Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

Grado en Ingeniería Informática de Gestión y Sistemas de Información

TRABAJO FIN DE GRADO

Torres versus Robots: a Real Unreal Game

Estudiante: Mata Fernández, Iván

Directora: Atutxa Salazar, Aitziber

Codirectora: Aguilera Anguita, Marta

Repositorio GitHub: https://github.com/IvanMataFernandez/Tower_Defense/tree/master

Curso: 2023-2024 **Fecha**: Bilbao, 25 de julio de 2024

Resumen

Español

Se ha diseñado, implementado y testado un videojuego de defensa de torres haciendo uso de la lógica de la programación orientada a objetos (POO) y creado en el motor de juego *Unreal Engine 5*, una herramienta profesional innovadora concebida para el desarrollo de videojuegos.

Es un juego que se basa en el enfrentamiento de torretas que luchan por el jugador y robots que se oponen a estas, habiendo distintos tipos de personajes en cada bando. Se han diseñado una gran variedad de acciones con animaciones por cada uno, así como algoritmos de inteligencia artificial (IA) que toman decisiones activas sobre sus funciones en el juego.

Además, el videojuego posee una gran variedad de niveles en un entorno tridimensional virtual que toma aspecto de campo de batalla. El jugador sitúa sus construcciones aquí, con el propósito de defenderse de oleadas de enemigos a lo largo de una gran variedad de niveles distintos. Cada uno de estos presenta distintas combinaciones de tipos de oponentes, junto con el posible desbloqueo de una nueva torreta utilizable al completarlo.

El programa también dispone de un menú principal que se carga por defecto desde donde se permite modificar ajustes, ver las normas de partida, continuar el juego desde el nivel en el que se dejó y más.

Palabras clave: juego de defensa de torres (*Tower Defense*), motor de juego, programación orientada a objetos (POO).

English

A *Tower Defense* game has been designed, implemented and debugged making use of object oriented programming (OOP) and created in *Unreal Engine 5*, an innovative professional tool made for the development of video games.

This game is based in the confrontation of turrets that fight for the player and robots that are opposed to these, there being different kinds of characters for each side. A wide range of actions have been designed with animations for each one, as well as artificial intelligence (AI) algorithms that make active decisions about their functions in the game.

Furthermore, the video game includes lots of levels that take place in a battlefield shaped as a three-dimensional environment. The player places his/her towers here, with the goal of defending themself from waves of enemies in many different levels. Each of these provide different combinations of opponent types, alongside the possible unlock of a brand new tower when completed.

The program includes a main menu that is loaded by default from where settings can be altered, game rules may be viewed, progress in the game can be continued and more.

Keywords:, *Tower Defense* type game, game engine, object oriented programming (OOP).

Euskera

Dorreak defendatzeko bideojoko bat diseinatu, inplementatu eta probatu da, objektuetara bideratutako programazioaren logika erabiliz (POO) eta Unreal Engine 5 joko-motorrean sortuta. Bideojokoak garatzeko sortutako tresna profesional berritzailea da.

Joko hau roboten eta jokalariaren alde jokatzen duten dorretxoen arteko liskarrean oinarritzen da, bando bakoitzean pertsonaia mota ezberdinak daudelarik. Ekintza ugari diseinatu dira, eta horietako bakoitzak animazioak ditu, baita adimen artifizialeko algoritmoak ere, jokoan dituzten funtzioei buruzko erabaki aktiboak hartzen dituztenak.

Gainera, bideojokoak maila ugari ditu hiru dimentsioko ingurune birtual batean, eta gudu-zelaiaren itxura hartzen du. Jokalariak hemen kokatzen ditu bere eraikuntzak, maila ezberdin ugaritan zehar etsai oldeetatik defendatzeko asmoz. Horietako bakoitzak aurkari-moten konbinazio desberdinak ditu, eta hori osatzean erabil daitekeen dorretxo berri bat desblokeatzeko aukera dago.

Programak, gainera, lehenespenez kargatzen den menu nagusi bat ere badu. Menu horretatik, doikuntzak aldatzeko, abiapuntuko arauak ikusteko, jokoa utzi zen mailatik jarraitzeko eta gehiago egiteko aukera ematen da.

Gako-hitzak: dorrearen defentsa joko mota, joko motorra, objektuetara bideratutako programazioa (OBP).

Índice de contenidos

Índice de tablas

Índice de tablas exclusivas en el contenido principal

Índice de tablas en los anexos

Índice de figuras

Índice de figuras exclusivas en el contenido principal

Índice de figuras en los anexos

1. Introducción y contexto

En el Grado en Ingeniería Informática de Gestión y Sistemas de Información de la Escuela de Ingeniería de Bilbao [1] se ha aprendido a gestionar, diseñar e implementar programas (también conocidos como aplicaciones *software*); donde cada asignatura de la titulación se focaliza en un tipo de programa o fase de desarrollo en concreto. Las que han tenido un mayor impacto en el trabajo han sido las siguientes:

- **"Programación Básica" y "Programación Orientada a Objetos"**: Aprender los fundamentos de la implementación de una aplicación *software* mediante la programación orientada a objetos (POO).
- **"Estructuras de Datos y Algoritmos", "Ingeniería del Software", y "Análisis y Diseño de Sistemas de Información"**: Aprender a diseñar programas de forma correcta, para facilitar entendimiento y desarrollo de código.
- **"Gestión de Proyectos"**: Conocer cómo gestionar el ciclo de vida de un proyecto, de manera que se puedan desarrollar los programas de forma ordenada y adecuada.

Gracias a la carrera se ha dispuesto de la posibilidad de aprender los fundamentos de la programación y demostrar lo adquirido mediante el desarrollo de varios proyectos de complejidad moderada. Ahora, con esta oportunidad final, se ha decidido llevar lo aprendido un paso más adelante: desarrollar un videojuego completamente de cero que tome parte en un entorno tridimensional virtual. Esta es una idea bastante ambiciosa, debido a que la titulización no va más allá del desarrollo de juegos de mesa sencillos (como pueden ser las damas, el ajedrez, o hundir la flota) respecto a este ámbito y no se trabaja la parte gráfica.

Aitziber Atutxa (la directora del proyecto) aceptó la propuesta del trabajo e incluso facilitó el desarrollo de este gracias al establecimiento de contacto con Marta Aguilera (la codirectora del trabajo). Aguilera se dedica profesionalmente al desarrollo de juegos, resultando ser una guía muy útil. Entre sus sugerencias, se incluiría usar la herramienta *Unreal Engine 5* como base para el proyecto.

Unreal Engine es un motor de juego, es decir, un editor o programa diseñado específicamente para el desarrollo de videojuegos. Si bien es cierto que ofrece una gran variedad de utilidades para la creación de juegos, estas pueden resultar muy abrumadoras al principio. Se debe desarrollar el juego entero de cero en este entorno completamente desconocido.

Para poder hacer el juego final posible, se necesitaría un paso de formación extensa en el que se incluiría aprender a usar este editor innovador. Aguilera sugeriría superar la total falta de conocimiento en el entornos de programación de videojuegos enrolándome en distintos cursos para poder adquirir los conocimientos mínimos necesarios con los que dar los primeros pasos del proyecto, y así adquirir seguridad en un mundo totalmente nuevo y abrumador. En estos programas hay que generar un universo único, donde se deben tener en cuenta todos los aspectos que van desde las formas y comportamiento de los personajes hasta los gráficos y decoraciones que representan los escenarios donde se desarrolla la aventura y suena la música. Todo siempre de manera que el juego sea robusto y dinámico.

No obstante, los conocimientos adquiridos en la carrera también ayudarían al trabajo, principalmente en la forma en la que se debería gestionar el proceso de desarrollo y diseño del programa. La filosofía de diseño de programas mediante el uso de la programación orientada a objetos sería también aplicable a este proyecto, siendo este uno de los principios en los que se focaliza la formación del grado.

2. Conexión con los Objetivos de Desarrollo Sostenible y los principios y valores democráticos

Desarrollar un juego es como contar una historia, cualquier cosa es posible. Esto quiere decir que un videojuego es capaz de fomentar valores éticos, pero también promover desigualdades e injusticias. Ejemplo de esto último puede ser crear un videojuego con una narrativa que infravalora a la mujer, razas, u orientaciones sexuales, entre otros [2].

Siempre existe el riesgo de que un videojuego con características no éticas pueda llegar al mercado. Aunque no en mayoría, existen casos de esto (generalmente por error) y es responsabilidad de sus autores asegurar que nadie se pudiera sentir excluido al probar su producto.

Un ejemplo de falta de inclusividad ocurre con el videojuego *Tomodachi Life* (traducido al español como "vida de amigos"), en el que los usuarios pueden crear réplicas suyas, además de familiares y/o amigos en un mundo virtual. Estos avatares pueden socializar entre ellos de manera automática, incluso pudiendo llegar a casarse. El problema reside en este último concepto, ya que el juego nunca ha dado la opción de crear parejas homosexuales. Esto causó una gran exclusividad a una porción de sus jugadores en su día e impidió que pudieran sentirse identificados con sus propias creaciones [3].

Para evitar caer en este problema se pueden seguir algunas prácticas recomendadas. Una característica que puede ayudar a la inclusividad es incluir una gran diversidad de personajes con distintas personalidades y apariencias [2]. Otra posible estrategia es dar una gran catálogo de customización para avatares de usuarios. En el caso de que no convenga programar opciones de personalización, entonces se puede implicar ambigüedad en algunos rasgos de los personajes predefinidos para que sea el jugador quien se los pueda suponer.

El juego *Undertale* da un buen ejemplo de inclusividad, en el que nunca se especifica el género del protagonista y está a merced del jugador imaginárselo. Esto ayuda al usuario a sentirse más incluido en la historia, no estando forzado a jugar como un personaje de un género concreto [4].

Además cabe destacar que no solo la propia historia importante, algunos otros detalles como la traducción del juego a varios idiomas ayuda a la inclusividad. Esto se debe a que más público puede disfrutar del producto y sentirse incluido como objetivo demográfico.

Respecto al juego desarrollado en este proyecto se ha decidido no mostrar el personaje del jugador, para ayudar a respetar esta misma inclusividad y dar la opción al usuario imaginarse la raza del protagonista, sexo y demás. Los únicos detalles que se especifican son algunas de sus destrezas y localización donde vive para poder hacerlo encajar con la narrativa de la historia.

Además, por el momento se incluyen traducciones del juego al español e inglés para aumentar la cantidad de gente que puede jugar al juego sin dificultades de idioma. Si se tuviera un equipo de desarrollo, se podrían haber hecho incluso más traducciones a otros idiomas muy conocidos como el francés o el chino. Esta es una de las varias mejoras que se habían planteado para el proyecto tras su finalización y se encuentran en el apartado de conclusiones (10.4).

3. Definición de la idea y objetivos

3.1 Concepción del tipo de juego a crear

Se necesita primero tener una idea o un tipo de juego en mente antes de empezar a desarrollarlo. Existen muchos de estos, cada uno con sus inconvenientes a mayor o menor escala a la hora de implementarlos.

Se decidiría desarrollar un juego de defensa de torres (también comúnmente denominado *Tower Defense*), debido a su moderada complejidad de implementación. De esta forma, se puede equilibrar la falta de personal en el proyecto, ya que se trata de un trabajo individual.

3.2 Qué es un juego “Tower Defense”

Tower Defense [5] es una de las muchas categorías existentes de videojuegos. En este género, el objetivo del jugador es defenderse de enemigos que se aproximan hacia él o ella mediante el uso de torretas en un entorno virtual, también comúnmente denominado “mapa”.

El área de juego se caracteriza por dos zonas claves: un punto de aparición de enemigos y otro de captura. Los adversarios aparecen en esta primera zona y se desplazan hacia la segunda, siendo el objetivo del jugador obstaculizar e impedir que lleguen a su destino. En un mismo mapa se pueden alojar varios niveles, donde cada uno de estos se distingue por su grado de dificultad (la calidad y cantidad de adversarios a los que el usuario deberá enfrentarse).

El usuario no puede interactuar directamente con sus enemigos, pero dispone de distintas torretas que puede colocar en el entorno para combatir a sus adversarios. Toda torre debe ser colocada y ocupar una posición en específico, desde donde permanece realizando una función concreta basada en su tipo. Por ejemplo, pueden existir cañones que disparan proyectiles o muros que bloquean el paso a enemigos.

Es común que exista una gran diversidad de torretas, donde cada una se diferencia por características como su finalidad y apariencia. Esto incita al jugador a usar distintos tipos de artilugios según el contexto: torretas con daño en área pueden ser eficaces para conjuntos de enemigos numéricos y endebles, pero inefectivas contra adversarios resistentes.

El usuario dispone de una cantidad de recursos limitados con los que poder permitirse sus construcciones, de manera que cada tipo de torreta tenga un coste asociado en relación con su impacto en el nivel. Además, en algunos juegos se da la opción de invertir costes adicionales para realizar mejoras sobre torres ya colocadas, mejorando su efecto.

Según el videojuego existen distintas restricciones sobre dónde se permiten colocar las torres. Algunos *Tower Defense* permiten situar torretas directamente en el camino de los enemigos, mientras que otros lo impiden y se deben colocar en sus laterales. Se les otorga algún tipo de habilidad especial a los adversarios en el primer caso, de manera que sean capaces de destruir las construcciones del jugador cuando se aproximen suficientemente a ellas. Esto se debe a que las torretas estarían bloqueando activamente el paso a sus enemigos, y estos necesitan despejarlo para seguir avanzando.

Respecto a la economía de recursos, el jugador empieza su partida con una cantidad finita de ellos, pudiendo adquirir más de estos a lo largo de esta. Curiosamente no existe un único estándar sobre cómo se pueden obtener, pudiendo ser distinto juego a juego. Aun así, las formas de ingreso más comunes suelen ser las siguientes: eliminando enemigos, recolectados por torres especializadas en la producción de estos, u otorgándose de manera automática por el juego periódicamente. Estas vías de ingreso no tienen por qué ser mutuamente exclusivas, algunos juegos incluyen varias de estas simultáneamente.

Tanto las torretas como los enemigos suelen ser controlados por una inteligencia artificial (IA), en otras palabras, un algoritmo que determina cuáles deberían ser sus mejores acciones para un momento dado. Deben existir tantos tipos de IA como posibles actitudes de personajes en el juego. Por ejemplo, puede haber una diseñada para ser usada en morteros que calcule el ángulo de disparo necesario con el que apuntar sus explosivos hacia los enemigos que osen entrar en su rango de ataque.

Siendo los personajes del campo de batalla de ambos lados controlados automáticamente por algoritmos, cae en manos del jugador planificar su estrategia y gestionar sus recursos con los que permitirse las torretas.

La calidad y cantidad de enemigos depende del nivel, dividiéndose por hordas. Una horda es un conjunto de enemigos que se invoca cada cierto tiempo en el mapa, el cual el jugador debe derrotar. Las primeras oleadas suelen ser las más fáciles de eliminar, incluyendo pocos enemigos y débiles. A medida que avanza el nivel, las hordas aumentan en poder (cantidad y/o calidad de enemigos), dificultando su eliminación.

Esta idea de oleadas favorece el flujo del juego, ya que al principio se espera que el jugador no disponga de numerosos recursos con los que permitirse los artilugios más poderosos. A medida que avanza el nivel más tiempo pasa, enriqueciendo al usuario y permitiéndose situar torretas más poderosas con las que combatir grupos de enemigos superiores.

Un nivel acaba cuando el usuario derrota todas las oleadas del nivel o suficientes adversarios capturan la zona objetivo. En el primer caso se supera el nivel y se permite pasar al siguiente, mientras que el otro implica la derrota del jugador y tener que intentar la partida de nuevo. Dependiendo del juego, es posible que la infiltración de un único oponente sea suficiente para terminar la partida.

Existen una gran variedad de juegos de este tipo, algunos de los más conocidos son *Plants vs. Zombies* [22], *Bloons Tower Defense 5* [23], y *Kingdom Rush Tower Defense* [24].

3.3 Definición del objetivo alcance del proyecto

3.3.1 Objetivo

El objetivo principal del proyecto es implementar un juego del género *Tower Defense*, como el que se ha descrito en el apartado anterior, adquiriendo todos los conocimientos necesarios para hacerlo posible.

3.3.2 Alcance

Para cumplir con la finalidad del trabajo es necesario conocer y aclarar qué debería implementar el programa exactamente (cómo debería funcionar, qué funcionalidades tener, de qué puede prescindir…). Esto es un paso que todo proyecto requiere, conociéndose como la denominación del alcance del trabajo.

Una de las mejores estrategias para definir el alcance de un proyecto se basa en definir su producto mínimo viable o *Minimum Viable Product* (MVP) de antemano. Un MVP indica las características mínimas que debería tener el producto o resultado final de un proyecto para poder declararse acabado [6].

En el ámbito empresarial de los videojuegos esto se recoge en un documento conocido como *Game Design Document* (GDD) o Documento de Diseño de Videojuego, que indica el alcance y los objetivos base del programa que debe cumplir [7]. El GDD se escribe antes de empezar a desarrollar el juego, para poder acordar qué se debe implementar y qué no. Aunque flexible en su extensión y modo de redacción según la empresa y la forma de trabajar, suele tratar los siguientes campos relevantes al juego:

- **Historia o contexto**: Definir el universo o entorno ficticio donde toma parte la trama, así como algunos detalles sobre hechos pasados en este.
- **Personajes**: Diseñar los protagonistas y otros seres que forman parte del universo, así como sus funciones o acciones posibles.
- **Diseño de niveles y forma de juego**: Plantear cómo se juega, qué objetivos se deben cumplir en los niveles, sus grados de dificultad, etc.
- **Interfaces de usuario (IU) y menús de juego**: Definir los paneles interactuables con los que el jugador comprueba información en su estado actual de partida y/o interactúa activamente con el juego.
- **Música y efectos de sonido**: Plantear qué archivos acústicos serán necesarios y en qué contextos.
- **Plataformas**: Elegir en qué dispositivos se podrá jugar al juego.

En el apartado a continuación se define el GDD creado para este proyecto en específico.

3.4 GDD del proyecto planteado

I. Contexto

El personaje sin nombre jugable vive en una mina situada en un desierto de donde extrae minerales valiosos para crear sus propias construcciones.

En un día corriente, sin ningún tipo de aviso robots del futuro empezaron a viajar a esta era. En el mañana ha habido una catástrofe, y ahora estos seres metálicos quieren aprovecharse de los bienes disponibles situados en la mina de nuestro constructor.

Está en manos del jugador defenderse de hordas sin fin de robots, poniendo en práctica sus grandes habilidades para construir torretas. Su objetivo es pararles los pies y enviarlos de vuelta al futuro de donde provienen.

II. Forma de jugar

El jugador dispondrá de una pantalla de selección de torres antes de empezar cada nivel para elegir cuáles en concreto usar. Cada uno de estos artilugios tiene una apariencia propia con una finalidad en concreto, no disponiendo de una gran variedad de ellos al principio del juego.

A medida que se completan niveles el usuario desbloqueará más de torretas, pero también se enfrentará a nuevos tipos de enemigos (generalmente más robustos que los anteriores). Todos los niveles ocurrirán en el mismo entorno desértico que se ha descrito en el contexto del juego.

El jugador empezará una partida con una cantidad limitada de recursos. Deberá colocar un tipo de torreta especializada en la recolección de estos para obtener más, siendo así la única forma de reponerlos.

Los enemigos se tratarán de robots que se aproximan por grupos u oleadas desde la parte derecha del mundo. Cada uno de estos individuos aparecerá en una de las 5 filas o caminos posibles, desplazándose así hacia la izquierda en línea recta. Algunas oleadas podrán ser grandes hordas, lideradas por un robot especial y siendo más numerosas que las corrientes.

Cada fila estará formada por múltiples casillas. El jugador deberá situar sus torres elegidas en la fase anterior sobre estos recuadros para defenderse de sus amenazas durante la partida. Cada torreta dispondrá de un coste, así como un tiempo de recarga tras ser colocada. De esta manera, se impide que se puedan colocar varias del mismo tipo en tiempo reducido. Una interfaz de usuario permitirá al usuario elegir qué categoría de torreta en concreto colocar en casillas en un momento dado.

Si se derrotan todas las oleadas de enemigos de un nivel, entonces se desbloqueará una torreta nueva y se pasará al siguiente. Por el contrario, si un enemigo llega a la mina del usuario situada en el extremo izquierdo del entorno virtual, entonces se perderá la partida y deberá reintentar el nivel.

El modo de juego está principalmente inspirado en *Plants vs Zombies* [22], otro juego *Tower Defense* ya existente desarrollado por la empresa *PopCap* (ver figura 1). Las torretas tomarán el rol de las plantas, mientras que los robots representarán los zombis.

Figura 1: El juego Plants vs. Zombies, desarrollado por *PopCap*

III. Personajes

Excluyendo el personaje del jugador que no va a ser visible, los restantes son los robots y torretas del juego.

III.I Torres

Los artilugios se colocan en las casillas por el usuario. Existen 5 de ellas planeadas para añadir al juego:

- **Panel Solar**: Produce una cantidad concreta de recursos cada cierto tiempo que puede ser recolectada por el jugador.
- **Cañón**: Ataca a enemigos en su fila.
- **Cañón Doble**: Ataca a enemigos en su fila con el doble de cadencia de un cañón normal.
- **Panel Solar Doble**: Produce recursos más eficientemente que un panel normal.
- **Bomba**: Explota, eliminando toda amenaza en un área de casillas 3x3.

III.II Robots

Los robots son los antagonistas del juego y tienen dos acciones fundamentales: moverse y atacar.

- Moverse: Los robots se desplazan hacia la mina del jugador si no tienen un objetivo cercano al que atacar.
- Atacar: En el caso de que una torreta esté situada enfrente suya, el robot se detendrá y empezará a disparar generando rayos láser por sus ojos para destruirla.

Estos antagonistas se diferenciarán principalmente por la velocidad de movimiento y su cantidad de salud (resistencia a ataques de torretas), estos son:

- **Simple**: Poca salud y lento.
- **Medio**: Cantidad de salud decente y lento.
- **Duro**: Mucha salud pero lento.
- **Líder**: Poca salud y veloz. Este tipo de enemigo se reserva para las grandes oleadas, en las cuales siempre aparece uno de ellos liderando la marcha.
- **Bomba**: Cantidad de salud moderada y es lento. Detona al perder todos sus puntos de vida, eliminando todas las construcciones del usuario en un área de casillas 3x3.
- **Bomba Radar**: Tiene muchísima salud y es lento. Es idéntico al tipo de enemigo anterior, pero puede detonar prematuramente si tiene suficientes torretas en su radio de explosión.

IV. Niveles

Los niveles se diferenciarán por la cantidad y categorías de enemigos que aparecerán en cada uno de ellos, así como por los tipos de torretas disponibles que el jugador tendría hasta el momento.

Contabilizando la cantidad de personajes propuestos se espera que existan al menos 5 niveles en el juego, donde se irán presentando nuevos tipos de enemigos y torretas (ver tabla 1).

Nivel	Torres disponibles	Grandes oleadas	Enemigos
1	Panel solar, cañón	1	Simple
2	Panel solar, cañón	2	Simple, bomba
3	Panel solar, cañón, cañón doble	1	Simple, medio
4	Panel solar, panel solar doble, cañón, cañón doble	2	Simple, medio, duro
5	Panel solar, panel solar doble, cañón, cañón doble, bomba	2	Simple, medio, duro, bomba, bomba radar

Tabla 1: Plan inicial de niveles del juego

V. Interfaces

El juego dispondrá de un menú principal en el que el usuario podrá al menos continuar desde el último nivel desbloqueado o empezar una nueva partida. Estas dos opciones serán los casos de uso principales del programa.

Además, se incluirán dos pantallas adicionales ya mencionadas en el apartado de las normas de juego:

- Una interfaz en la fase de selección de todo nivel donde el usuario puede seleccionar que torretas llevar a este.
- Otra pantalla durante la partida en sí que permite al jugador decidir qué torreta colocar una casilla, siempre existiendo esta dentro de la lista de torres elegidas en la fase de selección.

VI. Audio

Se planifica tener siempre música de fondo presente en el juego que cambie según el contexto en el que se encuentre el jugador (el principal factor siendo si se está en el menú principal o en una partida). Se quieren añadir efectos de sonido cuando las torretas disparan proyectiles, así como cuando se destruyan robots.

VII. Plataformas

Se priorizará el desarrollo del programa para que funcione por ordenador. No se descarta la idea de desarrollar ejecutables para jugar en consolas, aunque no será el objetivo inicial.

4. Estado del arte

4.1 Estilos de desarrollo

La creación de videojuegos es algo normalizado a día de hoy. Existen empresas que se dedican profesionalmente a esto para poder sobrevivir en el mercado. Ejemplos de estas son Sony y Nintendo, existiendo centenas de ellas más.

Aun así, no todos los juegos se originan en empresas profesionales. Existen individuos (o pequeños grupos de gente) que desarrollan sus propios videojuegos y los publican en internet. Los juegos desarrollados en estos entornos modestos se denominan juegos *indie* [8].

Como se puede observar, ambos grupos se diferencian por la manera de trabajo y finalidad:

Los juegos empresariales se crean bajo un presupuesto dado por la compañía, y existen distintos departamentos para poder hacerlo realidad, crear publicidad para venderlo, manejar la distribución del producto y más.

Los juegos *indies* se crean bajo la idea de una única persona o un número reducido de ellas por lo general, quienes la hacen realidad con recursos más limitados. Hay veces que se crean como un pasatiempo y no tienen ánimo de lucro, aunque no es siempre el caso.

Ya que este proyecto se trata de un TFG, la metodología de trabajo se parece más a este segundo tipo de entorno, debido a que se carece de personal profesional empresarial para desarrollarlo.

4.2 Herramientas necesarias

Para crear videojuegos es común utilizar herramientas que permitan crear el producto en sí, almacenar copias de su entorno de desarrollo remotamente, y generar la documentación que le acompaña.

Respecto a esta primera necesidad, los aspectos principales de un programa de estas características son el audio, los personajes y la lógica del juego en sí. Cada uno de estos requiere a su vez el uso de distintos tipos de programas, siendo necesario analizar en cada caso qué herramienta se quiere usar.

El motor de juego es el *software* que permite crear la lógica de programa, diseñar y alojar el entorno virtual donde toma parte, así como importar audios y aspectos visuales para poder hacer un juego realidad [9]. Existen varios *software* que cumplen estos requisitos, ejemplos pueden ser *Unity*, *Unreal Engine 5* y Godot. Tras deliberación con las directoras del TFG, se decidió usar el programa *Unreal Engine 5*, principalmente por los siguientes motivos:

En primer lugar, una de sus características diferenciadoras con respecto a la competencia es la disponibilidad de repositorios gratuitos exclusivos de donde adquirir decoraciones de entorno para juegos y no necesitar adquirir ninguna licencia de pago para eliminar marcas de agua en productos finales. Junto con esto, cabe destacar que su versión más reciente ha rediseñado su interfaz gráfica enormemente, haciéndola más fácil de usar en comparación con sus adversarios (por lo menos en opinión del autor del proyecto, este argumento tiende a ser subjetivo). Finalmente, la codirectora del proyecto Marra Aguilera tiene gran familiaridad con esta herramienta en concreto, pudiendo dar soporte cuando fuera necesario en el desarrollo del proyecto y recomendar cursos *online* durante el proceso de formación.

Para complementar el trabajo desarrollado en el motor de juego se necesita un entorno de desarrollo integrado (también conocido como IDE, sus siglas en inglés) [10]. Un IDE es una aplicación informática usada para implementar algoritmos de manera condensada en código escrito, o explicado en términos más simples, un editor de código que permite escribir instrucciones de programa en el juego por teclado.

Virtual Studio Code es la herramienta elegida que cumple con estas características, debido a su facilidad de integración en *Unreal Engine 5* y existencia de *plugins* incorporados como "C/C++" y "*Unreal Engine 4 Snippets*" (también compatible con la versión 5) que facilitan la experiencia de desarrollo. Además, *VSCode* ya se había usado previamente en otros proyectos del grado, por lo que existía familiaridad con la herramienta (siendo esta la característica definitiva para elegirlo como IDE del trabajo).

Respecto a la creación de los aspectos visuales del juego, es necesario disponer de un programa que permita crear figuras tridimensionales de cero y darles forma a gusto del diseñador. De esta forma, se pueden crear las apariencias de los personajes necesarios para el juego en el entorno virtual. Los programas de modelado 3D son los adecuados para esta necesidad [11].

Uno de los más usados es *Blender*, un editor gratuito que permite modelar figuras 3D de manera sencilla [12]. Esta herramienta permite diseñar y exportar figuras tridimensionales customizadas, siendo estos requisitos fundamentales para poder añadirlas al motor de juego. Además, existen muchos recursos de formación *online*.

Debido a todo lo anterior, *Blender* ha sido la opción elegida para la creación de aspectos tridimensionales. Aun así, existen varias plataformas que también cumplen estos requisitos como *TinkerCAD*, *SketchUp* y *Leopoly* [13]. *Blender* fue la primera opción a conocer, y como nunca dio problemas se mantuvo como el editor de figuras tridimensionales para el proyecto.

Para cumplir con el criterio del audio existen varias opciones disponibles. Específicamente, solo se necesita tener la opción de recortar pistas y juntarlas para poder eliminar porciones de sonido no deseados. Esto se debe a que los orígenes de todos los recursos acuáticos del juego son externos, requiriendo editar estos archivos como paso previo a añadirlos al programa.

La última versión del programa de edición de audio *Audacity* destaca sobre la competencia por su interfaz intuitiva, licencia gratuita, y comodidad de uso para las funcionalidades necesarias listadas anteriormente. Además, *Audacity* es el editor de audio por defecto para mucha gente y una de las opciones más elegidas en el mercado [14]. Durante el desarrollo del proyecto no decepcionó y se mantuvo como el editor de pistas de audio para el juego.

Para almacenar proyectos informáticos en la nube siempre se ha hecho uso de la plataforma *GitHub* [15] durante el grado, cuyo entorno permite hacer copias de programas y almacenarlas de manera externa. Existen otras alternativas, pero cualquier versión de *Unreal Engine* tiene la opción de conectar fácilmente con *GitHub* [16]. Además, ya teniendo experiencia con esta plataforma implica no gastar más tiempo en aprender a usar herramientas alternativas con la misma función.

Este último argumento sería el decisivo para mantener a *GitHub* como la opción de *back-ups*. Se puede encontrar el enlace al repositorio público de la aplicación en la portada de este mismo documento.

Respecto a la documentación, se han usado dos editores de texto principalmente durante la carrera (*Drive* y *LaTeX*), cada uno de ellos teniendo sus puntos altos y bajos. La decisión no estaría tan clara como en los apartados anteriores y se necesitarían contrastar sus puntos fuertes y débiles (ver tabla I en el anexo).

Tras gran deliberación, se decidiría usar *Drive* principalmente por tener una mayor familiaridad con ella. Es imperativo saber manejar la herramienta eficazmente para desarrollar un documento de gran extensión.

Estas herramientas en conjunto se complementan adecuadamente, permitiendo hacer el proyecto posible (ver figura 2).

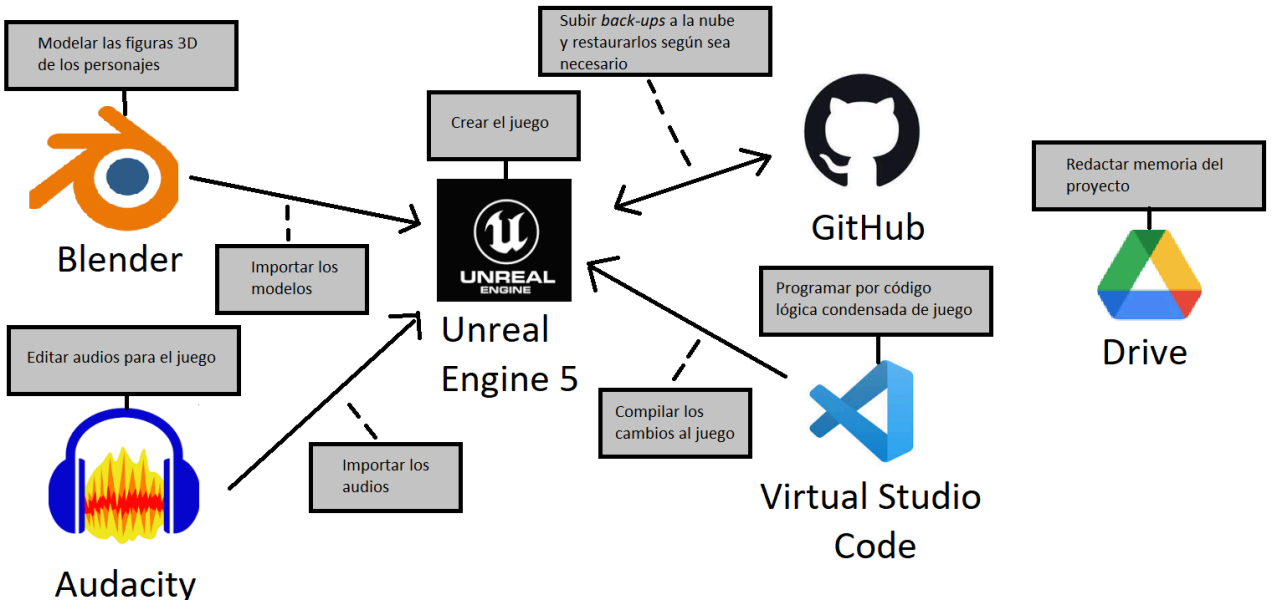


Figura 2: Herramientas usadas en el proyecto: sus funciones y relaciones

5. Riesgos identificados

Todo proyecto siempre se enfrenta a adversidades y es necesario identificarlas antes de empezar a trabajar, con el propósito de minimizar su probabilidad de materialización, e impacto. Se presentan en la siguiente tabla los riesgos identificados, sus valores estimados respecto a los dos atributos previamente mencionados, y sus planes para evitarlos (prevenciones) y minimizar el impacto (planes de contingencia):

Riesgo	**Prevención**	**Plan de Contingencia**	**Probabilidad**	**Impacto**
Pérdida de datos	Actualizar los cambios realizados en el juego a *GitHub* cada día. Guardar copias de recursos externos (modelos 3D, música…) en un dispositivo USB.	Restaurar los archivos perdidos desde su origen remoto correspondiente.	Baja	Muy alto
Desacuerdos con las directoras del proyecto	Comunicar con ellas cada cierto tiempo para informar los avances en el trabajo, y recibir propuestas o mejoras.	Corregir los cambios pedidos en su caso y tratar de llegar a un acuerdo.	Baja	Moderado
Falta de presupuesto	Estimar el coste y tratar de elegir herramientas gratuitas si fuera posible sin afectar la calidad.	Invertir más dinero en el proyecto.	Muy baja	Bajo
Fallo de estimación temporal	Trabajar a diario, siempre hacer progresos, priorizar la formación si hay dudas en el desarrollo.	En caso de no cumplir con el plazo, se tendría que aplazar la convocatoria de la defensa del trabajo.	Baja	Alto
Dificultad del proyecto superior a lo estimado	Conocer todos los recursos necesarios de antemano y saber qué se va a hacer. Si se quieren realizar mejoras o adiciones, asegurarse de que se tiene una idea de cómo hacerlo.	Si no se sabe cómo desarrollar un aspecto, entonces se debe buscar un tutorial o curso de cómo hacerlo.	Moderada	Moderado
Problemas legales	Si se usan recursos externos, comprobar sus contratos de propiedad intelectual antes de añadirlos al proyecto.	Eliminar el recurso que cause el problema, y reemplazarlo por otro o crearlo de cero.	Muy baja	Bajo
No se puede implementar el diseño con las herramientas planteadas	Formarse sobre las herramientas a usar. Después, formular los detalles específicos a implementar para saber si son compatibles.	Modificar el diseño lo más ligeramente posible para permitir su implementación en la realidad. Si no es posible, usar otra herramienta.	Baja	Bajo o alto (depende del caso)

Tabla 2: Riesgos identificados

6. Descripción de la solución propuesta (el juego creado)

El programa consiste de un ejecutable *Windows* que incluye el videojuego desarrollado en *Unreal Engine 5*, pudiéndose encontrar este archivo en el repositorio *GitHub* presentado en la portada del documento. Implementa todas las funcionalidades propuestas en el GDD del proyecto (exceptuando soporte multiplataforma), así como algunas adicionales. El programa incluye dos ambientes o mapas distintos: la zona del menú principal y el área de juego.

6.1 Casos de uso: menú principal

Al abrir el ejecutable se presenta el menú principal del juego con aspecto futurístico (ver figura 1.2.1). Se aloja en un entorno formado por una imagen de fondo, varios personajes estáticos, y proyectiles que se generan dinámicamente y se desplazan de lado a lado (ver figura 1.3.1).

Figura 1.2.1: Menú principal

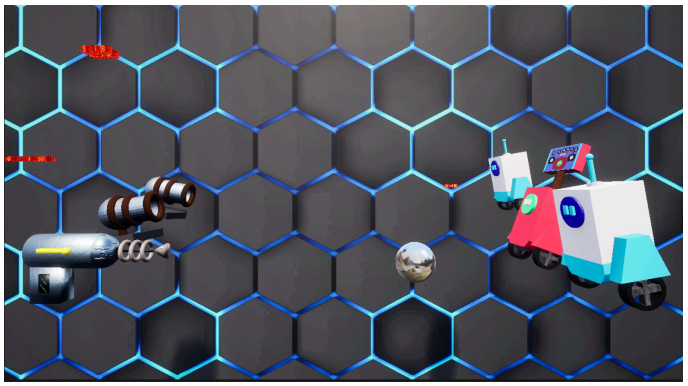


Figura 1.3.1: Entorno del menú principal

Las opciones disponibles a elegir son las siguientes:

- **Nuevo Juego**: Empezar una partida de cero, sobrescribiendo en el archivo de guardado del jugador el valor de su nivel actual al primero y cargarlo tras esto. Si ya se había completado al menos una partida, se mostrará un aviso de si realmente se desea restablecer el progreso.

- **Continuar**: Usar el archivo de guardado del jugador para cargar el nivel en el que se encuentra. Existen 10 de ellos alojados en el mismo entorno virtual, diferenciándose por la cantidad y tipos de enemigos que aparecen en estos. Si no existe progreso actual, la opción no está disponible.

- **Cómo jugar**: Muestra una pequeña presentación de varias páginas en la que se explica la narrativa del juego y sus normas (ver figura 1.2.2). Se ha mantenido la narrativa inicial planteada en el GDD, pero se han realizado algunas adiciones a las normas de juego. En el siguiente apartado (6.2) se describe en detalle el funcionamiento de una partida.

- **Ajustes**: Abre una pestaña que permite modificar algunas opciones del programa (ver figura 1.2.3). Se incluyen:
    - **Calidad de gráficos**: baja, media, alta o ultra.
    - **Idioma**: español o inglés.
    - **Modo de ventana**: pantalla completa o en ventana.
    - **Resolución de pantalla en modo ventana**: las opciones posibles dependen del dispositivo.
    - **Volumen de música**: escala (0-100)%.
    - **Volumen de efectos de sonido**: escala (0-100)%.
    - **Sincronización vertical**: habilitar o deshabilitar.
    - **Fotogramas por segundo**: entre 30 y 150, si la sincronización vertical está desactivada.

  Cabe destacar el ajuste de la sincronización vertical, cuya habilitación elimina imperfecciones en la pantalla sacrificando ligeramente la latencia en los controles del usuario. Por el contrario, fijar una tasa de fotogramas por segundo permite dar más fluidez al juego, a cambio de arriesgar la manifestación de imperfecciones en la pantalla durante el juego [17]. Estas configuraciones son soportadas por el motor de juego, pero se ha tenido que crear la interfaz que permite habilitarlas.

- **Créditos**: Muestra los nombres de toda la gente que ha contribuido de alguna forma al juego, junto con sus aportaciones (ver figura 1.2.4). Su funcionalidad principal es nombrar los autores de los recursos externos usados en el programa, cumpliendo con sus licencias respectivas y siendo también acreditados en las tablas V del anexo. Se incluye una página extra de agradecimientos en los créditos donde figuran las directoras del TFG y los profesores de los cursos visualizados que han hecho este proyecto realidad. Los agradecimientos formales redactados se pueden encontrar en el apartado 10.5 de este documento.

- **Salir**: Carga la ventana de confirmación de salida (ver figura 1.2.5).

Todas las opciones del menú (exceptuando el botón que cierra la aplicación) representan cada uno de los casos de uso del jugador, no existiendo otro tipo de rol en el programa (ver figura 3). En el anexo I se puede encontrar cada caso de uso desarrollado de manera técnica en mayor detalle.

Figura 1.2.2: Pantalla del tutorial del juego

Figura 1.2.3: Pantalla de ajustes

Figura 1.2.4: Pantalla de créditos

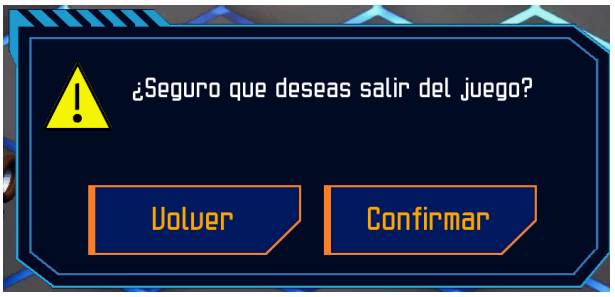


Figura 1.2.5: Pantalla de aviso

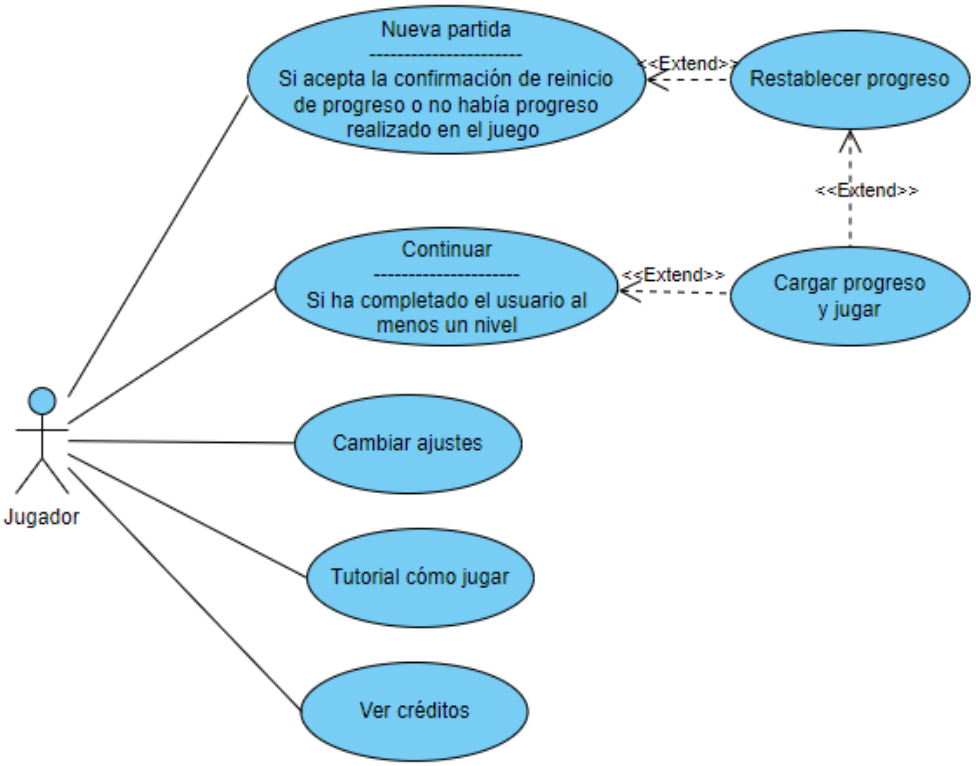


Figura 3: Casos de uso de la aplicación

6.2 Subcaso de uso (cargar progreso y jugar): área de juego

Al empezar una partida el usuario aparece en el área de juego: un entorno desértico con minas a ambos laterales de la pantalla. La estructura de la izquierda está en posesión del jugador y sirve como zona objetivo para los robots enemigos que desean capturarla (ver figura 1.3.2).

