El Desastre Humano

(Projecte de desenvolupament)
CFGS Desenvolupament d'Aplicacions Multiplataforma

Jiayi Chen
DAM2B
2025-2026

Aquesta obra està subjecta a una llicència de
Reconeixement-NoComercial-CompartirIgual 4.0 Espanya de Creative Commons

B) GNU Free Documentation License (GNU FDL)

Copyright © ANY Jiayi Chen

Se autoriza la copia, distribución y/o modificación de este documento bajo los términos de la Licencia de Documentación Libre de GNU, versión 1.3 o cualquier versión posterior publicada por la Free Software Foundation; sin secciones invariantes, sin textos de portada ni de contraportada. Una copia de la licencia se incluye en la sección titulada «Licencia de Documentación Libre de GNU».

C) Copyright

© (Jiayi Chen.)

Reservados todos los derechos. Está prohibido la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la impresión, la reprografía, el microfilme, el tratamiento informático o cualquier otro sistema, así como la distribución de ejemplares mediante alquiler y préstamo, sin la autorización escrita del autor o de los límites que autorice la Ley de Propiedad Intelectual.

ÍNDICE

ÍNDICE 2
FICHA DE TRABAJO FINAL 6
1. Introducción 9
1.1 Contexto y justificación del trabajo 9
1.2 Objetivos del trabajo 9
1.2.1 Objetivos principales 9
1.2.2 Objetivos secundarios o alternativos 10
1.3 Enfoque y método seguido 11
1.4 Metodología de trabajo 12
1.5 Planificación del trabajo 12
2. Herramientas de desarrollo y 15
2.1 Motor de videojuego 15
2.1.1 Godot Engine 15
2.1.2 GD Script 16
2.2 Blender 16
2.3 Backend 17
2.3.1 Backend de juego 17
2.3.2 Backend de comunidad 18
2.3.3 Base de datos y diseño 19
2.4 Frontend de comunidad 19
2.5 Herramientas de desarrollo y pruebas 19
2.6 Control y arquitectura del proyecto 19
2.7 ¿Por qué utilizo Godot, Blender y otras herramientas? 20
2.8 Lista de otras herramientas que he utilizado 21
2.9 Recursos que utilizamos 24
3. Descripción del juego 24
3.1 El diseño de juegos 24
3.2 Jugabilidad y objetivos 25
3.3 Público objetivo 25
3.4 Conceptualización de ideas y diseños 26
3.4.1 Contexto y trama del juego 26
3.4.2 Los protagonistas 27
(1) Introducción al protagonista unificado elegido por los jugadores 27
(2) Sistema de niveles 28
3.4.3 Los mapas 28
(1) Escenarios principales 28
(2) Escenario de práctica 29
(3) Escenario de tutorial 29
3.4.4 La trama principal y las misiones secundarias 29
3.4.5 Los enemigos y los NPCs 30
3.4.6 El combate 30
3.4.7 El inventario y las transacciones monetarias 31

3.4.8 El multijugador en LAN........31
3.4.9 Sistema de guardado de juego y datos estáticos........31
3.4.10 El sistema de progresión del personaje........32
4. Diseño de la aplicación de comunidad de juegos........33
4.1 Contexto general........33
4.2 Funciones principales........33
4.3 Arquitectura del sistema........34
4.4 Diseño........34
4.5 Integración básica con el sistema de juego........35
4.6 Seguridad........35
4.7 Planes futuros........35
4.8 Resumen de esta sección........36
5. BackEnd........36
5.1 Postgresql........36
5.2 FastApi........36
5.3 Springboot........37
6. Análisis........38
6.1 Casos de uso........38
6.1.1 Juego........38
6.1.2 Comunidad........39
6.2 Requisitos funcionales........41
6.3 Requisitos no funcionales........42
6.4 Análisis de alternativas tecnológicas........43
6.4.1 Motor de juego........43
6.4.2 Modelado 3D........44
6.4.3 Backend........44
6.4.4 Base de datos........45
6.4.5 Frontend........46
6.5 Análisis de usuarios y roles........46
7. La producción artística........47
7.1 Modelo........47
7.1.1 Creación de personajes........47
7.1.2 Escena........48
7.2 SFX........49
7.3 AI........50
8. Proceso de desarrollo e implementación........51
8.1 juego........51
8.1.0 Entrada del proyecto y arquitectura Autoload........51
8.1.1 Sistema de tutorial........51
8.1.1.1 Planificación del proceso tutorial........51
8.1.1.2 Sistema de bloqueo........53
8.1.2 Comportamiento hostil e inteligencia artificial........53
8.1.2.1 Comportamiento del personaje y máquina de estados........53
8.1.2.2 IA de enemigos........56

8.1.3 Cambio de escena.. 59
8.1.3.1 Activación y transición...59
8.1.3.2 Checkpoints y estado de escena...60
8.1.3.3 Rollback y consistencia.. 60
8.1.3.4 Implementación en el proyecto...61
8.1.4 Menú e interfaz de usuario..61
8.1.4.1 Interfaz de información del personaje.. 62
8.1.4.2 Inventario...64
8.1.4.3 Interfaz de inicio de sesión...65
8.1.4.4 Interfaz de combate...66
8.1.4.5 Teclado y ratón virtuales...66
8.1.4.6 Pantalla de carga...67
8.1.4.7 Ajustes, pausa y popups... 67
8.1.4.8 Sistema global de mensajes...68
8.1.4.9 Gestión y adaptación de UI... 68
8.1.4.10 Adaptación de resolución.. 69
8.1.5 Audio y efectos de sonido.. 70
8.1.5.1 Sonidos del personaje...71
8.1.5.2 Sonidos de habilidades... 71
8.1.5.3 Sonidos de armas...71
8.1.5.4 Música de fondo y mezcla...72
8.1.6 Sistema de habilidades.. 73
8.1.6.1 Sistema de impacto...73
8.1.6.2 Arquitectura de datos y ejecución.. 74
8.1.6.3 Estructura de datos y mapeo..74
8.1.6.4 Arquitectura del sistema...75
8.1.7 Sistema de inventario...77
Implementación en el proyecto...77
8.1.8 Sistema de armas.. 77
Implementación en el proyecto...77
8.1.9 Sistema genético...78
8.1.9.1 Efectos y relación con otros sistemas.. 79
8.1.9.2 Mecanismos de evolución... 79
8.1.9.3 Flujo de desbloqueo y activación.. 79
8.1.9.4 Implementación y consistencia...79
8.1.9.5 Balance y control del sistema...80
8.1.10 Sistema de guardado... 80
8.1.10.1 Almacenamiento local... 80
8.1.10.2 Sincronización con la nube...80
8.1.10.3 Control de versiones y consistencia..81
8.1.10.4 Compatibilidad y evolución...81
8.1.10.5 Implementación y flujo del sistema...81
8.1.10.6 Separación de responsabilidades de datos..81
8.1.11 Sistema de usuario...82

8.1.12 Combate y daño......83
8.1.12.1 Flujo de resolución......83
8.1.12.2 Modelo de daño......83
8.1.12.3 Arquitectura basada en eventos......83
8.1.12.4 Feedback de combate......84
8.1.12.5 Casos límite......84
8.1.13 Clases y estados......84
8.1.13.1 Sistema de clases......84
8.1.13.2 Sistema de estados......85
8.1.13.3 Reglas clave del sistema de estados......85
8.1.13.4 Diseño técnico......85
8.1.14 Sistema de atributos......85
8.1.14.1 Estructura del sistema......85
8.1.14.12 Flujo de cálculo......86
(3) Reglas de acumulación......86
(4) Sistema de crecimiento......86
(5) Balance del sistema......86
(6) Integración con el sistema de combate......87
8.2 Herramientas de depuración y pruebas......87
8.2.1 Requisitos y arquitectura de la información......87
8.2.2 Estructura del frontend y organización de rutas......87
8.2.3 Encapsulación de la capa de red y estrategia de alineación de API......88
8.2.4 Autenticación y gestión de sesión......90
8.2.5 Feed y mecanismos de interacción......91
8.2.6 Subida de imágenes y gestión de recursos multimedia......92
8.2.7 Notificaciones y perfil de usuario......93
8.2.8 Construcción y despliegue (Web / Capacitor)......93
8.3 Backend y base de datos (FastAPI + Spring Boot + PostgreSQL)......93
8.3.1 Backend del juego (FastAPI): organización de API y arquitectura basada en datos......93
8.3.2 Consistencia de guardado: sincronización local y en la nube......94
8.3.3 Backend de comunidad (Spring Boot): arquitectura y reglas de seguridad......95
8.3.4 Diseño de base de datos: separación por esquemas y modelo relacional......96
8.3.5 Migración y estrategia de inicialización......96
8.4 Partes transversales del sistema (testing, depuración, optimización, despliegue y gestión de versiones)......97
8.4.1 Estrategia de pruebas (unitarias, integración y contrato)......97
8.4.1 Estrategia de pruebas (unitarias, integración y contractuales)......97
8.4.2 Depuración y mecanismos de diagnóstico......98
8.4.3 Optimización de rendimiento y compatibilidad......98
8.4.4 Seguridad y control de acceso......99
8.4.5 Construcción, despliegue y gestión de versiones......99
9.2 Proceso de implementación en Blender......99
9.2.1 Modelado......99

9.2.2 Esculpido.....101
9.2.3 Sombreado.....102
9.2.4 Modificador.....102
9.2.5 UV.....103
9.2.6 Paint.....104
10. Pruebas y ejecución.....105
10.1 Metodología de pruebas y selección de herramientas.....105
10.2 Motivos de selección de herramientas.....106
10.3 Resumen de archivos de prueba.....107
10.3.1 FastAPI.....107
10.3.2 Springboot.....107
10.3.3 Juego y la aplicación de comunidad.....108
10.3.3.1 Cliente de juego（Godot）.....108
10.3.3.2 Aplicación de comunidad.....109
11. Riesgos, retos, innovaciones y perspectivas de la investigación.....110
11.1 Riesgos que se pueden encontrar en el proyecto.....110
11.2 Desafíos y soluciones de la investigación.....110
11.3 Puntos de Innovación.....112
11.3.1. Coordinación por dominios en un sistema multi-backend heterogéneo.....112
11.3.2. Arquitectura de capacidades transversales basada en Autoload (Godot).....112
11.3.3. Estrategia de consistencia: “guardado local cifrado + persistencia en la nube”. 113
11.3.4. Sistema de datos estáticos dirigido por backend con fallback local.....113
11.3.5. Sistema de habilidades con mapeo de tres niveles.....113
11.3.6. Sistema genético modular con restricciones consistentes.....113
11.3.7. Sistema de enemigos basado en etiquetas (tag-driven design).....114
11.3.8. Modelo de seguridad de mínimos privilegios en la comunidad.....114
11.3.9. La testabilidad como principio transversal del sistema.....114
11.4 Limitaciones.....115
11.5 Trabajo futuro y evolución del sistema.....117
11.5.1 Proyecto Macro.....117
11.5.2 Orientado a juego.....118
12. Conclusión.....120
13. Bibliografía y referencias.....121
Blender:.....121
Godot:.....121
Python:.....122
Java:.....122
Postgres:.....123
Otros:.....123
14. Anexos.....124
[1] Narrador del inicio.....124
[2] Escena de carga.....124
[3] Nuestro web.....124

[4] Lista de abreviaturas y glosario de términos....124
[5] Seguiment Setmanal....124
[6] Análisis....124
[7] Diagrama....124

FICHA DE TRABAJO FINAL

Título	El Desastre Humano (en inglés Human Disaster)
Autores	Jiayi Chen
Idioma del trabajo	Castellano
Palabras claves	Godot, Blender, 3D, Videojuego, 1ra y 3rd persona, Mundo libre, Narrativo, FastAPI, PostgreSQL, Spring Boot, Archivo en la nube, Separación de front-end/back-end, Arquitectura en capas, Ionic, React, JWT, Vite,

Resumen del proyecto

Este artículo se centra en el proyecto de juego Desastre Humano, describiendo una solución de implementación de ingeniería desde el diseño de la arquitectura general hasta el despliegue de subsistemas, haciendo hincapié en el desacoplamiento del sistema, la consistencia de los datos y la verificabilidad.

El proyecto emplea una arquitectura desacoplada de múltiples backends para separar las responsabilidades de la lógica del juego y los sistemas de la comunidad:

- Cliente del juego: Desarrollado con Godot 4, utiliza un singleton de carga automática para gestionar el estado global, lo que permite guardar localmente encriptado y sincronizar datos en la nube. Además, adopta un diseño orientado a recursos para garantizar la consistencia entre la estructura de datos estática del cliente y el modelo JSON del backend.
- Backend del juego: Basado en FastAPI + PostgreSQL, gestiona los datos de personajes, guardados y datos principales del juego, utilizando JSONB para mejorar la escalabilidad de la estructura de datos y la compatibilidad de versiones.
- Backend de la comunidad: Desplegado de forma independiente con Spring Boot, proporciona funciones sociales (como interacción y publicación de contenido), completamente desacoplado del servicio del juego.
- Interfaz móvil: Utiliza Ionic + React para lograr la interacción multiplataforma de las funciones de la comunidad.
- La capa de base de datos emplea múltiples esquemas (juego/comunidad) para el aislamiento de dominios. El cliente solo accede a la interfaz del servicio del juego, lo que garantiza la independencia y estabilidad de la jugabilidad y los vínculos sociales desde una perspectiva arquitectónica.

En comparación con la versión anterior, este proyecto ha experimentado una optimización sistemática en la arquitectura del sistema y la implementación funcional:

- Actualización de la arquitectura: Se ha evolucionado de un sistema único a una arquitectura de doble backend de "juego + comunidad", lo que mejora la escalabilidad del sistema y la independencia de los módulos.
- Mejora basada en datos: Se han unificado los recursos del cliente y la estructura de datos JSON del backend, reduciendo los riesgos derivados de las

inconsistencias de versiones.

- Optimización del mecanismo de archivado: Se ha implementado una solución colaborativa de "almacenamiento rápido local + almacenamiento en la nube", introduciendo estrategias de control de versiones y prevención de conflictos para mejorar la fiabilidad de los datos.
- Mejora de las prácticas de ingeniería: Se han introducido pruebas de contrato de interfaz (pytest + MockMvc), lo que mejora la verificabilidad y la estabilidad de la colaboración entre el frontend y el backend. Expansión del sistema de juego: Se añadieron múltiples escenas y mecanismos de juego, mejorando la división de módulos del sistema.
- Refinamiento del contenido: Se añadieron personajes y se realizó un modelado de mayor precisión, mejorando la expresividad general y la exhaustividad del diseño.

Esta versión no solo completó la expansión funcional, sino que también logró mejoras en los siguientes aspectos:

- La arquitectura evolucionó de "ejecutable" a "escalable y mantenible".
- La práctica de comparación de pilas tecnológicas es más completa (Godot / FastAPI / Spring Boot / Ionic + React).
- Capacidades de ingeniería mejoradas (pruebas, control de versiones, consistencia de datos).
- Proporciona una base sólida para futuras implementaciones multijugador, en línea o comerciales.

Project abstract

This article focuses on the Desastre Humano game project, outlining an engineering implementation solution from overall architecture design to subsystem deployment, emphasizing system decoupling, data consistency, and verifiability.

The project employs a multi-backend decoupled architecture to separate the responsibilities of game logic and community systems:

- Game Client: Developed based on Godot 4, it uses an Autoload singleton to manage global state, enabling local encrypted saves and cloud data synchronization; it also adopts a resource-driven design to ensure consistency between the client's static data structure and the backend JSON model.
- Game Backend: Based on FastAPI + PostgreSQL, it manages character, save, and core gameplay data, utilizing JSONB to enhance data structure scalability and version compatibility.
- Community Backend: Independently deployed based on Spring Boot, it provides social functions (such as interaction and content publishing), completely decoupled from the game service.
- Mobile Frontend: Utilizes Ionic + React to achieve cross-platform interaction of community functions.
- The database layer uses multiple schemas (game / community) for domain isolation. The client only accesses the game service interface, ensuring the independence and stability of the gameplay and social links from an architectural perspective.

Compared to the previous version, this project has undergone systematic optimization in system architecture and functionality:

- Architecture Upgrade: Evolved from a single system to a dual-backend architecture of "game + community," improving system scalability and module independence.
- Data-Driven Enhancement: Unified client resources and backend JSON data structure, reducing the risks associated with version inconsistencies.
- Archiving Mechanism Optimization: Implemented a collaborative solution of "local fast storage + cloud storage," introducing version control and conflict avoidance strategies to improve data reliability.
- Engineering Practice Improvement: Introduced interface contract testing (pytest + MockMvc), improving the verifiability and stability of front-end and back-end collaboration.
- Game System Expansion: Added multiple game scenes and gameplay mechanisms, improving system module division.
- Content Refinement: Added characters and performed higher-precision modeling, improving overall expressiveness and design completeness.

This version not only completed functional expansion but also achieved improvements in the following aspects:

- Architecture evolved from "runnable" to "scalable and maintainable."
- Technology stack comparison practice is more complete (Godot / FastAPI / Spring Boot / Ionic + React
- Enhanced engineering capabilities (testing, version control, data consistency)
- Provide a solid foundation for future multiplayer, online, or commercial applications.

1. Introducción

1.1 Contexto y justificación del trabajo

En la versión anterior del proyecto, completamos la construcción del sitio web y la lógica básica del juego. En esta versión, el juego independiente cuenta con un cliente jugable, seguimiento persistente del progreso y un diagrama de arquitectura demostrable. Este proyecto utilizará Godot para la interacción en tiempo real, persistencia relacional PostgreSQL y una API ligera: FastAPI. También incluirá un backend basado en la comunidad (Spring Boot) para demostrar la arquitectura por capas y la coexistencia de múltiples aplicaciones, facilitando la comparación entre la API de guardado de juegos y la API de redes sociales.

Paralelamente, continuaré utilizando Blender para el modelado. Gracias a nuestra experiencia acumulada y al uso de la asistencia de IA, el progreso en esta versión se ha acelerado significativamente, y la resolución e identificación de problemas son mucho más sencillas.

1.2 Objetivos del trabajo

1.2.1 Objetivos principales

En este proyecto, los principales objetivos son los siguientes:

- Para garantizar el correcto funcionamiento del juego, se pretende implementar un sistema backend completo junto con un mecanismo de almacenamiento de datos. Esto permitirá que el juego no solo se ejecute de forma local, sino también que pueda leer datos desde el servidor, mientras que los datos del jugador se almacenan de manera estable en una base de datos.

- En cuanto al almacenamiento de datos, se plantea implementar un mecanismo híbrido entre almacenamiento local y en la nube. El sistema priorizará los datos según su importancia: aquellos con menores requisitos de seguridad y que requieran acceso rápido se almacenarán localmente para mejorar la velocidad de carga; mientras que los datos críticos, como el progreso del personaje y el estado de las llaves, se almacenarán de forma centralizada en la base de datos del backend, garantizando la integridad de los datos y permitiendo su recuperación entre distintos dispositivos.

- A nivel estructural, el sistema se dividirá en dos partes: un backend del juego y un backend de la comunidad, cada uno con funciones diferenciadas. El backend del juego se encargará principalmente de la lógica del juego y la gestión de datos, mientras que el backend de la comunidad gestionará las funciones sociales. Ambos backends operarán de forma independiente, reduciendo el acoplamiento y facilitando futuras ampliaciones.

