Facultad 4

PAQUETE DE MECÁNICAS PARA EL DESARROLLO DE VIDEOJUEGOS DEL TIPO TOWER DEFENSE SOBRE UNITY

Trabajo de diploma para optar por el título de
Ingeniero en Ciencias Informáticas

Autor: Yilber Alfonso Reynaldo

Tutores: Dr.C. Yuniesky Coca Bergolla
Ing. Oscar Mastrapa Prats

La Habana, 2023

DECLARACIÓN DE AUTORÍA

Declaro ser autor único de este trabajo y autorizo a los tutores Oscar Mastrapa Prats, Yuniesky Coca Bergolla y al Centro I+D+i Entornos Interactivos 3D (Vertex), de la Universidad de las Ciencias Informáticas, para que hagan el uso que estimen pertinente con este trabajo.

Para que así conste firmo la presente a los 29 días del mes de Noviembre del año 2023.

Yilber Alfonso Reynaldo

Autor

Dr.C Yuniesky Coca Bergolla

Tutor

Ing. Oscar Mastrapa Prats

Tutor

DATOS DE CONTACTO

Datos de los tutores:

Nombre y apellidos: Dr.C Yuniesky Coca Bergolla.

Síntesis del tutor:

Licenciado en ciencias de la Computación y Doctor en Ciencias de la Educación. Posee 20 años de experiencia en la docencia, específicamente en temáticas de inteligencia artificial y realidad virtual. Cuenta con varios artículos en revistas y trabajos presentados en eventos científicos. En los últimos años ha dedicado las investigaciones a las tecnologías educativas. Se desempeña como jefe del departamento de Inteligencia Computacional.

Correo electrónico: ycoca@uci.cu

Nombre y apellidos: Oscar Mastrapa Prats.

Síntesis del tutor:

Graduado de ingeniería en Ciencias Informáticas en el año 2022. Durante su trayectoria laboral participo en el comité organizador del evento Global Game Jam 2023 y el festival de desarrolladores de videojuegos 2022 y 2023. Actualmente se desempeña como especialista B y artista 3D en la línea de videojuegos del centro de Tecnologías Interactivas.

Correo electrónico: oscarmp@uci.cu

AGRADECIMIENTOS

A mi familia, por su amor incondicional, paciencia y comprensión durante todo este proceso. Gracias por ser mi mayor motivación y por creer en mí en cada paso del camino.

A mis compañeros de estudio y colegas, por su colaboración.

A mis tutores por su gran confianza y apoyo dentro y fuera del período de tesis.

A todos los profesores que contribuyeron en mi formación.

DEDICATORIA

A mis padres.

A mi familia.

RESUMEN

Los videojuegos del tipo Defensa de las Torres (*Tower Defense*) son populares en la actualidad y han experimentado una evolución constante desde sus humildes comienzos. En tiempos pasados, desarrollar este tipo de juegos requería un conocimiento profundo de programación a bajo nivel. No obstante, en la actualidad, gracias a los motores de videojuegos, los desarrolladores cuentan con un conjunto de funcionalidades reutilizables que les permiten centrarse en los elementos característicos del juego. En el marco de este proyecto, se creó un paquete que abarca las mecánicas principales que conforman a los videojuegos del tipo *Tower Defense*. Estas mecánicas se basaron en el estudio detallado de dos ejemplos pioneros en este subgénero. El paquete fue específicamente diseñado para el motor de videojuegos Unity y se siguió la metodología de desarrollo de software Extreme Programing XP. El objetivo primordial era brindar a los desarrolladores una base técnica reutilizable, flexible y escalable, que les permitiera ahorrar tiempo en la implementación de funcionalidades básicas y, así, focalizarse en enriquecer el juego con elementos distintivos. Para asegurar que la solución cumpliera con los requerimientos establecidos por el cliente, se realizaron pruebas de aceptación al finalizar cada iteración, así como pruebas unitarias. Además, se creó un demo que utilizaba el paquete desarrollado, demostrando su capacidad de reusabilidad, flexibilidad y escalabilidad. Por último, se llevó a cabo una prueba de rendimiento en dicho demo.

Palabras clave: mecánicas, paquete, Unity, videojuego.

ABSTRACT

Tower Defense video games are tremendously popular nowadays and have undergone a constant evolution since their humble beginnings. In the past, developing this type of game required a deep knowledge of low-level programming. Today, however, thanks to game engines, developers have a set of reusable functionalities that allow them to focus on the game's characteristic elements. Within the framework of this project, a package was created that encompasses the main mechanics that comfort Tower Defense type videogames. These mechanics were based on the detailed study of two pioneering examples of this subgenre. The package was specifically designed for the Unity game engine and followed the XP software development methodology. The primary objective was to provide developers with a reusable, flexible and scalable technical foundation that would allow them to save time in implementing basic functionality and thus focus on enriching the game with distinctive elements.

To ensure that the solution met the requirements established by the client, acceptance tests were performed at the end of each iteration, as well as unit tests to guarantee the correct functioning of the code. In addition, a demo was created using the developed package, demonstrating its reusability, flexibility and extensibility. Finally, a performance test was carried out on the demo, which confirmed the satisfactory achievement of the objective set out in the work.

Keywords: *mechanics, package, Unity, video game*.

ÍNDICE

INTRODUCCIÓN 1

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA 4

1.1 Videojuegos 4
1.1.1 Subgénero Tower Defense 6
1.1.2 Mecánicas de Videojuegos 7
1.1.3 Análisis de videojuegos similares 9
1.2 Proceso de desarrollo de videojuegos 13
1.2.1 Motor de videojuegos Unity 15
1.2.2 Sistemas Relacionados 17
1.3 Descripción del marco de trabajo 17
Conclusiones del capítulo 20

CAPÍTULO 2: ANÁLISIS Y DISEÑO DE LA SOLUCIÓN 21

2.1 Propuesta de solución 21
2.2 Requisitos no funcionales del sistema 22
2.3 Fase de exploración 23
2.3.1 Historias de Usuario 24
2.4 Fase de planificación 25
2.4.1 Plan de iteraciones 25
2.4.2 Plan de entregas 27
2.5 Fase de diseño 27
2.5.1 Descripción de la arquitectura 27
2.5.2 Patrones de diseño 29
2.5.3 Descripción de las tarjetas CRC. 31
2.5.4 Estándares de codificación 32
Conclusiones del capítulo 35

CAPÍTULO 3: VALIDACIÓN DE LA SOLUCIÓN PROPUESTA 36

3.1 Fase Implementación 36
3.1.1 Primera Iteración 36
3.1.2 Segunda Iteración 37
3.1.3 Tercera Iteración 38
3.1.4 Cuarta Iteración 38
3.2 Fase de Pruebas 39
3.2.1 Pruebas Unitarias 40
3.2.2 Pruebas de Aceptación 41
3.3 Análisis de Resultados 43
3.4 Pruebas de rendimiento 45
Conclusiones parciales 47

CONCLUSIONES FINALES 48
RECOMENDACIONES 49
REFERENCIAS BIBLIOGRÁFICAS 50
ANEXOS 54

ÍNDICE DE TABLAS

Tabla 1 Comparación de las mecánicas. 12
Tabla 2.HU Insertar puntos de ruta para el movimiento de enemigos. 24
Tabla 3. HU Insertar mecánica Comportamiento de las Torres 24
Tabla 4. Insertar mecánica Cámara. 25
Tabla 5. Distribución de iteraciones por historias de usuario 26
Tabla 6. Plan de entregas del producto 27
Tabla 7. Tarjeta CRC EnemyMovement 31
Tabla 8. Tarjeta CRC Camera 31
Tabla 9. Tarjeta CRC Shop 32
Tabla 10. Tarea 1 iteración 1 36
Tabla 11. Tarea 2 iteración 1 37
Tabla 12. Tarea 1 iteración 2 37
Tabla 13. Tarea 2 iteración 2 37
Tabla 14. Tarea 1 iteración 3 38
Tabla 15. Tarea 3 iteración 3 38
Tabla 16. Tarea 1 iteración 4 39
Tabla 17. Tarea 2 iteración 4 39
Tabla 18. Caso de Prueba de Aceptación P7HU7 41
Tabla 19.Caso de Prueba de Aceptación P9HU9 41
Tabla 20.Caso de Prueba de Aceptación P13HU13 41
Tabla 21. HU Insertar script de enemigos. 54
Tabla 22. HU Configurar script de enemigos. 54
Tabla 23. HU Insertar mecánica de locomoción. 55
Tabla 24. HU Modificar mecánica de locomoción. 55
Tabla 25. HU Insertar sistema de generación de oleadas. 55
Tabla 26. HU Modificar mecánica de Comportamiento de las Torres. 56
Tabla 27. HU Modificar mecánica de Cámara. 56
Tabla 28. HU Insertar mecánica Comportamiento del proyectil. 57
Tabla 29. HU Configurar mecánica Comportamiento del proyectil. 57
Tabla 30. HU Insertar mecánica de Vida General. 58
Tabla 31. HU Configurar mecánica de Vida General. 58
Tabla 32. HU Visualizar estadísticas. 59
Tabla 33. HU Establecer condición de pérdida del juego. 59
Tabla 34. HU Establecer lógica del juego 60
Tabla 35. HU Insertar mecánica de selección. 60
Tabla 36. HU Visualizar estructuras seleccionadas. 61
Tabla 37. HU Insertar mecánica Tienda. 61
Tabla 38. HU Modificar mecánica Tienda. 62
Tabla 39. Tarea 3 Iteración 1 62
Tabla 40. Tarea 4 Iteración 1 62
Tabla 41. Tarea 5 Iteración 1 63
Tabla 42. Tarea 2 Iteración 3 63
Tabla 43. Tarea 4 Iteración 3 63
Tabla 44. Tarea 5 Iteración 3 63
Tabla 45. Tarea 3 Iteración 4 64
Tabla 46. Tarea 4 Iteración 4 64
Tabla 47. Caso de Prueba de Aceptación P2HU2 64

Tabla 48. Caso de Prueba de Aceptación P3HU3. 65
Tabla 49. Caso de Prueba de Aceptación P4HU4. 65
Tabla 50. Caso de Prueba de Aceptación P5HU5. 66
Tabla 51. Caso de Prueba de Aceptación P6HU6. 66
Tabla 52. Caso de Prueba de Aceptación P8HU8. 66
Tabla 53. Tabla 52. Caso de Prueba de Aceptación P10HU10. 67
Tabla 54. Caso de Prueba de Aceptación P11HU11. 67
Tabla 55. Caso de Prueba de Aceptación P12HU12. 68
Tabla 56. Tabla 55. Caso de Prueba de Aceptación P14HU14. 68
Tabla 57. Caso de Prueba de Aceptación P15HU16. 69
Tabla 58. Caso de Prueba de Aceptación P16HU15. 69
Tabla 59. Caso de Prueba de Aceptación P17HU17. 70
Tabla 60. Caso de Prueba de Aceptación P12HU12 70
Tabla 61. Caso de Prueba de Aceptación P19HU19 71
Tabla 62. Caso de Prueba de Aceptación P20HU20 71
Tabla 63. Caso de Prueba de Aceptación P21HU21. 72
Tabla 64. Tarjeta CRC Bullet 77
Tabla 65. Tarjeta CRC Lives 77
Tabla 66. Tarjeta CRC Enemy..... 77
Tabla 67. Tarjeta CRC BuildManager..... 77
Tabla 68. Tarjeta CRC GameManager..... 78
Tabla 69. Tarjeta CRC GameOver 78
Tabla 70. Tarjeta CRC MoneyUI 78
Tabla 71. Tarjeta CRC Node 78
Tabla 72. Tarjeta CRC NodeUI..... 79
Tabla 73. Tarjeta CRC PlayerStats..... 79
Tabla 74. Tarjeta CRC Torres 79
Tabla 75. Tarjeta CRC TurretBlueprint 79
Tabla 76. Tarjeta CRC Wave..... 80
Tabla 77. Tarjeta CRC WaveSpawner 80
Tabla 78. Tarjeta CRC WayPoints 80
Tabla 79. Tarjeta CRC PauseMenu..... 80

ÍNDICE DE FIGURAS

Figura 1. Videojuego Kingdom Rush Fronties(11)**.** 10
Figura 2.Captura de pantalla al videojuego Plantas vs Zombies...... 11
Figura 3. Etapas en la producción de videojuegos. Tomado de (14) 14
Figura 4. Modelo de calidad de un producto. Tomado de(29) 23
Figura 5. Ejemplo de Alta cohesión entre las clases. Elaboración propia...... 30
Figura 6. Patrones GOF Creacionales. Elaboración propia. 31
Figura 7. Diagrama de clases de la solución. Elaboración propia. 32
Figura 8. Variables. Elaboración propia. 33
Figura 9. Método Die. Elaboración propia...... 34
Figura 10. Método *TakeDamage*. Elaboración propia...... 34
Figura 11. Clase PlayerStats. Elaboración propia. 34
Figura 12. Caso de prueba unitaria SpawnTest 40
Figura 13. Resultado de las pruebas por iteraciones...... 43
Figura 14. Resultados de las pruebas de rendimiento sin la interacción del usuario. Elaboración propia. 46
Figura 15. Resultados de las pruebas de rendimiento con la interacción del usuario. Elaboración propia. 47
Figura 16. Caso de prueba unitaria ShootTest. 73
Figura 17. Caso de prueba unitaria BuildTowerTest 73
Figura 18. Caso de prueba unitaria MovementTest 74
Figura 19. Prueba de Rendimiento sin interacción del usuario y sin elementos gráficos...... 75
Figura 20. Prueba de Rendimiento con interacción del usuario y sin elementos gráficos. 75
Figura 21. Prueba de Rendimiento sin interacción del usuario y con elementos gráficos. 76
Figura 22. Prueba de Rendimiento con interacción del usuario y con elementos gráficos. 76
Figura 23.Imagen del juego Kingdom Rush Frontiers(11) 81

INTRODUCCIÓN

Los videojuegos pertenecen a la categoría de los juegos, específicamente a la categoría de los juegos electrónicos. Estos tienen la peculiaridad de estar construidos sobre sistemas informáticos, basados en la construcción de aplicaciones informáticas, multimedia e interactivas. Estas aplicaciones pretenden entretener y resultar de agrado para sus usuarios, a menudo consiguen hacer sentir como si se estuviera inmerso en verdaderos mundos virtuales, como si se fuera partícipe de increíbles sucesos e imaginativas aventuras, permitiendo interactuar con el entorno del videojuego(1).

Lo más característico del videojuego es su interactividad, un flujo de acciones y reacciones mediante las que el juego se comunica con sus jugadores. Esta interactividad, bien utilizada, puede transmitir ideas, provocar emociones y hacer todo lo que uno esperaría de otro tipo de manifestaciones artísticas. Desde su aparición a mediados del siglo XX, se tenía una expectativa de su impacto en el mundo. Sin embargo, nunca se imaginó que alcanzaría el nivel de auge y popularidad que ha logrado gracias a su constante evolución en términos de jugabilidad, modos de control y visualización.

Los primeros videojuegos solo mostraban en pantalla puntos o figuras geométricas. Sin embargo ha sido tanto su crecimiento que en la actualidad ya logran mostrar gráficos en 3D de alto realismo y diversos modos de control, así como una gran variedad de modos y de agrupaciones (2).

El Centro de Tecnologías Interactivas (VERTEX) en la Universidad de Ciencias Informáticas en Cuba juega un papel importante en el desarrollo de la industria de los videojuegos en el país, en dicho centro se desarrollan juegos de diferentes géneros y, aunque no se han desarrollado aun videojuegos del tipo defensa de las torres, más conocidos por su nombre en inglés *Tower Defense,* no se encuentran ajenos a la posición que ocupan en el *ranking* mundial estos tipos de juegos*.* Esto demuestra que los desarrolladores de videojuegos cubanos están interesados en crear juegos en línea con las tendencias actuales y las preferencias de los usuarios en todo el mundo. Sin embargo, el desarrollo de juegos de tipo *Tower Defense* puede ser un proceso complejo que requiere tiempo y esfuerzo para analizar los diferentes factores que influyen en su funcionamiento, esto puede ser especialmente difícil en un entorno de desarrollo donde los recursos pueden ser limitados y los plazos pueden ser ajustados.

La falta de reutilización de código puede ser una limitación para los desarrolladores. La reutilización de código es una práctica común entre los programadores modernos, que les permite ahorrar tiempo y esfuerzo al crear nuevos juegos utilizando código ya existente. Por lo tanto, la falta de esta práctica puede hacer que el proceso de desarrollo sea más lento y menos eficiente, por otra parte, se aprecia que varios de los desarrolladores actuales del centro son recién graduados o trabajadores que aún no

ha acumulado la experiencia necesaria en el proceso de desarrollo, lo cual dificulta bastante esta etapa debido a que no cuentan con las habilidades necesarias para enfrentarse a dichos videojuegos en el tiempo establecido.

Por lo antes expuesto se tiene como **problema de la investigación:** ¿Cómo contribuir al desarrollo de videojuegos de tipo *Tower Defense* sobre la base de componentes reutilizables en Unity? Se define como **objeto de estudio** el desarrollo de videojuegos y se establece como **campo de acción** las mecánicas de videojuegos de tipo *Tower Defense*.

Para dar solución a la problemática existente se propone como **objetivo general:** desarrollar un paquete de mecánicas reutilizables y flexibles para videojuegos del tipo *Tower Defense* sobre Unity.

Para darle cumplimiento al objetivo del trabajo de definieron varias tareas de la investigación, las mismas son las siguientes:

Tareas de investigación:

- Elaboración del marco teórico de la investigación, a partir del estado del arte de los videojuegos.
- Caracterización de los paquetes de Unity destinados al desarrollo de mecánicas para videojuegos de tipo *Tower Defense*.
- Desarrollo de los flujos de trabajo que propone la ingeniería de *software* para obtener la solución.
- Desarrollo de pruebas para validar el cumplimiento de los requerimientos definidos.

Métodos científicos, técnicas e instrumentos para la recogida de datos:

Para la realización de la investigación se utilizarán varios métodos científicos de investigación, entre los cuales se pueden mencionar:

Métodos Teóricos:

Histórico – Lógico: Se empleó para la fundamentación y sistematización de los aspectos teóricos contemplados en el desarrollo de la investigación acerca de la evolución y las tendencias actuales del desarrollo de mecánicas de videojuegos.

Analítico – Sintético: Se utilizó para analizar los videojuegos de tipo Tower Defense existentes y arribar a conclusiones sobre sus características y mecánicas esenciales.

Modelación: Se utilizó en el análisis y diseño del sistema, en diagramas que permitieron especificar la propuesta de solución.

Métodos empíricos:

Observación: Se empleó como método referencial al observar el comportamiento de distintos video-juegos del tipo Tower Defense, para establecer una comparación y determinar las características y mecánicas comunes que poseen.

El presente trabajo está compuesto por tres capítulos, estructurados de la siguiente forma:

- **Capítulo 1 Fundamentación teórica:** En este capítulo se definen los principales conceptos necesarios para el desarrollo de la investigación como lo son los videojuegos casuales. Se realiza un análisis de diferentes videojuegos para este fin, así como de las herramientas, tecnologías y métodos a usar.
- **Capítulo 2 Análisis y diseño de la solución:** En este capítulo se definen los requisitos no funcionales y funcionales de la solución. También se describe la propuesta de solución, así como la arquitectura, la estructura de clases y los patrones empleados para el desarrollo de la misma.
- **Capítulo 3 Implementación y prueba:** Describe cómo se llevó a cabo el proceso de implementación del sistema y las pruebas realizadas para asegurar el perfecto funcionamiento del código y para verificar si lo realizado cumplía con las especificaciones definidas por el cliente. Se creó un demo para verificar la flexibilidad, reusabilidad y escalabilidad de la solución.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

En el presente capítulo se realiza una descripción de los conceptos fundamentales relacionados con los videojuegos, con el objetivo de establecer una vía para avanzar hacia el resto del trabajo. Se profundiza sobre el estudio de los videojuegos Tower Defense para agrupar sus comportamientos principales en las mecánicas que se definen, haciendo énfasis en las tecnologías, herramientas y metodología a usar en el desarrollo del trabajo.

1.1 Videojuegos

Un videojuego es un software creado con el propósito de brindar entretenimiento, diversión e inmersión en un entorno virtual. Su base fundamental radica en la interacción entre una o varias personas y un dispositivo electrónico, como un computador, una videoconsola o un teléfono móvil. Estos dispositivos son comúnmente conocidos como plataformas de juego.

En esencia, un videojuego es un software diseñado para apoyar el entretenimiento, utilizando la interacción entre entornos virtuales, dispositivos electrónicos y usuarios para simular una apariencia de la realidad deseada y sumergir al menos a un jugador en ella. El concepto de juego se refiere a una actividad que es esencialmente libre y voluntaria, que se desarrolla en un tiempo y espacio separados, y que es incierta e improductiva. Está regida por las reglas de la fantasía y puede abarcar diversas formas de software de entretenimiento por computadora, haciendo uso de cualquier plataforma electrónica disponible. Además, los juegos pueden implicar la participación de uno o varios jugadores, ya sea en un entorno físico o en una red en línea(3).

La categorización de las aplicaciones informáticas se enfrenta a un desafío en términos de exhaustividad y precisión al definir las diversas naturalezas en el contenido y en la forma. Este problema se ve acentuado en el caso de los videojuegos debido a su enorme versatilidad, su gran variedad temática y formal, así como la mezcla de propuestas y géneros que presentan.