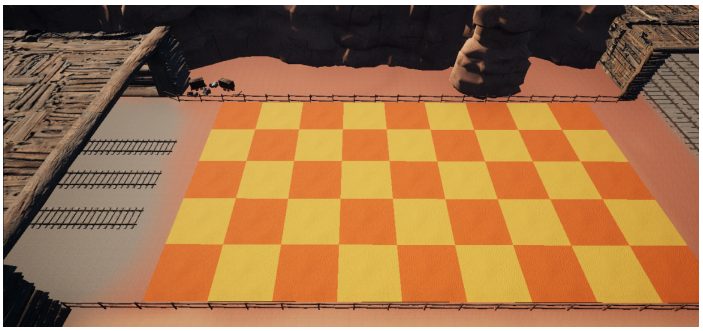


Figura 1.3.2: Área de juego

Todo nivel empieza por la fase de selección de torres. Aquí, se presentan todos los tipos de enemigos que van a aparecer durante el nivel (ver figura 4) . De esta forma, el usuario puede planificar que tipo de torres desea usar para combatirlos y elegir las correspondientes usando una interfaz de usuario (IU) (ver figura 1.2.9).

Figura 1.2.9: Interfaz de selección de torres.

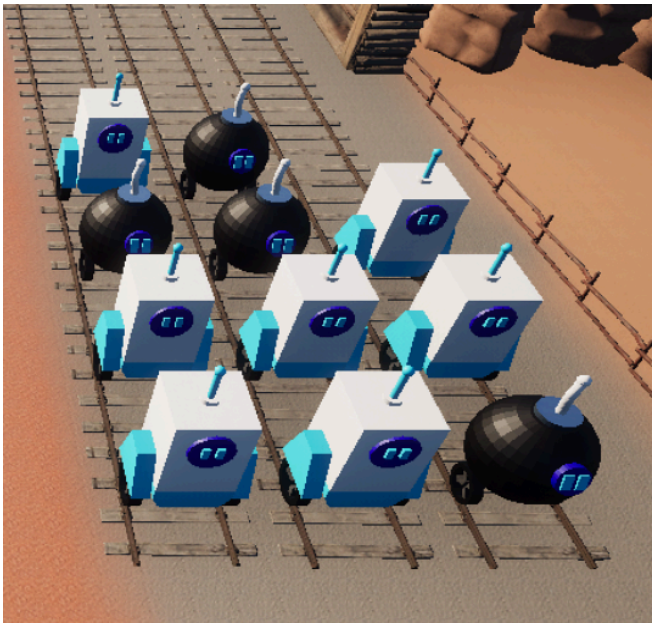


Figura 4: Presentación de tipos enemigos a aparecer en un nivel.

Tras haber confirmado su selección formada por 6 torretas (o todas las disponibles si no alcanzan ese valor), se da comienzo al juego.

Para colocar una torreta durante la partida, esta debe haber sido elegida en la fase de selección anterior. Si fue el caso, aparecerá su tarjeta en una interfaz situada en la parte superior de la pantalla (ver figura 1.1.11). Cada tarjeta indica el precio de su torre respectiva para poder ser colocada en el entorno y posee un tiempo de recarga asociado durante el cual no puede ser seleccionada (representado por un fondo gris que se va quitando progresivamente).

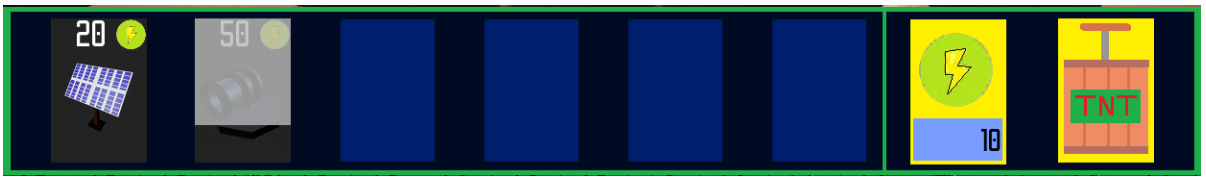


Figura 1.1.11: Interfaz principal de juego

El usuario debe poder permitirse el precio de la torre y no estar en recarga para seleccionarla. Si puede hacerlo, una vez pinchada sobre ella debe elegir una casilla vacía donde colocarla. Un recuadro se considera vacío si no existe una torre ahí y no hay un robot encima de él en el momento. Al colocar se paga el coste de la construcción y se impide seleccionar ese tipo de torre de nuevo durante su nuevo periodo de recarga.

También se da la opción de quitar torretas ya colocadas en casillas. Para esto, el jugador debe hacer click en la tarjeta con imágen de dinamita (TNT) en la interfaz superior, seguido de la casilla que desea liberar.

El usuario empieza con 40 puntos de energía que puede invertir en torres. Existen torretas con distintas funcionalidades: cañones que atacan a enemigos en su fila (ver figura 5), paneles que producen energía (ver figura 6), y demás. Todas las torretas que no estén destinadas a la producción de recursos empiezan descargadas en el nivel y no podrán ser colocadas durante los primeros segundos. No existen mejoras disponibles para las construcciones, pero hay una gran variedad de estas con distintos precios asignados según el impacto que tienen en el juego.

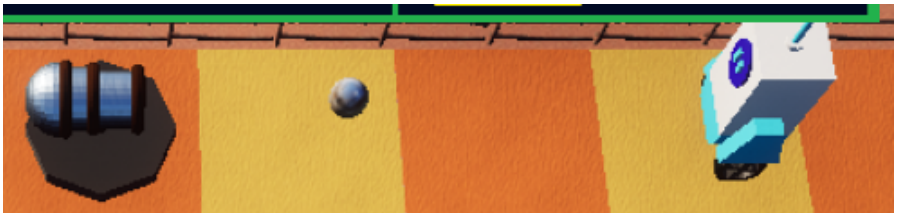


Figura 5: Un cañón disparando a un enemigo en su fila

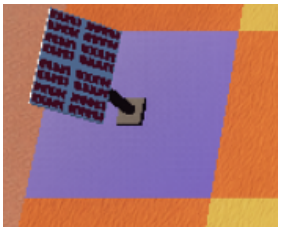


Figura 6: Un panel con energía producida

Los seres futurísticos aparecen en la zona derecha del nivel, desplazándose hacia la zona objetivo en la izquierda. Durante su recorrido, se pueden detener para atacar a toda torreta que se presente en su fila si se encuentra a suficiente proximidad. Para dañarlas, usan rayos láser que generan por sus ojos e impactan contra las construcciones del jugador (ver figura 7).

Figura 7: Robot disparando rayos láser a un panel

Los enemigos no se aproximan de golpe al empezar una partida, en su lugar, cada nivel define una lista de oleadas en un orden concreto donde aparecen conjuntos de estos. Cada tipo de robot tiene un peso o un valor numérico asociado que determina su coste de aparición en la partida, de manera que enemigos robustos sean más caros que el resto. Siguiendo esta filosofía, todas las oleadas incluyen un presupuesto máximo a poder usar y una lista de tipos de adversarios que se permiten generar. Junto con esto, toda horda intenta repartir la cantidad de robots que aparecen en ella de manera equivalente por cada fila, nunca teniendo más de uno en una que en otra.

En todo nivel la primera oleada tarda 26 segundos en aparecer. Esto da tiempo al jugador de que empiece a colocar paneles solares y pueda desarrollar su economía de energía, de manera que se pueda permitir situar más torretas después. Tras la primera aparición de enemigos, la siguiente en su lista se genera cada 20 o 30 segundos (depende según el caso), aunque se puede invocar prematuramente si la suma de daños en los robots de la horda actual superan el 50% de su salud total. La cantidad de oleadas invocadas en el nivel hasta el momento figura en la barra de progreso del nivel situada en la esquina inferior izquierda de la pantalla (ver figura 1.1.9).

Figura 1.1.9: Barra de progreso

Cada ciertas hordas superadas aparece una gran oleada en la que se presentan más enemigos de lo normal, siendo está la única manera en la que pueda aparecer un robot líder en el campo de batalla (ver figura 8). Estas apariciones se representan como una bandera en la barra de progreso, siendo la última de las oleadas siempre una de ellas. Aún así, es posible que haya más a lo largo del nivel.

Figura 8: Gran oleada de un nivel

El usuario debe derrotar a todos los enemigos de cada oleada para ganar. Si esto ocurre, se le recompensa (generalmente con una nueva torre) y se pasa al siguiente nivel (ver figura 1.2.12). Por el contrario, el usuario pierde si un robot recorre su fila entera y llega al lado izquierdo del nivel, necesitando internar el nivel desde el principio (ver figura 1.2.13).

Figura 1.2.12: Pantalla de victoria

Figura 1.2.13: Pantalla de derrota

Se felicita al jugador cuando supera el décimo y último nivel. Además, se le permite volver a jugarlos de nuevo, manteniendo todas las torretas del juego desbloqueadas desde el principio esta segunda vez.

En anexo se puede encontrar información adicional sobre los personajes disponibles en las tablas II, así como datos de cada nivel en la tabla III.

7. Plan de proyecto

7.1 Tareas realizadas

Para poder hacer el proyecto realidad se han identificado y seguido 6 tipos de tareas principales pero fundamentales que se pueden ver reflejadas en el siguiente diagrama:

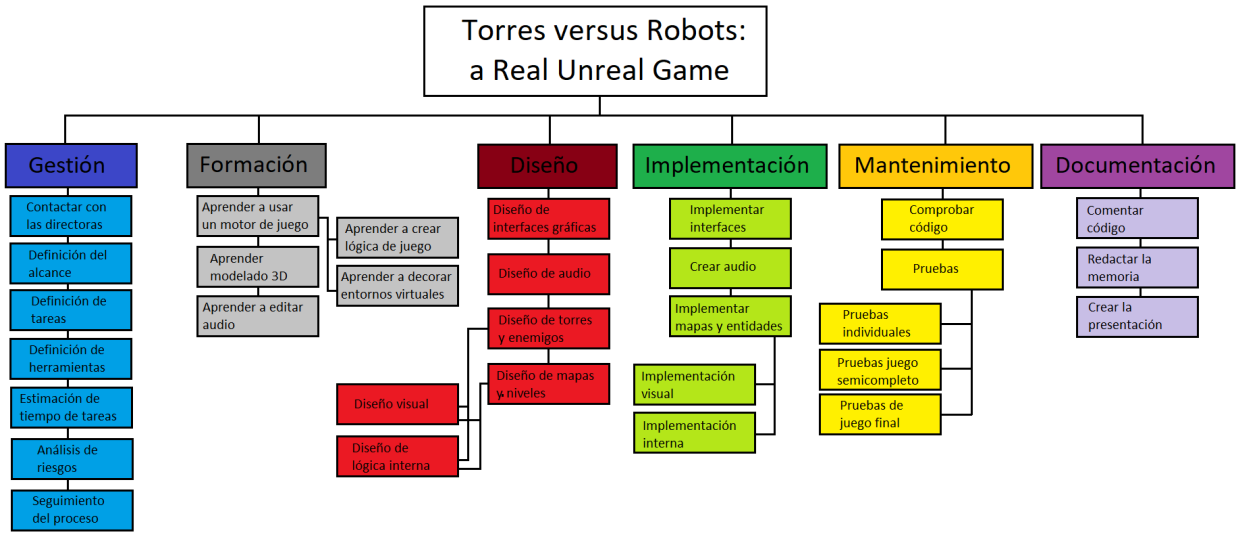


Figura 9: EDT del proyecto

La descripción que aparece a continuación sigue un orden cronológico:

- **Gestión**: Verificar los recursos y herramientas que se van a usar, junto con qué tareas se deben realizar en el proyecto. Además, realizar un seguimiento de control para comparar el progreso hasta el momento con la planificación inicial, asegurando que se siguen las estimaciones iniciales.

- **Formación**: Adquirir las competencias necesarias para poder hacer uso de las nuevas herramientas. Se incluye en este paso:

    - Seguir un tutorial de modelado 3D en *Blender* [27].
    - Acomodarse a la edición de audios en *Audacity*.
    - Aprender a hacer uso de *Unreal Engine 5* mediante un *pack* de cursos de pago en *gamedev.tv* que incluye:

- Lecciones de 30h en las que se enseña a cómo usar el programa mediante la implementación de 5 juegos sencillos de ejemplo [25]. La estimación temporal solo incluye la duración de los videos, siendo en la realidad bastante más.
  Esto se debe a que se incentiva al estudiante a parar los videos frecuentemente para que él o ella misma pueda asimilar los conceptos y realizar ejercicios recomendados. Las sumas alcanzan casi 20 días de dedicación en seguimientos diarios.

- Seguir lecciones adicionales en otro curso donde se aprende a diseñar entornos virtuales [26]. Este paso es más corto y se puede finalizar en una tarde.

- **Diseño**: Planificar cómo se va a crear e implementar el juego en *Unreal Engine 5* para poder tener una idea clara y eficaz. Esto incluye diseñar todos sus componentes de antemano y cómo van a interactuar entre ellos (las normas y controles de juego, los personajes, etc.). También se deben plantear diseños de recursos a crear en otros programas (modelos 3D y estilo de música).

- **Implementación**: Llevar a la práctica todo lo anterior planificado para poder crear el producto final.

- **Mantenimiento**: Realizar planes de prueba para verificar que todo funciona como es debido. También comprobar que las implementaciones y diseños llevados a cabo sean óptimos, así como que la experiencia de juego sea adecuada.

- **Documentación**: Redactar esta misma memoria y la presentación para la defensa del TFG. Además, escribir comentarios de código durante todo el proceso de desarrollo en archivos del proyecto para facilitar el entendimiento del programa.

7.2 Fases de desarrollo

El proyecto ha consumido mucho tiempo, llegando a alcanzar más de 5 meses de duración total. Se dividen 3 tramos principales de desarrollo, donde cada uno de ellos se caracteriza por contener más tareas de tipos concretos:

Fase 1: Idea, formación y diseño inicial (Feb. 8 - Mar. 7)

La fase 1 se caracteriza principalmente por contener tareas de gestión, formación y diseño.

El objetivo de la primera fase del trabajo es establecer qué se quiere hacer y cómo. Seguido de esto, recopilar las habilidades necesarias para poder hacer el producto realidad. Finalmente, conociendo qué es implementable, diseñar la base del juego.

Su cronograma junto con las tareas realizadas se presenta a continuación:

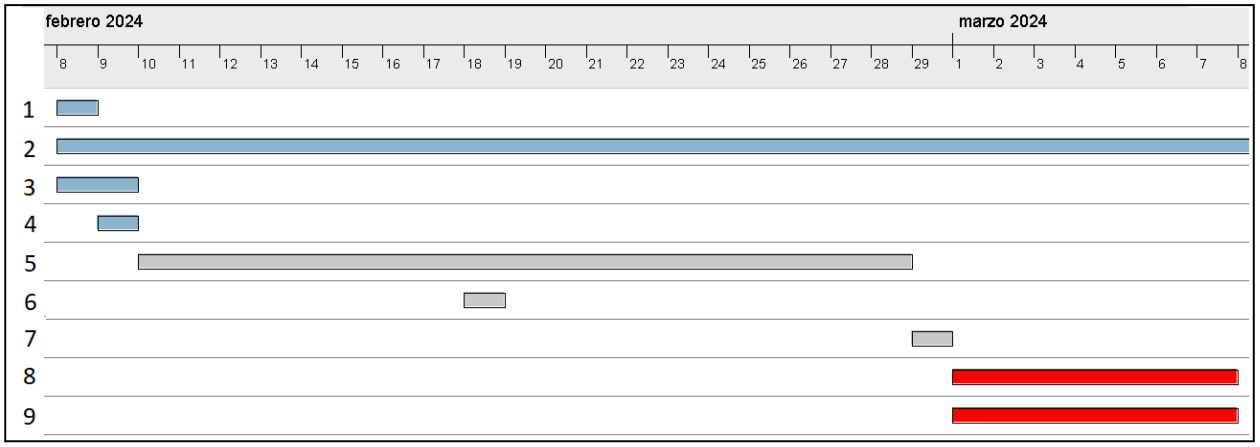


Figura 10: Cronograma de la fase 1 del proyecto

Tarea -> Subtarea	**Fecha inicio y fin**	**Comentarios**
1. Contacto con las directoras	Feb. 8 - Feb. 8	Videoconferencia inicial para introducir la idea y obtener recomendaciones.
2. Gestión y control	Todo el proyecto	…
3. Definir alcance	Feb. 8 - Feb. 9	Definir el GDD.
4. Análisis de riesgos, Estimaciones de tareas & Selección de herramientas	Feb. 9 - Feb. 9	…
5. Aprender a usar un motor de juego -> Aprender a crear un juego	Feb. 10 - Feb. 28	Seguir el primer curso de uso de *Unreal Engine 5* en donde se crean 5 juegos sencillos con los que familiarizarse con la herramienta. [25].
6. Aprender modelado 3D	Feb. 18 - Feb. 18	Visualizar tutorial de uso de *Blender* [27].
7. Aprender a editar audio	Feb. 29 - Feb. 29	Familiarizarse con *Audacity*.
8. Diseño de mapas y niveles -> Diseño de lógica interna & Diseño de interfaces gráficas	Mar. 1 - Mar. 7	Plantear las clases necesarias para hacer funcionar el juego base (derivados de *GameModeBase*, *PlayerPawn y PlayerController).* Planificar interfaces de juego básicas (las mencionadas en el GDD excluyendo el menú principal) y aprender a manejar el editor de interfaces incorporado en *Unreal Engine*.
9. Diseño de torres y enemigos	Mar. 1 - Mar. 7	Diseñar la apariencia y jerarquía de clases que se va a usar. Un diseño de clases adecuado permite organizar funcionalidades para poder reutilizarlas en personajes futuros no planeados todavía.

Tabla 3: Tareas de la fase 1 del proyecto

Fase 2: Implementación (Mar. 8 - May. 12)

La fase 2 del proyecto se focaliza principalmente en la implementación.

Hasta el momento el diseño solo incluye inicialmente las partes imprescindibles del juego, por lo que también existen fases de diseño intermedias para plantear y crear nuevas funcionalidades. Entre estas, se incluye el proceso de extensión del menú principal, quien pasaría a incluir opciones adicionales como la edición de ajustes y navegación de tutoriales de juego.

Finalmente, durante la implementación se incluyen periodos de prueba breves intermedios para comprobar que cada funcionalidad creada funciona como es debido.

Su cronograma junto con las tareas realizadas se presenta a continuación:

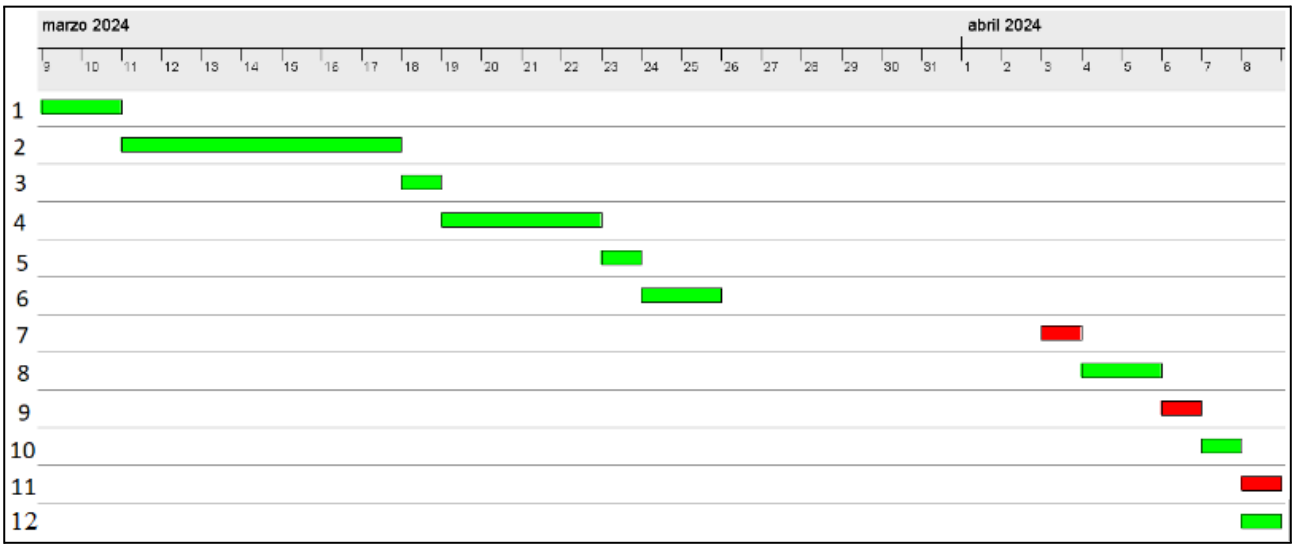


Figura 11: Cronograma de la fase 2 del proyecto (parte 1)

Tarea -> Subtarea	**Fecha inicio y fin**	**Comentarios**
1.Implementar entidades -> Implementación visual	Mar. 9 - Mar. 10	Crear en *Blender* las figuras 3D de las torretas y robots diseñados en la fase anterior. Además, importarlas a *Unreal Engine.*
2. Implementar entidades -> Implementación interna	Mar. 11 - Mar. 17	Crear las clases relativas a las torres y enemigos, y programarles IA para que puedan actuar por ellas mismas.
3. Implementar interfaces	Mar. 18 - Mar. 18	Dar vida a la IU que permite al jugador seleccionar qué tipo de torreta colocar en una casilla. (Tras varios retoques en pasos futuros, acabaría siendo la figura 1.11.1 del anexo).
4. Implementar mapas -> Implementación interna	Mar. 19 - Mar. 22	Implementar la lógica básica de juego: hacer aparecer enemigos y crear el concepto de energía para limitar los recursos del jugador.

5. Implementar entidades	Mar. 23 - Mar. 23	Crear animaciones básicas para los personajes del juego: se incluyen disparo y muerte.
6. Implementar interfaces	Mar. 24 - Mar. 25	Decorar más detalladamente la interfaz creada en el paso 3, y añadirle una barra de progreso del nivel.
7. Diseño de interfaces gráficas	Abr. 3 - Abr. 3	Diseñar la interfaz del menú principal (solo con las opciones de nuevo juego, continuar y salir). Diseñar la IU que permite al usuario elegir qué torretas quiere llevar al nivel antes de empezar la partida. (Tras varios retoques en pasos futuros, sería la figura 1.2.9 del anexo).
8. Implementar interfaces	Abr. 4 - Abr. 5	Implementar los diseños de la tarea anterior.
9. Diseño de torres	Abr. 6 - Abr. 6	Concepción de la mina, una nueva torreta
10. Implementar torres -> Implementación externa	Abr. 7 - Abr. 7	Modelar la mina en *Blender.*
11. Diseño de audio & Diseño de interfaces gráficas	Abr. 8 - Abr. 8	Decidir poner una cuenta atrás antes de empezar la partida. Buscar efectos de sonido adecuados para esto.
12. Implementar audio & Implementar interfaces	Abr. 8 - Abr. 8	Hacer realidad la idea planteada en la tarea anterior.

Tabla 4: Tareas de la fase 2 del proyecto (parte 1)

Figura 12: Cronograma de la fase 2 del proyecto (parte 2)

Tarea -> Subtarea	**Fecha inicio y fin**	**Comentarios**
13. Implementar mapas -> Implementación interna & Implementar audio	Abr. 10 - Abr. 10	Crear la lógica que comprueba las condiciones de victoria y derrota. Implementar el movimiento de la cámara hacia la izquierda en caso de derrota mientras suena música de suspense.
14. Implementar torres -> Implementación interna	Abr. 11 - Abr. 11	Importar el modelo 3D de la mina e implementar las clases, IA y animaciones necesarias para desarrollarla.
15. Diseño de interfaces gráficas	Abr. 12 - Abr. 12	Diseñar las interfaces de victoria y derrota, así como el menú de ajustes, paneles del tutorial de juego y el menú de pausa.
16. Implementar interfaces	Abr. 12 - Abr. 23	Implementar las interfaces mencionadas en el diseño. Expandir el menú principal para permitir acceder a los ajustes y tutoriales. Añadir soporte multi idioma al proyecto.
17. Diseño de audio	Abr. 16 - Abr. 16	Seleccionar pistas de audio para música de fondo del menú de pausa e interfaz de victoria de juego. Elegir más efectos de sonido para las acciones de personajes como disparos, muertes y demás.
18. Implementar audio & Implementar torres -> Implementación externa	Abr. 17 - Abr. 17	Añadir la música a las IU que están siendo desarrolladas en el momento en la tarea 16, y añadir efectos de sonido varios a las animaciones de entidades.
19. Contactar con las directoras	Abr. 17 - Abr. 17	Comunicar los avances realizados.
20. Comprobar código	Abr. 19 - Abr. 19	Retocar el árbol de decisión de paneles solares para incluir tres fases distintas distinguibles.
21. Aprender a usar un motor de juego -> Aprender a decorar niveles	Abr. 24 - Abr. 24	Seguir el curso de decoración de niveles en *Unreal Engine* [26].
22. Diseño de mapas -> Diseño visual	Abr. 25 - Abr. 25	Plantear el entorno árido donde ocurren las acciones principales del juego.

23. Implementar mapas -> Implementación visual	Abr. 26 - Abr. 26	Crear el entorno planificado en la tarea anterior en el juego.
24. Contactar con la directora	May. 3 - May. 3	Informar de que el producto está cerca de su finalización.
25. Diseño de torres y enemigos	May. 3 - May. 3	Concebir la idea de variar la apariencia de los robots por estados de salud. Diseñar el escudo, la torre con más salud del juego que también requiere uso de apariencia variable según la cantidad de daños absorbidos. Diseñar el robot ocultador, un enemigo que solo puede ser dañado cuando se detiene para atacar.
26. Implementar entidades -> Implementación visual	May. 3 - May. 3	Implementar apariencias distintas para robots según los daños absorbidos.
27. Implementar entidades-> Implementación visual e interna	May. 4 - May. 5	Dar vida a la torre escudo, creando su modelo 3D en *Blender* e importándolo a *Unreal Engine*. Crear sus estados de salud visibles y animaciones usando esta última herramienta. Implementar el robot ocultador, siguiendo los mismos pasos que con la torre escudo.
28. Diseño de torres y enemigos & Diseño de interfaces	May. 10 - May. 10	Tener la idea de generar efectos especiales por explosiones e impactos de proyectiles. Diseñar una interfaz gráfica para avisar al usuario de que se va a aproximar una gran oleada de enemigos.
29. Implementación de entidades & Implementación de interfaces	May. 10 - May. 11	Implementar los elementos diseñados en el paso anterior.

Tabla 5: Tareas de la fase 2 del proyecto (parte 2)

Nota: No se han incluido las tareas de comentar código, así como comprobación de pruebas por implementaciones individuales y producto semicompleto (partes colectivas) para simplificar el cronograma. Las dos primeras se hacen junto a toda tarea de implementación, mientras que la última periódicamente (aproximadamente una vez cada 3 días de implementación realizados).

Fase 3: Perfeccionamiento y documentación (May. 17 - Jul. 14)

La fase 3 del proyecto se centra en dos pasos principales: adecuar el producto final (mantenimiento) y redactar todo el recorrido vivido (documentación).

Para mejorar el programa resultante, esta fase da más importancia a las tareas de pruebas, no concentrándose en añadir características al juego que no sean necesarias. Así se sigue la finalidad de asegurarse de que no hayan fallos y mejorar la experiencia de juego existente.

El plan de pruebas planteado se recopila en la tabla IV del anexo.

Su cronograma junto con las tareas realizadas se presenta a continuación:

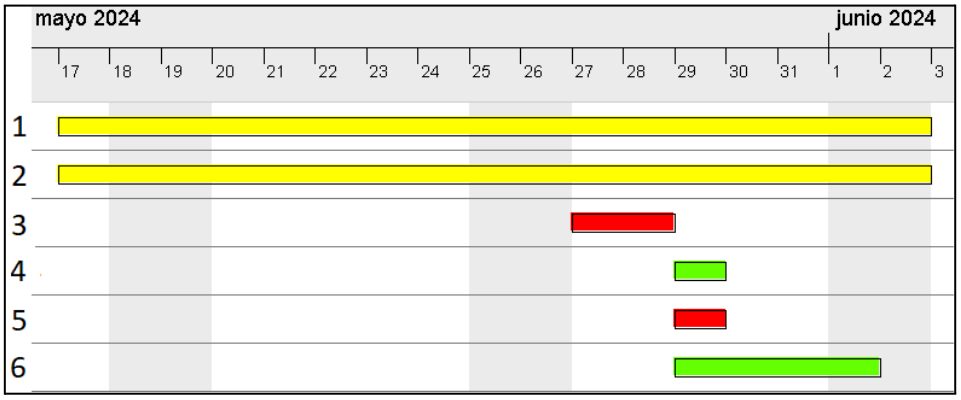


Figura 13: Cronograma de la fase 3 del proyecto (parte 1)

Tarea -> Subtarea	**Fecha inicio y fin**	**Comentarios**
1.Comprobar código	May. 17 - Jun. 2	Revisar y plantear algunos cambios en el diseño e implementación del programa. Las alteraciones más destacables son: - Modularizar más los *Behavior Trees* de los robots y torres disparadores. Esto se hace dando más *BTTasks* a cada árbol, ya que inicialmente tenían una tarea por estado general (como atacar o moverse, en el caso de los robots) y eran demasiado complejas internamente. - Diseñar *widgets* personalizados para usarlos en varias interfaces distintas a la vez. Previo a esto, cada componente repetido en distintas interfaces se creaba desde cero en cada una de ellas. Esto no es recomendable porque se pierde tiempo programando un mismo aspecto múltiples veces y reutilizar componentes es una mejor práctica. - La información interna sobre todas las

		selecciones de torretas y cantidad de energía en partida se guardaba en sus interfaces respectivas inicialmente. Se movieron estos datos a un componente interno (el mando de jugador), ya que es un diseño más coherente.
2. Pruebas -> Probar juego en completo	May. 17 - Jun. 2	Comprobar una vez más que el juego funciona y arreglar fallos inesperados. Realizar mejoras de jugabilidad, entre estas: - Modificar precios para torretas, y velocidades y cantidad de puntos de vida para robots. - Reducir tiempo de espera al principio del nivel antes de que aparezca el primer enemigo (de 50s a 26s). - Implementar lógica en los robots para que formen colas en vez de atravesarse, de manera que no se sobrepongan y oculten entre ellos en la pantalla.
3. Diseño de interfaces gráficas	May. 27 - May. 28	Diseñar y dibujar cursores propios para el juego.
4. Implementar interfaces	May. 29 - May. 29	Implementar los cursores personalizados en el juego.
5. Diseño de interfaces gráficas	May. 29- May. 29	Diseñar una interfaz de confirmación de salida del juego, pantalla de carga, y créditos.
6. Implementar interfaces	May. 29 - Jun. 1	Implementar las IU planificadas en la tarea anterior.

Tabla 6: Tareas de la fase 3 del proyecto (parte 1)

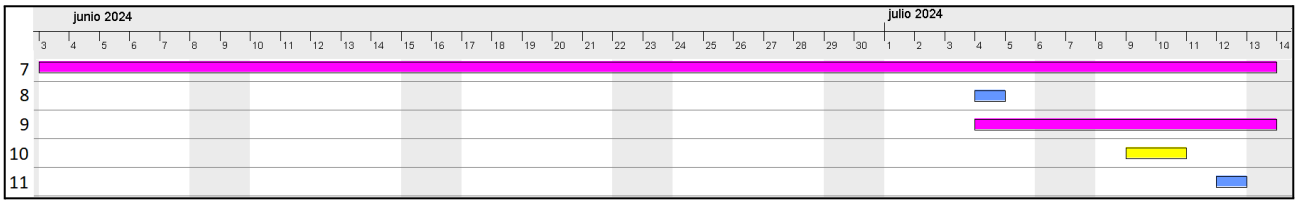


Figura 14: Cronograma de la fase 3 del proyecto (parte 2)

Tarea -> Subtarea	**Fecha inicio y fin**	**Comentarios**
7. Crear memoria	Jun. 3 - Jul. 13	Redacción y revisión de este mismo documento, además de crear las figuras y tablas relevantes.
8. Contacto con las directoras	Jul. 4 - Jul. 4	Enseñar en vivo el proyecto y el progreso de redacción de la memoria. Obtener sugerencias para cómo encarar la defensa del trabajo.
9. Crear presentación	Jul. 4 - Jul. 13	Desarrollar y preparar la presentación para la defensa del proyecto.
10. Pruebas -> Probar juego en completo	Jul. 9 - Jul. 10	Realizar pruebas finales, jugando al juego en su totalidad en búsqueda de fallos.
11. Contacto con las directoras	Jul. 12 - Jul. 12	Mostrar la presentación y obtener sugerencias.

Tabla 7: Tareas de la fase 3 del proyecto (parte 2)

8. Detalles de diseño e implementación

8.1 Fundamentos de la programación

La implementación de un programa de esta escala no es cosa sencilla y se requiere conocer algunos conceptos más técnicos sobre la programación para poder entender los siguientes apartados.

8.1.1 Programación Orientada a Objetos: clases y objetos

La creación de videojuegos en *Unreal Engine 5* se basa en la programación orientada a objetos (POO), una forma de programar. La POO no se limita a la industria de los videojuegos, también se usa en el desarrollo de otras aplicaciones *software* como pueden ser páginas web o sistemas de análisis de datos.

Este estilo de creación de programas se basa en diseñar y crear distintos componentes o **clases** que luego pueden ser materializadas en tiempo real durante la ejecución mediante **objetos.**

La **clase** es la plantilla de un componente *software*, definiendo qué debería ser capaz de hacer y cómo hacerlo. Cada clase definida se distingue mediante un nombre otorgado por su autor, soliendo referenciar su funcionalidad principal.

El **objeto** es un ejemplar que sigue la plantilla de una clase concreta. El objeto existe durante la ejecución de una aplicación y se ejecuta para hacer cumplir con las funcionalidades del programa.

Este concepto de clases es la manera en la que POO basa su **modularidad**, definiéndose como la característica en la que en la un programa se divide en varias clases o módulos. Cada uno de estos se plantea como una unidad lógica que cumple con una finalidad en concreto desde el punto de vista del programador. Agrupando y complementando todos estos componentes como es debido es la forma en la que se hace funcionar a un programa en su totalidad. [18]

POO también destaca por la **reusabilidad**. Mediante una clase se define qué y cómo hacer algo una única vez, pudiendo tener centenas de ejemplares mediante objetos. Cada una de estas instancias finales comparten la misma implementación en su clase, sin tener que requerir ser programadas una a una.

Relacionado con la reusabilidad entre instancias de una misma clase, es posible utilizar código de una misma clase en varias mediante la **herencia.** Se pueden definir clases hijas que heredan de otras, de forma que implementen las funcionalidades de sus superiores por defecto (no requiriendo su definición de nuevo). Por lo general una clase puede tener una cantidad indefinida de hijos, pero solo un único padre.

Además, se puede repetir este proceso sucesivamente ya que es transitivo, y se pueden representar las relaciones de herencia en forma de árbol familiar (conocido como un diagrama de clases). Sabiendo de antemano vía diseño qué funcionalidades pueden ser reutilizadas, se puede ahorrar una gran cantidad de código y facilitar su lectura. Crear código legible es especialmente importante en proyectos colectivos, donde es imprescindible que más de una persona lo sepa interpretar y expandir correctamente.

Cada entorno de trabajo incluye clases por defecto con distintas utilidades para crear programas, siendo conocidos como librerías o repositorios en el mundo de la programación. El programador puede hacer uso de estas clases por defecto como base de la herencia para crear las suyas propias, las cuales dan forma al programa final.

8.1.2 Cómo leer un diagrama de clases

Los diagramas de clase indican las comunicaciones que existen entre estas, de manera que representen sus coordinaciones en conjunto. Cada clase toma forma de rectángulo: si es azul, se genera en forma de objeto en algún momento durante el juego; si es blanco, se usa para agrupar y organizar funcionalidades sin instanciarse de forma directa. Las conexiones se indican mediante líneas, existiendo tres tipos de ellas:

- Heredar: “B” hereda de “A” implica que “B” incluye las funcionalidades de “A” en sí mismo. (Figura 15).

- Usar: “A” usa “B” indica que ambas clases son independientes, pero “A” puede pedir ayuda a “B” a que realice una funcionalidad suya durante la ejecución. (Figura 16).

- Relación: “A” se relaciona con “B” quiere decir que “A” o “B” es un componente fundamental del otro, incluyendo posibles cardinalidades distintas mostradas en el diagrama. En la relación de una clase A con otra B, el número situado en el lado opuesto de A indica con cuantos objetos B se relaciona una instancia de estas. Si no existe un valor asociado en alguna de las cardinalidades, se asume que es 1. (Figura 17).

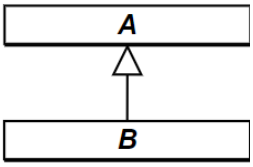


Figura 15: Heredar

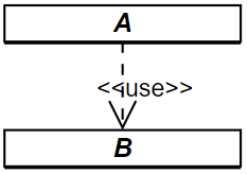


Figura 16: Usar

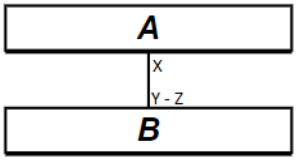


Figura 17: Relacionarse

8.2 Programando en Unreal Engine 5

8.2.1 Clases ofrecidas

Unreal Engine 5 posee una amplia librería que incluye todo tipo de clases con las que crear un juego de cero y personalizarlo al gusto del creador. Cada una de ellas implementa funcionalidades básicas especializadas y están pensadas para ser usadas en contextos específicos. Por ejemplo, se incluyen distintas para definir las normas de juego, los personajes, IA y demás. Es objetivo del programador elegir las adecuadas para su juego y definir mucha más lógica detrás de ellas mediante la herencia para hacer funcionar sus creaciones.

Las clases proporcionadas por *Unreal Engine 5* que han servido como base en este proyecto han sido las siguientes:

- **Object**: Se traduce del inglés como “objeto” y es la clase raíz en la herencia de *Unreal Engine*.

- **GameModeBase***:* Se traduce del inglés como “modo de juego base”. Se vincula un *GameModeBase* por cada mapa o entorno definido en el juego, principalmente sirviendo para definir las normas de este y qué serie de pasos se deben realizar en cada contexto de partida dado.

- **SaveGame**: Se traduce del inglés como “guardar juego”. Un *SaveGame* es una clase que permite definir información que se quiere guardar persistentemente entre reinicios del juego.

- **Actor**: Se traduce del inglés como “actor”. Un actor es cualquier elemento que se puede colocar en un mapa, pudiendo tener otro como dueño. La identidad del actor se define por los componentes que se pueden vincular a él (reproductores de sonido, material físico, campos invisibles, etc). Estos últimos se detallarán más adelante.

- **Pawn**: Se traduce del inglés como “peón”, hereda de *Actor*. Un peón es un actor que tiene capacidades de ser controlado por una IA o el propio jugador. Las clases que representan los personajes de un juego se deberían basar en esta misma.

- **Controller**: Se traduce del inglés como “mando”. Un mando es un actor que se vincula con un peón para poder ser controlado por el usuario o una IA.

- **PlayerController**: Se traduce del inglés como “mando de jugador”, hereda de *Controller*. Un mando de jugador puede ser enlazado a un peón para dar control al usuario sobre su personaje jugable, o recordar información sobre sus acciones y selecciones realizadas. Además, esta clase permite abrir y cerrar interfaces de usuario sobrepuestas en la pantalla del juego.

- **AIController**: Se traduce del inglés como “mando de IA”. Un mando de IA puede ser vinculado a un peón para que un algoritmo o IA lo controle. Los mandos de IA se suelen enlazar a su vez con árboles de decisión (o *Behavior Trees* en inglés), quienes definen la lógica de IA en cuestión.

- **ActorComponent**: Se traduce del inglés como “componente de actor”. Se trata de un objeto que puede ser vinculado a un actor para poder darle una identidad o funcionalidad propia. Un actor puede tener múltiples componentes de actor (incluso varios del mismo tipo), pero cada uno de ellos solo se vinculan a un único actor. Esto permite que los componentes se definan en jerarquías con forma de árbol vinculados a cada actor, de manera que siempre exista un único componente raíz en cada uno de estos.

- **SceneComponent**: Se traduce del inglés como “componente de escena”, hereda de ActorComponent. Un componente de escena es un objeto vinculado a un actor que tiene una posición, rotación y escala concretos (esto conocido como un *transform*) relativos a su componente padre en su jerarquía respectiva. La raíz del árbol debe ser de tipo SceneComponent o derivado, siendo el que referencia la posición, escala y rotación base del actor respecto al centro del mundo o mapa. Los *SceneComponents* también suelen ser útiles para definir puntos de aparición de otros actores, como pueden ser proyectiles.

- **PrimitiveComponent**: Se traduce del inglés como “componente primitivo”, hereda de SceneComponent. Además de poseer un *transform*, este componente incluye lógica por defecto para poder manejar colisiones y activar eventos vinculados a código customizado después. Siendo hijo indirecto de ActorComponent, permite vincularse a un actor.

- **StaticMeshComponent**: Se traduce del inglés como “componente de malla estático”, hereda de PrimitiveComponent. Este componente permite crear materia rígida visible mediante el enlace de una figura tridimensional con materiales (colores). Es posible importar figuras tridimensionales creadas en otros programas y vincularlas a estos.

- **BoxComponent**: Se traduce del inglés como “componente caja”, hereda de PrimitiveComponent. Un componente caja es un rectángulo tridimensional redimensionable invisible que permite detectar colisiones y activar eventos cuando esto ocurra.

- **CameraComponent**: Se traduce del inglés como “componente cámara”, hereda de SceneComponent. Un componente cámara sirve para definir el punto de vista de un actor. Generalmente se enlazan con el peón del jugador, para así poder darle visión en el mundo virtual del juego.