- Además, para garantizar la fiabilidad del sistema, se prevé implementar procesos de prueba para las funciones principales, incluyendo pruebas de interfaz y pruebas de módulos, con el objetivo de verificar la funcionalidad y mejorar la estabilidad y mantenibilidad del sistema.

- En cuanto al despliegue, los servicios backend se ejecutarán en un entorno de máquina virtual, lo que facilitará una gestión unificada y sentará las bases para una futura migración a un despliegue mediante contenedores Docker, permitiendo un inicio y despliegue más sencillo con un solo comando.

- Respecto al contenido del juego, se planea mejorar varios sistemas clave, como el inventario, las armas, las habilidades, el cálculo de daño, el sistema genético, los tutoriales y la interfaz de usuario, con el fin de ofrecer una experiencia de juego más completa y fluida.

- En el apartado artístico, se contempla la creación y optimización de un nuevo modelo de personaje femenino mediante Blender, con el objetivo de lograr una presentación visual más refinada.

1.2.2 Objetivos secundarios o alternativos

Si hay suficiente tiempo, también hacemos los siguientes objetivos:

- Multijugador: La colaboración multijugador mejorada y las capacidades de sincronización en línea permiten a los jugadores interactuar en tiempo real dentro del mismo mundo de juego, mejorando los aspectos sociales y la jugabilidad general.

- Sistema de guardado: Se optimizó el mecanismo de guardado existente, introduciendo una gestión de versiones más granular y estrategias para resolver conflictos, como la compatibilidad con múltiples ramas de guardado y la función de reversión, mejorando así la seguridad de los datos y la tolerancia a fallos.

- Seguridad: Actualmente, los mecanismos antitrampas y de seguridad en las capas del juego y la aplicación se encuentran en sus etapas iniciales. Los planes futuros incluyen la mejora gradual de las estrategias pertinentes para aumentar la estabilidad y la seguridad del sistema en entornos de red.

- Arte y presentación: Se optimizó unificadamente el sistema de interfaz de usuario para crear un estilo más coherente, mejorando la experiencia de usuario y la fluidez de la interacción; las mejoras adicionales en la calidad de la animación y los efectos especiales aumentan el atractivo visual y la inmersión.

- **Enriquecer la narrativa de nuestro juego:** Las limitaciones de tiempo pueden afectar el desarrollo de una trama completa, sin embargo, pretendemos enriquecer y optimizar constantemente la historia, permitiendo que evolucione según las elecciones del jugador y conduzca a múltiples finales posibles.

- **Aumentar la cantidad de personajes y enemigos en el juego:** Nos gustaría incluir una mayor variedad de personajes y enemigos en nuestro juego.

- **Expandir nuestro mapa:** Nos esforzaremos por desarrollar un mapa de juego más grande y rico, ofreciendo a los jugadores un espacio más amplio para explorar.

- **Desarrollar un plan de marketing y promoción:** Si es posible, crearemos un plan de marketing y promoción, incluyendo publicidad en redes sociales, establecimiento de asociaciones, participación en exposiciones de juegos, entre otros, para aumentar la visibilidad del juego y atraer a jugadores potenciales.

1.3 Enfoque y método seguido

Partiendo de la arquitectura básica y la lógica central de la versión anterior, esta versión actualiza principalmente la arquitectura del sistema y amplía tanto el front-end como el back-end.

En primer lugar, el diseño del sistema adopta un enfoque modular y por capas, desacoplando el cliente del juego, el back-end y el sistema de la comunidad. Esto clarifica las responsabilidades de cada módulo, reduce las dependencias entre sistemas y mejora la mantenibilidad y la escalabilidad generales.

En segundo lugar, la gestión de datos incorpora un enfoque de diseño basado en datos. Una estructura de datos unificada conecta el cliente y el servidor, asegurando que la lógica del juego dependa de la configuración y los datos en lugar de datos codificados, lo que mejora la flexibilidad del sistema y la eficiencia de las iteraciones posteriores.

Para el almacenamiento, se utiliza una combinación de almacenamiento local y en la nube, con los datos procesados jerárquicamente según su importancia: los datos no críticos se almacenan en caché localmente para mejorar la eficiencia de carga y ejecución, mientras que los datos principales se gestionan de forma persistente en el back-end para garantizar la seguridad y la coherencia de los datos.

En el proceso de desarrollo, se adopta un enfoque de diseño centrado en la interfaz. Los contratos de interfaz estandarizan las interacciones entre el front-end y el back-end, y se utilizan métodos de prueba para verificar las funciones clave, reduciendo la incertidumbre durante las pruebas de integración y mejorando la estabilidad general del sistema.

Para el despliegue y el mantenimiento, se utilizan máquinas virtuales para desplegar uniformemente los servicios de back-end, y se reserva una solución de migración a contenedores para mejorar la flexibilidad del despliegue y la expansión posteriores del sistema.

Finalmente, durante el desarrollo, se utilizan herramientas como Blender, IntelliJ IDEA y VS Code para facilitar el modelado, el desarrollo del back-end y el desarrollo del lado del cliente, mejorando la eficiencia general del desarrollo y las capacidades de colaboración en ingeniería.

1.4 Metodología de trabajo

El presente proyecto adopta la metodología de desarrollo ágil como principal enfoque organizativo, con el objetivo de adaptarse a la complejidad del sistema de juego, la elevada cantidad de módulos y la naturaleza iterativa del proceso de desarrollo.

El desarrollo ágil se basa en ciclos cortos de iteración (sprints), en los cuales el conjunto del trabajo se divide en módulos funcionales más pequeños y entregables, tales como el sistema de personajes, el sistema de combate, el sistema de inventario, las interfaces backend y las funcionalidades de comunidad. Cada ciclo de desarrollo incluye fases de análisis de requisitos, diseño, implementación, validación y entrega de funcionalidades, lo que permite una evolución continua del sistema y una retroalimentación rápida ante posibles problemas.

En la práctica, el proyecto se organiza mediante la descomposición de tareas y la gestión de prioridades, diferenciando entre funcionalidades principales (core features) y funcionalidades extendidas (extended features). Las funcionalidades principales, como el control del personaje, el sistema de combate y la persistencia de datos, se implementan en primer lugar, mientras que las funcionalidades adicionales, como el sistema de habilidades, la interacción con NPCs y los módulos de comunidad, se desarrollan en fases posteriores.

Asimismo, mediante la integración continua (CI) y el uso de control de versiones (Git), se garantiza la coordinación estable entre los distintos módulos del sistema. Bajo una arquitectura monorepo, los subsistemas (cliente del juego, servicios backend y sistema de comunidad) pueden desarrollarse de forma independiente, manteniendo al mismo tiempo una gestión unificada de versiones, lo que reduce el acoplamiento entre componentes y mejora la eficiencia del desarrollo.

Finalmente, el desarrollo ágil se caracteriza por un mecanismo de retroalimentación continua, en el cual las pruebas y validaciones iterativas permiten refinar progresivamente el diseño del sistema. Este enfoque facilita la detección temprana de inconsistencias entre sistemas, por ejemplo, posibles conflictos entre el sistema de habilidades y el sistema genético, los cuales pueden ser corregidos y ajustados en iteraciones posteriores, mejorando así la coherencia global del sistema y su estabilidad.

1.5 Planificación del trabajo

Jira es la herramienta principal para la gestión de tareas y la planificación del desarrollo en este proyecto. Sus funciones de cronograma y panel de tareas proporcionan una representación visual clara de todo el proceso iterativo, desde la identificación de requisitos y el diseño del sistema hasta el desarrollo, la implementación y el lanzamiento final.

En este proyecto, mi periodo de desarrollo abarcó desde el 22 de septiembre de 2025 hasta el 17 de mayo de 2026. Dado que era mi primera vez desarrollando una aplicación, la creación de nuevos escenarios, así como la implementación de la lógica y la compatibilidad del sistema de habilidades, presentaron una mayor dificultad, por lo que dediqué más tiempo a estos tres módulos. En cambio, como ya contaba con cierta experiencia en bases de datos, el tiempo invertido en esta parte fue relativamente menor.

Los gráficos de esta sección incluyen principalmente la siguiente información:

- Cronograma y desglose del progreso general para cada fase de desarrollo
- Desglose de tareas para los diferentes módulos (cliente, backend, sistema de la comunidad, etc.)
- Hitos de desarrollo y secuencia de iteraciones para las funcionalidades clave
- Transiciones de estado de las tareas entre los diferentes estados (p. ej., pendiente, en curso, completada)

Archivo de imagen:
https://drive.google.com/file/d/1ZoRxHbXC-ucSaElbqEXtb0NkBByiEFeK/view?usp=sharing

Tarea	Responsable
Investigació	Jiayi Chen
Desenvolupament	Jiayi Chen
Sistema de inventario	Jiayi Chen
Diseño de interfaz de usuario	Jiayi Chen
Lógica de funcionamiento de la mochila	Jiayi Chen
Diseño tipográfico	Jiayi Chen
Estilos de artículos y diseños de patrones	Jiayi Chen
Implementación del código	Jiayi Chen
Prueba	Jiayi Chen
Documento Funcional	Jiayi Chen
Sistema de Usuario	Jiayi Chen
Arquitectura general del sistema de usuar	Jiayi Chen
Módulo de inicio de sesión y registro	Jiayi Chen
Interfaz de usuario del cliente	未分配
Gestión de datos de usuario	未分配
Sistema de personajes y guardado	未分配
Seguridad y autorización	未分配
Interacción con el cliente y encapsulación	未分配
Base de dato	Jiayi Chen
Crear una base de datos	未分配
API	未分配
Establecer una conexión con Godot	未分配
Operaciones CRUD	未分配
Sistema de habilidades	Jiayi Chen
Diseño de habilidades	Jiayi Chen
Diseño del valor de la habilidad	Jiayi Chen
Planificación de acciones y rendimiento	Jiayi Chen
Producción de animación y efectos espec	Jiayi Chen
Implementación técnica	Jiayi Chen
Detección de daños y golpes	Jiayi Chen
Sistema de buffs/debuffs	Jiayi Chen
Efectos de sonido y trabajo de cámara	Jiayi Chen
Optimización del rendimiento	Jiayi Chen
UI	Jiayi Chen
Nuevo mapa	Jiayi Chen
Diseño de mapas	未分配
Diseño del mapa, la visión del mundo y la	未分配
Modelado del terreno y creación del relie	未分配
Ecología y colocación de objetos	未分配
Jugabilidad dentro del mapa	未分配
Renderización de arte y atmósfera	未分配
Pruebas y optimización	未分配
Software para plataformas comentarios	Jiayi Chen
Registro de usuario / Inicio de sesión	未分配
Publicación	未分配
Ver la lista de publicaciones	未分配
Gusta / Comentario	未分配
Información del usuario	未分配
BackEnd	Jiayi Chen
Interfaz	Jiayi Chen
Proves	Jiayi Chen
Defensa	Jiayi Chen

2. Herramientas de desarrollo

2.1 Motor de videojuego

2.1.1 Godot Engine

Godot es un motor de juegos de código abierto y **multiplataforma**, **potente** y **altamente flexible**, que soporta el desarrollo de juegos en **2D** y **3D** y puede ejecutarse en diversos sistemas operativos como **Windows**, **macOS** y **Linux**. Sus funcionalidades avanzadas y su facilidad de uso permiten a los desarrolladores crear juegos visuales y fluidos que pueden ser fácilmente desplegados en múltiples plataformas como **PC**, **Android**, **iOS** y **HTML5**, entre otros.

En lo que se refiere a los lenguajes de programación, Godot ofrece varias opciones como **GD Script**, **C++** o **C#**, así como la capacidad de integrar otros lenguajes mediante **GD Native**, como **C** o **C++**. La comunidad también ofrece soporte para otros lenguajes, como **Rust, Nim, JavaScript, Haskell, Clojure, Swift y D**, ofreciendo a los desarrolladores una gran flexibilidad.

El **motor gráfico 2D y 3D** de Godot funciona de forma independiente y permite la combinación de contenido 2D y 3D mediante nodos de vista. Además ofrece un **lenguaje de shaders** personalizado similar a **GLSL** para la creación de materiales y el renderizado, y proporciona herramientas de renderizado visual para simplificar el proceso.

En cuanto a animación, Godot dispone de un potente sistema de animación que permite a los desarrolladores crear **animaciones de huesos**, **mezclas**, **árboles de animación**, **animaciones de transición** en tiempo real, entre otros. Prácticamente todas las variables de los objetos del juego pueden ser animadas, añadiendo dinamismo e interactividad al juego.

En cuanto al tratamiento de colisiones, Godot permite la creación de diferentes **estructuras de colisión** para distintos objetos, utilizando formas **primitivas** o **convexas** para representarlas, satisfaciendo así las necesidades de colisión de cada objeto.

Además de las funcionalidades mencionadas, Godot incluye la creación de terrenos, efectos de luz y sombra, sistemas de partículas, reproducción de audio, procesamiento de texturas, diseño de interfaces gráficas y soporte para diferentes periféricos, ofreciendo a los desarrolladores una solución integral para el desarrollo de juegos. Tanto los principiantes como los desarrolladores experimentados pueden encontrar en Godot las herramientas adecuadas para crear obras maestras de juegos.

Los plugins:
Complementos: En Godot, podemos añadir varios complementos para mejorar el juego, incluyendo un complemento de inventario personalizado (que genera objetos leyendo archivos JSON como fuente de datos).

Las versiones anteriores también incluían los complementos Terrain3D y Nakama Client, pero estos se eliminaron debido a limitaciones funcionales.

2.1.2 GD Script

En nuestro proyecto, para la escritura de scripts de juegos, hemos escogido el lenguaje de programación GDScript que viene integrado en el entorno de Godot. GDScript es un lenguaje de alto nivel, imperativo y orientado a objetos, cuya sintaxis es similar a la de Python, lo que lo hace fácil de aprender y usar. Esta lengua ha sido específicamente diseñada para integrarse con el entorno de Godot y ofrece una experiencia de desarrollo de juegos eficiente.

La razón principal para elegir GDScript es su alta integración con el entorno de Godot. Esta integración nos permite operar de forma fluida y optimizar el rendimiento, facilitándonos un desarrollo de juegos más intuitivo y eficiente. Como GDScript está hecho a medida para Godot, aprovecha al máximo las características y optimizaciones del entorno, aumentando la eficiencia en el desarrollo y rendimiento del juego.

2.2 Blender

En este proyecto, la herramienta de modelado 3D que usamos es Blender. Blender es un software de gráficos por computadora en 3D, libre y de código abierto, que tiene una amplia gama de funciones, desde modelado y animación hasta renderizado e incluso edición de video. Blender es completamente gratuito y es adecuado tanto para estudios profesionales como para principiantes que empiezan desde cero. A continuación, explicaremos detalladamente algunas utilidades de Blender:

- **En cuanto al modelado,** Blender nos ofrece una variedad de herramientas de edición de mallas, como extrusión, corte, subdivisión y fusión, que nos permiten crear modelos 3D complejos. Además del modelado de mallas tradicional, los usuarios también pueden modelar utilizando curvas y superficies NURBS.
  El modo de escultura de Blender es perfecto para crear detalles y texturas, y ofrece una amplia biblioteca de pinceles. Además, los modificadores del sistema en Blender pueden ayudarnos a automatizar tareas comunes de modelado, como subdivisión de superficies, espejo y array, entre otros.

- **En cuanto a la rigidez de los huesos**, Blender ofrece potentes herramientas para el rigging, que es el proceso de vincular un esqueleto a un modelo 3D para animarlo. Este proceso implica asignar los vértices del modelo a los huesos del esqueleto, lo que permite que el modelo se deforme de manera dinámica a medida que el esqueleto se mueve.

- **En el área de animación**, Blender permite controlar el movimiento de los objetos mediante fotogramas clave y realiza interpolaciones automáticamente. La funcionalidad de Animación No Lineal (NLA) permite

repetir y combinar segmentos de animación. Dispone de potentes herramientas de rigging, incluyendo kinematics hacia adelante y hacia atrás.

- **Para el renderizado**, Cycles de Blender es un motor de renderizado de trazado de rayos imparcial, que proporciona efectos de iluminación realistas. Eevee es un motor de renderizado en tiempo real, ideal para previsualizaciones rápidas y animaciones. Las capas y los pases de renderizado ofrecen un mejor control y facilitan la composición posterior.

- **En materiales y texturas**, la función de despliegue UV permite desplegar la superficie de un modelo 3D en un plano 2D para un mapeado preciso. Los nodos de sombreado se utilizan para crear materiales complejos que simulan diferentes efectos de superficie.

- **En términos de administración y extensibilidad**, Blender tiene buenas funciones de gestión de archivos, con soporte para enlazar y reutilizar bibliotecas. Su capacidad de extensión es muy alta, ya que las funciones pueden ampliarse con scripts en Python y hay una gran cantidad de complementos y recursos de la comunidad disponibles.

- **La interfaz de usuario es personalizable**, lo que permite a los usuarios ajustar la disposición de la interfaz y los atajos de teclado según su flujo de trabajo. Además, su potente API permite a los desarrolladores de terceros crear nuevas herramientas y complementos.

2.3 Backend

2.3.1 Backend de juego

- **FastAPI:** Framework web de Python de alto rendimiento basado en ASGI para crear APIs de backend de juegos (como la gestión de usuarios, personajes y guardado en la nube).

- **Uvicorn:** Un servidor ASGI ligero y de alto rendimiento para ejecutar e implementar aplicaciones FastAPI.

- **SQLAlchemy 2.x:** Un framework ORM de Python para definir modelos de datos e interactuar con bases de datos de forma orientada a objetos.

- **psycopg2-binary:** Un controlador de Python para PostgreSQL, que sirve como interfaz subyacente para que SQLAlchemy se comunique con la base de datos.

- **Pydantic v2:** Una biblioteca de validación y serialización de datos para definir estructuras de datos de solicitud y respuesta, y realizar la validación de tipos.

- **python-multipart:** Se utiliza para analizar solicitudes multipart/form-data, compatible con la carga de archivos y el envío de formularios.

- **email-validator:** Se utiliza para validar formatos y validez de correo electrónico, comúnmente utilizado para el registro de usuarios y la verificación de cuentas.

- **python-jose:** Una biblioteca de implementación de JWT para generar y validar tokens de autenticación, implementando un mecanismo de inicio de sesión sin estado.

- **passlib (Argon2):** Una biblioteca de hash de contraseñas que utiliza algoritmos seguros para cifrar, almacenar y verificar las contraseñas de los usuarios.

- **python-dotenv:** Se utiliza para cargar archivos de variables de entorno .env para una gestión sencilla de las configuraciones (como conexiones a bases de datos y claves).

2.3.2 Backend de comunidad

- **Spring Boot** : Framework backend basado en Java 21 para construir servicios independientes del sistema de comunidad (APIs REST y lógica de negocio).

- **Spring Web (Spring MVC)**: Módulo para el desarrollo de APIs RESTful, encargado de manejar solicitudes HTTP, enrutamiento y controladores.

- **Spring IoC (Inversión de Control)**: Contenedor central que gestiona el ciclo de vida de los componentes mediante inyección de dependencias (DI), facilitando el desacoplamiento.