Se hace complejo clasificar algunos juegos en una categoría exclusiva, debido a que la mayoría de ellos son una mezcla entre aventura, estrategia, lucha y competición. La mezcla de elementos y mecánicas de juego es una característica distintiva de los videojuegos contemporáneos, trascendiendo las limitaciones tradicionales de género. Los videojuegos actuales fusionan y entrelazan múltiples elementos y mecánicas de juego. Por ejemplo, es común encontrar juegos de aventura que incorporan elementos estratégicos, permitiendo a los jugadores tomar decisiones tácticas mientras exploran un vasto mundo virtual. Asimismo, los juegos de lucha pueden combinar elementos de competición y

narrativa, brindando una experiencia más inmersiva y profunda. Esta diversidad y mezcla de géneros en los videojuegos refleja la constante evolución de la industria y la creatividad de los desarrolladores. Los principales géneros que se pueden encontrar son:

Arcade: Llamados así porque mantienen los principios de los antiguos juegos de Arcade, basados en el entretenimiento inmediato y una jugabilidad sencilla y adictiva. Su presencia está muy extendida en los sistemas móviles. Estos videojuegos presentan controles muy simples, niveles de corta duración y los objetivos son fáciles de comprender al instante. Se caracterizan por tener una gran cantidad de niveles cortos en donde la dificultad va aumentando rápidamente al progresar(4), dentro de estos se pueden encontrar los videojuegos de laberintos, velocidad, camino de obstáculos, entre otros.

Plataformas: Los videojuegos de plataformas, también conocidos como «plataformas», es un género de juegos que consisten en seguir caminos llenos de plataformas por las que se debe ir saltando mientras se esquivan enemigos y acantilados. Se suelen utilizar vistas de derecha a izquierda en las que se van mostrando otras partes de la pantalla según se avanza (5).

Disparos: Existen dos tipos de videojuegos de disparos, los que son en primera persona y los que son en tercera persona.

- **Videojuegos de disparos en primera persona:** En los videojuegos de disparos en primera persona, conocidos también como ***FPS*** (*first person shooter*), se maneja al protagonista desde una perspectiva subjetiva, es decir, se observa en la pantalla lo que puede observar nuestro personaje, y por tanto no se observa a él. Sí se puede observar en cambio su arma, en primer plano, la cual se debe usar para abatir a los diferentes enemigos que aparecerán al frente. La perspectiva en primera persona, en un entorno 3D, tiene por meta dar al jugador la impresión de ser el personaje, buscando con ello una experiencia más realista de juego(6).
- **Videojuegos de disparos en tercera persona:** Los videojuegos de disparos en tercera persona, conocidos también como *TPS* (*third person shooter*), se basan en la alternancia entre disparos y pelea o interacción con el entorno, pero a diferencia de los juegos de mira (primera persona), se juega con un personaje visto desde atrás o, en ocasiones, desde una perspectiva isométrica(6).

Estrategia: En este tipo de juegos el jugador asume una identidad específica y se familiariza con un objetivo final al que debe dirigir todas sus acciones. En el proceso, se desarrollan estrategias tácticas de diversos tipos para lograr un desenlace exitoso y fructífero.

Los juegos de estrategia son de los más reconocidos y motivantes para el jugador. Esta investigación se centra en este tipo de videojuegos, por lo que se profundizará en ellos, específicamente en los de tipo Tower Defender.

1.1.1 Subgénero Tower Defense

En los juegos de estrategia la acción se lleva a cabo mediante el uso de una serie de comandos verbales reconocidos por el programa, así como la obtención y posesión de objetos y elementos que aparecen en escenarios cambiantes. Estos elementos resultan indispensables para avanzar de nivel, alcanzar la victoria y el éxito final. Los juegos de estrategia se centran más en la duración y la reflexión que en la rapidez. Es decir, su interés radica en el tema propuesto, el desarrollo argumental, las múltiples opciones ofrecidas y la complejidad de las soluciones posibles. Sin embargo, esto no significa que se descuide el aspecto gráfico del juego o la calidad de los efectos de sonido.

En este tipo de juegos, se enfatiza la toma de decisiones estratégicas, la planificación cuidadosa y la consideración de las consecuencias a largo plazo. Los jugadores deben evaluar diferentes escenarios, anticipar movimientos y gestionar recursos para alcanzar sus objetivos. A menudo, se requiere pensar de manera estratégica y encontrar soluciones creativas para superar obstáculos y derrotar a los oponentes (7).

Un Tower Defense es un subgénero del videojuego de estrategia en el que el jugador se enfrenta al desafío de proteger sus territorios o posesiones de las hordas de atacantes enemigos. La premisa básica consiste en obstaculizar y frenar el avance de los enemigos mediante la cuidadosa colocación de diversas estructuras defensivas a lo largo de las rutas de ataque.

Aunque el nombre del género hace referencia a las torres, las cuales son una elección popular para la defensa, la creatividad y la diversidad de los elementos defensivos van mucho más allá. Los jugadores pueden aprovechar una amplia gama de opciones estratégicas, que van desde personajes especializados y estructuras fortificadas hasta elementos naturales como minerales y vegetación.

La construcción de una defensa sólida implica tomar decisiones tácticas inteligentes. Los jugadores deben considerar cuidadosamente la colocación estratégica de sus defensas para maximizar la eficacia y crear una barrera impenetrable para los enemigos. Algunas estructuras pueden bloquear el paso, ralentizar el avance enemigo o incluso infligir daño directo a los atacantes. Otras pueden proporcionar apoyo a las defensas existentes, como reparar estructuras dañadas o debilitar a los enemigos.

A medida que el juego avanza, los desafíos se vuelven más difíciles y los enemigos más poderosos. Los jugadores deben adaptar constantemente su estrategia, mejorando y actualizando sus defensas para mantenerse un paso por delante de sus oponentes. La gestión de recursos y la toma de decisiones rápidas son fundamentales para el éxito en este género.

Además de la emoción estratégica, los juegos de defensa de torres ofrecen una experiencia visualmente atractiva. Los entornos suelen ser detallados y coloridos, con diseños de personajes y anima-

ciones cautivadoras. Esto sumerge a los jugadores en un mundo virtual lleno de acción y tensión mientras protegen sus tierras de la invasión enemiga(8).

1.1.2 Mecánicas de Videojuegos

Las mecánicas son la esencia de la interacción en el desarrollo de videojuegos. Las combinaciones de las mismas configuran la jugabilidad y el estilo del desempeño que tendrán que adoptar los jugadores en estos (9).

Las mecánicas son un aspecto esencial en un videojuego, convierten al videojuego en una aplicación interactiva. Los paquetes de mecánicas de los videojuegos son conjuntos predefinidos de funciones y herramientas que permiten a los desarrolladores de videojuegos implementar mecánicas de juego complejas de manera más rápida y eficiente. Estos paquetes suelen incluir componentes como sistemas de física, inteligencia artificial, animación, sonido, iluminación, entre otros, que son necesarios para crear un juego(10).

Los paquetes de mecánicas de los videojuegos permiten a los desarrolladores centrarse en la creación de contenidos y de la experiencia de juego, en lugar de dedicar tiempo y recursos a la creación de sistemas complejos desde cero. Además, estos paquetes suelen ser flexibles, escalables y reutilizables, lo que significa que los desarrolladores pueden ajustar y adaptar las mecánicas de juego para adecuarse a las necesidades específicas de su proyecto.

Las mecánicas de los videojuegos juegan un papel fundamental en la comprensión de las diversas formas en que los jugadores interactúan y se sumergen en el mundo virtual. Estas mecánicas, que actúan como los cimientos de la jugabilidad, abarcan una amplia gama de elementos clave que definen la experiencia del jugador. Desde el movimiento y el combate hasta los puzles y la progresión, cada mecánica desempeña un papel único en la forma en que los jugadores interactúan con el juego y experimentan su narrativa. En este epígrafe, se explorarán y clasificarán algunas de las mecánicas fundamentales presentes en los videojuegos, destacando su importancia y cómo contribuyen a la creación de experiencias de juego inmersivas y emocionantes. Entre la amplia lista que las comprenden se pueden encontrar:

- **Movimiento:** El movimiento es una mecánica esencial en la mayoría de los videojuegos. Permite a los jugadores desplazarse por el entorno del juego, interactuar con objetos y personajes, y explorar el mundo virtual. Puede incluir correr, saltar, trepar, nadar o volar, dependiendo del juego y su temática.
- **Acción y combate:** Muchos videojuegos involucran algún tipo de acción y combate, donde los jugadores enfrentan desafíos y enemigos. Esta mecánica implica realizar ataques, utilizar

habilidades especiales, esquivar o bloquear ataques enemigos y tomar decisiones estratégicas durante los enfrentamientos. Los juegos de acción y los juegos de lucha son ejemplos claros de géneros que se centran en esta mecánica.

- **Resolución de puzles:** Los puzles son una mecánica común en muchos videojuegos, especialmente en los juegos de aventura o de rompecabezas. Requieren que los jugadores resuelvan problemas lógicos, encuentren patrones, recolecten objetos y utilicen su ingenio para avanzar en el juego. Los puzles pueden variar en complejidad y pueden ser cruciales para avanzar en la historia o desbloquear nuevas áreas.
- **Progresión y logros:** La progresión es una mecánica que se encuentra en muchos videojuegos, donde los jugadores avanzan y desarrollan su personaje, obtienen habilidades o desbloquean contenido a medida que superan desafíos o alcanzan objetivos. La progresión puede incluir ganar experiencia, subir de nivel, obtener mejoras de equipo o desbloquear nuevas áreas del juego. Los logros también son parte de esta mecánica, ofreciendo recompensas adicionales por completar tareas o alcanzar hitos específicos.
- **Interacción social y multijugador:** Muchos videojuegos permiten a los jugadores interactuar entre sí, ya sea cooperando o compitiendo. Esta mecánica social puede incluir el juego en línea con otros jugadores, el chat o la comunicación en tiempo real, el comercio de objetos virtuales o la participación en competiciones en línea

Estas mecánicas son generales para juegos de estrategia, sin embargo, es importante para la presente investigación profundizar en las específicas para los juegos de tipo Tower Defense.

Específicamente los videojuegos de tipo Tower Defense están conformados por una serie de mecánicas que en gran parte son las que se encargan de hacer este subgénero tan popular en el mundo. Entre las mecánicas generales que conforman este tipo de videojuego se encuentran las siguientes:

- **Cámara:** se refiere a la vista del juego, que puede ser en tercera persona o en vista aérea, y que permite a los jugadores ver el campo de batalla y planificar su estrategia.
- **Selección:** permite a los jugadores seleccionar y controlar diferentes unidades o elementos del juego, como torres, plantas o tropas.
- **Desplazamiento:** permite a los jugadores moverse por el mapa del juego y colocar torres, plantas o tropas en diferentes lugares estratégicos.
- **Vida general:** se refiere a la salud o vida de las unidades del juego, como las torres, plantas o tropas del jugador. Los jugadores deben proteger y mantener la vida de estas unidades para tener éxito en el juego.

- **Comportamiento de las torres:** se refiere al modo en que las torres o defensas del jugador disparan o atacan a los enemigos. Las torres pueden tener diferentes tipos de armamento, alcance, velocidad de disparo, etc.
- **Comportamiento del proyectil:** se refiere al movimiento de los proyectiles disparados por las torres o defensas del jugador, como balas, cohetes, misiles, etc.
- **Posicionamiento de tropas:** se refiere a la colocación de las tropas del jugador en diferentes lugares del campo de batalla para proteger la base o atacar a los enemigos.
- **Tienda:** se refiere al lugar donde los jugadores pueden comprar nuevas torres, plantas o tropas, o mejorar las existentes.
- **Habilidades especiales:** Algunos juegos de Tower Defense ofrecen habilidades especiales que los jugadores pueden utilizar para ayudar en la defensa. Estas habilidades pueden incluir ataques aéreos, trampas, ralentización de enemigos, congelación, entre otros. Estas habilidades a menudo tienen un tiempo de reutilización o un costo asociado, lo que requiere una gestión adecuada para utilizarlas en los momentos clave.
- **Recolección de recursos:** se refiere a la recolección de recursos, como dinero, soles, oro, entre otras, que se utilizan para comprar y mejorar torres, plantas o tropas.

Es importante realizar un análisis de las mecánicas más utilizadas en otros videojuegos de este tipo para decidir cuales incorporar en la presente investigación.

1.1.3 Análisis de videojuegos similares

Existe un gran número de videojuegos de este tipo, entre ellos se encuentran:

1. ***Plants vs. Zombies***: uno de los Tower Defense más populares y exitosos, en el que el jugador debe defender su jardín de zombis utilizando plantas con habilidades especiales.
2. ***Kingdom Rush*****:** un juego de estrategia en tiempo real que requiere que el jugador construya torres y defensas para proteger su reino de enemigos invasores.
3. ***Bloons TD*****:** una serie de juegos en los que el jugador debe defender su territorio de globos invasores, utilizando una variedad de torres y armas.
4. ***Defense Grid*****:** un juego de estrategia en el que el jugador debe proteger su base de alienígenas invasores utilizando torres y armas avanzadas.
5. ***Dungeon Defenders*****:** un juego que combina elementos de *Tower Defense* con *RPG*, en el que el jugador debe defender su mazmorra de invasores utilizando personajes con habilidades especiales.

Del análisis de estos videojuegos se seleccionaron dos de los más aceptados para realizar un estudio de las principales mecánicas que incorporan y decidir las que se incorporarán como solución a la presente investigación.

Los dos juegos seleccionados para el análisis fueron el Kingdom Rush y Plantas vs Zombies. A continuación se analizan las principales características, específicamente en las mecánicas que incorporan.

En la figura 1 se muestra una captura de pantalla del videojuego Kingdom Rush:

Figura 1. Videojuego Kingdom Rush Fronties*(11).*

Kingdom Rush es un juego de estrategia en tiempo real de *Tower Defense* desarrollado por *Ironhide Game Studio*. El juego fue lanzado por primera vez en julio de 2011 para dispositivos móviles iOS y más tarde fue lanzado para Androide y PC(13). La jugabilidad de "*Kingdom Rush*" se centra en construir torres y defensas para proteger el reino del jugador de los enemigos invasores. Los jugadores deben colocar torres en puntos estratégicos del mapa para detener a los enemigos antes de que lleguen a la base del jugador. A medida que el jugador avanza en el juego, se desbloquean nuevas torres y habilidades especiales para ayudar en la defensa.

El juego también cuenta con una variedad de enemigos con habilidades y fortalezas únicas, lo que requiere que el jugador adapte su estrategia y construya torres específicas para contrarrestar las habilidades de los enemigos. Además de su jugabilidad adictiva y desafiante, Kingdom Rush también es conocido por su estética de dibujos animados y su humor. Los personajes y enemigos son caricaturescos y están llenos de personalidad, lo que hace que el juego sea divertido y atractivo para una amplia audiencia.

En la figura 2 se muestra una captura de pantalla del videojuego Plantas vs Zombies:

Figura 2.Captura de pantalla al videojuego Plantas vs Zombies.

Plants vs. Zombies es un juego de estrategia de Tower Defense desarrollado por PopCap Games. El juego fue lanzado por primera vez en mayo de 2009 para dispositivos móviles iOS y PC, y desde entonces ha sido lanzado en una variedad de plataformas(15).

La jugabilidad de Plants vs. Zombies se centra en defender la casa del jugador de los zombis invasores utilizando plantas con habilidades especiales. El jugador debe estratégicamente colocar las plantas en el jardín delantero para detener a los zombis antes de que lleguen a la casa del jugador. A medida que el jugador avanza en el juego, se desbloquean nuevas plantas y habilidades especiales para ayudar en la defensa.

El juego también cuenta con una variedad de zombis con habilidades y fortalezas únicas, lo que requiere que el jugador adapte su estrategia y utilice diferentes plantas para contrarrestar las habilidades de los zombis. En cuanto a los controles, en Plants vs. Zombies el jugador selecciona y coloca plantas haciendo clic en ellas en el menú de selección y luego haciendo clic en el lugar deseado en el jardín. También se pueden utilizar habilidades especiales haciendo clic en los iconos correspondientes en la parte inferior de la pantalla.

En la siguiente tabla se describen las mecánicas recurrentes de estos videojuegos, tomando como guía las definidas en el epígrafe anterior. Se pondrá “Si” en caso que sí sean recurrentes, en caso contrario “No” y si varía algún mecanismo o incorporan otros diferentes serán descritos.

Tabla 1 Comparación de las mecánicas.

Mecánicas	Plantas vs Zombies	*Kingdom Rush*
Cámara	Si El enfoque de la cámara se puede manipular con la rueda del mouse o con un clic en la pantalla y arrastrando.	Si El enfoque de la cámara se puede manipular con la rueda del mouse o con un clic en la pantalla y arrastrando.
Selección	Si Se utiliza el mouse para seleccionar en el lugar que se desea ubicar la unidad, actualizarla o modificarla.	Si Se utiliza el mouse para seleccionar en el lugar que se desea ubicar la unidad, actualizarla o modificarla.
Locomoción	Si Los enemigos tienen una ruta predefinida.	Si Los enemigos tienen una ruta predefinida.
Vida general	Si	Si
Comportamiento de las torres	Si Las plantas pueden ser colocadas en cualquier lugar del jardín y tienen habilidades únicas, como disparar proyectiles, ralentizar a los zombis o explotar.	Si Las torres solo pueden ser colocadas en lugares específicos y tienen diferentes tipos de ataque, como ataques a distancia, ataques de área o ataques mágicos.
Comportamiento del proyectil	Si Los proyectiles disparados por las plantas viajan en línea recta hacia el objetivo y pueden ser bloqueados por otros zombis.	Si Los proyectiles de las torres tienen diferentes velocidades y trayectorias, y algunas torres pueden disparar proyectiles que rebotan entre múltiples enemigos.
Posicionamiento de Tropas	No	Si Las tropas se posicionan en diferentes partes del mapa para servir de apoyo a las torres.
Tienda	Si	Si
Recolección de recursos	Si Brinda la posibilidad de recolectar	no

	una serie de recursos adicionales para mejorar las torres y las estructuras.	

Como se puede observar no todas las mecánicas que comprenden el videojuego Kingdom Rush están presentes en el videojuego Plantas vs Zombies por lo que se realizó una selección de aquellas que son esenciales para todo tipo de Tower Defense y que se pueden observar en la tabla para ambos ejemplos, estas son:

1. Selección
2. Locomoción
3. Tienda
4. Comportamiento del proyectil
5. Vida general
6. Comportamiento de las torres
7. Cámara

1.2 Proceso de desarrollo de videojuegos

Desarrollar un videojuego es un proceso apasionante que requiere una planificación cuidadosa. A lo largo de la producción, se abordan distintos desafíos y se involucran profesionales especializados. Cada etapa del desarrollo del videojuego (Figura 3) presenta oportunidades únicas para dar vida a la visión creativa, las etapas se dividen de la siguiente manera:

- **Fase de Pre-Producción:** En esta etapa inicial, el objetivo principal es establecer los fundamentos del videojuego y proporcionar un concepto claro sobre cómo se llevará a cabo su desarrollo. Esto incluye la planificación y la organización del proyecto, considerando el equipo de trabajo, el presupuesto y el cronograma de ejecución(12).

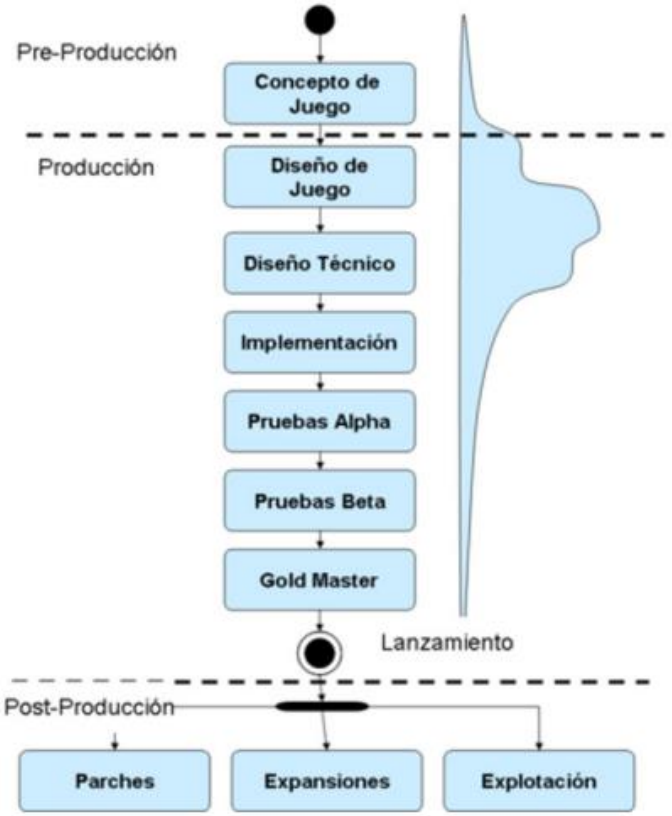

Figura 3. Etapas en la producción de videojuegos. Tomado de *(14)*

- **Fase de Producción:** En esta etapa se desarrolla la codificación o programación del videojuego que se ha defino en la etapa anterior, es una de las etapas más importantes y de mayor duración, pues en esta el equipo de desarrollo de videojuego realiza diferentes actividades con el fin de dejar un prototipo del juego(14).

- **Fase de Post-producción:** Esta etapa es fundamental en el proceso de desarrollo de un videojuego, pues va más allá de su lanzamiento al mercado. Durante esta fase, se genera un informe detallado que describe en profundidad todas las actividades realizadas a lo largo del proceso de desarrollo. Este informe tiene como objetivo identificar tanto los aspectos exitosos como los desafíos encontrados, con el propósito de ofrecer sugerencias para solucionar problemas y evitar que se repitan en futuros proyectos (16).

Estas etapas fueron tomadas en cuenta en la planificación del proceso de desarrollo, tomando en cuenta que la presente investigación se centra en desarrollar las mecánicas que ayudarán a mejorar todo el proceso de desarrollo de los videojuegos.