- **AudioComponent**: Se traduce del inglés como “componente de audio”, hereda de SceneComponent. Un componente de audio permite reproducir pistas de sonido en un punto concreto del nivel. Se puede definir que dicho audio se escuche desde cualquier punto del entorno al mismo volumen o que pierda intensidad según más lejos el usuario esté de su origen.

- **AmbientSound**: Se traduce del inglés como “sonido de ambiente”, hereda de Actor. Un sonido de ambiente es un actor invisible formado por un AudioComponent que permite hacer sonar música de fondo escuchable desde cualquier parte del mapa.

- **BehaviorTree**: Se traduce del inglés como “árbol de actitud o decisión”. Un *BehaviorTree* permite definir lógica de IA para enlazarla a mandos después. Estos árboles se suelen construir visualmente haciendo uso de nodos.

- **BTNode**: Se traduce del inglés como “nodo de árbol de actitud”. Un *BTNode* es un nodo cualquiera de un *Behavior Tree*.

- **BTTaskNode**: Se traduce del inglés como “nodo tarea de árbol de actitud”, hereda de *BTNode*. Una tarea es un nodo asignable a un *Behavior Tree* que define una instrucción ejecutable por el personaje asociado a dicho árbol.

- **BTService**: Se traduce del inglés como “servicio de árbol de actitud”, hereda de *BTNode*. Un servicio es un componente que se puede añadir a una tarea en un árbol. Permite realizar comprobaciones periódicamente siempre y cuando se esté ejecutando la tarea sobre la que está vinculada o una que le sigue en la jerarquía del árbol.

- **Blackboard**: Se traduce del inglés como “pizarra”. Trabaja en conjunto con un *Behavior Tree* y sirve como “la memoria” de la IA, almacenando y recordando datos relevantes del algoritmo.

- **UserWidget**: Se trata de la clase base usada para crear interfaces en *Unreal Engine 5*. Un *UserWidget* (también denominado *widget*) puede ser una ventana que cubre toda la pantalla con todo tipo de complementos o ser tan simple como un único componente insertable a otra interfaz. Ejemplos de algunos de estos últimos pueden ser texto, botones, o imágenes individuales.

La representación visual de relaciones entre estas clases se puede ver en forma de diagrama de clases en la figura 3.1.1.

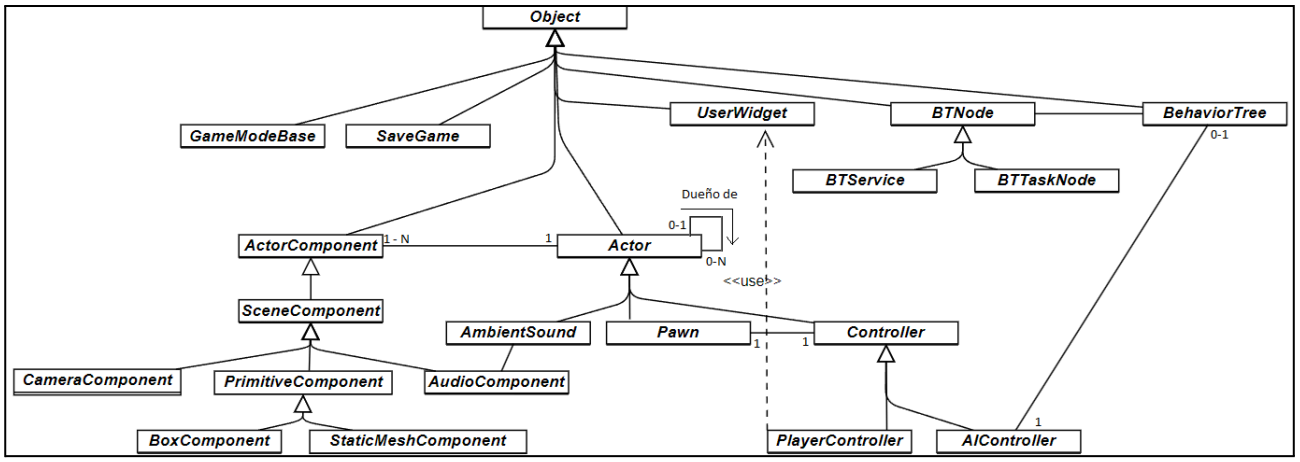


Figura 3.1.1: Diagrama de clases de clases *Unreal Engine* ya predefinidas usadas en el proyecto como base de herencia

Además, destacan las siguientes clases que aunque no se han usado como base de herencia y no sean unidades lógicas por sí mismas en el juego, complementan a las demás:

- **GameplayStatics**: Se traduce del inglés como “valores estáticos de juego”. Incluye funcionalidades muy diversas y útiles para usar en el juego. Entre estas se incluyen cargar un mapa distinto, obtener el objeto *SaveGame* asociado al archivo de guardado del jugador, y buscar referencias de actores en el mundo.

- **KismetInternationalizationLibrary**: Se traduce del inglés como “Librería de internacionalización de Kismet”. Permite obtener la lista de idiomas definidos en la aplicación y modificar la selección de la lengua actual durante la ejecución.

- **KismetStringTableLibrary y KismetTextLibrary**: Se traducen del inglés como “Librería de tablas de cadenas de caracteres y librería de texto de Kismet”. Estas clases permiten recoger la referencia de una frase que aparece en una pantalla de juego usando su clave correspondiente asociada y mostrarla en el idioma actual seleccionado.

  Esto funciona porque todo texto visible en el producto final con sus traducciones correspondientes se han vinculado a una clave en una tabla de frases. De esta forma, usando una misma clave, se pueden mostrar frases en según el idioma actual en las interfaces de juego.

  Se ofrece como ejemplo los contenidos de la tabla 8:

Clave	**Español**	**Inglés**
BContinuar	Continuar	Continue
BAjustes	Ajustes	Settings
A_0	Cañón	Cannon

Tabla 8: Representación esquemática parcial de la tabla de frases

8.2.2 Lenguajes de programación disponibles

En la mayoría de entornos de programación se diseña e implementa sobre un único lenguaje de programación en mente. *Unreal Engine 5* diverge de la norma, disponiendo de dos que usados en conjunto facilitan la experiencia de crear un juego. Las clases listadas anteriormente se permiten heredar e implementar directamente en cualquiera de los dos lenguajes de programación:

- **C++**: Uno de los lenguajes compilados más conocidos y eficientes, evolucionado de C. Se hace uso de un editor de código o IDE (como por ejemplo *Virtual Studio Code*) para programar en un lenguaje compilado, en donde se escribe en un fichero la lógica que se quiere ejecutar usando la sintaxis del lenguaje en concreto (ver figura 6.1). Una vez finalizado este proceso, se compila el archivo para traducir su contenido a lenguaje máquina (cuyo proceso puede durar varios minutos), el cual el ordenador lo entiende y usa durante la ejecución. De esta manera, no se pierde tiempo interpretando y traduciendo el código inicial escrito mientras se está jugando [19].

Las clases C++ en *Unreal* siguen un estándar de añadir un prefijo de una letra a sus nombres, pero estos son introducidos automáticamente al nombrar y crear una clase. Además, no es visible cuando el componente se usa en diseños de otros creados en editores visuales del programa. Por ejemplo, en el IDE del proyecto la clase ***U**BTTaskAccionDeEjemplo* figuraría como *BTTaskAcciónDeEjemplo* al usarla en el editor de *Behavior Trees* de *Unreal Engine*. Esta segunda nomenclatura es la que se utilizará para listar las clases C++ creadas en este proyecto en específico.

```
void AEntidad::Matar() {
    // Se llama a este método cuando vida = 0. Es la última llamada de la cadena del procesamiento de la eliminación de una en


    this->QuitarIA(); // Desactivar la IA si no se había hecho ya antes
    this->DesactivarHitbox(); // Quitar la hitbox de la entidad


    // Comprobar si la entidad tenia componente de vida y es vulnerable, si lo es, entonces animar su muerte
    // Si no cumple la condicion, la unidad se habria automatado y no tiene sentido animar esa muerte (ej, matarse tras haber

    UComponenteVida* ComponenteVida = FindComponentByClass<UComponenteVida>();

    if (ComponenteVida && ComponenteVida->EsVulnerable()) {
        this->RealizarAnimacion(0); // realizar animación de muerte (por blueprints)
        GetWorld()->GetTimerManager().SetTimer(TimerFrame, this, &AEntidad::Destruir,this->TiempoDeAnimacionDeMuerte, false);

    } else {
        this->AutoDestruir(); // Sirve para ocultar temporalmente la entidad fuera de la pantalla (para que suenen sus SFXs) h
                              // definitivamente
    }
}
```

Figura 6.1: Ejemplo de algoritmo C++ desarrollado en el proyecto

- **Blueprints**: Un lenguaje de programación visual innovador y exclusivo a *Unreal Engine*. En lugar de escribir las instrucciones por teclado en código escrito, se hace uso de bloques y conexiones visuales para crear la lógica de programación, asimilandose a un diagrama de bloques (ver figura 6.2). Se le suele denominar un lenguaje de *scripting* debido a su naturaleza de ser un lenguaje interpretado, dándole una mayor facilidad para intercambiar trozos de código en sus clases y ejecutarlas al instante sin complicaciones ni compilaciones. Esto a su vez trae un inconveniente: la traducción a lenguaje máquina se hace durante la ejecución, perdiendo rendimiento [19].

  Las clases *Blueprints* siguen el estándar de incluir prefijos con dos letras y una barra baja para indicar que han sido creadas en este lenguaje, no siendo añadidos automáticamente por el programa. Las letras especifican la clase base usada para definir el componente, siendo "WB_" y "BP_" las más usadas en este proyecto. Estos prefijos indican si su componente se trata específicamente de una interfaz de usuario (***W**idget**B**lueprint)* o no (***B**lue**P**rint)* respectivamente.

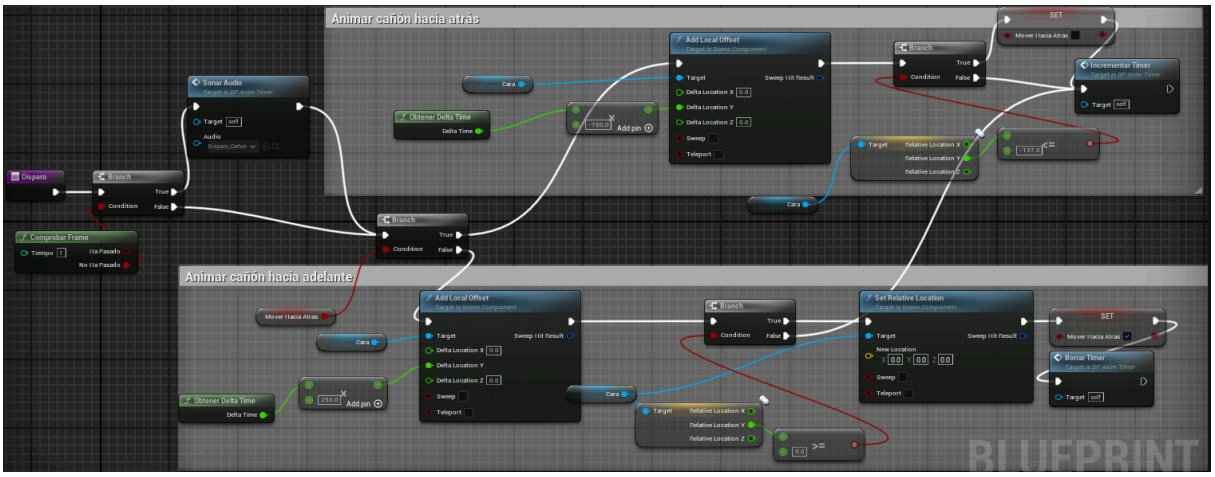


Figura 6.2: Ejemplo de algoritmo desarrollado en *Blueprints* en el proyecto

Se recomienda usar ambos lenguajes en un proyecto *Unreal*: C++ permite condensar código de manera eficiente para programar la parte más compleja, abstracta e interna del juego (como puede ser la lógica de juego, o las acciones básicas que pueden realizar los personajes). Mientras tanto, los *Blueprints* ayudan a diseñar e implementar funcionalidades sencillas de manera visual, siendo la opción ideal para crear la parte visible del juego (donde se incluyen las animaciones y la asignación de los aspectos visuales de los personajes) [20].

Siguiendo esta lógica, *Unreal Engine* está pensado para que el programador cree clases C++ base para definir las acciones y funciones internas de cada componente del juego (personajes, proyectiles, etc). Después se pueden heredar a clases *Blueprint*, donde se les pule el aspecto visual y se les dan animaciones propias a cada tipo de instancia creable en el juego. Por esta misma razón, el motor de juego permite expandir la herencia de clases C++ a *Blueprints* pero no al inverso.

Epic Games, la desarrolladora de *Unreal Engine*, incluye documentación en Internet donde se explica todo lo mencionado en estos últimos apartados junto con mucha más información del motor de juego que no es relevante para el proyecto actual [21].

8.3 Clases personalizadas creadas

En los siguientes puntos se van a explicar los componentes del juego que se han creado y sus relaciones.

Las descripciones que se van a dar a continuación pueden ser representadas gráficamente mediante diagramas de clases, siendo demasiado complejos como para ser añadidos aquí. En su lugar, se incorporan en el apartado de figuras 3.X en el anexo de la memoria.

8.3.1 Elementos usados en el menú principal

El menú principal de un juego necesita ser llamativo e intuitivo con diversas opciones y submenús. Se usan clases para proyectiles, zonas invisibles, interfaces, música de ambiente, y gestión del programa.

I. Proyectiles

La función principal de los proyectiles es dañar a los personajes del bando contrario a quien los invocó. A pesar de esto, también se usan para decorar el fondo del menú principal, donde se pueden ver moviéndose de lado a lado para añadir dinámica al entorno. Su jerarquía de tipos es la siguiente (ver figura 3.2.1 en el anexo):

- **Proyectil**: Toda instancia de proyectil tiene como raíz esta clase, y se trata de un actor formado por un StaticMeshComponent. Incluye lógica para desplazar el actor en tiempo real, indicar la dirección y sentido de desplazamiento, manejar eventos de colisiones, y contabilizar un máximo de impactos posibles antes de ser destruído. No define exactamente la forma del proyectil, animaciones de impacto, ni contra qué elementos activa eventos de colisión.

- **BP_Proyectil**: Hereda de proyectil. Se encarga de animar el golpe de un proyectil contra un personaje cuando ocurre un evento de colisión. Todos los tipos de proyectiles usan los mismos efectos especiales, por lo que toda instancia de proyectil final se hereda de esta clase. Su figura y la lógica de qué debería causar un evento de colisión se deja sin definir para que sus clases hijas lo implementen.

- **BP_BolaDeCañon**: Hereda de BP_Proyectil. Su colisión está configurada de manera que solo afecte a robots, y es una esfera gris visualmente (ver figura 1.4.1). Esta clase se emplea para crear proyectiles de cañones. Siendo un proyectil de torre, se desplaza hacia la derecha de la pantalla.

- **BP_LaserDePistola**: Hereda de BP_Proyectil. Su colisión está configurada de manera que solo afecte a robots, permita impactar hasta 3 objetivos, y es un rectángulo rojo visualmente (ver figura 1.4.2). Esta clase se emplea para crear proyectiles de pistolas láser. Siendo un proyectil de torre, se desplaza hacia la derecha de la pantalla.

- **BP_ProyectilRobot**: Hereda de BP_Proyectil. Su colisión está configurada de manera que solo afecte a torres, y son dos rectángulos rojos paralelos visualmente (ver figura 1.4.3). Este tipo de disparo se usa por todos los robots existentes en el juego. Siendo un proyectil de robot, se desplaza hacia la izquierda de la pantalla, y es el único que tiene un efecto de caída.

Figura 1.4.1: Bola de cañón

Figura 1.4.2: Rayo de pistola láser

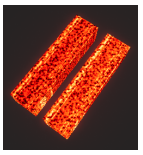


Figura 1.4.3: Proyectiles de robots

II. Áreas invisibles

Las zonas invisibles materializan proyectiles en el menú principal para hacerlo más dinámico, y luego los eliminan cuando están fuera del ángulo de la cámara para ahorrar recursos.

Las siguientes jerarquía de clases permite hacer esto realidad (ver figura 3.2.2 en el anexo):

- **ZonaMenuSpawnProyectiles**: Un actor que posee un BoxComponent como único componente, presentándose como un rectángulo invisible en el mapa. Incluye lógica en la que calcula una posición aleatoria dentro de él para indicar coordenadas donde materializar proyectiles cada cierto tiempo. No define el tipo de proyectil concreto a generar.

- **BP_ZonaSpawnProyectilesRobot**: Hereda de ZonaMenuSpawnProyectiles. Implementa lógica para materializar proyectiles de tipo BP_ProyectilRobot.

- **BP_ZonaSpawnProyectilesTorre**: Hereda de ZonaMenuSpawnProyectiles. Implementa lógica para materializar proyectiles de tipo BP_LaserDePistola y BP_BolaDeCañon.

- **BP_ZonaBlockProyectiles**: Es un actor que posee un BoxComponent como único componente. Está configurado como un rectángulo invisible que crea eventos de colisión con cualquier tipo de proyectil, de manera que desaparezca si entra en contacto con la zona. Se colocan a los lados del entorno, para que cuando un disparo ya haya pasado fuera del ángulo de vista del jugador se quite.

III. Pantallas visuales

Las pantallas visuales o interfaces son menús con los que el usuario puede interactuar; permitiendo cambiar ajustes del juego, empezar una partida, ver las normas de juego y más.

Este apartado se divide en “componentes” e “interfaces”. El primero de ellos describe *widgets* que implementan lógica concreta reutilizable en varias interfaces, mientras que el segundo explica las pantallas al completo que representan un menú por sí mismos. El diagrama de clases resultante se encuentra en la figura 3.2.3 en el anexo.

III.I Componentes

- **WB_Boton**: Se trata de una imagen de un botón futurístico con texto editable encima de esta (ver figura 1.1.1). Incluye lógica para cambiar su color y la figura del cursor personalizado del juego cuando pasa por encima. Por defecto un WB_Boton no realiza ninguna acción cuando se pincha, pero las interfaces que los incluyen pueden definir su lógica. También posee la funcionalidad de poder desactivarse e impedir que se pueda hacer click en este.

- **WB_Flecha**: Es un icono de un triángulo cuyo pico apunta hacia la izquierda o derecha (ver figura 1.1.2). Fundamentalmente es un botón e implementa las mismas funcionalidades que el componente anterior, si bien su apariencia es distinta.

- **WB_Paginas**: Se trata de un componente para permitir cambiar entre múltiples páginas de una interfaz (ver figura 1.1.3). Algunos menús no caben en una única pantalla y se necesita tener una funcionalidad adicional que permita al usuario navegar entre los distintos paneles del mismo menú.

  Este componente incluye dos WB_Flecha a sus lados y un campo de texto en el centro que indica el número de página actual que se está visualizando, junto con la cantidad de paneles disponibles que el usuario puede ver. WB_Páginas se encarga de visibilizar o invisibilizar los paneles de una interfaz según el usuario usa las flechas para navegar entre estas.

  No se permite al usuario ir más atrás del primer panel o más adelante del último, desactivando la flecha correspondiente si fuera necesario.

- **WB_Slider**: Posee una descripción, así como una barra con una circunferencia y un valor numérico asociado (ver figura 1.1.4). Este componente permite definir un número mínimo y otro máximo, de manera que se muestre por pantalla el valor asociado a la posición porcentual del círculo en el rectángulo respecto a los límites definidos. También se permite activar y desactivar interacciones con ella según sea necesario.

  Este componente es muy útil para definir opciones de juego que el usuario puede ajustar mediante un valor entre dos límites, como pueden ser el volumen de música y efectos especiales.

- **WB_SeleccionOpcion**: Está formado por un título y otro componente de texto, este último rodeado de dos WB_Flecha a sus lados. El *widget* permite al usuario navegar entre distintos ajustes predefinidos, haciendo click en las flechas correspondientes para verlos (ver figura.1.1.5).

Figura 1.1.1: WB_Boton

Figura 1.1.2: WB_Flecha

Figura 1.1.3: WB_Paginas

Figura 1.1.4: WB_Slider

Figura 1.1.5: WB_SeleccionOpcion

III.II Interfaces

- **WB_InterfazMenuPrincipal**: Es el menú principal del juego. Incluye botones personalizados para empezar un nuevo juego, continuar con la partida existente, revisar el manual del juego, acceder a los ajustes, ver los créditos, o salir del programa (ver figura 1.2.1).

- **WB_InterfazTutorial**: Es el menú de manual del programa (ver figura 1.2.2). Incluye varios paneles con texto e imágenes que explican cómo jugar y un componente WB_Páginas en la parte inferior. Gracias a este último, el usuario puede ver y navegar entre las distintas páginas del tutorial.

- **WB_MenuDeAjustes**: Es el menú de ajustes del juego (ver figura 1.2.3) y permite una gran variedad de customización:

  Incluye cuatro WB_SeleccionOpcion donde el usuario puede cambiar el idioma (español o inglés), su modo de ventana (pantalla completa o en ventana), la calidad de gráficos (bajo, medio, alto, ultra), y la resolución de pantalla si está en modo ventana (las opciones dependen del dispositivo que tiene abierto el juego).

  Se incluyen además 3 WB_Slider para ajustar el volumen de la música, efectos de sonido, y la tasa de fotogramas por segundo (FPS) del programa.

  Existe también un interruptor para activar o desactivar la sincronización vertical del juego. Con esta opción activada, se desactiva la barra que ajusta los FPS.

- **WB_InterfazCreditos**: Es la pantalla que posee los créditos (ver figura 1.2.4). Los nombres de todos los contribuidores y sus roles se pueden visualizar en las distintas páginas navegables mediante el componente WB_Páginas colocado en la parte inferior de la interfaz. En las tablas V del anexo también se detallan quiénes son estos contribuyentes.

- **WB_Aviso**: Es una pequeña interfaz con dos botones personalizados y un texto de aviso. Se usa como pantalla de confirmación para cuando el usuario quiere salir del juego (ver figura 1.2.5) o para asegurarse de que desea reiniciar su progreso en el juego y empezar una partida de cero (ver figura 1.2.6 en el anexo).

- **WB_InterfazPopUp**: Se trata de una pequeña interfaz con un botón (ver figura 1.2.7). Se emplea para felicitar al jugador por completar el juego o para informarle de que ha perdido su progreso debido a datos corruptos en los archivos de niveles del juego. El botón sirve para confirmar que se ha leído el aviso y redirigirlo al menú principal.

- **WB_Carga**: Es la pantalla de carga (ver figura 1.2.8). Está formada principalmente por una imagen de fondo, incluyendo un campo de texto en la parte inferior donde se muestra un consejo del juego elegido aleatoriamente de una lista predefinida.

Figura 1.2.1: Menú principal

Figura 1.2.2: Pantalla del tutorial

Figura 1.2.3: Pantalla de ajustes

Figura 1.2.4: Interfaz de créditos

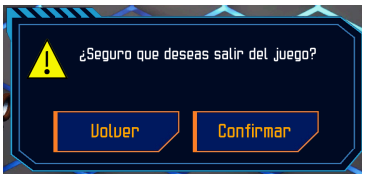


Figura 1.2.5: Pantalla de aviso con dos botones

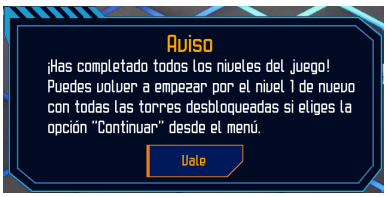


Figura 1.2.7: Pantalla de aviso con un botón

Figura 1.2.8: Pantalla de carga

IV. Música de ambiente

Se emplea la clase *AmbientSound* ofrecida por *Unreal Engine* para reproducir la música de fondo. Las únicas funcionalidades que se necesitan son hacer sonar una pista de audio por defecto y modificar su volumen, estando esto ya implementado aquí por los desarrolladores del motor de juego sin necesidad de hacer ningún cambio. Esto quiere decir que solo se requiere editar sus atributos asociados para elegir la pista de música, y programar el juego para que pida realizar el cambio de volumen según sea necesario.

V. Gestión de programa

Se necesitan clases adicionales para poder coordinar interacciones entre distintos menús (generalmente cerrar el que está visible y abrir otro). Para ello, se hacen uso de las siguientes clases personalizadas para controlar el menú principal:

- **MandoDeJugador_EnMenu**: Heredando directamente de *PlayerController*, esta clase incluye la funcionalidad de cerrar la interfaz actual y abrir otra disponible de una lista concreta sin definir.

- **BP_MandoDeJugador_EnMenu**: Heredando de MandoDeJugador_EnMenu, define la lista de interfaces disponibles. Estas ya fueron mencionadas en el apartado III.II de interfaces.

- **BP_GameMode_EnMenu**: Heredando de *GameModeBase,* esta clase es el cerebro que organiza todos los componentes usados este primer entorno: se encarga de dar la orden al mando de jugador de abrir la interfaz del menú principal cuando se carga el juego. También sirve de intermediario entre el mando y las clases de interfaces gráficas cuando se desea cambiar de pantalla visual. Además, incluye otras capacidades como cerrar el juego y cargar el otro mapa donde ocurren las partidas.

- **BP_PlayerPawn_EnMenu**: Hereda de *Pawn*. Incluye un *CameraComponent,* quien da visión al usuario. Además, está enlazado con el mando de jugador mencionado en este mismo apartado para que el jugador pueda ver e interactuar con las interfaces.

Estas interacciones que hacen funcionar el menú se pueden ver gráficamente en la figura 3.2.4 en el anexo.

8.3.2 Componentes usados para el juego en sí

Un programa no es un juego sin componentes que definen el juego en general. Se incluyen clases para definir el escenario, música de ambiente, pantallas interactivas, personajes, proyectiles, visión del jugador, y gestión de la partida. Ellas dan la identidad al juego y lo hacen funcionar, estando muchas colocadas en el mapa y ejecutándose activamente durante la partida.

I. Escenario

Si bien es cierto que la mayoría del escenario solo sirve como decoración visual, algunos componentes son imprescindibles para que el juego funcione, ya que pueden interactuar de forma activa con otras clases del juego y el jugador. Estos son:

I.I Casillas

Las casillas son los espacios en los que el usuario puede situar sus torres para defender su terreno. Cada casilla sólo puede alojar una torre, pero pueden ser destruidas por robots o quitadas por el propio jugador para hacer espacio. La siguiente jerarquía de clases hace esto posible (ver figura 3.3.1 en el anexo):

- **Casilla**: Es un actor con varios componentes que le permiten dar materia propia, así como crear efectos de sonido. Esta clase define la lógica de enlace de referencias de torres con ella, de manera que pueda saber en todo momento qué torreta está situada sobre ella. También permite eliminar estos vínculos (por si su torreta asociada deja de existir). Incluye funcionalidades extra como verificar si un robot la está pisando.

- **BP_Casilla**: Hereda de casilla. Se encarga de crear las torretas que su clase padre enlaza internamente después. Además, implementa animaciones como iluminarse de otro color (si un panel ha generado energía sobre ella) o crear partículas de explosión (si se acaba de usar la TNT sobre esta misma para eliminar su torreta relacionada).

- **BP_CasillaOdd y BP_CasillaEven**: Heredan de BP_Casilla. Definen el color base y de iluminación del espacio. Cada una de ellas usa tonos diferentes.

- **BP_SueloDeCasillas**: Es un actor que agrupa 45 BP_CasillaOdd y BP_CasillaEven en una formación 5 filas por 9 columnas. Estas se distribuyen en forma de tablero de ajedrez para poder diferenciar sus fronteras. Se sitúa en el centro de la zona de juego, y aquí el jugador coloca sus torretas y los robots circulan (ver figura 18).

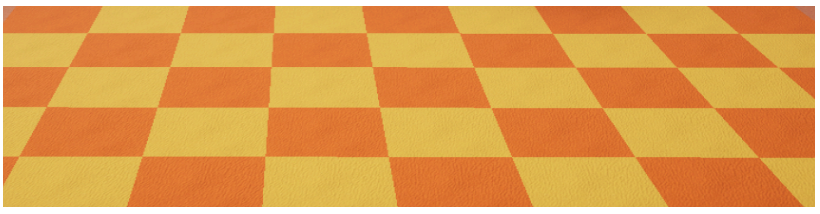


Figura 18: Suelo de casillas

I.II Áreas invisibles

Se hace uso de zonas invisibles para gestionar eventos de juego, siendo estas las siguientes (ver figura 3.3.2 en el anexo):

- **ZonaSpawnRobot**: Un actor que posee un BoxComponent como único componente. Implementa la funcionalidad de calcular las posiciones donde materializar robots en el nivel, dado el tipo concreto a generar. Esta zona incluye también funciones para decidir en qué fila horizontal situar el enemigo, así como a qué distancia. De esta forma, se puede evitar que nuevos robots aparezcan encima de otros ya existentes y se solapen.

- **BP_ZonaSpawnRobotEnPartida**: Hereda de ZonaSpawnRobot. Dada la instrucción de hacer aparecer un robot y sus coordenadas por parte de su padre, esta clase lo materializa en el mundo orientándose de cara a la mina del jugador. Existe una instancia de esta clase situada a la derecha del suelo de casillas del nivel, siendo el lugar donde aparecen los enemigos en una partida.

- **ZonaSpawnRobotPreview**: Un actor que posee un BoxComponent como único componente. Dados los tipos de robot a hacer aparecer y sus cantidades, calcula las posiciones donde colocarlos para que aparezcan en líneas de 3 en 3.

- **BP_ZonaSpawnPreviewRobot**. Hereda de ZonaSpawnRobotPreview. Usando las coordenadas de su clase padre, materializa los robots en el mundo. Esta zona se coloca al fondo de la estructura desde donde los adversarios aparecen durante la fase de selección de torres del nivel, que ocurre antes de empezar la partida. Gracias al diseño de formaciones planteado en su clase padre, los robots aparecen de forma ordenada y organizada.

- **ZonaTargetRobot**: Un actor que posee un BoxComponent como único componente. Implementa el evento de colisión que causa la derrota del usuario si un robot entra en ella.

- **BP_ZonaTargetRobot**: Hereda de ZonaTargetRobot. Implementa la configuración de colisiones correspondiente para que un robot pueda activar un evento de colisión con la zona. Se sitúa a la izquierda del mapa, entre la mina del jugador y el suelo de casillas, aunque no es visible al jugador.

- **BP_ZonaBlockProyectiles**: Esta clase también se usa en el menú principal. En este mapa en concreto existen varios de ellos a los extremos del suelo de casillas, de manera que los proyectiles que se escapen del área de juego desaparezcan.

II. Música de ambiente

La música ayuda a complementar a un juego dándole una sensación de atmósfera propia. Se necesita un reproductor de audio propio debido a que las funcionalidades por defecto otorgadas por el motor de juego no son suficientes. Para esto, se hace uso de la siguiente clase heredada (ver figura 3.3.3 en el anexo):

- **BP_Musica_EnPartida**: Hereda de *AmbientSound*. Almacena las referencias a todos los archivos de música necesarios para la partida e incluye procedimientos con los que se permite elegir el audio a reproducir, detenerlo, o modificar su volumen. Junto con el componente de audio por defecto que se proporciona en *AmbientSound*, esta clase incluye uno adicional para reproducir un efecto de sonido junto con música de manera simultánea.

III. Pantallas visuales

Igual que en el caso del menú principal, existen interfaces de usuario y componentes customizados que se añaden a ellas para poder generar las pantallas necesarias durante una partida (ver figura 3.3.4 en el anexo):

III.I Componentes

- **WB_BotonPausa**: Similar a WB_Boton, pero tiene un diseño visual distinto e invoca el menú de pausa al ser pulsado (ver figura 1.1.6).

- **WB_TorreCard**: Tiene forma de rectángulo, mostrando la imagen de una torre en un fondo oscuro (ver figura 1.1.7). Además de la apariencia base, se incluyen filtros para dar un tono verde o rojizo a la tarjeta, así como una capa gris que se elimina gradualmente durante el tiempo. El recuadro es interactuable, de manera que la interfaz que lo posee puede definir qué lógica debería ejecutarse al pinchar sobre él. También incluye funcionalidades extra como mover la tarjeta de sitio dadas unas coordenadas y un tiempo de desplazamiento concretos, o regenerar su capa grisácea.

- **WB_WidgetEnergia**: Incluye un icono de energía, y posee un fondo azul junto con un contador en su parte inferior (ver figura 1.1.8). Este *widget* sirve para indicar de cuánta energía (recursos) dispone el jugador en un momento dado de una partida, siendo editable el valor mostrado.

- **WB_BarraDeProgreso**: Es el *widget* que indica en qué nivel se encuentra el jugador, así como el porcentaje de oleadas ya invocadas en el nivel. Está formada por una barra progresable, texto y diversas imágenes (ver figura 1.1.9). Se incluye un icono de un robot en la posición donde el progreso de la barra se encuentra, y banderas en los puntos porcentuales donde aparecen grandes oleadas. Dichas banderas se sitúan a la misma altitud que la barra en sí al principio, pero se elevan cuando su horda relativa es invocada.

  Incluye la función de avanzar la barra de manera gradual en incrementos de una oleada. Para hacer esto, se anima el movimiento del icono robot sobre ella y se alza la bandera respectiva (en su caso).

- **WB_AvisoOleada**: Es la pantalla de aviso que aparece cuando se aproxima una gran oleada, o para indicar que está ocurriendo la generación de la última horda del nivel. Incluye símbolos de aviso amarillos en forma de triángulo a sus lados y un campo de texto que indica lo que está ocurriendo (ver figura 1.1.10).

Figura 1.1.6: WB_BotonPausa

Figura 1.1.7: WB_TorreCard

Figura 1.1.8: WB_WidgetEnergia

Figura 1.1.9: WB_BarraDeProgreso

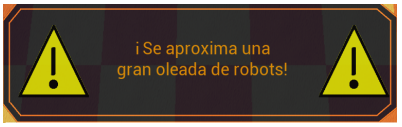


Figura 1.1.10: WB_AvisoOleada

III.II Interfaces

- **WB_InterfazSeleccion**: Es la pantalla que permite al jugador elegir sus torres antes de empezar el nivel. Se forma principalmente por dos imágenes: una ocupa la parte superior y otra la inferior de la pantalla (ver figura 1.2.9).

  Cada una incluye una lista de WB_TorreCards, cuyas longitudes pueden sumar hasta 8 en total (la cantidad de tipos de torres en el juego). La parte superior contiene el listado de torretas que el usuario quiere llevar al nivel, mientras que la inferior almacena las restantes sin seleccionar. El jugador puede pinchar sobre las tarjetas para realizar sus selecciones, moviéndose de conjunto al hacerlo.

- **WB_InterfazCuentaAtras**: Es una pantalla formada por una única imagen. En esta interfaz aparece la cuenta atrás (“3”, “2”, “1”, “Defiende”) que se realiza justo antes de empezar un nivel (ver figura 1.2.10).

- **WB_MenuDePausa**: Es la interfaz que acompaña al usuario cuando se pausa el juego (ver figura 1.2.11). Esta pantalla se forma por dos WB_Slider para permitir la alteración del volumen de música y efectos de sonido de manera independiente. Junto con estos, existen tres WB_Boton en la parte inferior para continuar la partida, reiniciar el nivel, o volver al menú principal.

  En adición a esto, cuando el jugador abre el menú de pausa se reproduce automáticamente una pista de música explícitamente usada para esta pantalla.

- **WB_InterfazEnPartida**: Es la interfaz que es visible e interactuable cuando el jugador está jugando un nivel (ver figura 1.1.11). Sus componentes principales son seis WB_TorreCard sobre una imagen de fondo (permiten al usuario elegir un tipo de torreta a colocar después), un WB_WidgetEnergia (muestra de cuánta energía dispone el jugador), y un WB_TorreCard adicional (habilita la selección de la TNT).

Además, incluye tres componentes secundarios: un WB_BarraDeProgreso (en esquina inferior izquierda), un WB_BotonPausa (en esquina superior derecha), y un BP_AvisoOleada (en el centro).

- **WB_InterfazVictoria**: Es la interfaz que se carga cuando el jugador elimina el último enemigo del nivel. Su WB_TorreCard es el componente más importante, el cual adopta la apariencia de la torre que se ha desbloqueado por superar el nivel (en su defecto será un trofeo o una nota, dependiendo del nivel en cuestión). La tarjeta aparece en el lugar donde se destruyó el último robot del nivel.

  La recompensa se desplaza hacia el centro de la pantalla cuando el usuario pincha sobre ella, visibilizando a su vez el resto de la interfaz. Esta incluye una imagen de fondo, el nombre de la recompensa, su descripción, y un botón para pasar al siguiente nivel (ver figura 1.2.12).

- **WB_InterfazDerrota**: Al contrario de la pantalla anterior, esta es la interfaz que se carga cuando el usuario pierde la partida. Sus componentes principales son un campo de texto donde se describe la causa de la derrota, así como dos botones personalizados: uno para reintentar el nivel y otro para volver al menú principal (ver figura 1.2.13).

Figura 1.2.9: Interfaz de selección

Figura 1.2.10: Cuenta atrás de partida

Figura 1.2.11: Menú de pausa

Figura 1.1.11: Parte superior de WB_InterfazEnPartida

Figura 1.2.12: Pantalla de victoria de nivel

Figura 1.2.13: Pantalla de derrota de nivel

IV. Entidades (personajes) y sus componentes

IV. I Entidades

Las entidades son los personajes del juego en sí. Su existencia en la partida es imperativa, ya que son la razón por la que la trama del videojuego sucede. Existen una gran diversidad de tipos de entidades en el juego, cuyas apariencias e información está listada en la tabla II del anexo.

Existe una jerarquía extensa de clases que incluye todas las torres y enemigos del juego (ver figura 3.3.5 en el anexo), con el propósito de agrupar y reutilizar funcionalidades según su tipo. Las clases son las siguientes:

- **Entidad**: Hereda de *Pawn* y es la base de toda torre y robot. Entre sus funcionalidades básicas está hacer desaparecer el personaje al morir. Además, todo tipo de entidad final tiene un identificador numérico propio, y según este, se asigna la IA adecuada a la instancia cuando se genera.

  Durante las acciones de un personaje, esta clase y sus derivadas permiten comandar la ejecución de animaciones concretas siguiendo un protocolo en el que cada tipo de animación tiene un valor numérico arbitrario asignado. La clase entidad en sí solo tiene la opción de querer animar su muerte (con identificador 0), pero las clases hijas pueden expandir esta lista según los distintos tipos de animaciones que requieran. De esta forma, cuando se les da forma física a cada torreta y robot al final de la cadena de herencia, se permite implementar un mismo tipo de animación de forma distinta según su apariencia pero manteniendo el mismo código que pide que se realice por detrás.

  Entidad posee los componentes básicos que todo personaje necesita: un *StaticMeshComponent* (su forma física), un *AudioComponent* (reproducir de sonido en animaciones), y un *BoxComponent* (manejar eventos de colisión con proyectiles y otros actores para procesar daños).

- **Robot**: Hereda de Entidad y es la clase base de todo enemigo. Sus funcionalidades más destacables son desplazarse hacia la izquierda o de frente, y crear colas con otros robots al colisionar (conocen las referencias a sus entidades vecinas y se comunican entre ellas para coordinar sus movimientos y ataques). La velocidad de desplazamiento es editable según la instancia de robot final creada.

  Todo enemigo incluye dos acciones de animación extra que una entidad por defecto no posee: activar su animación de desplazamiento en bucle, y desactivarla.

  Además de los componentes de su clase padre, incluye un *StaticMeshComponent* adicional para poder dar forma a sus ruedas y animarlas por separado.

- **Robot_Basico**: Hereda de Robot e implementa la habilidad de poder encontrar torretas en su rango de visión y atacarlas. Además, la frecuencia de ataque es editable por cada instancia final que herede de esta clase.

Robot_Básico incluye un identificador de animación extra que pide que se anime un disparo.

Esta clase incluye componentes adicionales: un *SceneComponent (*determina el origen de sus proyectiles), y un *StaticMeshComponent* extra (representa su cara, la cual se anima por separado de su cuerpo durante disparos).

Todos los enemigos del juego también implementan está clase, ya que tienen la capacidad de atacar. Se decidió crear esta clase y diferenciarla de "Robot" porque se había considerado la posibilidad de crear algunos enemigos que podrían desplazarse pero no atacar de forma directa.

- **Robot_Bomba**: Hereda de Robot_Basico e implementa la capacidad de explotar. Para esto, incluye funciones para crear una explosión cuando la entidad debe morir. En este proceso se detiene, daña torretas cercanas y se autodestruye usando la funcionalidad de morir implementada en Entidad.

  Robot_Bomba añade un identificador extra de animación para animar la detonación.

  Esta clase incluye un componente adicional para poder hacer funcionar la lógica de detonación: un *BoxComponent* que cubre el rango de explosión.

- **Robot_BombaRadar**: Hereda de Robot_Bomba y añade la funcionalidad de poder comprobar cuántas torretas hay en el área de explosión en un momento dado. Si hay suficientes, la entidad se muere automáticamente ejecutando la lógica de explosión de Robot_Bomba. Si no se fueran a encontrar suficientes torres antes de perder toda su vida, la entidad estalla igualmente usando la implementación de su clase padre directa.

  Robot_BombaRadar añade un identificador de animación extra para dar la orden de empezar a girar las hélices de su radar en su apariencia visual, pero no requiere componentes adicionales.

- **Robot_Ocultador**: Hereda de Robot_Basico e implementa la habilidad de poder ocultar y desocultar. Mientras esté ocultado, el personaje no sufre daños.

  Con el propósito de hacer visualmente coherente esta acción, se incluyen dos identificadores de animaciones extra para sus habilidades adicionales respectivas.

  Esta clase no requiere componentes nuevos.

- **BP_RobotBasico**: Hereda de Robot_Basico y agrupa componentes extra para la gestión de su salud e implementación de animaciones que todo robot corriente necesita (desplazar, deteriorar, disparar, morir). En futuros apartados se detallan estos nuevos componentes usados por los personajes (IV.II Vida y IV.III Animaciones).

- **BP_RobotSimple, BP_RobotMedio, BP_RobotDuro, BP_RobotLider**: Heredan de BP_RobotBasico y son las clases que se usan para instanciar cuatro tipos de robots en los niveles, teniendo el mismo aspecto físico (un rectángulo con una cara y ruedas). Se diferencian por su color, la cantidad de vida, y velocidad de ataque y movimiento. Estos son atributos que se pueden modificar directamente heredando la lógica y componentes de BP_RobotBasico, ejecutando así el mismo algoritmo todos.

  La única diferencia destacable entre ellos es que BP_RobotLider incluye un componente de animación extra para rotar la bandera de su cabeza.