- **Spring Security**: Framework de seguridad que implementa autenticación y autorización, comúnmente basado en JWT para proteger endpoints.

- **Spring Data JPA**: Capa de acceso a datos que simplifica la persistencia y permite generar automáticamente operaciones CRUD sobre bases de datos relacionales.

- **Hibernate (implementación de JPA)**: Framework ORM que gestiona el mapeo objeto-relacional y la persistencia de entidades en la base de datos.

- **Maven**: Herramienta de gestión de dependencias y construcción del proyecto, encargada del ciclo de vida, compilación y empaquetado.

- **spring-boot-starter-validation**: Módulo de validación basado en Bean Validation (Jakarta Validation), utilizado para validar automáticamente los datos de entrada mediante anotaciones como @NotNull, @Size o @Email.

- **H2 Database**: Base de datos en memoria ligera, utilizada principalmente para desarrollo y pruebas, permitiendo ejecutar el sistema sin necesidad de una base de datos externa.

2.3.3 Base de datos y diseño

- **PostgreSQL**
  - Sistema principal de base de datos relacional.
  - Uso de esquemas:
    - `auth` (usuarios, seguridad, auditoría)
    - `game` (progreso, personajes, inventario, habilidades…)
    - `community` (posts, comentarios, likes, follows...)
    - chat (chat del comunitat y del juego)
- **JSONB:** Almacenamiento flexible de datos dinámicos (loadout, scene_state…).

2.4 Frontend de comunidad

- **Ionic + React + Vite**: Interfaz de usuario para la aplicación móvil de la comunidad.

- **TypeScript**: Lógica de frontend.

- **Axios (cliente HTTP):** Consumo de APIs REST del backend Spring Boot.

- **Node.js + npm:** Entorno de ejecución y gestión de dependencias.

2.5 Herramientas de desarrollo y pruebas

- **pytest:** Framework de pruebas para Python utilizado para tests unitarios y de integración del backend desarrollado con FastAPI.

- **Godot test framework (modo headless)**: Sistema de pruebas automatizadas del cliente de juego ejecutado sin interfaz gráfica, utilizado para validar lógica del juego y comportamiento del motor.

- **JUnit / Spring Boot Test**: Framework de pruebas para aplicaciones Java, utilizado para tests unitarios, de integración y pruebas del backend del sistema de comunidad basado en Spring Boot.

- **Mermaid**: Herramienta para la creación de diagramas mediante texto, utilizada para documentar arquitecturas, flujos de procesos y modelos entidad-relación (ER) directamente en la documentación.

2.6 Control y arquitectura del proyecto

Este proyecto adopta una arquitectura de múltiples subsistemas basada en Monorepo y logra un desacoplamiento arquitectónico completo entre el sistema de juego (Godot + FastAPI) y el sistema de comunidad (Ionic + React + Spring Boot) mediante estrategias estrictas de control de proyecto, que incluyen la partición de módulos, contratos de API unificados y gestión de la consistencia de datos.

Además, a través de un diseño de separación front-end/back-end y una arquitectura por capas, y mediante el uso de un sistema de datos unificado multimodo basado en PostgreSQL, se logra el aislamiento lógico y la evolución independiente de los diferentes dominios de negocio (juego/comunidad/autenticación/chat), construyendo así un sistema de ingeniería de juegos completo, altamente cohesivo, débilmente acoplado, escalable y de fácil mantenimiento.

2.7 ¿Por qué utilizo Godot, Blender y otras herramientas?

En este proyecto, al elegir las herramientas de desarrollo, consideré no solo sus capacidades técnicas, sino también factores como el control del proyecto, los costos de desarrollo, la flexibilidad y la coherencia arquitectónica general.

En primer lugar, elegí Godot como motor de desarrollo de videojuegos y Blender como herramienta de modelado 3D, principalmente porque ambos son software libre y de código abierto. Esto me permite usar, modificar y distribuir libremente el juego desarrollado y los recursos relacionados sin incurrir en costos de licencia comercial, lo que garantiza la propiedad total de los resultados del proyecto. Esto es crucial para mí, ya que puedo usar el proyecto libremente en mi portafolio personal o en futuros proyectos de desarrollo.

En segundo lugar, estas herramientas gratuitas reducen significativamente los costos de desarrollo, lo cual es especialmente importante para un proyecto de fin de carrera con recursos limitados. A pesar de ser software libre, Godot y Blender poseen capacidades similares a las de herramientas comerciales: Godot ofrece un sistema de escenas completo, mecanismos de scripting flexibles y un excelente soporte para arquitectura modular; Blender puede gestionar el modelado, la animación y la creación de materiales, cubriendo todo el flujo de trabajo de producción de contenido 3D.

Para el backend, elegí FastAPI principalmente porque necesitaba un framework que permitiera la construcción rápida de API con una estructura clara. FastAPI, combinado con el sistema de tipos de Python, puede realizar la validación de datos automáticamente, lo que facilita mantener la coherencia en las estructuras de datos entre el cliente y el servidor. Además, ofrece un alto rendimiento, lo que lo hace ideal para sistemas de guardado de juegos.

Para el sistema de la comunidad, elegí Spring Boot porque quería implementar una arquitectura por capas más empresarial. En comparación con FastAPI, Spring Boot ofrece una estructura por capas más estandarizada (Controlador, Servicio, Repositorio) y mecanismos de seguridad robustos, lo que lo hace más adecuado para gestionar la lógica de negocio compleja en aplicaciones comunitarias.

Para la base de datos, elegí PostgreSQL porque admite tanto estructuras de datos relacionales tradicionales como almacenamiento de datos flexible mediante JSONB. Esto es particularmente importante para los sistemas de juegos, ya que el estado del personaje, la configuración del equipo y otros datos son altamente dinámicos y no se adaptan bien a estructuras de tabla completamente fijas.

Finalmente, elegí Ionic + React para desarrollar el frontend comunitario porque me permite crear aplicaciones multiplataforma utilizando tecnologías web, compatibles con dispositivos web y móviles, mejorando la reutilización del código y la escalabilidad del sistema.

Estas herramientas cuentan con una comunidad activa y abundantes recursos de aprendizaje, lo que me ayudó enormemente durante el desarrollo, permitiéndome resolver problemas rápidamente y mejorar continuamente mis habilidades técnicas. En conclusión, creo que la combinación de estas herramientas me permitió implementar un sistema completo con alta flexibilidad, autonomía tecnológica y buena escalabilidad a bajo costo, cumpliendo con los objetivos de diseño de este proyecto.

2.8 Lista de otras herramientas que he utilizado

1 - **Mixamo**

En nuestro proyecto elegimos utilizar Mixamo para manipular huesos y aplicar movimiento de personajes. Mixamo es una plataforma de servicios de animación basada en la nube que ofrece una serie de recursos de animación de alta calidad, incluyendo acciones de personajes y modelos de personajes. Mediante Mixamo, podemos añadir rápida y fácilmente animaciones a nuestros personajes.

....

2 - **Photopea**

Es un editor gráfico cuyas funciones son básicamente las mismas que Photoshop, pero es gratuito y se puede editar online en la Web, lo que lo convierte en un excelente sustituto de Photoshop. Se utiliza para editar imágenes, realizar ilustraciones, diseñar páginas web o convertir entre diferentes formatos de imágenes.

....

3 - **Visual Studio Code**

Visual Studio Code se utiliza como editor de código ligero para la escritura de código frontend, scripts y archivos de configuración, además de mejorar la eficiencia del desarrollo mediante su ecosistema de extensiones. En el proyecto se emplea específicamente para el desarrollo del frontend de la aplicación, es decir, Ionic + React.

....

4 - **IntelliJ IDEA**

IntelliJ IDEA se utiliza para el desarrollo del backend con Spring Boot, principalmente para la implementación de APIs, la capa de acceso a datos y la depuración de la lógica de negocio. En el proyecto se emplea para el desarrollo del backend de la aplicación, encargado de la comunicación con la base de datos, así como de la obtención y almacenamiento de datos.

....

5 - **Git**

Git se utiliza como sistema de control de versiones del proyecto, permitiendo la gestión de commits, el desarrollo mediante ramas y el seguimiento de versiones, garantizando la trazabilidad del proceso de desarrollo y la eficiencia en el trabajo colaborativo.

....

6 - **Productos de Google**

El servicio de **Google Drive** integrado en nuestro Gmail, proporcionado por el instituto, nos ayuda a guardar y hacer copias de seguridad de nuestros proyectos. Además, también tenemos un disco duro portátil para almacenar datos, siguiendo la regla 3-2-1.

Para nuestro sitio web, también usamos algunas herramientas gratuitas que nos da Google:
El **Google Search Console** nos ayuda a que la gente pueda encontrar nuestro sitio web cuando buscan cosas en Google.

....

7 - **VirtualBox**

VirtualBox es una aplicación gratuita y de código abierto para crear máquinas virtuales que ejecutan sistemas operativos.

En este proyecto, utilizaremos VirtualBox para construir nuestros propios servidores, como por ejemplo, un servidor de correo interno, un servidor backend y una base de datos, entre otros, usando Postfix.

....

9 - **PostgreSQL**

PostgreSQL es un sistema de gestión de bases de datos relacional y objeto-relacional de código abierto, conocido por su alta fiabilidad, extensibilidad y soporte para tipos de datos avanzados como JSONB.

En el proyecto se utiliza PostgreSQL como sistema principal de almacenamiento de datos, encargado de gestionar la información de usuarios, progreso del juego, estadísticas, inventarios y otros datos persistentes. Además, se

aprovecha su capacidad de soporte para JSONB para implementar un modelo híbrido relacional–documental, lo que permite almacenar tanto datos estructurados como datos dinámicos de forma eficiente, garantizando la integridad, consistencia y flexibilidad del sistema.

....

10 - **Rider IDE**

Rider IDE se utiliza para facilitar la subida de datos del proyecto Godot a GitHub, ya que el uso de la terminal para operaciones Git resulta menos conveniente. Rider permite realizar estas operaciones de forma más eficiente y cómoda.

....

11 - **Mermaid Live**

Mermaid Live se utiliza para la creación y mantenimiento de diagramas de arquitectura del sistema y diagramas de flujo, permitiendo la generación y actualización rápida de representaciones visuales dentro de la documentación del proyecto.

....

12 - **Github**

GitHub se utiliza como plataforma de alojamiento del código fuente, permitiendo la gestión del repositorio del proyecto, el control de versiones y el soporte para procesos de revisión de código. En GitHub se almacena todo el conjunto de repositorios del proyecto.

....

13 - **Jira**

Jira se utiliza para la gestión de tareas y el seguimiento del progreso del proyecto, mediante la división de tareas, la gestión de estados y el control del flujo de trabajo, mejorando la organización y la trazabilidad del desarrollo.

....

14 - **AI Coding**

En este proyecto, también utilicé diversas herramientas de IA para mejorar la eficiencia del desarrollo y facilitar la construcción del proyecto, entre ellas herramientas de programación asistida por IA como Claude, Cursor, Chat GPT y Gemini.

....

15 - **Otras herramientas no tan utilizadas**

- **Ilovepdf** - Ilovepdf nos ha servido para cambiar el tamaño de las imágenes y comprimir el tamaño de los documentos, es muy útil y fácil de usar.
- **Photokit:** Es una herramienta en línea de edición de imágenes con IA que permite eliminar fondos, retocar fotos y mejorar la calidad de las imágenes.
- **Krita:** Krita es un programa de código abierto para pintura digital e ilustración profesional.

2.9 Recursos que utilizamos

Además de utilizar una serie de herramientas, también recurrimos a algunos recursos en Internet, o podríamos decir que son nuestros proveedores. Aquí está la lista:

- Pixabay: Usamos Pixabay para buscar efectos de sonido y música para nuestro juego. El contenido es de uso gratuito, no requiere atribución al autor y se puede modificar o adaptar para crear nuevas obras.
- YouTube: Buscamos tutoriales en esta plataforma sobre modelado y algunos diseños en Godot.
- BiliBili: Además de ver tutoriales en YouTube, también buscamos tutoriales en BiliBili (una plataforma china).
- Github: Podemos encontrar los recursos, herramientas y plugins que necesitamos en GitHub.

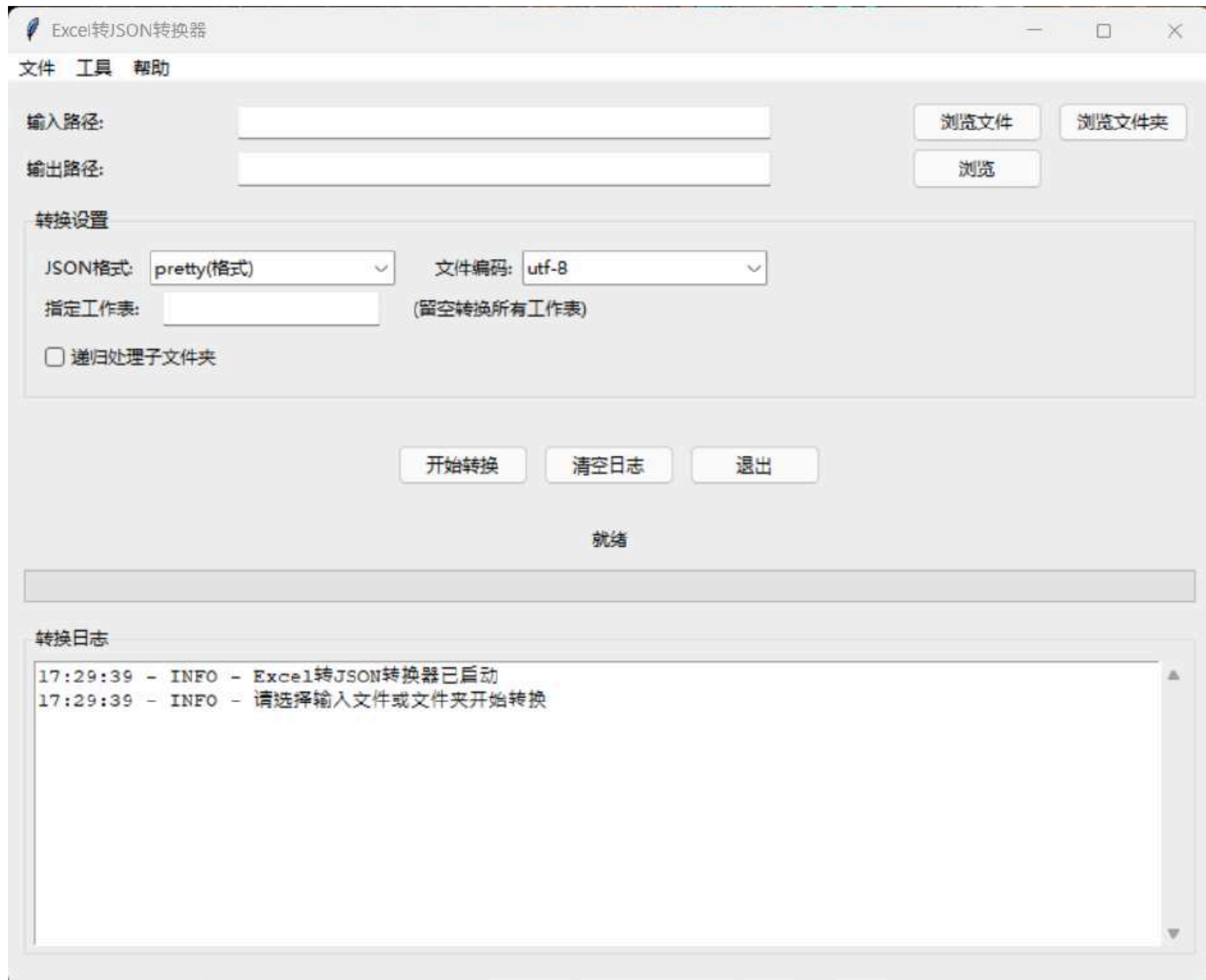
2.10 Herramientas desarrolladas para el proyecto

2.10.1 Excel a JSON

He desarrollado una herramienta de conversión de Excel a JSON para facilitar la gestión de datos del proyecto. Esta herramienta permite transformar automáticamente archivos Excel en archivos JSON compatibles con el sistema del juego, lo que simplifica la importación y lectura de datos dentro de Godot y del backend.

Gracias a esta herramienta, es posible editar información como estadísticas, configuraciones, enemigos, habilidades u otros datos del juego directamente desde tablas Excel, evitando modificar manualmente archivos JSON y mejorando la eficiencia durante el desarrollo y la iteración de contenido.

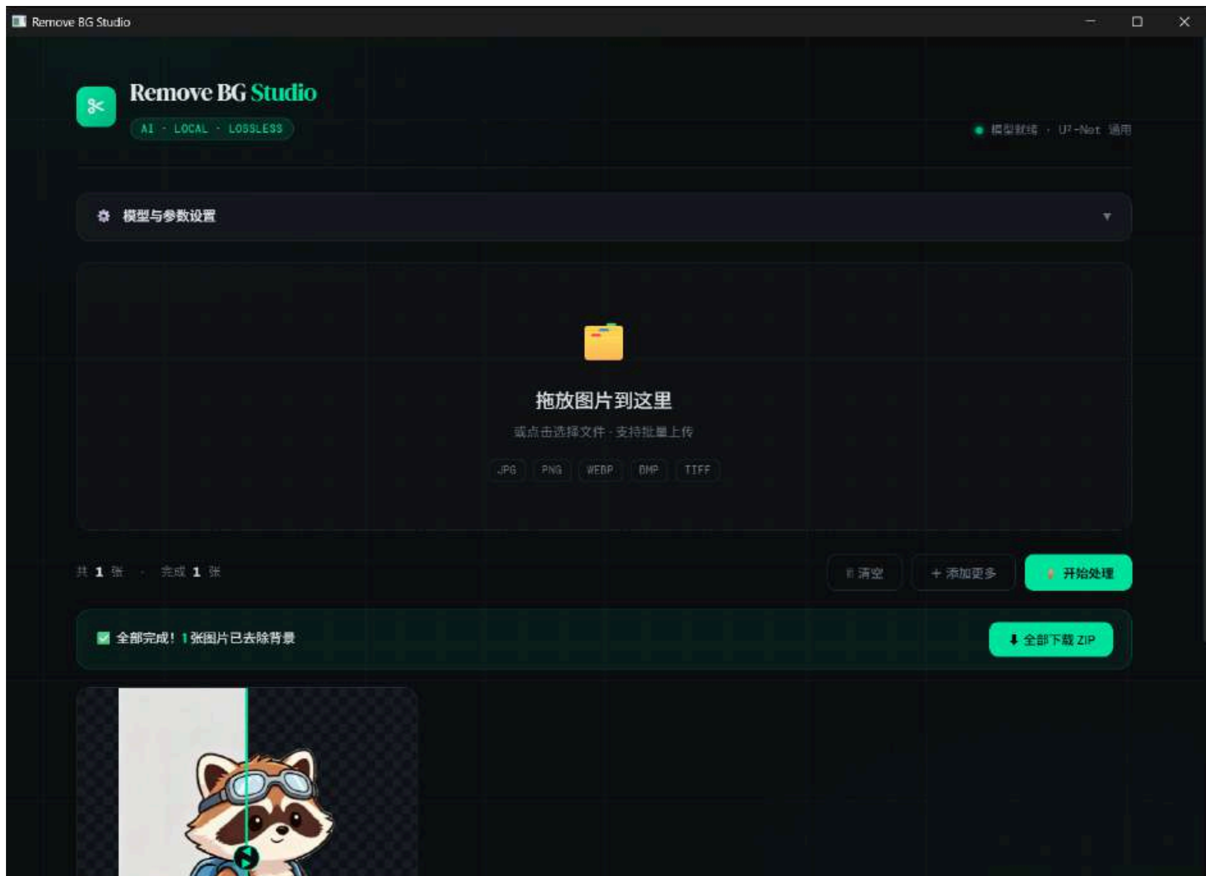
A	B	C	D	E	F	G
tem_id	item_name	item_desc	item_icon	max_stack	item_type	rarity
100	实验室钥匙	用来开启实验室	res://addons/xuanBag/img/item_icon/实验室钥匙.jpg	1	QUEST	RARE
101	门禁卡	用来开门		99	CONSUMABLE	COMMON
102	星币	货币		99999	CONSUMABLE	COMMON
103	改名卡	用来改名		99	CONSUMABLE	EPIC



2.10.2 Eliminar fondo

Debido a que algunas herramientas de eliminación de fondo disponibles en línea presentan limitaciones como número de usos, pérdida de calidad o restricciones de tamaño de archivo, en muchas ocasiones era necesario repetir el proceso varias veces y la calidad final de las imágenes no siempre era satisfactoria.