1.2.1 Motor de videojuegos Unity

Unity es un motor de videojuegos multiplataforma ampliamente utilizado para la creación de juegos y contenido 3D interactivo. Fue creado por *Unity Technologies* y se lanzó originalmente en 2005 para equipos *Mac*. Desde entonces, ha evolucionado y se ha convertido en una herramienta versátil para el desarrollo de juegos en diversas plataformas. Con Unity, los desarrolladores pueden crear juegos para múltiples plataformas a partir de un único desarrollo. Esto incluye consolas de juegos como *PlayStation*, *Xbox y Wii*, así como computadoras de escritorio con sistemas operativos como *Linux*, PC y Mac. También ofrece soporte para navegadores web y dispositivos móviles y tabletas como *iOS, Android, Windows Phone y BlackBerry*(17).

Una de las ventajas de Unity es su compatibilidad con una amplia gama de programas y herramientas. Puede integrarse con programas de modelado y animación 3D populares como *Blender, 3ds Max, Maya, Softimage, Modo, ZBrush, Cinema 4D*, *Cheetah3D* y *Allegorithmic Substance*. Además, permite la importación de archivos de imágenes y gráficos creados en programas como *Adobe Photoshop* y *Adobe Fireworks*. En Unity, las mecánicas de los videojuegos se implementan mediante scripts y se compilan utilizando una versión de *JavaScript* o *C#*. Esto brinda a los desarrolladores flexibilidad y control para crear interacciones y comportamientos personalizados en sus juegos. El motor de Unity se centra en los bloques de construcción, conocidos como "*assets*". Estos *assets* pueden incluir texturas, modelos 3D, archivos de audio, prefabricados (prefabs), materiales y animaciones. Unity facilita la compresión y manipulación de estos *assets* durante la fase de edición, lo que ayuda a mejorar el rendimiento y la eficiencia del juego.(17).

En la figura 4 se muestran los elementos esenciales del editor visual de Unity.

No 1. Ventana de jerarquía: esta ventana visualiza todos los objetos presentes en la escena actual.

No 2. Ventana de escena: es el área de construcción de Unity donde se confecciona visualmente cada escena del juego.

No 3. Ventana de juego: en esta ventana se obtiene una pre visualización del juego. Permite reproducirlo en cualquier momento del desarrollo para ir chequeando que funcione correctamente.

No 4. Ventana de proyecto: permite manipular y visualizar el conjunto de assets que se emplea en el juego. En ella se pueden importar objetos 3D de distintas aplicaciones, texturas, audios, scripts, prefabs, escenas y animaciones, para usarlos en el videojuego.

No 5. Ventana de inspector: la ventana de inspector tiene varias utilidades, si se selecciona un objeto, entonces mostrará sus propiedades y podrán ser personalizadas. También permite configurar ciertas como la de terrenos si se tiene un terreno de Unity seleccionado.

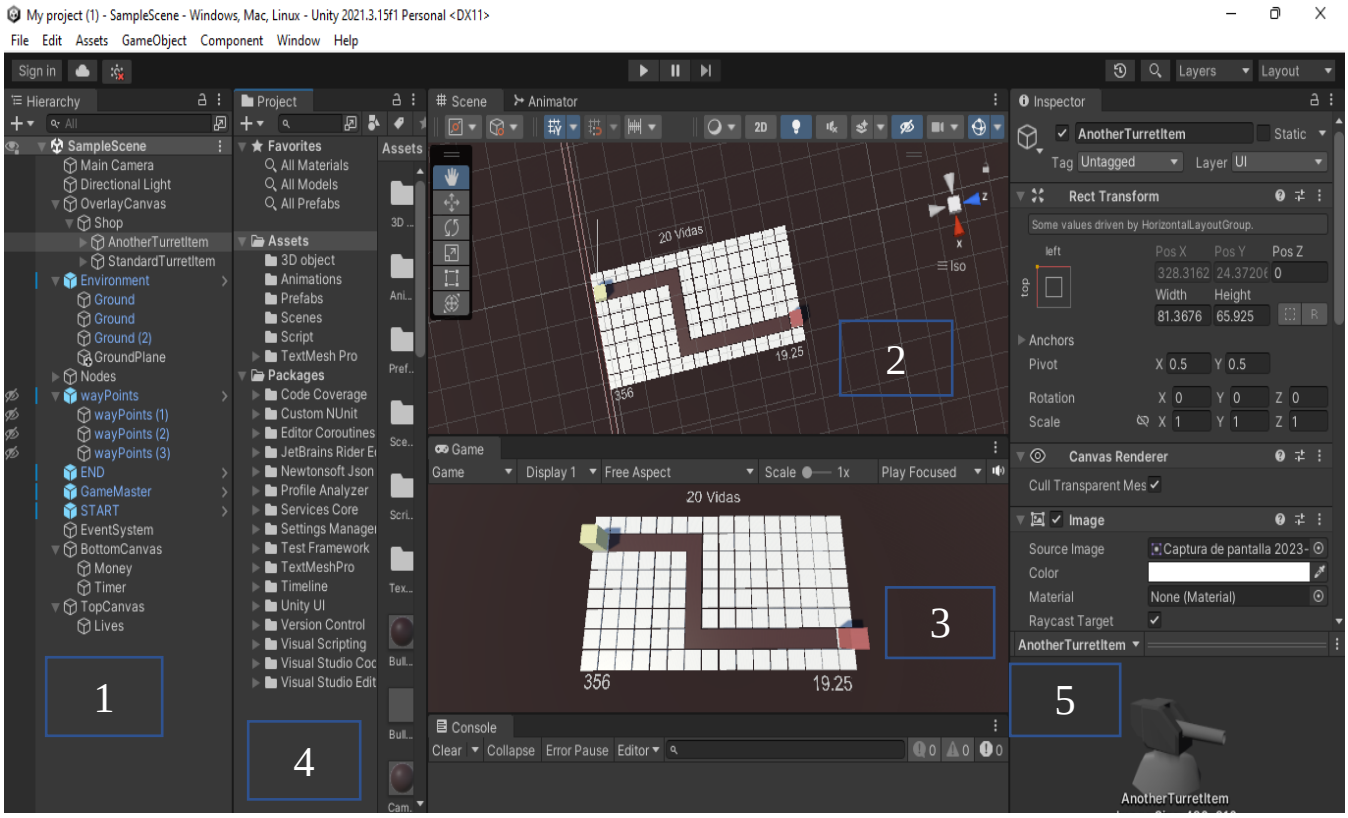

Figura 4. Editor Visual de Unity. Elaboración propia.

Unity utiliza máquinas de estado para controlar las animaciones y transiciones entre los estados de los objetos. Las máquinas de estado se representan mediante un diagrama de flujo o programa visual en la ventana Animator de Unity 3D, donde los nodos representan los estados y los arcos representan las transiciones entre ellos. Estas máquinas de estado se guardan en *assets* llamados *Animator Controller* y pueden ser editados en la ventana *Animator* de Unity.

Este enfoque de programación visual permite a los desarrolladores de juegos crear y controlar de manera más fácil y eficiente la lógica de los diferentes estados y animaciones de cada objeto en el juego. Además, el uso de máquinas de estado puede mejorar el rendimiento del juego al permitir que Unity optimice la manera en que las animaciones se reproducen y las transiciones entre estados se realizan.

1.2.2 Sistemas Relacionados

El desarrollo de videojuegos es un proceso que consume tiempo en su diseño e implementación, sin embargo, existen formas de reducir dicho tiempo, una forma de hacer esto es mediante la reutilización de código. Durante la investigación se analizaron las siguientes herramientas por tener características similares a lo que se propone desarrollar:

Tower Defense Toolkit: Es un *framework* de pago, de codificación en C# para la fácil construcción de juegos *Tower Defense* de cualquier tipo. Viene con un paquete de *scripts* que son flexibles para adaptarse a una gran variedad de escenarios(18).

Tower Defense Starter Kit: Está diseñado para crear juegos en el género de defensa de torres. Se puede encontrar en la *Unity Store* y contiene todo lo necesario para crear defensa de las torres: modo de construcción, componentes de torres, enemigos, *spawn* y *waypoints* sistema, interfaz de juego básico, avanzado, pero fácil de personalizar enemigos y torres de configuración. Es compatible con Unity 2017 – 2020(19).

Estas herramientas creacionales ofrecen una gran flexibilidad y eficiencia en la instanciación de objetos en el desarrollo de videojuegos. Sin embargo, es importante tener en cuenta que su uso conlleva un costo significativo, lo que puede suponer una limitación para un proyecto de desarrollo de videojuegos en el centro Vertex.

1.3 Descripción del marco de trabajo

Para el desarrollo de la solución se determinó utilizar como metodología de desarrollo de software XP (*Extreme Programming*) y como Herramientas para Ingeniería de Software Asistida por Computadora *(herramienta CASE*), *Visual Paradigm*, por la experiencia de trabajo y los buenos resultados que se han obtenido con el uso de estos en VERTEX. También se estableció como lenguaje de programación C#, debido a que es el empleado para desarrollar videojuegos con Unity en dicho centro, así como el Entorno de Desarrollo Integrado (IDE) Microsoft Visual Estudio 2022. A continuación se describen sus principales características:

La metodología ***Extreme programming*** o XP, es una de las metodologías ágiles más reconocida. Fue desarrollada por Kent Beck en la búsqueda por guiar equipos de trabajo pequeños o medianos, entre dos y diez programadores, en ambientes de requerimientos imprecisos o cambiantes La princi-

pal particularidad de esta metodología son las historias de usuario, las cuales corresponden a una técnica de especificación de requisitos; se trata de formatos en los cuales el cliente describe las características y funcionalidades que el sistema debe poseer. En esta metodología se realiza el proceso denominado *Planning game*, que define la fecha de cumplimiento y el alcance de una entrega funcional, el cliente define las historias de usuario y el desarrollador con base en ellas establece las características de la entrega, costos de implementación y número de iteraciones para terminarla. Para cada iteración el cliente estipula cuales son las historias de usuario que componen una entrega funcional. Se realizan entregas pequeñas que son el uso de ciclos cortos de desarrollo, llamado iteraciones, que muestra al cliente una funcionalidad del *software* terminado y se obtiene una retroalimentación de él. Algo muy característico de esta metodología es la programación en parejas, indica que cada funcionalidad debe ser desarrollada por dos programadores, las parejas deben cambiar con cierta frecuencia, para que el conocimiento no sea solo de una persona sino de todo el equipo(20).

Entre los lenguajes de modelado que define OMG (*Object Management Group*) uno de los más conocido y usado es UML (*Unified Modelling Language*). UML es un lenguaje gráfico para especificar, construir y documentar los artefactos que modelan un sistema. UML fue diseñado para ser un lenguaje de modelado de propósito general, por lo que puede utilizarse para especificar la mayoría de los sistemas basados en objetos o en componentes, y para modelar aplicaciones de muy diversos dominios de aplicación (telecomunicaciones, comercio, sanidad, etc.) y plataformas de objetos distribuidos(21).

Las herramientas CASE cuentan con credibilidad, exactitud y tienen un reconocimiento universal. Son usadas por cualquier desarrollador y/o programador que busca un resultado óptimo y eficiente, actualmente brindan una gran gama de componentes que incluyen todos o la mayoría de los requisitos necesarios para el desarrollo de los sistemas. Han sido creadas con una gran precisión en torno a las necesidades de los desarrolladores de software para la automatización de procesos, incluyendo el análisis, diseño e implantación. Ofrecen una gran plataforma de seguridad a sistemas que las usan. Entre las más usadas se encuentra el ***Visual Paradigm***. La misma puede crear ingeniería inversa del código de diagramas y ofrecer ida y vuelta de ingeniería para diversos lenguajes de programación. Esta herramienta ayuda a los desarrolladores de software a crear un modelo de excelencia durante la creación y distribución del proceso de desarrollo de aplicaciones(22).

Microsoft Visual Estudio Community es un (IDE) que se puede utilizar para escribir, editar, depurar y compilar código y, a continuación, implantar tu aplicación. Además de la edición y depuración de código, Visual Studio incluye compiladores, herramientas de completado de código, control de código fuente, extensiones y muchas más funciones para mejorar cada etapa del proceso de desarrollo de software. Además de ser un *software* multiplataforma tiene una amplia compatibilidad con múltiples lenguajes de programación como son *java*, c++, c#, *python*, entre otros (23).

Uno de los lenguajes aceptados en el motor Unity es **C#**. Este es uno de los lenguajes más populares de la industria del software. Es el lenguaje de cabecera de Microsoft, aunque se puede usar en múltiples plataformas de desarrollo de aplicaciones de todo tipo.
Este es un lenguaje de tipado estático y multiparadigma, aunque principalmente orientado a objetos. Microsoft presenta actualizaciones muy frecuentes, por lo que resulta bastante evolucionado, ofreciendo herramientas poderosas para los desarrolladores(24).
Las características esenciales de c# son:

- Multiplataforma, ejecutable en los sistemas más comunes como Windows, MacOs, Linux.
- Sintaxis similar a C, C++, Java y otros.
- Lenguaje de paradigma de programación orientada a objetos, con expresiones de control heredadas de la programación estructurada.
- Incluye algunas características de programación funcional como clojures, aunque es imperativo.
- Fuertemente tipado (tipado estático).
- Lenguaje moderno con actualizaciones de mejoras frecuentes.
- Dispone de un nutrido conjunto de librerías.
- Orientado a componentes.

Con los elementos presentados se conforma el entorno de desarrollo de la solución. Este entorno está fundamentado en la experiencia y las políticas definidas en el centro VERTEX, las cuales se sustentan en las condiciones de cada una y que se han mostrado en el presente capítulo.

Conclusiones del capítulo

- A través del estudio de los videojuegos más relacionados al tipo de Estrategia Tower Defense, se han determinado que no existen paquetes libres disponibles para ser reutilizados en este tipo de videojuegos, confirmando la necesidad de desarrollar el paquete de mecánicas.
- Se identificaron como mecánicas a implementar las de selección, locomoción, tienda, comportamiento del proyectil, vida general, comportamiento de las torres y cámara.
- Tomando en cuenta las características de estos videojuegos, así como las especificidades de la plataforma Unity utilizada en el desarrollo de videojuegos en el centro VERTEX, se determinaron las metodologías y herramientas necesarias para llevar a cabo el desarrollo del presente trabajo.

CAPÍTULO 2: ANÁLISIS Y DISEÑO DE LA SOLUCIÓN

En el presente capítulo se detallan las características de las mecánicas a implementar, se definen los requisitos no funcionales y funcionales de la solución. También se describe la propuesta de solución, así como la arquitectura, la estructura de clases y los patrones empleados para el desarrollo de la misma.

2.1 Propuesta de solución

Para dar solución al problema de investigación planteado, se desarrollará un paquete de mecánicas para la creación de videojuegos del tipo Tower Defense en Unity. Este paquete contiene las mecánicas de juego esenciales para el desarrollo de videojuegos de este tipo, incluyendo la cámara, selección, desplazamiento, tienda, comportamiento del proyectil, vida general y el comportamiento de las torres (Figura 5).

- **Selección:** Esta mecánica les debe permitir seleccionar y controlar diferentes elementos del juego, como torres o habilidades especiales, así como poder seleccionarlas para actualizarlas y venderlas.
- **Locomoción:** Mediante esta mecánica se establece el recorrido que deben hacer las tropas, así como la posibilidad de aumentar su velocidad y vida particular.
- **Tienda:** Esta mecánica debe permitir que se puedan tanto vender como comprar las torres disponibles, así como hacer mejoras y modificar precios.
- **Comportamiento del proyectil:** Debe permitir establecer un tipo de proyectil para cada tipo de torre, así como poder modificar su velocidad, daño, rango de acción.
- **Vida general:** Debe permitir visualizar de una forma fácil la vida general del videojuego, así como poder modificarla para adaptarla a las necesidades.
- **Comportamiento de las torres:** Debe permitir ubicar torres en el mapa en las posiciones donde se pueda, así como modificar los valores del alcance, velocidad de disparo, tipo de proyectil.
- **Cámara:** Debe permitirle al usuario poder navegar por todo el mapa del videojuego, haciendo uso del mouse o utilizando las teclas (w) para subir, (s) para bajar, (a) para desplazarse a la izquierda y (d) para desplazarse a la derecha, así como también poder acercar o alejar el mapa para tener una mejor visión del contorno.

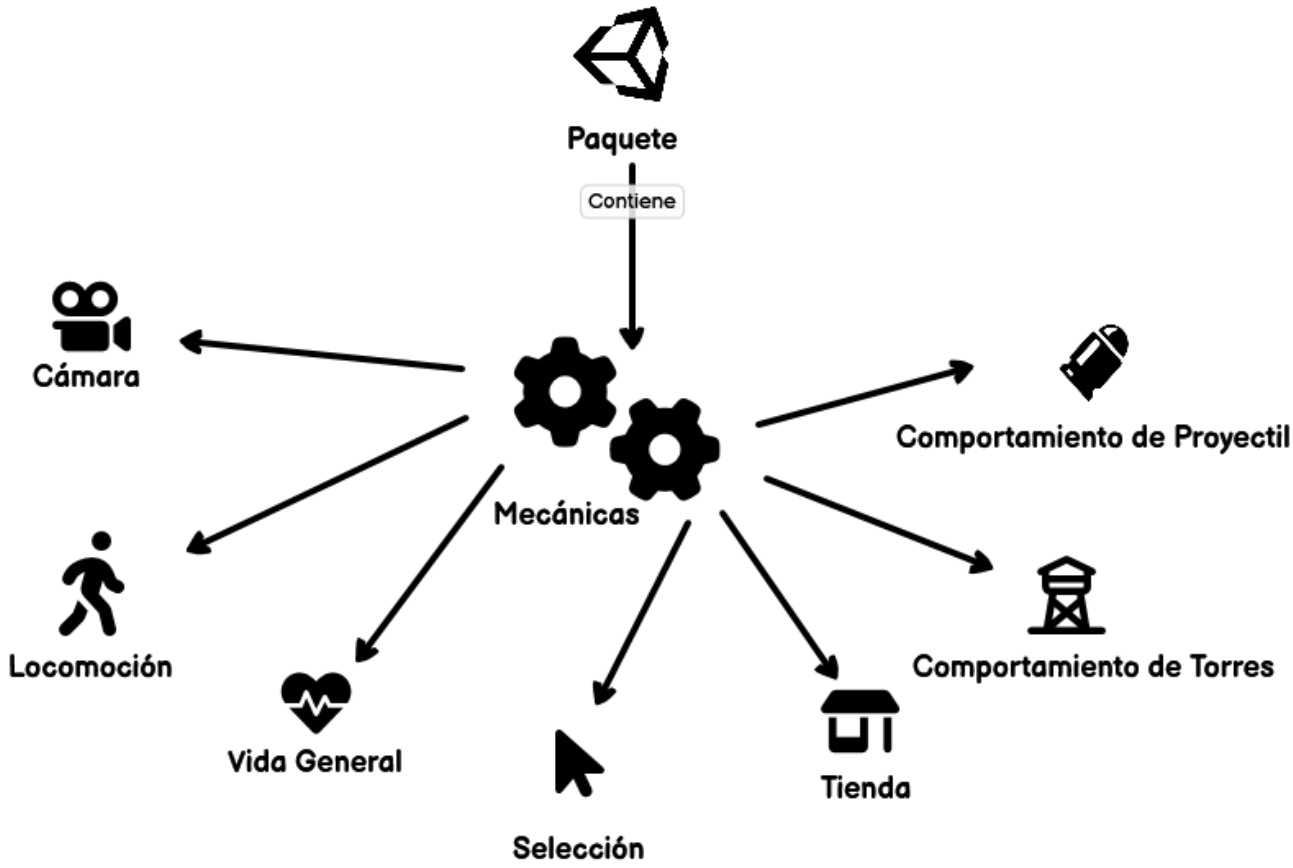

Figura 5. Propuesta de solución. Elaboración propia.

Es importante destacar que el objetivo principal de esta solución es facilitar el proceso de desarrollo de videojuegos de tipo Tower Defense, permitiendo a los desarrolladores enfocarse en la creación de una correcta experiencia de juego para los usuarios. Con esta solución, se espera que los desarrolladores puedan ahorrar tiempo y recursos al no tener que diseñar y programar estos mecanismos y mecánicas de juego desde cero.

2.2 Requisitos no funcionales del sistema

Los Requisitos No Funcionales (RNF) son aquellos que definen las restricciones o cualidades que debe tener un sistema en términos de calidad, en lugar de describir una funcionalidad específica. Estos requisitos se refieren a características como precisión, usabilidad, seguridad, rendimiento, confiabilidad, entre otros aspectos importantes para el sistema(25).

A diferencia de los Requisitos Funcionales (RF) que describen las funciones y comportamientos que el sistema debe tener, los RNF son más abstractos e intangibles. Esto puede hacer que sean más difíciles de especificar o documentar formalmente, su naturaleza a menudo implica aspectos subjetivos y no se puede medir de manera directa (25).

Según la **ISO/IEC 25010** La calidad del producto software se puede interpretar como el grado en que dicho producto satisface los requisitos de sus usuarios aportando de esta manera un valor. Son precisamente estos requisitos (funcionalidad, rendimiento, seguridad, mantenibilidad, entre otros,) los que se encuentran representados en el modelo de calidad, el cual categoriza la calidad del producto en características y subcaracterísticas(29).

Figura 4. Modelo de calidad de un producto. Tomado de*(29)*

Restricciones en la implementación

RNF1. Para el desarrollo del sistema debe emplearse como motor de videojuego Unity.

RNF2. Como lenguaje de programación se empleará C#.

Usabilidad

RNF3. El sistema debe ser manipulado por personas con conocimiento previo en el manejo de la herramienta Unity.