- **BP_RobotBomba**: Hereda de Robot_Bomba y es la clase usada para instanciar robots bomba en el nivel. Incluye un componente de vida, y componentes de animación para las cuatro acciones básicas de todo robot junto con otro extra para animar su explosión. Visualmente tiene aspecto de bomba con una mecha encima y un par de ruedas en su parte inferior.

- **BP_RobotBombaRadar**: Hereda de Robot_BombaRadar y es la clase usada para instanciar robots bomba radar en el nivel. Incluye un componente de vida, y los mismos componentes de animación que la clase anterior pero incluyendo uno adicional para rotar las hélices de su radar. Visualmente es parecido a un robot bomba, poseyendo un radar con 3 hélices en vez de una mecha.

- **BP_RobotOcultador**: Hereda de Robot_Ocultador y es la clase usada para instanciar robots ocultadores en el nivel. Incluye un componente de vida, y componentes de animación para las cuatro acciones básicas de todo robot junto otro adicional para manejar sus escudos con los que se oculta y desoculta. Visualmente es un cuadrado con una cara y tiene dos grandes semiesferas metálicas a sus lados, que sirven de escudo para bloquear proyectiles y rota sobre ellos para desplazarse.

- **Torre**: Hereda de Entidad y es la clase padre de toda torreta. Una torre incluye como única funcionalidad adicional desvincularse de la casilla que ocupa justo antes de morir.

- **Torre_Disparador**: Hereda de Torre y toda torreta con capacidades de disparo posee esta clase. Implementa las acciones de comprobar si tiene enemigos en su fila y disparar proyectiles. Además, se permite modificar su cadencia y el tipo de proyectil usado según la instancia final diseñada.

  Incluye dos identificadores de animaciones para apuntar y despuntar. Junto con esto, existen más para animar cada uno de los disparos realizados en un ciclo de ataque. Un ciclo de ataque incluye tantos disparos como cadencia tenga la torreta.

  Contiene un *SceneComponent* adicional para indicar la zona de aparición de los proyectiles.

- **Torre_Producidor**: Hereda de Torre y toda torreta que resembla un panel solar implementa esta clase. Contiene funciones para producir energía, permitir que el jugador la recoja, y preparar el siguiente ciclo de producción.

  En adición a los identificadores de animaciones heredadas, esta clase incluye dos extra: encender y apagar paneles.

  Además, se ha añadido un *StaticMeshComponent* para animar los paneles independientemente del resto del cuerpo.

- **Torre_Muro**: Hereda de torre y es implementado por la torre escudo. Esta clase solo añade la funcionalidad de empezar a realizar una animación cuando la torreta se sitúa (rotar sus escudos), y pararla (dejar de rotar sus escudos) cuando vaya a morir. Se incluyen dos identificadores de animaciones extra para hacer esto posible, no teniendo componentes adicionales.

- **Torre_UsoUnico**: Hereda de Torre y toda torreta que pueda estallar hereda de esta clase. Sus habilidades implementadas incluyen dañar robots en su radio de explosión y eliminarse (matarse) usando la lógica creada en sus clases padre tras esto.

  Torre_UsoUnico posee un identificador de animaciones adicional para visualizar su detonación.

  Existe un *BoxComponent* extra en estas entidades que permite determinar el área de explosión.

- **Torre_Mina**: Hereda de Torre_UsoUnico y es usado por la mina. Esta clase implementa una función para activar la torreta, en la que pasa a un estado invulnerable y preparado para detonar cuando un enemigo pasa cerca de ella. Para detonar, usa la lógica de explosión de la clase anterior.

  La mina en sí debe realizar una animación visual para indicar que se ha activado, por lo que existe un identificador de animación extra aquí para hacer eso posible.

  Esta clase tiene un *BoxComponent* adicional para manejar colisiones con robots, y por lo tanto, detectar si alguien la está pisando.

- **BP_Cañon, BP_CañonDoble y BP_PistolaLase**r: Heredan de Torre_Disparador y son las torretas que el usuario coloca en casillas que disparan proyectiles a enemigos en su fila. Se diferencian por su aspecto físico, la cadencia de disparo, el tipo de proyectil disparado, y sus animaciones. Cada clase incluye los componentes de animación necesarios para atacar y morir, y un componente de vida para contabilizar daños.

- **BP_PanelSolar y BP_PanelSolarDoble**: Heredan de Torre_Producidor y son las torretas que el jugador coloca en casillas para producir más recursos. Se diferencian por su aspecto físico y la cantidad de energía generada en cada ciclo de producción. Incluye un componente de animación que ilumina los paneles cuando tienen energía preparada para ser recolectada, y otro para morir. Tienen un componente de vida para contabilizar daños.

- **BP_Bomba**: Hereda de Torre_UsoUnico y es la bomba que el jugador coloca en casillas para eliminar enemigos con una gran explosión. Incluye componentes de animación para animar su detonación, pero no incluye ningún componente de vida. Esto se debe a que las bombas son invulnerables y se eliminan automáticamente tras su explosión.

- **BP_Mina**: Hereda de Torre_Mina y es una de las torretas que el usuario coloca en casillas en casillas. Incluye componentes de animación variados para visualizar su preparación, explosión y muerte. Incluye un componente de vida para contabilizar daños.

- **BP_Escudo**: Hereda de Torre_Muro y es una de las torretas que el usuario coloca en casillas. Posee componentes de animación variados para rotar sus paneles escudo, mostrar deterioros y morir. Tiene un componente de vida para contabilizar daños.

IV.II Vida

Los personajes en un juego *Tower Defense,* en este caso las torres y los robots, deben poder tener alguna manera de definir cómo de resistentes son y cuánto daño pueden absorber antes de ser destruidos. La vida de una entidad se almacena en un componente diferenciado del resto de la lógica de los personajes en sí. Siguiendo esta filosofía, se puede utilizar una misma implementación de gestión de salud en múltiples seres distintos.

Se han creado los siguientes tipos de contabilizadores de salud para personajes (ver figura 3.3.6 en el anexo):

- **ComponenteVida**: Hereda de *ActorComponent*. Esta clase define el máximo de vida que puede tener una entidad, así como su cantidad actual. Las funciones principales procesan el daño absorbido (incluyendo la opción de matar la entidad que posee dicho componente si pierde todos sus puntos de vida), además de activar o desactivar su invencibilidad.

  Se usa en todas las torres a excepción del escudo, quienes requieren tener una cantidad limitada de salud pero no necesitan actualizar su apariencia respecto a daños recibidos (poseen tan poca salud máxima que es irrelevante crear distintas apariciones visuales en relación con su salud restante).

- **ComponenteVidaConEstados**: Hereda de ComponenteVida. Además de implementar todo lo anterior, incluye la funcionalidad de accionar ciertos tipos de animaciones en entidades cuando pasan por porcentajes de daños totales definidos, de manera que se pueda actualizar la apariencia visual del personaje respecto al castigo total recibido.

  Se usa únicamente en la torre escudo, la cual cambia de aspecto visual según los daños recibidos.

- **ComponenteVidaDeRobots**: Hereda de ComponenteVidaConEstados. Esta clase añade la funcionalidad de informar del daño recibido a la clase que arbitra la partida (*GameMode_EnPartida*).

  Se enlaza a robots, ya que además de mostrar daños visuales, también informan al núcleo del juego de la cantidad de vida que han perdido o incluso si han fallecido. Estos datos son fundamentales para poder hacer funcionar el juego, explicados en el apartado de gestión de partida (VII).

IV.III Animaciones

Las animaciones de personajes ayudan a dar más vida al juego e identificar las acciones que las entidades están realizando. Estas se incorporan en componentes externos que se enlazan a los personajes después, de esta manera que se puedan utilizar en distintos tipos de entidades a la vez. Las animaciones suelen alterar el tamaño, rotación, posición, y/o color de partes de su aspecto físico, además de tener la capacidad de producir algún efecto de sonido en relación con lo que está ocurriendo.

La jerarquía de los tipos de componentes en cuestión es la siguiente (ver figura 3.3.7 en el anexo):

- **BP_Anim_Timer**: Hereda de *ActorComponent* y es el padre de todo componente de animación. Esta clase se usa como frontera con las entidades para poder elegir el tipo de animación en concreto que se desea ejecutar según el componente concreto, así como ofrecer a estos últimos los recursos necesarios para poder implementarlas.

  Las funciones que realizan animaciones deben ser usadas múltiples veces por segundo. En cada ocasión, la apariencia de la entidad se modifica ligeramente para poder dar un efecto de fluidez. Por esta razón, BP_Anim_Timer incluye un temporizador y contador interno que sus hijas pueden utilizar para poder conocer en qué punto de animación se encuentran en todo momento. Además, existe un procedimiento adicional para hacer sonar efectos de sonido.

  El resto de componentes listados a continuación heredan directamente de esta misma clase e implementan la lógica de las animaciones en sí. Estas clases piden a sus entidades respectivas los componentes necesarios para poder modificarlos en sus animaciones (siendo estos *StaticMeshComponents y AudioComponents*, que gestionan el aspecto físico y reproductor de audio de la entidad respectivamente).

- **BP_Anim_BombaDetonar**: Dado el cuerpo de una bomba, anima la detonación y explosión de esta. Durante la detonación, altera el color del cuerpo entre rojo y negro, así como su tamaño lígeramente. Al explotar, se emiten partículas de explosión, cuyos tamaños son editables según la entidad que lo use. También se reproducen efectos de sonido durante todo el proceso.

- **BP_Anim_CaraCañon**: Dado el tubo de un cañón, realiza sus animaciones de apuntado, desapuntado y disparo. Para desapuntar, se inclina hacia abajo de forma que tenga un ángulo de 30 grados con el suelo. Para apuntar, se eleva hasta que forma un ángulo de 90 grados con este. Para disparar, se mueve rápidamente hacia atrás para dar la sensación de inercia del disparo, seguido de un movimiento más lento hacia adelante para recolocarlo en su sitio de nuevo tras el ataque. En esta última acción mencionada, se reproduce un efecto de sonido de disparo.

- **BP_Anim_CaraLaser**: Dada una cara de robot o la parte delantera de una pistola láser, realiza sus animaciones de disparo correspondientes. Para esto, se cambia el color del componente brevemente y se hace sonar un efecto de sonido. Se incluyen animaciones de apuntado y desapuntado que solo la pistola láser utiliza. Para desapuntar, se altera el color de su punta a gris y se desplaza hacia atrás lentamente, ocultándose detrás del cuerpo del personaje. Para apuntar, se invierte el proceso pero realizándose de forma veloz.

- **BP_Anim_DeterioroBomba**: Anima los estados de deterioro de un robot bomba. Para ello, recibe la mecha de este y la desplaza hacia abajo cada vez que se le da la orden de pasar al siguiente estado de salud. De esta manera, se da la impresión de contra más daño absorba, más cerca está de detonar.

- **BP_Anim_DeterioroRobotBásico**: Anima los estados de deterioro de un robot simple, medio, duro o líder. En cada caso, recibe el cuerpo del robot en sí y los colores que se les debe aplicar en cada estado de salud. Así, cuando la entidad absorba muchos daños, empieza a cambiar de color y aparenta estar más dañada.

- **BP_Anim_DeterioroRobotOcultador**: Anima los estados de deterioro de un robot ocultador. Esencialmente hace lo mismo que el componente anterior, pero requiere más *StaticMeshComponents* para funcionar (visibiliza daños en tanto en sus escudos como el cuerpo del robot, los cuales están diferenciados en trozos distintos).

- **BP_Anim_DeterioroTripleSpinner**: Anima los estados de deterioro de una torre escudo y un robot bomba radar. Al instanciarse, el robot posee hélices y la torre paneles escudo que rotan alrededor de su base. Cuando bajan un estado de salud, pierden uno de sus trozos y los restantes giran más despacio. Recibe las referencias de las tres materias rotantes iniciales para poder realizar estos cambios y hace uso del siguiente componente para complementarse.

- **BP_Anim_RotarComponente**: Es el componente usado para realizar las rotaciones mencionadas en la clase anterior.

- **BP_Anim_MinaPreparada**: Realiza la animación de activación de una mina. Para esto, el componente recibe la materia que forma el centro de esta, la cual cambia de color y se mueve hacia arriba cuando esté preparada para poder explotar.

- **BP_Anim_MoverRuedas**: Anima la rotación de escudos de un robot ocultador o las ruedas del resto de robots cuando se mueven. Recibe la referencia del elemento a rotar, junto con su radio y la velocidad de desplazamiento deseada. Así, se puede sincronizar la velocidad angular del componente con su lineal.

- **BP_Anim_Paneles**: Se encarga de encender y apagar los paneles solares visualmente. Para este propósito, recibe la referencia de la figura y cambia el color de los paneles a rojo cuando se encienden y a azul al apagarse. Incluye efectos de sonido exclusivos para encenderse, apagarse y mantenerse encendido.

- **BP_Anim_Protectores**: Anima la acción de ocultar y desocultar del robot ocultador. Al esconderse, los escudos semiesféricos empiezan separados, los cuales se acercan de manera que encajen y formen una esfera. Además, se encoge la cabeza del robot para que quepa dentro de estos. Al revelarse, se realiza el proceso contrario. Se necesitan las referencias de la cabeza y escudos del robot para funcionar.

- **BP_AnimMuerte_XXX**: Realiza la animación y sonido de muerte de “XXX”, donde “XXX” es el nombre de la entidad. Cada una de estas es exclusiva a la entidad en concreto, por lo que usan una clase distinta a la que dan referencias de partes concretas suyas para poder animarlos. Esto excluye a los robots simple, medio, duro y líder, quienes comparten la misma animación de muerte y componente que la ejecuta.

IV.IV Inteligencia Artificial

Es necesario tener un órgano en cada personaje que decida qué hacer en cada situación. La inteligencia artificial (IA) se encarga de esto, siendo el cerebro de toda entidad. La IA se incorpora en la clase *AIController*, poseyendo un *Behavior Tree* y *Blackboard*.

Los *Blackboards* son la memoria del cerebro y pueden almacenar distintos tipos de datos en variables que pueden ser usadas después. Estos datos siempre son valores binarios en este proyecto en concreto (es decir, cada variable sólo puede tomar uno de dos posibles valores en un momento dado), aunque podrían almacenar otros tipos de datos en contextos de IA distintos a los planteados aquí.

Los *Behavior Trees* toman apariencia de árboles de decisión y ejecutan la lógica establecida en ellos. Se forman por nodos, los cuales pueden ser de distintos tipos: *BTTasks,* que deben ser definidas por el programador; así como secuenciadores y selectores, ofrecidos ya por el motor de juego.

Las acciones que una entidad puede realizar se enlazan mediante *BTTasks,* y estas representan las tareas que pueden ser ejecutadas en la lógica del árbol de IA.

Los selectores permiten crear bifurcaciones en el árbol, donde solo se permite seguir una de sus ramas según condiciones concretas. Los secuenciadores hacen lo contrario: agrupan y ejecutan en orden todos los nodos que cuelgan de ellos antes de ceder el hilo del programa al siguiente nodo.

Para poder establecer las condiciones sobre bifurcaciones y almacenar sus decisiones en el *Blackboard*, se hace uso de *BTServices* en conjunto con decoradores.

Los decoradores se pueden añadir en tareas para definir condiciones sobre ellas, y suelen ser usados sobre secuenciadores para afectar a múltiples nodos a la vez. Existe una categoría de decorador con la capacidad de impedir que se ejecute su nodo correspondiente (o rama entera en caso de enlazarse a un secuenciador) si la condición asociada a ella no se cumple, o forzar a que la ejecución vaya ahí el momento que se satisfaga. Existen más tipos, como el *loop*, quien crea un bucles infinitos sobre tareas a las que se asocia.

Los servicios se enlazan a selectores situados en la raíz del árbol para verificar constantemente una condición cada cierto tiempo y almacenar su resultado en el *Blackboard*. El cambio de valor de una variable en la memoria de la IA puede forzar a un decorador existente con la condición enlazada a ella a que cambie el rumbo de la ejecución del árbol.

Se necesitan crear clases en código C++ que hereden de *BTTask* para indicar qué acciones de personajes vincular a cada nodo. Además, se requieren clases derivadas de *BTService* para enlazar las funciones de un personaje que permiten verificar condiciones concretas relacionadas a su comportamiento. Por ejemplo, la acción de disparar un proyectil se almacena en un *BTTask*, mientras que la función que verifica si hay enemigos en rango en un momento dado se recopila en un *BTService.*

Al contrario de las tareas y servicios, los *Behavior Trees* se diseñan e implementan de manera gráfica colocando y ordenando sus nodos según corresponda. El orden de tareas es importante, ya que los árboles ejecutan sus nodos de izquierda a derecha por defecto, siempre que no haya una condición que lo impida.

Las clases personalizadas creadas relativas a la IA en el proyecto son las siguientes (ver figura 3.3.8 en el anexo):

IV.IV.I Mando de IA

- **MandoDeIA**: Hereda de *AIController* y es la clase encargada de recoger y alojar su *Behavior Tree* y *Blackboard* correspondiente. MandoDeIA se enlaza a las clases Entidad para ser el intermediario entre el personaje y su inteligencia artificial.

IV.IV.II Complementos de Behavior Trees

- **BTService_XXX**: Recopila todas las acciones implementadas (y no heredadas) en la clase de personaje “XXX” que necesitan ser condiciones de bifurcación. Cada una que hace uso de un *BTService* solo necesita comprobar una condición exclusiva a ella como máximo, por lo que no hace falta múltiples *BTService* distintos asociados al mismo.

- **BTTask_XXX**: Recopila todas las acciones implementadas (y no heredadas) en la clase de personaje "XXX" que necesitan ser tareas. A diferencia del caso anterior, aquí existen casos en los que una misma clase necesita poder elegir entre varios tipos de acciones distintas en su *BTTask*. Para resolver este problema, se ha añadido un identificador adicional a aquellos nodos que pueden alojar varias acciones distintas, de manera que cada valor representa una función distinta.

  En el editor gráfico del árbol se puede usar el mismo tipo de nodo de tarea varias veces con identificadores distintos, de manera que cada uno ejecute una acción distinta. Además, se pueden añadir *BTTask* de clases distintas al *Behavior Tree* diseñado para una entidad en concreto, siempre y cuando contengan acciones de una clase padre al personaje relevante. De esta forma, la mina puede implementar la tarea de explotar definida en el *BTTask* de torres de uso único, por ejemplo.

IV.IV.III Behavior Trees

Al diseñar e implementar visualmente un *Behavior Tree*, cada tipo de componente usa un color distinto en su aspecto visual para facilitar su comprensión:

- Selector y secuenciador: Negro
- BTTask: Morado
- BTService: Verde
- Decorador: Azul

Las apariencias gráficas de los árboles finales utilizados en el juego son demasiado extensas como para ser mostradas aquí, en su lugar, se encuentran en el apartado de figuras 2.X del anexo. Existen tantos *Behavior Tree*s como distintos tipos de comportamiento de entidades existan, detallándose a continuación:

- **BT_TorreDisparador**: Usado por el cañón, cañón doble y pistola láser; este árbol tiene dos estados principales: atacar y descansar. (Ver figura 2.1.1 en el anexo).

  En el estado de ataque, la torre empieza por animar su apuntado y después repite ciclos de ataque de forma indefinida:

  > Por cada ciclo de ataque se deben disparar la cantidad de proyectiles definidos en la cadencia de disparo de la torreta (este valor es 1 para toda torre de este tipo, excluyendo el cañón doble para quien es 2).
  >
  > > Para disparar, se anima la acción de disparo y se crea el proyectil.
  >
  > El ciclo de ataque finaliza con un periodo de recarga en el que la torreta permanece inmóvil hasta realizar la siguiente secuencia de disparos.

  En el estado de descanso, la torreta realiza su animación de despunte, seguido de quedarse inmóvil.

  (Para elegir entre uno de los dos estados principales la torreta comprueba cada medio segundo si tiene un objetivo al que poder atacar o no mediante un servicio. Solo si la respuesta es contraria a la última observada, entonces se empieza a ejecutar el otro estado desde su primer paso).

- **BT_TorreProducidor**: Usado por el panel solar y panel solar doble, repite un mismo ciclo de tres acciones: generar energía, encenderse y apagarse. (Ver figura 2.1.2 en el anexo).

  En el primer paso, el panel queda inmóvil durante 15 segundos aproximadamente.

  En el segundo paso, la torreta se carga de energía y se ilumina hasta durante un máximo de 10 segundos. Aquí, el usuario puede recolectar sus recursos.

  El tercer paso ocurre cuando el jugador pincha en la torre o pasan 10 segundos. Indiferentemente, el panel se apaga y se vuelve al paso 1.

- **BT_Mina**: Usada por la mina, este árbol posee tres estados: uno en el que está esperando para activarse, otro en el que está activa, y el último en el que detona. (Ver figura 2.1.3 en el anexo).

  En el primer paso, la torreta espera 20 segundos sin hacer nada. En este periodo es vulnerable.

  En el segundo paso, la mina se activa visiblemente mediante una animación y pierde su estado de vulnerabilidad.

  El tercer paso ocurre cuando un robot la pisa: empezando por la animación de su detonación, seguido de una explosión en la que elimina a todo enemigo en su casilla. Al finalizar el proceso, la torre se autodestruye.

- **BT_TorreUsoInstantaneo**: Usada por la bomba, este árbol fuerza la ejecución inmediata de su proceso de explosión como única tarea. (Ver figura 2.1.4 en el anexo).

- **BT_RobotBásico**: Usado por todo enemigo sin habilidades especiales, este árbol posee dos estados principales: avanzar hacia la zona objetivo o detenerse para atacar. (Ver figura 2.2.1 en el anexo).

  En el estado de movimiento, se da la orden de realizar la animación de desplazamiento y mover la entidad hacia delante de manera indefinida.

  En el estado de ataque, el primer paso es detenerse (parando la animación de desplazamiento también). Tras esto, se pasa a repetir ciclos de ataque de forma infinita:

  Por cada iteración, se anima la preparación del disparo (normalmente se requiere iluminar los ojos del robot en cuestión). Tras esto, se materializa el proyectil y espera un tiempo reducido de tiempo para repetir el bucle.

  (Se permite cambiar de estado de la misma forma que con torres disparadoras, pero usando otro servicio que ejecuta la función de detección de torretas en robots).

- **BT_RobotBombaRadar**: Usado por el robot bomba radar, es igual al *Behavior Tree* anterior, pero incluyendo la comprobación de si tiene más de una torreta en su rango de explosión. Detona en caso afirmativo. (Ver figura 2.2.2 en el anexo).

- **BT_RobotOcultador**: Usado por el robot ocultador, es muy parecido al *Behavior Tree* básico para robots. Su única diferencia es la inclusión de tareas de destapar su punto débil tras detenerse, y ocultarse de nuevo antes de empezar a moverse. (Ver figura 2.2.3 en el anexo).

- **BT_Preview**: Usado por todo robot durante la presentación del nivel, fuerza a toda entidad que lo posea a que se desplace hacia el sentido donde el jugador está posicionado durante unos segundos. Tras haber recorrido la distancia necesaria, se detiene y elimina el árbol de la entidad.

  Esta IA permite a los robots salir de la mina de donde aparecen durante la fase de selección de torres y es implementable por todo tipo de enemigos en el juego. (Ver figura 2.2.4 en el anexo).

Cabe destacar que la torre escudo es la única entidad que carece de un *Behavior Tree*, debido a que se limita a quedarse en su casilla absorbiendo daños sin ningún tipo de habilidad adicional.

V. Proyectiles

Muchos de los personajes hacen uso de proyectiles para poder dañar al bando contrario. Se puede encontrar la explicación de la jerarquía de clases usadas para hacer esto posible en el apartado previo de proyectiles, así como las relaciones de estas con los personajes en la figura 3.3.9 del anexo.

VI. Jugador

El usuario necesita estar en el mapa donde ocurre la partida para poder interactuar con el entorno. Con este fin en mente, se ha creado las siguiente clase (ver figura 3.3.10 en el anexo):

- **PlayerPawn_EnPartida**: Se trata de un *Pawn* con un *CameraComponent*. Esta clase permite instanciar la cámara que captura la visión del jugador en el entorno e incluye funciones para desplazarla hacia los lados y rotarla. De esta manera, se puede cambiar el trozo del entorno que debe capturar según las circunstancias dadas:

  - Al cargar un nivel, la cámara se mueve hacia la derecha y se inclina hacia abajo para enseñar los robots que van a aparecer en la partida.
  - Tras elegir las torretas a usar en el nivel, el punto de vista del jugador se desplaza hacia la izquierda para capturar un ángulo perfecto del campo de batalla formado por casillas donde el usuario puede colocar sus artilugios.
  - En el caso de derrota, la cámara se desplaza hacia la izquierda para capturar el momento en el que un robot se infiltra en la mina del jugador.

VII. Gestión de la partida

Se necesitan clases adicionales para almacenar datos, controlar las acciones del usuario, y establecer el flujo y normas de juego. Con esta finalidad, se han creado las siguientes clases:

- **ConstructoraDeBlueprints**: Almacena referencias a algunas constantes con distintos valores por cada tipo de personaje, estos son:

    - Para torres y robots: la referencia al par *Behavior Tree* y *Blackboard* correspondiente.
    - Para robots: sus valores de pesos en oleadas.
    - Para torres: sus costes, tiempos de recarga, imágenes de tarjeta, y si deberían empezar ya recargadas al iniciar el nivel o no.

- **MandoDeJugador_EnPartida**: Hereda de *PlayerController* y se vincula a la instancia PlayerPawn_EnPartida en el mapa de juego. Sus funciones incluyen:

    - Recordar qué torretas está eligiendo el jugador durante la fase de selección de torres.
    - Recordar qué torreta (o TNT) tiene seleccionada el usuario actualmente en una partida.
    - Arbitrar los temporizadores de recarga de cada torre del usuario durante el juego.
    - Contabilizar la cantidad de energía (o recursos) de los que dispone el jugador. Realizar verificaciones de si se permite seleccionar una torreta, basadas en su tiempo de recarga y coste.
    - Procesar el pago y reinicio de recarga de una torreta tras ser colocada.
    - Incluir una lista de referencias a interfaces que su clase hija puede definir y usar.

- **BP_MandoDeJugador_EnPartida**: Hereda de MandoDeJugador_EnPartida. Define la lista de interfaces disponibles que el mando de jugador puede cargar. Estas han sido descritas en el apartado III de esta misma sección.

- **GameMode_EnPartida**: Hereda de *GameModeBase.* Esta es la clase que maneja las normas del juego y conoce su estado en todo momento. Según las circunstancias, arbitra cómo encaminar el juego. Sus funciones más importantes son las siguientes:

    - Al empezar una partida, comprobar el perfil de guardado del usuario para cargar el archivo .json (un fichero de texto) correspondiente con los datos del nivel relevante. Los datos genéricos de cada nivel se encuentran en la tabla III de los anexos.

    - Arbitrar que primero se deben seleccionar torres a llevar al nivel antes de empezar. Para ello, dar la orden a la cámara que se mueva y al mando de jugador que cargue la interfaz respectiva. Además, informar a la BP_ZonaSpawnPreviewRobot de que haga aparecer instancias de robots de ejemplo contra los que el usuario se deberá enfrentar en el nivel después.

- Tras elegir las torres deseadas, dar la orden de pasar a la fase de juego. El primer paso es pedir a la cámara de que se vuelva a mover la hacia la izquierda. Además, se ordena al mando de juego que cargue la cuenta atrás del nivel (ver figura 1.2.10 en el anexo) y muestre la interfaz en la que el usuario puede seleccionar una torreta para ponerla en casillas (ver figura 1.1.11 en el anexo).

- Una vez en la partida, leer los datos de oleadas en el nivel con la finalidad de elegir qué y cuántos enemigos deben aparecer en cada una de ellas, e informar a BP_ZonaSpawnRobotEnPartida que materialice dichos oponentes en el nivel.

- Manejar cada cuánto tiempo se deben cargar oleadas y controlar la lógica de hacerlas aparecer antes si los robots actuales sufren suficientes daños. Para esto último, los componentes de vida exclusivos de los robots informan a esta clase de cuánta salud pierden cada vez que son dañados o incluso de si pierden toda su salud.

- Informarse por parte de BP_ZonaTargetRobot si un enemigo llega hasta el final del nivel. Se debe realizar la cinemática de derrota en su caso, que incluye pedir a la cámara que se mueva a la izquierda y al mando de jugador que cargue la interfaz de derrota.

- Verificar la condición de victoria del nivel, cargando la interfaz de victoria vía el mando de jugador cuando el último robot es derrotado.

- Tras completar un nivel, guardar el progreso realizado junto con los desbloqueos de torretas correspondientes y pasar al siguiente. Todos los niveles son arbitrados por esta misma clase siguiendo los pasos anteriores, y solo se diferencian por los tipos de enemigos y sus cantidades enviadas al jugador.

- **BP_GameMode_EnPartida**: Hereda de GameMode_EnPartida. Esta clase complementa las acciones de su superior, redirigiendo las órdenes de cargar las distintas interfaces al mando del jugador y gestionando interacciones con el reproductor de música de fondo.

- **Guardador**: Hereda de *SaveGame.* Este almacena la información persistente que debe ser recordada entre reinicios de juego:

  - El nivel actual en el que se encuentra el usuario.
  - Si se ha completado el juego al menos una vez.
    (No permitir el desbloqueo de torretas en su caso, porque ya las debería poseer).
  - La lista de tipos de torretas desbloqueadas.
  - El valor del volumen de música y efectos de sonido.
    (Los volúmenes son los únicos ajustes que el motor de juego no puede guardar en sus archivos de configuración, por lo que se deben almacenar aparte en el archivo de guardado).

El diagrama de estas clases junto con el resto de componentes anteriores que forman la partida se puede encontrar en la figura 3.3.11 del anexo.

8.4 Funcionamiento de la ejecución del juego

8.4.1 Cómo leer un diagrama de secuencia

Los diagramas de secuencia muestran las comunicaciones entre objetos creados durante la ejecución de un programa. Si bien los diagramas de clase son aplicables a un proyecto entero, cada diagrama de secuencia representa una parte de su ejecución en concreto. Esto quiere decir que es posible crear decenas o incluso centenas de diagramas de secuencia, donde cada uno explica una parte concreta del funcionamiento.

Fundamentalmente, cada objeto en el esquema ejecuta funcionalidades para poder llevar a cabo el objetivo de la ejecución en cuestión. Es común que durante el proceso un objeto necesite ayuda de otro, siendo este primero interrumpido hasta que el solicitado le devuelva su respuesta. Estas peticiones pueden compartir datos entre comunicaciones, pudiendo principalmente ser de los siguientes tipos:

- **Bool**: Un valor binario que solo puede tomar valores 0 o 1, falso o verdadero respectivamente.
- **Int**: Un valor entero. Ej.: 4.
- **Float**: Un valor real usando "." como delimitador decimal. Ej.: 4.5.
- **Vector**: Conjuntos de valores reales que generalmente indican coordenadas 2D o 3D. Ej.: [0.5, 1.2].
- **String**: Cadenas de caracteres que forman frases. En los lenguajes de programación se representan como palabras entre comillas. Ej.: "Hola mundo".
- **Otras clases**: Se pueden transmitir referencias de objetos existentes usando su clase directa o una de sus padres como puntero a la posición en memoria del dispositivo donde se encuentra.
  Ej.: El tipo de puntero Entidad que está referenciando a una instancia BP_Cañon concreta, de manera que luego se le pueda pedir que realice una acción a ese personaje en concreto.

Cada funcionalidad o acción disponible de una clase se recoge en una función donde se programa lo que tiene que hacer. Las funciones disponen de diversas características, aunque las que se indican en los diagramas de secuencia suelen ser las siguientes:

- Su nombre que implica lo que hace.
- Unos parámetros (datos) de entrada que necesita para realizar la funcionalidad. Pueden ser opcionales.
- Otros parámetros de salida que sirven como respuesta al objeto que lo llamó, o "void" si no hay.

La denotación de una función sintetiza estas características de la siguiente forma: "*NombreDeLaFunción*(*ParámetrosDeEntrada*): *ParámetrosDeSalida*". Ej.: MiFunción(): void.

Cada dato de entrada puede ser enviado mediante una constante (representado por su valor) o vía una variable que puede tomar valores distintos según la ejecución previa hasta ese punto (representado esta de la siguiente forma: "*NombreDeLaVariable*: *TipoDeDato*". Ej.: MiEntero: int).

En los diagramas de secuencia los objetos que forman parte de la ejecución se identifican por su clase directa y se representan mediante cuadrados azules. Las funciones, en cambio, son flechas que originan en en su solicitante y terminan en el solicitado, ordenadas por números en orden cronológico. (Figura 19).

El solicitado es siempre el que debe conocer la funcionalidad a ejecutar, siendo esto opcional para el solicitante. Esto quiere decir que un objeto puede pedir a otro ejecutar funcionalidades originadas en su clase directa o en una de sus padres en la herencia. (Figura 20).

Las llamadas a otras funciones no tienen porqué estar implementadas en instancias ajenas, siendo posible que un mismo objeto se llame a sí mismo para procesar otra función adicional. Estos subprocesos pueden terminar sin peticiones adicionales, o pedir ayuda a más objetos (él mismo incluido mediante más llamadas reflexivas). Estas peticiones adicionales se representan mediante subíndices numéricos en el diagrama de secuencia. (Figura 21).

En un algoritmo es muy común llegar a bifurcaciones en las cuales se debe tomar una decisión bajo una condición, para decidir qué procesar después. En estos esquemas, se representan mediante rectángulos de color oscuro, acompañados de su condición respectiva. (Figura 22).

De forma similar al concepto anterior, es posible repetir un mismo trozo de función varias veces siempre que una condición concreta se cumpla. En estos diagramas, se denotan por rectángulos amarillos, junto con su condición. (Figura 23).

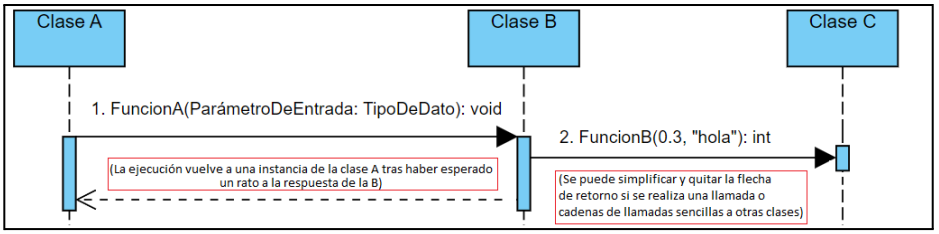


Figura 19: Representación gráfica sencilla de un diagrama de secuencia

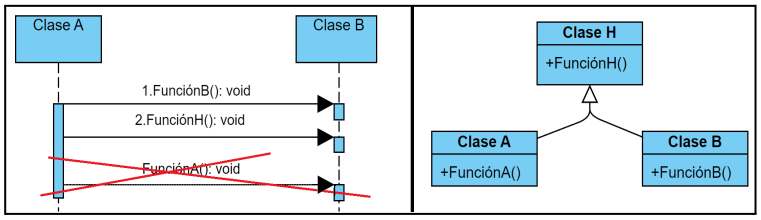


Figura 20: Ejemplo de funciones ejecutables por un objeto representados en un diagrama de secuencia

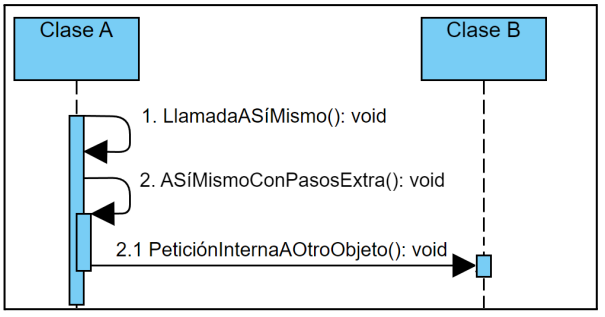


Figura 21: Llamadas reflexivas en diagramas de secuencia

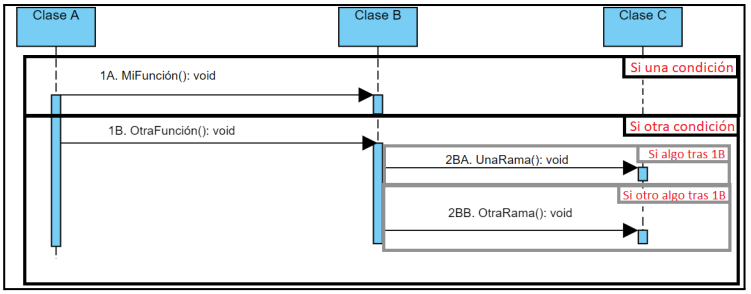


Figura 22: Bifurcaciones en diagramas de secuencia

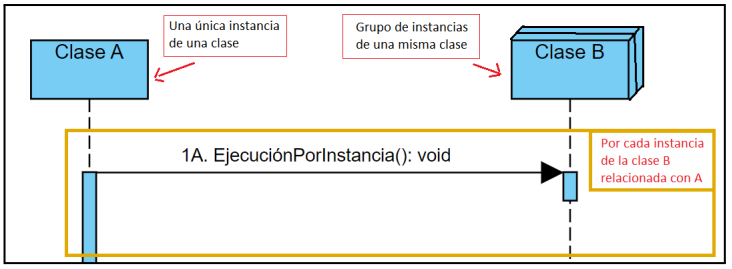


Figura 23: Bucles en diagramas de secuencia

8.4.2 Funcionalidades más importantes

En los siguientes apartados se describen en detalle la lógica de funcionamiento de los eventos más importantes de una partida en un orden fácil de entender, estos son:

- **I.** Los primeros momentos al entrar en una partida.
- **II.** La lógica de aparición de oleadas.
- **III.** Hacer una selección sobre qué se quiere hacer en una casilla después (colocar o quitar una torre).
- **IV.** Elegir la casilla en cuestión donde performar la acción.
  - **IV.I** La acción es colocar una torreta.
  - **IV.II** La acción es quitar su torreta.
  - **IV.III** La acción es recolectar la energía del panel solar que aloja.
- **V.** El protocolo de derrota junto con su condición de activación
- **VI.** El proceso en el que se informa la absorción de daños en robots al modo de juego
  - **VI.I** Procesar el protocolo de victoria por eliminar el último enemigo del nivel, ofreciendo la recompensa respectiva
    - **VI.I.I** Recoger la recompensa del nivel

Cada uno de estos procesos están detallados en diagramas de secuencia, pero son tan extensos que no es posible que acompañen a sus descripciones. En su lugar, se presentan en la sección de figuras 4.X del anexo.

I. Empezar una partida

Contexto

El juego ha acabado la cuenta atrás previa al inicio de la partida y ahora se debe cargar la partida en sí.

Funcionamiento

Se comienza en el modo de juego (BP_GameMode_EnPartida) inicializando su contador de oleadas al valor mínimo posible y, mediante *Blueprints*, se informa al reproductor del juego de cambiar la música de fondo. Una vez esto, se inicializan los datos del mando de juego del usuario usando código C++, donde se almacenan los tiempos de recarga y costes de las torretas escogidas por el jugador en la fase de selección.

Tras esto, implementado en la parte *Blueprints* del modo de juego, se da la orden al mando de juego de crear la interfaz que le permite al usuario pinchar en su tipo de torre deseado a colocar en casillas después (WB_InterfazEnPartida), pero todavía no tiene su información relevante inicializada para poder funcionar. Para hacer esto posible, se obtiene el número de nivel actual (lo mostrará después) y el volumen de efectos especiales (para hacer sonar efectos de sonido cuando se pinche en sus tarjetas) desde el archivo de guardado del juego, y los identificadores de las torretas elegidas para el nivel mediante el mando del jugador.

Con estos datos, se inicializa la interfaz en cuestión al completo vía *Blueprints*, en cuyo proceso interno pide al mando de jugador obtener los precios, imágenes, y periodos de recarga de las torres correspondientes para poder cargar sus tarjetas visuales. Una vez completado esto, se inicia un bucle infinito para actualizar visualmente el proceso de recarga de las torretas.

Finalmente, el modo de juego programa la aparición de la primera oleada del nivel a los 26 segundos, y cuando esto ocurre, se da la orden de generar un efecto de sonido especial usando *Blueprints* para señalizar la aparición de la primera horda del nivel. A su vez, se procede a cargar los datos de dicha oleada (este último paso está documentado en el siguiente paso).

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.1 del anexo.

II. Hacer aparecer una oleada

Contexto

Se ha dado la orden de hacer aparecer la siguiente oleada del nivel. Esto se puede deber a que se ha eliminado una cantidad elevada de robots y se necesitan hacer aparecer más enemigos, o ha pasado una cantidad finita de tiempo desde la aparición de la anterior (26 segundos para la primera, o unos 20-30 segundos en su defecto).

Funcionamiento

Empezando en el modo de juego, se incrementa en uno su contador de oleadas y se leen los archivos del juego (fichero nivelX.json) que contiene los datos relativos a la horda actual a hacer aparecer. Los datos interesantes de la horda incluyen los tipos de enemigos disponibles, sus distribuciones de probabilidades de aparición, y el peso o fuerza máxima a poder generar en la horda actual.

Si tienen formato incorrecto, se vuelve al menú principal para informar al jugador de lo ocurrido.

En caso contrario, verificar si la horda a hacer aparecer se trata de una gran oleada o no.

Si es el caso, usando la parte implementada en *Blueprints* del modo de juego, se ordena cambiar la música de fondo al reproductor del juego para dar el aviso y se pide a la interfaz de partida mostrar una ventana que informa al jugador de que se aproxima una gran horda. Si además se trata de la última del nivel, entonces esto también se muestra en un segundo aviso seguido del primero.

Independientemente del tipo de horda, se hacen aparecer los enemigos necesarios (implementado en C++). Para esto, se informa a la zona que se encarga de generarlos (BP_ZonaSpawnRobotEnPartida) de que busque la posición del robot vivo más alejado en cada fila actualmente con el propósito de empezar a colocarlos detrás de estos si están sobrepasando la zona de aparición. De esta manera, los robots nuevos no se solapan con los antiguos al generarse.

Habiendo hecho esto, se informa mediante *Blueprints* a la interfaz de usuario de que avance en uno el progreso de oleadas del nivel, animando su *widget* en la esquina inferior izquierda de la pantalla. Ahora ya se pueden empezar a generar los robots, definido en C++:

Mientras haya peso disponible en la horda, se intenta generar un robot.

Para generar un robot, elegirlo aleatoriamente usando las distribuciones de probabilidades disponibles de la horda actual. Elegido el tipo, informar a la zona de aparición del tipo de enemigo a generar y poner internamente el identificador de oleada de la entidad generada a la actual.

Además, internamente en el modo de juego, se resta al peso total disponible de la horda actual el consumido por el enemigo recién generado. Además, la mitad de su salud máxima es añadida a un total de vida que la oleada actual debe perder para avanzar a la siguiente antes de lo previsto. Junto con esto, se incrementa en uno el contador de robots vivos actuales en el nivel.

La zona de aparición se encarga de elegir una fila donde situar el robot (nunca teniendo una que genere más de un enemigo respecto a otra en la misma oleada) y calcula la posición en el mundo donde colocarlo de manera que no se solape con otros invocados (situándolos detrás de los últimos que se hicieron aparecer durante esta misma oleada si fuera necesario).