Por esta razón, desarrollé una herramienta propia utilizando Python junto con librerías de código abierto disponibles en GitHub. Esta herramienta permite realizar el procesamiento de imágenes de forma local, reduciendo la dependencia de servicios externos y manteniendo una mejor calidad visual, además de facilitar el tratamiento masivo de recursos como personajes, interfaces y texturas del proyecto.



3. Descripción del juego

3.1 El diseño de juegos

“El Human Disaster” es un videojuego de rol de pequeño formato con mundo abierto, cuyo estilo general combina el absurdo y el humor negro. El juego transforma las anomalías sistémicas y los comportamientos no previstos surgidos durante el proceso de desarrollo en mecánicas de juego, convirtiendo el “caos y la pérdida de control” en una parte central de la experiencia.

Los jugadores pueden explorar libremente múltiples escenarios, como ciudades, zonas industriales y naves extraterrestres, experimentando una estructura de mundo abierto que incluye combate, resolución de puzles e interacción narrativa. El juego admite diversas combinaciones de atributos de personaje, combate de disparos, progresión de habilidades, cooperación multijugador en línea y un sistema de misiones con narrativa dinámica, donde las acciones del jugador influyen directamente en el estado del mundo y en el desarrollo de la historia.

A nivel técnico, el proyecto adopta una arquitectura modular que incluye un cliente, servicios backend y un sistema de base de datos, con el objetivo de soportar combate en tiempo real, sincronización del mundo, gestión de misiones y almacenamiento de datos de jugadores. Además, se extiende una plataforma independiente de comunidad de jugadores que proporciona foros, intercambio de contenido, cuentas sociales y sistemas de formación de equipos, creando un ecosistema integrado entre el juego y el entorno social.

En conjunto, el proyecto es un sistema de múltiples mecánicas que combina exploración de mundo abierto, rol, supervivencia multijugador e infraestructura comunitaria, con el objetivo de diseñar una experiencia unificada de “jugabilidad + servicios backend + ecosistema comunitario”.

El juego admitirá la ejecución en los sistemas operativos **Windows** y **Linux**.

3.2 Jugabilidad y objetivos

En las primeras etapas del juego, el diseño se centra principalmente en la exploración y en una estructura de progresión basada en misiones. El jugador debe avanzar en la narrativa principal completando distintos tipos de tareas, tales como la búsqueda de objetos específicos, la resolución de puzles y el combate contra enemigos.

Asimismo, se introduce un sistema de progresión del personaje, mediante el cual el jugador puede obtener puntos de experiencia para subir de nivel y desbloquear distintas habilidades, mejoras genéticas o módulos tecnológicos. Estos elementos permiten fortalecer progresivamente al personaje para afrontar desafíos cada vez más complejos.

El objetivo principal del juego consiste en completar las misiones de la historia principal y derrotar a las fuerzas invasoras que amenazan el mundo, impulsando así el desarrollo narrativo global. Durante el proceso de progresión, el personaje experimenta una evolución o mutación progresiva que le permite adquirir capacidades más avanzadas en combate y supervivencia.

A medida que avanza la partida, el jugador va descubriendo gradualmente la verdad detrás de la invasión extraterrestre, al mismo tiempo que desbloquea nuevas áreas y funcionalidades del sistema, incrementando así la profundidad y complejidad del juego.

3.3 Público objetivo

El público objetivo de este proyecto está formado por jugadores que prefieren videojuegos de mundo abierto con un estilo humorístico. El diseño del juego busca abarcar usuarios de diferentes rangos de edad, con un enfoque particular en aquellos jugadores que disfrutan de experiencias de exploración profunda y elementos de rol.

Asimismo, mediante un diseño narrativo atractivo y un desarrollo cuidado de los personajes, el juego también pretende atraer a jugadores que prefieren experiencias basadas en la narrativa, especialmente aquellos interesados en historias elaboradas y en mecánicas de resolución de puzles con cierto nivel de desafío.

3.4 Conceptualización de ideas y diseños

3.4.1 Contexto y trama del juego

La historia se sitúa en el año 2060, cuando la humanidad desarrolla con éxito un nuevo tipo de onda de señal y decide enviarla al espacio exterior. Este avance tecnológico, aunque representa un hito científico, provoca de manera no intencionada la exposición de la posición de la Tierra, atrayendo la atención de una civilización extraterrestre que posteriormente inicia una invasión.

Tras su llegada, los seres extraterrestres comienzan a secuestrar humanos con fines de investigación biológica. Debido a la enorme disparidad tecnológica, la humanidad se encuentra en clara desventaja y solo puede sobrevivir mediante una resistencia continua, enfrentándose además a desapariciones masivas de población.

En este contexto, la especie extraterrestre somete a los humanos a experimentos biológicos extremadamente crueles, de los cuales prácticamente ningún sujeto logra sobrevivir. Paralelamente, la humanidad establece una organización conocida como la “Unión Terrestre” con el objetivo de contrarrestar la invasión.

El protagonista y su compañero son seleccionados para infiltrarse en la flota alienígena con el fin de obtener información sobre sus operaciones y planes estratégicos. Sin embargo, durante la misión, ambos son capturados y obligados a participar en los experimentos.

Por razones desconocidas, el experimento resulta en una transformación anómala que permite la supervivencia de algunos sujetos. Estos individuos, conocidos como experimentos supervivientes, pierden el control y provocan graves daños en la nave extraterrestre. Entre ellos, un total de cinco supervivientes logran sobrevivir, incluido el protagonista.

Estos cinco supervivientes constituyen los personajes controlables por el jugador, cada uno asociado a una forma animal distinta: pez, gato, gallina, perro y conejo. Al despertar, el protagonista descubre que ha sido transformado en una de estas formas animales, determinada por la elección del jugador.

A nivel de mecánicas de juego, la elección de un personaje implica que los otros cuatro supervivientes quedan bajo control mental extraterrestre y pasan a ser entidades hostiles. El jugador deberá enfrentarlos en misiones secundarias para liberarlos del control mental y reclutarlos como aliados, con el objetivo de enfrentarse al jefe final.

En caso de no completar dichas misiones secundarias, el jugador deberá enfrentarse al jefe final en solitario, sin el apoyo de los demás personajes, lo que genera diferentes rutas de dificultad y finales alternativos dentro del juego.

3.4.2 Los protagonistas

En el presente proyecto se definen cinco protagonistas principales. Entre ellos destacan el “FishMan”, un personaje masculino con características de pez, y la “CatLady”, un personaje femenino con rasgos felinos.

El juego permite al jugador seleccionar uno de estos cinco personajes como protagonista principal. Todos ellos han sido previamente secuestrados por una civilización extraterrestre y han sobrevivido a una serie de experimentos biológicos, lo que ha dado lugar a su transformación en entidades mutadas con características animales. Cada uno de los cinco protagonistas dispone de una narrativa inicial diferenciada; sin embargo, en la versión actual del proyecto, únicamente se han desarrollado dos introducciones específicas correspondientes a FishMan y CatLady (véase Apéndice [1]).

Debido a sus distintas características animales y a su condición de supervivientes de los experimentos, cada protagonista posee habilidades especiales únicas. Por ejemplo, el personaje FishMan puede respirar indefinidamente bajo el agua y desplazarse sobre la superficie acuática. No obstante, cada personaje también presenta debilidades específicas; en el caso de CatLady, el personaje tiene aversión al agua y requiere completar misiones secundarias para obtener objetos que le permitan sumergirse.

Tomando como ejemplo a FishMan, al inicio del juego el personaje no posee una forma humanoide, sino que se presenta como un pez común contenido en un acuario. El jugador debe interactuar con el entorno para evitar que el acuario se rompa, al mismo tiempo que resuelve puzles y busca una vía de escape. A medida que se completan los objetivos, el personaje comienza un proceso de evolución progresiva, desarrollando gradualmente una forma humanoide.

Este proceso corresponde a la transición entre diferentes etapas evolutivas. A medida que el protagonista se fortalece, su forma humana se vuelve más completa, y el jugador puede seleccionar distintas rutas genéticas para definir su evolución. Con el crecimiento del personaje, se amplían sus capacidades de interacción y se desbloquean habilidades más avanzadas, enriqueciendo así la experiencia de juego.

(1) Introducción al protagonista unificado elegido por los jugadores

El/La protagonista era originalmente un ser humano común, hasta que un día, por infortunio, fue secuestrado por extraterrestres. Durante los experimentos ajenos, le inyectaron un extraño elixir que provocó cambios sorprendentes en su cuerpo. Ahora, se ha convertido en un monstruo/mutante mitad humano, mitad animal, pero en su corazón todavía mantiene el coraje y el sentido de la justicia humanos.

Durante los días de cautiverio en los extraterrestres, el/la protagonista vio de primera mano el terrorífico poder de la tecnología alienígena y los experimentos despiadados que realizaban con los humanos. Esto despertó su ira hacia los extraterrestres y su deseo de salvar al mundo. Decidió levantarse en resistencia, enfrentándose en una lucha encarnizada contra los ajenos por defender la Tierra y todos los humanos cautivos.

(2) Sistema de niveles

El sistema de progresión del personaje se basa en un mecanismo de niveles impulsado por puntos de experiencia, diseñado para controlar el crecimiento progresivo de las capacidades del personaje y la liberación de sistemas del juego.

El jugador obtiene experiencia mediante la realización de misiones, la derrota de enemigos y la activación de eventos específicos. Cuando la experiencia acumulada alcanza determinados umbrales, el nivel del personaje aumenta. Sin embargo, a partir de ciertos niveles clave, el jugador no puede seguir progresando únicamente mediante experiencia, sino que debe completar una “misión de ascenso” para desbloquear el límite de nivel actual. En caso de no completar dicha misión, el nivel quedará bloqueado aunque la experiencia haya alcanzado el umbral requerido.

El incremento de nivel no solo mejora directamente los atributos básicos del personaje, como la salud, el ataque y la defensa, sino que también actúa como requisito para la liberación de sistemas avanzados. A medida que el nivel progresa, se desbloquean progresivamente sistemas como el sistema de habilidades, el sistema genético y los módulos de mejora del núcleo. El sistema de habilidades amplía las capacidades de combate y las opciones de interacción del jugador, el sistema genético define la ruta evolutiva y la orientación de atributos del personaje, y el sistema de núcleo proporciona mejoras avanzadas y ampliaciones funcionales, como la potenciación de habilidades o la sincronización asistida por inteligencia artificial.

Asimismo, el sistema de niveles está estrechamente vinculado al sistema de evolución del personaje. En determinados puntos de progresión, el personaje puede experimentar transformaciones evolutivas o saltos de capacidad, lo que modifica significativamente su estilo de combate y la experiencia de juego, introduciendo una progresión estructurada por etapas.

En conjunto, este mecanismo de niveles no solo cumple una función de crecimiento numérico, sino que también actúa como un sistema clave de desbloqueo de contenido y expansión de la jugabilidad, constituyendo uno de los ejes centrales de la estructura del juego.

3.4.3 Los mapas

El proyecto contempla tres tipos de escenarios: escenarios principales, escenario de práctica y escenario de tutorial.

(1) Escenarios principales

Los escenarios principales se componen de tres áreas fundamentales:

Nave espacial alienígena: constituye el escenario inicial del juego, donde el protagonista despierta. Forma parte de una base extraterrestre y, en la versión actual, se encuentra en fase de desarrollo, incluyendo únicamente una estructura básica de laboratorio y pasillos.
Escenario terrestre (año 2060): representa el mundo original del protagonista, mostrando un entorno afectado por la invasión alienígena.

Flota principal alienígena: corresponde a una zona avanzada del juego, destinada a presentar la estructura central de las fuerzas enemigas y los desafíos finales.

(2) Escenario de práctica

El escenario de práctica proporciona un entorno libre donde el jugador puede experimentar con habilidades, armas y mecánicas de daño. En este espacio, se permite la modificación y prueba de equipamiento de forma flexible, facilitando la familiarización con los sistemas del juego.

(3) Escenario de tutorial

El escenario de tutorial tiene como objetivo introducir a los nuevos jugadores en las mecánicas básicas del juego. En la primera ejecución, aquellos usuarios que no estén familiarizados con el sistema deberán completar este tutorial, el cual cubre aspectos como el control del personaje, el uso de armas, el combate contra enemigos, la interacción con el entorno y el uso de habilidades.

Actualmente, el tutorial se implementa como un escenario independiente con funcionalidades básicas. En futuras iteraciones, se prevé integrar estos elementos dentro de la narrativa principal, mediante un enfoque progresivo que permita al jugador aprender las mecánicas de forma contextualizada durante la experiencia de juego.

3.4.4 La trama principal y las misiones secundarias

La trama principal del juego es bastante fija y los jugadores no pueden desviarse demasiado de ella. A continuación, presentamos un borrador básico de nuestra trama principal:

1. **Trama inicial:** El protagonista es capturado y llevado a la nave espacial alienígena. Aquí, el jugador debe luchar, resolver acertijos, evolucionar y escapar. Las tramas iniciales de los cinco protagonistas aquí antes de ser capturados por extraterrestres son todas diferentes. Por ejemplo, Fishman y CatLady son combatientes de la Alianza Terrestre.
   Los detalles del borrador se pueden encontrar en el [Anexo 1].
2. **Trama intermedia:** El protagonista regresa a la Tierra y comienza su camino hacia el fortalecimiento. En esta etapa, busca ayuda, resuelve una serie de problemas y lucha contra otros sujetos experimentales capturados y modificados, es decir, los otros cuatro protagonistas. Finalmente, el protagonista encuentra una oportunidad para llegar a la nave espacial principal de los extraterrestres.
3. **Trama final:** El protagonista encuentra la manera de llegar a la nave espacial principal y lucha contra el jefe final en una gran batalla. Al final, el protagonista gana, salva el mundo y obtiene pistas sobre cómo volver a ser un humano normal.

Misiones secundarias:
Además de la misión principal, nuestro juego cuenta con muchas misiones secundarias. Estas misiones secundarias son en tiempo real, lo que significa que si

el jugador avanza en la misión principal sin completar las secundarias, las perderá para siempre.

Cada misión secundaria incluye una trama, que puede ser independiente o estar relacionada con otras. A diferencia de las misiones principales, los jugadores pueden decidir la dirección de las misiones secundarias. Esto significa que, dependiendo de las elecciones del jugador, la trama secundaria se desarrollará de manera diferente y afectará la trama principal en distintos grados.

Sin embargo, debido a problemas de tiempo y complejidad, nuestra versión beta no incluye la línea principal ni los ramales por el momento.

3.4.5 Los enemigos y los NPCs

En este proyecto, los enemigos y los NPC (personajes no jugables) constituyen componentes fundamentales del sistema de interacción y jugabilidad, compartiendo una arquitectura unificada basada en el sistema de nodos y escenas de Godot, implementados mediante un diseño modular que separa la lógica, la representación y los datos, reutilizando componentes comunes como movimiento, detección y gestión de daño, y siendo coordinados por sistemas globales como señales y gestores.

A nivel de comportamiento, los enemigos se controlan principalmente mediante un sistema de inteligencia artificial basado en máquinas de estados finitos (Finite State Machine, FSM), en el cual el comportamiento se organiza en estados claramente definidos como Idle (inactividad), Chase (persecución), Attack (ataque) y Dead (muerte), permitiendo transiciones controladas que garantizan un comportamiento predecible y estructurado, mientras que los NPC se diseñan como entidades centradas en la interacción, encargadas de proporcionar información, diálogos y construcción del mundo, pudiendo además actuar como nodos de eventos para la futura expansión del sistema de misiones, y considerando la posible incorporación de árboles de comportamiento (Behavior Tree) para dotar a la IA de una toma de decisiones más compleja, basada en jerarquías, condiciones y prioridades, en contraste con el modelo lineal de estados, lo que permite ampliar la expresividad y flexibilidad del sistema.

En conjunto, el sistema de enemigos se orienta al combate mientras que el sistema de NPC se orienta a la interacción, contribuyendo ambos a la profundidad jugable y a la construcción del mundo del juego, además de sentar las bases para futuras expansiones en sistemas de IA, diálogo y misiones.

3.4.6 El combate

Antes de completar su evolución hacia una forma humana o humanoide, los personajes presentan limitaciones en sus capacidades de combate y no pueden participar directamente en enfrentamientos. En esta fase, la jugabilidad se centra principalmente en la resolución de puzles, la interacción con el entorno y la evasión de amenazas, permitiendo al jugador progresar mediante mecánicas no basadas en el combate.

Una vez finalizado este proceso evolutivo, se desbloquean las capacidades básicas de combate, lo que permite al jugador utilizar armas y enfrentarse directamente a los enemigos, accediendo así a una fase de juego más completa.

3.4.7 El inventario y las transacciones monetarias

Dado que el juego incorpora un sistema de misiones, se ha diseñado e implementado un sistema de inventario. Este sistema se utiliza para almacenar objetos, armas y elementos clave de misión, los cuales desempeñan un papel fundamental en la resolución de puzles y en la progresión del juego. En la versión actual, el sistema de inventario ha sido implementado en su funcionalidad básica y permite la persistencia de datos mediante su almacenamiento en la base de datos del backend, garantizando la consistencia y recuperabilidad de la información.

Asimismo, el diseño del juego contempla la inclusión de un sistema de economía basado en moneda, que permitirá a los jugadores interactuar con NPCs para la compra e intercambio de objetos, enriqueciendo así la experiencia de interacción y la dinámica económica del juego. No obstante, en la versión actual, esta funcionalidad aún no ha sido implementada y se encuentra en fase de diseño.

3.4.8 El multijugador en LAN

Nuestro juego cuenta con un modo multijugador en red local, que por ahora no tiene ninguna trama y solo permite a los jugadores luchar juntos contra enemigos. Este modo todavía está en desarrollo. En nuestra versión beta solo habrá un esbozo básico del modo multijugador, sin mucho contenido adicional por el momento.

