A Hero's Resolve

Alejandro Hu, Alfonso Manrique

Diego Baratto Valdivia

Desarrollo de Aplicaciones Multiplataformas
Curso 2024-2025

Centro Educativo iFP Campus Madrid 2
Convocatoria de Presentación: Junio 2025 (Calibri 18)

Índice

Resumen

El presente Trabajo de Fin de Grado detalla el proceso de diseño y desarrollo de un prototipo funcional de videojuego del género Rol (RPG) por turnos, titulado A Hero's Resolve". Inspirado en JRPGs clásicos, el proyecto busca ofrecer una experiencia de juego con una estética pixel art y una ambientación de fantasía, implementando mecánicas esenciales como la exploración de entornos 2D, un sistema de diálogo interactivo, gestión de inventario y equipamiento, y una estructura básica de combate por turnos.

El desarrollo se ha llevado a cabo utilizando el motor Unity y el lenguaje de programación C#, aplicando una metodología ágil adaptada para la gestión del proyecto. Se han implementado sistemas clave como el control del personaje jugador, el comportamiento de NPCs con diálogos dinámicos, un sistema de inventario con capacidad de apilamiento y uso de objetos, y una pantalla de equipamiento que permite la personalización de los personajes de la party. Asimismo, se ha establecido la base para un sistema de combate por turnos y una pantalla de estado del personaje que muestra estadísticas detalladas y habilidades.

Como principales conclusiones, se valora positivamente la consecución de gran parte de los objetivos específicos planteados, logrando un prototipo jugable que demuestra las mecánicas centrales. El proyecto ha supuesto un importante aprendizaje en el uso de herramientas de desarrollo como Unity y LibreSprite, así como en la resolución de problemas técnicos y la gestión del trabajo en equipo. Aunque algunas funcionalidades avanzadas, como un sistema de misiones complejo o la IA de combate completa, quedaron en una fase más inicial debido a limitaciones de tiempo, la base desarrollada es sólida y ofrece potencial para futuras expansiones.

Introducción

El presente Trabajo de Fin de Grado (TFG) se centra en el diseño y desarrollo de un prototipo funcional de un videojuego perteneciente al género Rol (RPG) por turnos, caracterizado por una estética pixel art y una ambientación de fantasía. El proyecto explora la creación de una experiencia de juego que combina la exploración de entornos 2D desde una perspectiva top-down con un sistema de combate por turnos presentado en una vista 3/4. La narrativa principal, que sumerge al jugador en la piel de un joven héroe destinado a restaurar el equilibrio en un reino amenazado por una antigua oscuridad, guiará al jugador a través de un mundo donde deberá completar misiones para progresar de forma lineal y acceder a nuevas zonas. Los jugadores controlarán a un aventurero principal, con la posibilidad de reclutar y gestionar un equipo de personajes.

El desarrollo del prototipo se enfocará en la implementación de las mecánicas fundamentales del género RPG, incluyendo un sistema de diálogo interactivo con personajes no jugadores (NPCs), la gestión de un inventario de objetos compartido por la party, un sistema de equipamiento para los personajes, y la estructura básica del combate por turnos iniciado mediante el contacto con enemigos visibles en el mapa, con opciones como ataque, habilidades, defensa, uso de objetos y huida. Este proyecto se enmarca en el contexto de un mercado de videojuegos donde los RPGs con estética retro y mecánicas clásicas mantienen una notable popularidad, especialmente en el sector indie(independiente), buscando atraer tanto a jugadores nostálgicos por los clásicos RPG, como a aquellos que se inician en el género. El trabajo se abordará mediante una metodología ágil adaptada, utilizando el motor Unity y el lenguaje C# como principales herramientas de desarrollo.

Introducción: Justificación

La elección de desarrollar un videojuego de Rol (RPG) como Trabajo de Fin de Grado surge de un fuerte interés compartido por los miembros del grupo hacia este género. Desde temprana edad, hemos sido jugadores de títulos clásicos y modernos que han definido el RPG, como Pokémon, Dragon Quest y Final Fantasy. La capacidad de estos juegos para contar historias envolventes, permitir la progresión y personalización de personajes, el sistema de combates y ofrecer mundos ricos para la exploración siempre nos ha resultado especialmente atractiva y motivadora.

Adicionalmente, la elección de un videojuego RPG para el Trabajo de Fin de Grado se fundamenta en la experiencia previa del grupo en el desarrollo de un juego de género similar, aunque más limitado, durante el transcurso del ciclo formativo. Esta experiencia previa proporciona una base de conocimientos que facilita un enfoque más eficiente y estructurado del proyecto.

Más allá de la atracción por el género y la experiencia previa, el grupo se sintió motivado por el considerable desafío técnico y creativo que implica la creación de un RPG. Este tipo de proyecto permite aplicar de las habilidades desarrolladas en el ciclo DAM y las habilidades adquirides mediante el autoaprendizaje ,desde la programación avanzada hasta el diseño de sistemas complejos y la creatividad en el diseño de mecánicas.

Desde una perspectiva técnica, el desarrollo de un RPG por turnos con las características planteadas (exploración 2D, combate 3/4, sistemas de diálogo, inventario, equipamiento, misiones y combate) representa una excelente oportunidad para aplicar y profundizar en conceptos de programación orientada a objetos, gestión de estados, diseño de interfaces de usuario interactivas, inteligencia artificial básica para enemigos y gestión de datos de juego. La elección del motor

Unity y el lenguaje C# se compatibiliza con las enseñanzas adquiridas del durante el cicló y el mercado del género.

Finalmente, la justificación objetiva se basa en la continua relevancia y demanda del género RPG, especialmente aquellos con estética pixel art que llaman a la nostalgia y ofrecen mecánicas de juego sólidas. Títulos independientes recientes como *Chained Echoes* o *Sea of Stars* han demostrado el interés del público por este tipo de propuestas, lo que valida la temática escogida y por lo que este proyecto tiene interés en el panorama actual de los videojuegos.

Introducción: Objetivos

Objetivo General (Principal):

El objetivo general de este Trabajo de Fin de Grado es el diseño y desarrollo de un modelo funcional de un videojuego del género Rol (RPG) por turnos, con una estética pixel art y ambientación de fantasía. Este prototipo deberá implementar las mecánicas esenciales que caracterizan al género, incluyendo la exploración de un entorno 2D, un sistema de misiones lineal, un sistema de diálogo interactivo con personajes no jugadores (NPCs), la gestión de un inventario y un sistema de equipamiento para los personajes del jugador, y una estructura básica para el combate por turnos. La consecución de este objetivo permitirá aplicar de manera práctica e integral los conocimientos y habilidades adquiridos durante el ciclo formativo de Desarrollo de Aplicaciones Multiplataforma y conocimientos adquiridos mediante autoaprendizaje, dando lugar a un producto software jugable y bien documentado.

Objetivos Específicos(Secundarios):

- **Diseño y Planificación del Proyecto:**

- Definir la narrativa fundamental, el flujo de juego principal y las mecánicas centrales del RPG, incluyendo la estructura de progresión lineal basada en misiones.
- Diseñar la estructura y el flujo de la interfaz de usuario (UI) para las pantallas clave: menú principal, exploración (HUD), diálogo, inventario, equipamiento, diario de misiones y combate.
- Establecer la arquitectura de datos para los elementos del juego, incluyendo la definición de las propiedades y tipos de los objetos, la estructura de datos de los personajes y el sistema de gestión del inventario y la estructura básica para los datos de las misiones.