Una vez acabada la generación de la oleada actual, se programa la siguiente para que ocurra a los 20-30s, siguiendo las indicaciones de los datos cargados de la horda actual al principio de este proceso.

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.2 del anexo.

III. Elegir un tipo de torre para colocarlo después

Contexto

El jugador ha pinchado en la interfaz de juego durante una partida (ver figura 1.1.11 en el anexo) para marcar un tipo de torreta con la intención de colocarla después, o ha elegido la TNT para eliminar una torreta existente.

Funcionamiento

El primer paso es, desde la propia interfaz pinchada, informar al mando de jugador de la posición de la tarjeta pinchada (puede ser la TNT o una torreta). El mando almacena de forma interna el orden en el que figuran los tipos de torretas en la pantalla y qué tiene seleccionado el usuario en un momento dado.

Así, este puede procesar la solicitud y responder de vuelta con 3 valores: si se puede permitir la selección (la TNT está libre las condiciones de coste y recarga, por lo que siempre estará disponible), si se está seleccionando algo (pinchar en la selección actual la desmarca en vez de seleccionarla de nuevo), y la posición de la selección previa. De esta forma, la interfaz puede deducir y mostrar gráficamente lo ocurrido:

- Una selección correcta y seleccionando algo indica que el usuario quiere elegir algo distinto a su selección previa y se lo puede permitir. Para esto, se debe marcar en verde la tarjeta que el jugador seleccionó y desmarcar la anterior visualmente.
- Una selección incorrecta quiere decir que el jugador trató de seleccionar una torreta, pero no se le permite (debido a falta de energía y/o estar en periodo de recarga). Para esto, se debe marcar en rojo la carta durante unos instantes y desmarcar su selección anterior.
- Una selección correcta pero sin seleccionar algo quiere decir que el usuario está desmarcando su selección actual. Para ello, se debe quitar el tono verde de la tarjeta que el usuario eligió.

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.3 del anexo.

IV. Pinchar en una casilla

Contexto

El jugador ha pinchado en una casilla durante una partida. Esto puede ser debido a que quiere colocar su torre seleccionada en dicha posición, quitar la torreta de la casilla, u obtener la energía producida del panel solar alojado en esa localización.

Funcionamiento

Al procesar una acción de pinchado, el motor de juego redirige la ejecución al peón del jugador por defecto. No se quiere esto, por lo que se redirige el flujo al mando del jugador, donde se almacenan los datos relativos a los tipos de torretas elegidos por el usuario. Una vez en el mando, se verifica si se pinchó en una casilla o en un panel solar (siendo estos los únicos elementos interactuables que activan este paso).

Si fue una casilla, se verifica si ya tiene una torreta colocada en ella

Si no es el caso, se obtiene la selección actual del usuario, cuyo proceso de selección fue explicado en el diagrama anterior. También se verifica si el usuario tiene una torreta elegida, y si hay robots pisando la casilla o si el tipo de torreta seleccionada (en el caso de que haya una seleccionada) puede ser colocada en una casilla ocupada por robots.

En caso de que haya una torreta elegida y se pueda colocar en la casilla, se siguen las instrucciones de generación de torretas, explicadas en el apartado "colocar una torre".

En caso negativo, se verifica si la TNT está seleccionada.

Si es el caso, se procede a la eliminación de la torreta, cuyo proceso está explicado en el apartado "quitar una torre".

Si no es el caso, se obtiene la torreta alojada en la casilla. Si se trata de un panel, entonces tratar de recolectar su energía siguiendo las indicaciones en el apartado "recolectar energía de un panel solar".

Si fue un panel, se verifica si la TNT está seleccionada

Si es el caso, se obtiene la referencia de la casilla desde la torreta y se quita el panel de la casilla, siguiendo el proceso detallado en el apartado "quitar una torre".

Si no es el caso, se trata de recolectar su energía siguiendo los pasos del apartado "recolectar energía de un panel solar".

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.4 del anexo.

IV.I Colocar una torre

Contexto

El jugador ya ha pinchado en una casilla durante una partida y el programa ha detectado que se va a colocar la torreta seleccionada por el jugador ahí.

Funcionamiento

El proceso empieza informando el mando del jugador a la casilla en cuestión del tipo de torreta que debe generar. La implementación en *Blueprints* de la casilla se encarga de crear la torre encima de ella (los tipos de torretas disponibles son clases *Blueprints* que no pueden ser encontradas por código C++). Junto con esto, se anima la colocación mediante un efecto de sonido.

Una vez generada la torre, el mando de jugador desmarca la selección del usuario y paga el coste de la torre internamente (debido a que se necesita tener suficiente energía para elegir la torreta en primer lugar con el propósito de colocarla después, el jugador nunca acabará en deuda tras pagar su coste). Junto con esto, calcula el instante de partida en el que la torreta volverá a estar disponible tras su periodo de recarga.

Tras consumir la energía necesaria en el mando del jugador, se informa a la interfaz de partida por uso de *Blueprints* de que actualice su cantidad de energía mostrada al nuevo valor calculado. Finalmente, se deja de marcar en verde la torreta seleccionada porque ya se ha colocado, y en su lugar, empieza a animar el proceso de recarga de esta.

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.4.A del anexo.

IV.II Quitar una torre

Contexto

El jugador ya ha pinchado en una casilla durante una partida y el programa ha detectado que se quiere eliminar la torreta en esa posición.

Funcionamiento

El mando de jugador informa a la casilla respectiva que elimine vía TNT la torre que aloja. Para esto, la casilla destruye inmediatamente su torreta haciendo uso de una función predefinida en el motor de juego (*Destroy()*). Seguido de esto, se elimina la referencia de la torre en la casilla para poder permitir enlazar una nueva a esta en el futuro. Después, se pide a la implementación *Blueprints* de la casilla que genere partículas de explosión y un efecto de sonido especial para indicar el borrado de la torreta.

Una vez eliminada la torre, el mando de jugador desmarca la selección de TNT internamente e informa a la interfaz de partida mediante *Blueprints* de que haga lo mismo visualmente.

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.4.B del anexo.

IV.III Recolectar energía de un panel solar

Contexto

El jugador ya ha pinchado en una casilla durante una partida, y el programa ha detectado que se va a intentar recolectar energía del panel que está colocado en ella.

Funcionamiento

El mando de jugador pide al panel (torre) en cuestión que trate de recolectar su energía.

> Si el panel no está en el estado de energía producida y esperando un click del jugador en su árbol de IA, entonces la ejecución finaliza aquí.
>
> En caso contrario, se avanza al siguiente estado de IA del panel y se informa de vuelta al mando de jugador de cuánta energía se ha recolectado.
>
> Tras esto, se actualiza internamente la cantidad total de energía disponible y se informa a la interfaz de partida mediante el uso de *Blueprints* de que muestre este mismo valor.

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.4.C del anexo.

V. Procesar derrota

Contexto

Un robot acaba de llegar a la zona objetivo situada en la parte izquierda del mapa. El juego debe proceder al protocolo de derrota en el programa.

Funcionamiento

Todo proceso de derrota empieza con un evento que se activa cuando ocurre una colisión en la zona objetivo situada en la izquierda del nivel. Primero se comprueba si la colisión en cuestión ocurrió con un robot.

Si no es el caso, la ejecución finaliza aquí.

Si es el caso, el jugador acaba de perder y se deben realizar varios pasos en el modo de juego. Primero, se cambia la música a un tono de suspense. Después, se detienen las programaciones de futuras apariciones de oleadas. Junto con esto, el modo de juego avisa al mando del jugador que quite la interfaz de partida de la pantalla.

Tras esto, se iteran por todos los proyectiles y personajes del nivel. Por cada uno de estos, se les da la orden de detenerse completamente, parando temporizadores internos. Además, en el caso de los personajes, también detienen su IA contactando con sus mandos respectivos.

Con todo el mundo quieto, el modo de juego da la orden al peón de jugador (la cámara) de que se mueva hacia la izquierda lentamente para que se muestre en primer plano la mina del usuario. Tras finalizar este movimiento, la ejecución vuelve al modo de juego y este despausa al robot que cruzó la zona objetivo del nivel. Así, se puede desplazar rápidamente hacia dentro de la estructura durante un par de segundos antes de ser destruido fuera de la vista del jugador.

Con esta cinemática terminada, se pasa a la implementación *Blueprints* del modo de juego, quien da la orden de cambiar la música de nuevo a un tono triste. Finalmente, se pide al mando de jugador que genere la interfaz de derrota, donde el usuario tendrá la opción de reiniciar el nivel o volver al menú principal.

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.5 del anexo.

VI. Procesar daños de robot

Contexto

Un robot acaba de ser dañado en una partida. El juego debe contabilizar los daños absorbidos totales de la oleada (para acelerar la aparición de la siguiente si fuera oportuno) e incluso llegar a verificar la condición de victoria de juego en el caso de muerte de la entidad.

Funcionamiento

El proceso empieza con el origen del daño causado informando al componente de vida del robot en cuestión de que actualice su cantidad de salud respecto a los daños recibidos. Esto ha podido ser por la colisión con un proyectil o explosión de torreta.

Tras calcular la nueva cantidad de salud, se informa al modo de juego de la cantidad de daños absorbidos, quien a su vez pregunta al robot dañado en qué oleada fue generado.

> Si el robot fue creado en la oleada más reciente del nivel, entonces se acumulan sus daños al contador de vida total de la oleada más reciente.
>
> > Si los daños provocaron que estos en su totalidad excedan el 50% de la salud total y la siguiente oleada que se aproxima no es una gran oleada, entonces se detiene el temporizador interno que gestiona cuando debería aparecer la siguiente para llamarla ahora mismo en su lugar. (La generación de oleadas fue explicada en el algoritmo II).

Tras la contabilización de daños, el componente de salud comprueba que la entidad en cuestión no ha perdido todos sus puntos de vida.

> Si ha ocurrido, se informa al modo de juego de que contabilice su muerte. Aquí, se decrementa el contador de robots vivos en 1.
>
> > Si da la casualidad de que no quedan enemigos en el nivel y ya se hizo aparecer la última horda en este, entonces se ejecuta la lógica de victoria del nivel, explicada en el siguiente algoritmo.
> >
> > Si no es el caso, se detiene el temporizador que indica cuando debería aparecer la siguiente horda para generarla ahora mismo en su lugar, esto se debe a que no hay enemigos en el nivel. (Este paso está pensado para forzar a adelantar una gran horda en el caso de que no hayan enemigos vivos en el nivel).

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.6 del anexo.

VI.I Victoria

Contexto

Se acaba de eliminar el último robot del nivel, el programa debe proceder a dar la victoria al usuario y recompensarlo dependiendo del nivel actual.

Funcionamiento

La función de victoria empieza en el propio modo de juego. Primero de todo, se pide al último robot eliminado del nivel que ofrezca sus coordenadas en el mundo, de manera que el mando de jugador pueda calcular esa posición respecto al punto de vista del usuario en la pantalla.

Tras este paso, se leen los archivos del juego (fichero desbloqueos.json) para ver qué desbloqueo está vinculado al nivel actual.

> Si los datos no son interpretables, se vuelve al menú principal informando al jugador de esto.
>
> Si se pueden leer, se obtiene el desbloqueo correspondiente (una torreta nueva, un trofeo que no tiene ninguna funcionalidad especial, o una nota que indica que el último nivel ha sido superado).
>
> Conocido el desbloqueo, se informa al mando de jugador usando la parte *Blueprints* del modo de juego para que detenga la lógica de la interfaz de partida (bloqueando cualquier tipo de interacción con el jugador).
>
> Completado esto, se genera la interfaz de victoria usando el mando de jugador, la cual solo hace visible la tarjeta que representa la recompensa a obtener al principio. También hace uso del mando del usuario para obtener la referencia de la imagen de la torreta desbloqueada y su coste para mostrarlos en la imagen (este último paso es relevante en caso de desbloqueo de torreta).
>
> Con la interfaz inicializada, se sitúa la recompensa en la posición donde el último robot murió en la pantalla y se realiza una breve animación. Este movimiento consiste en desplazar la recompensa un poco hacia arriba y luego hacia abajo, simulando un rebote. Finalmente, se permite pinchar en esta para recogerla y pasar al siguiente nivel (esto último estando detallado en el siguiente paso).

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.6.1 del anexo.

VI.I.I Recoger recompensa

Contexto

El programa ya ha generado la recompensa del nivel y es visible en la pantalla. El jugador ahora pincha en ella para obtenerla.

Funcionamiento

La función empieza con el usuario pinchando en la recompensa situada donde el último robot fue eliminado. Al hacerlo, la interfaz de victoria (implementada en *Blueprints*) da la orden al modo de juego de que quite completamente la interfaz de partida de la pantalla (la cual en el momento actual estaba detenida). Junto con esto, se hace sonar otra pista de música usando el modo de juego como intermediario (siendo más alegre que las demás), y se desplaza lentamente la recompensa al centro de la pantalla. Tras acabar su movimiento se visualiza completamente la interfaz de victoria, mostrando el nombre de la recompensa y su descripción.

El programa se queda en pausa hasta que el jugador pinche en el botón para continuar al siguiente nivel.

Cuando el usuario lo haga se pasa la ejecución al modo de juego en código C++, quien carga el archivo de guardado del jugador para almacenar la torreta nueva desbloqueada (si había) y suma uno al contador del nivel actual. Tras actualizar los datos, se guardan internamente para actualizar los cambios de forma permanente en el archivo de guardado.

> Finalmente, si al usuario le queda algún nivel por superar, se carga el mismo mapa de nuevo usando los datos del siguiente nivel. De esta forma, se puede volver a utilizar la misma lógica de juego, pero variando en las torretas disponibles, así como las hordas de enemigos a los que el usuario se deberá enfrentar.
>
> En caso contrario, se redirige al menú principal para felicitar al jugador por haber superado el juego.

Representación gráfica

Su diagrama de secuencia respectivo se encuentra en la figura 4.1.6.1.1 del anexo.

9. Evaluación económica

9.1 Evaluación de gastos

Se consideran las siguientes vías de gasto en el proyecto realizado:

- Costes por desarrollo del producto (diseño, implementación, depuración), y documentación
- Costes por formación
- Amortización del hardware
- Costes de licencias
- Otros gastos no directos

9.1.1 Gastos por desarrollo del producto y documentación

Tomando como referencia la planificación temporal, se tasa una media de 2 horas invertidas por cada día en tareas de diseño, 2,5 en documentación, y 3 en implementación o depuración. En adición a esto, se han estimado costes de 25€/h en tareas de documentación y 35€/h para el resto.

Con la finalidad de simplificar los cálculos, se asume que las tareas de pruebas que comprueban el funcionamiento de aspectos incompletos del programa se han realizado en los periodos de implementación (es decir, solo se contabilizan las tareas de mantenimiento del programa que figuran en el plan de proyecto).

Finalmente, en el caso de que se hayan realizado varias tareas en paralelo, se acumulan sus costes solo si son de distinto tipo. Por el contrario, para actividades simultáneas de una misma categoría principal se asumen costes iguales repartidos de su total respectivo (en otras palabras, solo se contabiliza hasta una tarea en su totalidad por cada categoría en un día).

Los resultados se pueden observar en la siguiente tabla:

Categoría de tarea	**Duración total (h)**	**Coste / tiempo (€/h)**	**Costes totales (€)**
Diseño	36h (18 periodos de 2h)	35€/h	1260€
Implementación	135h (45 periodos de 3h)	35€/h	4725€
Depuración	60h (20 periodos de 3h)	35€/h	2100€
Documentación	102,5h(41 periodos de 2,5h)	25€/h	2562,20€
Suma total			**10647.2€**

Tabla 9: Costes por tareas

9.1.2 Costes por formación

El *pack* de recursos de pago de *gamedev.tv* para el aprendizaje de uso de *Unreal Engine 5* ha sido el único coste de capital invertido en este apartado, siendo su precio 24€.

A día de hoy la oferta no está disponible y no puede ser citada bibliográficamente, pero se puede comprar cada elemento del conjunto por separado (superando los 300€ en costes). Esto incluye varios cursos para uso de *Unreal Engine* [25] [26], recursos útiles para la creación de juegos en esta misma plataforma que han sido referenciados en las tablas V del anexo, así como utilidades adicionales que no han sido relevantes en este proyecto en concreto.

9.1.3 Amortización del *hardware*

Un motor de juego como *Unreal Engine* es una herramienta muy poderosa que requiere un ordenador personal (PC) con buenos componentes para simular su lógica interna. Se necesita un procesador (CPU) potente y suficiente memoria (RAM) para gestionar las acciones realizadas en el editor y juego. Adicionalmente, se requiere una tarjeta gráfica (GPU) y monitor adecuados que procesen y representen entornos tridimensionales de manera visual eficazmente [28].

El equipo empleado para el proyecto se trata de un PC con un procesador AMD Ryzen 5 PRO 3400G y 16 GB de RAM valorado en 630€ [29]. Por otro lado, el dispositivo incluye una GPU adquirida por separado de modelo NVIDIA GeForce RTX 3060 valorada en 300€ [30] y un monitor Omen 27 de 165 Hz valorado en 220€ [31].

Es común calcular la amortización de ordenadores y sus componentes a 10 años, asumiendo 5 meses de trabajo los cálculos son los siguientes:

Dispositivo / Componente	Valor inicial	Tiempo de uso	Costes por amortización
PC de procesador AMD Ryzen 5 PRO 3400G, 16 GB RAM	630€	5 meses	26,25€
Tarjeta gráfica NVIDIA GeForce RTX 3060	300€	5 meses	12,5€
Monitor Omen 27, 165hz	220€	5 meses	9.16€
Suma total			**47.91 €**

Tabla 10: Costes por amortización

9.1.4 Costes por licencias

Este apartado es el más sencillo de calcular, ya que el proyecto se ha desarrollado en su totalidad usando licencias gratuitas por cada una de las herramientas (*Unreal Engine 5*, *Virtual Studio Code*, *Blender*, *Audacity*, *GitHub*, y *Drive*).

9.1.5 Costes indirectos

Los gastos indirectos se tratan de costes que no están directamente asociados con el proyecto, pero surgen durante su elaboración. Existen numerosos, aunque el más destacable para un trabajo de esta categoría se trata del gasto de electricidad. Se asume que esta pérdida supone un incremento del 6% respecto al coste acumulado previo a este apartado.

9.1.6 Gastos totales

Tras tener en cuenta todos los costes anteriores, la suma total supone la siguiente:

Tipo de coste	**Valor (€)**
Costes por tareas de desarrollo	10647.2€
Coste por formación	24€
Coste por licencias	0€
Amortización del *hardware*	47,91 €
Gastos indirectos	643,14€ (6% de 10719,11€)
Total	**11362.25€**

Tabla 11: Costes totales

9.2 Potenciales ingresos

El desarrollo de un proyecto de esta escala es un proceso costoso y se necesita encontrar una forma de recuperar el capital perdido. La vía tradicional es publicar el videojuego en una plataforma virtual que permita monetizarlo y venderlo a usuarios dispuestos a adquirir el producto. La más conocida se trata de *Steam*, en donde figura a la venta *Plants vs. Zombies* (la inspiración de la idea del juego) con un precio de 4,99€ [32].

Asumiendo que se quisiera llevar Torres vs. Robots al mercado vía *Steam* con el mismo precio, en principio se requerirían vender 2277 copias virtuales para acabar con la deuda (ver figura 24).

$$\frac{11362.25€}{4{,}99€/ejemplar} \simeq 2277\ ejemplares$$

Figura 24: Cálculo inicial de rentabilidad

Desgraciadamente esto no es tan sencillo porque todas las plataformas distribuidoras siempre recogen una comisión por cada venta, siendo el 30% del precio en el caso de *Steam* [33]. Corrigiendo los cálculos, se necesitarían 3253 descargas para compensar los costes (ver figura 25).

$$\frac{11362.25€}{70\% \times 4{,}99€/ejemplar} \simeq 3253\ ejemplares$$

Figura 25: Cálculo de rentabilidad en *Steam*

Una comisión del 30% impacta gravemente en la cantidad de copias necesarias que se requieren vender. Una de las alternativas más populares a *Steam* es *Epic Games*, que además de ofrecer el propio motor de juego en el que se ha desarrollado el producto, también permite publicar y vender videojuegos en su tienda. El reparto de ingresos ofrecido es de 88% / 12% a favor del vendedor, comisionando así menos de la mitad que su contraparte en el mercado [34]. Teniendo en cuenta estos datos, se requerirían vender 2588 ejemplares en la tienda virtual de *Epic Games* para compensar los costes (ver figura 26).

$$\frac{11352{,}55€}{88\% \times 4{,}99€/ejemplar} \simeq 2588\ ejemplares$$

Figura 26: Cálculo de rentabilidad en *Epic Games*

Epic Games no tiene tanto público como la opción anterior, en las cifras de fin de 2023 alcanza dos tercios de la cantidad de usuarios en *Steam* (80 contra 120 millones de usuarios mensuales), siendo más eficiente vender el programa en *Steam* aun pidiendo una mayor comisión (ver figura 27). No obstante, la rentabilidad podría cambiar en los siguientes años según alteraciones en la demanda, especialmente considerando que la popularidad de la plataforma menor ha estado aumentando en estos últimos años [35].

$$120M\ usuarios \times 70\%\ valor = \ ventas\ en\ precio\ completo\ a\ 84\ M\ de\ usuarios\ en\ Steam$$
$$80M\ usuarios \times 88\%\ valor = \ ventas\ en\ precio\ completo\ a\ 70{,}4\ M\ de\ usuarios\ en\ Epic\ Game$$

Figura 27: Relación de rentabilidad entre plataformas de venta

10. Conclusiones

10.1 Resumen de la experiencia

Este trabajo ha sido una oportunidad de oro para explorar nuevas oportunidades sobre lo que es posible hacer en el ámbito de la programación, si bien hayan habido dificultades especialmente al principio del proceso. No obstante, ha sido interesante probar algo distinto a lo que se estudia en el grado, además de ser una experiencia muy agradable.

Si bien es cierto que las asignaturas de la titulación han ayudado en la formación del arte de programar, este ámbito es muy diverso e imposible de cubrir en su totalidad en solo 4 años de carrera. El TFG ha contribuido a explorar otra cara más de la programación que no se había planteado en clase y a adquirir nuevos conocimientos, ya que para crear un videojuego se requieren dominar varias habilidades. Ejemplos de esto son la programación en un motor de juego, diseño de modelos 3D de personajes, edición de música y más. Al empezar el proyecto no se sabía hacer nada de esto, y solo con el tiempo y esfuerzo diario se ha aprendido a cómo tratar con ello.

Si se tuviera que desarrollar otro juego, la base establecida por esta experiencia sería de gran ayuda, pero dependiendo de qué precondiciones se querrían establecer al producto podría ser posible tener que volver a la fase de formación para adquirir las competencias necesarias.

Esto se debe a que todo juego en *Unreal Engine* comparte ciertas características, principalmente la forma en la que se manejan sus editores, menús y creación de clases para componentes. Aun así, es posible que de juego a juego haya elementos concretos que sean exclusivos a cada uno de ellos. Por ejemplo, *Unreal Engine* permite crear estructuras esqueléticas para personajes humanoides, así como darles animaciones a cada una de sus extremidades. También permite crear juegos multijugador con estructura cliente-servidor. Estos son algunos de los conceptos que no han sido explorados en el proyecto por tener una mayor complejidad y no encajar con la temática del videojuego.

Es importante establecer una meta adecuada en base al contexto del desarrollador, tener objetivos muy ambiciosos la primera vez que se trata con un motor de juego no es muy buena idea porque puede resultar ser una experiencia muy abrumadora, y como consecuencia no finalizar el proyecto dentro del periodo establecido.

Ahora que ya se está más acostumbrado a la forma de trabajo en estos entornos, sería posible seguir más cursos para aprender incorporar algunas de las características anteriormente mencionadas en juegos futuros.

10.2 Dificultad inicial y solución a esto

En la carrera se ha aprendido a manejar editores de código principalmente (ver figuras 5.1.X en el anexo) para implementar programas informáticos, nunca tratando con aquellos que alojan entornos virtuales (ver figuras 5.2.X en el anexo).

Debido a esto, el proceso de aprendizaje inicial ha sido el paso más dificultoso del proyecto, necesitando acostumbrarse a manejar los nuevos editores. Al principio puede parecer muy desesperante, ya que estos nuevos programas contienen centenas de opciones que no son muy intuitivas cuando no se sabe cómo trabajar con ellas. Se requerirían 20 días de esfuerzo diario para poder hacer la formación posible.

Una vez familiarizado con las herramientas en cuestión, la experiencia de desarrollo no ha sido tan difícil como la fase anterior. La programación interna del juego ha sido facilitada en gran medida por los cursos destinados al manejo del motor de juego, los cuales habían sido principalmente usados para aprender a manejar sus aspectos visuales y navegar entre menús.

Además, las habilidades de diseño de programas y metodología de programación orientada a objetos adquiridas en la carrera han sido aplicables en este proyecto también, esto es muy aparente en el complejo pero efectivo diseño de herencia de funcionalidades de personajes en el juego (ver figura 3.3.5).

Esto no quiere decir que crear el programa haya sido tarea trivial, necesitando 3 meses de trabajo diario para su diseño, implementación y depuración.

10.3 Recomendaciones

La clave fundamental de este tipo de proyectos es conseguir motivación inicial y mantenerla, no recomendando la experiencia a la gente que se dé por vencida muy fácilmente. Esto es especialmente cierto si se carece de práctica en este ámbito, ya que la experiencia de aprendizaje puede resultar ser muy agotadora al principio. Los motores de juego incluyen centenas de opciones que ni la gente que se dedica profesionalmente al desarrollo de videojuegos usa en su totalidad, repeliendo a muchos posibles aspirantes cuando abren uno de estos editores por primera vez.

Es imperativo conocer las características esenciales que se usan en todo proyecto (en el caso de *Unreal Engine* sería principalmente manejar los editores de código C++ y/o *Blueprints*, así como la sintaxis de los lenguajes). Los cursos en *gamedev.tv* son una forma ideal de aprender a adquirir las destrezas necesarias, asumiendo que se tenga un nivel moderado de inglés y se dispongan de unas 30 horas de tiempo libre como mínimo.

Cabe destacar que aunque estos recursos puedan resultar muy útiles, no conviene comprarlos en su precio base. Muchos de estos alcanzan precios de tres cifras en euros cuando no están de rebajas. Se recomendaría esperar a ofertas con varios elementos en conjunto para adquirirlos en grupo a un precio mucho menor. Este fue uno de los factores principales que Marta Aguilera tuvo en cuenta en sus

recomendaciones de cursos *online* para este proyecto, y estando ahora en perspectiva también daría el mismo consejo a gente que quisiera empezar desde cero.

Tampoco se puede subestimar la importancia de empezar por algo **sencillo** si es la primera vez que se está usando un motor de juego, no siendo recomendable entrar de golpe en los aspectos más técnicos de la herramienta. Los cursos mencionados anteriormente son ideales para esto, ya que los videojuegos de ejemplo que enseñan a desarrollar no son de gran complejidad, explicando cada componente y razón por la que se incorpora al programa en paso a paso.

Conociendo cómo crear y hacer funcionar cada elemento es la forma en la que se descubre que sus combinaciones e implementaciones son infinitas, sustentando la creatividad. El hecho de que haya posibilidades sin fin a desarrollar es lo que hace la experiencia de trabajo tan agradable, porque siempre va a existir algo que te gustaría crear.

Además, *Unreal Engine* en específico tiene la ventaja de ser más atractivo para principiantes gracias a la incorporación de *Blueprints*, haciendo el aprendizaje de desarrollo de algoritmos mucho más fácil que en la programación tradicional. Otro punto fuerte es que este modo de implementación por bloques admite realizar cambios en su lógica interna al instante y probarla en tiempo real sin necesidad de incluir procesos largos intermedios como compilaciones. Esto ayuda principalmente a los novatos porque son los más propensos a fallos en sus planteamientos iniciales y necesitan más iteraciones en sus algoritmos para hacerlos funcionar.

La gente ya acostumbrada a programar puede hacer uso de C++ en lugar de *Blueprints*. La parte de desarrollo de juegos en C++ no es muy distinta al de cualquier otra aplicación, siendo conocer las funciones proporcionadas por las librerías de *Unreal Engine* la dificultad inicial. Siempre se encuentra este problema cuando se cambia de entorno de trabajo en el mundo de la programación, por lo que los expertos ya estarían acostumbrados a afrontar este paso. Otro inconveniente es esperar a que los procesos de compilación finalicen cuando se hacen cambios en los ficheros de código para depurar estas alteraciones realizadas en el programa, pero no debería traer mayores problemas ya que los veteranos no tienden a cometer tantos fallos en sus algoritmos (no teniendo que crear varias versiones distintas de la misma idea por consecuencia).

A cambio de estos sacrificios, la programación en C++ es más condensada y fácil de organizar, además de ser más eficiente en la práctica porque se trata de un lenguaje compilado. Estas ventajas son especialmente valiosas en entornos profesionales, donde pueden existir varios desarrolladores y sus clientes esperan productos de calidad.

Junto con *Unreal Engine,* también se recomendaría aprender a manejar un editor de modelado 3D, ya que los recursos disponibles en Internet son limitados y no siempre se pueden encontrar las figuras perfectas que un diseñador tiene en mente. Haciendo uso de esta herramienta adicional se puede combatir el problema y garantizar llevar las ideas a la realidad. *Blender* en concreto es un candidato ideal, ya que en 45 minutos de formación se puede aprender a crear aspectos tridimensionales básicos.

En resumen, el desarrollo de un juego desde cero puede resultar ser un proceso muy laborioso, pero también entretenido y satisfactorio una vez se entiende cómo trabajar.

10.4 Posibles mejoras

Si se hubiera dispuesto de más tiempo y personal, se podrían haber realizado más mejoras y adiciones. Principalmente se consideran las siguientes:

- Traducciones a más idiomas.
- Crear más personajes. Se había planteado la idea de crear un mortero que causara daños en área a robots pero se canceló por falta de tiempo.
- Crear más niveles usando los personajes nuevos que se habrían desarrollado.
- Diseñar nuevos mapas en los que se mantendrían las normas de juego pero cambiase el entorno ficticio (cambiar la localización a un paisaje montañoso tras superar X niveles, por ejemplo).
- Diseñar un sistema de logros en el que el usuario pudiese ver un listado de desafíos opcionales.
- Expandir las plataformas en las que el juego pudiera estar disponible, acomodando los controles a estas. Ejemplo de esto sería desarrollar una versión del juego para móvil, cosa que es posible hacer en *Unreal Engine.*

10.5 Agradecimientos

En primer lugar quiero agradecer a la directora del trabajo Aitziber Atuxta por aceptar la concepción del trabajo. También ha facilitado enormemente el empuje inicial del proyecto por poder permitir el contacto con Marta Aguilera, una ex-estudiante de esta misma facultad y experta en el ámbito relevante del trabajo.

También quiero expresar mis mayores gracias a la codirectora del trabajo Marta Aguilera por ser una muy buena guía respecto a recomendaciones de por dónde empezar en este mundo inicialmente desconocido, y por dar ayudas y sugerencias durante todo el trayecto vía *e-mail*.

Además, quiero expresar mi gratitud a Sam Pattuzzi [25], Stephen Ulibarri [25], Michael Kocha [26] y Aura Prods [27] por la ayuda ofrecida mediante cursos *online* que han permitido fortalecer mis habilidades como desarrollador de videojuegos.

Finalmente, me gustaría dar las gracias a todos los autores de recursos externos empleados en el juego por facilitar especialmente el aspecto acústico del programa (ver las tablas V del anexo para más información).

Bibliografía

[1] ehu.eus
Grado en Ingeniería Informática de Gestión y Sistemas de Información
https://www.ehu.eus/es/web/graduak/grado-ingenieria-informatica-de-gestion-y-sistemas-de-informacion-bizkaia (Consultado el 4 de junio de 2024)

[2] Bukola Joel (5 de mayo de 2023)
By recognising the importance of representation and inclusivity, we can build a more vibrant and respectful gaming culture for everyone.
(Traducción: Reconociendo la importancia de la representación e inclusividad, podemos construir una cultura de juego más respetuosa y llena de vida para todos)
https://centric.org.uk/latest-news/why-is-there-a-lack-of-diversity-in-the-video-game-industry/
(Consultado el 4 de junio de 2024)

[3] NBC News (9 de mayo de 2014)
Nintendo Apologizes For Omitting Gay Marriage From 'Tomodachi Life'
(Traducción: Nintendo pide perdón por omitir matrimonio homosexual en 'Tomodachi Life')
https://www.nbcnews.com/tech/video-games/nintendo-apologizes-omitting-gay-marriage-tomodachi-life-n101861 (Consultado el 4 de junio de 2024)

[4] Connor Christie (mayo de 2024)
Undertale Frisk lore, gender, age, and real name
(Traducción: Historia de Frisk Undertale, género, edad y nombre real)
https://www.pockettactics.com/undertale/frisk (Consultado el 4 de junio de 2024)

[5] Lauriane Guillouxel (26 de mayo de 2021)
Qué es un juego Tower Defense: definición y características
https://www.malavida.com/es/articulos/que-es-un-juego-tower-defense-definicion-y-caracteristicas
(Consultado el 4 de junio de 2024)

[6] productplan.com
Minimum Viable Product (MVP)
(Traducción: Producto mínimo viable (PMV))
https://www.productplan.com/glossary/minimum-viable-product/ (Consultado el 5 de junio de 2024)

[7] Danielle Riendeau (15 de agosto de 2023)
How To: Write a Game Design Document
(Traducción: Como hacer: Escribir un documento de diseño de videojuegos
https://www.gamedeveloper.com/design/how-to-write-a-game-design-document)
(Consultado el 4 de junio de 2024)

[8] BigMiniMentor (20 de febrero de 2022)
Desarrollo Indie vs Empresarial
https://comohacervideojuegos.weebly.com/blog/desarrollo-indie-vs-empresarial
(Consultado el 4 de junio de 2024)

[9] Universidad Europea (19 de mayo de 2023)
¿Qué es un motor de juego?
https://creativecampus.universidadeuropea.com/blog/motor-juegos/ (Consultado el 4 de junio de 2024)

[10] Unir Revista (6 de julio de 2021)
¿Qué es un IDE en programación?
https://www.unir.net/ingenieria/revista/ide-programacion/

[11] TokioSchool
¿Qué es el modelado 3D y cuáles son sus aplicaciones?
https://www.tokioschool.com/formaciones/creacion-modelado-personajes-3d-videojuegos/que-es/
(Consultado el 4 de junio de 2024)

[12] Sam from ThreeDee (1 de enero de XXXX)
5 reasons why to use Blender
(Traducción: 5 razones para usar Blender)
https://threedee.design/blog/5-reasons-why-to-use-blender (Consultado el 4 de junio de 2024)

[13] Domestika
Top 7 programas de diseño 3D
https://www.domestika.org/es/blog/11489-top-7-programas-de-diseno-3d
(Consultado el 4 de junio de 2024)

[14] JJ Lyon (27 de febrero de 2024)
Audacity has Been Improving! Here's What's New in the Free Audio Editor
(Traducción: ¡Audacity ha estado mejorando! He aquí que cosas nuevas hay en el editor gratuito de audio.)
https://krotos.studio/blog/audacity-improvements (Consultado el 4 de junio de 2024)

[15] github.com
Página de información
https://github.com/about (Consultado el 4 de junio de 2024)

[16] Javier Espinoza
Crear repositorio de GitHub (source control) | Unreal Engine 4
(El tutorial indica como hacerlo en UE 4 pero el proceso es muy parecido en versiones más modernas)
https://www.youtube.com/watch?v=sffjP6g96AE (Consultado el 4 de junio de 2024)

[17] Yúbal Fernández
VSync: qué es y cuáles son sus ventajas y desventajas
https://www.xataka.com/basics/vsync-que-cuales-sus-ventajas-desventajas
(Consultado del 11 de junio de 2024)

[18] Daniel Lara (7 de julio de 2015)
Modularidad en la programación orientada a objetos
https://styde.net/modularidad-en-la-programacion-orientada-a-objetos/
(Consultado el 10 de junio de 2024)

[19] Germán Escobar (6 de noviembre de 2017)
Lenguajes compilados e interpretados
https://blog.makeitreal.camp/lenguajes-compilados-e-interpretados/
(Consultado el 21 de julio de 2024)

[20] Alex Forsythe (Video de *Youtube*)
Blueprints vs. C++: How They Fit Together and Why You Should Use Both
(Traducción: *Blueprints* contra C++: Cómo encajan juntos y por qué deberías usar ambos)
https://www.youtube.com/watch?v=VMZftEVDuCE (Consultado el 10 de junio de 2024)

[21] Epic Games
Unreal Engine 5.4 Documentation
https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-4-documentation
(Consultado el 10 de julio de 2024)

[22] Plants vs. Zombies (Canal de *Youtube*)
Plants vs. Game Trailer
(Traducción: Plantas contra zombis trailer de juego)
Plants vs. Zombies Game Trailer (Consultado el 4 de junio de 2024)

[23] Ninja Kiwi (Canal de *Youtube*)
Bloons TD5 Trailer
(Traducción: Defensa de Torres de "Bloons" 5 tráiler de juego)
https://www.youtube.com/watch?v=_dprpzb677A (Consultado el 2 de julio de 2024)

[24] Ironhide Game Studio (Canal de *Youtube*)
Kingdom Rush Trailer
(Traducción: Kingdom Rush tráiler de juego)
https://www.youtube.com/watch?v=IfYQHO1xqrA (Consultado el 2 de julio de 2024)

[25] Sam Pattuzzi & Stephen Ulibarri (Curso en gamedev.tv)
UE5 C++ Developer: Code Your Own Unreal Games
(Traducción: Desarrollador UE5 C++: Programa tus propios juegos Unreal)
https://www.gamedev.tv/courses/unreal-5-0-c-developer-learn-c-and-make-video-games
(Consultado el 5 de junio de 2024)

[26] Michael Kocha (Curso en gamedev.tv)
Unreal Environment Design
(Traducción: Diseño de entorno Unreal)
https://www.gamedev.tv/courses/unreal-environment-design (Consultado el 5 de junio de 2024)

[27] Aura Prods (Video de *Youtube*)
🔥LA GUÍA DEFINITIVA DE BLENDER 4.0! (Tutorial completo en Español) | Desde cero! 2023
https://www.youtube.com/watch?v=O-tV7uBf5LI (Consultado el 5 de junio de 2024)

[28] The Factory School
Unreal Engine 5: Requisitos que necesita tu ordenador
https://thefactoryschool.com/blog/unreal-engine-5-requisitos-que-necesita-tu-ordenador/
(Consultado 24 de julio de 2024)

[29] PC Componentes
PC Racing Gaming AMD Ryzen 5 3400G/16GB/480GB SSD (PC equivalente al usado en el proyecto)
https://www.pccomponentes.com/pc-racing-gaming-amd-ryzen-5-3400g-16gb-480gb-ssd (Consultado 24 de julio de 2024)

[30] PC Componentes
Inno3D TWIN X2 GeForce RTX 3060 12GB GDDR6 (GPU equivalente al usado en el proyecto)
https://www.pccomponentes.com/inno3d-twin-x2-geforce-rtx-3060-12gb-gddr6

[31] PC Componentes
HP OMEN 27" LED IPS FullHD 165Hz FreeSync (Monitor equivalente al usado en el proyecto)
https://www.pccomponentes.com/hp-omen-27-led-ips-fullhd-165hz-freesync?gad_source=1&gclid=CjwKCAjwzIK1BhAuEiwAHQmU3mncHWAFS5THcyr-3V-PQZj_8HSyKj9vux0isprn7X1dr7BQyxGUqRoCiMoQAvD_BwE

[32] Steam
Plants vs. Zombies GOTY Edition
https://store.steampowered.com/app/3590/Plants_vs_Zombies_GOTY_Edition/

[33] José D. Villalobos
PREGUNTAS Y RESPUESTAS SOBRE VIDEOJUEGOS. (16 de septiembre de 2023)
https://www.laps4.com/preguntas-y-respuestas/cuanto-se-queda-steam-por-cada-venta

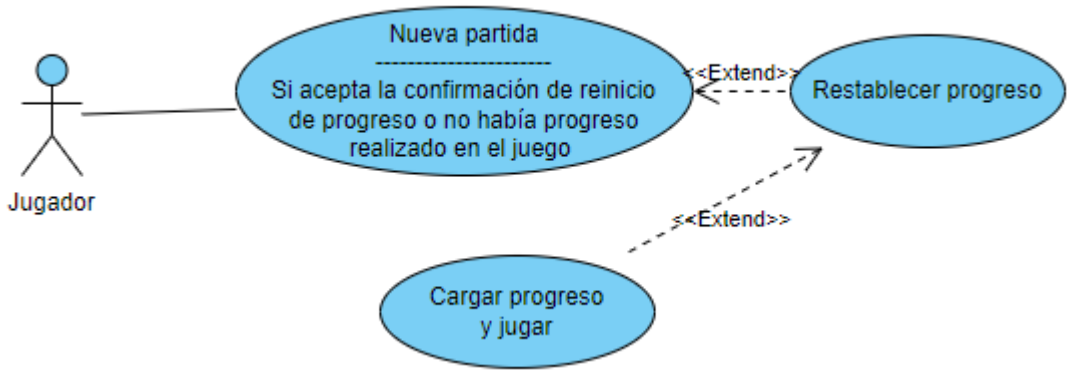
[34] Epic Games
Epic Games Store (recurso principal, sección “distribuir”)
https://store.epicgames.com/es-ES/distribution#:~:text=La%20Epic%20Games%20Store%20llega,llevar%20tu%20producto%20al%20mercado.

[35] Victor Méndez (18 de diciembre de 2023)
Epic quiere superar a STEAM y alcanza los 80 millones de usuarios mensuales, aunque la mayoría llega por los juegos gratis y no realiza compras en la tienda
https://es-us.finanzas.yahoo.com/noticias/epic-superar-steam-alcanza-80-130100252.html#:~:text=%E2%80%9CLa%20Epic%20Games%20Store%20tiene,millones%E2%80%9D%2C%20dijo%20el%20ejecutivo.