3.4.9 Sistema de guardado de juego y datos estáticos

En este proyecto, el sistema de datos se basa en un enfoque integrado de “datos estáticos + almacenamiento dinámico + arquitectura dirigida por datos”, con el objetivo de desacoplar el contenido del juego de su lógica e incrementar la escalabilidad del sistema.

En primer lugar, a nivel de datos estáticos, elementos como armas, atributos de enemigos, configuración de habilidades y valores base del sistema se definen mediante archivos JSON. Estos datos son cargados y analizados por un módulo central de gestión de datos durante la inicialización del sistema. Dado que presentan una naturaleza estable, no dependen del estado del usuario y se utilizan principalmente para definir las reglas fundamentales y la estructura del juego.

En segundo lugar, en cuanto al almacenamiento dinámico, los datos del jugador (como progreso del personaje, experiencia, inventario y estado de misiones) se almacenan de forma persistente a través del backend, utilizando una base de datos PostgreSQL. Para aquellos datos con estructuras más complejas y dinámicas, el sistema adopta un modelo híbrido relacional–documental mediante el uso de campos JSONB, lo que permite equilibrar la eficiencia de consulta con la flexibilidad estructural.

En relación con la arquitectura dirigida por datos, el cliente no implementa parámetros de lógica de juego de forma rígida en el código, sino que basa su

funcionamiento en la carga de configuraciones estáticas y en los datos proporcionados por la API del backend. De este modo, atributos de enemigos, efectos de armas y parámetros de habilidades están definidos por datos, mientras que la capa lógica se encarga únicamente de la ejecución y el cálculo, siguiendo un paradigma de “data-driven design”. Este enfoque reduce significativamente el acoplamiento del sistema y mejora la escalabilidad y la capacidad de iteración del contenido.

Adicionalmente, el cliente utiliza un módulo unificado (DataManager) para la gestión de datos, encargado de la carga de configuraciones estáticas, la gestión de caché local y la sincronización con el backend, garantizando la consistencia de los datos y la eficiencia en la ejecución. En conjunto, este diseño permite la expansión y modificación del contenido del juego sin necesidad de alterar la lógica central del sistema.

3.4.10 El sistema de progresión del personaje

El sistema de progresión del personaje se compone de tres subsistemas principales: el sistema de habilidades, el sistema genético y el sistema de núcleo, los cuales en conjunto impulsan el desarrollo de las capacidades de combate y amplían la jugabilidad general del juego.

El **sistema de habilidades** se encarga de ampliar el estilo de combate y las capacidades de interacción del personaje. El jugador puede desbloquear diferentes habilidades mediante la progresión de nivel, la finalización de misiones y la activación de determinados elementos genéticos. Estas habilidades se asocian generalmente a acciones específicas o efectos funcionales, como mejoras de ataque, aumento de movilidad o la activación de habilidades especiales, enriqueciendo así la estrategia de combate y la profundidad del control.

El **sistema genético** define la dirección evolutiva y la orientación de atributos del personaje. Basado en las características biológicas y la trayectoria evolutiva del mismo, este sistema ofrece múltiples ramas de desarrollo, como potenciación del ataque, mejora defensiva, incremento de probabilidad de crítico o especialización en habilidades concretas, proporcionando así mayor libertad y diversidad en la progresión del personaje.

Por su parte, el **sistema de núcleo** actúa como un módulo de mejora de nivel superior, proporcionando ampliaciones avanzadas como la potenciación de habilidades, el incremento de multiplicadores de atributos y mecanismos de sincronización asistida por inteligencia artificial. Asimismo, este sistema puede mejorar el rendimiento general de armas y capacidades, constituyendo un componente clave en la fase avanzada del juego.

En conjunto, estos tres sistemas forman una estructura jerárquica clara: el sistema de habilidades se centra en la expansión de las acciones del personaje, el sistema genético define su dirección de crecimiento, y el sistema de núcleo proporciona mejoras de alto nivel. La interacción de estos subsistemas constituye el marco central del sistema de progresión del personaje.

4. Diseño de la aplicación de comunidad de juegos

4.1 Contexto general

La aplicación de comunidad del juego se desarrolla como un subsistema independiente dentro del proyecto, con el objetivo de ampliar las funcionalidades del juego principal y proporcionar un entorno social orientado a los jugadores. Esta plataforma permite a los usuarios publicar contenido, interactuar entre sí y acceder a información relevante del juego, facilitando el intercambio de experiencias, estrategias y actualizaciones, lo que contribuye a mejorar la participación y la experiencia global del usuario.

Adicionalmente, el sistema incorpora una serie de herramientas auxiliares diseñadas para facilitar el acceso a información del juego desde dispositivos móviles, como datos de personajes, eventos o notificaciones del sistema. Estas herramientas complementan las limitaciones del cliente del juego y optimizan la accesibilidad a la información.

Desde el punto de vista arquitectónico, la aplicación de comunidad mantiene una integración conceptual con el juego principal mediante mecanismos de intercambio de datos, como la sincronización de progreso y la distribución de recompensas. No obstante, se ha diseñado siguiendo un enfoque desacoplado, con despliegue independiente y arquitectura modular, lo que permite su evolución autónoma.

En conjunto, el sistema se concibe como una plataforma que integra interacción social, gestión de contenido y capacidad de expansión tecnológica, contribuyendo tanto a la retención de usuarios como a la construcción de un ecosistema completo alrededor del juego.

4.2 Funciones principales

La aplicación implementa una serie de funcionalidades centradas en la interacción del usuario:

- Publicación de contenido: los usuarios pueden crear publicaciones que incluyen texto, imágenes y ubicaciones dentro del juego.
- Sistema de me gusta, comentarios y compartidos: los usuarios interactúan mediante respuestas, “me gusta” y compartiendo contenido.
- Sistema de autenticación: registro e inicio de sesión basados en JWT.
- Visualización de contenido: presentación paginada en orden cronológico.
- Perfil de usuario: gestión básica de la información del usuario.
- Sistema de chat: los usuarios pueden enviarse mensajes entre sí para comunicarse.
- Funciones de herramientas: la aplicación ofrece herramientas integradas para consultar y calcular información del juego, como mapas, cálculo de daño de armas y visualización de objetos.
- Sistema de notificaciones: permite recibir anuncios oficiales, información de eventos y mensajes de chat.

4.3 Arquitectura del sistema

La aplicación de comunidad adopta una arquitectura desacoplada entre frontend y backend, lo que permite una clara separación de responsabilidades y mejora la mantenibilidad y escalabilidad del sistema.

La estructura del sistema es la siguiente:

- Frontend (Ionic + React): encargado de la interacción con el usuario, la renderización de la interfaz y la comunicación con el backend mediante APIs REST.
- Backend (Spring Boot): responsable de la lógica de negocio, incluyendo la gestión de usuarios, publicaciones, comentarios e interacciones.
- Base de datos (PostgreSQL): organizada mediante múltiples esquemas (por ejemplo, autenticación y datos de comunidad), lo que permite una separación lógica entre distintos dominios.

El backend sigue una arquitectura en capas (controlador–servicio–repositorio), lo que facilita la organización del código y mejora la mantenibilidad, la testabilidad y la escalabilidad.

Este enfoque evita la complejidad de una arquitectura de microservicios, manteniendo al mismo tiempo un alto nivel de desacoplamiento que permite la evolución independiente de los componentes del sistema.

4.4 Diseño

El diseño de la aplicación se basa en dos principios fundamentales: la modularidad y la claridad en las interacciones entre componentes.

El frontend adopta una arquitectura basada en componentes, donde las distintas vistas funcionales (como inicio de sesión, feed de contenido, perfil de usuario y publicaciones) están desacopladas y organizadas de forma jerárquica. La gestión del estado se implementa mediante hooks de React, mientras que la comunicación con el backend se realiza a través de solicitudes REST API estructuradas, garantizando un flujo de datos claro y controlado.

El backend sigue los principios de la arquitectura RESTful, lo que se refleja en:

- La definición de endpoints claros y semánticos (por ejemplo, /posts, /comments, /auth)
- El uso de JSON como formato estándar de intercambio de datos
- La validación de datos y el procesamiento de la lógica de negocio en la capa de servicio
- La separación entre la lógica de negocio y la capa de persistencia

Además, el sistema fue diseñado con la escalabilidad en mente mediante una arquitectura modular y en capas. Actualmente, ya se han implementado el sistema de notificaciones y el mecanismo de “me gusta”, y la estructura del sistema permite futuras extensiones (como sistemas de recomendación) sin afectar el núcleo de la arquitectura.

4.5 Integración básica con el sistema de juego

La integración entre la aplicación de comunidad y el sistema de juego se ha diseñado de forma ligera y desacoplada, evitando dependencias directas entre ambos backends.

Los puntos clave de integración son los siguientes:

- Uso compartido de un identificador de usuario unificado (UID) entre sistemas para garantizar la consistencia de la identidad
- Asociación de datos básicos del jugador mediante el UID, como el perfil de usuario y la progresión en el juego
- Reserva de capacidad de extensión en el diseño del sistema para futuras funcionalidades como logros, estadísticas y rankings

Actualmente, la integración se limita a la consistencia a nivel de identidad, lo que reduce significativamente la complejidad del sistema y minimiza el acoplamiento. Sin embargo, la arquitectura permite ampliaciones futuras mediante APIs compartidas o mecanismos basados en eventos para lograr una integración más profunda entre sistemas.

4.6 Seguridad

La seguridad de la aplicación de comunidad se centra en la autenticación, el control de acceso y la protección de datos.

Se utiliza un sistema de autenticación basado en JWT, donde el usuario se identifica mediante tokens con expiración y mecanismos de renovación. En cuanto a la autorización, se protegen los endpoints sensibles y se garantiza que solo el propietario de un recurso pueda modificarlo o eliminarlo.

Todas las entradas del cliente se validan en el backend para prevenir vulnerabilidades comunes, y la comunicación entre cliente y servidor se realiza mediante HTTPS para asegurar la confidencialidad e integridad de los datos.

En general, el sistema implementa medidas básicas de seguridad, aunque aún existen mejoras posibles como la integración de OAuth, protección contra CSRF y mecanismos avanzados de auditoría.

4.7 Planes futuros

El desarrollo futuro de la aplicación de comunidad se centrará en mejorar la experiencia de usuario, el rendimiento y la mantenibilidad del sistema.

A nivel funcional, se prevé mejorar la interacción del usuario mediante notificaciones en tiempo real y una comunicación basada en WebSocket. En cuanto al rendimiento, se introducirán mecanismos de caché (como Redis), optimización de consultas en PostgreSQL y estrategias eficientes de paginación y carga de datos.

En términos de seguridad y arquitectura, se planea integrar sistemas más avanzados de autenticación y control de acceso (como OAuth2 y RBAC), junto con prácticas de integración continua y pruebas automatizadas. Además, se reforzará la

integración con el sistema de juego, permitiendo la sincronización de datos y la visualización de estadísticas dentro de la comunidad.

4.8 Resumen de esta sección

Esta aplicación se ha desarrollado como una extensión del juego, actuando como su plataforma comunitaria. He utilizado una arquitectura desacoplada y tecnologías modernas para implementar la gestión de contenido y la interacción entre usuarios.

Además, se han identificado áreas de mejora con el objetivo de reforzar aspectos clave como la seguridad, el rendimiento, la escalabilidad y la experiencia de usuario.

En definitiva, esta aplicación de comunidad no solo complementa el sistema de juego, sino que también constituye una plataforma escalable con potencial para evolucionar hacia un entorno social más completo y robusto.

5. BackEnd

Este proyecto consta de dos servicios principales: un backend ligero para el sistema de juego y un backend de nivel empresarial para la plataforma social. Ambos se basan en la misma base de datos PostgreSQL.

5.1 Postgresql

La base de datos principal del proyecto está implementada con PostgreSQL, seleccionada por su robustez, consistencia y capacidad para gestionar tanto datos estructurados como semiestructurados.

La arquitectura de la base de datos se organiza mediante una separación lógica por schemas, lo que permite aislar los distintos módulos del sistema:

- auth: gestión de autenticación y inicio de sesión para el juego y la aplicación
- game: almacenamiento de los datos principales del juego, como progreso y estado del jugador
- community: gestión de los datos de la plataforma social, incluyendo publicaciones, comentarios e interacciones
- chat: almacenamiento de mensajes y conversaciones entre los distintos sistemas

El modelo de datos combina estructuras relacionales con campos JSONB, lo que permite almacenar información dinámica como configuraciones o metadatos sin perder consistencia ni eficiencia en las consultas.

Además, se han implementado índices en los campos más consultados con el objetivo de optimizar el rendimiento y mejorar la velocidad de recuperación de datos en escenarios de alta frecuencia de acceso.

5.2 FastApi

FastAPI se utiliza en este proyecto como un backend ligero dedicado al sistema de juego, encargado de la lógica principal relacionada con los datos de los jugadores.

Funciones principales:
- Gestión de datos del jugador: almacenamiento y actualización de estadísticas, progreso y estado del juego
- API de alto rendimiento: exposición de interfaces REST de baja latencia para la interacción en tiempo real
- Validación y transformación de datos: uso de Pydantic para validar solicitudes y realizar el mapeo automático entre modelos y ORM
- Autenticación y seguridad: implementación de JWT para control de acceso, junto con cifrado de contraseñas para proteger las cuentas de usuario
- Registro y validación de usuarios: verificación de formatos como el correo electrónico y validación de datos básicos
- Procesamiento asíncrono: soporte para peticiones concurrentes mediante APIs asíncronas, mejorando el rendimiento del sistema

FastAPI actúa como un módulo independiente del sistema de juego, completamente desacoplado del backend de comunidad. Esto evita sobrecargar el sistema principal y permite una mayor escalabilidad, además de una clara separación de responsabilidades.

5.3 Springboot

Spring Boot constituye el núcleo del backend de la aplicación de comunidad, encargado de implementar todas las funcionalidades sociales y la gestión de contenido.

El sistema sigue un modelo de arquitectura en capas, que incluye:

- Capa de controladores: exposición de APIs REST
- Capa de servicios: procesamiento de la lógica de negocio
- Capa de repositorios: acceso y persistencia de datos mediante JPA

Funciones principales:
- Gestión de usuarios y autenticación basada en JWT
- Sistema de publicaciones y comentarios
- Validación de datos y control de flujos de negocio
- Comunicación con la base de datos PostgreSQL

Ventajas técnicas:
- Alta escalabilidad y mantenibilidad
- Estructura clara basada en estándares de la industria
- Sólidos mecanismos de seguridad y validación de datos
- Integración nativa con el ecosistema Java

Spring Boot actúa como el backend principal del sistema de comunidad, centralizando toda la lógica social y de gestión de contenido, y sirviendo como base estructural del módulo comunitario.

6. Análisis

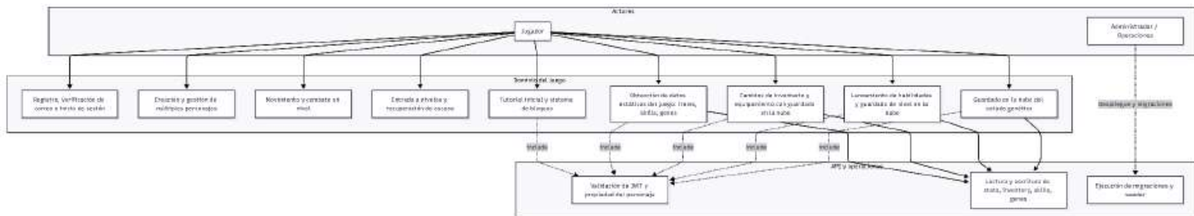
6.1 Casos de uso

6.1.1 Juego

Código	Nombre del caso de uso	Actor principal	Descripción	Resultado esperado
CU 1	Registro de usuario	Visitante	El usuario se registra con correo, nombre y contraseña.	Usuario creado correctamente en la base de datos.
CU 2	Inicio de sesión	Usuario	Autenticación mediante credenciales y JWT.	Acceso concedido y datos del jugador cargados.
CU 3	Selección de personaje	Usuario	El jugador selecciona uno de los personajes disponibles.	Personaje activo asignado correctamente.
CU 4	Control del personaje	Usuario	Movimiento, interacción y acciones mediante teclado/ratón.	El personaje responde correctamente.
CU 5	Sistema de combate	Usuario	Combate contra enemigos mediante armas y habilidades.	Daño calculado y estado actualizado.
CU 6	Sistema de habilidades	Usuario	Uso y desbloqueo de habilidades en combate.	Habilidades ejecutadas correctamente.
CU 7	Sistema genético	Usuario	Selección de rutas evolutivas del personaje.	Atributos modificados según evolución.
CU 8	Sistema de núcleo	Usuario	Mejora avanzada de atributos y capacidades.	Potenciaciones aplicadas correctamente.
CU 9	Sistema de niveles	Usuario	Ganancia de experiencia y subida de nivel.	Nivel actualizado y recompensas desbloqueadas.
CU 10	Misiones y progreso	Usuario	Ejecución de misiones principales y secundarias.（aún no se ha implementado）	Progreso de historia actualizado.
CU 11	Inventario	Usuario	Gestión de objetos, armas y recursos.	Inventario sincronizado con backend.
CU 12	Equipamiento	Usuario	Equipar o cambiar armas y objetos.	Estado del personaje actualizado.
CU 13	Sistema de tutorial	Usuario	Introducción a mecánicas básicas del juego.	Progreso del tutorial registrado.
CU 14	Sistema de bloqueo (gate system)	Usuario	Restricción de contenido según progreso.	Acceso desbloqueado progresivamente.
CU	Guardado en	Usuario	Persistencia de datos del jugador en	Datos almacenados

15	la nube		backend.	correctamente.
CU 16	Recuperación de partida	Usuario	Carga del estado anterior del jugador.	Estado restaurado correctamente.
CU 17	Cambios de escena	Usuario	Transición entre mapas y zonas del juego.	Escena cargada sin errores.
CU 18	Interacción con NPC	Usuario	Diálogo e interacción con personajes no jugables.	Eventos y diálogos activados.
CU 19	Sistema de IA de enemigos	Sistema	Control de enemigos mediante FSM/AI.	Los Enemigos reaccionan correctamente.
CU 20	Obtención de datos estáticos	Sistema	Carga de items, skills, genes desde API.	Datos sincronizados correctamente.