- **Implementación de la Base de Mecánicas de Exploración e Interacción:**
  - Implementar un sistema de movimiento controlable por el jugador para el personaje principal en un entorno 2D top-down, incluyendo animaciones básicas de idle, caminar en cuatro direcciones y atacar.
  - Desarrollar un sistema de control de cámara utilizando Cinemachine que siga al jugador de forma fluida y respete los límites del mapa mediante la configuración de sus componentes.
  - Crear la lógica para el comportamiento de Personajes No Jugadores (NPCs), que incluya: movimiento de patrulla programado, comportamiento de idle dinámico para NPCs estáticos, y capacidad para establecer una pose inicial fija.
  - Desarrollar un sistema de diálogo interactivo con visualización de texto en UI dedicada, efecto de máquina de escribir, paginación de texto, indicador visual de continuación, pausa de personajes durante el diálogo y giro del NPC hacia el jugador.
- **Implementación de Sistemas de Gestión del Jugador y Progresión:**
  - Implementar un sistema de inventario funcional compartido por la party, permitiendo añadir, quitar y apilar objetos.

- Desarrollar la interfaz de usuario para el inventario principal, con visualización en cuadrícula, actualización en tiempo real y un panel de información con acciones (Usar, Equipar, Tirar).
- Implementar la funcionalidad de "Usar" objetos consumibles, afectando los stats del personaje y controlando su uso si no es necesario.
- Desarrollar la estructura de datos para los personajes (stats base, HP/MP actuales, gestión de equipo).
- Implementar la funcionalidad de "Equipar" y "Desequipar" objetos, con actualización dinámica de stats y movimiento de objetos entre inventario y equipo.
- Diseñar e implementar la interfaz de usuario de la "Pantalla de Equipamiento" (visualización de personaje, stats, slots de equipo, inventario, selección de party, interacción para equipar/desequipar).
- Diseñar e implementar la interfaz de usuario de la "Pantalla de Datos" (visualización de datos del personaje, stats, nivel, EXP, HP, y habilidades con su efecto).
- Desarrollar un sistema de misiones básico (aceptar vía diálogo, seguimiento de estado simple, desbloqueo de zonas o eventos por finalización).
- Establecer la estructura para una progresión lineal del juego basada en la finalización de misiones clave.

- **Implementación del Sistema de Combate (Básico):**
  - Diseñar la transición visual desde la exploración al combate al interactuar con enemigos visibles.
  - Configurar la presentación visual del combate (vista 3/4, posicionamiento de personajes/enemigos) y crear sprites de personajes detallados para el combate (48x48). ○ Desarrollar un HUD de combate básico (HP/MP, menú de acciones:

Ataque, Habilidades/Especial, Defensa, Objetos, Huir).

- Implementar la lógica para acciones de "Ataque" básico y "Defensa".
- Establecer la estructura para un sistema de combate por turnos (determinación de turnos, ejecución de acciones).

- **Contenido y Pruebas del Prototipo:**
  - Diseñar y crear un mapa de exploración de prueba con un NPC interactivo (diálogo, misión), un cofre y un enemigo.
  - Crear y buscar los assets de pixel art necesarios (personaje jugador, NPC, enemigo, tiles básicos, iconos de objetos, UI).
  - Realizar pruebas funcionales de todas las mecánicas implementadas para asegurar su correcto funcionamiento y corregir errores.
- **Documentación y Gestión del Proyecto:**
  - Elaborar la memoria del Trabajo de Fin de Grado siguiendo la estructura y los requisitos formales.
  - Utilizar un sistema de control de versiones (Git/GitHub) para el código fuente y la gestión colaborativa.

Marco teórico (fundamentación teórica)

1. Historia y evolución de los videojuegos RPG

Los videojuegos de rol (RPG) se originaron a partir de juegos de mesa como *Dungeons & Dragons* en los años 70, adoptando conceptos como la progresión de personajes y el combate estratégico. Esta base se trasladó al formato digital, con títulos tempranos como *Dragonstomper* (1982). Sin embargo, fueron juegos japoneses como *Dragon Quest* (1986) y *Final Fantasy* (1987) los que consolidaron pilares del género como la exploración cenital (top-down), el combate por turnos y una narrativa profunda. Durante los 90, la tecnología de discos ópticos permitió una

mayor complejidad, resultando en hitos como *Final Fantasy VII* (1997) (Wikipedia, 2024).

2. Elementos clave en el diseño de RPG por turnos

Los RPG por turnos clásicos se construyen sobre una serie de sistemas fundamentales:

- **Sistema de combate por turnos:** Se basa en que cada personaje (jugador o enemigo) actúa en un orden determinado, lo que permite planificar estrategias y aprovechar debilidades enemigas. Este sistema fomenta la reflexión táctica y está presente en juegos como *Pokémon*, *Persona* o *Chrono Trigger* .
- **Exploración en vista top-down 2D:** Ofrece una visualización clara del entorno y es un recurso gráfico muy eficiente, aún hoy utilizado por muchos juegos indie por su facilidad de diseño y legibilidad.
- **Sistema de progresión:** Los personajes ganan experiencia, suben de nivel y pueden equiparse con armas, armaduras y objetos que modifican sus estadísticas base. Esta evolución incentiva al jugador a seguir progresando en la historia y a optimizar su equipo.
- **Inventario y objetos:** Elemento imprescindible que permite administrar pociones, equipamiento y objetos clave. Su buena gestión puede marcar la diferencia entre ganar o perder un combate.
- **Narrativa lineal con desbloqueo de zonas:** En muchos RPG clásicos, el avance está condicionado por cumplir misiones o vencer ciertos jefes, lo que permite una estructura controlada pero progresiva del mundo del juego.

(Punto de Respawn, 2023)

3. Herramientas y tecnologías aplicadas

Este TFG ha sido desarrollado con Unity, uno de los motores de juego más versátiles del mercado actual, especialmente popular en la comunidad indie por su facilidad de uso, documentación y soporte multiplataforma. Unity permite desarrollar videojuegos 2D de forma muy eficiente gracias a herramientas como:

- Tilemap: Para la creación modular de mapas 2D.
- Animator y animaciones 2D: Para controlar el estado de los personajes (caminar, atacar, recibir daño).
- Sistema de UI Canvas: Para implementar menús de combate, inventario y HUD.
- Sistema de prefabs: Que permite reutilizar objetos y personajes manteniendo la escalabilidad del proyecto.
- Lenguaje de programación C#: Potente y ampliamente documentado, ideal para implementar lógica de juego, sistema de turnos, combate y eventos personalizados.

Unity también facilita el despliegue en múltiples plataformas como PC, móviles o consolas, lo que lo convierte en una solución robusta para el desarrollo profesional (Unity Technologies, 2024).

4. Desarrollo actual y aplicaciones del RPG por turnos

Los RPG por turnos continúan siendo relevantes, especialmente en la escena independiente, donde se renuevan fórmulas clásicas con mejoras modernas e innovaciones en mecánicas, calidad y narrativa. Estos juegos atraen por su profundidad estratégica en combate, que exige planificación y gestión de recursos, y por la inmersión narrativa que ofrecen. Además, su adaptabilidad les permite estar

presentes en una amplia gama de plataformas actuales, desde consolas y PC hasta smartphones (paradigmadigital, 2023).

