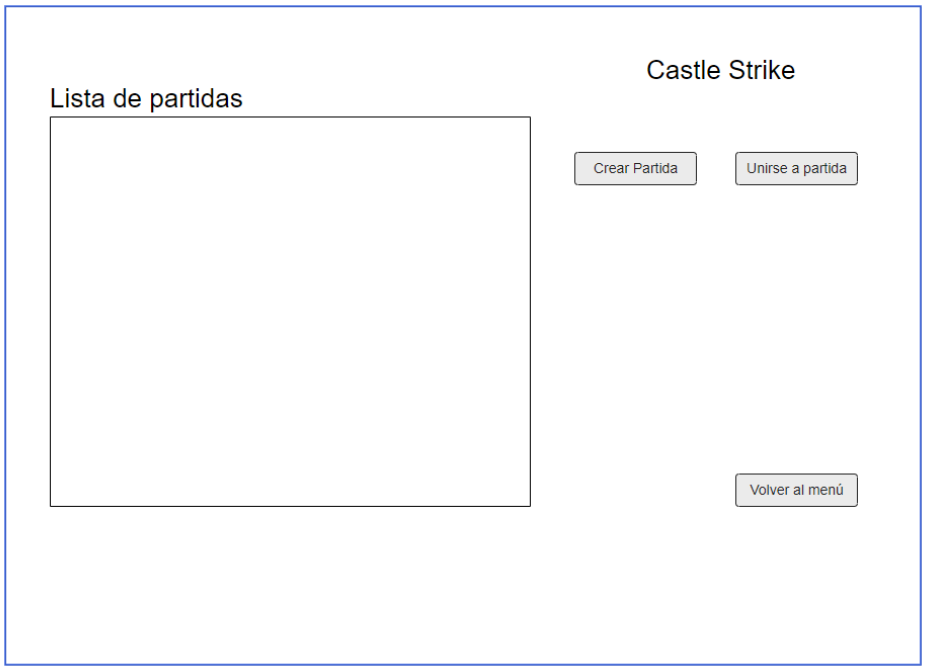
Respecto a la interfaz de la partida estará inspirada en la de juegos RTS clásicos como Warcraft 3 o Age of Empires, siendo similar a la de estos.



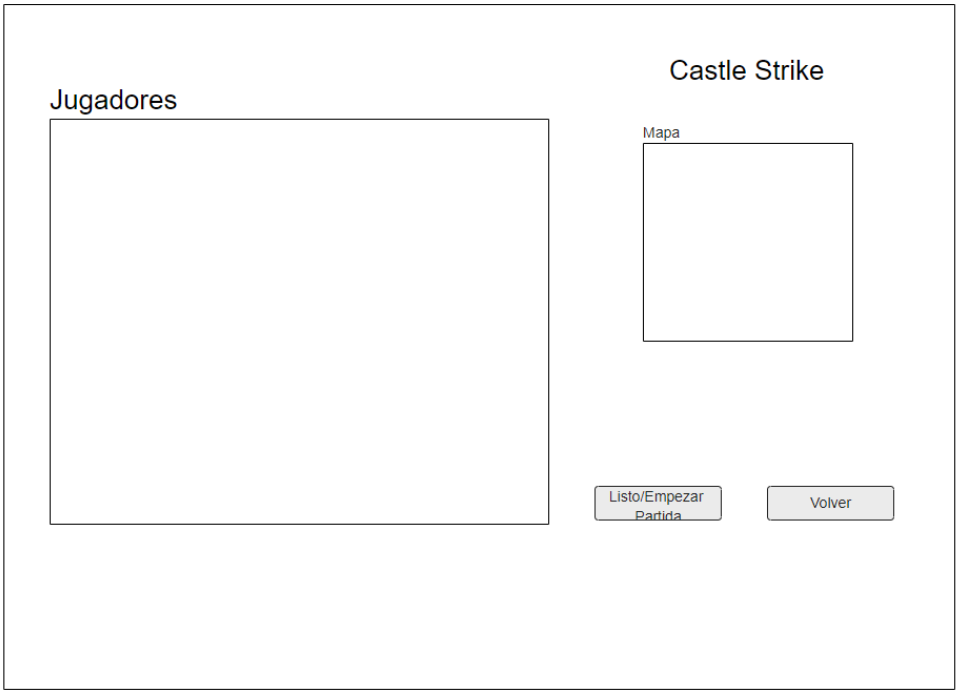
Mockup del menú principal



Mockup del menú de opciones



Mockup de la lista de selección de partida



Mockup del lobby de una partida

Interfaz del Warcraft 3.

Interfaz del Age of Empires 2.