Game Case.png



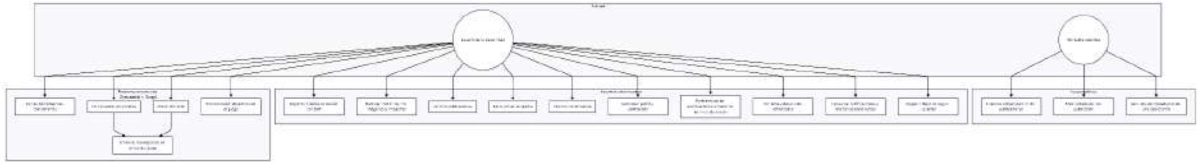
6.1.2 Comunidad

Código	Nombre del caso de uso	Actor principal	Descripción	Resultado esperado
CU21	Ver publicaciones públicas	Visitante	Visualización del feed público de la comunidad.	Lista de publicaciones cargada correctamente.
CU22	Ver detalle de publicación	Visitante / Usuario	Acceso al contenido completo de una publicación.	Publicación mostrada correctamente.
CU23	Registro / inicio de sesión en comunidad	Usuario	Autenticación mediante JWT para acceder a funciones sociales.	Usuario autenticado correctamente.
CU24	Crear publicación	Usuario	Publicación de contenido con texto, imágenes y etiquetas.	Publicación almacenada en el sistema.
CU25	Editar publicación	Usuario	Modificación de contenido previamente publicado.	Cambios actualizados correctamente.
CU26	Eliminar publicación	Usuario	Eliminación de publicaciones propias.	Publicación eliminada del sistema.
CU27	Dar / quitar	Usuario	Interacción social mediante “like”	Estado actualizado

	“me gusta”		o “unlike”.	correctamente.
CU28	Comentar publicación	Usuario	Añadir comentarios a publicaciones.	Comentario almacenado y visible.
CU29	Ver comentarios	Usuario / Visitante	Visualización de comentarios asociados a una publicación.	Comentarios cargados correctamente.
CU30	Seguir / dejar de seguir usuarios	Usuario	Gestión de relaciones sociales entre usuarios.	Relación actualizada en la base de datos.
CU31	Sistema de notificaciones	Usuario	Recepción de eventos como likes, comentarios o seguimiento.	Notificación generada correctamente.
CU32	Marcar notificaciones como leídas	Usuario	Gestión del estado de notificaciones.	Estado actualizado correctamente.
CU33	Gestión de perfil	Usuario	Modificación de información personal y contraseña.	Perfil actualizado correctamente.
CU34	Historial de actividad	Usuario	Visualización del historial de acciones del usuario.	Datos mostrados correctamente.
CU35	Administración de usuarios	Administrador	Gestión de cuentas de usuario.	Cambios aplicados globalmente.
CU36	Moderación de contenido	Administrador	Eliminación o revisión de contenido inapropiado.	Contenido moderado correctamente.
CU37	Uso de herramientas comunitarias	Usuario	Acceso a herramientas internas de la comunidad (estadísticas, utilidades sociales, etc.).	Herramienta ejecutada correctamente.
CU38	Participación en eventos de la comunidad	Usuario	Participación en eventos organizados dentro del sistema.	Evento registrado y progreso actualizado.
CU39	Sistema de vinculación con el juego	Usuario	Sincronización de datos entre comunidad y juego (progreso, logros).	Datos sincronizados correctamente.
CU40	Check-in diario y recompensas	Usuario	Inicio de sesión diario en la comunidad para obtener recompensas.	Recompensa generada correctamente.

CU41	Envío de recompensas al correo del juego	Sistema	El sistema envía recompensas obtenidas a la bandeja de correo del juego.	Recompensas recibidas en el sistema de juego.

Community Case.png



6.2 Requisitos funcionales

Los requisitos funcionales describen las funciones y comportamientos concretos que el sistema debe proporcionar. En este proyecto, incluyen la gestión de usuarios, la lógica del juego, el sistema de inventario, el combate, la progresión del personaje, así como la integración con el backend y el sistema de comunidad. Estos requisitos definen qué debe hacer el sistema desde la perspectiva del usuario y de la interacción con las distintas mecánicas del juego.

Codigo	Descripción del requisito funcional	Tipo
RF1	Registro de nuevos usuarios mediante la introducción de credenciales básicas (nombre, correo y contraseña).	Gestión de usuarios
RF2	Autenticación de usuarios y acceso al sistema mediante credenciales válidas.	Gestión de usuarios
RF3	Validación de la unicidad del nombre de usuario durante el proceso de registro.	Validación
RF4	Cifrado y verificación segura de contraseñas antes de generar credenciales de acceso basadas en JWT.	Seguridad
RF5	Interacción del jugador con objetos del entorno para la obtención de ítems.	Inventario
RF6	Gestión completa del inventario, incluyendo la adición, uso y eliminación de objetos.	Inventario
RF7	Activación de efectos asociados al uso de objetos, como la recuperación de salud u otros estados.	Inventario
RF8	Obtención de objetos mediante recompensas, derrotas de enemigos o compras dentro del juego.	Inventario
RF9	Definición de atributos del personaje, incluyendo estadísticas, estados y niveles.	Datos
RF10	Comunicación entre cliente (Godot) y backend mediante APIs para operaciones CRUD.	Backend

RF11	Almacenamiento de datos de sesión tanto en el cliente local como en la base de datos.	Persistencia
RF12	Persistencia automática del progreso del jugador en la base de datos.	Persistencia
RF13	Visualización detallada de habilidades, incluyendo nivel, efecto y tiempo de reutilización.	Habilidades
RF14	Desbloqueo de habilidades condicionado a recursos disponibles o nivel del personaje.	Habilidades
RF15	Mejora progresiva de habilidades mediante incremento de nivel.	Habilidades
RF16	Integración de las habilidades dentro del sistema de combate del juego.	Combate
RF17	Aplicación de tiempos de reutilización (cooldown) tras el uso de habilidades.	Habilidades

6.3 Requisitos no funcionales

Los requisitos no funcionales definen las propiedades de calidad del sistema, más allá de sus funcionalidades específicas. En este proyecto, abarcan aspectos como el rendimiento, la seguridad, la escalabilidad, la mantenibilidad, la portabilidad y la consistencia de los datos. Estos requisitos garantizan que el sistema funcione de manera estable, eficiente y coherente, asegurando una experiencia de usuario adecuada.

Código	Categoría	Descripción del requisito no funcional
RNF1	Rendimiento	El tiempo de respuesta de las APIs backend no deberá superar los 500 ms en condiciones normales.
RNF2	Portabilidad	El sistema será compatible con Windows, Linux y macOS.
RNF3	Usabilidad	La interfaz de usuario deberá ser clara, intuitiva y consistente.
RNF4	Usabilidad	Todas las interacciones deberán proporcionar retroalimentación visual (confirmación, cancelación, carga).
RNF5	Usabilidad	Los mensajes del sistema deberán ser comprensibles y evitar terminología técnica innecesaria.
RNF6	Mantenibilidad	El código seguirá convenciones estándar de nomenclatura y documentación.
RNF7	Arquitectura	El sistema adoptará un diseño modular con bajo acoplamiento entre subsistemas.
RNF8	Escalabilidad	El sistema permitirá la incorporación de nuevos módulos sin cambios estructurales significativos.
RNF9	Fiabilidad	Los datos del usuario se guardarán automáticamente en el servidor para evitar pérdidas.
RNF10	Seguridad	Se implementará autenticación segura (JWT) y cifrado de contraseñas.

RNF11	Disponibilidad	Los servicios backend deberán garantizar alta disponibilidad y tolerancia a fallos.
RNF12	Consistencia	Se garantizará la coherencia entre datos locales y datos en la nube.
RNF13	Testabilidad	El sistema permitirá pruebas automatizadas.
RNF14	Despliegue	El sistema podrá desplegarse en entornos virtualizados y será adaptable a contenedores (Docker).

6.4 Análisis de alternativas tecnológicas

La selección de herramientas, bibliotecas y tecnologías para este proyecto se basó en los siguientes criterios:

Costo: Priorización de soluciones gratuitas y de código abierto

Control del sistema: Modificabilidad y adaptabilidad técnica

Escalabilidad: Potencial de crecimiento futuro

Facilidad de desarrollo: Curva de aprendizaje y eficiencia del desarrollo

Compatibilidad arquitectónica: Integración con otras partes del sistema

6.4.1 Motor de juego

Se eligió Godot por ser de código abierto (lo que garantiza independencia tecnológica), contar con una arquitectura basada en nodos que permite una alta modularidad, tener un bajo coste adecuado para entornos académicos y facilitar la integración con sistemas personalizados, además de que sus funcionalidades continúan ampliándose y perfeccionándose gracias a la constante evolución de su comunidad.

Criterio	Godot	Unity	Unreal Engine
Licencia	Código abierto	Comercial	Comercial + reparto de ingresos
Lenguaje	GDScript / C#	C#	C++ / Blueprint
Curva de aprendizaje	Baja	Media	Alta
Control	Alto	Medio	Bajo
Tipo de proyectos	Pequeños y medianos	Todo tipo	AAA

6.4.2 Modelado 3D

Se eligió Blender porque es una herramienta gratuita y de código abierto con un conjunto de funcionalidades muy completo, suficiente para satisfacer los requisitos del proyecto sin costes adicionales. Además, cuenta con una comunidad muy activa que facilita el aprendizaje y la resolución de problemas, y ofrece una buena integración con otros motores y herramientas de desarrollo. Asimismo, dentro de su ecosistema se pueden encontrar numerosos plugins y recursos que simplifican y agilizan el proceso de modelado.

Criterio	Blender	Autodesk Maya / Autodesk 3ds Max
Coste	Gratis	Comercial
Funcionalidad	Completa	Más profesional
Comunidad	Muy activa	Activa
Integración	Buena	Muy buena

6.4.3 Backend

Se adoptó una arquitectura híbrida utilizando FastAPI y Spring Boot para aprovechar las ventajas de ambos frameworks en diferentes módulos del sistema.

FastAPI

Se empleó para el desarrollo de servicios backend ligeros y de alto rendimiento, especialmente aquellos relacionados con la lógica del juego, como el sistema de guardado, el procesamiento de datos en tiempo real y servicios de baja latencia. Sus principales ventajas son la rapidez de desarrollo, la sintaxis sencilla y su excelente rendimiento en aplicaciones asíncronas.

Spring Boot

Se utilizó para construir el sistema principal de comunidad, el cual requiere una arquitectura más estructurada y escalable. Spring Boot ofrece un modelo de desarrollo en capas que facilita la organización y el mantenimiento del código, además de proporcionar un sistema de seguridad maduro, adecuado para la gestión de usuarios, autenticación y control de permisos.

Motivos para no elegir otras tecnologías backend

Otras alternativas como Node.js o frameworks más ligeros como Flask no resultaban óptimas para este proyecto, ya que presentan limitaciones en cuanto a estructuración en sistemas grandes, seguridad empresarial o rendimiento en escenarios altamente concurrentes. Dado que el proyecto combina servicios de alto rendimiento para lógica de juego y un sistema de comunidad complejo, una única tecnología no podía satisfacer simultáneamente todos los requisitos de rendimiento

y arquitectura. La arquitectura híbrida permite seleccionar la herramienta más adecuada para cada módulo, mejorando la estabilidad, escalabilidad y mantenibilidad del sistema en su conjunto.

Criterio	FastAPI	Spring Boot	Node.js
Lenguaje	Python	Java	JavaScript
Rendimiento	Alto	Alto	Medio
Estructura	Ligera	Empresarial	Flexible
Tipado	Fuerte (Pydantic)	Fuerte	Débil
Complejidad	Baja	Alta	Media

6.4.4 Base de datos

Se eligió PostgreSQL como sistema de base de datos principal debido a su capacidad para adaptarse a distintos tipos de carga de trabajo dentro del proyecto.

Soporte JSONB: permite almacenar datos en formato semi-estructurado, lo que resulta ideal para la gestión de datos de juego con estructuras dinámicas y variables.
Modelo relacional: proporciona una base sólida para el sistema de comunidad, donde se requieren relaciones complejas entre usuarios, publicaciones e interacciones.
Multi-schema: facilita la separación lógica de distintos dominios de negocio dentro de una misma base de datos, mejorando la organización y escalabilidad del sistema.

En conjunto, PostgreSQL ofrece un equilibrio entre flexibilidad y robustez, lo que lo convierte en una solución adecuada para un sistema híbrido como el desarrollado en este proyecto.

Criterio	PostgreSQL	MySQL	MongoDB
Tipo	Relacional + JSONB	Relacional	NoSQL
Flexibilidad	Alta	Media	Muy alta
Consistencia	Alta	Alta	Media
JSON	Sí (JSONB)	Limitado	Nativo

6.4.5 Frontend

Se eligió la combinación de Ionic con React debido a su equilibrio entre eficiencia de desarrollo, reutilización de código y facilidad de integración con el ecosistema web existente del proyecto.

En comparación con la alternativa basada en Vue.js, React ofrece un ecosistema más amplio, mayor madurez en el desarrollo de aplicaciones complejas y una mejor integración con herramientas modernas del entorno JavaScript, lo que resulta especialmente útil en proyectos con múltiples módulos interactivos como este.

Por otro lado, aunque Flutter proporciona un rendimiento superior y una experiencia más cercana al desarrollo nativo, su curva de aprendizaje es más alta y su integración con ciertas tecnologías web y librerías existentes puede ser más limitada en comparación con una solución basada en tecnologías web híbridas.

En este contexto, la combinación de Ionic y React permite reutilizar tecnologías web estándar (HTML, CSS y JavaScript), acelerar el desarrollo y mantener una arquitectura flexible, lo que la convierte en la opción más adecuada para un proyecto con necesidades de iteración rápida y compatibilidad multiplataforma.

Criterio	Ionic + React	Ionic + Vue.js	Flutter
Tipo	Web híbrido	Web híbrido	Multiplataforma nativa
Reutilización	Alta	Alta	Media
Curva de aprendizaje	Media	Baja	Alta
Soporte móvil	Bueno (Ionic + React)	Bueno (Ionic + Vue)	Excelente

6.5 Análisis de usuarios y roles

Rol	Descripción	Permisos principales
Administrador	El usuario de máxima autoridad del sistema, responsable de supervisar y mantener todo el sistema.	- Administrar cuentas de usuario (CRUD) - Administrar bibliotecas de objetos y habilidades - Ver registros y el estado del sistema
Usuario	Utiliza el sistema para iniciar sesión, interactuar con juegos y administrar mochilas y habilidades.	- Inicia sesión y crea una cuenta - Juega
Visitante	Prueba el juego y comprende la interfaz y el funcionamiento.	Sin permiso / Navegación limitada y modo de prueba.

7. La producción artística

En este proyecto, la producción artística desempeña un papel secundario en comparación con la programación y el diseño de sistemas. La mayor parte de los recursos visuales fueron generados mediante herramientas de inteligencia artificial, mientras que una parte menor fue creada manualmente mediante modelado y diseño propio, especialmente en escenarios y personajes principales. Además, se desarrollaron algunos efectos visuales y texturas dentro del motor Godot.

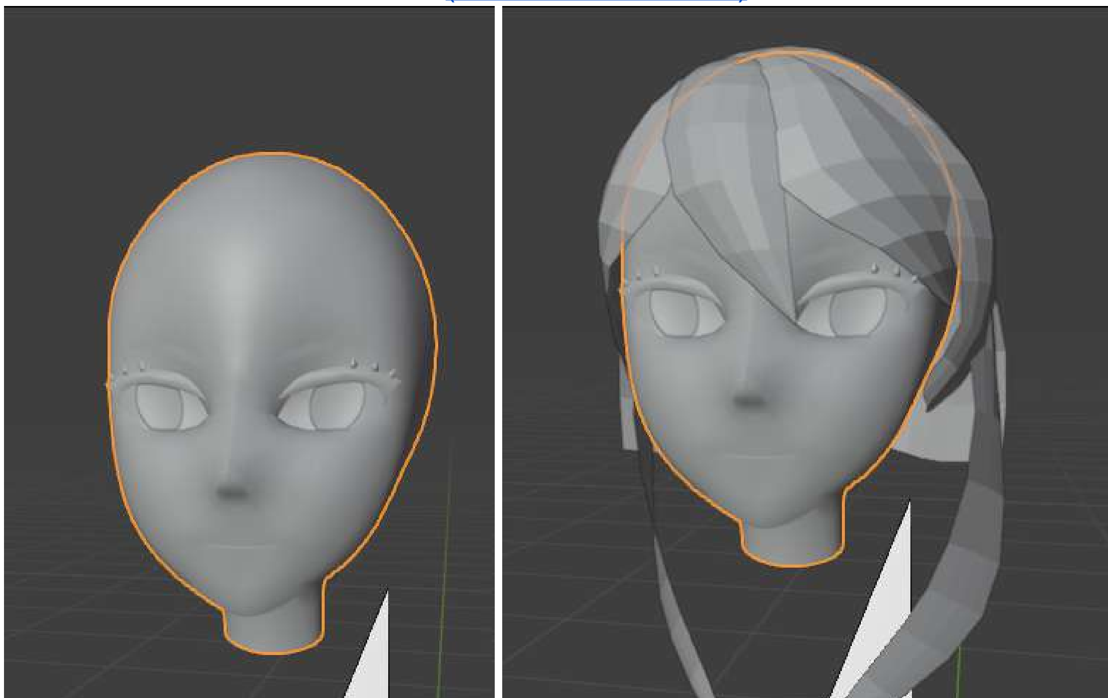
7.1 Modelo

7.1.1 Creación de personajes

El diseño inicial del proyecto contemplaba la creación de cinco personajes basados en animales. En esta versión del proyecto, se desarrolló principalmente un personaje femenino como modelo principal.

Para la fase de diseño conceptual, se utilizó inteligencia artificial (Gemini) para generar vistas ortogonales del personaje (frontal, lateral y posterior). Posteriormente, estas referencias fueron ajustadas manualmente según la dirección artística del proyecto.

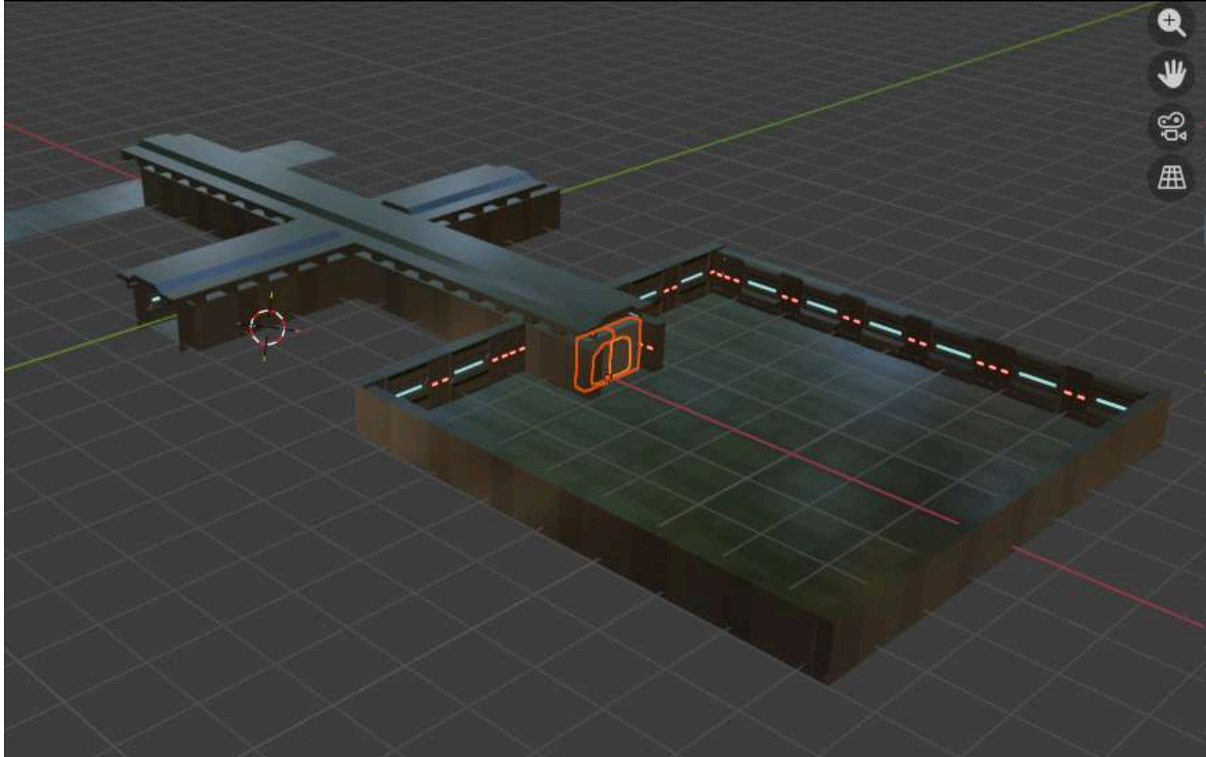
El modelado 3D se realizó en Blender, siguiendo flujos de trabajo de modelado poligonal básico. Para el proceso de aprendizaje y optimización del modelo, se consultaron tutoriales externos (ver sección Blender).