Bibliografía(formato APA)

- TutsPlus. (s.f.). *Equilibrar los RPG basados en turnos: el panorama general*. Recuperado de https://gamedevelopment.tutsplus.com/es/equilibrar-losrpg-basados-en-turnos-el-panorama-general--gamedev-8286a
- Unity Technologies. (2024). *Unity for 2D games*. Recuperado de https://unity.com/es/solutions/2d
- Wikipedia contributors. (2024). *Videojuego de rol*. Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Videojuego_de_rol
- Wikipedia contributors. (2024). *Historia de los videojuegos de rol occidentales*. Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Historia_de_los_videojuegos_de_rol_occidentales
- Punto de Respawn. (2023). *Elementos clave diseño RPG por turnos.* Recuperado de https://puntoderespawn.com/articulos/opinion/combatepor-turnos-un-sistema-clasico-y-estrategico/
- Paradigma Digital .(2023). Experiencias de usuarios en los videojuegos RPG. Recuperado de https://www.paradigmadigital.com/dev/experienciausuario-videojuegos-rpg/

Desarrollo

Desarrollo: materiales y métodos

Para la realización de este proyecto, se ha empleado un conjunto de herramientas de software especializadas en las diferentes etapas del desarrollo de videojuegos. A

continuación, se detallan las principales herramientas utilizadas y la justificación de su elección:

- **Motor de Videojuegos: Unity (Versión 2022.3.x LTS)** ○ **Descripción:** Motor de desarrollo multiplataforma versátil, ampliamente usado para juegos 2D y 3D, destacando por su sistema de componentes y su extenso Asset Store.
    - ○ **Uso en el proyecto:** Entorno principal para creación de escenas, implementación de lógica de juego (scripting), creación de UI, gestión de animaciones y compilación del proyecto.
    - ○ **Alternativas consideradas:** Se consideraron brevemente otros motores como Godot Engine (por su naturaleza open-source y enfoque en 2D) o Unreal Engine (más orientado a proyectos 3D de alta fidelidad).
    - ○ **Justificación de la elección:** Se seleccionó Unity por la familiaridad y habilidades previas del equipo adquiridas durante el ciclo formativo, su extensa documentación y gran cantidad de recursos de aprendizaje (críticos para plazos de TFG).
- **Lenguaje de Programación: C# (C Sharp)** ○ **Descripción:** C# es un lenguaje de programación moderno, orientado a objetos, desarrollado por Microsoft y es el lenguaje principal de scripting para Unity.
    - ○ **Uso en el proyecto:** Todo el código para la lógica de juego, mecánicas (movimiento, diálogo, inventario, combate), gestión de datos y control de la interfaz de usuario se ha implementado en C#. ○ **Alternativas consideradas:** Dentro de Unity, C# es la opción predominante. Otros motores podrían usar C++, GDScript (Godot), etc.
    - ○ **Justificación de la elección:** La elección de C# viene determinada por su uso como lenguaje principal en Unity. Sus

características, como la fuerte tipificación, la gestión automática de memoria y las capacidades de programación orientada a objetos, son idóneas para desarrollar sistemas de juego complejos y mantenibles, además de ser el lenguaje con el que se ha aprendido a lo largo del curso.

- **Software de Creación de Gráficos Pixel Art: LibreSprite** ○ **Descripción:** LibreSprite es un editor de imágenes y animación de código abierto, derivado de Aseprite, especializado en la creación de pixel art.
    - **Uso en el proyecto:** Se ha utilizado para diseñar y crear todos los assets visuales del juego, incluyendo los sprites de los personajes (exploración y combate), enemigos, elementos del entorno (tilesets), iconos de objetos y componentes de la interfaz de usuario.
    - **Alternativas consideradas:** Aseprite (versión de pago con características similares), GIMP (editor de imágenes más generalista con capacidades para pixel art), Photoshop (herramienta profesional con funcionalidades extensas).
    - **Justificación de la elección:** Se eligió LibreSprite por ser una herramienta gratuita, de código abierto y específicamente diseñada para pixel art, ofreciendo funcionalidades como la gestión de paletas de colores, capas, herramientas de animación por frames y una interfaz intuitiva para este estilo artístico. Su compatibilidad con formatos de imagen estándar facilita la importación de los assets a Unity.
- **Sistema de Control de Versiones: Git y GitHub**
    - **Descripción** Git es un sistema de control de versiones distribuido para el seguimiento de cambios, mientras que GitHub es una plataforma web para alojar repositorios Git que facilita la colaboración.

- **Uso en el proyecto:** Se utiliza Git para el control de versiones local, permitiendo registrar los cambios en el código y los assets, crear ramas para desarrollar nuevas funcionalidades de forma aislada y fusionar los cambios. GitHub se emplea como repositorio remoto centralizado para facilitar el trabajo en equipo, la revisión de código y como copia de seguridad del proyecto.
- **Alternativas consideradas:** Otros sistemas de control de versiones como Subversion (SVN) o plataformas como GitLab o Bitbucket.
- **Justificación de la elección:** Git es el estándar de facto en la industria del desarrollo de software para el control de versiones debido a su eficiencia y flexibilidad. GitHub es la plataforma más popular para alojar repositorios Git, ofreciendo una excelente integración con herramientas de desarrollo y facilitando la colaboración, además de ser una herramienta que se ha hecho de su uso durante el ciclo.

- **Entorno de Desarrollo Integrado (IDE): Microsoft Visual Studio Code (o Visual Studio Community)**
  - **Descripción:** Un editor de código fuente ligero pero potente (VS Code) o un IDE completo (Visual Studio) que ofrece funcionalidades como resaltado de sintaxis, depuración, autocompletado (IntelliSense) y gestión de proyectos.
  - **Uso en el proyecto:** Para escribir, editar y depurar el código C# de los scripts de Unity.
  - **Alternativas consideradas:** Otros editores como Rider (de JetBrains, específico para Unity y C#).
  - **Justificación de la elección:** Visual Studio Code (o Visual Studio Community) ofrece una excelente integración con Unity, herramientas de depuración robustas y una amplia gama de

extensiones que mejoran la productividad del desarrollador, además de ser el entorno en el que se hizo uso durante el curso.

- **Herramientas de Comunicación: Whatsapp** ○ **Descripción:** Software para la comunicación y delegación de tareas.
  - **Uso en el proyecto:** Para comunicarse, notificar, gestionar y delegar tareas.
  - **Alternativas consideradas:** Microsoft Teams, Trello etc.
  - **Justificación de la elección:** Se ha considerado esta opción por comodidad y facilidad.

Desarrollo: resultados y análisis

1. Diseño detallado del Sistema (Análisis y Diseño Previo a Implementación)

1.1. Requisitos del Sistema (Funcionales y No Funcionales) A. Requisitos

Funcionales (RF):

Describen las funcionalidades específicas que el sistema debe ser capaz de realizar.

- **RF1: Control del Personaje y Exploración:** El sistema permitirá al jugador controlar un personaje principal en un entorno 2D con perspectiva topdown. El personaje se moverá en cuatro direcciones cardinales, con animaciones de caminar y reposo. Se implementará una cámara con Cinemachine que seguirá al jugador, restringida a los límites del mapa.
- **RF2: Interacción con el Entorno y NPCs:** El jugador podrá interactuar con Personajes No Jugadores (NPCs) mediante una tecla de acción. Los NPCs exhibirán comportamientos básicos como movimiento de patrulla, estados de reposo dinámicos (cambiando su orientación) o una pose inicial fija.

- **RF3: Sistema de Diálogo Interactivo:** Al interactuar con un NPC, se activará una UI de diálogo con efecto de "máquina de escribir" para el texto, paginación automática para mensajes largos (intentando respetar el corte de palabras), y un indicador visual animado para avanzar. Durante el diálogo, el movimiento del jugador y del NPC se pausará, y el NPC se orientará hacia el jugador. Se incluirán sonidos básicos para el tipeo y avance.
- **RF4: Gestión de Inventario y Objetos:** Se gestionará un inventario para el jugador con capacidad definida, permitiendo añadir y quitar objetos. Soportará el apilamiento de ítems según un máximo por slot. Una UI dedicada mostrará los objetos en slots, se actualizará en tiempo real y, al seleccionar un ítem, presentará un panel con su información (nombre, descripción, stats) y acciones relevantes (Usar, Equipar, Tirar).
- **RF5: Sistema de Equipamiento de Personajes:** Los personajes podrán equipar Armas y Armaduras (Casco, Pechera, Botas) en ranuras específicas, lo que modificará dinámicamente sus estadísticas. Una pantalla de equipamiento permitirá visualizar al personaje, sus stats y equipo, el inventario del jugador, y cambiar entre miembros de la party para gestionar su equipamiento.
- **RF6: Sistema de Misiones y Progresión:** Se implementará un sistema básico de misiones que el jugador podrá aceptar mediante diálogos. Se realizará un seguimiento simple del estado de las misiones, y su finalización podrá desbloquear nuevas zonas o eventos, facilitando una progresión lineal.
- **RF7: Sistema de Combate por Turnos:** El combate por turnos se iniciará por contacto con enemigos visibles en el mapa, con una transición visual. Se presentará en una vista 3/4 con sprites de 48x48. Un HUD básico mostrará HP/MP y un menú de acciones (mínimo Ataque y Defensa), junto a una estructura inicial para la gestión de turnos.