Compatibilidad

RNF4. El sistema debe ser compatible con Unity de la versión 5.2 en adelante.

Mantenibilidad

RNF5. El sistema debe permitir incorporar nuevas funcionalidades para facilitar el crecimiento del mismo en el futuro

2.3 Fase de exploración

Es la fase donde los clientes plantean a grandes rasgos las historias de usuario que son de interés para la primera entrega del producto. Al mismo tiempo el equipo de desarrollo se familiariza con las

herramientas, tecnologías y prácticas que se utilizarán en el proyecto. Se prueba la tecnología y se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo(26).

2.3.1 Historias de Usuario

Las historias de usuario son descripciones cortas y simples de una característica contada desde la perspectiva de la persona que desea la nueva capacidad, generalmente un usuario o cliente del sistema. Por lo general, siguen una plantilla simple (27) y se escriben en fichas o notas.

Tabla 2.HU Insertar puntos de ruta para el movimiento de enemigos.

Historia de Usuario	
No 1. Insertar puntos de ruta para el movimiento de enemigos.	
Prioridad: Alta	Nivel de complejidad: baja
Estimación: 1 días	Iteración asignada: 1
Descripción: El Usuario tendrá la posibilidad de asignarle un script al prefabricado de los puntos de ruta para definir los diferentes caminos que pueden tomar los enemigos.	
Observaciones:	

Tabla 3. HU Insertar mecánica Comportamiento de las Torres

Historia de Usuario	
No 7. Insertar mecánica Comportamiento de las Torres.	
Prioridad: Alta	Nivel de complejidad: alto
Estimación: 10 días	Iteración asignada: 2

Descripción: El Usuario tendrá la posibilidad de asignarle a un objeto en la escena un *script* que contiene los atributos y mecanismos que conforman las torres.	
Observaciones:	

Tabla 4. Insertar mecánica Cámara.

Historia de Usuario	
No 9. Insertar mecánica Cámara.	
Prioridad: Alta	Nivel de complejidad: alto
Estimación: 10 días	Iteración asignada: 2
Descripción: El Usuario tendrá la posibilidad de asignarle un script a la cámara del juego para hacer uso de esta.	
Observaciones:	

2.4 Fase de planificación

En esta fase el cliente establece la prioridad de cada historia de usuario, y correspondientemente, los programadores realizan una estimación del esfuerzo necesario de cada una de ellas. Se toman acuerdos sobre el contenido de la primera entrega y se determina un cronograma en conjunto con el cliente. Una entrega debería obtenerse en no más de tres meses (26).

2.4.1 Plan de iteraciones

Esta fase incluye varias iteraciones sobre el sistema antes de ser entregado. El Plan de Entrega está compuesto por iteraciones de no más de tres semanas. En la primera iteración se puede intentar establecer una arquitectura del sistema que pueda ser utilizada durante el resto del proyecto. Esto se logra escogiendo las historias que fuercen la creación de esta arquitectura, sin embargo, esto no siempre es posible. Muchas veces el cliente decide qué historias se implementarán en cada iteración

(para maximizar el valor de negocio). Al final de la última iteración el sistema estará listo para entrar en producción(26).

Tabla 5. Distribución de iteraciones por historias de usuario

Iteraciones	Historias de usuario	Tiempo de trabajo
Iteración 1	➢ Insertar puntos de ruta para el movimiento de enemigos. ➢ Implementación del sistema de generación de oleadas. ➢ Insertar *script* de enemigos. ➢ Configurar *script* de enemigos. ➢ Insertar mecánica de locomoción. ➢ Modificar mecánica de locomoción.	25 días
Iteración 2	➢ Insertar mecánica Comportamiento de las Torres. ➢ Modificar mecánica Comportamiento de las Torres. ➢ Insertar mecánica Comportamiento del Proyectil. ➢ Configurar mecánica Comportamiento del Proyectil.	20 días
Iteración 3	➢ Insertar mecánica Vida General. ➢ Configurar mecánica Vida general. ➢ Insertar mecánica Cámara. ➢ Configurar mecánica Cámara. ➢ Visualizar estadísticas. ➢ Establecer condición de pérdida del juego. ➢ Establecer lógica del juego.	15 días

Iteración 4	➢ Insertar mecánica Selección. ➢ Visualizar estructuras seleccionadas. ➢ Insertar mecánica Tienda. ➢ Modificar mecánica Tienda.	25 días

2.4.2 Plan de entregas

La idea del plan de entregas es producir rápidamente versiones del sistema que sean operativas, aunque obviamente no cuenten con toda la funcionalidad pretendida para el sistema pero sí que constituyan un resultado de valor para el negocio (26), lo que es muy importante para llevar a cabo la estimación del tiempo de entrega del producto y contribuir a eliminar insatisfacciones por parte del cliente.

Tabla 6. Plan de entregas del producto

Producto	1ra iteración	2da iteración	3ra iteración	4ta iteración
Paquete de mecánicas para videojuegos del tipo Tower Defense.	**Versión 0.1 del producto.**	**Versión 0.2 del producto.**	**Versión 0.3 del producto.**	**Versión 1.0 del producto.**

2.5 Fase de diseño

Según XP se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto. La complejidad innecesaria y el código extra debe ser removido inmediatamente. Kent Beck dice que en cualquier momento el diseño adecuado para el software es aquel que: supera con éxito todas las pruebas, no tiene lógica duplicada, refleja claramente la intención de implementación de los programadores y tiene el menor número posible de clases y métodos (26).

2.5.1 Descripción de la arquitectura

Unity presenta una arquitectura por capas, por tanto el sistema a desarrollar debe adaptarse a este tipo de arquitectura. En la Figura 6 se describen cada una de sus capas.

- **Capa de interfaz:** Esta capa se encarga de mostrar los gráficos y la interfaz de usuario del juego. Incluye elementos como la cámara, los efectos visuales, la interfaz de usuario, entre otros.
- **Capa de lógica del negocio:** Esta capa contiene la lógica del juego, incluyendo la inteligencia artificial, la física, las mecánicas de juego. Aquí se definen los comportamientos de los personajes y objetos del juego, así como las reglas y la dinámica del mismo.
- **Capa de soporte:** Esta capa es responsable de la gestión de los recursos del juego, como el sonido, las imágenes, las animaciones, los modelos 3D. Aquí se definen y organizan los recursos que se utilizan en el juego, y se asegura su correcta carga y descarga durante la ejecución del mismo.

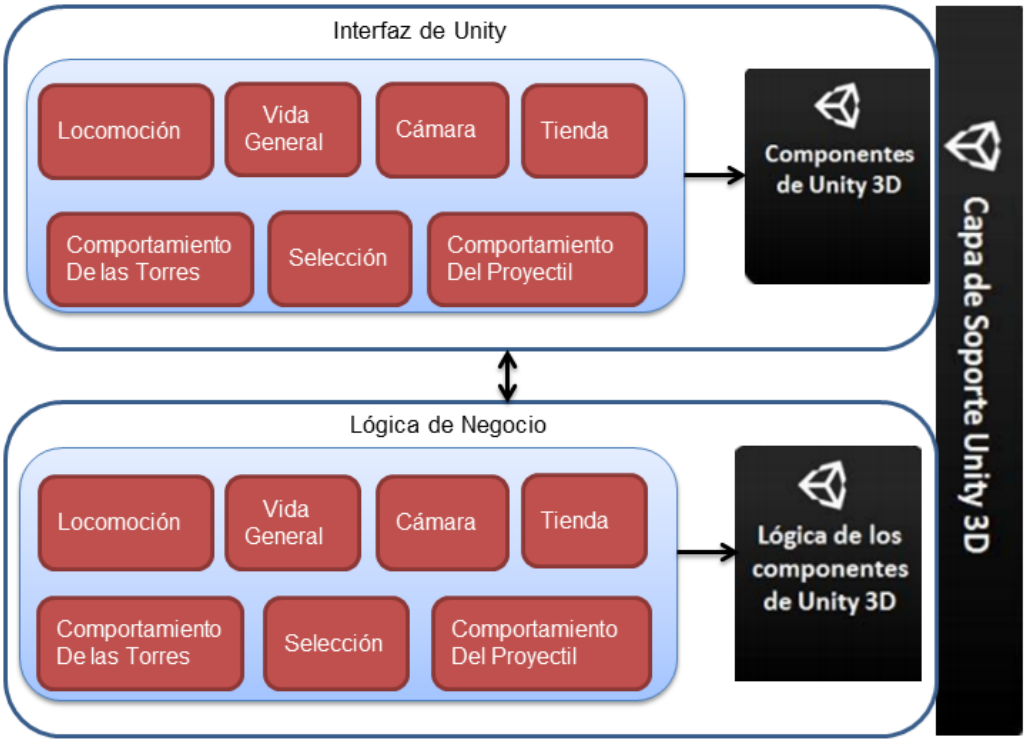

Figura 4. Diseño de la arquitectura. Elaboración propia.

2.5.2 Patrones de diseño

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma. En la ingeniería del software, un patrón constituye el apoyo para la solución a los problemas más comunes que se presentan durante las diferentes etapas del ciclo de vida del software(28). Los desarrolladores lo utilizan como una forma de reutilizar la experiencia.

A continuación se representan los patrones GRASP que se usarán en el desarrollo de la solución.

Bajo acoplamiento: es una medida de la fuerza con que una clase se relaciona con otras, porque las conoce y recurre a ellas(30), de esta forma se logra que el número de relaciones entre estas clases sea el mínimo para evitar que en caso de modificación entre ellas la repercusión en otras sea la mínima posible.

Alta cohesión: es una medida que determina cuán relacionadas y adecuadas están las responsabilidades de una clase, de manera que no realice un trabajo colosal, una clase con baja cohesión realiza un trabajo excesivo, haciéndola difícil de comprender, reutilizar y conservar(30).

La siguiente figura muestra un fragmento tomado del diagrama de clases para demostrar la alta cohesión que existe entre las clases Torres, *Bullet y Enemy*, las cuales se relaciones entre sí pero solo su relación está enfocada en la misma tarea.

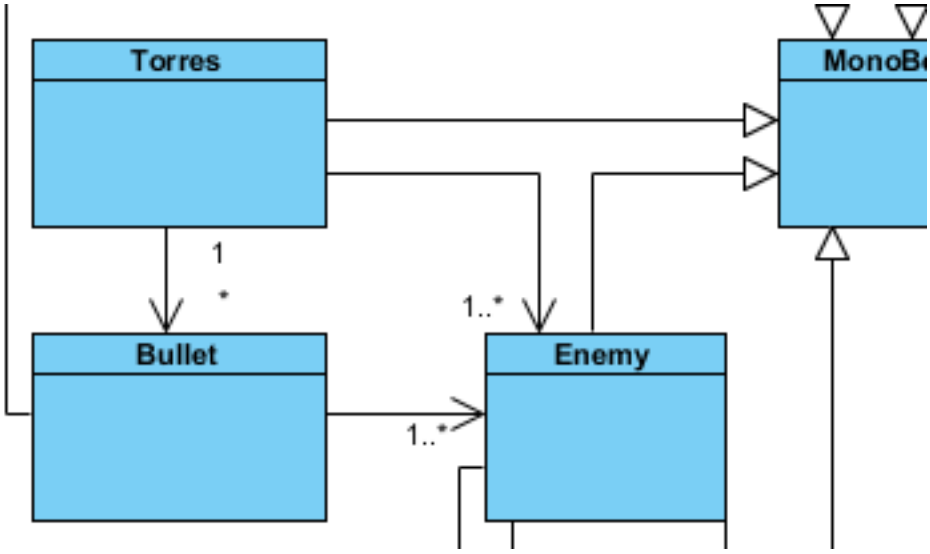

Figura 5. Ejemplo de Alta cohesión entre las clases. Elaboración propia.

También de utilizaran patrones GOF para el desarrollo de la solución, los mismos se dividen en tres grupos:

Comportamiento: Definen la comunicación e iteración entre los objetos de un sistema. El propósito de este tipo de patrón es reducir el acoplamiento entre los objetos(31). (Cadena de responsabilidad, Orden, Intérprete, Iterador, Mediador, Observador, Estado, Estrategia, Método plantilla, Visitante).

Estructurales: Describen cómo clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos(31).

Creacionales: Tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados(31).

De los patrones GOF anteriormente mencionados para la solución se emplearán los patrones creacionales. Dichos patrones se utilizarán para instanciar los objetos necesarios para lograr acceder a ellos desde otras ubicaciones. En la figura 7 se muestra un ejemplo de cómo se utiliza el patrón creacional en el código:

```
void HitTarget()
{
    GameObject effectIns = (GameObject)Instantiate(impactEffect, tr
ansform.position, transform.rotation);
    Destroy(effectIns, 5f);

    if (explosionRadius > 0f)
    {
        Explode();
    }
    else
    {
        Damage(target);
    }

    Destroy(gameObject);
}
```

Figura 6. Patrones GOF Creacionales. Elaboración propia.

2.5.3 Descripción de las tarjetas CRC.

Una tarjeta CRC es una ficha que representa una entidad del sistema, a la cual se le asignan responsabilidades y colaboraciones, facilita la interacción entre los participantes del proyecto en sesiones donde se aplican técnicas de grupo como tormenta de ideas y juego de roles, y se ejecutan escenarios a partir de la especificación de requisitos, historias de usuario o casos de uso(32).

Tabla 7. Tarjeta CRC EnemyMovement

Tarjeta CRC	
Clase: *EnemyMovement.*	
Responsabilidades	**Clases Relacionadas**
• Dirigir a la unidad hasta el destino especificado. • Cambiar de punto de ruta • Finalizar el recorrido	• *MonoBehaviour* • *Enemy* • *WayPoints*

Tabla 8. Tarjeta CRC Camera

Tarjeta CRC	
Clase: *Camera*	
Responsabilidades	**Clases Relacionadas**

• Controlar el movimiento de la cámara. • Controlar el zoom de la cámara. • Dirigir la cámara hacia una posición específica.	• *MonoBehaviour* • *GameManager*

Tabla 9. Tarjeta CRC Shop

Tarjeta CRC	
Clase: *Shop*	
Responsabilidades	**Clases Relacionadas**
• Insertar una nueva unidad en el terreno • Manipular los recursos existentes	• *MonoBehaviour* • *BuildManager* • *TurretBlueprint*

Para un mejor entendimiento de la solución se desarrolló el diagrama de clases que se muestra en la Figura 8:

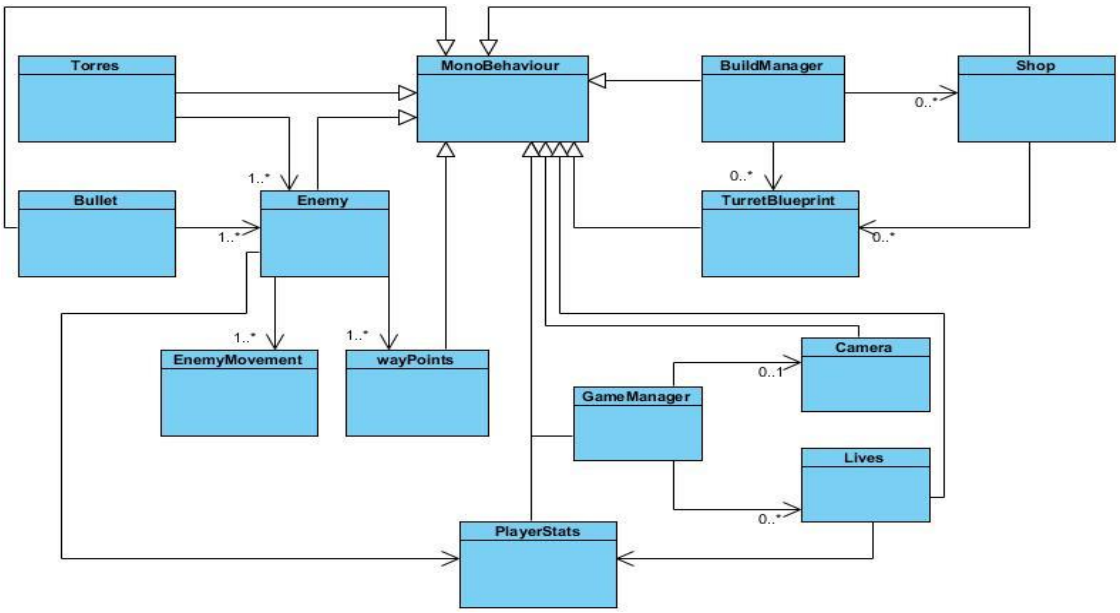

Figura 7. Diagrama de clases de la solución. Elaboración propia.

2.5.4 Estándares de codificación

Durante el desarrollo de la solución, se establecen convenciones para la escritura del código fuente, siguiendo el estándar de codificación del lenguaje C#. Estas convenciones abarcan aspectos como el

nombre de las variables, métodos y clases, el estilo de sangrado y el uso del idioma inglés. El objetivo principal de este estándar de codificación es mejorar la legibilidad del código, lo cual tiene un impacto directo en la comprensión del software por parte de los programadores. Al enfocarse en la legibilidad del código, se facilita el mantenimiento del sistema, lo que implica la capacidad de realizar modificaciones, añadir nuevas características, corregir errores y mejorar el rendimiento. Cuando el código es fácil de entender, los programadores pueden navegar y comprender rápidamente su estructura, lo que agiliza el proceso de desarrollo y reduce la posibilidad de introducir nuevos errores.

Definiciones generales

Las definiciones se redactan en inglés de forma descriptiva, evitando el uso de abreviaturas y nombres cortos. Se prefiere utilizar este idioma debido a que las palabras son sencillas, no llevan acento y es ampliamente utilizado en el ámbito informático. Esta elección busca mejorar la legibilidad y comprensión del código, facilitando el mantenimiento y permitiendo la incorporación de nuevas funcionalidades, correcciones de errores y mejoras de rendimiento en el software.

Declaración de variables: Los nombres de las variables siempre aparecen en minúscula menos en el caso en que el nombre sea compuesto, en este caso la primera letra del segundo nombre comenzara con mayúscula y todo lo demás con minúscula.

Ejemplo:

float speed;

int worth;

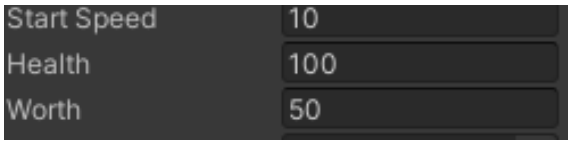

Figura 8. Variables. Elaboración propia.

Métodos: Los nombres de los métodos deben comenzar con mayúscula y en caso que sean compuestos la segunda palabra debe comenzar también, los constructores deben llevar el nombre de la clase.

Ejemplo:

Métodos:

void Die();

TakeDamage();

```
void Die()
{
    PlayerStats.Money += worth;
    GameObject effect = (GameObject)Instantiate(deathEffect, transform.transform.position, Quaternion.identity);
   Destroy(effect, 5f);
    Destroy(gameObject);
}
```

Figura 9. Método Die. Elaboración propia.

```
public void TakeDamage(float amount)
{
    health-=amount;

    if (health <= 0)
    {
        Die();

    }
```

Figura 10. Método *TakeDamage*. Elaboración propia.

Clases: Las clases comienzan con mayúscula al inicio de la palabra y en caso de estar conformada por palabras compuestas, la definición debe ser continua y cada palabra debe iniciar con mayúscula siguiendo el estilo determinado.

Ejemplo:

public class *PlayerStats{*

}

```
public class PlayerStats : MonoBehaviour
{
    public static int Money;
    public int startMoney = 400;

    public static int Lives;
    public int startLives = 20;

    public static int Rounds;

    Unity Message | 0 references
    private void Start()
    {
        Money = startMoney;
        Lives = startLives;

        Rounds = 0;
    }

}
```

Figura 11. Clase PlayerStats. Elaboración propia.

Conclusiones del capítulo

- Los artefactos generados por la metodología empleada han facilitado la comprensión de las funcionalidades, elementos y guía de trabajo para los desarrolladores a lo largo de todo el ciclo de desarrollo de la solución.
- La planificación llevada a cabo ha permitido asignar el tiempo necesario para cada historia de usuario. Además, la adaptación de la solución a la arquitectura de Unity ha permitido que la lógica de los mecanismos sea independiente del usuario, lo que brinda la posibilidad de interactuar con ellos a través de interfaces visuales intuitivas y amigables.
- La descripción detallada de las clases de la solución y sus relaciones mediante el uso de tarjetas CRC y el diagrama de clases ha proporcionado una visión más clara y un mejor entendimiento de la estructura del sistema.

CAPÍTULO 3: VALIDACIÓN DE LA SOLUCIÓN PROPUESTA

Después de completar el diseño de un proyecto, es fundamental establecer tareas específicas en cada iteración de implementación para orientar el trabajo. En este capítulo se muestra el proceso de implementación de la solución, donde se definen diversas actividades de ingeniería en cada iteración. El capítulo concluye con la realización de pruebas en colaboración con el cliente para evaluar si la solución cumple con los objetivos establecidos y se creó un demo para verificar la flexibilidad, reusabilidad y escalabilidad de la solución.

3.1 Fase Implementación

De acuerdo con la metodología XP (Programación Extrema), la etapa de implementación o desarrollo del sistema se considera fundamental. Esta permite obtener resultados concretos al finalizar cada iteración, generando así una versión funcional del producto. Esta versión es presentada y probada ante el cliente, brindando valiosa retroalimentación al equipo de trabajo. A continuación, se detallan las cinco iteraciones generadas a partir de la planificación descrita en el capítulo anterior, así como las tareas de ingeniería definidas para la realización de cada Historia de Usuario (HU)(33).