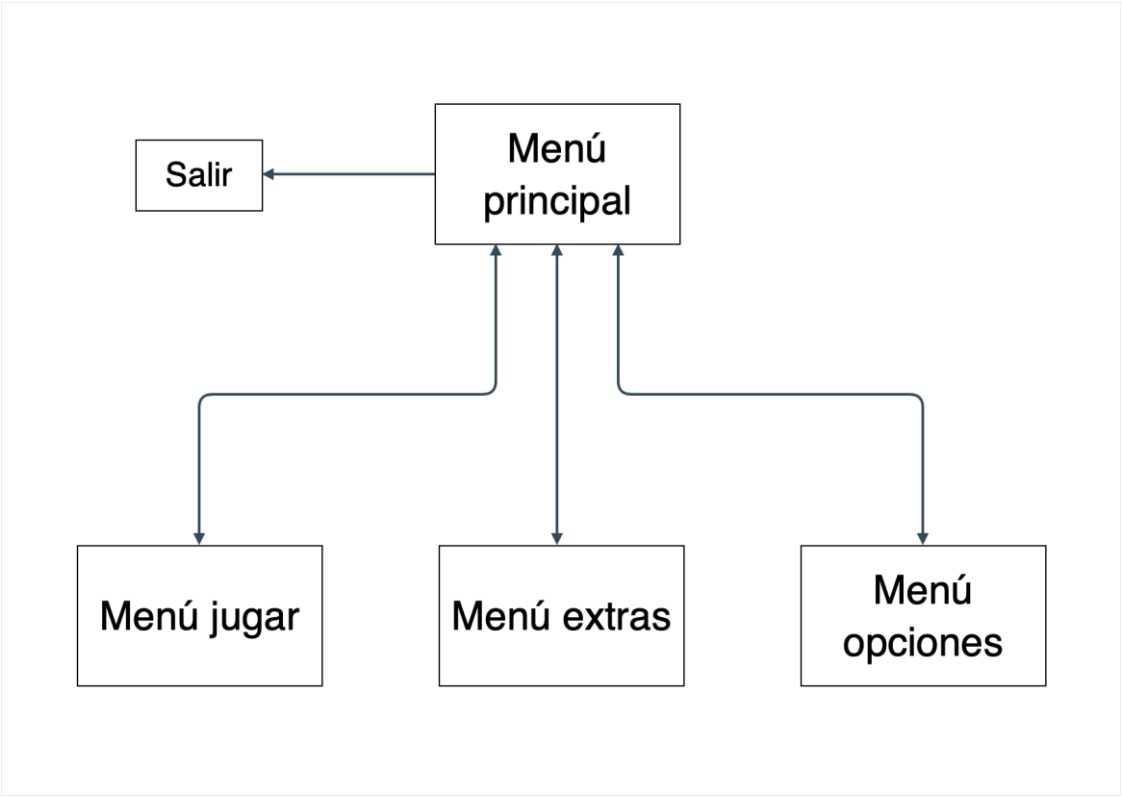


Diagrama de navegación entre menús.

8.2 Objetivos de desarrollo sostenible

8.2.1 Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS)

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**		X		
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**		X		
ODS 9. **Industria, innovación e infraestructuras.**				X
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

8.2.2 Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

La naturaleza de este trabajo no contempla la sostenibilidad como un objetivo a perseguir puesto que es un videojuego y, de cualquier forma, la infraestructura necesaria para la conservación y distribución del proyecto produce un impacto negativo en el medio ambiente ya que el consumo energético de los datacenters en los que se alojaría el proyecto del trabajo en caso de ser distribuido es muy elevado.

Sin embargo, el trabajo podría relacionarse con el ODS 8 y en menor medida con el ODS 5.

Como se ha explicado al principio de este trabajo el sector de los videojuegos está en constante crecimiento, proporcionando trabajo a cada vez más personas. Durante la crisis del COVID ha demostrado su robustez, al menos en España, ya que la mayoría de estudios de videojuegos mantuvieron su plantilla e incluso el número de personas empleadas en este sector se incrementó. En un contexto de crisis económica internacional el pronóstico en el sector de los videojuegos es de crecimiento económico y se prevé que el número de personas empleadas en el sector continúe creciendo. Por estos motivos podemos concluir que este trabajo se relaciona con el ODS 8.

Por otra parte, como es común en muchas ramas de la informática, es un sector en el que la mayoría de puestos están ocupados por hombres, sin embargo, cada vez más mujeres se adentran en el mundo del desarrollo de videojuegos. Alrededor del 18,5% de los empleos en el sector de los videojuegos en España están ocupados por mujeres y la mayoría de estudios afirman tener en marcha planes de igualdad para aumentar esta cifra. Desde el punto de vista del consumidor en el sector de los videojuegos hay una mayor igualdad, el 42% de *gamers* son mujeres lo cual es una noticia alentadora puesto que antiguamente los videojuegos se consideraban un entretenimiento principalmente masculino.

A pesar de haber desarrollado el trabajo sin considerar ODS dentro del sector de los videojuegos existen analistas y muchos estudios que los consideran a la hora de orientar sus procesos. Es por esto por lo que el trabajo ha podido relacionarse con estos dos objetivos.

Glosario

Componente: Unidad que dota de funcionalidad a un gameObject.

FPS: First Person Shooter, juego de disparos en primera persona en castellano.

Gameobject: Unidad básica para construir las escenas de Unity.

Los Sims: Serie de videojuegos de simulación social desarrollada por Electronic Arts

Navmesh: Estructura de datos que describe las superficies caminables del mundo del juego y permite encontrar el camino de una ubicación a otra.

p2p: Peer-to-peer o red de pares, es un tipo de red de ordenadores en la que los clientes se conectan directamente entre sí mismos.

PhotonView: Componente de PUN utilizado para sincronizar los gameObjects a través de la red.

Profiler: Herramienta de análisis de costes utilizada para medir el rendimiento de una aplicación

RPC: En computación distribuida, técnica utilizada para ejecutar código de forma remota en una máquina distinta a la que realiza la llamada

Tag: Referencia o etiqueta asignable a un gameObject para identificarlo durante la partida.

Utilitarismo: Corriente filosófica surgida en el siglo XVIII que establece que la mejor acción es la que produce el mayor beneficio para la mayoría de personas.