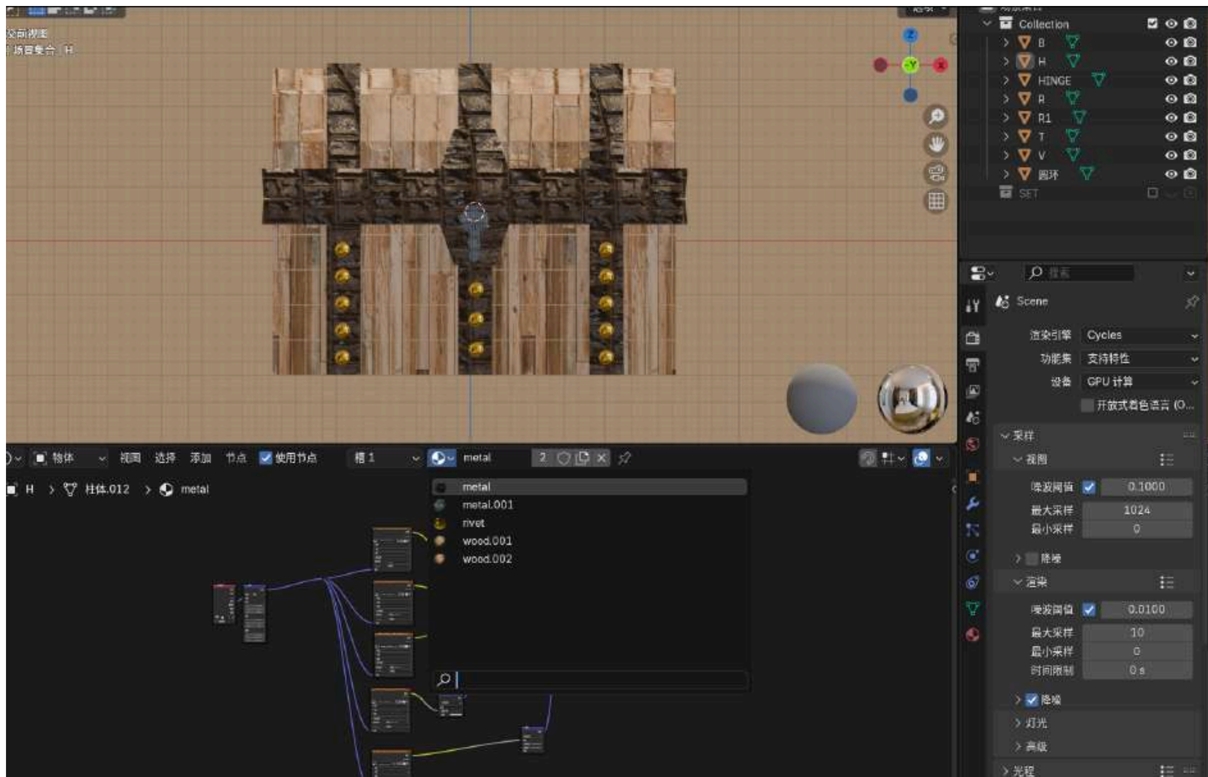


7.1.2 Escena

El diseño de escenarios se basó parcialmente en activos del proyecto anterior, reutilizando una parte de los modelos existentes. A partir de estos recursos, se seleccionaron elementos específicos y se reorganizaron para construir una escena inicial destinada al nivel de tutorial.

Este enfoque permitió reducir el tiempo de producción y mantener coherencia visual en la etapa temprana del desarrollo.



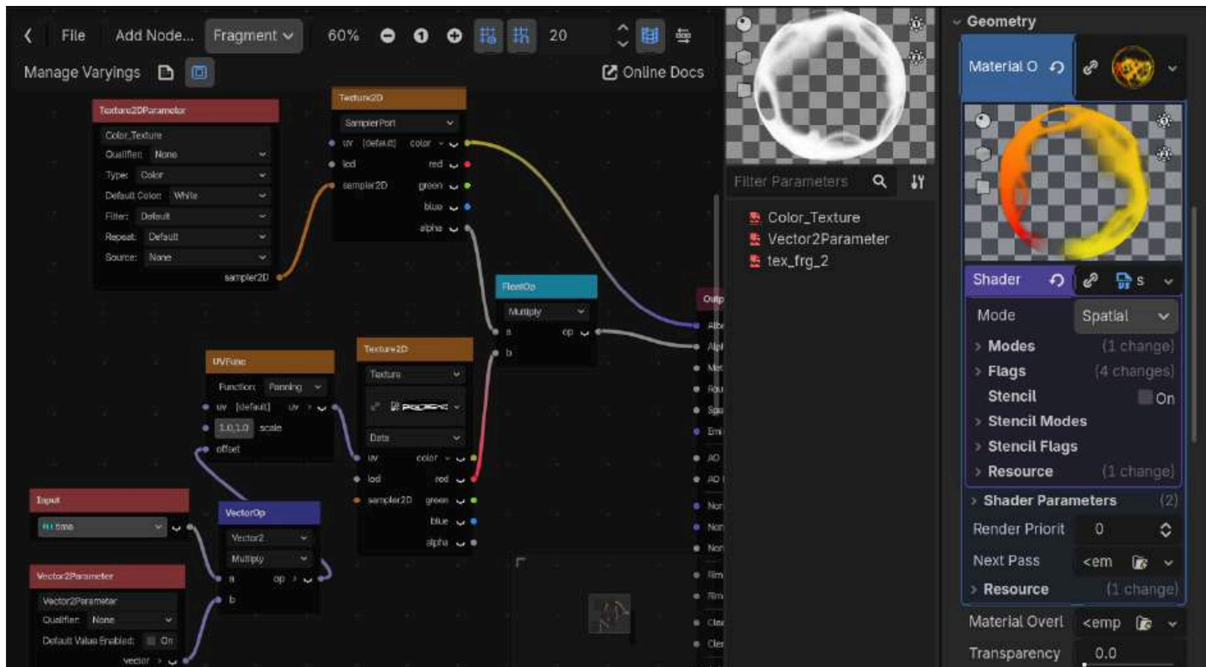


7.2 SFX

En este proyecto, los efectos visuales (SFX) se implementaron mediante una combinación de texturas generadas en Krita y sistemas de shaders dentro del motor Godot.

Las texturas creadas en Krita se utilizaron principalmente para habilidades, efectos de UI y elementos visuales del juego. Estas texturas fueron posteriormente integradas en shaders para lograr animaciones dinámicas y efectos de iluminación.

Además, se desarrollaron efectos directamente mediante código y shaders, lo que permitió generar efectos visuales en tiempo real tanto en habilidades como en la interfaz de usuario.



7.3 AI

Durante el desarrollo del proyecto, una parte significativa de los recursos visuales fue generada mediante herramientas de inteligencia artificial, incluyendo iconos de interfaz, fondos y elementos decorativos.

Estos recursos fueron utilizados principalmente en la fase de prototipado y pruebas del sistema, con el objetivo de acelerar el desarrollo visual y validar la dirección artística del proyecto.

Se establece que estos assets son temporales y serán reemplazados en futuras iteraciones del proyecto por recursos originales optimizados y consistentes con la dirección artística final.

8. Proceso de desarrollo e implementación

8.1 juego

8.1.0 Entrada del proyecto y arquitectura Autoload

El cliente del juego de este proyecto está desarrollado con Godot, donde la entrada del proyecto y la configuración global se gestionan de forma centralizada mediante project.godot. A través del sistema [autoload], se registran varios gestores globales (como ApiManager, GameDataManager, TutorialManager, SceneManager, UiManager y AudioManager).

El objetivo de esta arquitectura es mantener funcionalidades transversales reutilizables como singletons dentro de la raíz del SceneTree, evitando la pérdida de estado o la reinicialización innecesaria durante los cambios de escena, al mismo tiempo que se mejora la modularidad y la mantenibilidad del sistema.

En cuanto al sistema de entrada, project.godot define de forma centralizada las acciones mediante el bloque [input], incluyendo mapeos como movement (movimiento con WASD) y shoot (disparo con el botón izquierdo del ratón). Esta unificación permite que sistemas como el tutorial (8.1.1) y la interfaz de usuario (8.1.4) compartan los mismos identificadores de acción para la lógica de control, reduciendo inconsistencias y mejorando la escalabilidad del sistema de interacción.

8.1.1 Sistema de tutorial

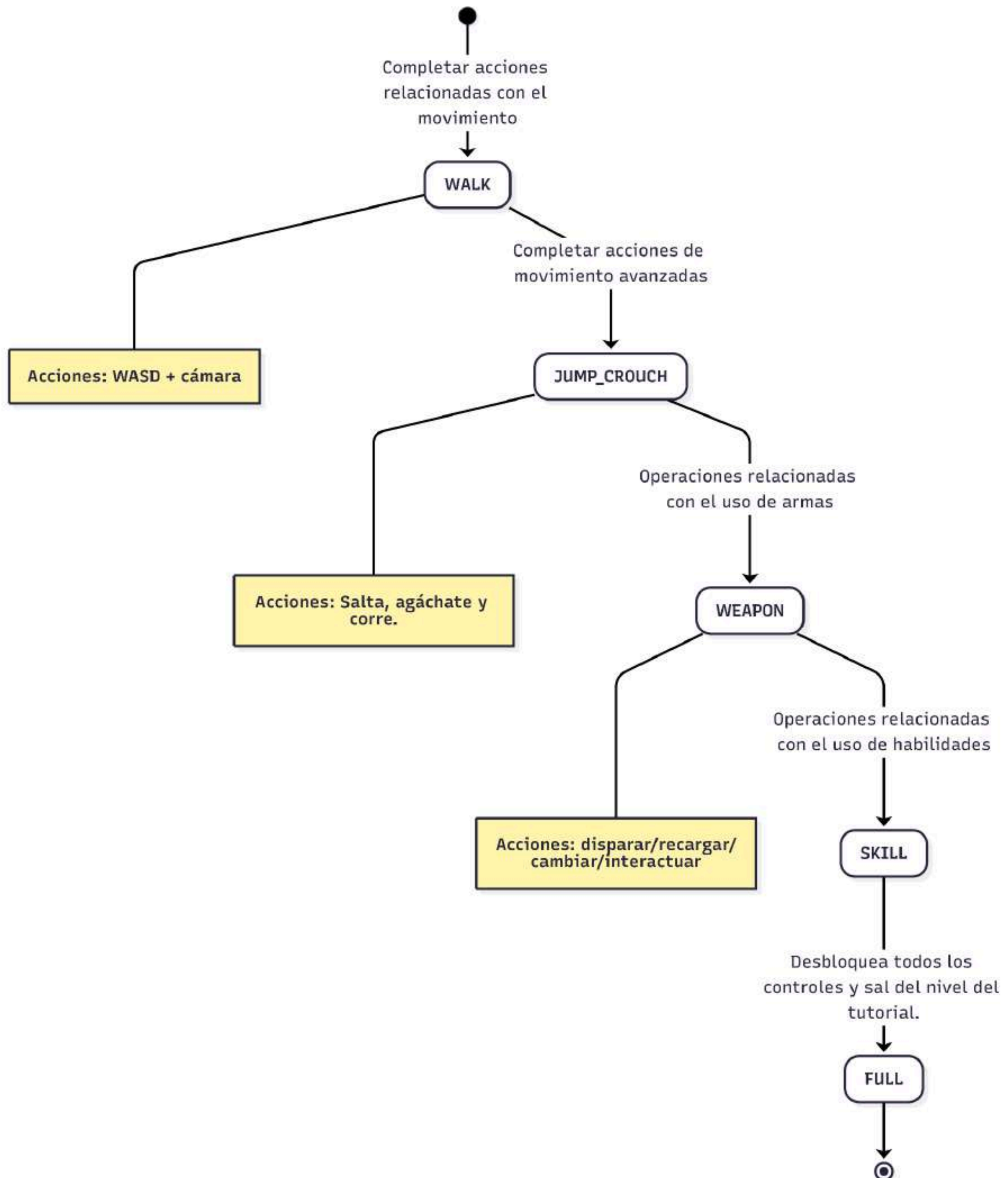
El sistema de tutorial tiene como objetivo reducir la curva de aprendizaje del jugador y garantizar el control del flujo principal del juego. En un juego de disparos de acción, el jugador debe aprender rápidamente mecánicas como movimiento, puntería, disparo, uso de habilidades, recogida de objetos y gestión de inventario. Sin un sistema de guía adecuado, el jugador puede experimentar una falta de orientación al entrar en el juego principal, afectando negativamente la experiencia. Por ello, el proyecto adopta una combinación de “guía por fases” y un sistema de bloqueo progresivo.

8.1.1.1 Planificación del proceso tutorial

El tutorial se divide en varias fases secuenciales, donde cada fase está compuesta por: “condición de activación – objetivo – verificación – retroalimentación”.

Condición de activación: entrada en zonas específicas, interacción con objetos, eliminación de enemigos o recogida de ítems
Verificación: validación de que el jugador ha completado el objetivo (movimiento, disparo, uso de habilidades, etc.)
Retroalimentación: indicadores visuales o de interfaz que guían al jugador hacia la siguiente fase

(Figura 8-1: máquina de estados del tutorial)



Relación entre fases y acciones

Cada fase se controla mediante conjuntos de acciones definidos en el sistema, donde una acción se considera completada tras su primera activación.

Fase	Acciones destacadas	Objetivo	Condición de finalización

WALK	Movimiento (WASD)	Aprender desplazamiento básico	Activación de cualquier tecla de movimiento
JUMP_CROUCH	Saltar / agacharse / correr	Aprender movilidad básica	Todas las acciones ejecutadas al menos una vez
WEAPON	Disparar / recargar / cambiar arma / interacción	Aprender combate básico	Todas las acciones del sistema de armas
SKILL	Habilidades 1/2/3	Uso de habilidades	Todas las habilidades ejecutadas
FULL	Ninguna	Final del tutorial	Desbloqueo completo del sistema

8.1.1.2 Sistema de bloqueo

Para evitar que el jugador acceda prematuramente a sistemas complejos, se implementa un sistema de bloqueo progresivo en el menú principal y la selección de modos.
Mientras el tutorial no se haya completado:

- Algunas funciones permanecen deshabilitadas
- La interfaz muestra mensajes de orientación
- El acceso a sistemas avanzados depende del progreso del tutorial

Este enfoque garantiza un recorrido de aprendizaje estructurado y evita la sobrecarga cognitiva en las primeras fases del juego.

8.1.2 Comportamiento hostil e inteligencia artificial

El sistema de inteligencia artificial del proyecto sigue una arquitectura en capas: la capa inferior se encarga de la percepción y navegación, la capa intermedia gestiona la toma de decisiones y la capa superior controla las estrategias de combate y la coordinación de comportamientos. Además, se incorpora un sistema de depuración visual para analizar el comportamiento de la IA y sus decisiones, reduciendo la dificultad de depuración de sistemas complejos.

8.1.2.1 Comportamiento del personaje y máquina de estados

(1) Mapeo de entrada y modelo de movimiento

El juego soporta entrada combinada de teclado y ratón. El modelo de movimiento define parámetros clave como velocidad, aceleración, fricción, sprint y rotación, además de manejar transiciones suaves ante cambios o interrupciones de entrada.

En un sistema de acción en tercera persona, la relación entre la cámara y la orientación del personaje se define explícitamente, por ejemplo:

- Movimiento relativo a la cámara
- Orientación hacia el objetivo en combate

Este diseño garantiza una experiencia de control consistente y predecible.

```
move_forward
    W
    Up (Physical)
move_left
    A (Physical)
    Left (Physical)
move_back
    S (Physical)
    Down
move_right
    D (Physical)
    Right (Physical)
```

```
const JUMP_VELOCITY: float = 5.0
const SPEED_WALK: float = 5.0
const SPEED_RUN: float = 8.0
const SPEED_CROUCH: float = 2.0
const SLIDE_TIME_MAX: float = 1.0
const SLIDE_SPEED: float = 10.0
const HEAD_HEIGHT: float = 1.0

var speed_lerp: float = 10.0
var air_lerp: float = 3.0
```

(2) Sistema de colisiones e interacción

El sistema de colisiones se organiza en capas, separando entidades como personajes, enemigos, proyectiles, objetos interactivos y triggers.

Las interacciones se gestionan mediante detección por rayos (Raycast) y áreas (Area Detection), y son procesadas a través de un sistema centralizado de eventos, incluyendo:

- Eventos de recogida de objetos
- Eventos de diálogo
- Teletransporte y cambio de escena

Este enfoque mejora el desacoplamiento del sistema y reduce dependencias directas entre módulos.

Layer 1	World
Layer 2	Character
Layer 3	Hurtbox
Layer 4	SkillArea
Layer 5	Pickup
Layer 6	PhysicsProp
Layer 7	Interactable
Layer 8	GameplayVolume
Layer 9	Tutorial

(3) Máquina de estados de animación y eventos de frame

El sistema de animación del personaje está sincronizado con la lógica de combate mediante una máquina de estados, asegurando coherencia entre la representación visual y la lógica de daño.

Los ataques se validan mediante hitboxes, y el sistema se amplía con frames de invencibilidad y frames de aturdimiento, permitiendo un control más preciso de la respuesta del combate.

(Figura 8-2: máquina de estados del personaje y eventos de animación)

(4) Estado del personaje y orden de actualización
El sistema de estados del personaje gestiona los diferentes comportamientos del jugador, como movimiento, ataque, daño y muerte, mediante una máquina de estados que garantiza que solo un estado principal esté activo a la vez.
El orden de actualización del personaje sigue un flujo fijo:

Procesamiento de entrada → actualización del estado → movimiento y colisiones → actualización de animación → lógica de combate (ataque/daño) → renderizado.

Este orden asegura la coherencia entre controles, física y lógica de combate, evitando desincronización entre animación y mecánicas.

8.1.2.2 IA de enemigos

El objetivo de la IA enemiga es proporcionar una experiencia de combate comprensible, permitiendo que los enemigos detecten al jugador, lo persigan y lo ataquen, y que regresen a su estado inicial cuando el combate finaliza. Para ello, la lógica de la IA se divide en cuatro módulos principales: percepción, toma de decisiones, navegación y ataque, organizados mediante una máquina de estados finitos (FSM).