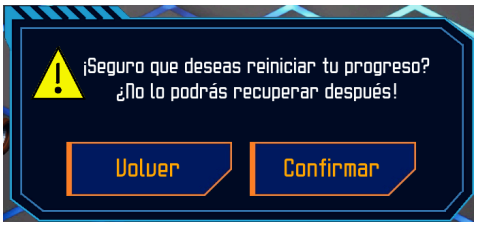
Anexos

Anexo I: Casos de uso extendidos

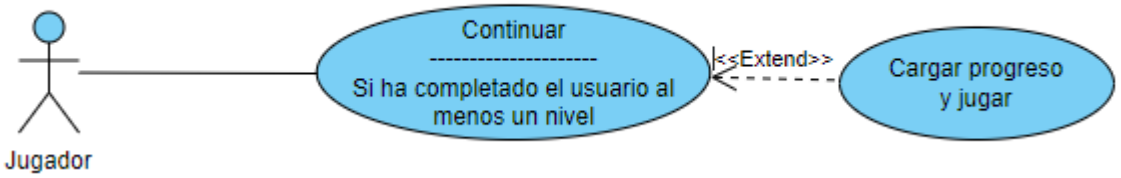
I. Nueva partida



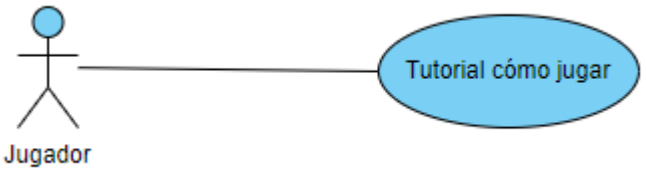
Nombre: Nueva partida
Actores: Jugador
Precondiciones: Estar en el menú principal
Requisitos no funcionales: Ninguno
Flujo de eventos: 1. El usuario pincha en empezar una nueva partida desde el menú principal (figura 1.2.1). [Si el usuario ya tenía progreso realizado en el juego] 2A. Se muestra una pantalla de confirmación (figura 1.2.6). [Si acepta] 3AA. Se reinicia su progreso y se empieza desde el nivel 1. [Si rechaza] 3AB. Se vuelve al menú principal sin hacer nada. [Si el usuario no tenía progreso en el juego, mostrado mediante la desactivación del botón de "continuar" situado en este mismo menú] 2B. Se empieza el juego desde el primer nivel.

Postcondiciones: Se carga el primer nivel del juego.
Figuras relevantes:    Figura 1.2.1: Menú principal    Figura 1.2.6: Aviso de reinicio de progreso

II. Continuar


Nombre: Continuar
Actores: Jugador
Precondiciones: El jugador debe haber superado al menos un nivel y estar en el menú principal
Requisitos no funcionales: Ninguno
Flujo de eventos: 1. El usuario pincha en el botón de continuar su partida desde el menú principal (figura 1.2.1), cargando el nivel adecuado según la información en su archivo de guardado.
Postcondiciones: Se abre el nivel en el que se encuentra el usuario actualmente según su progreso
Figuras relevantes: Mostradas en apartados anteriores

III. Tutorial cómo jugar



Nombre: Tutorial cómo jugar

Actores: Jugador

Precondiciones: Estar en el menú principal

Requisitos no funcionales: Ninguno

Flujo de eventos:
1. El usuario pincha en el botón del tutorial desde el menú principal (ver figura 1.2.1), abriendo una nueva interfaz (ver figura 1.2.2).

[Si el jugador pincha en una de las flechas disponibles en la parte inferior, proceso repetible]

2A. Se avanza o retrocede una página en las normas de juego según el recuadro pinchado.

[Si este cambio hace que se llegue a la primera o última página del manual]

2AA. Se bloquea la flecha correspondiente para no poder avanzar más en ese sentido.

[Si este cambio hace que se deje de estar en la primera o última página del manual]

2AB. Se desbloquea la flecha correspondiente para volver a permitir avanzar en ese sentido.

[Cuando el jugador pincha en el botón de volver]

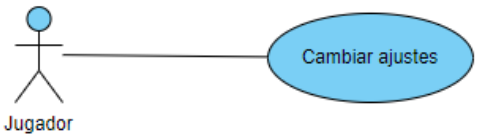
2B. Se abre el menú principal de nuevo, cerrando esta interfaz.

Postcondiciones: N/A

Figuras relevantes:

Figura 1.2.2: Interfaz del tutorial del juego

IV. Cambiar ajustes



Nombre: Cambiar ajustes

Actores: Jugador

Precondiciones: Estar en el menú principal

Requisitos no funcionales: Ninguno

Flujo de eventos:
1. El usuario pincha en el botón de ajustes del menú principal (figura 1.2.1), abriendo una nueva pantalla de configuración (figura 1.2.3).

2. El jugador altera las opciones disponibles que tiene, pueden ser:
- Idioma
- Resolución de pantalla
- Modo de ventana
- Calidad de gráficos
- Volumen de audios
- Volumen de efectos de sonido
- Activación o desactivación de sincronización vertical
- Tasa de fotogramas por segundo (si el ajuste anterior está desactivado)

[Tras realizar sus modificaciones, pulsa en aceptar]

3A. Se guardan los cambios de ajustes y se redirige al menú principal.

[Tras realizar sus modificaciones, pulsa en volver]

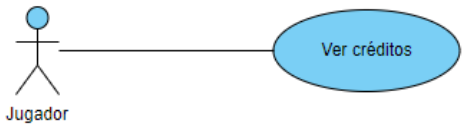
3B. Se vuelve al menú principal sin realizar cambios.

Postcondiciones: Se realizan los cambios de ajustes en el juego, según lo deseado.

Figuras relevantes:

Figura 1.2.3: Pantalla de ajustes

V. Ver créditos



Nombre: Ver créditos

Actores: Jugador

Precondiciones: Estar en el menú principal

Requisitos no funcionales: Ninguno

Flujo de eventos:
1. El usuario pincha en el botón de créditos del menú principal (ver figura 1.2.1), abriendo una nueva interfaz (ver figura 1.2.4).

[Si el jugador pincha en una de las flechas disponibles en la parte inferior]

2A. Se avanza o retrocede una página en la interfaz.

[Si este cambio hace que se llegue a la primera o última página del manual]
2AA. Se bloquea la flecha correspondiente para no poder avanzar más en ese sentido.

[Si este cambio hace que se deje de estar en la primera o última página del manual]

2AB. Se desbloquea la flecha correspondiente para volver a permitir avanzar en ese sentido.

[Si el jugador pincha en el botón de volver]

2B. Se abre el menú principal de nuevo, cerrando esta interfaz.

Postcondiciones: N/A

Figuras relevantes:

Figura 1.2.4: Pantalla de créditos

Tablas

Tabla I: Contraste entre editores de texto

	Drive	LaTeX
Usabilidad	Fácil de usar y manejar, insertar tablas e imágenes es tarea trivial. La portada de la memoria es proporcionada por la universidad en la *web*, la cual se puede importar muy fácilmente.	Requiere familiarizarse con comandos exclusivos para poder crear distintas figuras, como tablas e imágenes. La portada de la memoria disponible en *eGela* no puede importarse fácilmente.
Visualización y guardado	Los contenidos escritos se guardan y visualizan al instante.	Los contenidos escritos se guardan al instante, pero se requiere compilar el archivo para visualizar los cambios. Esto puede reducir la velocidad a la que se escribe la memoria.
Apariencia	Los diseños de documentos pueden hacerse repetitivos y se requiere uso ingenioso de las funcionalidades limitadas del *software* para hacerlos destacar. Esto incluye tener que crear tablas invisibles para alinear figuras en columnas distintas de una página.	Permite crear una gran variedad de diseños de documentos gracias a su gran variedad de comandos exclusivos.

Tablas II: Personajes del juego

Tabla II.I Robots

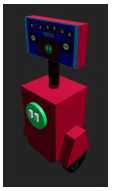
Nombre	Dureza	Velocidad	Peso (fuerza)	Habilidades especiales	Aspecto
Simple	Baja	Moderada	1	No tiene. Solo se desplaza hacia la zona objetivo y dispara a torretas cercanas en su fila como cualquier otro robot.	
Medio	Media	Moderada	2	No tiene. Solo se desplaza hacia la zona objetivo y dispara a torretas cercanas en su fila como cualquier otro robot.	
Duro	Alta	Moderada	3	No tiene. Solo se desplaza hacia la zona objetivo y dispara a torretas cercanas en su fila como cualquier otro robot.	
Líder	Baja	Alta	1	No tiene. Solo se desplaza hacia la zona objetivo y dispara a torretas cercanas en su fila como cualquier otro robot.	
Bomba	Media	Moderada	2	Detona al morir, destruyendo torretas en un rango de 3x3 casillas centrado sobre la entidad.	
Bomba Radar	Muy alta	Baja	4	Además de detonar al morir, también puede hacerlo prematuramente si tiene al menos dos torres en su rango de explosión.	
Ocultador	Muy baja	Baja	3	Es inmune a todo daño mientras se desplaza, pero está expuesto a daños como cualquier otro enemigo mientras ataca.	

Tabla II.II Torretas

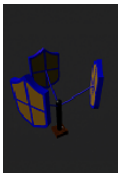
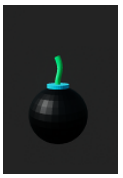
Nombre	Desblq.	Coste	Recarga	Dureza	Función	Aspecto
Cañón	Usable por defecto	50 ptos. de energía	Rápida (5s)	Muy baja	Dispara proyectiles al enemigo más cercano en su fila. Prefiere descansar apuntando hacia abajo si no tiene nada que hacer.	
Panel Solar	Usable por defecto	20 ptos. de energía	Rápida (5s) *Empieza recargada en el nivel.*	Muy baja	Genera 10 puntos de energía a los 3,75s de ser colocada. A partir de ahí, genera 10 extra cada 15s. El jugador debe pinchar en ella para recolectar.	
Mina	Acabar nivel 1	10 ptos. de energía	Lenta (20s)	Muy baja (fase 1) Invencible (fase 2)	Espera 20s tras ser colocada sin hacer nada (fase 1). Tras esto, se prepara y puede explotar, eliminando todos los robots en su casilla (fase 2).	
Cañón Doble	Acabar nivel 2	100 ptos. de energía	Rápida (5s)	Muy baja	Fundamentalmente es un cañón corriente, pero con el doble de cadencia de disparo.	
Panel Solar Doble	Acabar nivel 4	35 ptos. de energía	Rápida (5s) *Empieza recargada en el nivel.*	Muy baja	Fundamentalmente es un panel solar corriente, pero generando 15 puntos de energía en vez de 10 por ciclo.	
Escudo	Acabar nivel 5	20 ptos. de energía	Lenta (20s)	Alta	Se queda quieto en su sitio absorbiendo daños de enemigos.	
Pistola Láser	Acabar nivel 7	110 ptos. de energía	Rápida (5s)	Muy baja	Fundamentalmente es un cañón corriente, pero sus proyectiles penetrantes dañan hasta los 3 objetivos más cercanos en su fila.	
Bomba	Acabar nivel 9	70 ptos. de energía *(colocable sobre robots)*	Rápida (5s)	Invencible	Explota a los pocos segundos de ser colocada, eliminando a todo robot en un área de casillas 3x3 centrada en ella.	

Tabla III: Niveles del juego

Los datos de niveles y desbloqueos de torres se almacenan en archivos .json (ficheros de texto con datos) dentro del directorio principal del juego. Sus contenidos de forma resumida se traducen en lo siguiente:

Nivel	Torres disponibles por el usuario en el nivel	Oleadas totales	Grandes oleadas	Enemigos a enfrentarse
1	**Panel solar**, **cañón**	10	1	**Simple**
2	Panel solar, cañón, **mina**	20	2	Simple
3	Panel solar, cañón, mina, **cañón doble**	10	1	Simple, **medio**
4	Panel solar, cañón, mina, cañón doble	20	2	Simple, medio
5	Panel solar, cañón, mina, cañón doble, **panel solar doble**	10	1	Simple, **bomba**
6	Panel solar, cañón, mina, cañón doble, panel solar doble, **escudo**	10	1	Simple, medio, **ocultador**
7	Panel solar, cañón, mina, cañón doble, panel solar doble, escudo	10	1	Simple, medio, **duro**
8	Panel solar, cañón, mina, cañón doble, panel solar doble, escudo, **pistola láser**	20	2	Simple, medio, bomba, ocultador
9	Panel solar, cañón, mina, cañón doble, panel solar doble, escudo, pistola láser	20	2	Simple, medio, duro, bomba, **bomba radar**
10	Panel solar, cañón, mina, cañón doble, panel solar doble, escudo, pistola láser, **bomba**	20	3	Simple, medio, duro, bomba, bomba radar, ocultador

Nota: La primera vez que una torreta o un enemigo aparece en un nivel figura en **negrita.**
Nota 2**:** La segunda columna hace referencia a qué torretas tiene el jugador la primera vez que llega a ese nivel. Una vez completado el juego puede repetir todos los niveles con todas las torres en posesión.

Los desbloqueos están diseñados de forma que las nuevas torretas desbloqueadas puedan hacer frente a los enemigos más poderosos que aparecen en los siguientes niveles. Por ejemplo:

- El cañón doble se desbloquea al final del nivel 2, justo antes de que el robot medio empiece a aparecer a partir del 3. De esta forma, el jugador puede usar la nueva torreta con cadencia de disparo superior para causar más daños al nuevo enemigo, quien tiene más puntos de vida que los oponentes corrientes contra los que se ha enfrentado hasta el momento.

- El escudo se desbloquea al final del nivel 5, justo antes de que el robot ocultador empiece a aparecer a partir del 6. De esta manera, el usuario tiene una manera de forzar al nuevo enemigo a destapar su punto débil de forma eficaz (los escudos pueden absorber daños durante más tiempo).

Tabla IV: Plan de pruebas

Prueba	Descripción	Cómo comprobarlo
Redirecciones entre menús	Cuando el usuario tiene abierto un menú y pincha en uno de sus botones, se hace la acción correcta.	Verificar que todas y cada una de las interfaces ejecutan la lógica correspondiente según el contexto. Las posibles acciones pueden ser: - Redirecciones entre menús. - Ediciones en archivos de guardado. - Ediciones en archivos de ajustes. - Cargar otro mapa e inicializarse correctamente.
Fase de selección de torres	El usuario es capaz de elegir entre sus torretas existentes y llevar hasta 6 al nivel.	Comprobar que todas las acciones que puede tomar el usuario en la fase de selección funcionan como es debido. Estas incluyen: - Pinchar en una tarjeta de torreta para llevarla a su selección si el usuario no tiene ya 6 elegidas. Debe ocupar el primer hueco libre empezando por la izquierda de su lista actual. - Deseleccionar una torreta para llevarla de vuelta a su sitio. Además, rotar el resto de selecciones a su derecha un espacio para reorganizar la lista. (Se verifican casos de prueba en los que hayan y no hayan tarjetas a su derecha para mover). - El botón que permite empezar la partida solo se activa cuando hayan 6 torres elegidas o todas las desbloqueadas. Además, se puede desactivar si esto se deja de cumplir. - Tras la selección, verificar que las torretas que se han llevado al nivel y las que se muestran en la otra interfaz para colocar en casillas son las mismas.
Colocación de torretas en las casillas de juego	El jugador puede pinchar en la tarjeta de torreta correspondiente seguido la casilla donde ponerla.	Asegurarse que el proceso funciona. Incluye comprobar estos pasos: - Cuando el jugador pincha en la torreta solo se permite marcarla (seleccionarla) si tiene suficientes recursos para su coste y no está en recarga (comprobando en cuatro casos de prueba distintos en los que ambas condiciones se cumplen, una en concreto se verifique, o ninguna). - Teniendo seleccionada una torreta, pinchando en una casilla se coloca ahí. Esto causa su pago, desmarque y reinicio de su proceso de recarga (visto en IU y procesado en el mando). - En el caso de haber un enemigo u otra torreta ya en la casilla, o no había selección marcada; entonces no se permite colocarla (la excepción siendo la bomba, que permite colocarse encima de un robot pero no sobre otra torreta).

Destrucción de entidades	Los personajes desaparecen del mundo cuando deberían.	Comprobar que los personajes son destruidos en los siguientes casos: - La salud de una entidad llega a 0. - El personaje detona. - El jugador tiene seleccionada la TNT y pincha en la casilla con la torreta correspondiente a eliminar. Verificar jugando que las entidades no se eliminan por otros motivos sin sentido.
Interacción entre entidades	Los personajes interactúan entre sí como es debido.	Verificar las siguientes condiciones: - Las torretas y los robots se atacan entre ellos (y solo entre ellos). Una entidad no ataca si no tiene un enemigo a su vista (las torretas ven toda su fila, los robots media casilla delante suya). - Los proyectiles o detonaciones no dañan a aliados de su causante. - Los robots no se atraviesan si uno alcanza a otro por detrás, en su lugar, se encolan en forma de fila y todos los de atrás imitan al que los encabeza en las acciones que realiza (moverse a una misma velocidad o atacar a una torreta).
Acciones individuales de entidades	Comprobar que cada personaje actúa como es debido de forma independiente.	Verificar las siguientes condiciones: - Un robot avanza hacia delante si no ve enemigos. - Un panel solar produce energía periódicamente y es recolectable por el jugador al pinchar sobre él. Esta energía caduca a los 10s de no ser adquirida. - Un robot con forma de bomba detona cuando pierde todos sus puntos de vida. La torre bomba, en cambio, empieza su proceso de detonación nada más se coloca. - Un robot bomba radar detona prematuramente si tiene dos torres en su rango de explosión, dañándolas en su caso. - Una mina tiene un periodo de vulnerabilidad al prepararse para poder explotar y puede ser eliminada por robots en esa fase. Tras haber sobrevivido este periodo inicial, pasa a ser indetectable por enemigos y detona cuando uno la pisa. - Un robot ocultador es inmune a todo daño (explosiones y/o proyectiles) mientras se desplaza, pero vulnerable cuando está detenido para atacar.

Condiciones de fin de juego	Las condiciones de victoria y derrota funcionan como es debido.	Comprobar jugando: - Si un robot (de cualquier tipo) llega a la esquina izquierda del mapa, se detiene el juego y el jugador pierde la partida. Este puede reiniciar el nivel usando la interfaz visual que aparece tras esto. - Si se elimina al último robot del nivel (y solo cuando este muere), se da la recompensa del nivel correcta según el valor del nivel y la cantidad de veces que ha completado el juego: ● Una nota si es el último nivel del juego. ● La torreta desbloqueable correspondiente al nivel en cuestión si existe y no se había completado el juego previamente. En este caso, se obtiene dicha torre y se guarda en el archivo de guardado del juego. ● Un trofeo en cualquier otro caso.
Animaciones	Cada personaje realiza su animación visual coherente según el contexto.	Comprobar cada animación por separado, esto incluye: Cada robot: - Moverse hacia adelante si no hay enemigos (girar sus ruedas o escudos semiesféricos según el tipo). - Iluminar sus ojos y hacer aparecer láseres si tiene una torreta en rango. Robot ocultador: - Esconderse detrás de sus escudos mientras se mueve y mostrarse al atacar. Torres disparadoras: - Realizan sus animaciones de apuntado respectivas cuando empiezan a ver enemigos en su fila. - Ejecutan sus animaciones de disparo tras su apuntado y las sincronizan con la generación de sus proyectiles. - Hacen su desapuntado, si no quedan enemigos de su fila. Paneles solares: - Iluminarse y hacer sonar un efecto de sonido al generar energía. - Apagarse y hacer sonar otro efecto de sonido al ser recolectada o pasar 10s. Explosiones: - Cambiar el color del cuerpo de la entidad que va a explotar a rojo y negro de manera sucesiva. Crear partículas especiales y hacer sonar un efecto de sonido al explotar.

		Robots y torre escudo: - Cambiar la apariencia cada vez que pierden una parte porcentual concreta de su vida. Robot ocultador -> Cada 50% de pérdida de vida total El resto -> Cada 33,33% de pérdida de vida total Toda entidad: - Se anima su muerte con su efecto de sonido cuando la vida de esta llega a 0 (no aplicable si detonan).
Apariciones de oleadas	Los robots que aparecen en un nivel se generan en los contextos previstos, y sus cantidades y tipos encajan con los datos del archivo .json del nivel respectivo.	Verificar: - La oleada inicial aparece a los 26s de empezar el juego (sincronizado con la música). El resto de ellas aparecen a los 20-30s (depende del caso). - Las siguientes oleadas pueden ser adelantadas y aparecen al instante de cuando se pierde el 50% de vida de los robots que forman la actual. Esto solo debe ocurrir si la siguiente a crear no se trata de una gran horda. En caso de que lo sea, se puede adelantar su aparición solo si no quedan enemigos en el campo de batalla. - Si va a aparecer una gran horda, entonces se avisa correctamente al usuario por pantalla mediante una interfaz y se cambia la música. Además, se muestra un mensaje adicional si se trata de la última oleada del nivel. - Los enemigos aparecen repartidos por filas, de manera que una de ellas nunca tenga más de un robot en comparación con otra. Además, cada robot generado no se solapa con otros de oleadas previas (verificando la generación de nuevos enemigos cuando ya haya antiguos en la propia zona de aparición inicial, forzando a los nuevos a aparecer más atrás de lo normal), ni con otros robots creados en la propia oleada (comprobando hordas que generen al menos 6 enemigos, para que las filas generen más detrás de los iniciales). - Las cantidades de robots por cada tipo que aparecen en las oleadas son coherentes con los datos de su nivel (quienes indican los tipos de enemigos disponibles a crear, además del peso total admitido por cada horda y la distribución de probabilidades de generación por cada tipo). - Cada vez que se invoca una oleada, se avanza la barra de progreso adecuadamente y completándose cuando aparezca la última horda del nivel. Si se invoca una gran oleada, se alza la bandera correspondiente en la barra.

Tablas V: Orígenes de recursos externos del juego empleados

Música

Autor	Nombre del recurso	Uso en el juego	Enlace
StudioKolomna	Risk	Música de tensión reproducida durante la infiltración de un robot en la mina del jugador.	https://pixabay.com/es/music/titulo-principal-risk-136788/
Good_B_Music	Cinematic Story	Es la música de victoria de nivel.	https://pixabay.com/es/music/titulo-principal-cinematic-story-loop-2-197104/
Gamedev.tv team	Asset Pack - FPS Music Tracks	El resto del repertorio de música.	https://www.gamedev.tv/asset-packs/asset-pack-fps-music-tracks

Efectos de sonido

Autor	Nombre del recurso	Uso en el juego	Enlace
SUBMORITY	Hit low Gravity Absorber	En la cuenta atrás antes de empezar la partida.	https://pixabay.com/es/sound-effects/hit-low-gravity-absorber-cinematic-trailer-sound-effects-124761/
EdR	Electronic Impact (Hard)	En la cuenta atrás antes de empezar la partida.	https://pixabay.com/es/sound-effects/electronic-impact-hard-10018/
EdR	Electric fan motor	Encender y apagar paneles.	https://pixabay.com/es/sound-effects/electric-fan-motor-blades-removed-13169/
floraphonic	Wood Smash 4	Algunos sonidos de muerte de torres.	https://pixabay.com/es/sound-effects/wood-smash-4-170420/

u_31vnwfmzt6	Error	Sonido de error por no permitirse la selección de una torre durante una partida (está en recarga o sin recursos suficientes).	https://pixabay.com/es/sound-effects/error-126627/
LordSonny	Cannon Fire	Quitar una torreta de su casilla usando TNT.	https://pixabay.com/es/sound-effects/cannon-fire-161072
deifandubs	Hits on table	Colocar una torreta en una casilla.	https://pixabay.com/es/sound-effects/hits-on-table-149497/
Pixabay	21 gun salute	Sonido de disparo de cañón.	https://pixabay.com/es/sound-effects/21-gun-salute-25444/
Pixabay	Laser compilation 5	Disparos láser (para robots y pistola láser).	https://pixabay.com/es/sound-effects/laser-compilation-05-31597/
Pixabay	Medium explosion	Sonido de toda explosión del juego.	https://pixabay.com/es/sound-effects/medium-explosion-40472/
Pixabay	Beep warning	En las detonaciones previas a explosiones.	https://pixabay.com/es/sound-effects/beep-warning-6387/
Pixabay	Robot noises	Sonidos de muerte de robots.	https://pixabay.com/es/sound-effects/robot-noises-70217/
Pixabay	Enemy detected	La voz que se escucha cuando se aproxima el primer robot del nivel.	https://pixabay.com/es/sound-effects/enemy-detected-103347/
Pixabay	Power down	En algunas animaciones de muerte de torres.	https://pixabay.com/es/sound-effects/power-down-7103/
Pixabay	pen_clicking	Seleccionar o deseleccionar una tarjeta de torre durante una partida.	https://pixabay.com/es/sound-effects/pen-clicking-45543/

Imágenes

Autor	Nombre del recurso	Uso en el juego	Enlace
Joshgmit	[Nombre no encontrado]	La imagen de fondo usada en el menú principal.	https://pixabay.com/es/illustrations/resumen-tecnolog%C3%ADa-antecedentes-5035778/
OpenClipart-Vectors	[Nombre no encontrado]	El trofeo que a veces se otorga al jugador al completar un nivel.	https://pixabay.com/es/vectors/taza-otorgar-premio-raza-carreras-160117/
geralt	[Nombre no encontrado]	La nota que se da al jugador cuando completa el último nivel del juego.	https://pixabay.com/illustrations/paper-stationery-parchment-old-68829/

Fuentes (tipo de letra)

Autor	Nombre del recurso	Uso en el juego	Enlace
GGBot	Unitblock Font	El estilo de letra usado en todas las interfaces gráficas del juego.	https://fontesk.com/unitblock-font/

Materiales para figuras 3D

Autor	Nombre del recurso	Uso en el juego	Enlace
gamedev.tv team	Unreal Material Pack #1	Dar diversos tonos (como por ejemplo metálicos o madereros) a algunas partes de torres y robots.	https://www.gamedev.tv/asset-packs/unreal-material-pack1

Nota 1: Los recursos de pago tienen su enlace marcado en rojo.

Nota 2: Estas tablas indican los orígenes de todos los recursos externos que no estaban disponibles en repositorios gratuitos ofrecidos por *Unreal Engine*. Esto quiere decir que tienen algún tipo de licencia en vigor y muchos de estos no están diseñados para ser usados en el desarrollo de videojuegos exclusivamente. Las restricciones impuestas incluyen impedir su uso de forma inmoral, no poder ser redistribuidos sin modificaciones, no permitir usarlos en logos de marcas corporativas, y/o ser de pago.

Nota 3: Existen más recursos utilizados que no figuran aquí, principalmente usados como decoraciones en los entornos del juego. Estos componentes son completamente gratuitos y están integrados en *Unreal Engine 5*, no necesitando tener que ser importados de fuera.

Figuras

Aspectos visuales (figuras 1.X)

Trozos de interfaces (figuras 1.1.X)

Figura 1.1.1: WB_Boton

Figura 1.1.2: WB_Flecha

Figura 1.1.3: WB_Paginas

Figura 1.1.4: WB_Slider

Figura 1.1.5: WB_SeleccionOpcion

Figura 1.1.6: WB_BotonPausa

Figura 1.1.7: WB_TorreCard

Figura 1.1.8: WB_WidgetEnergia

Figura 1.1.9: WB_BarraDeProgreso

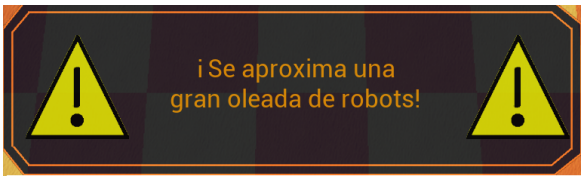


Figura 1.1.10: WB_AvisoOleada

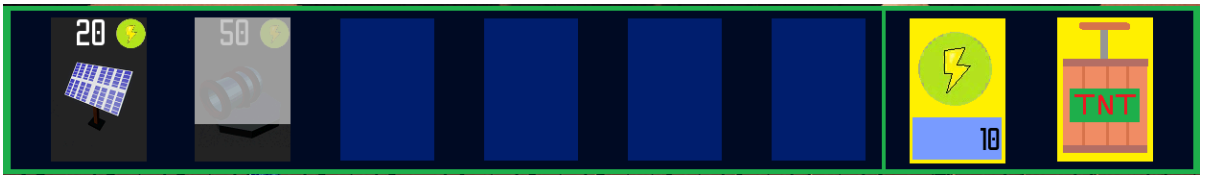


Figura 1.1.11: Parte superior de WB_InterfazEnPartida

Interfaces (figuras 1.2.X)

Figura 1.2.1: Menú principal del juego

Figura 1.2.2: Interfaz del tutorial del juego

Figura 1.2.3: Menú de ajustes

Figura 1.2.4: Pantalla de créditos

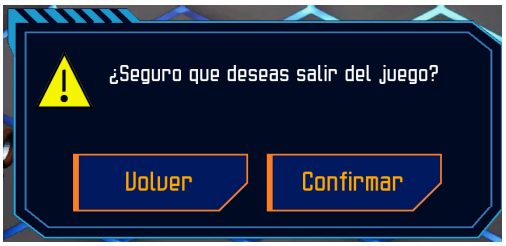


Figura 1.2.5: Pantalla de aviso con dos botones distintos.
Ejemplo: confirmación de salir del juego.

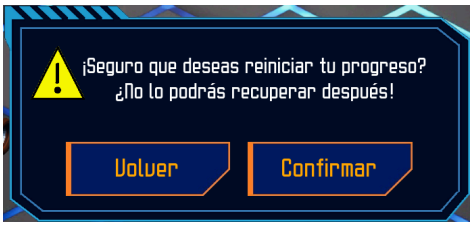


Figura 1.2.6: Pantalla de aviso con dos botones distintos.
Ejemplo: aviso de reinicio de progreso.

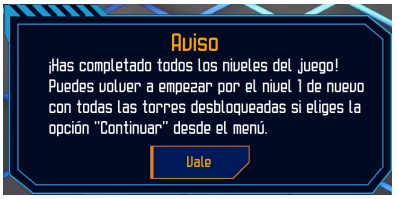


Figura 1.2.7: Pantalla de aviso con botón de confirmación.
Ejemplo: felicitaciones por completar el juego.

Figura 1.2.8: Pantalla de carga

Figura 1.2.9: Interfaz de selección de torres

Figura 1.2.10: Cuenta atrás de inicio de partida

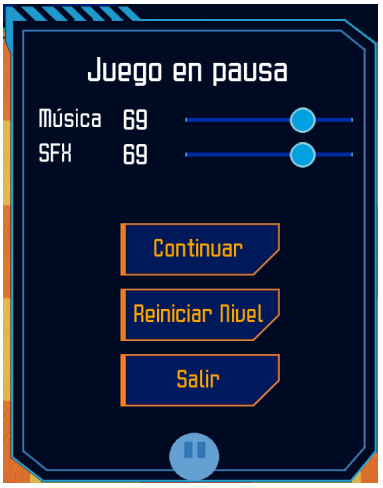


Figura 1.2.11: Menú de pausa

Figura 1.2.12: Pantalla de victoria de nivel

Figura 1.2.13: Pantalla de derrota de nivel

Entornos (figuras 1.3.X)

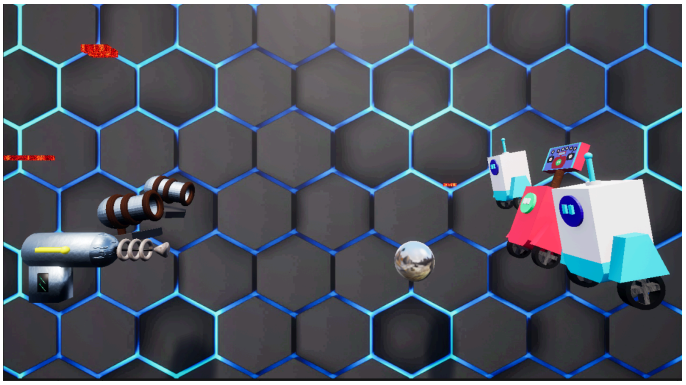


Figura 1.3.1: Mapa (entorno) del menú principal

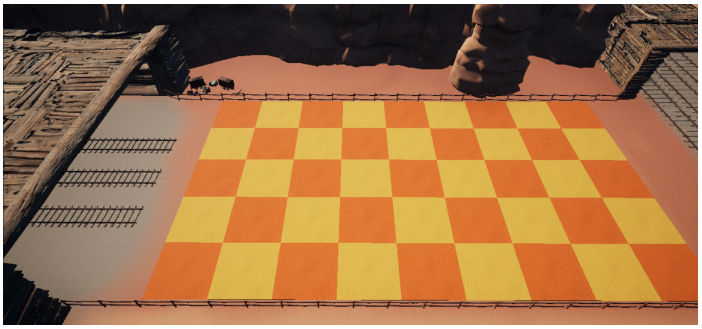


Figura 1.3.2: Mapa (entorno) de partida

Proyectiles (figuras 1.4.X)

Figura 1.4.1: Bola de cañón

Figura 1.4.2: Rayo de pistola láser

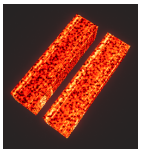


Figura 1.4.3: Proyectiles de robots

Árboles de decisión o *Behavior Trees* (figuras 2.X)

Árboles de decisión de torres (figuras 2.1.X)

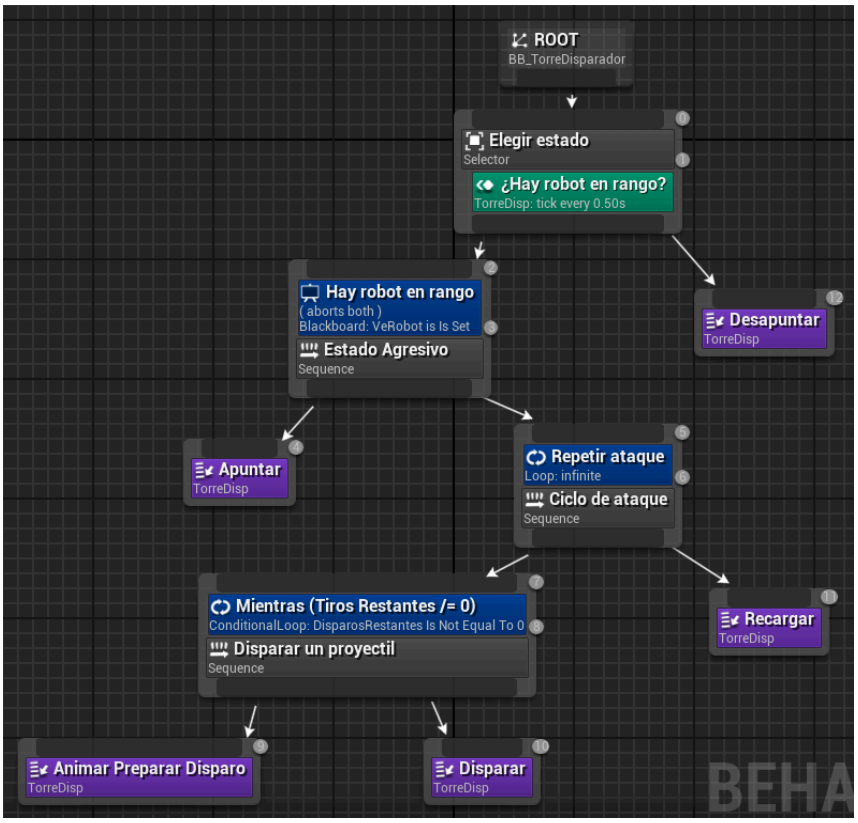


Figura 2.1.1: IA de disparadores (cañón, cañón doble, pistola láser)

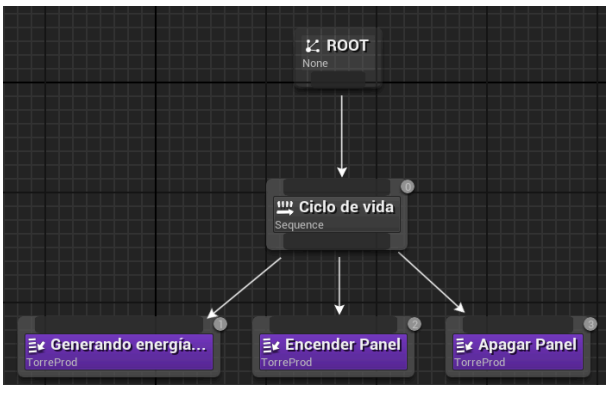


Figura 2.1.2: IA de producidores (panel solar, panel solar doble)

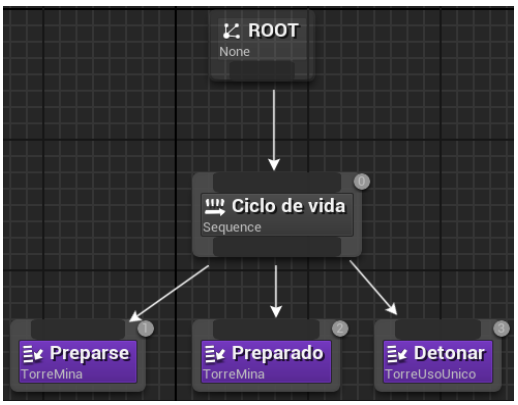


Figura 2.1.3: IA de la mina

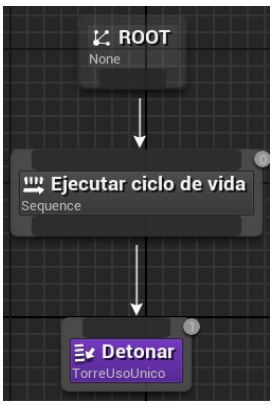


Figura 2.1.4: IA de la bomba

Árboles de decisión de robots (figuras 2.2.X)

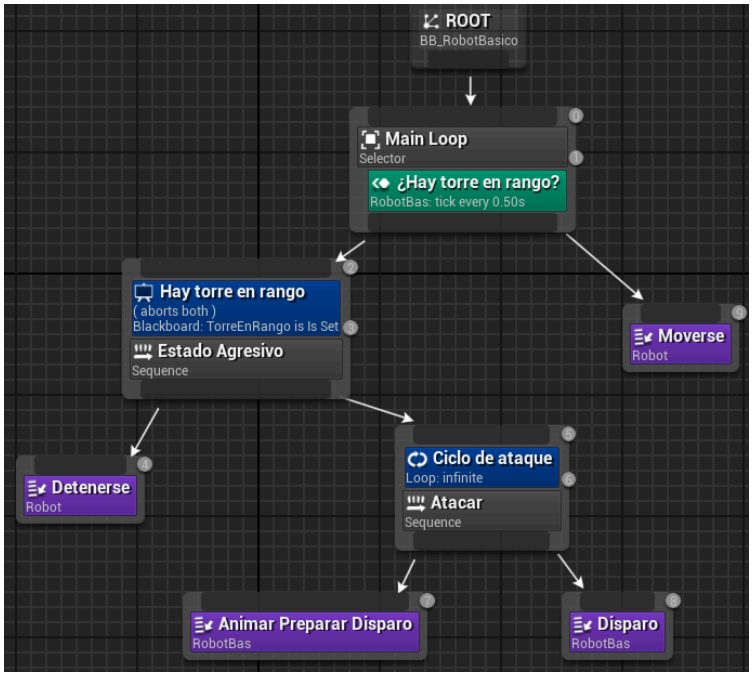


Figura 2.2.1: IA de robots sin habilidades especiales activables (simple, medio, duro, líder, bomba)

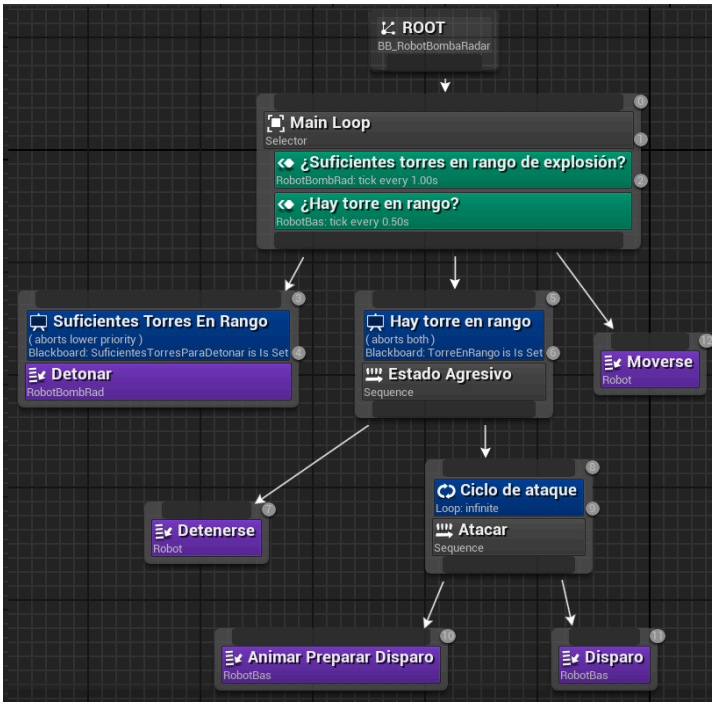


Figura 2.2.2: IA de robot bomba radar

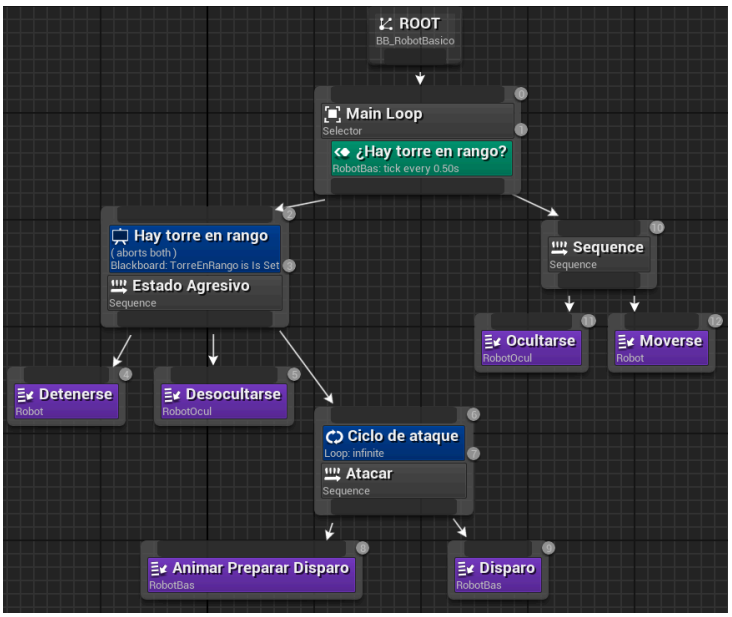


Figura 2.2.3: IA de robot ocultador

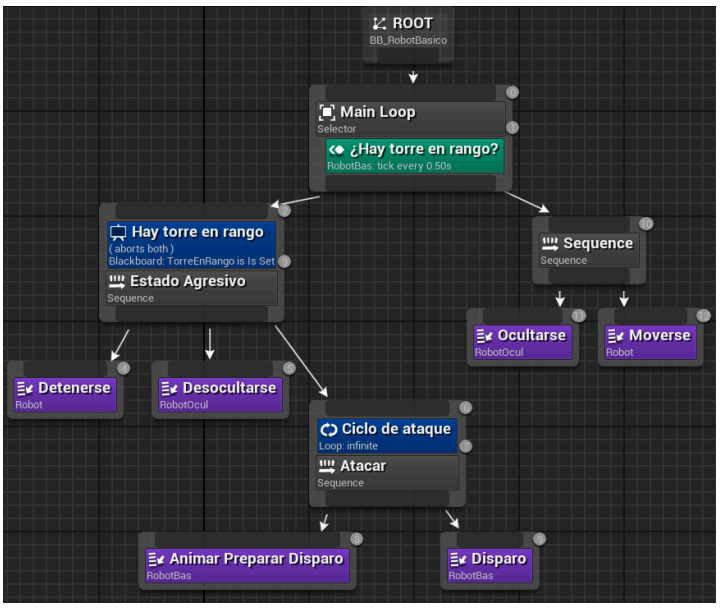


Figura 2.2.4: IA de robots generados en la fase de selección de torres.