- **RF8: Contenido y Assets del Proyecto:** Se creará un mapa de prueba con al menos un NPC interactivo (con diálogo y misión), un cofre y un tipo de enemigo. Se desarrollarán los assets de pixel art esenciales (personaje jugador, NPC, enemigo, tileset básico, iconos y UI). **B. Requisitos No Funcionales (RNF):**

Describen las cualidades del sistema y las restricciones bajo las cuales debe operar.

- **RNF1: Usabilidad:** La interfaz de usuario y los controles del juego serán intuitivos y de fácil aprendizaje, especialmente para jugadores familiarizados con el género RPG.
- **RNF2: Estética Visual:** Se mantendrá una estética pixel art coherente y unificada en todos los elementos visuales del juego, incluyendo personajes, escenarios, objetos e interfaz.
- **RNF3: Rendimiento:** El prototipo del juego se ejecutará de manera fluida, buscando una optimización adecuada del rendimiento.
- **RNF4: Mantenibilidad y Escalabilidad del Código:** El código fuente estará organizado modularmente, aplicando principios de programación orientada a objetos y patrones de diseño (como Singleton para managers). Se utilizará control de versiones (Git/GitHub) y se incluirán comentarios en el código para facilitar su comprensión, mantenimiento y futura expansión.

1.2. Diseño del Software del Juego

La arquitectura del software del proyecto se basa en un enfoque modular y la separación de responsabilidades, utilizando scripts y "managers" dedicados a aspectos específicos del juego. Esta estructura busca promover la cohesión, reducir

el acoplamiento entre sistemas y facilitar el desarrollo, mantenimiento y la futura escalabilidad del videojuego.

- **Patrones de Diseño Clave Utilizados:**
  - **Singleton:** Este patrón se ha empleado para los principales sistemas de gestión que requieren un punto de acceso global y único a lo largo del juego. Esto incluye:
    - **PlayerInventory**: Para la gestión centralizada de los datos del inventario del jugador.
    - **InventoryUIManager**: Para el control de la interfaz de usuario del inventario principal.
    - **EquipmentScreenManager**: Para la gestión de la interfaz de usuario de la pantalla de equipamiento.
    - **PartyManager**: Para la gestión de los miembros de la party del jugador.
    - **CombatManager**: Para gestión de combate por turnos
    - **CharacterStatsScreenManager**: Para la gestión de la pantalla de estado/datos del personaje.
  - **ScriptableObjects para Datos:** Se ha hecho un uso extenso de ScriptableObjects para definir los datos base de elementos del juego como los objetos (ItemData) y habilidades (AbilityData) o misiones (QuestData). Esta elección permite una gestión de datos flexible y independiente del código, facilitando la creación y modificación de contenido directamente desde el editor de Unity sin necesidad de recompilar scripts.
  - **Sistema de Eventos:** Para la comunicación entre ciertos sistemas, especialmente entre la lógica de datos (ej: PlayerInventory) y la UI (InventoryUIManager, EquipmentScreenManager), se utiliza un sistema de eventos de C# (delegados y event Action). Esto permite

que los sistemas de UI reaccionen a los cambios en los datos (ej: PlayerInventory.OnInventoryChanged) sin que el sistema de datos necesite conocer los detalles específicos de la UI, promoviendo un diseño más limpio y flexible.

- **Organización de Scripts y GameObjects:**
    - **Scripts de Componentes:** La lógica específica de GameObjects individuales (como el jugador, NPCs, ítems interactuables) se implementa en scripts que se adjuntan como componentes a dichos GameObjects (ej: PlayerMovement.cs, NPCDialogue.cs, Character.cs).
    - **Managers Globales:** Los scripts Singleton mencionados anteriormente están adjuntos a GameObjects persistentes (que utilizan DontDestroyOnLoad) para asegurar la continuidad de los datos y la gestión entre escenas.
    - **UI Managers Específicos:** Cada pantalla principal de la UI (Inventario, Equipamiento, Diálogo, Estado) tiene su propio script gestor que controla sus elementos visuales y su lógica de interacción, activándose y desactivándose según sea necesario.

- **Flujo de Datos e Interacción entre Módulos (General):**

  El sistema se articula en torno a la interacción de varios módulos clave:

    - El Personaje Jugador (PlayerMovement, PlayerInteraction, Character) es la entidad central en la exploración. PlayerInteraction detecta NPCs y objetos interactuables.
    - Al interactuar con un NPC, el script NPCDialogue del NPC gestiona el flujo de la conversación, mostrando el texto a través de una UI de diálogo dedicada y pausando la acción del jugador. Los

diálogos pueden desencadenar eventos o modificar el estado de misiones.

- El Sistema de Inventario (PlayerInventory) almacena los objetos del jugador. El InventoryUIManager se suscribe al evento OnInventoryChanged del PlayerInventory para actualizar la UI del inventario principal en tiempo real. Al seleccionar un objeto, el InventoryUIManager muestra un panel de información (ItemInfoActionPanel) con opciones. ○ La acción “Equipar” desde el inventario principal invoca al EquipmentScreenManager para abrir la Pantalla de Equipamiento. ○ El EquipmentScreenManager utiliza el PartyManager (que mantiene la lista de personajes) para permitir la selección del personaje a equipar. Muestra los datos del personaje seleccionado (obtenidos de su componente Character) y su equipo actual. También muestra una vista del PlayerInventory y gestiona la lógica de equipar/desequipar objetos, actualizando tanto el Character (sus equippedItems y stats) como el PlayerInventory.
- La acción “Ver Datos” desde la pantalla de equipamiento invoca al CharacterStatsScreenManager para mostrar la pantalla de estado detallado del personaje seleccionado.
- El CharacterStatsScreenManager accede a los datos del Character (stats, XP, knownAbilities) para poblar su UI. ○ El Control de Cámara (Cinemachine) sigue al jugador y se confina a los límites del mapa.
- El Sistema de Combate se activará por contacto con enemigos, gestionando los turnos, acciones y la actualización del HUD de combate.

- El Sistema de Misiones interactuará con los diálogos (para aceptar/completar misiones) y con el estado del juego (para desbloquear contenido).

- **Estructura de Carpetas del Proyecto:**
  "Assets/Scripts" (con subcarpetas por sistema), "Assets/Sprites" (con subcarpetas por tipo), "Assets/Prefabs" (UI, Personajes, Objetos), "Assets/ScriptableObjects" (Items, Abilities), "Assets/Scenes".

1.3. Diseño de la Interfaz de Usuario (UI) y Experiencia de Usuario (UX)

El diseño de la interfaz de usuario (UI) y la experiencia de usuario (UX) del proyecto se ha centrado en la claridad, la facilidad de uso y la coherencia con la estética pixel art general del juego.