3.1.1 Primera Iteración

En esta iteración se desarrollan las Historias de usuario que se definieron como las más importantes y por ende las de mayor prioridad, con el objetivo de obtener una primera versión del producto con la locomoción de los enemigos completamente funcional. Para ello se trazaron 5 tareas, las siguientes tablas muestran 2 de ellas y las restantes se encuentran en el anexo 2:

Tabla 10. Tarea 1 iteración 1

Tarea	
Número de la tarea: 1	**Número de HU: 1**
Nombre de la tarea: Implementación de los atributos configurables de la clase *WayPoints*	
Tipo de tarea: Implementación.	**Estimación:** 1 día
Descripción: Se definen los atributos necesarios para almacenar y acceder a los puntos de ruta de manera conveniente desde otros *scripts.*	

Tabla 11. Tarea 2 iteración 1

Tarea	
Número de la tarea: 2	**Número de HU: 2**
Nombre de la tarea: Implementación de la clase *Enemy.*	
Tipo de tarea: Implementación	**Estimación:** 6 días
Descripción: Se define el comportamiento de los enemigos, incluyendo su salud, velocidad, ralentización y muerte de forma tal que se puedan manipular estos atributos de forma fácil desde el inspector de Unity.	

Al culminar el desarrollo de la iteración se realizaron las pruebas de aceptación No. 1, 2, 3 y 4, que se explican en el anexo 3. Estas arrojaron un total de 4 defectos, 3 significativos y 1 no significativo que fueron solucionados antes de dar inicio a la siguiente iteración.

3.1.2 Segunda Iteración

En esta iteración se incorpora a la solución todo lo relacionado a las defensas, para definir un rango y cadencia, así como establecer qué proyectil se utilizará y las prestaciones que este tendrá. A continuación se muestran dichas iteraciones:

Tabla 12. Tarea 1 iteración 2

Tarea	
Número de la tarea: 1	**Número de HU: 7**
Nombre de la tarea: Implementación de la mecánica para el comportamiento de las Torres.	
Tipo de tarea: Implementación	**Estimación:** 10 días
Descripción: Se define el comportamiento de las torres, incluyendo la búsqueda y seguimiento de enemigos, el disparo de proyectiles o el uso de láseres y la rotación de la torre para apuntar al enemigo objetivo.	

Tabla 13. Tarea 2 iteración 2

Tarea	
Número de la tarea: 2	**Número de HU: 11**
Nombre de la tarea: Implementación del *Script Bullet* para la mecánica de comportamiento del proyectil.	
Tipo de tarea: Implementación	**Estimación:** 10 días
Descripción: Se definen atributos y se implementan métodos para lograr el correcto funcionamiento de los proyectiles, así como su sencilla manipulación desde el inspector de Unity y su fácil integración en los *prefab* de torres.	

En el acápite 3.2.2 y en el anexo 3 se explican las pruebas de aceptación No. 5 y 6, realizadas al culminar esta iteración.

3.1.3 Tercera Iteración

En esta iteración se incorpora a la solución el control de la cámara para posibilitar el desplazamiento por todo el campo de juego, así como la posibilidad de manipular y observar la vida general del juego para tener una referencia de lo que está sucediendo. A continuación se muestran dos de ellas y las demás en el anexo 2:

Tabla 14. Tarea 1 iteración 3

Tarea	
Número de la tarea: 1	**Número de HU: 10**
Nombre de la tarea: Implementación de los atributos configurables de la clase Camera	
Tipo de tarea: Implementación	**Estimación:** 4 días
Descripción: Se definen los atributos configurables de la clase Camera para garantizar el correcto funcionamiento del *script* Camera y de esta forma garantizar el correcto funcionamiento de la cámara del videojuego.	

Tabla 15. Tarea 3 iteración 3

Tarea	
Número de la tarea: 3	**Número de HU: 15**
Nombre de la tarea: Implementación del *script PlayerStats*	
Tipo de tarea: Implementación	**Estimación:** 3 días
Descripción: Se inicializan las variables necesarias para mantener y administrar las estadísticas del jugador como son: dinero, vidas y cantidad de rondas sobrevividas y además poder modificar el valor inicial de la vida y los recursos mediante el inspector de Unity.	

Al culminar el desarrollo de la iteración se llevaron a cabo las pruebas de aceptación No. 7, 8, 9, 10 y 11, las cuales se explican en el acápite 3.2.2 y en el anexo 3.

3.1.4 Cuarta Iteración

Esta iteración tiene como objetivo incorporar un script que el usuario pueda utilizar para construir la tienda del videojuego, también incorpora a la solución todo lo necesario para realizar las selecciones de estructuras y elementos en la tienda. A continuación se muestran dos de estas tareas y las demás en el anexo 2:

Tabla 16. Tarea 1 iteración 4

Tarea	
Número de la tarea: 1	**Número de HU: 18**
Nombre de la tarea: Implementación de selección de puntos en el terreno con un clic.	
Tipo de tarea: Implementación	**Estimación:** 10 días
Descripción: Se implementa la lógica de selección en un punto del mapa mediante un clic y proporciona los medios para saber si se puede construir o no.	

Tabla 17. Tarea 2 iteración 4

Tarea	
Número de la tarea: 2	**Número de HU: 19**
Nombre de la tarea: Implementación del *script NodeUI.*	
Tipo de tarea: Implementación.	**Estimación:** 7 días
Descripción: Se implementan los métodos necesarios para permitir seleccionar estructuras y hacer sobre ella una serie de acciones como son: actualizar, vender.	

En el anexo 3 se explican las pruebas de aceptación 12, 13, 14 y 15 realizadas al terminar la iteración.

3.2 Fase de Pruebas

Las pruebas de software son esenciales para garantizar el correcto cumplimiento de la funcionalidad del producto. Al realizar las pruebas, se verifica que el software se comporte de acuerdo con las especificaciones establecidas y las necesidades del usuario. Esto garantiza que el producto entregado sea confiable y cumpla con las expectativas de los usuarios.

Los usuarios confían en un producto que ha sido sometido a pruebas y ha demostrado su calidad. Esta confianza se traduce en una mejor recepción y aceptación del producto en el mercado, lo que puede tener un impacto positivo en la reputación y el éxito económico de la empresa.

Otro aspecto importante de las pruebas de software es que permiten detectar y corregir defectos antes de que el producto se lance al mercado. Al encontrar y solucionar estos defectos durante las etapas de desarrollo y pruebas, se evita que lleguen a producción y afecten a los usuarios finales. Esto no solo reduce los costos asociados con la corrección de errores en etapas posteriores, sino que también evita posibles consecuencias negativas (34).

3.2.1 Pruebas Unitarias

Las pruebas unitarias son las pruebas creadas para comprobar que cada unidad de código funcione correctamente y cumpla con su propósito previsto. Se escriben utilizando un marco de pruebas (como *NUnit* o *Microsoft's Unit Testing Framework*) y se ejecutan de manera automatizada, se diseñan para probar escenarios específicos y verificar que el código produzca los resultados esperados. Pueden incluir diferentes casos de prueba para cubrir diferentes caminos de ejecución, como entradas válidas e inválidas. Las pruebas unitarias son una parte fundamental del desarrollo de software, se enfocan en verificar el funcionamiento individual de unidades de código, como funciones, métodos o clases, de forma aislada (35).

En la Figura 13 se muestra un caso de prueba unitaria realizada al procedimiento *SpawnTest* de la clase *WaveSpawner,* el cual inicialmente arrojó un defecto. La oleada que se ejecutaba era la primera, sin embargo se habían inicializado los valores para que iniciara en la segunda. Dicho defecto fue solucionado en una segunda iteración. En el anexo 4 se pueden encontrar otros ejemplos de estas pruebas, realizadas a los principales procedimientos de la solución.

Caso de Prueba Unitaria

```
void SpawnTest()
  {
    // Reiniciar las variables necesarias para la prueba
    waveIndex = 2;
    countdown = 0f;
    EnemiesAlive = 0;

    // Iniciar la generación de oleadas manualmente
    StartCoroutine(SpawnWave());
    Debug.Log("Al ejecutarse el juego debe visualizarse solo la tercera oleada, ya que el contador se
inicializo en 2");
  }
```

Resultado

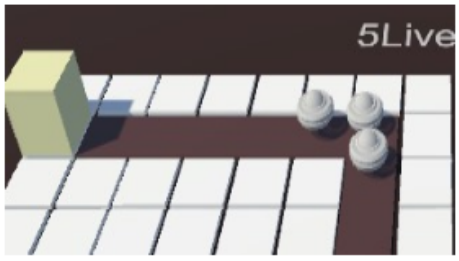

Figura 12. Caso de prueba unitaria SpawnTest

Eliminar los defectos en el código de un software no es suficiente para garantizar que la solución cumple con las especificaciones del cliente. Incluso si el código ha sido corregido y no presenta defectos, todavía existe la posibilidad de que la funcionalidad implementada no se ajuste a los requerimientos y expectativas del cliente. Es por eso que se requiere realizar otro tipo de prueba como la que a continuación se describe.

3.2.2 Pruebas de Aceptación

Las pruebas de aceptación se llevan a cabo en colaboración entre el cliente y el equipo de desarrollo, y su objetivo principal es verificar que el sistema cumpla con las especificaciones y requisitos acordados desde la perspectiva del usuario final(36). A continuación se muestran algunas de las pruebas de aceptación realizadas a la solución, las restantes se pueden encontrar en el anexo 3.

Tabla 18. Caso de Prueba de Aceptación P7HU7

Caso de Prueba de Aceptación	
Código : P7HU7	**Número de HU:** 7
Nombre: Insertar mecánica Comportamiento de las Torres.	
Descripción: Se debe probar que se puedan insertar torres en el terreno de juego.	
Condiciones de ejecución: Se debe tener un prefabricado de torres en la escena para llevar a cabo la asignación del *script*.	
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado de torres que se encuentra en la escena. Arrastra el *script* Torres hacia la ventana del inspector de Unity donde se encuentra abierto el prefabricado.	
Resultado esperado: Correcta inserción de la torre en el mapa del juego y correcta modificación de los valores configurables en el inspector de Unity.	
Evaluación de la prueba: Resultado satisfactorio.	

Tabla 19.Caso de Prueba de Aceptación P9HU9

Caso de Prueba de Aceptación	
Código : P9HU9	**Número de HU:** 9
Nombre: Insertar mecánica de Cámara.	
Descripción: Se debe probar que se pueda insertar correctamente el *script Camera* en el juego.	
Condiciones de ejecución: Se debe asegurar que no exista otra cámara en el juego.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto cámara Arrastra el *script Camera* hacia el inspector de Unity donde se encuentra abierto el objeto cámara.	
Resultado esperado: Correcta inserción del *script Camera*.	
Evaluación de la prueba: Resultado no satisfactorio.	

Tabla 20.Caso de Prueba de Aceptación P13HU13

Caso de Prueba de Aceptación

Código : P13HU13	**Número de HU:** 13
Nombre: Insertar mecánica de Vida General.	
Descripción: Se debe probar que se pueda insertar un *script* que se encargue de manipular la vida general del juego.	
Condiciones de ejecución: Se debe tener en la escena un objeto para realizar la inserción.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script*, se recomienda que sea en un objeto que se encuentre en una interfaz de usuario. Arrastra el *script Lives* hacia el inspector de Unity donde se encuentra abierto el objeto a utilizar.	
Resultado esperado: Correcta inserción del *script Lives.*	
Evaluación de la prueba: Resultado satisfactorio.	

Durante el proceso de desarrollo, se llevaron a cabo pruebas de las funcionalidades después de cada una de las cuatro iteraciones. Se identificaron defectos y se corrigieron antes de comenzar la siguiente iteración. Algunos de los defectos significativos estaban relacionados con fallas específicas, como la falta de detención de la cámara en los límites del terreno, desplazamiento erróneo de los enemigos. Por otro lado, los defectos no significativos estuvieron asociados fundamentalmente a errores ortográficos a la hora de desarrollar el código. En la figura 14 se puede observar, por cada iteración, el número de fallas significativas, no significativas y las recomendaciones.

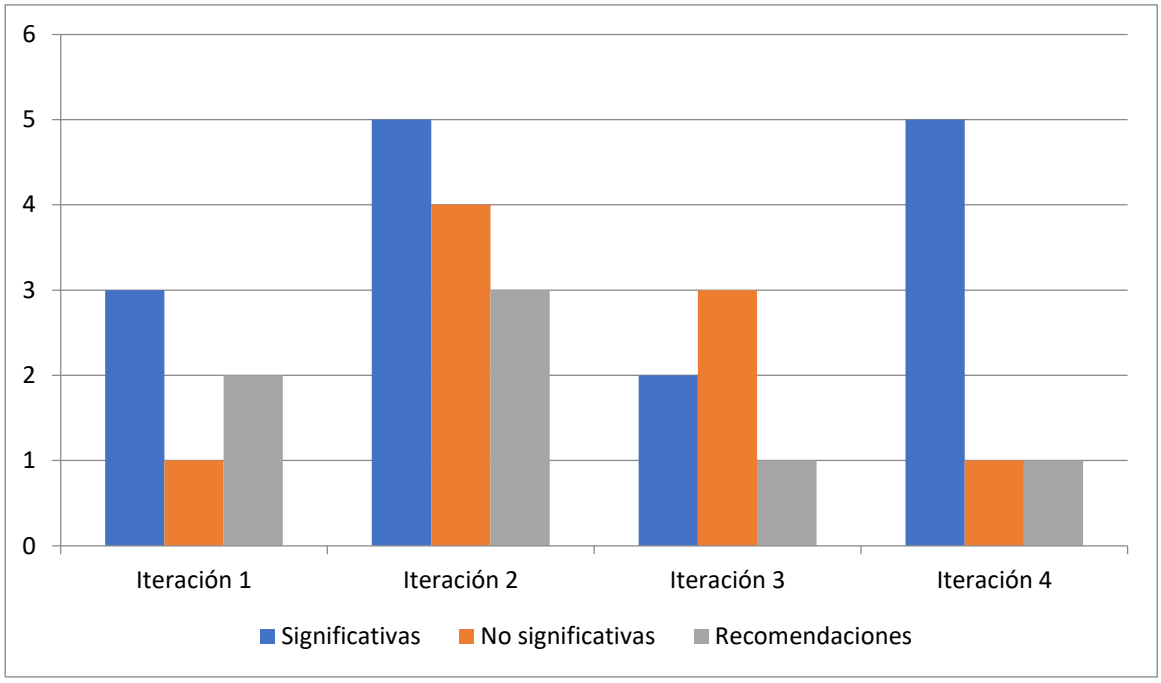

Figura 13. Resultado de las pruebas por iteraciones.

Una vez culminadas las pruebas, se procedió a desarrollar un demo en Unity para demostrar su reusabilidad, flexibilidad y escalabilidad.

3.3 Análisis de Resultados

En el demo que se ha desarrollado, se presenta un escenario con un camino que atraviesa un mapa, donde ocurren interacciones entre los defensores (torres) y los nvasores (enemigos). Las torres desempeñan un papel crucial en la defensa, su objetivo es evitar que los enemigos logren cruzar el camino y llegar a su destino final.

Los enemigos, por su parte, siguen el camino, confiando en su velocidad y resistencia, que varían según su tipo. Sin embargo, las torres están preparadas para enfrentar esta amenaza con una variedad de proyectiles, como láseres, explosivos, flechas y proyectiles convencionales. Cada uno de estos proyectiles posee diferentes velocidades y capacidades de daño, lo que les permite detener el avance enemigo y reducirlos a cenizas.

Es importante mencionar que los objetos visuales utilizados en el demo fueron obtenidos de la página oficial del artista gráfico Kenney (30). En la figura 15 se muestra una captura de pantalla del demo en ejecución.

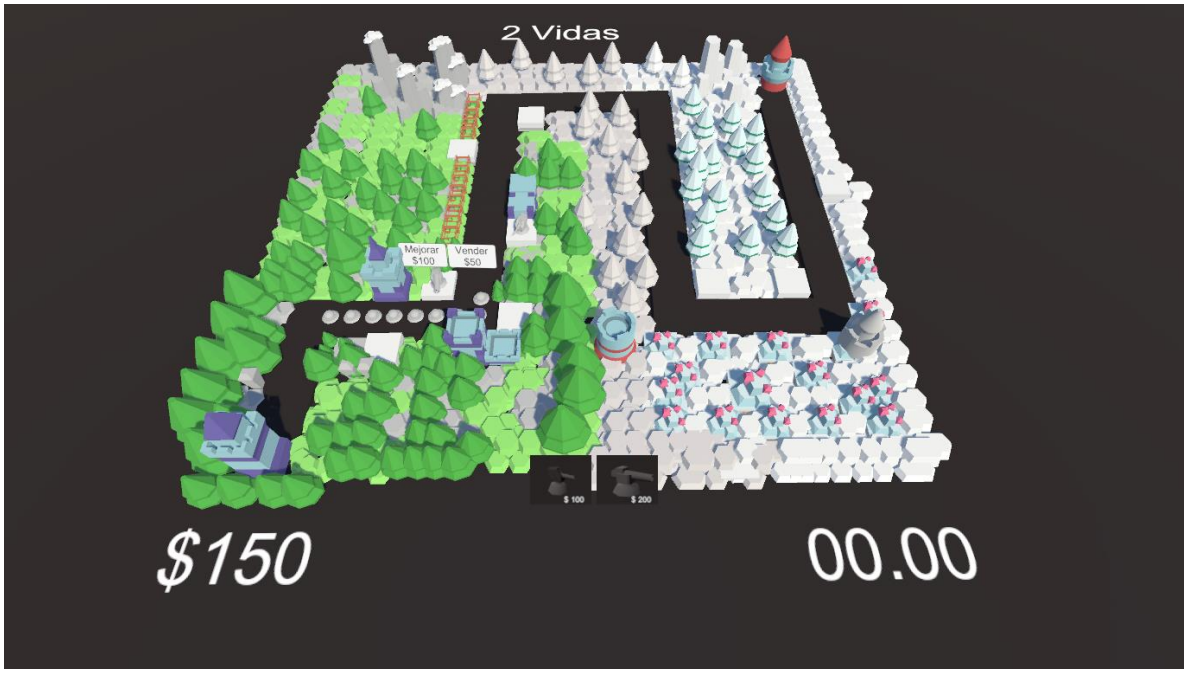

Figura 15. Imagen del demo. Elaboración propia.

En la figura se aprecia el desplazamiento de los enemigos desde su punto de origen hacia su punto de destino, por otro lado, se puede observar la selección de las torres y se muestran las acciones que se pueden desarrollar sobre ellas una vez seleccionadas.

Para evidenciar la presencia de reusabilidad, escalabilidad y flexibilidad en el demo, primero que todo se deben conocer los conceptos de cada uno de ellos que se evidencian a continuación y que se extrajeron utilizando como referencia (37):

Reusabilidad: Capacidad de utilizar componentes o módulos de software existentes en diferentes contextos o aplicaciones, sin necesidad de volver a desarrollarlos desde cero.

Flexibilidad: Capacidad de adaptarse y responder a cambios de manera eficiente.

Escalabilidad: capacidad de un sistema para manejar un aumento en la carga de trabajo.

A continuación se describirá cómo se abordaron cada una de estas características en el demo.

El primer paso para utilizar el paquete desarrollado fue crear un nuevo proyecto en Unity y construir un campo de batalla haciendo uso de nodos, luego se trazó el camino haciendo uso de los puntos de ruta que trae definidos Unity, para poder acceder a estos puntos de ruta se creó un objeto al que se le inserto el *script Way Points*. Para insertar los enemigos, se arrastran hacia la escena los *prefabs* de enemigos creados y se les asigna el *script Enemy*. De esta forma se pueden configurar los enemigos.

Igualmente se realizó la inserción de las torres y los proyectiles, pero estos no se insertan directamente en la escena, sino que se insertan dentro del prefabricado de la torre. Para manipular la cámara del juego a conveniencia del usuario sólo se le añadió un *script* al objeto de la cámara que se encuentra por default en la escena. Para Las mecánicas Vida general y Tienda se crearon *canvas* que no son más que interfaces de usuario, luego se crearon los objetos que contendrían los scripts y se agregaron en dichos canvas.

Una vez estando funcionando el demo se configuraron los valores de algunas de las variables. Por ejemplo, se modificaron los valores de la velocidad delos enemigos, la salud y la cantidad de recursos que este genera, se insertaron las animaciones correspondientes al impacto de los proyectiles y al efecto de muerte de los enemigos, los valores para la velocidad de la cámara, rango de visión, como cada oleada enemiga es distinta que la otra se desarrolló la lógica de cada una en una clase serializable, en la que se definieron los principales componentes para cada oleada, tales como el *prefab* de enemigos que se utilizara en cada una, la frecuencia con que se visualizarán los enemigos y la cantidad de enemigos que conformarán la oleada. Una vez terminado el demo se necesitaba comprobar que todo lo estructurado e implementado cumplía con los requisitos de rendimiento para el correcto funcionamiento de los videojuegos.

3.4 Pruebas de rendimiento

Las pruebas de rendimiento se realizan para determinar cómo se comporta un sistema en términos de velocidad, estabilidad y escalabilidad cuando se somete a una carga de trabajo típica o incluso a cargas extremas(38). El objetivo principal de las pruebas de rendimiento es identificar cualquier cuello de botella, debilidad o deficiencia en el rendimiento de un sistema antes de que sea implementado en un entorno de producción o utilizado por un gran número de usuarios. Estas pruebas pueden revelar problemas como tiempos de respuesta lentos, degradación del rendimiento con la carga, errores de software, problemas de configuración del servidor o problemas de capacidad.