Diagramas de clase (figuras 3.X)

Diagramas de clases ya existentes previas al proyecto (figuras 3.1.X)

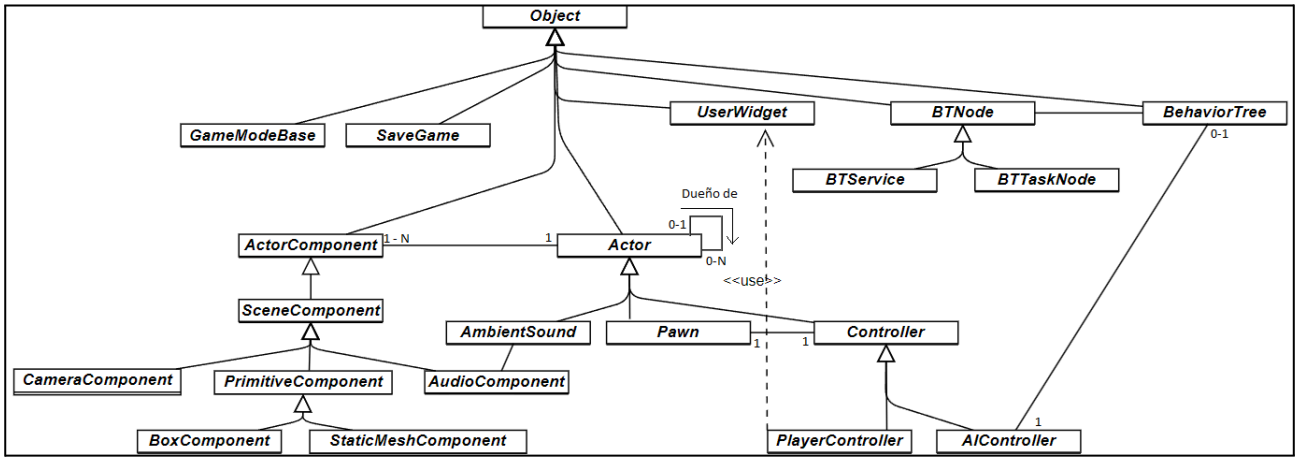


Figura 3.1.1: Diagrama de clases de clases *Unreal Engine* ya predefinidas usadas en el proyecto como base de herencia

Diagramas de clases en relación con el menú principal (figuras 3.2.X)

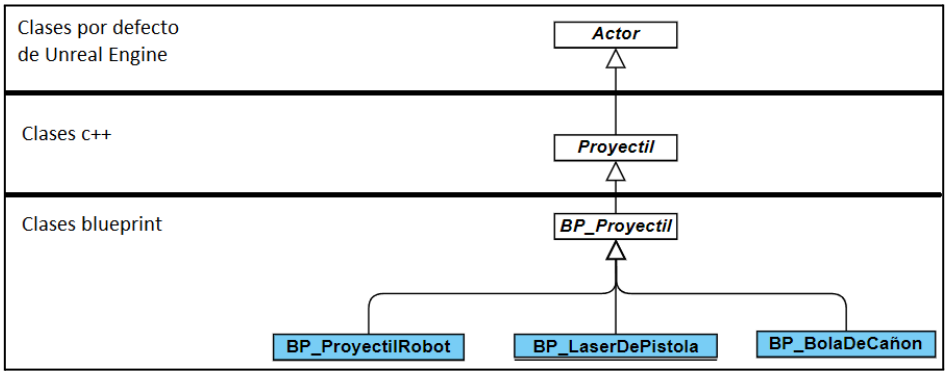


Figura 3.2.1: Diagrama de clases de proyectiles

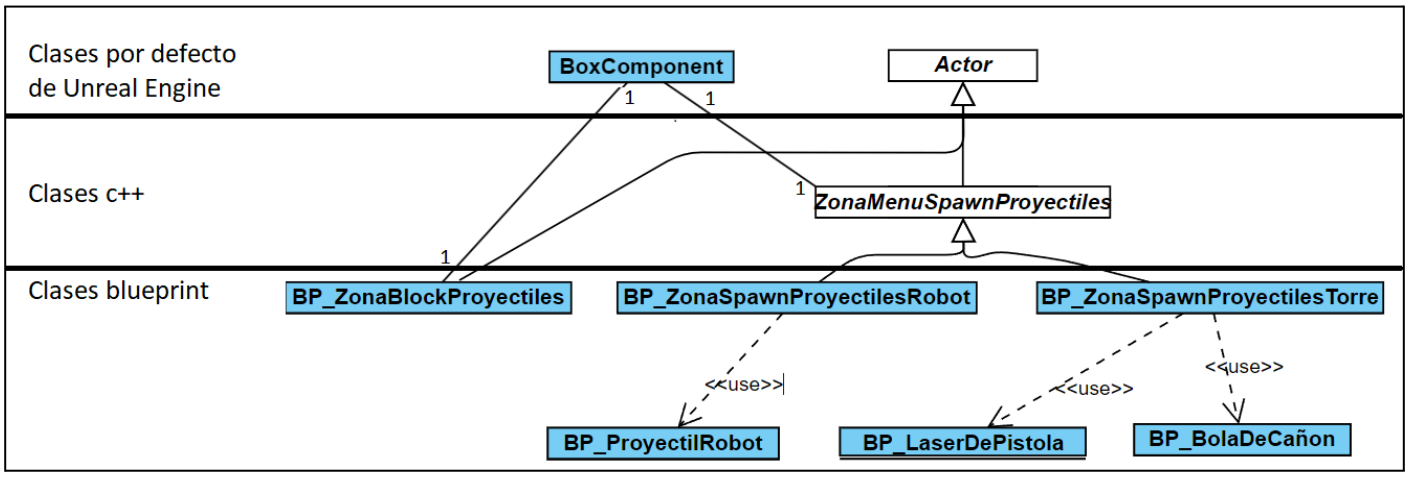


Figura 3.2.2: Diagrama de clases de zonas invisibles colocadas en el menú principal

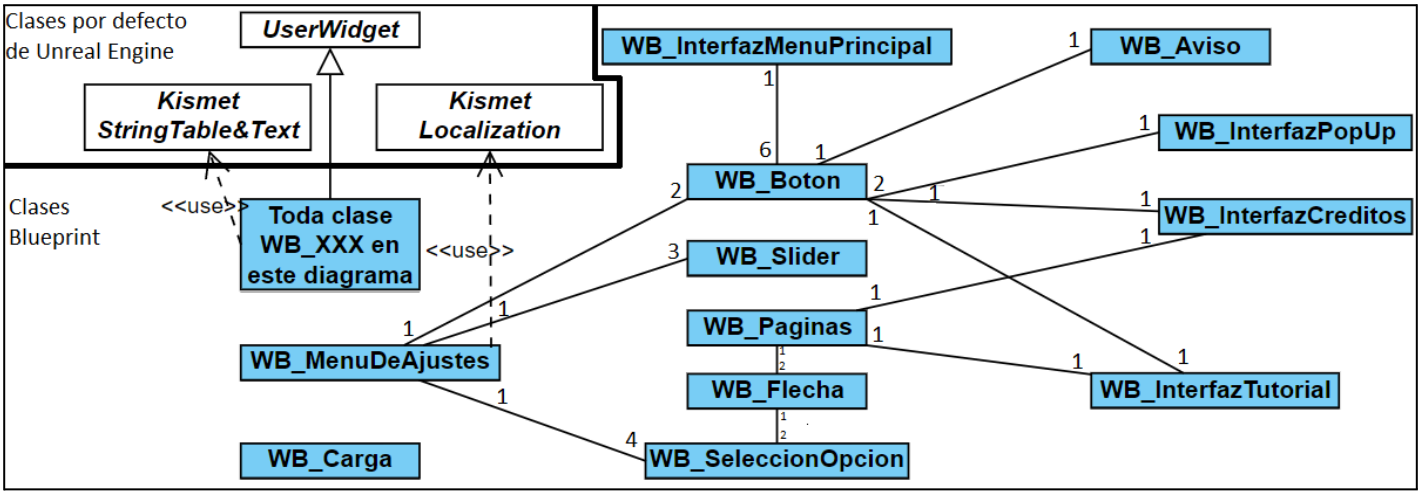


Figura 3.2.3: Diagrama de clases de las interfaces del menú principal

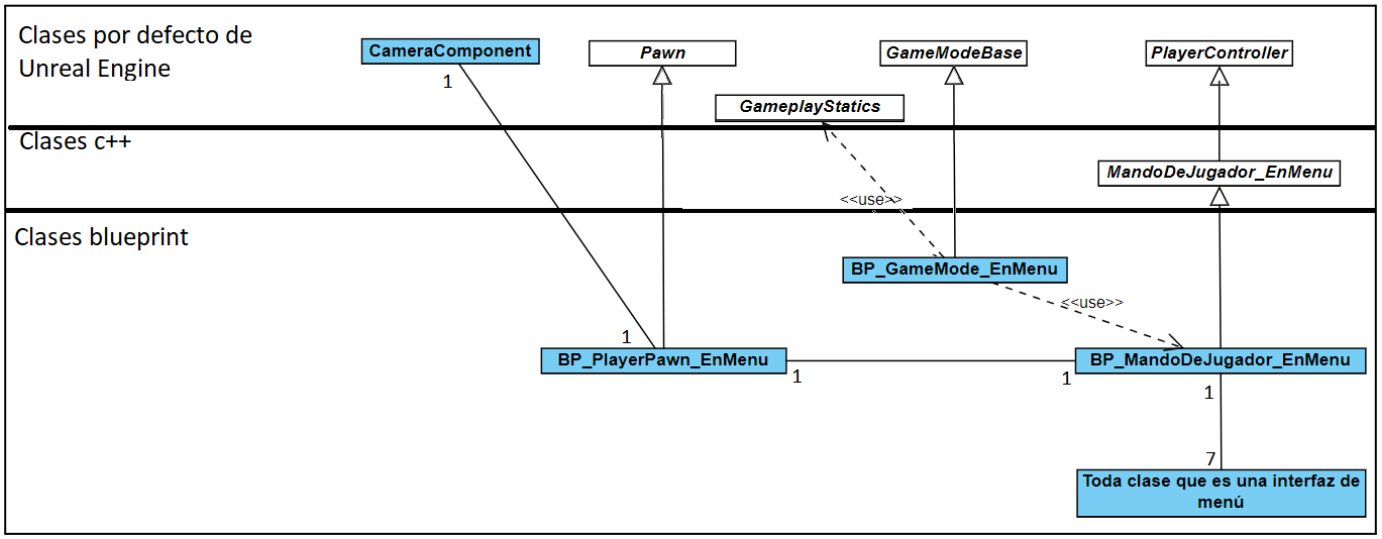


Figura 3.2.4: Diagrama de clases de gestión y control de interfaces del menú principal

Diagramas de clases en relación con la partida en sí (figuras 3.3.X)

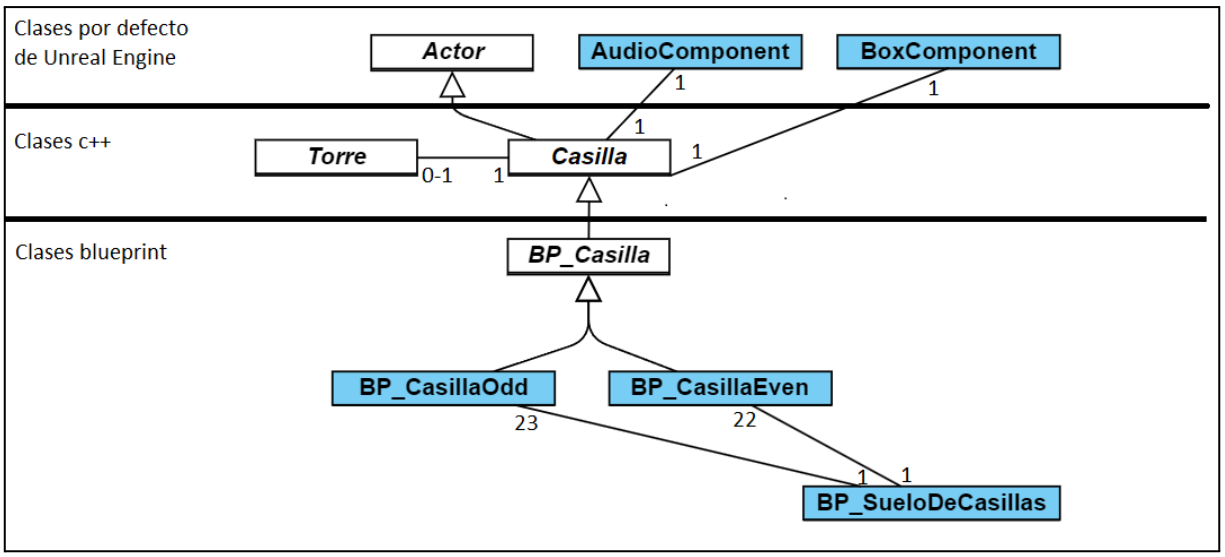


Figura 3.3.1: Diagrama de clases de casillas

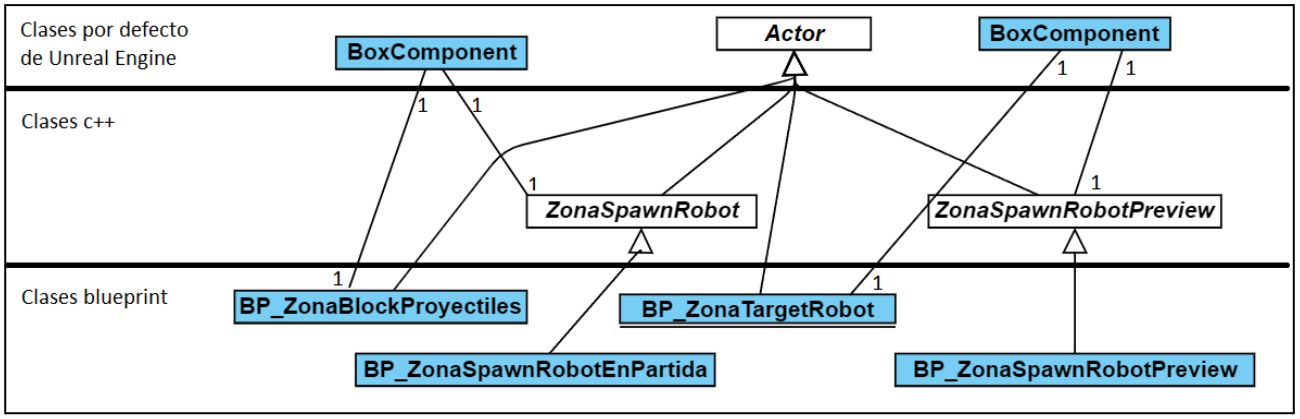


Figura 3.3.2: Diagrama de clases de zonas invisibles

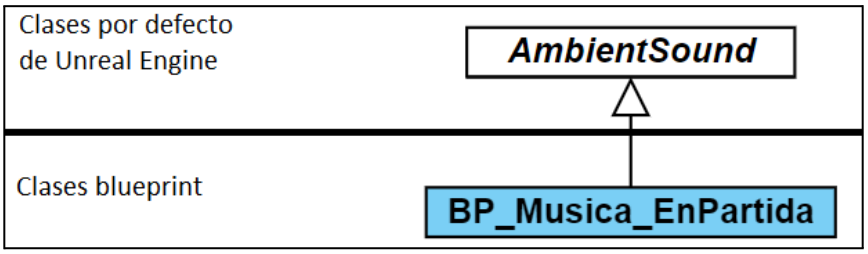


Figura 3.3.3: Diagrama de clases de música

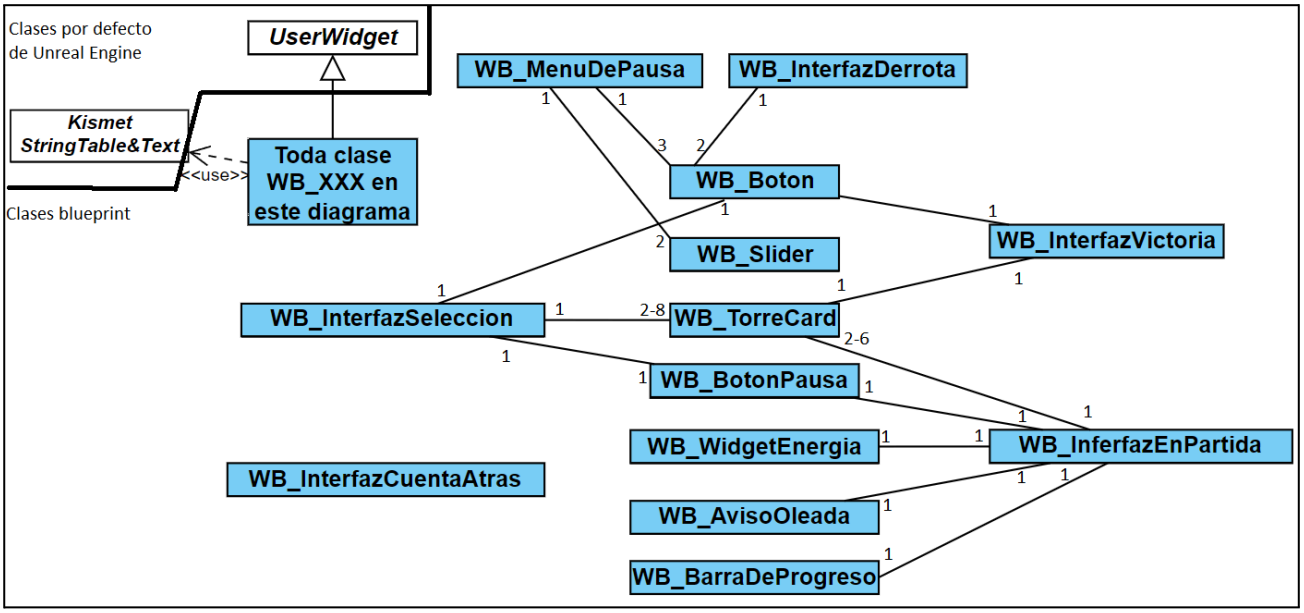


Figura 3.3.4: Diagrama de clases de interfaces gráficas en partida

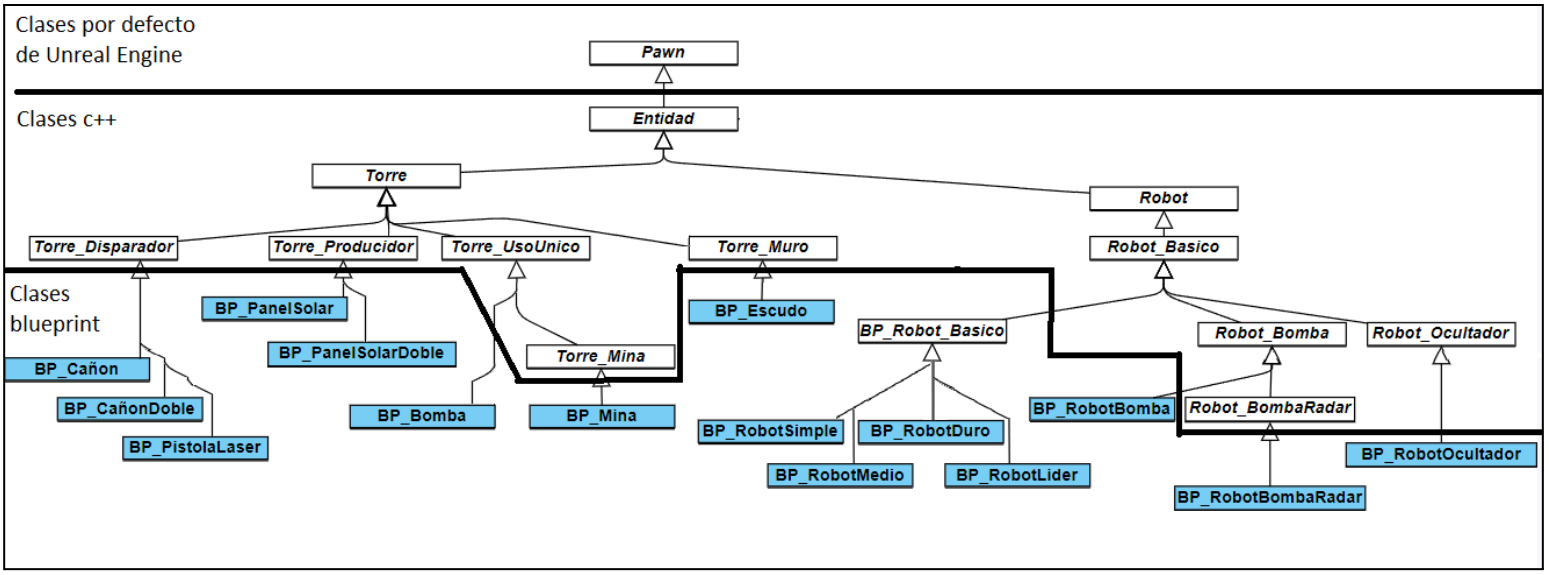


Figura 3.3.5: Diagrama de clases de tipos de entidades

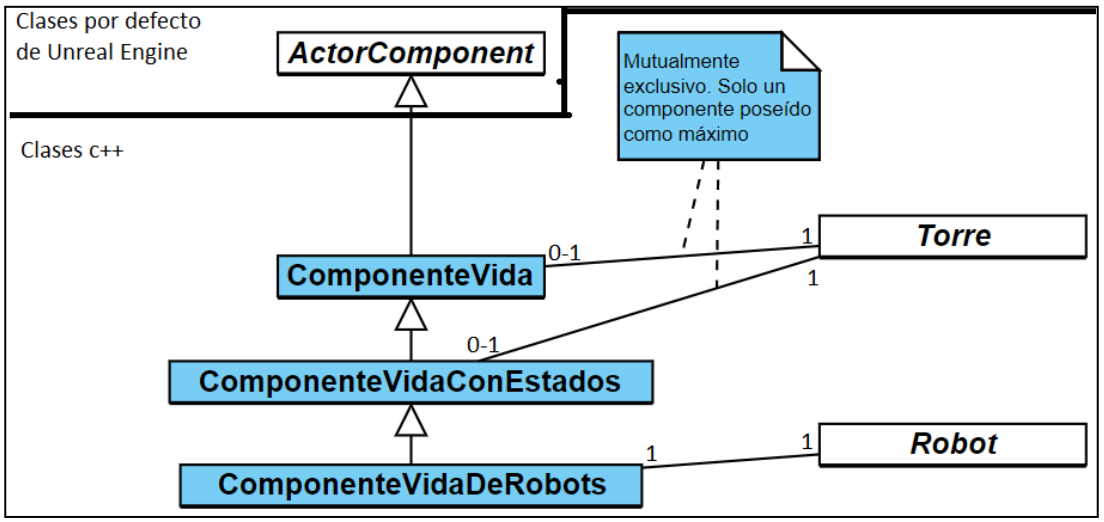


Figura 3.3.6: Diagrama de clases de componentes de vida de entidad

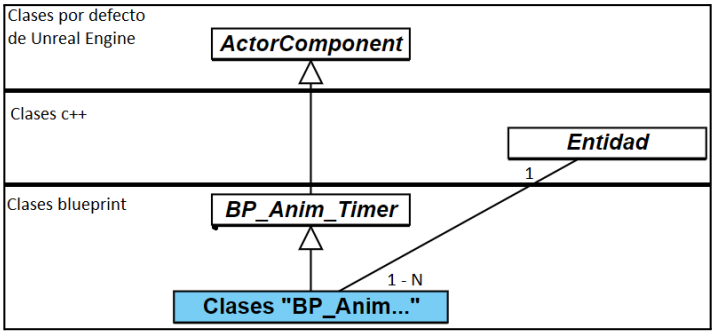


Figura 3.3.7: Diagrama de clases de componentes de animación de entidad

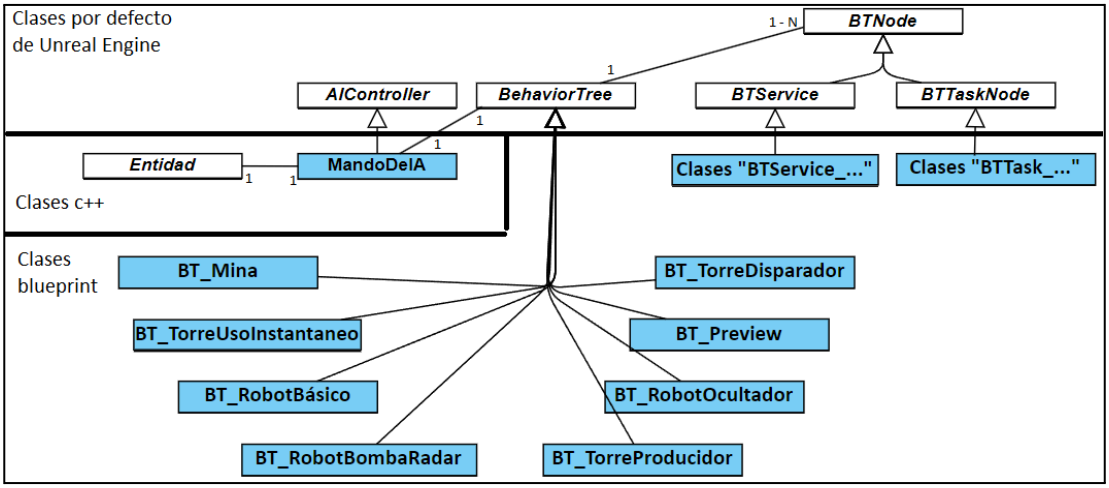


Figura 3.3.8: Diagrama de clases de componentes de IA de entidad

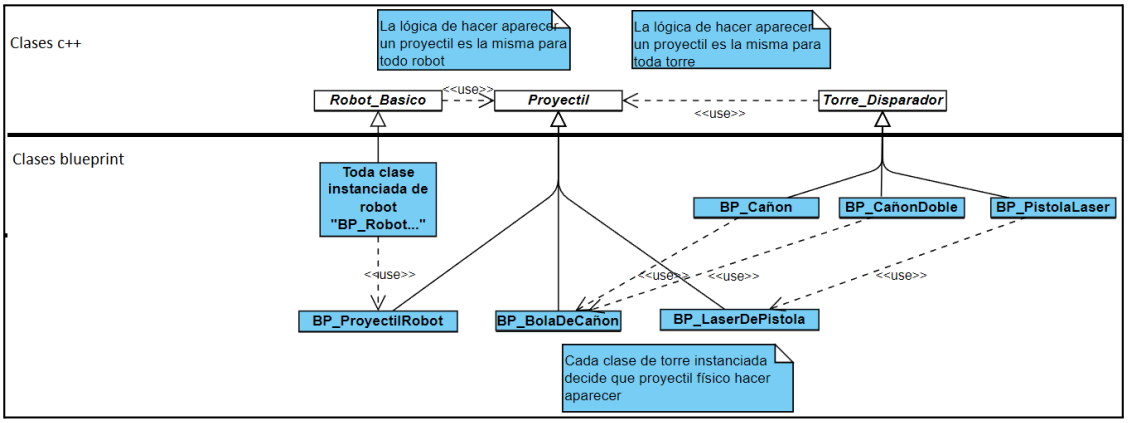


Figura 3.3.9: Diagrama de clases de proyectiles de entidades

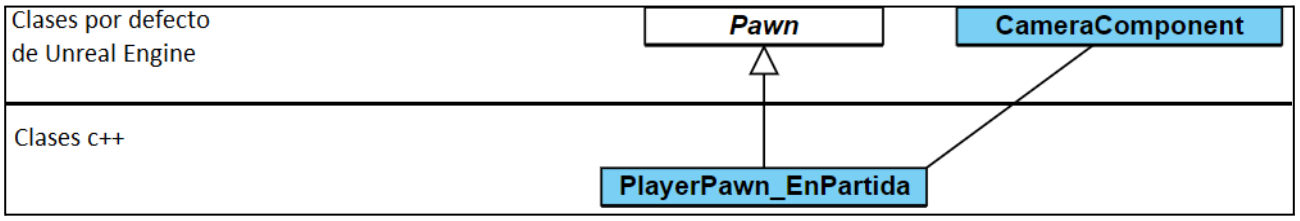


Figura 3.3.10: Diagrama de clases del jugador en la partida

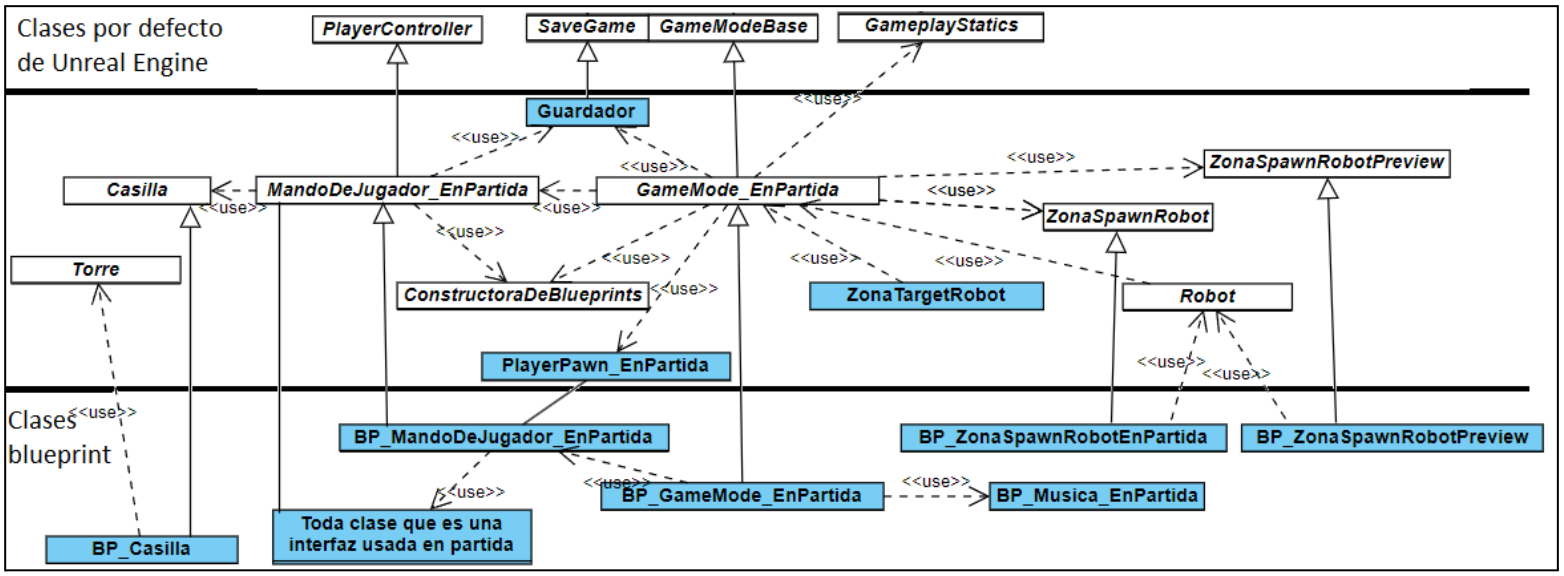


Figura 3.3.11: Diagrama de clases de la gestión de la partida

Diagramas de secuencia (figuras 4.X)

Diagramas de secuencia de funcionalidades clave (figuras 4.1.X)

La mayoría de los objetos instanciados en estas simulaciones heredan de una clase *Blueprint* directa y otra C++ indirecta (superior en la herencia), pudiendo llegar a causar confusión sobre qué lenguaje está siendo ejecutado en cada momento. Con el fin de evitar esta confusión; se denotan las llamadas a una clase *Blueprint* en color azul, código C++ en negro, y funciones por defecto del motor de juego en rojo.

Los diagramas de secuencia presentados a continuación detallan la implementación de los procesos más importantes del juego descritos en el apartado 8.4:

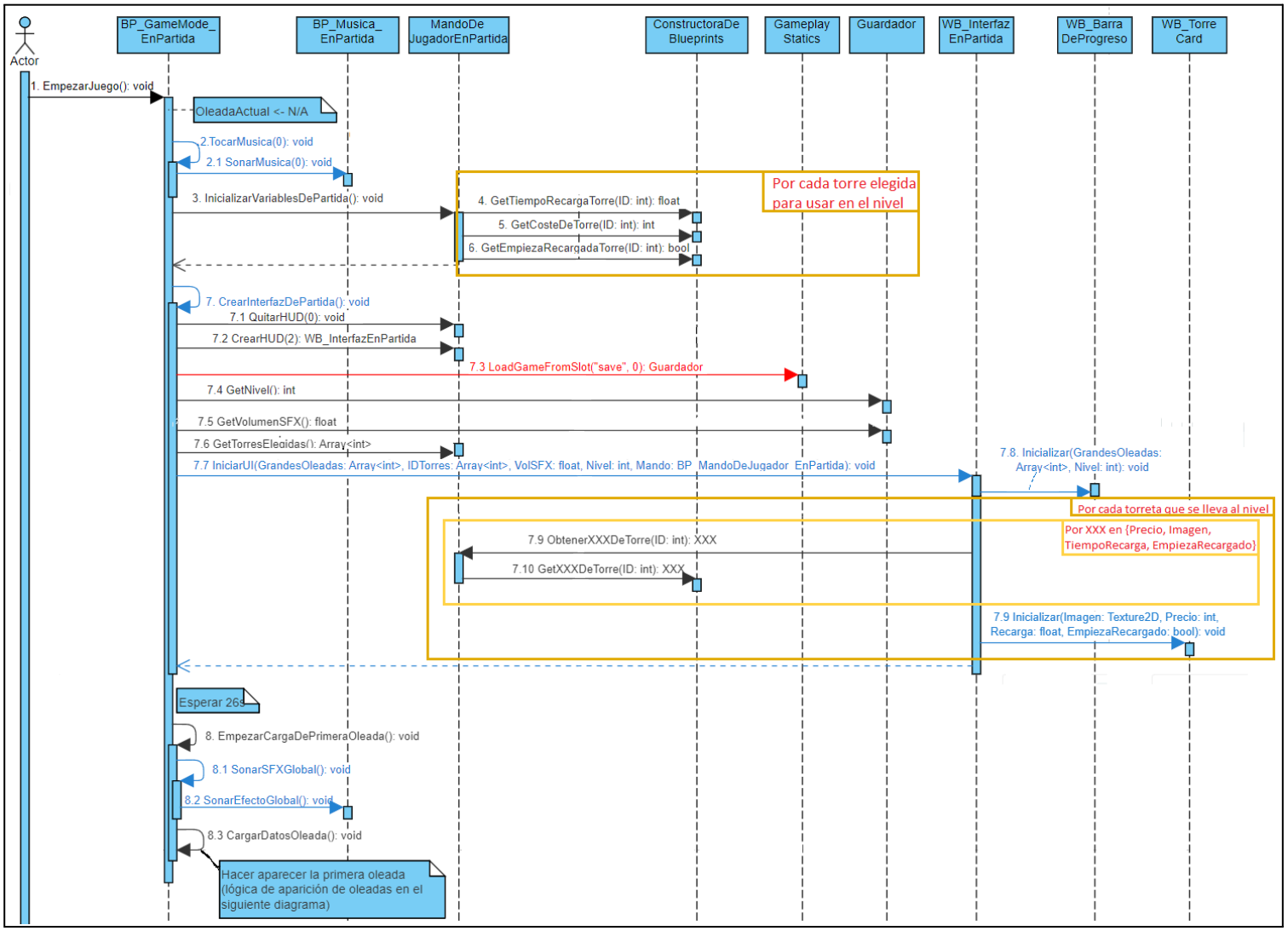


Figura 4.1.1: EmpezarJuego(): void en GameMode_EnPartida

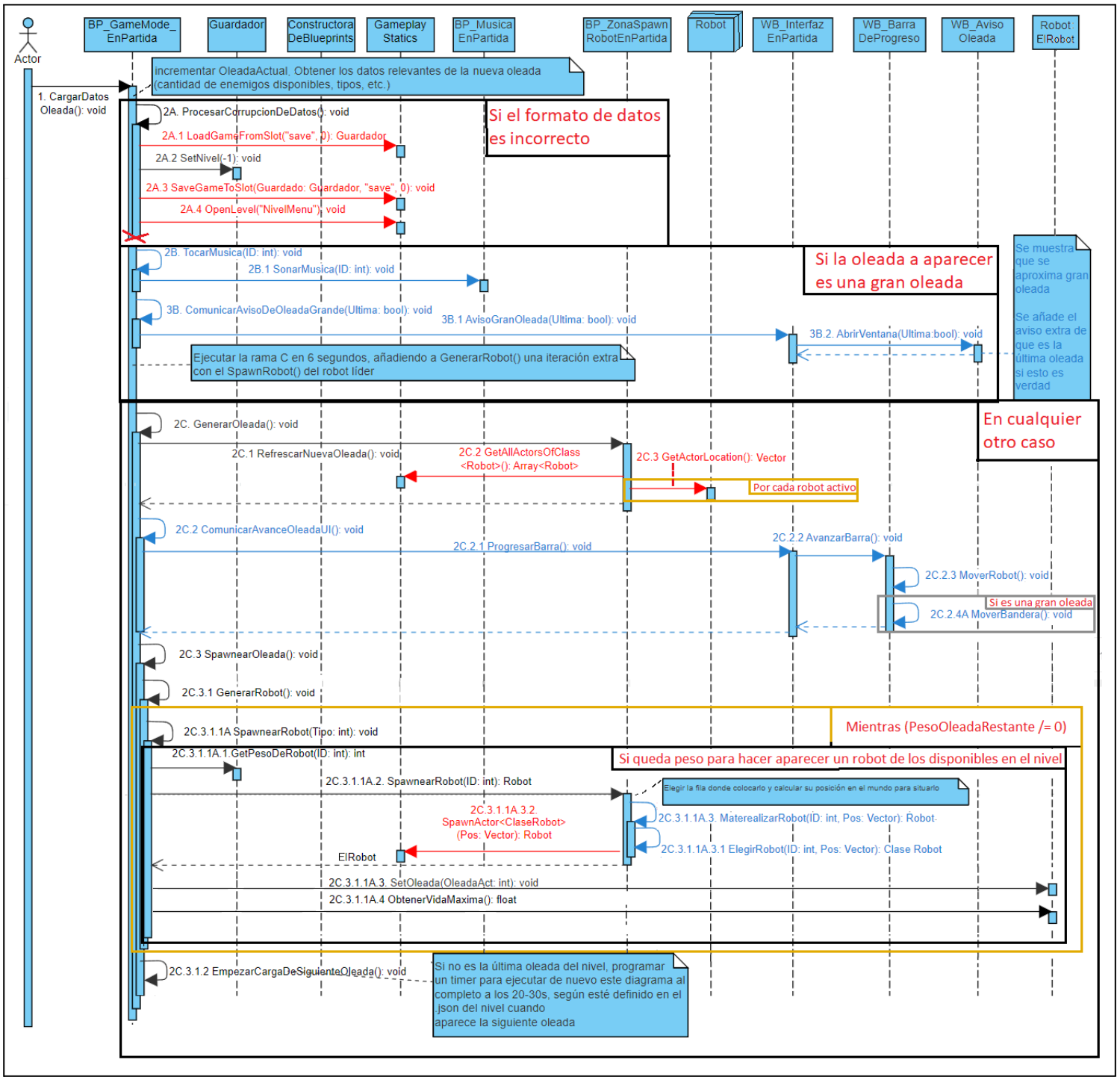


Figura 4.1.2: CargarDatosOleada(): void en GameMode_EnPartida

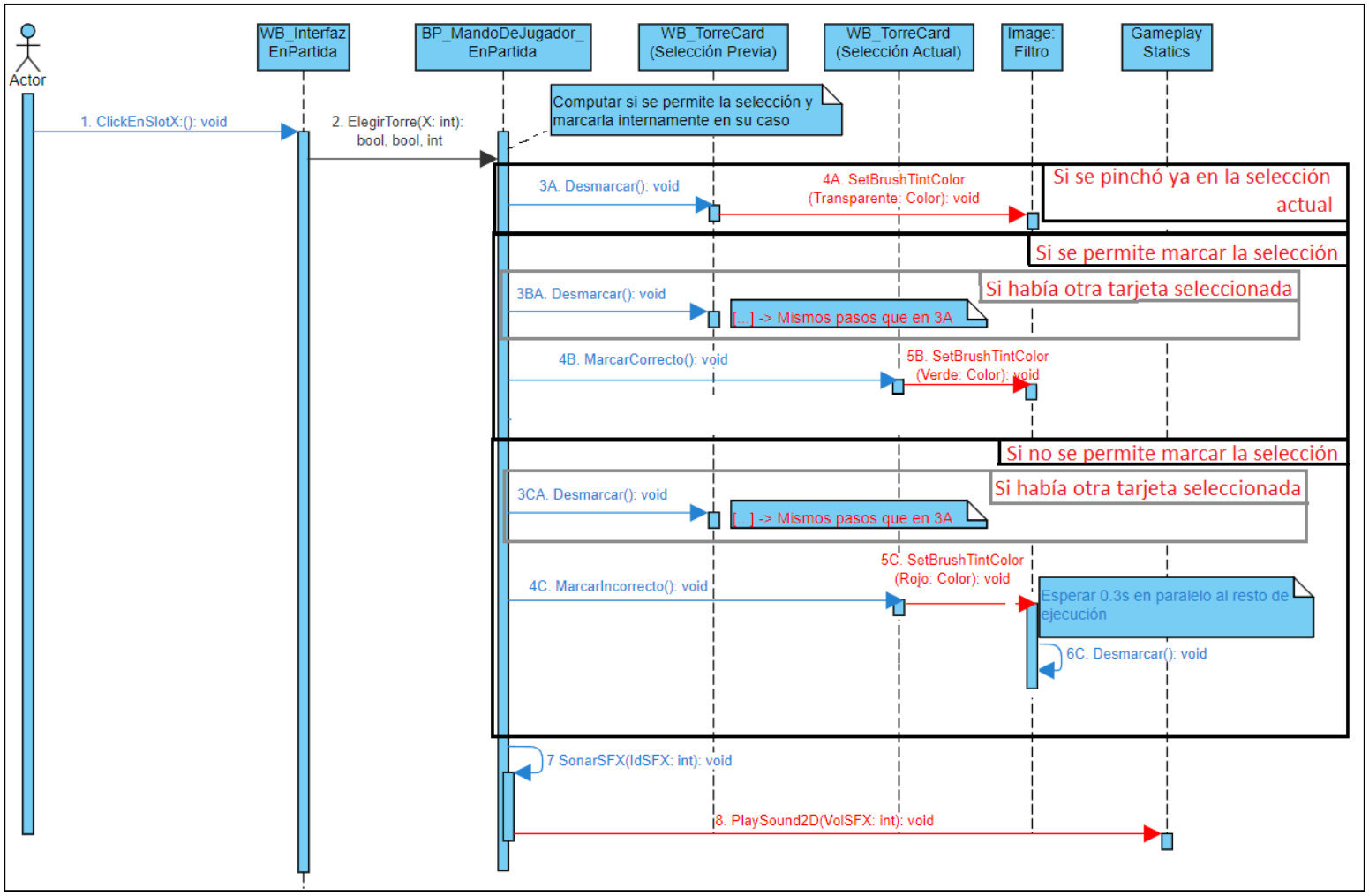


Figura 4.1.3: ClickEnSlotX(): void en WB_InterfazEnPartida

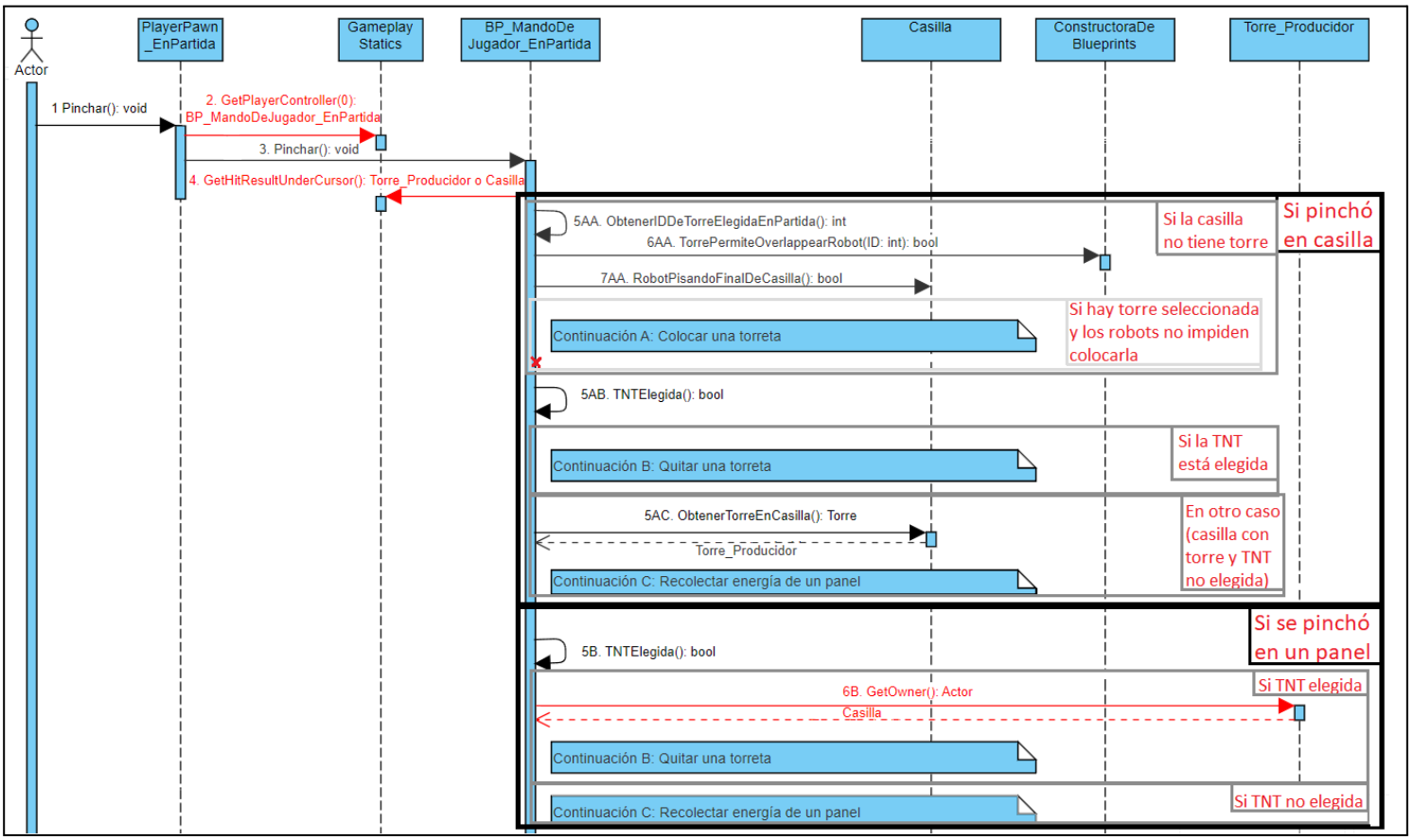


Figura 4.1.4: Pinchar(): void en PlayerPawn

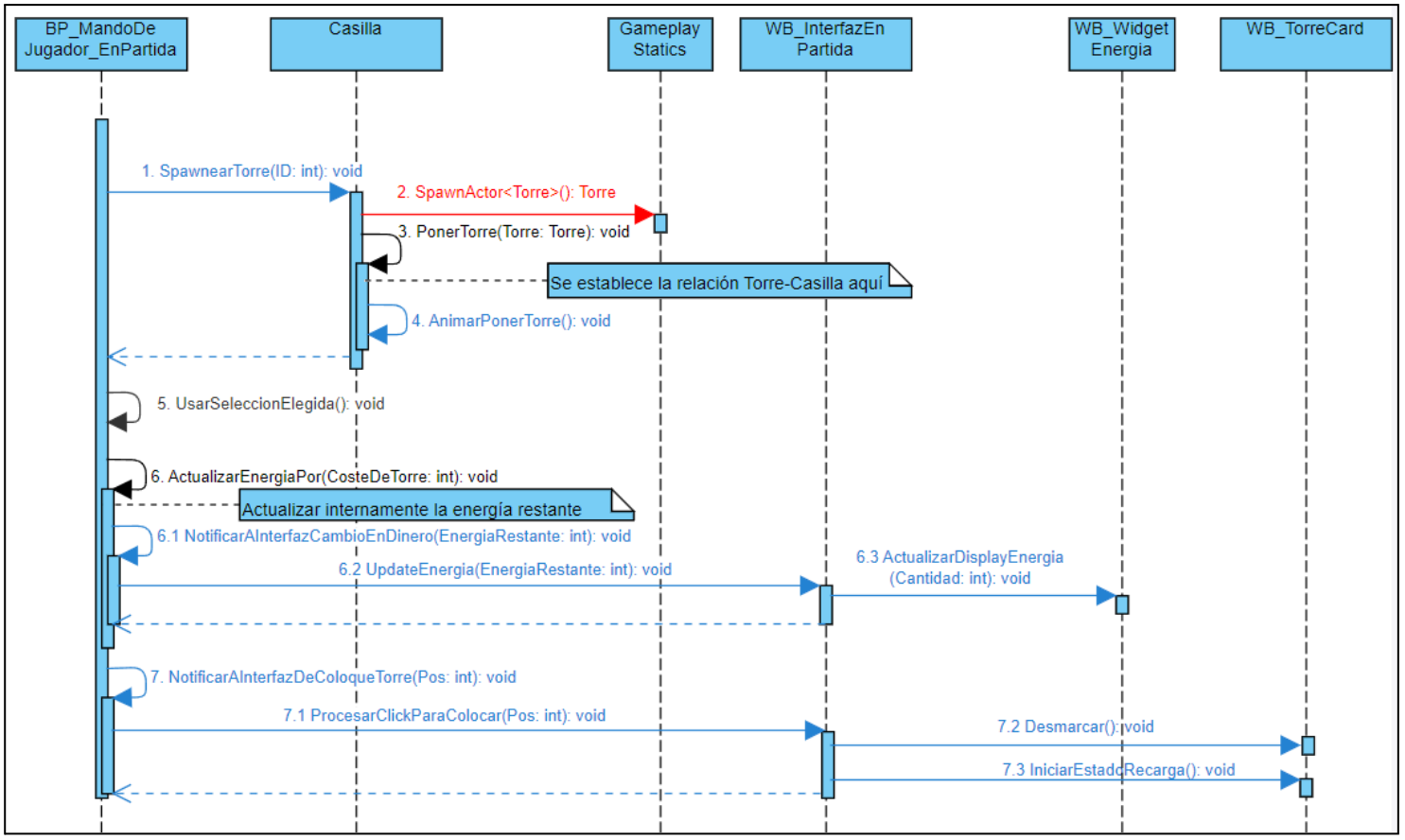


Figura 4.1.4.A: SpawnearTorre(): void en Casilla

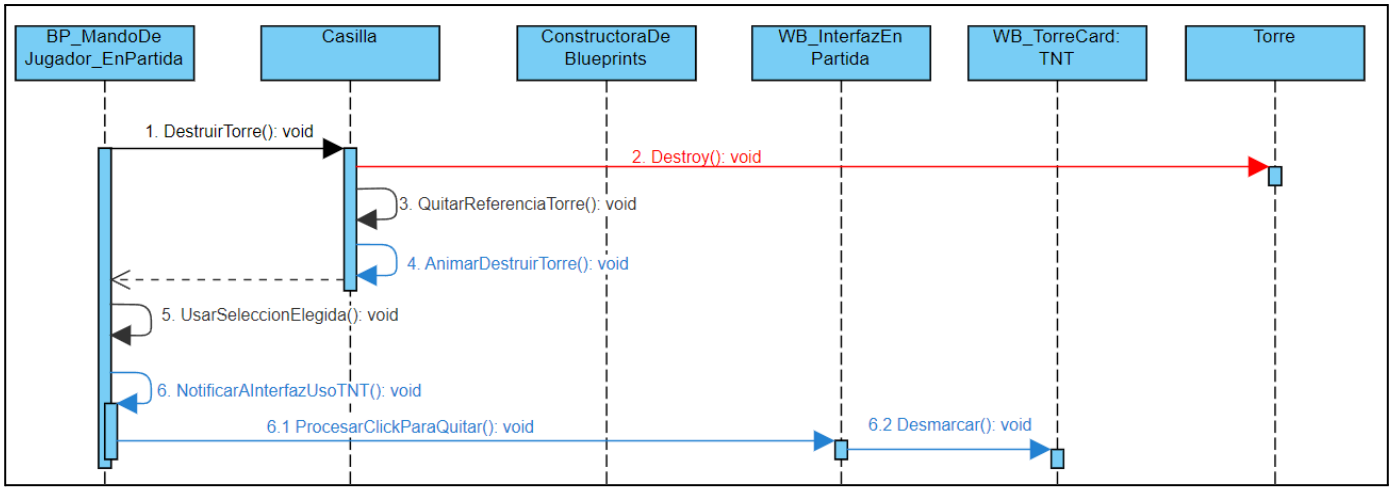


Figura 4.1.4.B: DestruirTorre(): void en MandoDeJugador_EnPartida

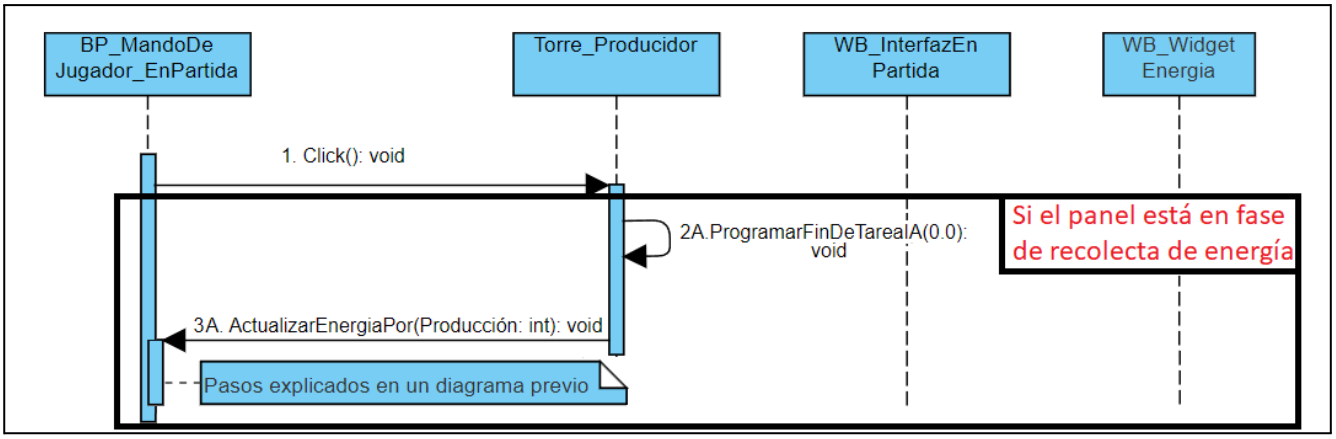


Figura 4.1.4.C: Click(): void en Torre_Producidor

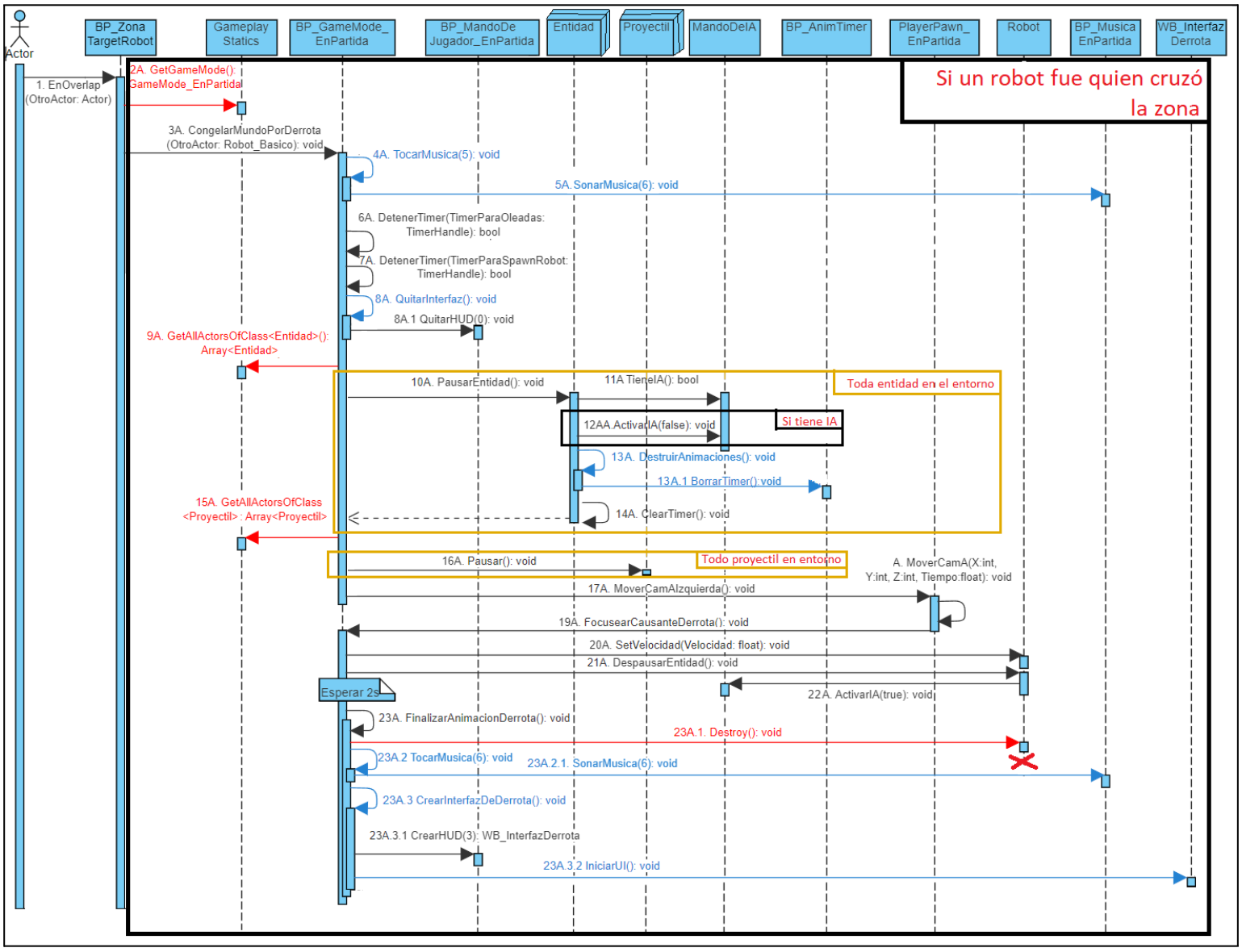


Figura 4.1.5: EnOverlap(OtroActor: Actor): void en BP_ZonaTargetRobot

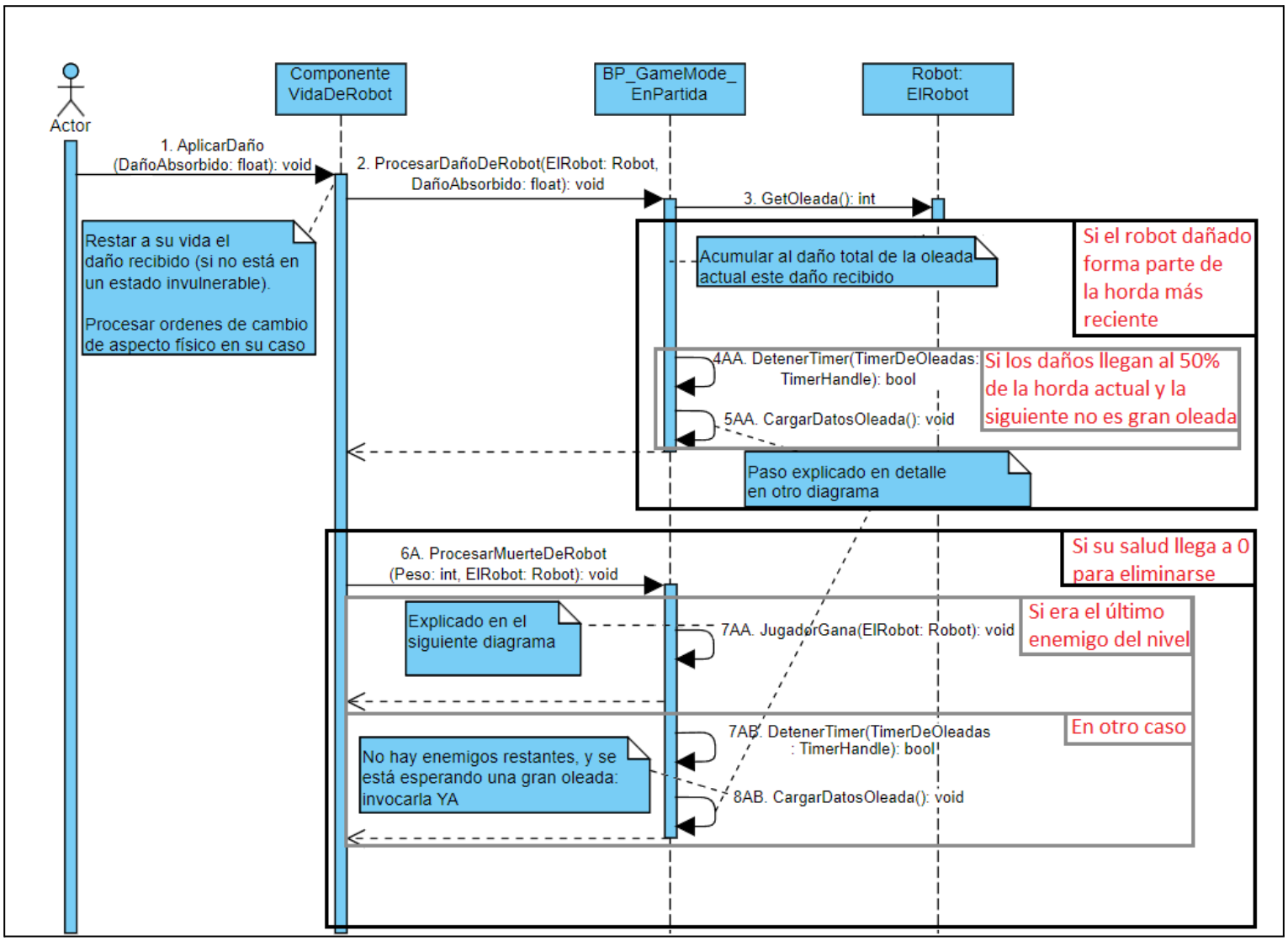


Figura 4.1.6: AplicarDaño(DañoAbsorbido: float): void en ComponenteVidaDeRobot

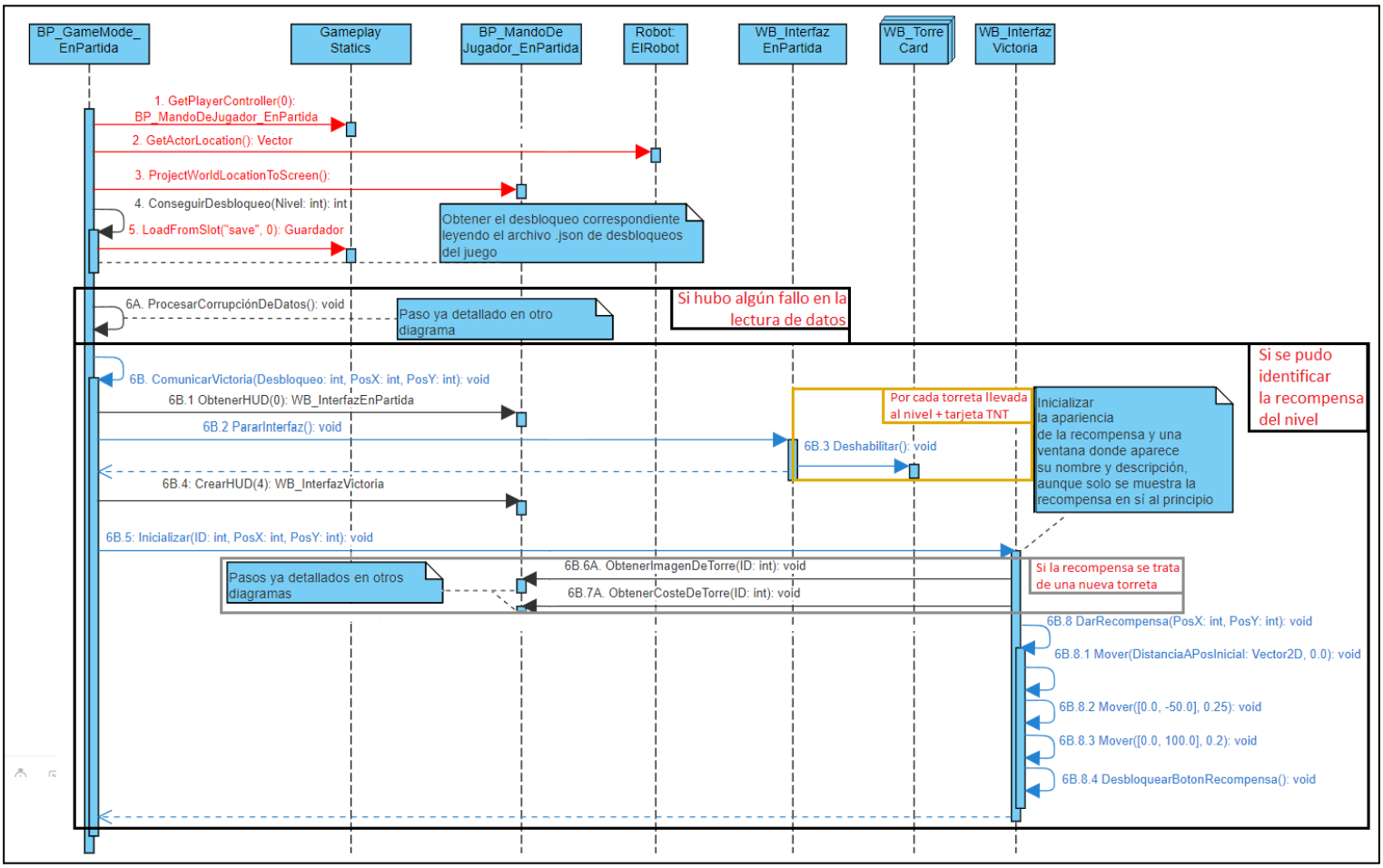


Figura 4.1.6.1: JugadorGana(ElRobot: Robot): void en GameMode_EnPartida

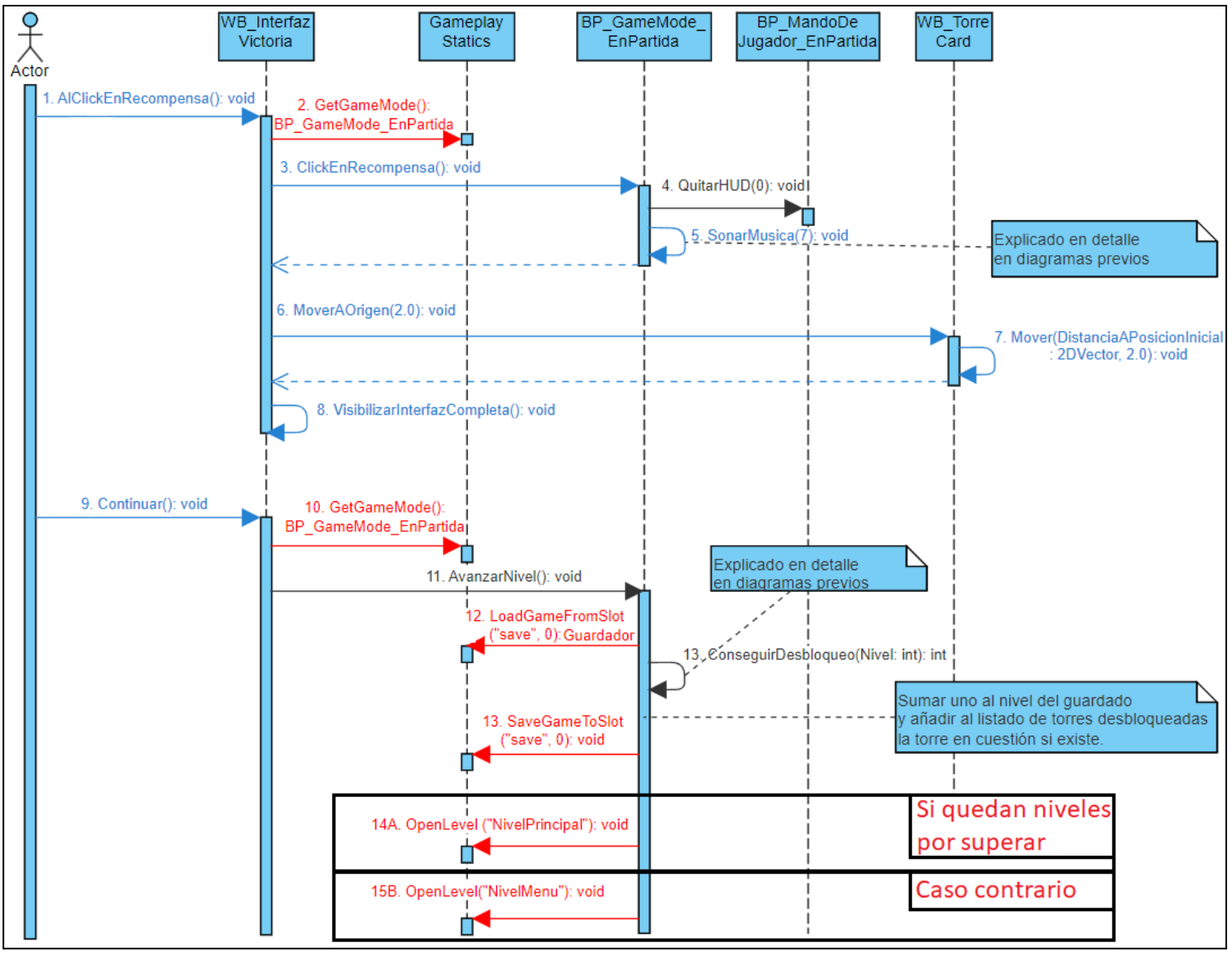


Figura 4.1.6.1.1: AlClickEnRecompensa(): void y Continuar(): void en WB_InterfazVictoria

Comparativa de editores (figuras 5.X)

Ejemplos de editores usados en el grado (figuras 5.1.X)

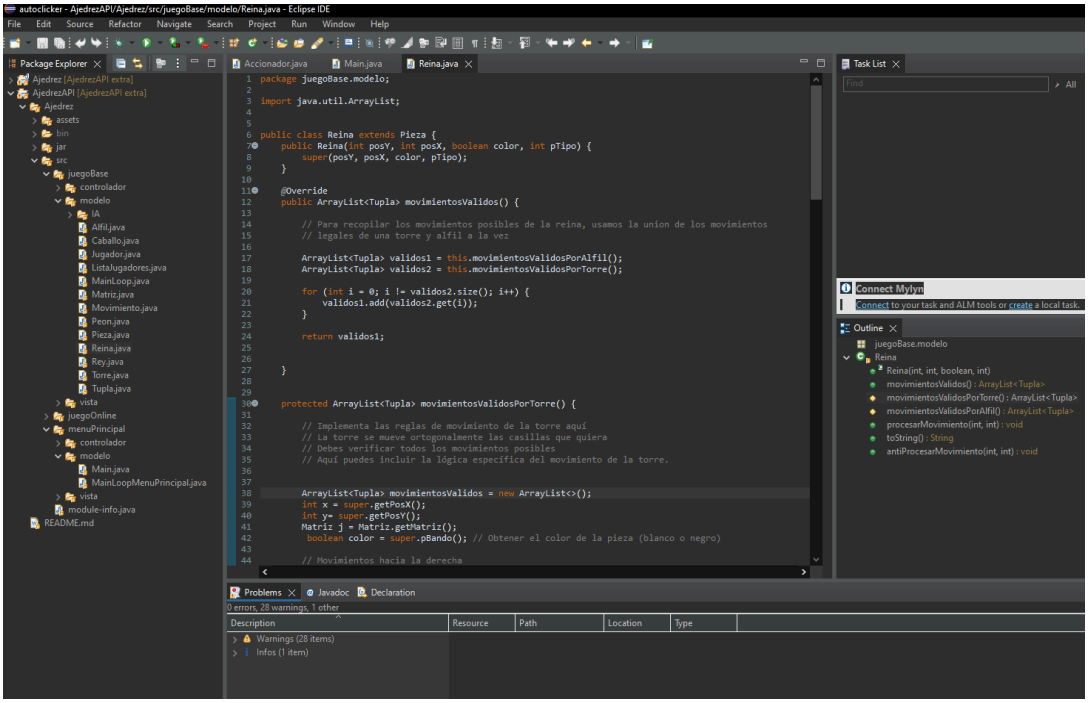


Figura 5.1.1: Interfaz de Eclipse IDE for Java Developers, un editor para crear programas Java. La figura muestra el proyecto de un simulador de ajedrez.

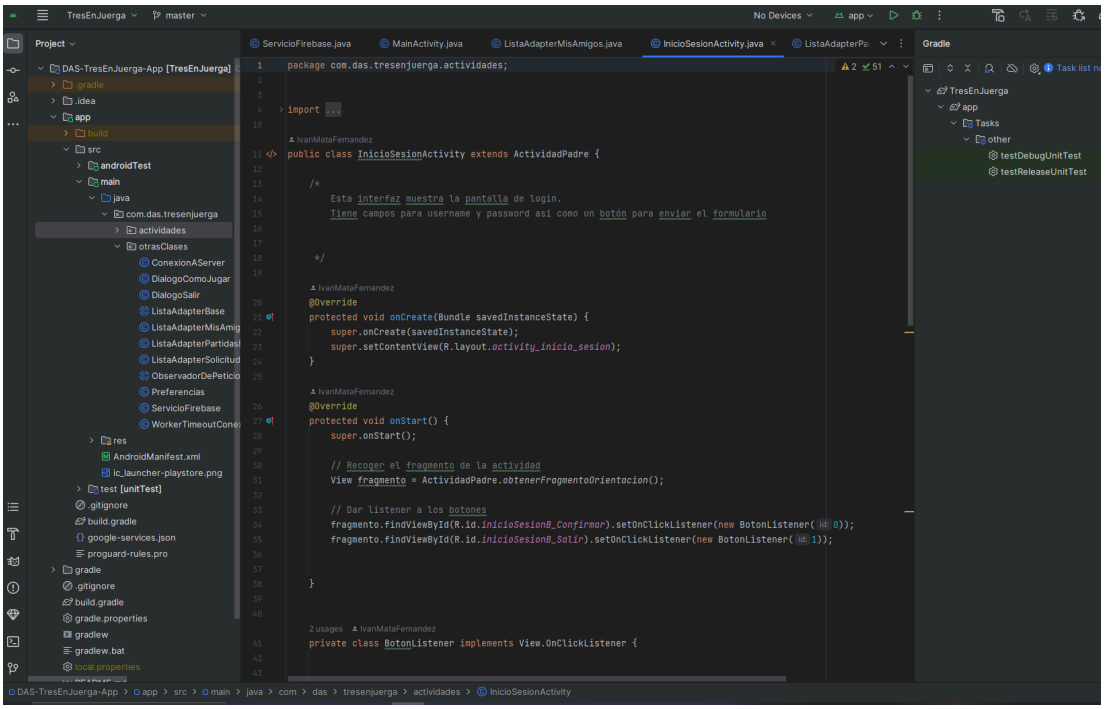


Figura 5.1.2: Interfaz de Android Studio, un editor para crear apps para móviles Android. La figura muestra el proyecto de una app para jugar al tres en raya en línea.

Ejemplos de editores nuevos usados en este proyecto (figuras 5.2.X)

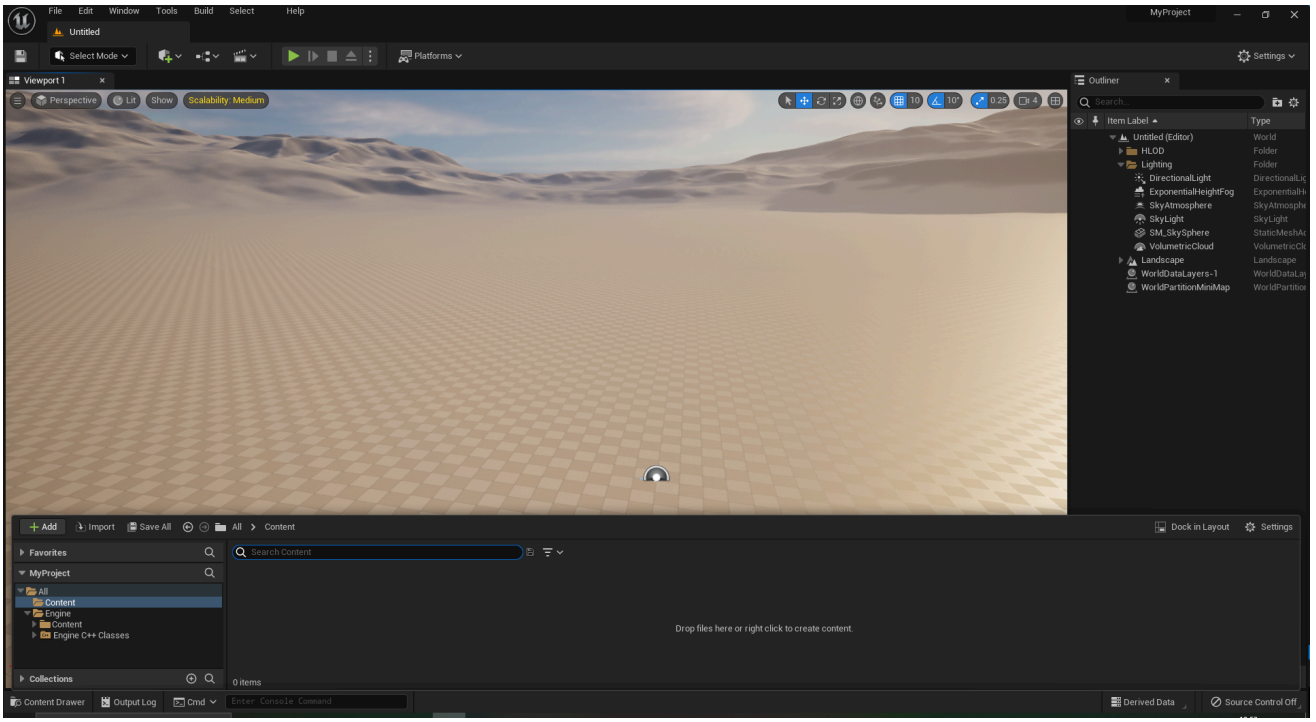


Figura 5.2.1: Interfaz de Unreal Engine 5

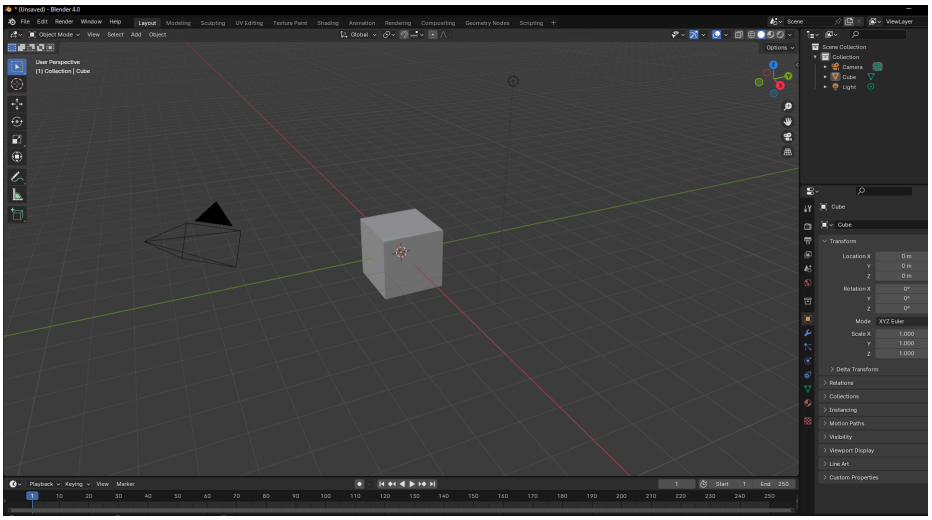


Figura 5.2.2: Interfaz de Blender

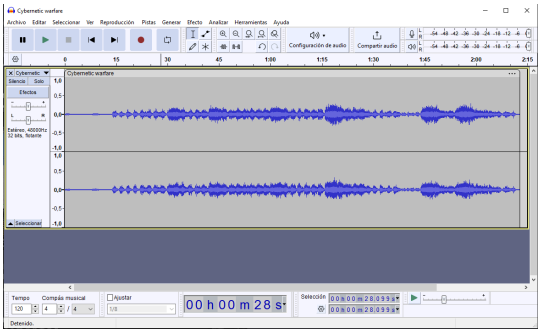


Figura 5.2.3: Interfaz de Audacity

Modos de programación en Unreal Engine (figuras 6.X)

```cpp
void AEntidad::Matar() {
    // Se llama a este método cuando vida = 0. Es la última llamada de la cadena del procesamiento de la eliminación de una en


    this->QuitarIA(); // Desactivar la IA si no se había hecho ya antes
    this->DesactivarHitbox(); // Quitar la hitbox de la entidad


    // Comprobar si la entidad tenia componente de vida y es vulnerable, si lo es, entonces animar su muerte
    // Si no cumple la condicion, la unidad se habria automatado y no tiene sentido animar esa muerte (ej, matarse tras haber

    UComponenteVida* ComponenteVida = FindComponentByClass<UComponenteVida>();

    if (ComponenteVida && ComponenteVida->EsVulnerable()) {
        this->RealizarAnimacion(0); // realizar animación de muerte (por blueprints)
        GetWorld()->GetTimerManager().SetTimer(TimerFrame, this, &AEntidad::Destruir,this->TiempoDeAnimacionDeMuerte, false);

    } else {
        this->AutoDestruir(); // Sirve para ocultar temporalmente la entidad fuera de la pantalla (para que suenen sus SFXs) h
                              // definitivamente
    }
}
```

Figura 6.1: Ejemplo de algoritmo desarrollado en C++ en el proyecto.

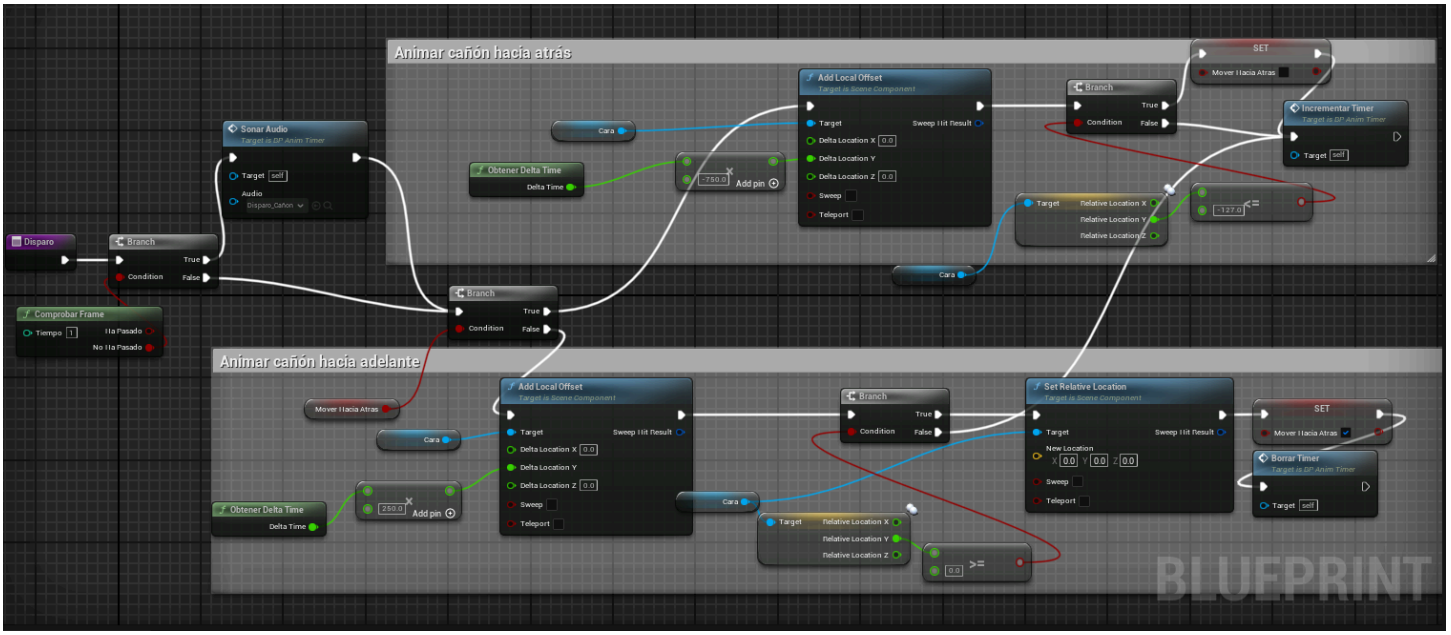


Figura 6.2: Ejemplo de algoritmo desarrollado en *Blueprints* en el proyecto