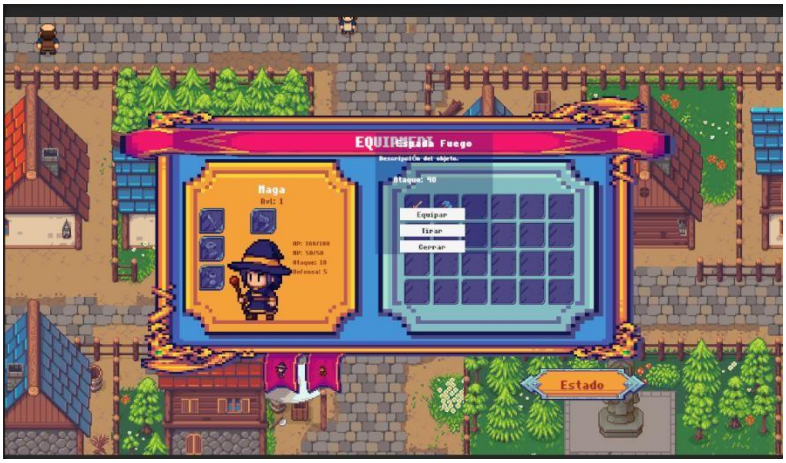
- **Principios de Diseño Aplicados:**

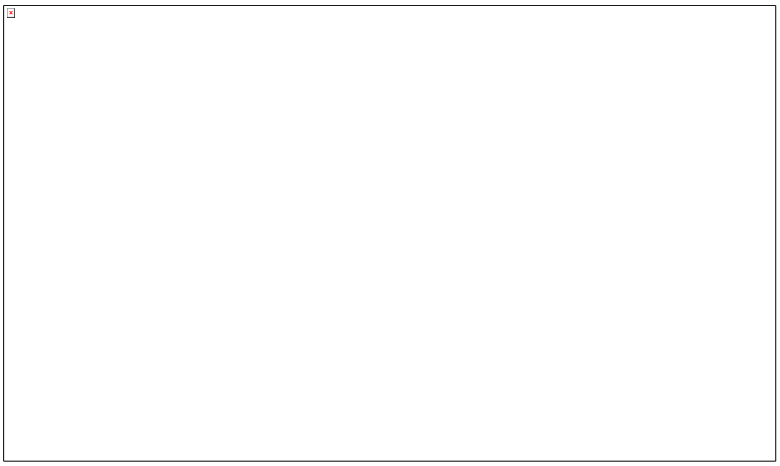
  - **Consistencia Visual:** Todos los elementos de la UI (paneles, botones, textos, iconos) comparten un estilo pixel art unificado en términos de fuentes, paleta de colores y diseño de bordes/fondos.
  - **Jerarquía de Información:** La información se presenta de forma organizada, destacando los datos más relevantes y utilizando agrupaciones lógicas para facilitar su lectura.
  - **Feedback al Usuario:** Se proporcionan indicaciones visuales (ej: el indicador "v" en los diálogos, resaltado de slots seleccionados) para informar al jugador sobre el estado del sistema y las acciones disponibles.
  - **Navegación Intuitiva:** El flujo entre las diferentes pantallas de menú (Inventario, Equipamiento, Estado) se ha diseñado para ser lógico y accesible, permitiendo al jugador volver fácilmente a la pantalla anterior o al juego.

- **Diseño de Pantallas Clave:**
  - **Caja de Diálogo:**
    - **Descripción:** La Caja de Diálogo se presenta como un panel rectangular inferior con borde distintivo y fondo semitransparente. El texto del NPC aparece con efecto de máquina de escribir, y un indicador 'v' animado guía al jugador para avanzar. Se ha seleccionado una fuente pixel art coherente, asegurando su legibilidad mediante un tamaño y contraste adecuados.
  - **Pantalla de Inventario Principal:**
    - **Descripción:** La pantalla de inventario presenta los objetos del jugador en una cuadrícula de slots. Cada slot muestra el icono del objeto y la cantidad si es apilable. Al seleccionar un objeto, aparece un panel de información contextual (ItemInfoActionPanel) adyacente al slot, mostrando el nombre, descripción, stats (si es equipable) y botones de acción relevantes ('Usar', 'Equipar', 'Tirar', 'Cerrar'). El diseño busca una gestión rápida y eficiente de los objetos.

  - **Pantalla de Equipamiento:**

- **Descripción:** La Pantalla de Equipamiento se estructura en dos secciones. A la izquierda, se visualiza el personaje seleccionado (sprite, nombre, nivel, estadísticas clave) junto a sus ranuras de equipo con los objetos actuales y los iconos para cambiar de miembro de la party. A la derecha, se muestra el inventario del jugador para seleccionar objetos a equipar. Al seleccionar un ítem (del inventario o ya equipado), emerge un panel contextual con información y acciones pertinentes (ej: 'Equipar', 'Desequipar'), buscando siempre la claridad visual e intuitividad para el jugador.





- **Pantalla de Estado/Datos del Personaje:**

- **Descripción:** La pantalla de estado mostrará un sprite grande del personaje seleccionado, su nombre, nivel, información detallada de experiencia (XP actual y para el siguiente nivel, con una barra de progreso), una lista completa de sus estadísticas (HP, MP, Ataque, Defensa, etc., incluyendo bonos de equipo), y una sección para visualizar sus habilidades conocidas con sus descripciones. Se busca transmitir información específica al jugador.

- **HUD de Combate:**
  - **Descripción**: El HUD de combate mostrará la información vital de los personajes de la party (barras de HP/MP, nombres) y de los enemigos. Un menú de acciones por turnos permitirá al jugador seleccionar entre 'Atacar', 'Habilidades', 'Defender', 'Objetos' y 'Huir'. Una interfaz clara e intuitiva.

1.4. Diseño de Datos Detallado:

En esta sección se describe la estructura y el propósito de las principales entidades de datos que conforman el núcleo del sistema de juego. Se ha optado por el uso de clases C# y ScriptableObjects para una gestión flexible y organizada de la información.

Objetos (Items - ItemData):

Los objetos del juego se definen mediante ScriptableObjects, utilizando una clase denominada ItemData. Esta aproximación permite crear y configurar diferentes tipos de objetos directamente desde el editor de Unity como assets independientes, facilitando la gestión de una gran variedad de ítems. Cada ItemData contiene la siguiente información esencial:

- **itemID (string):** Un identificador único para el objeto
- **itemName (string):** El nombre del objeto tal como se mostrará al jugador en las interfaces de usuario (inventario, descripciones, etc.).
- **icon (Sprite):** La imagen visual del objeto que se muestra en el inventario y otros menús.
- **description (string):** Un texto descriptivo que proporciona información adicional al jugador sobre el objeto, sus usos o su historia.
- **itemType (enum ItemType):** Clasifica el objeto en una categoría general (ej: Weapon, Armor, Consumable, Quest), lo que determina su comportamiento principal y cómo interactúa con otros sistemas del juego.
- **isStackable (bool):** Un valor booleano que indica si múltiples unidades de este objeto pueden ocupar un solo slot del inventario (ej: pociones) o si cada unidad requiere un slot individual (ej: armas).
- **maxStackSize (int):** Si el objeto es apilable (isStackable = true), este campo define la cantidad máxima de unidades que pueden almacenarse en un único slot del inventario.
- **isEquipable (bool):** Un valor booleano que indica si el objeto puede ser equipado por un personaje para mejorar sus atributos o capacidades.
- **equipmentSlot (enum EquipmentSlot):** Si el objeto es equipable, este campo especifica la ranura de equipamiento a la que pertenece (ej:

MainHand para armas, Head para cascos, Body para armaduras, Feet para botas).

- **Bonificaciones a Estadísticas (int):** Diversos campos numéricos (attackBonus, defenseBonus, maxHpBonus, maxMpBonus, speedBonus, etc.) que representan las mejoras a las estadísticas base del personaje cuando el objeto está equipado.
- **isConsumable (bool):** Indica si el objeto se consume y desaparece del inventario tras su uso.
- **Efectos de Consumible (int):** Campos como hpToRestore y mpToRestore que definen la cantidad de Puntos de Vida o Puntos de Maná que el objeto restaura al ser utilizado.