Para que un videojuego sea interpretado en tiempo real, es necesario que los FPS (Fotogramas por segundo) oscilen dentro de un rango de 30 a 60 FPS. Esto se debe a que el ojo humano tiene ciertas limitaciones perceptivas. Aproximadamente, a partir de los 30 FPS, el ojo humano es capaz de percibir el movimiento como continuo y fluido, sin notar grandes interrupciones entre los cuadros. Por otro lado, alrededor de los 60 FPS, la mayoría de las personas no pueden distinguir una mejora significativa en la fluidez del juego, debido a que la transición entre cuadros es lo suficientemente rápida como para que parezca un movimiento continuo(39). Algunos de los procesos fundamentales que se evidencian en el ciclo de actualización de un videojuego son:

Scripts: Se utilizan para controlar el comportamiento de los elementos del juego.

Rendering: Encargado de generar imágenes o secuencias de imágenes a partir de datos y modelos tridimensionales.

Physics: Es el encargado de realizar la simulación e implementación de comportamientos físicos realistas e interacciones dentro del mundo virtual del juego.

Mediante la herramienta *Unity Profiler*, se realizaron las pruebas de rendimiento, específicamente pruebas de carga al proceso de *scripts* en el demo anteriormente descrito, con el objetivo de analizar la estabilidad, desempeño y consumo de CPU. En una primera ejecución se probó sin elementos gráficos y sin interacción externa por parte del usuario, luego con elementos gráficos y sin interacción del usuario y por último con elementos gráficos y con interacción por parte del usuario, ver anexo 5. La computadora donde se llevaron a cabo las pruebas tenía las siguientes propiedades: procesador Intel Core i5 8265U a 1.8 GHz, 12 Gb de memoria RAM y sistema operativo Windows 11 de 64 bits. En las siguientes figuras se observan los tiempos en (ms) que consumen los *Scripts* y en general todos los elementos del demo en cada prueba realizada.

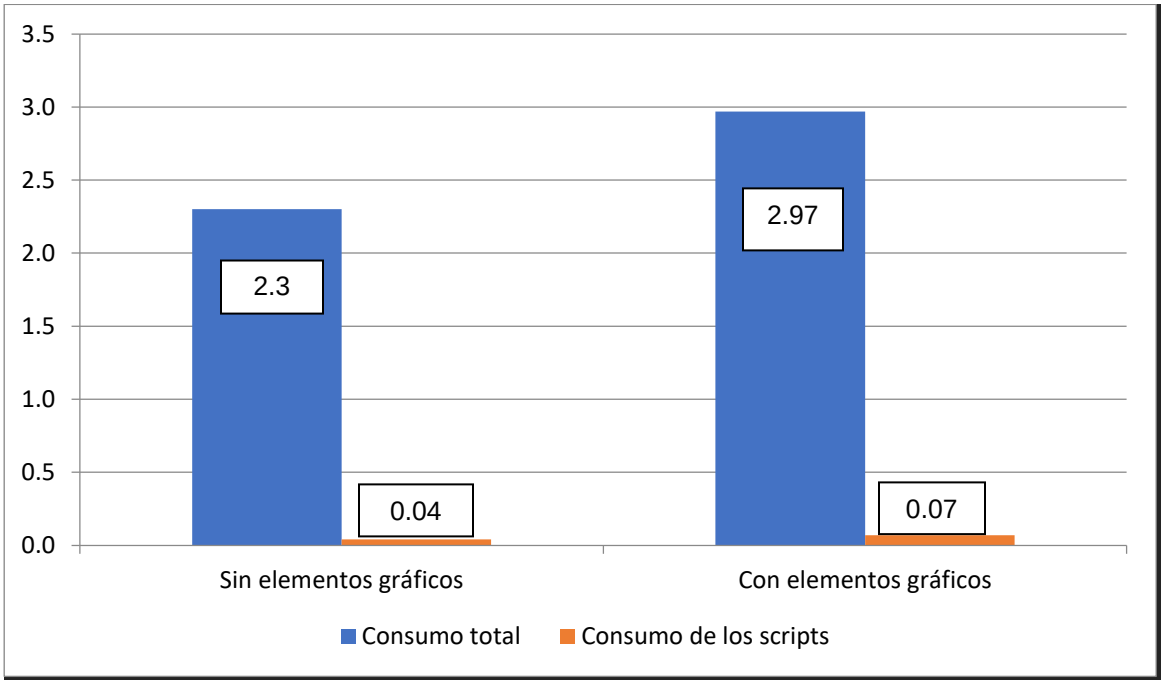

Figura 14. Resultados de las pruebas de rendimiento sin la interacción del usuario. Elaboración propia.

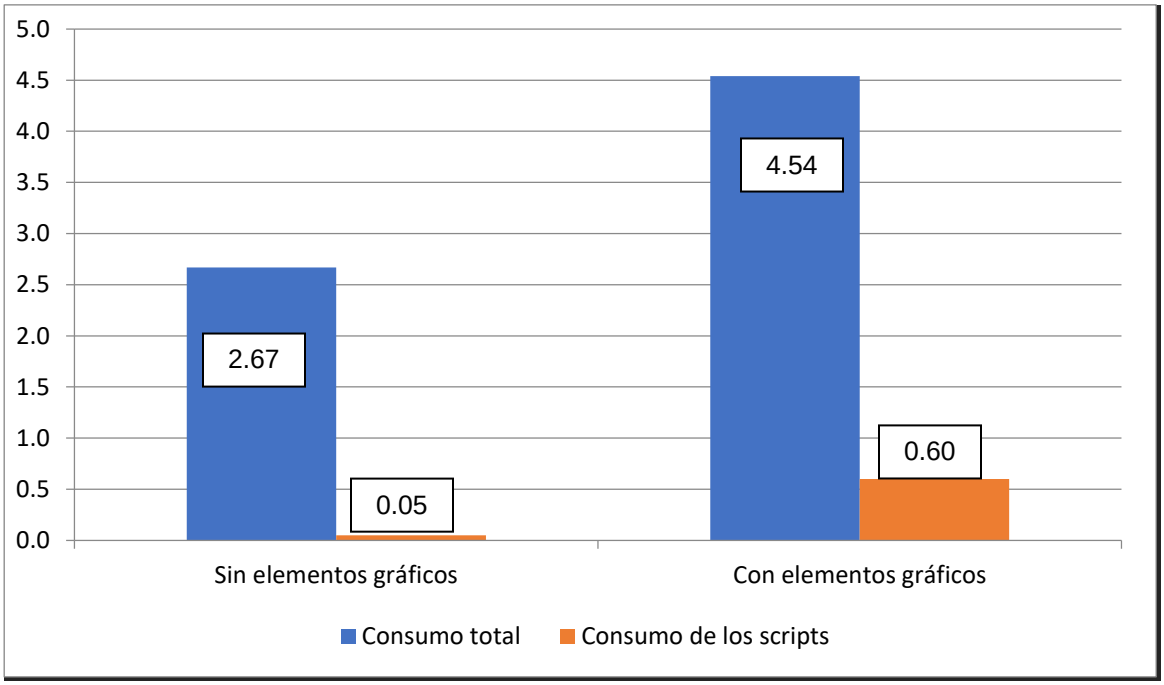

Figura 15. Resultados de las pruebas de rendimiento con la interacción del usuario. Elaboración propia.

Según los resultados obtenidos en las pruebas realizadas, se ha observado que el proceso de los Scripts representa una parte significativa, pero aún inferior al 40% del tiempo total necesario para procesar un cuadro en el juego. Al considerar el conjunto completo de procesos involucrados, se ha determinado que el tiempo de respuesta se encuentra en el rango de tiempo real, que abarca desde 33 ms (30 FPS) hasta 16 ms (60 FPS) (40). No obstante, es fundamental destacar la importancia de realizar un enfoque en la optimización del paquete para enriquecer el rendimiento de los videojuegos desarrollados con etas mecánicas.

Conclusiones parciales

- Al completar las iteraciones, se ha reconocido la importancia de organizar el desarrollo por tareas, lográndose un avance significativo y un orden en el proceso de desarrollo de la solución. La adecuada asignación del tiempo a cada iteración permitió cumplir con los objetivos establecidos.
- Las pruebas unitarias y de aceptación, garantizaron que el código funcionara correctamente. Estas pruebas fueron fundamentales para asegurar la calidad y la funcionalidad del código implementado.
- Finalmente, la demostración de uso del paquete proporcionó una validación tangible de la solución desarrollada y permitió obtener la aprobación del cliente.

CONCLUSIONES FINALES

Luego de culminado el trabajo se llegaron a las siguientes conclusiones:

- El análisis de la bibliografía permitió caracterizar los juegos de estrategia de tipo Tower Defender, identificando como mecánicas específicas a implementar: selección, locomoción, tienda, comportamiento del proyectil, comportamiento de las torres, vida general y cámara.
- Las características de los paquetes disponibles de Unity para este tipo de videojuegos no reúnen las características necesarias para ser utilizados en el centro VERTEX, con lo cual se confirma la problemática abordada en el presente trabajo.
- Los artefactos generados por la metodología empleada, han facilitado la comprensión de las funcionalidades, elementos y guía de trabajo para los desarrolladores a lo largo de todo el ciclo de desarrollo de la solución. La planificación llevada a cabo permitió asignar el tiempo necesario para cada historia de usuario y obtener una aplicación funcional.
- Las pruebas unitarias y de aceptación garantizaron que el código funcionara correctamente y permitió que el cliente obtuviera resultados tangibles desde las primeras etapas del desarrollo de la solución. Las pruebas demostraron el cumplimiento de las características de reusabilidad, flexibilidad y escalabilidad de la solución, lográndose alcanzar el objetivo de la investigación.

RECOMENDACIONES

Para asegurar una continua evolución y desarrollo de la solución propuesta, se recomienda lo siguiente:

.

- Agregar nuevas funcionalidades que puedan ser de gran utilidad como la implementación de una inteligencia artificial para los enemigos.
- Desarrollar un editor de niveles para crear y personalizar los niveles que conformarán el videojuego.
- Agregar las funcionalidades necesarias para el modo multijugador.

REFERENCIAS BIBLIOGRÁFICAS

1. GÓMEZ DEL CASTILLO SEGURADO, María Teresa. Videojuegos y transmisión de valores. *Revista Iberoamericana de Educación*.

2. *53701409.pdf*. Online. [Accessed 27 June 2023]. Available from: https://www.redalyc.org/pdf/537/53701409.pdf

3. *53701409.pdf*. Online. [Accessed 27 June 2023]. Available from: https://www.redalyc.org/pdf/537/53701409.pdf

4. Géneros de videojuegos | Wikijuegos | Fandom. Online. [Accessed 27 September 2023]. Available from: https://videojuegos.fandom.com/es/wiki/G%C3%A9neros_de_videojuegos

5. Videojuegos de plataformas | ¿Qué significa Videojuegos de plataformas? Online. [Accessed 27 September 2023]. Available from: https://www.geekno.com/glosario/videojuegos-de-plataformas

6. Videojuegos de disparos: tipos y características - Clebert. Online. 4 December 2022.

7. *08 LOS VIDEOJUEGOS.pdf*. Online. [Accessed 27 June 2023]. Available from: http://ardilladigital.com/DOCUMENTOS/TECNOLOGIA%20EDUCATIVA/TICs/T8%20VIDEOJUEGOS/08%20LOS%20VIDEOJUEGOS.pdf

8. *memoria firmada. FIDALGO RODRIGUEZ, DIEGO.pdf*. Online. [Accessed 27 June 2023]. Available from: https://gredos.usal.es/bitstream/handle/10366/149951/memoria%20firmada.%20FIDALGO%20RODRIGUEZ%2c%20DIEGO.pdf?sequence=1&isAllowed=y

9. SÁEZ SORO, Emilio. ¿Qué hacemos en los videojuegos? Un análisis de las mecánicas de juego como núcleo de la actividad de los jugadores.

10. SCHOOL, Tokio. Mecánicas de juego más habituales en los videojuegos. *Tokio School*. Online. 21 January 2020.

11. Kingdom Rush Frontiers | Programas descargables Nintendo Switch | Juegos | Nintendo. Online. [Accessed 22 October 2023]. Available from: https://www.nintendo.es/Juegos/Programas-descargables-Nintendo-Switch/Kingdom-Rush-Frontiers-1720963.html

12. RECALDE, Herny and AGUAS, Luis Fernando. Diseño de un videojuego aplicando una arquitectura en capas basada en el framework de unity.

13. Kingdom Rush. Online. [Accessed 24 November 2023]. Available from: https://www.kingdomrush.com/

14. MANRUBIA PEREIRA, Ana M[a]. El proceso productivo del videojuego: fases de producción.

15. Plantas contra Zombis | Wiki Plants vs. Zombies | Fandom. Online. [Accessed 24 November 2023]. Available from: https://plantsvszombies.fandom.com/es/wiki/Plantas_contra_Zombis

16. Vista de Diseño de un videojuego aplicando una arquitectura en capas basada en el framework de unity. Online. [Accessed 3 October 2023]. Available from: https://nexoscientificos.vidanueva.edu.ec/index.php/ojs/article/view/59/206

17. MOYA, Ignacio González. DESARROLLO DE UNA HERRAMIENTA DE ENSEÑANZA CON UNITY. .

18. Tower Defense Toolkit 4 (TDTK-4) | Game Toolkits | Unity Asset Store. Online. [Accessed 9 November 2023].

19. Tower Defense Starter Kit | Game Toolkits | Unity Asset Store. Online. [Accessed 9 November 2023].

20. MONTERO, Bryan Molina, CEVALLOS, Harry Vite and CUESTA, Jefferson Dávila. Agile methodologies against traditional methods in the software development process.

21. FUENTES, Lidia and VALLECILLO, Antonio. Una Introducción a los Perfiles UML. . 1 January 2004.

22. SÁNCHEZ, Antonio Barreto. Ingeniero en Ciencias Informáticas. .

23. ANANDMEG. What is Visual Studio? Online. 5 May 2023. [Accessed 27 June 2023]. Available from: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ideLearn.

24. C#. Online. [Accessed 27 June 2023]. Available from: https://desarrolloweb.com/home/c#track210

25. HERNÁNDEZ, Ing Yenisel Molina, DIHIGO, DrC Ailec Granda and CINTRA, Alionuska Velázquez. Los requisitos no funcionales de software. Una estrategia para su desarrollo en el Centro de Informática Médica. . 2019. Vol. 13, no. 2.

26. LETELIER, Patricio and LETELIER, Patricio. Métodologías ágiles para el desarrollo de software: eXtreme Programming (XP).

27. Historias de Usuario, Escritura, Definición, Contexto y Ejemplos. *SCRUM MÉXICO*. Online. 2 August 2018.

28. GUERRERO, Carlos A., SUÁREZ, Johanna M. and GUTIÉRREZ, Luz E. Patrones de Diseño GOF.

29. ISO 25010. Online. [Accessed 24 November 2023]. Available from: https://iso25000.com/index.php/normas-iso-25000/iso-25010

30. TABARES, Ricardo Botero. Patrones Grasp y Anti-Patrones: un Enfoque Orientado a Objetos desde Lógica de Programación.

31. *133122679013.pdf*. Online. [Accessed 17 September 2023]. Available from: https://www.redalyc.org/pdf/1331/133122679013.pdf

32. CASAS, Sandra I. and REINAGA, Héctor H. ASPECTOS TEMPRANOS: UN ENFOQUE BASADO EN TARJETAS CRC.

33. JOSKOWICZ, José. Reglas y Prácticas en eXtreme Programming. .

34. METODOLOGÍA HÍBRIDA DE DESARROLLO DE SOFTWARE COMBINANDO XP Y SCRUM | Mikarimin. Revista Científica Multidisciplinaria. Online. [Accessed 8 October 2023]. Available from: https://revista.uniandes.edu.ec/ojs/index.php/mikarimin/article/view/1233

35. ARIAS, Santiago Viteri, SORIA, Tatiana Mayorga, MOYA, Patricio Navas and PALMA, Patricio Molina. Control de calidad del software mediante pruebas automatizadas de integración y pruebas unitarias. *Ciencia Digital*. Online. 1 July 2018.

36. View of Software quality testing process analysis. Online. [Accessed 21 October 2023]. Available from: https://revistas.ucc.edu.co/index.php/in/article/view/1482/1724

37. PRESSMAN, Roger S. *Software engineering: a practitioner's approach*. . 7th ed. New York : McGraw-Hill Higher Education, 2010. ISBN 978-0-07-337597-7. QA76.758 .P75 2010

38. GIL-VERA, Victor Daniel and SEGURO-GALLEGO, Cristian. Machine learning aplicado al análisis del rendimiento de desarrollos de software. *Revista Politécnica*. Online. 29 April 2022.

39. *TD_08697_16.pdf*. Online. [Accessed 22 October 2023]. Available from: https://repositorio.uci.cu/jspui/bitstream/123456789/7864/1/TD_08697_16.pdf

40. *T-112662 López - Villamar.pdf*. Online. [Accessed 22 October 2023]. Available from: https://www.dspace.espol.edu.ec/bitstream/123456789/56334/1/T-112662%20L%c3%b3pez%20-%20Villamar.pdf

ANEXOS

ANEXO 1. HISTORIAS DE USUARIO

Tabla 21. HU Insertar script de enemigos.

Historia de Usuario	
No 2. Insertar *script* de enemigos.	
Prioridad: Alta	Nivel de complejidad: medio
Estimación: 6 días	Iteración asignada: 1
Descripción: El Usuario tendrá la posibilidad de asignarle este *script* a un prefabricado de enemigos y de esta forma crear los enemigos del videojuego.	
Observaciones:	

Tabla 22. HU Configurar script de enemigos.

Historia de Usuario	
No 3. Configurar *script* de enemigos.	
Prioridad: Alta	Nivel de complejidad: medio
Estimación: 5 días	Iteración asignada: 1
Descripción: El Usuario tendrá la posibilidad de modificar las características de los enemigos como son: vida, recursos, entre otras.	
Observaciones:	

Tabla 23. HU Insertar mecánica de locomoción.

Historia de Usuario	
No 4. Insertar mecánica de locomoción.	
Prioridad: Alta	Nivel de complejidad: medio
Estimación: 6 días	Iteración asignada: 1
Descripción: El Usuario tendrá la posibilidad de asignarle un *script* al prefabricado de enemigos para definir hacia qué dirección debe moverse.	

Tabla 24. HU Modificar mecánica de locomoción.

Historia de Usuario	
No 5. Modificar mecánica de locomoción.	
Prioridad: Alta	Nivel de complejidad: medio
Estimación: 3 días	Iteración asignada: 1
Descripción: El Usuario tendrá la posibilidad de modificar los atributos que se utilizan para realizar el movimiento.	

Tabla 25. HU Insertar sistema de generación de oleadas.

Historia de Usuario
No 6. Insertar el sistema de generación de oleadas.

Prioridad: media	Nivel de complejidad: medio
Estimación: 4 días	Iteración asignada: 1
Descripción: Debe permitirle al usuario tomar el *script* de las oleadas y de los enemigos y asignárselo a un objeto en la escena para gestionar el funcionamiento de las oleadas enemigas.	
Observaciones:	

Tabla 26. HU Modificar mecánica de Comportamiento de las Torres.

Historia de Usuario	
No 8. Modificar mecánica de Comportamiento de las Torres.	
Prioridad: Alta	Nivel de complejidad: alto
Estimación: 10 días	Iteración asignada: 2
Descripción: El Usuario tendrá la posibilidad de modificar los atributos configurables característicos de las torres.	
Observaciones:	

Tabla 27. HU Modificar mecánica de Cámara.

Historia de Usuario	
No 10. Modificar mecánica de Cámara.	
Prioridad: Alta	Nivel de complejidad: alto

Estimación: 10 días	Iteración asignada: 2
Descripción: El Usuario tendrá la posibilidad de modificar los atributos configurables propios de la cámara para variar su velocidad de desplazamiento, altura máxima entre otros.	
Observaciones:	

Tabla 28. HU Insertar mecánica Comportamiento del proyectil.

Historia de Usuario	
No 11. Insertar mecánica Comportamiento del proyectil.	
Prioridad: Alta	Nivel de complejidad: medio
Estimación: 10 días	Iteración asignada: 2
Descripción: El Usuario tendrá la posibilidad de asignarle un *script* y de esta forma crear los proyectiles.	
Observaciones:	

Tabla 29. HU Configurar mecánica Comportamiento del proyectil.

Historia de Usuario	
No 12. Configurar mecánica Comportamiento del proyectil.	
Prioridad: Alta	Nivel de complejidad: medio
Estimación: 10 días	Iteración asignada: 2
Descripción: El Usuario tendrá la posibilidad	

de modificar los atributos configurables propios de los proyectiles como son: rango, impacto, entre otros.
Observaciones:

Tabla 30. HU Insertar mecánica de Vida General.

Historia de Usuario	
No 13. Insertar mecánica de Vida General.	
Prioridad: media	Nivel de complejidad: medio
Estimación: 2 días	Iteración asignada: 3
Descripción: El usuario tendrá la posibilidad de asignarle un script a un objeto en la escena y de esta forma visualizar la vida general del juego.	
Observaciones:	

Tabla 31. HU Configurar mecánica de Vida General.

Historia de Usuario	
No 14. Configurar mecánica de Vida General.	
Prioridad: media	Nivel de complejidad: medio
Estimación: 2 días	Iteración asignada: 3
Descripción: El usuario tendrá la posibilidad de modificar los atributos propios de la vida	

del juego.
Observaciones:

Tabla 32. HU Visualizar estadísticas.

Historia de Usuario	
No 15. Visualizar estadísticas.	
Prioridad: baja	Nivel de complejidad: baja
Estimación: 3 días	Iteración asignada: 3
Descripción: El usuario debe asignarle el *script* de estadísticas a un objeto en la escena y de esta forma podrá visualizar en tiempo real el estado del juego.	
Observaciones:	

Tabla 33. HU Establecer condición de pérdida del juego.

Historia de Usuario	
No 16. Establecer condición de pérdida del juego.	
Prioridad: media	Nivel de complejidad: baja
Estimación: 3 días	Iteración asignada: 3
Descripción: El usuario tendrá opción de asignarle el *script* de pérdida a un elemento visual y de esta forma poder mostrar el final	

del juego.
Observaciones:

Tabla 34. HU Establecer lógica del juego

Historia de Usuario	
No 17. Establecer lógica del juego.	
Prioridad: baja	Nivel de complejidad: baja
Estimación: 7 días	Iteración asignada: 3
Descripción: El usuario debe asignarle el *script* de la lógica del juego a un objeto en la escena y de esta forma poder manipular el límite de las vidas y la condición de pérdida.	
Observaciones:	

Tabla 35. HU Insertar mecánica de selección.

Historia de Usuario	
No 18. Insertar mecánica de selección.	
Prioridad: media	Nivel de complejidad: media
Estimación: 10 días	Iteración asignada: 4
Descripción: El usuario tendrá la posibilidad de asignarle un *script* a un objeto en la escena y de esta forma seleccionar los elementos definidos.	

Observaciones:

Tabla 36. HU Visualizar estructuras seleccionadas.

Historia de Usuario	
No 19. Visualizar estructuras selecciona-das.	
Prioridad: media	Nivel de complejidad: baja
Estimación: 7 días	Iteración asignada: 4
Descripción: Brinda la posibilidad de observar las opciones que brindan las estructuras seleccionadas.	
Observaciones: Se debe tener implementado el script de selección.	

Tabla 37. HU Insertar mecánica Tienda.

Historia de Usuario	
No 20. Insertar mecánica Tienda.	
Prioridad: media	Nivel de complejidad: baja
Estimación: 8 días	Iteración asignada: 4
Descripción: Brinda al usuario la posibilidad de crear una tienda donde se encuentran disponibles los objetos para comprar y vender.	

Observaciones:

Tabla 38. HU Modificar mecánica Tienda.

Historia de Usuario	
No 21. Modificar mecánica Tienda.	
Prioridad: media	Nivel de complejidad: baja
Estimación: 8 días	Iteración asignada: 4
Descripción: Brinda al usuario la posibilidad de realizar modificaciones en los elementos que se encuentran en la tienda.	
Observaciones:	

ANEXO 2. TAREAS DE INGENIERÍA

Tabla 39. Tarea 3 Iteración 1

Tarea	
Número de la tarea: 3	**Número de HU: 4**
Nombre de la tarea: Implementación de un *script* para la locomoción de los enemigos.	
Tipo de tarea: Implementación	**Estimación:** 6 días
Descripción: Se definen atributos, se inicializan las referencias necesarias, se actualiza el movimiento del enemigo y se obtiene el siguiente punto de ruta para el desplazamiento, este script se debe agregar a un *pre-fab* de enemigos junto al script de enemigos.	

Tabla 40. Tarea 4 Iteración 1

Tarea	
Número de la tarea: 4	**Número de HU: 6**
Nombre de la tarea: Implementación de la clase serializable *Wave*.	
Tipo de tarea: Implementación	**Estimación:** 1 día
Descripción: Se declaran e inicializan los atributos necesarios para definir las oleadas enemigas.	

Tabla 41. Tarea 5 Iteración 1

Tarea	
Número de la tarea: 5	**Número de HU: 6**
Nombre de la tarea: Implementación del *script WaveSpawner*	
Tipo de tarea: Implementación	**Estimación:** 4 día
Descripción: Se implementa la lógica para el sistema de generación de oleadas de enemigos, el punto de aparición enemiga, el tiempo entre oleadas, propiciándole al usuario su fácil manipulación desde el inspector de Unity.	

Tabla 42. Tarea 2 Iteración 3

Tarea	
Número de la tarea: 2	**Número de HU: 13**
Nombre de la tarea: Implementación del *script Lives*	
Tipo de tarea: Implementación	**Estimación:** 2 días
Descripción: Se implementa un método para mantener actualizado el contador de vidas en la interfaz de usuario.	

Tabla 43. Tarea 4 Iteración 3

Tarea	
Número de la tarea: 4	**Número de HU: 16**
Nombre de la tarea: Implementación de la clase *GameOver*	
Tipo de tarea: Implementación.	**Estimación:** 3 días
Descripción: Proporciona la funcionalidad de reiniciar la partida o volver al menú principal después de perder. También muestra al jugador la cantidad de rondas que ha sobrevivido antes de perder la partida.	

Tabla 44. Tarea 5 Iteración 3

Tarea	
Número de la tarea: 5	**Número de HU: 17**
Nombre de la tarea: Implementación del script *GameManager.*	
Tipo de tarea: Implementación	**Estimación:** 6 días

Descripción: Permite controlar el flujo del juego y activar la pantalla de pérdida cuando se cumplan las condiciones establecidas.

Tabla 45. Tarea 3 Iteración 4

Tarea	
Número de la tarea: 3	**Número de HU: 20**
Nombre de la tarea: Implementación del *script Shop* para el funcionamiento de la tienda.	
Tipo de tarea: Implementación.	**Estimación:** 7 días
Descripción: Se implementan los mecanismos necesarios para agregar este script a un objeto, y obtener como resultado una tienda donde se encuentran las estructuras(torres) del juego, también se declaran e implementan los atributos y métodos necesarios para que sea fácil su configuración haciendo uso de prefabricados en el inspector de Unity.	

Tabla 46. Tarea 4 Iteración 4

Tarea	
Número de la tarea: 4	**Número de HU: 19**
Nombre de la tarea: Implementación del *script MoneyUI.*	
Tipo de tarea: Implementación	**Estimación:** 1 días
Descripción: Se debe incorporar a la tienda un objeto que se encargue de mantener actualizada la cantidad de dinero que se tiene en cierto momento.	

ANEXO 3. CASOS DE PRUEBA DE ACEPTACIÓN

Tabla 47. Caso de Prueba de Aceptación P2HU2

Caso de Prueba de Aceptación	
Código : P2HU2	**Número de HU:** 2
Nombre: Insertar *script* de enemigos	
Descripción: Se debe probar que se pueda insertar fácilmente el *script Enemy* en un objeto de la escena.	
Condiciones de ejecución: Debe existir un *prefab* de enemigos en la escena al cual se le asignara el *script* de enemigos.	
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado de enemigos en la ventana de escena o jerarquía. Arrastra el *script* de enemigos hacia el editor de Unity donde se encuentra abierto el prefab.	

Resultado esperado: Correcta inserción del *script Enemy.*
Evaluación de la prueba: Resultado satisfactorio.

Tabla 48. Caso de Prueba de Aceptación P3HU3.

Caso de Prueba de Aceptación	
Código : P3HU3	**Número de HU:** 3
Nombre: Configurar *script* de enemigos	
Descripción: Se debe probar se puedan modificar los valores de velocidad, salud y recursos de cada enemigo desde el inspector de Unity.	
Condiciones de ejecución: Debe existir un *prefab* de enemigos en la escena al cual se le asignara el *script* de enemigos.	
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado de enemigos en la ventana de escena o jerarquía. Arrastra el *script* de enemigos hacia el editor de Unity donde se encuentra abierto el prefab. Modifica los valores de las variables *Start Speed, Health, Worth,* y también puede añadir un efecto de muerte insertando un prefabricado con el efecto en el apartado *Death Effect.*	
Resultado esperado: Correcta modificación de los valores de las variables que conforman los enemigos.	
Evaluación de la prueba: Resultado no satisfactorio.	

Tabla 49. Caso de Prueba de Aceptación P4HU4.

Caso de Prueba de Aceptación	
Código : P4HU4	**Número de HU:** 4
Nombre: Insertar mecánica de locomoción.	
Descripción: Se debe probar que al añadirle el *script EnemyMovement* a un enemigo, este sea capaz de comenzar su desplazamiento hacia su lugar de destino.	
Condiciones de ejecución: Debe existir un *prefab* de enemigos en la escena con el *script* de enemigos ya incorporado.	
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado de enemigos en la ventana de escena o jerarquía. Arrastra el *script* de enemigos hacia el editor de Unity donde se encuentra abierto el *prefab.* Arrastra el *script EnemyMovement* hacia el editor de Unity donde se encuentra el *script* de enemigos.	
Resultado esperado: Correcto funcionamiento del desplazamiento del enemigo de un punto a otro.	
Evaluación de la prueba: Resultado no satisfactorio.	

Tabla 50. Caso de Prueba de Aceptación P5HU5.

Caso de Prueba de Aceptación	
Código : P5HU5	**Número de HU:** 5
Nombre: Modificar mecánica de locomoción.	
Descripción: Se debe probar que se pueda añadir el *script EnemyMovement* en cualquier objeto de la escena.	
Condiciones de ejecución: Debe existir un *prefab* de enemigos en la escena con el *script* de enemigos ya incorporado.	
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado de enemigos en la ventana de escena o jerarquía. Arrastra el *script* de enemigos hacia el editor de Unity donde se encuentra abierto el *prefab.* Arrastra el *script EnemyMovement* hacia el editor de Unity donde se encuentra el *script* de enemigos. Modifica los atributos configurables que conforman el *script*.	
Resultado esperado: Correcto funcionamiento del desplazamiento del enemigo de un punto a otro.	
Evaluación de la prueba: Resultado satisfactorio.	

Tabla 51. Caso de Prueba de Aceptación P6HU6.

Caso de Prueba de Aceptación	
Código : P6HU6	**Número de HU:** 6
Nombre: Implementación del sistema de generación de oleadas.	
Descripción: Se debe probar que se puedan generar correctamente las oleadas y que se pueda realizar la correcta modificación de las variables que las conforman.	
Condiciones de ejecución: Se debe tener un objeto vacío en la escena para poder asignarle el *script.* Lo más recomendable es que sea el objeto que se encargue de manipular la lógica del juego.	
Entrada/Pasos de ejecución: El usuario selecciona el objeto al que le va a asignar el *script.* Arrastra el *script WaveSpawner* hacia el editor de Unity donde se encuentra abierto el objeto. Añade la cantidad de oleadas deseadas y modifica los valores configurables de estas.	
Resultado esperado: Correcta generación de oleadas.	
Evaluación de la prueba: Resultado satisfactorio.	

Tabla 52. Caso de Prueba de Aceptación P8HU8.

Caso de Prueba de Aceptación	
Código : P8HU8	**Número de HU:** 8
Nombre: Modificar mecánica Comportamiento de las Torres.	

Descripción: Se debe probar que se puedan insertar torres en el terreno de juego.
Condiciones de ejecución: Se debe tener un prefabricado de torres en la escena para llevar a cabo la asignación del *script*.
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado de torres que se encuentra en la escena. Arrastra el *script* Torres hacia la ventana del inspector de Unity donde se encuentra abierto el prefabricado. Modifica los atributos configurables que proporciona el *script.*
Resultado esperado: Correcta modificación de los valores configurables en el inspector de Unity.
Evaluación de la prueba: Resultado satisfactorio.

Tabla 53. Tabla 52. Caso de Prueba de Aceptación P10HU10.

Caso de Prueba de Aceptación	
Código : P10HU10	**Número de HU:** 10
Nombre: modificar mecánica de Cámara.	
Descripción: Se debe probar que se puedan modificar correctamente los atributos configurables del *script Camera* en el juego.	
Condiciones de ejecución: Se debe asegurar que no exista otra cámara en el juego.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto cámara Arrastra el *script Camera* hacia el inspector de Unity donde se encuentra abierto el objeto cámara. Modifica los valores configurables que aparecen tales como: velocidad de movimiento, distancia de enfoque, entre otros.	
Resultado esperado: Correcta modificación de los elementos configurables en el inspector de Unity.	
Evaluación de la prueba: Resultado satisfactorio.	

Tabla 54. Caso de Prueba de Aceptación P11HU11.

Caso de Prueba de Aceptación	
Código : P11HU11	**Número de HU:** 11
Nombre: Insertar mecánica Comportamiento del proyectil.	
Descripción: Se debe probar que se puedan insertar los proyectiles correspondientes a cada torre.	
Condiciones de ejecución: Se debe tener un prefabricado de torres en la escena con el *script* de torres ya asignado y un prefabricado del proyectil a utilizar.	
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado del proyectil a utilizar.	

Arrastra el *script Bullet* hacia la ventana del inspector de Unity donde se encuentra abierto el prefabricado del proyectil. Selecciona el prefabricado de la torre a la cual se le realizará la acción de inserción del proyectil a utilizar. Se arrastra hacia la ventana del inspector de Unity el prefabricado del proyectil previamente modificado y se inserta en el apartado *Bullet Prefab* que aparecerá en el inspector dentro del prefab de la torre.
Resultado esperado: Correcta inserción del proyectil.
Evaluación de la prueba: Resultado satisfactorio.

Tabla 55. Caso de Prueba de Aceptación P12HU12.

Caso de Prueba de Aceptación	
Código : P12HU12	**Número de HU:** 12
Nombre: Modificar mecánica Comportamiento del proyectil.	
Descripción: Se debe probar que se pueda realizar la correcta modificación de los elementos configurables de dicha mecánica en el inspector de Unity.	
Condiciones de ejecución: Se debe tener un prefabricado de torres en la escena con el *script* de torres ya asignado y un prefabricado del proyectil a utilizar.	
Entrada/Pasos de ejecución: El usuario selecciona el prefabricado del proyectil a utilizar. Arrastra el *script Bullet* hacia la ventana del inspector de Unity donde se encuentra abierto el prefabricado del proyectil. Modifica los elementos configurables que aparecen una vez insertado el *script.*	
Resultado esperado: Correcta modificación de los elementos configurables.	
Evaluación de la prueba: Resultado satisfactorio.	

Tabla 56. Tabla 55. Caso de Prueba de Aceptación P14HU14.

Caso de Prueba de Aceptación	
Código : P14HU14	**Número de HU:** 14
Nombre: configurar mecánica de Vida General.	
Descripción: Se debe probar que se pueda manipular la vida general del juego.	
Condiciones de ejecución: Se debe tener en la escena un objeto para realizar la inserción.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script*, se recomienda que sea en un objeto que se encuentre en una interfaz de usuario. Arrastra el *script Lives* hacia el inspector de Unity donde se encuentra abierto el objeto a utilizar.	

Modifica los atributos configurables que facilita e *script*.
Resultado esperado: Correcta modificación de los atributos configurables.
Evaluación de la prueba: Resultado satisfactorio.

Tabla 57. Caso de Prueba de Aceptación P15HU16.

Caso de Prueba de Aceptación	
Código : P15HU16	**Número de HU:** 16
Nombre: Establecer condición de pérdida del juego.	
Descripción: Se debe probar que se pueda insertar un *script* que se encargue de detener el juego bajo cierta condición.	
Condiciones de ejecución: Se debe tener en la escena un objeto para realizar la inserción.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script*, se recomienda que sea en un objeto que se encuentre en una interfaz de usuario. Arrastra el *script GameOver* hacia el inspector de Unity donde se encuentra abierto el objeto a utilizar. Arrastra un archivo de tipo *text* hacia el apartado *Round Text* que aparecerá una vez se halla insertado el *script* en el objeto, dicho archivo debe contener lo necesario para mostrar la cantidad de oleadas sobrevividas.	
Resultado esperado: Correcto funcionamiento del script *GameOver* y del archivo *Round Text.*	
Evaluación de la prueba: Resultado Satisfactorio.	

Tabla 58. Caso de Prueba de Aceptación P16HU15.

Caso de Prueba de Aceptación	
Código : P16HU15	**Número de HU:** 15
Nombre: Visualizar Estadísticas.	
Descripción: Se debe probar que se pueda insertar un *script* que se encargue de manipular las estadísticas del juego, así como inicializar los valores de la vida del juego y los recursos.	
Condiciones de ejecución: Se debe tener en la escena un objeto para realizar la inserción, se recomienda que dicho objeto sea el *GameObject* que se encarga de manipular la lógica del juego, además se deben tener insertado y configurado el *script Lives* para poder manipular las vidas del juego.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script*. Arrastra el *script PlayerStats* hacia el inspector de Unity donde se encuentra abierto el objeto a utilizar. Configura los elementos modificables con los cuales debe comenzar el juego y se podrán observar durante toda la partida para que el usuario tenga una visión de los recursos y las vidas que tiene disponible en tiempo real.	

Resultado esperado: Correcto funcionamiento del *script PlayerStats* y correcta modificación de los elementos configurables.
Evaluación de la prueba: Resultado no satisfactorio.

Tabla 59. Caso de Prueba de Aceptación P17HU17.

Caso de Prueba de Aceptación	
Código : P17HU17	**Número de HU:** 17
Nombre: Establecer Lógica del juego.	
Descripción: Se debe probar que se pueda insertar un *script* que se encargue de manipular los elementos de fin de fin del juego, los elementos del menú y las opciones adicionales que se tienen.	
Condiciones de ejecución: Se debe tener en la escena un objeto para realizar la inserción, se recomienda que dicho objeto sea el *GameObject* que se encarga de manipular la lógica del juego, además debe encontrar insertado correctamente en un objeto el *script GameOver.*	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script*. Arrastra el *script GameManager* hacia el inspector de Unity donde se encuentra abierto el objeto a utilizar. Añade el objeto que contiene el *script GameOver* hacia el apartado *Game Over UI* que aparecerá una vez insertado el *script*.	
Resultado esperado: Se obtiene como resultado un menú de opciones al terminar el juego.	
Evaluación de la prueba: Resultado satisfactorio.	

Tabla 60. Caso de Prueba de Aceptación P12HU12

Caso de Prueba de Aceptación	
Código : P18HU18	**Número de HU:** 18
Nombre: Implementación de selección de puntos en el terreno con un clic.	
Descripción: Se debe probar que se puedan seleccionar puntos en el terreno para colocar las estructuras (Torres).	
Condiciones de ejecución: Se debe tener agregado en la escena el prefabricado del mapa del juego con los nodos que lo conforman.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script*. Arrastra el *script Node* hacia el inspector de Unity donde se encuentra abierto el objeto a utilizar. Define a conveniencia los colores que deben tener los puntos de construcción. En el apartado *Hover Color* define el color que se le va a dar a los puntos en los que se pueda realizar la construcción, en el apartado *Not Enough Money C* define el color que se le va a dar a los puntos cuando no se	

cuente con el dinero suficiente para realizar una construcción.
Resultado esperado: Se obtiene como resultado un recuadro de selección dibujado de un color.
Evaluación de la prueba: Resultado no satisfactorio.

Tabla 61. Caso de Prueba de Aceptación P19HU19

Caso de Prueba de Aceptación	
Código : P19HU19	**Número de HU:** 19
Nombre: Implementación del *script NodeUI.*	
Descripción: Se debe probar que cuando se seleccione una estructura (Torre) se visualicen y se ejecuten una serie de acciones que se pueden realizar sobre ella.	
Condiciones de ejecución: Se debe tener en la escena un objeto que contenga los botones para realizar las acciones de actualización y venta de la estructura, además se debe tener en dicho objeto una interfaz de usuario para mostrar dichos botones y cada botón debe tener asignado el *script Node* para poder acceder a la actualización o venta de la estructura.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script*. Arrastra el *script NodeUI* hacia el inspector de Unity donde se encuentra abierto el objeto a utilizar. En el apartado *UI* inserta la interfaz de usuario con la que cuenta el objeto y en el apartado *Upgrade Button* inserta el objeto donde contiene el botón de actualización.	
Resultado esperado: Se obtiene como resultado una vista sobre la que se puede realizar las opciones de actualización y venta de la torre seleccionada.	
Evaluación de la prueba: Resultado satisfactorio.	

Tabla 62. Caso de Prueba de Aceptación P20HU20

Caso de Prueba de Aceptación	
Código : P20HU20	**Número de HU:** 20
Nombre: Insertar mecánica Tienda.	
Descripción: Se debe probar que se pueda insertar un script que se encargue de dar funcionamiento a la tienda del juego.	
Condiciones de ejecución: Se debe tener en la escena un objeto que contenga la tienda ya diseñada, se recomienda que sea un objeto de interfaz de usuario, también se debe contar con los prefabricados de las torres que se van a utilizar con sus respectivos *scripts* insertados y configurados.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script Shop.* Arrastra el *script Shop* hacia la ventana del inspector de Unity donde se encuentra abierto el objeto de la	

tienda.
Resultado esperado: Se puede insertar correctamente en el campo la torre seleccionada.
Evaluación de la prueba: Resultado satisfactorio.

Tabla 63. Caso de Prueba de Aceptación P21HU21.

Caso de Prueba de Aceptación	
Código : P20HU20	**Número de HU:** 20
Nombre: Insertar y configurar mecánica Tienda.	
Descripción: Se debe probar que se puedan configurar a gusto del usuario los elementos que se encuentran en la tienda del juego.	
Condiciones de ejecución: Se debe tener en la escena un objeto que contenga la tienda ya diseñada, se recomienda que sea un objeto de interfaz de usuario, también se debe contar con los prefabricados de las torres que se van a utilizar con sus respectivos *scripts* insertados y configurados.	
Entrada/Pasos de ejecución: El usuario selecciona en la escena o jerarquía el objeto al que le realizará la inserción del *script Shop.* Arrastra el *script Shop* hacia la ventana del inspector de Unity donde se encuentra abierto el objeto de la tienda. Añade los prefabricados de las torres en el apartado *Prefab,* define un costo para la torre en el apartado *Cost,* en el apartado *Upgraded Prefab* define un prefabricado para que se actualice la torre cuando el usuario seleccione el botón "*Upgrade*" y define un costo de actualización.	
Resultado esperado: Se puede configurar correctamente la torre seleccionada.	
Evaluación de la prueba: Resultado no satisfactorio.	

ANEXO 4. CASOS DE PRUEBA UNITARIA

Las pruebas unitarias que a continuación se describen se decidió realizarlas pues las funcionalidades que ellas representan son esenciales para el correcto funcionamiento de un videojuego de tipo *Tower Defense*.