Personajes (Character):

La información y el estado de cada personaje jugable, se gestionan a través de un componente Character (script C#). Los campos y propiedades fundamentales de esta clase son:

- **Información Básica:** characterName (string) para el nombre visible, level (int) para el nivel actual, y portraitSprite (Sprite) para su representación visual en menús.
- **Experiencia y Progresión:** currentXP (int) para los puntos de experiencia acumulados y experienceToNextLevel (int) para la cantidad necesaria para el siguiente nivel.
- **Estadísticas Base:** Un conjunto de campos públicos que definen los atributos base del personaje.
- **Estadísticas Actuales:** currentHP (int) y currentMP (int) para los puntos de vida y maná actuales del personaje.
- **Gestión de Equipo:** equippedItems (Dictionary<EquipmentSlot, ItemData>) es un diccionario que almacena las referencias a los ItemData

de los objetos actualmente equipados por el personaje, utilizando el EquipmentSlot como clave para identificar la ranura (Casco, Cuerpo, etc.).

- **Estadísticas Totales Calculadas:** Se utilizan propiedades públicas de solo lectura (ej: MaxHP, Attack, Defense) que calculan dinámicamente el valor total de cada estadística, suma base con equipamiento.
- **Habilidades Conocidas:** knownAbilities (List<AbilityData>) es una lista que contiene las referencias a los AbilityData de todas las habilidades que el personajetiene y puede utilizar.
- **Métodos de Interacción:** La clase Character encapsula métodos públicos para modificar su estado, como EquipItem(), UnequipItem() (que interactúan con el PlayerInventory para el intercambio de objetos), Heal(), RestoreMana(), TakeDamage(), SpendMana() (para ser llamados por objetos consumibles o durante el combate), y GainXP() (para la progresión).

Inventario del Jugador (PlayerInventory y InventorySlot):

El inventario del jugador, que es compartido por toda la party, se gestiona mediante una clase Singleton denominada PlayerInventory. Su estructura principal es:

- **inventorySlots (List<InventorySlot>):** Es la lista que contiene todos los slots del inventario. Cada elemento de esta lista es una instancia de la clase anidada (o separada) InventorySlot.
- **maxInventorySlots (int):** Un campo público serializado que define la capacidad máxima del inventario (el número de slots distintos que puede contener). La clase InventorySlot está diseñada para representar un único slot en el inventario y contiene:

- **item (ItemData):** Una referencia al ScriptableObject ItemData del objeto que se encuentra en ese slot. Si el slot está vacío, esta referencia es null.
- **quantity (int):** La cantidad de unidades del objeto item que hay en ese slot. Si el objeto no es apilable, esta cantidad suele ser 1.

Habilidades (AbilityData):

De forma similar a los objetos, las habilidades se definen mediante ScriptableObjects derivados de una clase base AbilityData. Esta estructura permite crear y configurar diversas habilidades como assets reutilizables en el editor de Unity. Cada AbilityData contiene:

- **Información Identificativa:** abilityID (string) para un identificador único, abilityName (string) para el nombre visible al jugador, icon (Sprite) para su representación visual, y description (string) para un texto explicativo.
- **Costes y Requisitos:** mpCost (int) para el coste de Puntos de Maná necesarios para ejecutar la habilidad.
- **Efectos y Objetivos:**
    - **effectType (enum AbilityEffectType):** Define la categoría principal del efecto de la habilidad (ej: Damage, Heal, Buff, Debuff).
    - **targetType (enum AbilityTargetType):** Especifica a quién o quiénes se puede aplicar la habilidad (ej: SingleEnemy, AllAllies, Self).
- **power (float):** Un valor numérico que representa la potencia base de la habilidad, utilizado en los cálculos de daño, curación, o la magnitud de un buff/debuff.

- **Presentación:** Campos para referenciar animaciones o efectos visuales/sonoros asociados a la habilidad (casterAnimationTrigger, vfxPrefab, sfxClip).
- **Lógica de Ejecución:** Un método virtual ExecuteEffect(Character caster, List<Character> targets) que define la lógica base de cómo la habilidad afecta a los objetivos, incluyendo la deducción del coste de MP del lanzador y la aplicación del efecto principal según su effectType y power. Esta lógica será invocada principalmente por el sistema de combate. **2.**

Implementación de Módulos de Juego (Proceso de Desarrollo)

2.1. Sistema de Movimiento del Jugador y Control de Cámara:

- **Funcionalidad:** Este módulo permite el control del personaje principal en el mapa de exploración, con movimiento en cuatro direcciones (WASD) y animaciones de caminar y reposo (idle) correspondientes a cada dirección.

Una cámara fluida sigue al jugador, restringida a los límites del mapa actual.

- **Implementación Técnica:**

El script “PlayerMovement.cs”, adjunto al jugador, gestiona el control utilizando el Input System de Unity, con una acción “Move” mapeada a WASD que invoca el método “OnMove”. Este método procesa la entrada (InputValue) para determinar la dirección de movimiento cardinal (“movementDirection”) y la orientación del personaje (“lastDirectionString”, “LastFacingVector”). El desplazamiento físico se aplica al Rigidbody2D del jugador mediante la propiedad velocity en “FixedUpdate()”, usando una velocidad (moveSpeed) configurable para asegurar consistencia. Las animaciones (ej: "WalkingDown", "IdleRight") son gestionadas por un Animator Controller y activadas desde el método “HandleAnimations()” en

"PlayerMovement.cs". Para la cámara, se utiliza Cinemachine: una "CinemachineVirtualCamera" sigue al jugador y un "CinemachineConfiner2D", asociado a un "PolygonCollider2D" que define los límites del mapa, restringe el movimiento de la cámara.

- **Problemas Encontrados y Soluciones:**
  - **Problema**: Se detectó vibración en la cámara al seguir al personaje pixel art.
    - **Solución**: Este problema se solucionó ajustando los parámetros de "Damping" en la "CinemachineVirtualCamera" y mediante la adición y configuración del componente "Pixel Perfect Camera" de Unity.

2.2. Comportamiento de Personajes No Jugadores (NPCs):

- **Funcionalidad:** Los NPCs aportan vida, inmersión e interacción al mundo del juego. Para este prototipo, se han implementado comportamientos básicos como: movimiento de patrulla (desplazamiento definido por una dirección y distancia desde un punto inicial), un estado de reposo ("idle") dinámico para NPCs estáticos (que cambian su orientación periódicamente), y la capacidad de orientarse hacia el jugador al iniciar un diálogo.
- **Implementación Técnica:**
  - **Movimiento de Patrulla (NPCMovement.cs):** Este script gestiona a los NPCs que se mueven. El NPC realiza una patrulla A-B: se desplaza una distancia ("moveDistance") desde su posición inicial ("startPosition") en una dirección configurable ("patrolDirection") y luego regresa, repitiendo el ciclo. El movimiento se implementa mediante corrutinas ("MovePattern" y "MoveToPosition"), donde "MovePattern" organiza el ciclo y "MoveToPosition" gestiona el

desplazamiento físico usando ""Rigidbody2D.MovePosition()"" (cuya nueva posición se calcula progresivamente con ""Vector3.MoveTowards()""). Las animaciones ("Walking" + Dirección, "Idle" + Dirección) se controlan con el método ""HandleAnimations()"". Se utiliza ""Physics2D.Raycast"" para una detección simple de obstáculos, deteniendo al NPC si detecta al jugador (con tag ""Player"") en su camino.

- **Idle Dinámico (DynamicIdleBehavior.cs):** Para NPCs estáticos, este script varía aleatoriamente su dirección de "idle" (arriba, abajo, izquierda, derecha) tras un intervalo de tiempo configurable, actualizando su "Animator Controller".
- **Pose Inicial Fija (SetInitialAnimationState.cs):** Este script establece un estado de animación predefinido en el ""Animator Controller"" del NPC en su método ""Start()"", útil para personajes que deben iniciar en una pose específica.
- **Orientación hacia el Jugador:** La lógica para que el NPC se gire hacia el jugador al iniciar un diálogo está integrada en el script NPCDialogue.cs.

- **Problemas Encontrados y Soluciones:**
  - **Problema:** Problema con la detección de jugador de los npcs, no se detenían ante el jugador, el componente raycast para detección de jugador interfería con el propio collider del npc.
    - **Solución:** Se ajustaron los raycast de los npc para que este mas separado de los colliders.

2.3. Sistema de Diálogo Interactivo:

- **Funcionalidad:** Este sistema permite al jugador interactuar con los NPCs para obtener información, avanzar en la historia o recibir misiones. Presenta el

texto de forma legible y coherente con la estética pixel art, incluyendo características como el efecto de "máquina de escribir" y la paginación del texto.

- **Implementación Técnica:**
    - La lógica principal reside en el script ""NPCDialogue.cs"", adjunto a cada NPC interactivo. Este script almacena las líneas de diálogo (un array de strings) y gestiona el estado de la conversación.
    - Se utiliza una interfaz de usuario (UI) dedicada, compuesta por un panel de fondo, un componente ""TextMeshProUGUI"" para mostrar el texto, y una imagen para el indicador visual de continuación ('v'). ○ El efecto de "máquina de escribir" se logra mediante una corrutina que añade caracteres progresivamente al ""TextMeshProUGUI"", con un retardo configurable entre ellos y un sonido de tipeo asociado. ○ La paginación del texto, para diálogos extensos, se maneja a través del método ""GetSegmentThatFits()"", el cual calcula la porción de texto que cabe en el área visible de la UI, procurando no dividir palabras.
    - El indicador visual 'v' se anima para señalar al jugador que puede avanzar una vez que la página de texto actual se ha mostrado por completo.
    - Durante las secuencias de diálogo, los scripts ""PlayerMovement"" (del jugador) y ""NPCMovement"" (del NPC interlocutor) son pausados para evitar movimientos. Además, el NPC se orienta para encarar al jugador.
    - La interacción para iniciar o avanzar el diálogo se gestiona desde el script ""PlayerInteraction.cs"", que detecta al NPC cercano y, al presionar la tecla de interacción, invoca los métodos públicos correspondientes en ""NPCDialogue.cs"".

2.4. Sistema de Inventario y Objetos:

- **Funcionalidad:** Este módulo permite al jugador recoger, almacenar y utilizar una variedad de objetos, incluyendo equipamiento, consumibles y objetos de misión. El inventario posee una capacidad limitada y soporta el apilamiento de ítems. Se proporciona una interfaz de usuario específica para su visualización y gestión. • **Implementación Técnica:**
    - **Definición de Objetos (""ItemData.cs""):** Los objetos se definen mediante ""ScriptableObject"" basados en la clase ""ItemData.cs"", lo que permite crear y configurar sus propiedades (nombre, icono, tipo, apilamiento, efectos, etc.) como assets directamente en el editor de Unity.
    - **Gestión del Inventario (""PlayerInventory.cs""):** Un script Singleton, ""PlayerInventory.cs"", maneja la lógica central del inventario. Este contiene una lista de ""InventorySlot"" (una clase que contiene ""ItemData"" y una ""quantity""). Proporciona métodos para añadir objetos (gestionando el apilamiento en slots existentes o buscando nuevos slots vacíos), quitar objetos y verificar cantidades. El inventario tiene una capacidad máxima de slots (""maxInventorySlots"") y dispara un evento ""OnInventoryChanged"" cuando su contenido se modifica, permitiendo que otros sistemas reaccionen.
    - **Interfaz de Usuario del Inventario (""InventoryUIManager.cs"" y ""InventorySlotUI.cs""):**
        - El Singleton ""InventoryUIManager.cs"" controla el panel principal del inventario y se suscribe al evento ""OnInventoryChanged"" de ""PlayerInventory.cs"" para mantener la interfaz actualizada.

- Cada slot visual en la UI es un prefab con el script ""InventorySlotUI.cs"", responsable de mostrar el icono y la cantidad del objeto, además de gestionar el evento de clic para la selección del ítem.
- Al seleccionar un objeto, se despliega un panel de información y acciones (""ItemInfoActionPanel"") que muestra detalles del ítem y ofrece botones contextuales como ""Usar"", ""Equipar"" (si aplica), ""Tirar"", etc.

- **Problemas Encontrados y Soluciones:**
  - Problema: Al colocar objetos equipable en las ranuras de la izquierda, si en la ranura de después apilas consumibles, convierte los item equipables en consumibles, pero no se visualiza.

2.5. Sistema de Equipamiento y Pantalla de Equipo:

- **Funcionalidad:** Este sistema permite al jugador equipar y desequipar objetos en los personajes de su party, visualizando el impacto en sus estadísticas. La "Pantalla de Equipamiento" centraliza esta gestión, mostrando el personaje seleccionado, sus ranuras de equipo, el equipo actual, y el inventario del jugador para seleccionar nuevos objetos. También permite cambiar entre los miembros de la party.
- **Implementación Técnica:**
  - La clase Character.cs fue extendida para incluir un diccionario equippedItems (que mapea EquipmentSlot a ItemData) y propiedades públicas que calculan los stats totales del personaje (ej: Attack, Defense) sumando los stats base y los bonos del equipo. Se implementaron los métodos EquipItem() y UnequipItem(), que manejan la lógica de añadir/quitar objetos del diccionario

equippedItems y de interactuar con el PlayerInventory para devolver el objeto previamente equipado.

- El script EquipmentScreenManager.cs (Singleton) gestiona la UI de la pantalla de equipamiento. Se encarga de:
  - Mostrar la información del personaje actualmente seleccionado (_currentlyDisplayedCharacter), incluyendo su sprite, nombre, nivel y stats actualizados.
  - Poblar y actualizar los slots visuales de equipo del personaje (utilizando instancias de un prefab con el script CharacterEquipmentSlotUI.cs) con los iconos de los objetos equipados.
  - Mostrar una vista del inventario del jugador (PlayerInventory) utilizando instancias del prefab InventorySlotUI_Prefab, permitiendo la selección de objetos.
  - Gestionar la selección de personajes de la party a través de iconos (instancias de PartyMemberSelectIcon_Prefab con el script PartyMemberSelectIconUI.cs) poblados desde PartyManager.Instance.
  - Mostrar un panel de información y acciones (itemInfoActionPanel) contextual al seleccionar un objeto del inventario (con opción "Equipar") o un objeto ya equipado (con opción "Desequipar").
  - Implementar la lógica de los botones "Equipar" (llamando a PerformEquipAction) y "Desequipar" (llamando a Character.UnequipItem).
- El script CharacterEquipmentSlotUI.cs se adjunta a cada slot visual de equipo del personaje, muestra el icono del objeto equipado y maneja los clics para notificar al EquipmentScreenManager.

- El script PartyMemberSelectIconUI.cs se usa para cada icono de personaje en la barra de selección, llamando a PartyManager.Instance.SetSelectedMenuCharacter() al ser clickeado. EquipmentScreenManager se suscribe a eventos del PartyManager para actualizar el personaje mostrado.