Caso de Prueba Unitaria

```
void ShootTest()
    {

        GameObject bulletGo = (GameObject)Instantiate(BulletPrefab, firePoint.position, firePoint.rotation);
        Bullet bullet = bulletGo.GetComponent<Bullet>();

        if (bullet != null)
        {
            //  para comprobar el funcionamiento del disparo
            Debug.Assert(bullet.target != null, "El proyectil no tiene un objetivo asignado.");
            Debug.Assert(bullet.target == target, "El objetivo del proyectil no coincide con el objetivo
esperado.");
        }
    }
```

Resultado

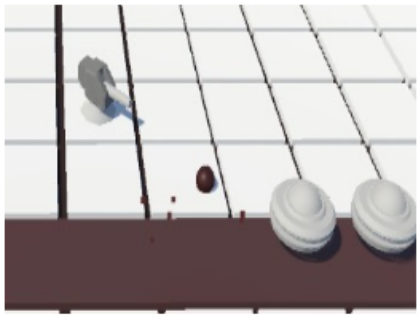

Figura 16. Caso de prueba unitaria ShootTest.

Al finalizar la implementación de esta prueba se detectó un defecto debido a que el proyectil no era capaz de seguir el objetivo asignado, dicho defecto fue resuelto antes de pasar a la siguiente prueba, como se aprecia en la figura anterior el proyectil ya es capaz de seguir el objetivo para realizar su eliminación.

Caso de Prueba Unitaria

```
void BuildTowerTest()
  {
    if (!buildManager.CanBuild)
    {
      Debug.Log("No se puede construir una torre en este momento.");
      return;
    }

    BuildTorres(buildManager.GetTorresToBuild());
  }
```

Resultado

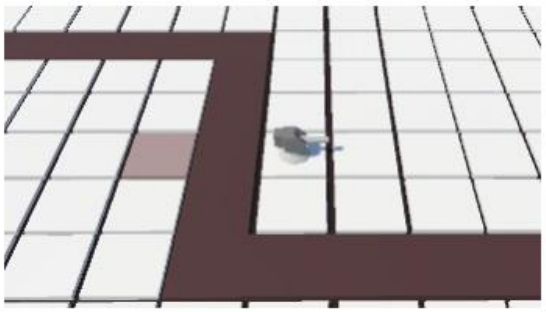

Figura 17. Caso de prueba unitaria BuildTowerTest

Al finalizar la implementación de esta prueba se detectó un defecto al observarse que siempre se podía realizar la inserción de la torre en el terreno, luego de realizar un análisis al código correspondiente a esta funcionalidad, se corrigió el error, como se muestra en la figura anterior el recuadro esta coloreado del color que se definió para mostrar que ya no se podía construir una torre en ese espacio.

Caso de Prueba Unitaria

```
void MovementTest()
  {
    Debug.Log("Posición inicial: " + transform.position);
    StartCoroutine(MoveEnemyForTest(2f));
    StartCoroutine(ShowFinalPositionAfterDelay(6f));
  }
  IEnumerator MoveEnemyForTest(float duration)
  {
    float elapsedTime = 0f;
    while (elapsedTime < duration)
    {
      Vector3 dir = target.position - transform.position;
      transform.Translate(enemy.speed * Time.deltaTime * dir.normalized, Space.World);
      elapsedTime += Time.deltaTime;
      yield return null;
    }
  }
  IEnumerator ShowFinalPositionAfterDelay(float delay)
  {
    yield return new WaitForSeconds(delay);
    Debug.Log("Posición final: " + transform.position);
  }
```

Resultado

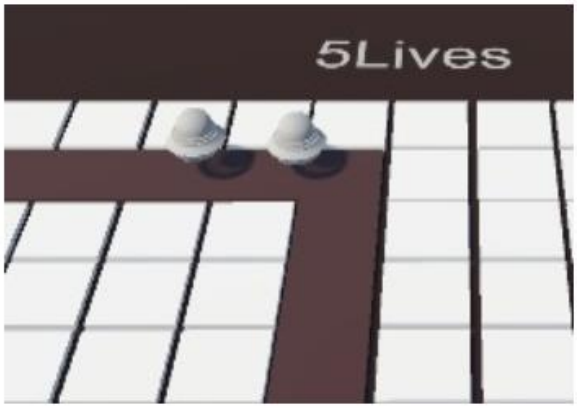

Figura 18. Caso de prueba unitaria MovementTest

La prueba unitaria que se aprecia en la figura anterior se encarga de verificar que los enemigos lleguen de un destino a otro de forma correcta, al final el desarrollo de la misma se llegó a la conclusión de que la funcionalidad está correctamente desarrollada pues no arrojo ningún error.

ANEXO 5. CASOS DE PRUEBA DE RENDIMIENTO

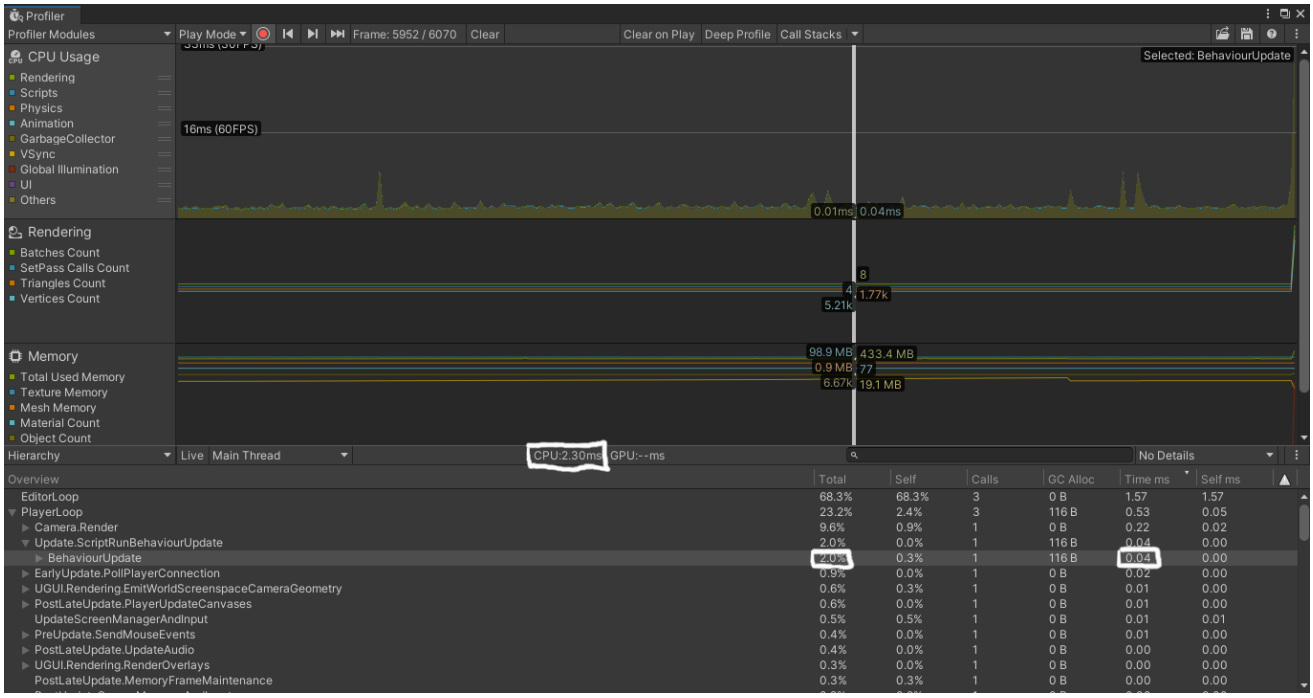

Figura 19. Prueba de Rendimiento sin interacción del usuario y sin elementos gráficos.

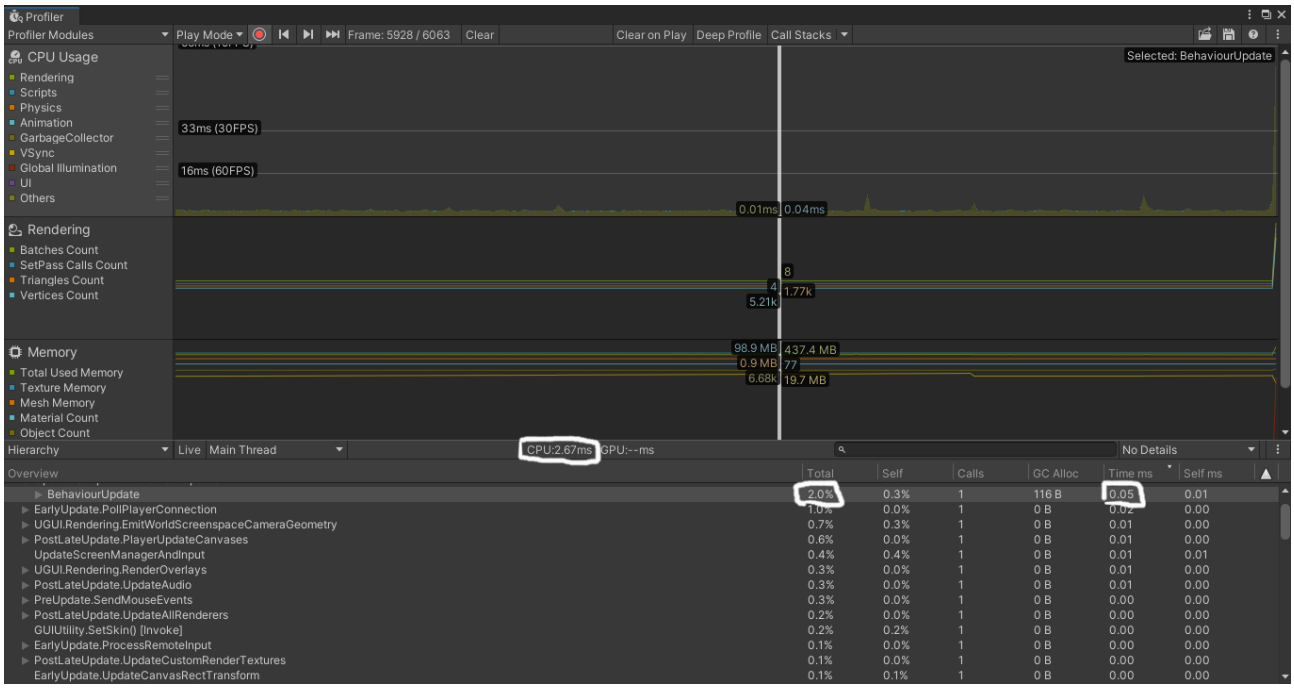

Figura 20. Prueba de Rendimiento con interacción del usuario y sin elementos gráficos.

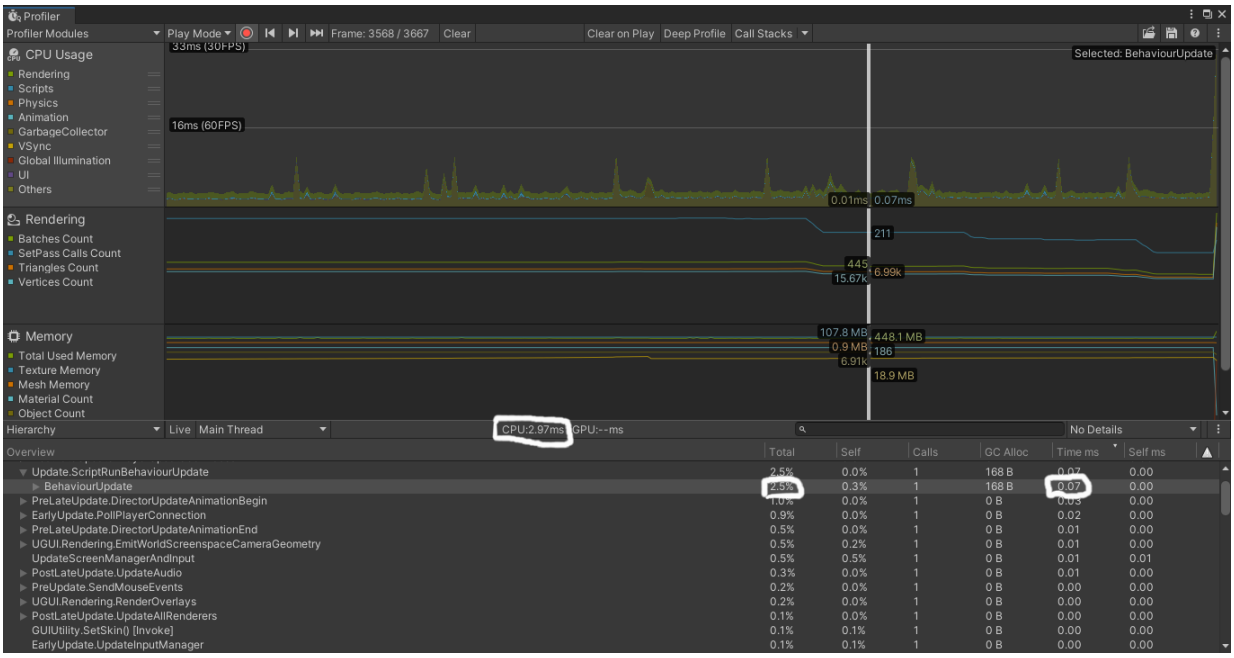

Figura 21. Prueba de Rendimiento sin interacción del usuario y con elementos gráficos.

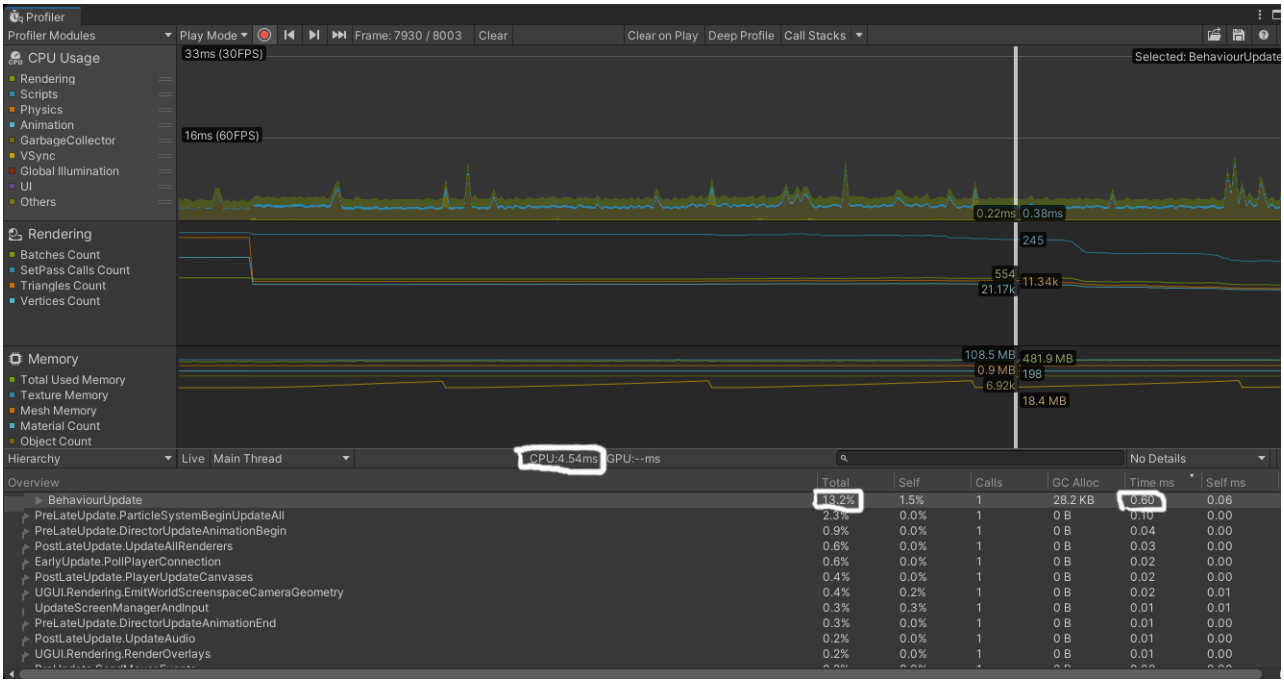

Figura 22. Prueba de Rendimiento con interacción del usuario y con elementos gráficos.

ANEXO 6. Tarjetas CRC

Tabla 64. Tarjeta CRC Bullet

Tarjeta CRC	
Clase: *Bullet*	
Responsabilidades	**Clases Relacionadas**
• Movimiento del proyectil • Impacto en el objetivo • Daño a los enemigos • Efectos visuales	• *MonoBehaviour* • *Enemy*

Tabla 65. Tarjeta CRC Lives

Tarjeta CRC	
Clase: *Lives*	
Responsabilidades	**Clases Relacionadas**
• Manipula la vida general del juego.	• *MonoBehaviour* • *PlayerStats* • *GameManager*

Tabla 66. Tarjeta CRC Enemy

Tarjeta CRC	
Clase: *Enemy*	
Responsabilidades	**Clases Relacionadas**
• Manipula los elementos esenciales de las tropas enemigas tales como: salud, velocidad. • Verifica que la unidad esté eliminada.	• *MonoBehaviour* • *PlayerStats* • *WaveSpawner* • *WayPoints*

Tabla 67. Tarjeta CRC BuildManager

Tarjeta CRC	
Clase: *BuildManager*	
Responsabilidades	**Clases Relacionadas**
• Gestiona la construcción de torres. • Permite la selección de nodos. • Muestra la interfaz de usuario del nodo	• *MonoBehaviour* • *PlayerStats* • *TurretBlueprint*

seleccionado.	• Torres • *NodeUI* • *Node* • *GameManager*

Tabla 68. Tarjeta CRC GameManager

Tarjeta CRC	
Clase: *GameManager*	
Responsabilidades	**Clases Relacionadas**
• Verifica si el juego ha terminado • Controla la entrada del jugador • Verifica las vidas del jugador • Finaliza el juego	• *MonoBehaviour* • *GameOver* • *BuildManager* • *PlayerStats*

Tabla 69. Tarjeta CRC GameOver

Tarjeta CRC	
Clase: *GameManager*	
Responsabilidades	**Clases Relacionadas**
• Actualiza la puntuación final • Reinicia el juego • Retorna al menú.	• *MonoBehaviour* • *GameManager* • *PlayerStats*

Tabla 70. Tarjeta CRC MoneyUI

Tarjeta CRC	
Clase: *MoneyUI*	
Responsabilidades	**Clases Relacionadas**
• Actualiza el texto del dinero.	• *MonoBehaviour* • *PlayerStats*

Tabla 71. Tarjeta CRC Node

Tarjeta CRC	
Clase: *Node*	
Responsabilidades	**Clases Relacionadas**
• Selección de nodos	• *MonoBehaviour*

• Mostrar opciones de los elementos señalados	• *BuildManager* • *NodeUI*

Tabla 72. Tarjeta CRC NodeUI

Tarjeta CRC	
Clase: *NodeUI*	
Responsabilidades	**Clases Relacionadas**
• Mostrar opciones de los elementos señalados • Mostrar opciones de construcción • Ocultar interfaz de usuario cuando no se necesita.	• *MonoBehaviour* • *BuildManager* • *NodeUI*

Tabla 73. Tarjeta CRC PlayerStats

Tarjeta CRC	
Clase: *PlayerStats*	
Responsabilidades	**Clases Relacionadas**
• Contar las vidas disponibles • Contar dinero disponible • Verificación de vidas restantes	• *MonoBehaviour* • *NodeUI* • *GameManager*

Tabla 74. Tarjeta CRC Torres

Tarjeta CRC	
Clase: *Torres*	
Responsabilidades	**Clases Relacionadas**
• Identificar y apuntar al enemigo más cercano • Disparar proyectiles de diferentes formas.	• *MonoBehaviour* • *Enemy* • *Bullet* • *BuildManager* • *PlayerStats*

Tabla 75. Tarjeta CRC TurretBlueprint

Tarjeta CRC	
Clase: *TurretBlueprint*	
Responsabilidades	**Clases Relacionadas**
• Almacena información sobre las torres • Realiza los cálculos para compra y venta de torres.	• *MonoBehaviour* • *BuildManager*

	• *NodeUI*

Tabla 76. Tarjeta CRC Wave

Tarjeta CRC	
Clase: *Wave*	
Responsabilidades	**Clases Relacionadas**
• Gestiona la cantidad de enemigos a generar en cada oleada • Gestiona la frecuencia en que se generan los enemigos en cada oleada	• *WaveSpawner* • *GameManager*

Tabla 77. Tarjeta CRC WaveSpawner

Tarjeta CRC	
Clase: *Wave*Spawner	
Responsabilidades	**Clases Relacionadas**
• Genera oleadas de enemigos • Monitorea el estado de las oleadas • Realiza acciones al terminar una oleada	• *MonoBehaviour* • *Wave* • *GameManager* • *Enemy*

Tabla 78. Tarjeta CRC WayPoints

Tarjeta CRC	
Clase: *WayPoints*	
Responsabilidades	**Clases Relacionadas**
• Almacena y proporciona acceso a los puntos de ruta • Actualiza los puntos de ruta	• *MonoBehaviour* • *EnemyMovement*

Tabla 79. Tarjeta CRC PauseMenu

Tarjeta CRC	
Clase: *WayPoints*	
Responsabilidades	**Clases Relacionadas**
• Brinda la posibilidad de pausar el juego en cualquier instante de tiempo.	• *MonoBehaviour*

ANEXO 7. Videojuego *Kingdom Rush*

En la siguiente figura se observa una captura de pantalla del videojuego Kingdom Rush Fronties, donde se puede apreciar la selección y visualización de opciones de construcción.

Figura 23.Imagen del juego Kingdom Rush Frontiers*(11)*