- **Problemas Encontrados y Soluciones:**

7.2.6. Gestión de Party (Básico):

- **Funcionalidad:** El sistema de gestión de party permite definir los miembros que componen el equipo del jugador y seleccionar cuál de ellos está activo para la visualización en menús como la pantalla de equipamiento o de estado.
- **Implementación Técnica:**
  - Se desarrolló el script PartyManager.cs implementando el patrón Singleton y configurado para persistir entre escenas mediante DontDestroyOnLoad(gameObject).
  - Contiene una lista serializada initialPartyMemberGameObjects para asignar los personajes iniciales desde el Inspector de Unity. En Awake(), estos GameObjects se procesan para obtener sus componentes Character y se almacenan en una lista privada _currentPartyMembers.
  - Se implementó un método público AddCharacterToParty(Character newMember) que permite añadir nuevos personajes a la lista _currentPartyMembers durante el juego (simulando reclutamiento por eventos de la historia), evitando duplicados. Este método dispara el evento OnPartyRosterChanged.
  - Se mantiene una referencia al _selectedMenuCharacter, que es el personaje actualmente seleccionado para ser visualizado en las

pantallas de gestión. Este se puede cambiar mediante el método público SetSelectedMenuCharacter(Character character), el cual dispara el evento OnSelectedMenuCharacterChanged solo si el personaje seleccionado cambia y pertenece a la party. ○ Se proporcionan propiedades públicas de solo lectura (CurrentPartyMembers, SelectedMenuCharacter) para que otros sistemas puedan acceder a esta información.

- **Problemas Encontrados y Soluciones:**
  - **Problema***:* Coordinar la actualización de las diferentes pantallas de UI (Equipamiento, Estado) cuando el personaje seleccionado en el PartyManager cambiaba.
    - **Solución***:* Se implementó el evento OnSelectedMenuCharacterChanged en PartyManager, y los scripts EquipmentScreenManager y CharacterStatsScreenManager se suscribieron a este evento para llamar a sus respectivos métodos de actualización de UI (SelectCharacterForDisplay y ShowScreen).

7.2.7. Pantalla de Estado/Datos del Personaje:

- **Funcionalidad:** Esta pantalla permite al jugador visualizar información detallada del personaje actualmente seleccionado, incluyendo su sprite, nombre, nivel, experiencia (actual y para el siguiente nivel, con una barra de progreso), una lista completa de sus estadísticas (HP, MP, Ataque, Defensa, etc., ya considerando los bonos de equipo), y una lista de sus habilidades conocidas con sus descripciones. Se implementó la lógica para que la pantalla se actualice si el personaje sube de nivel mientras está visible.
- **Implementación Técnica:**

- Se creó el script CharacterStatsScreenManager.cs (Singleton) para gestionar esta pantalla. Este script se encarga de mostrar/ocultar el panel principal (CharacterStatsScreen_Panel) y de poblar todos sus elementos de UI.
- El método público ShowScreen(Character characterToShow) recibe el personaje a mostrar, lo almacena en _currentlyDisplayedCharacter, activa el panel y llama a UpdateAllCharacterInfo().
- UpdateAllCharacterInfo() actualiza los TextMeshProUGUI para el nombre, nivel, XP actual y XP para el siguiente nivel. También actualiza un Slider (xpProgressBar_StatsScreen) para representar visualmente el progreso de la XP. Llama a UpdateDetailedStatsDisplay() y PopulateAbilitiesList().
- UpdateDetailedStatsDisplay() actualiza los TextMeshProUGUI individuales para cada estadística (HP, MP, Ataque, Defensa, etc.) obteniendo los valores de las propiedades calculadas del Character (ej: _currentlyDisplayedCharacter.MaxHP, _currentlyDisplayedCharacter.Attack).
- PopulateAbilitiesList() limpia la lista visual de habilidades en el ScrollView (abilitiesListContainer_StatsScreen), itera por la lista _currentlyDisplayedCharacter.knownAbilities, e instancia un prefab AbilityListItem_UI_Prefab para cada habilidad. Llama al método SetupAbilityItem() del script AbilityListItem_UI.cs (adjunto al prefab) para pasarle los datos de la habilidad.
- El script AbilityListItem_UI.cs muestra el nombre de la habilidad y, al ser clickeado, llama al método OnAbilityListItemClicked(AbilityData selectedAbility) del CharacterStatsScreenManager.
- OnAbilityListItemClicked() actualiza un TextMeshProUGUI (abilityDescriptionText_StatsScreen) con la descripción detallada de la habilidad seleccionada.

- El CharacterStatsScreenManager se suscribe al evento estático Character.OnCharacterLevelUp. Cuando este evento se dispara y el personaje que subió de nivel es el que se está mostrando, se llama a UpdateAllCharacterInfo() para refrescar toda la pantalla.

- **Problemas Encontrados y Soluciones:**
  - **Problema***:* Inicialmente, el script Character.cs no tenía campos para currentXP o experienceToNextLevel, lo que impedía mostrar esta información.
    - **Solución***:* Se añadieron los campos necesarios a Character.cs y se implementó la lógica básica de ganancia de XP y subida de nivel.
  - **Problema***:* La barra de progreso de XP no se actualizaba correctamente o no reflejaba el porcentaje adecuado.
    - **Solución***:* Se verificó la fórmula de cálculo del value del Slider ((float)currentXP / experienceToNextLevel) y se aseguró que experienceToNextLevel no fuera cero para evitar divisiones por cero.
  - **Problema***:* La descripción de la habilidad no se mostraba o no se actualizaba al seleccionar diferentes habilidades en la lista.
    - **Solución***:* Se depuró la conexión entre el evento OnClick del AbilityListItem_UI y el método OnAbilityListItemClicked en CharacterStatsScreenManager, asegurando que la referencia a abilityDescriptionText_StatsScreen estuviera correctamente asignada y que el texto se actualizara con la información de la habilidad seleccionada.

Conclusiones

En conclusión, el desarrollo del proyecto A Hero's Resolve ha permitido al equipo aplicar de forma práctica los conocimientos adquiridos durante el ciclo formativo de Desarrollo de Aplicaciones Multiplataforma, así como habilidades obtenidas mediante el autoaprendizaje. A lo largo del proceso, se han implementado los sistemas fundamentales de un videojuego RPG por turnos, incluyendo el movimiento del jugador, la gestión del inventario, los diálogos interactivos, el sistema de combate y diversas pantallas de usuario.

Durante las últimas fases del proyecto, se incorporó un entorno interior adicional correspondiente a una posada, lo cual ha supuesto una mejora significativa en la exploración. Este nuevo mapa incluye un diseño detallado, elementos visuales coherentes con la estética general del juego, personajes no jugables con diálogos funcionales, zonas navegables con colisiones ajustadas y transiciones entre escenas. Su integración ha aportado profundidad narrativa y jugable al prototipo final.

A pesar de que algunas funcionalidades previstas quedaron en una fase inicial, el resultado alcanzado demuestra una base técnica sólida y coherente con los objetivos marcados. El equipo ha logrado resolver numerosos desafíos técnicos, adaptarse a nuevas herramientas, gestionar eficazmente el trabajo colaborativo y mantener un enfoque constante en la mejora del producto. Todo ello refleja una evolución significativa en el proceso de desarrollo, ofreciendo un prototipo funcional con posibilidades reales de ampliación futura.

Bibliografía

- TutsPlus. (s.f.). *Equilibrar los RPG basados en turnos: el panorama general*. Recuperado de https://gamedevelopment.tutsplus.com/es/equilibrar-losrpg-basados-en-turnos-el-panorama-general--gamedev-8286a

- Unity Technologies. (2024). *Unity for 2D games*. Recuperado de https://unity.com/es/solutions/2d
- Wikipedia contributors. (2024). *Videojuego de rol*. Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Videojuego_de_rol
- Wikipedia contributors. (2024). *Historia de los videojuegos de rol occidentales*. Wikipedia. Recuperado de https://es.wikipedia.org/wiki/Historia_de_los_videojuegos_de_rol_occidentales
- Punto de Respawn. (2023). *Elementos clave diseño RPG por turnos.* Recuperado de https://puntoderespawn.com/articulos/opinion/combatepor-turnos-un-sistema-clasico-y-estrategico/
- Paradigma Digital .(2023). Experiencias de usuarios en los videojuegos RPG. Recuperado de https://www.paradigmadigital.com/dev/experienciausuario-videojuegos-rpg/