(1) Capa de percepción
La capa de percepción se basa en detección por distancia y eventos de impacto. La detección por distancia permite identificar rápidamente la presencia del jugador, mientras que la detección de daño garantiza coherencia en la respuesta de combate.

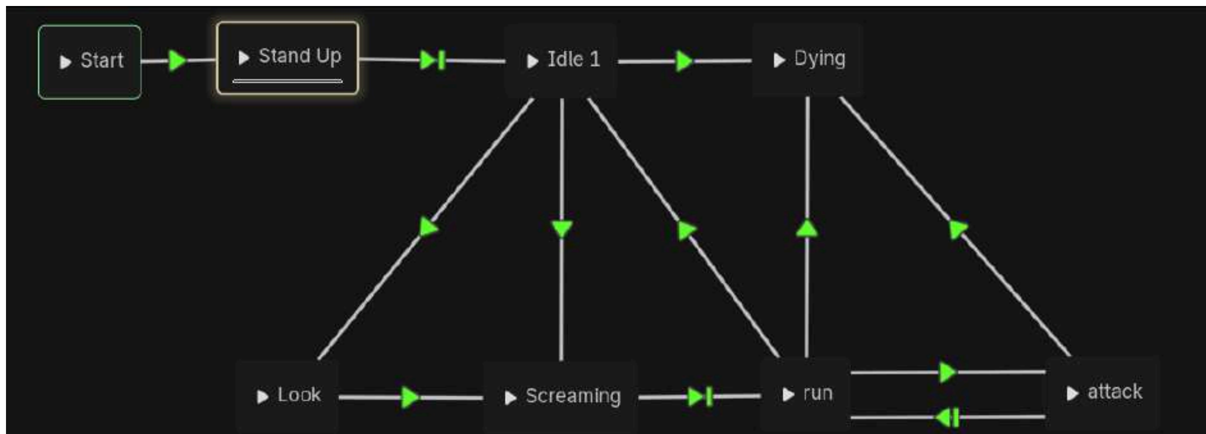
```
@export var detection_range:   float = 15.0
@export var lose_target_range: float = 22.0
## 进入该距离后从 Look 进入 Alert，并在短暂 Alert 后开始追击
@export var alert_range:       float = 8.0
```

Esta capa genera información clave como la presencia del objetivo, su posición y el nivel de amenaza (aggro), que es utilizada por la capa de decisión. En el futuro, se prevé la incorporación de un sistema de visión para mejorar la coherencia del comportamiento.

```
@export var threat_decay_per_second: float = 0.35
@export var threat_radius_max: float = 0.0
## 超出该距离（米）时威胁额外衰减倍率；0 表示不启用距离衰减
@export var out_of_range_decay_mult: float = 2.5
```

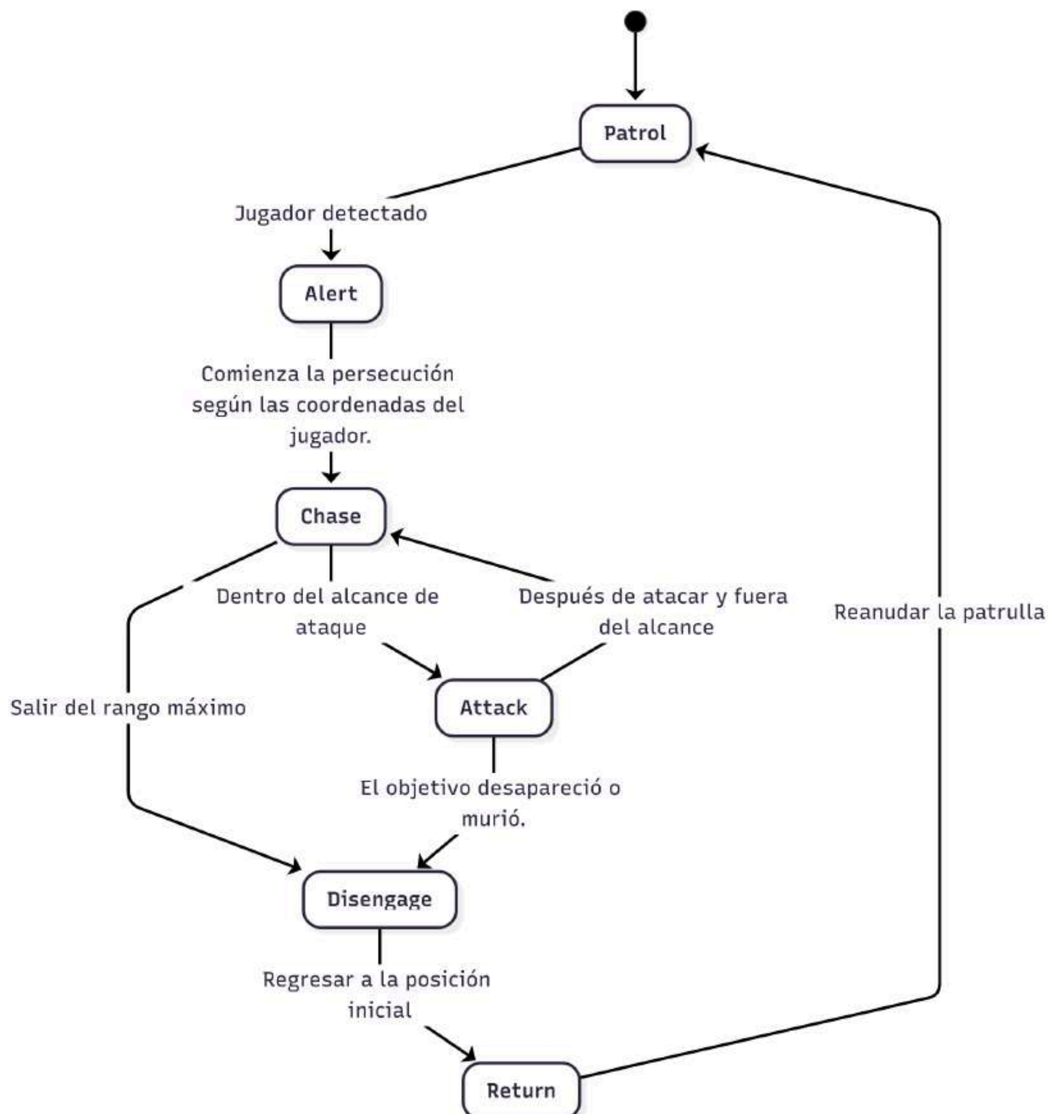
(2) Capa de decisión (FSM)
El comportamiento del enemigo se organiza mediante una máquina de estados finitos con el siguiente flujo:

Patrulla → Alerta → Persecución → Ataque → Desenganche → Regreso



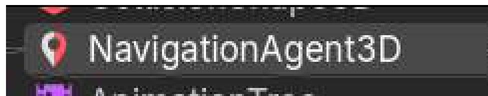
Cada estado define condiciones de entrada, salida y comportamiento interno. La FSM permite una estructura clara y fácil de depurar, aunque en sistemas más complejos puede complementarse con árboles de comportamiento para decisiones más avanzadas.

(Figura 8-3: diagrama de estados del enemigo)



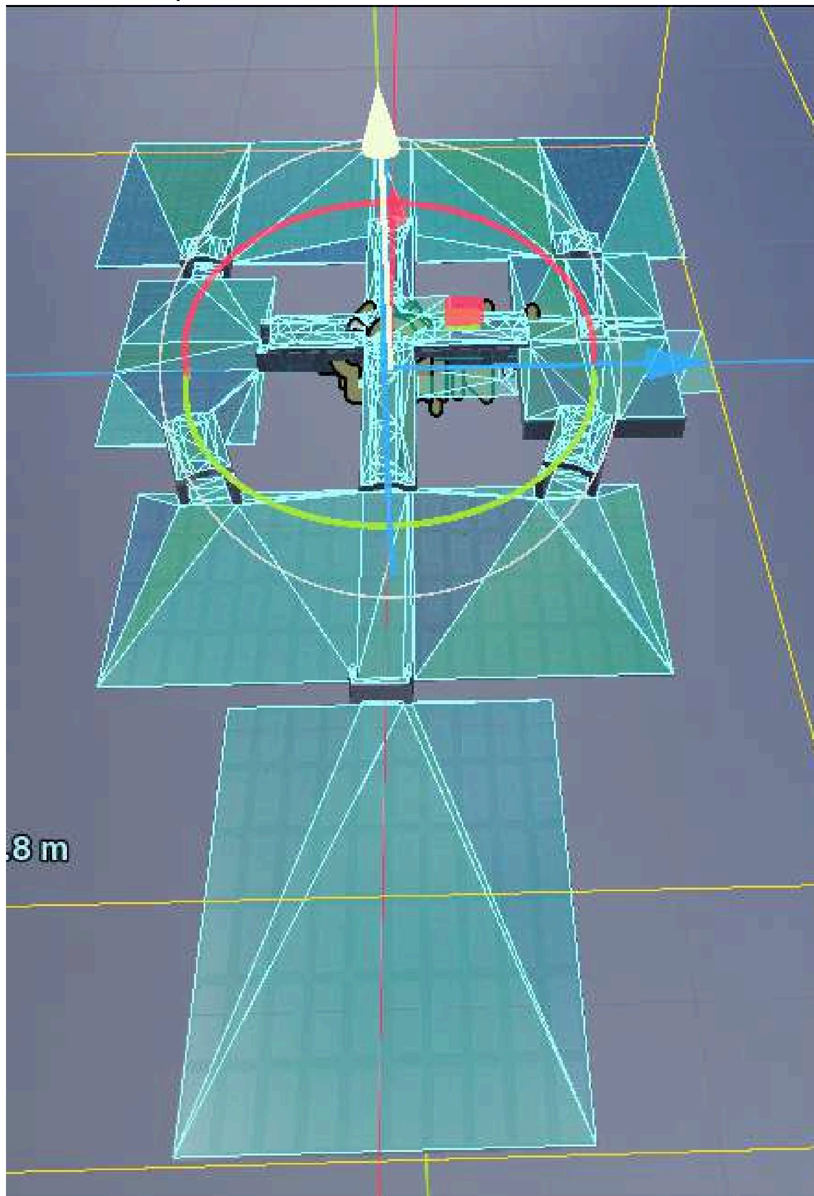
(3) Navegación y evasión
Durante la persecución, los enemigos utilizan NavMesh y agentes de navegación para el cálculo de rutas. Sin embargo, pueden aparecer problemas como vibración de movimiento o colisiones en entornos estrechos, lo que requiere ajustes mediante suavizado de ruta y recalculo dinámico.

Un agente 3D utilizado para encontrar una ruta hacia una ubicación objetivo evitando obstáculos:



Una región 3D transitable que puede utilizarse para la búsqueda de rutas:

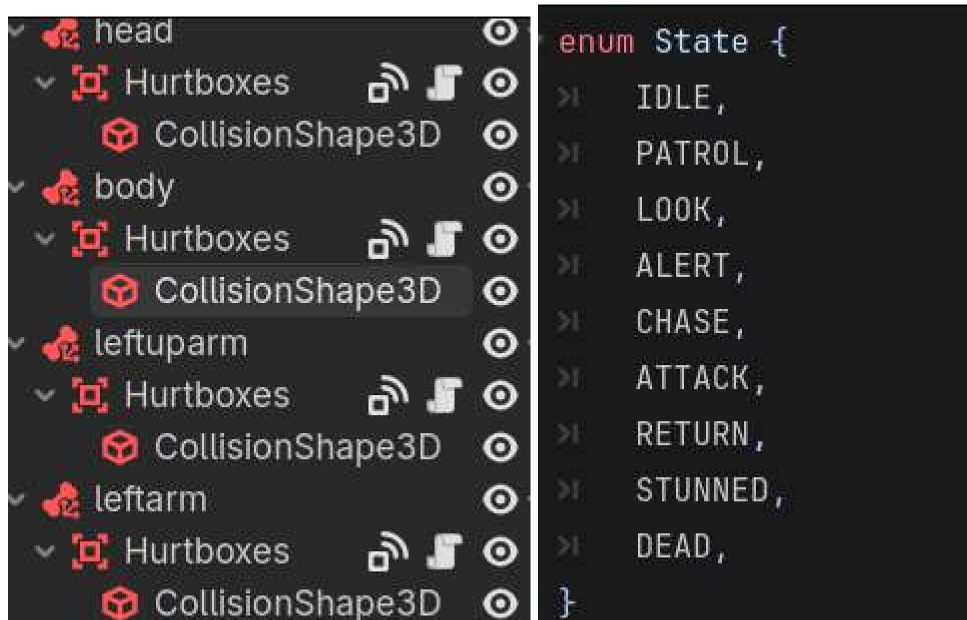
Ruta del mapa horneado:



(4) Estrategia de ataque y cooldown
El sistema de ataque incluye verificación de rango, tiempos de pre-animación y post-animación, detección de impacto y control de enfriamiento (cooldown). Para evitar comportamientos anómalos, el sistema implementa bloqueo de estado durante la ejecución de ataques.

```
@export var attack_range:      float = 2.0
## 平面近战判定在 attack_range 上的额外容差 ( 双 CharacterBody 挤开时仍应能进入攻
@export var attack_range_slack: float = 0.75
@export var attack_cooldown:   float = 1.2
@export var stun_on_hit_chance: float = 0.25
@export var stun_duration:     float = 0.8
@export var base_damage:       float = 10.0
```

Además, la sincronización entre animación y lógica de daño se realiza mediante **event frames** y **hitboxes**, garantizando coherencia entre la representación visual y la mecánica de combate.



8.1.3 Cambio de escena

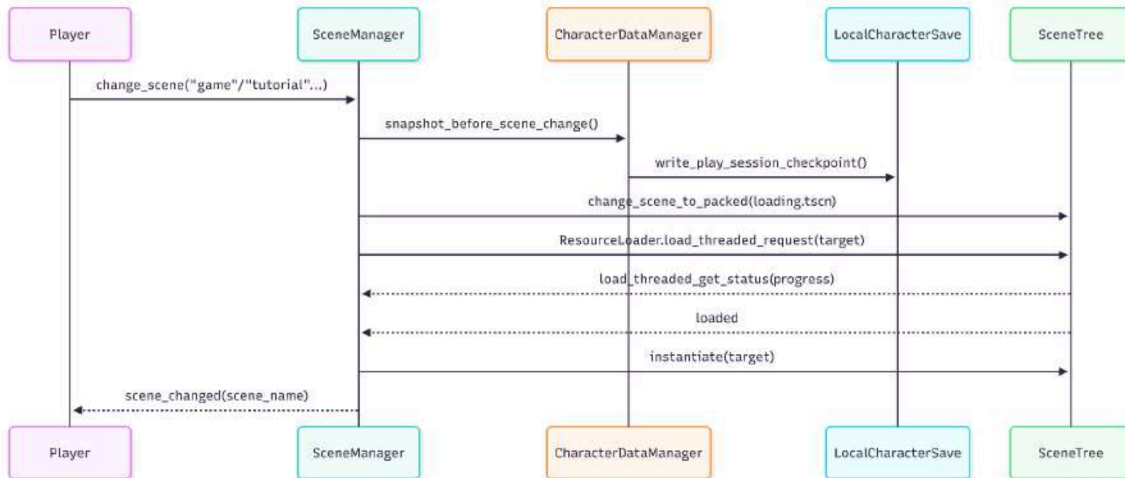
El sistema de cambio de escena implica tres aspectos clave: carga de recursos, guardado de estado y restauración de estado. Una gestión inadecuada puede provocar caídas de rendimiento o inconsistencias en el progreso del jugador. Por ello, el flujo se estructura de forma estandarizada como:
guardar → transición → descarga/carga → restaurar.

8.1.3.1 Activación y transición

El cambio de escena puede activarse mediante portales, interacciones, eventos narrativos o muerte del jugador. Las transiciones (como fundidos o pantallas de

carga) cumplen una doble función: mejorar la presentación visual y ocultar la latencia de carga asíncrona.

La pantalla de carga incluye elementos visuales y mensajes informativos, y ofrece opciones de reintento o retorno en caso de error, mejorando la robustez del sistema. (Figura 8-4: Diagrama de secuencia de cambio de escena)



8.1.3.2 Checkpoints y estado de escena

El sistema de checkpoints almacena información esencial como posición del jugador, vida y progreso de misión. En escenarios más complejos, el estado del entorno puede modelarse como datos serializables que se restauran al cargar la escena.

```
func _change_scene_internal(path: String, save_history: bool) -> void:
	CharacterDataManager.snapshot_before_scene_change()
	is_loading = true
	load_scene = path
```

8.1.3.3 Rollback y consistencia

En situaciones donde el guardado es exitoso pero la carga falla, o existen errores de sincronización, el sistema implementa mecanismos de rollback, como la restauración desde una copia local. En casos críticos, se permite regresar a la escena anterior o a un estado seguro, garantizando la continuidad del juego.

```
func on_scene_load_failed() -> void:
	var failed_path: String = load_scene
	push_error("场景加载失败: %s" % failed_path)
	GlobalMessage.emit_toast("加载失败，正在返回…", "warning")
	is_loading = false
	load_scene = ""
	# 尝试返回上一个场景
	if not scene_history.is_empty():
		go_back()
```

8.1.3.4 Implementación en el proyecto

El cambio de escena se gestiona mediante un SceneManager centralizado, que mantiene un registro de escenas y proporciona interfaces unificadas (change_scene, change_scene_to_file).

Antes de cambiar de escena, se realiza un snapshot del estado mediante CharacterDataManager y se guarda localmente. Posteriormente, se carga una escena intermedia y se ejecuta la carga asíncrona del destino. Una vez completada, se actualiza el estado actual y se emiten eventos de cambio de escena.

Este enfoque unifica la gestión del estado y la transición, reduciendo inconsistencias y mejorando la fiabilidad del sistema.

```
func change_scene(scene_name: String, save_history: bo
	if is_loading:
		push_warning("场景正在加载中，请稍候")
		GlobalMessage.emit_toast("Cargando escena, por
		return

	if not scenes.has(scene_name):
		push_error("场景 '%s' 未在场景字典中注册" % scene_
		GlobalMessage.emit_toast("Esta zona no está di
		return

	var scene_path = scenes[scene_name]
	_change_scene_internal(scene_path, save_history)
```

```
func change_scene_to_file(path: String, save_his
	if is_loading:
		push_warning("场景正在加载中，请稍候")
		GlobalMessage.emit_toast("正在加载场景中，
		return

	_change_scene_internal(path, save_history)
```

```
func _change_scene_internal(path: String, save_history: bool) -> void:
	CharacterDataManager.snapshot_before_scene_change()
	is_loading = true
	load_scene = path
```

```
func snapshot_before_scene_change() -> void:
	# 退出/预删除阶段可能已不在 SceneTree 中；此时调用 get_tr
	if not is_inside_tree():
		return
	_take_snapshot()
	_write_local_checkpoint_from_memory()
```

8.1.4 Menú e interfaz de usuario

Este módulo se centra en la información que el jugador recibe en cada etapa, la forma en que interactúa con el sistema y la coherencia entre la interfaz y la lógica interna. Se abordan la estructura de las interfaces, los mecanismos de interacción y la gestión de estados, garantizando la consistencia con las reglas del sistema.

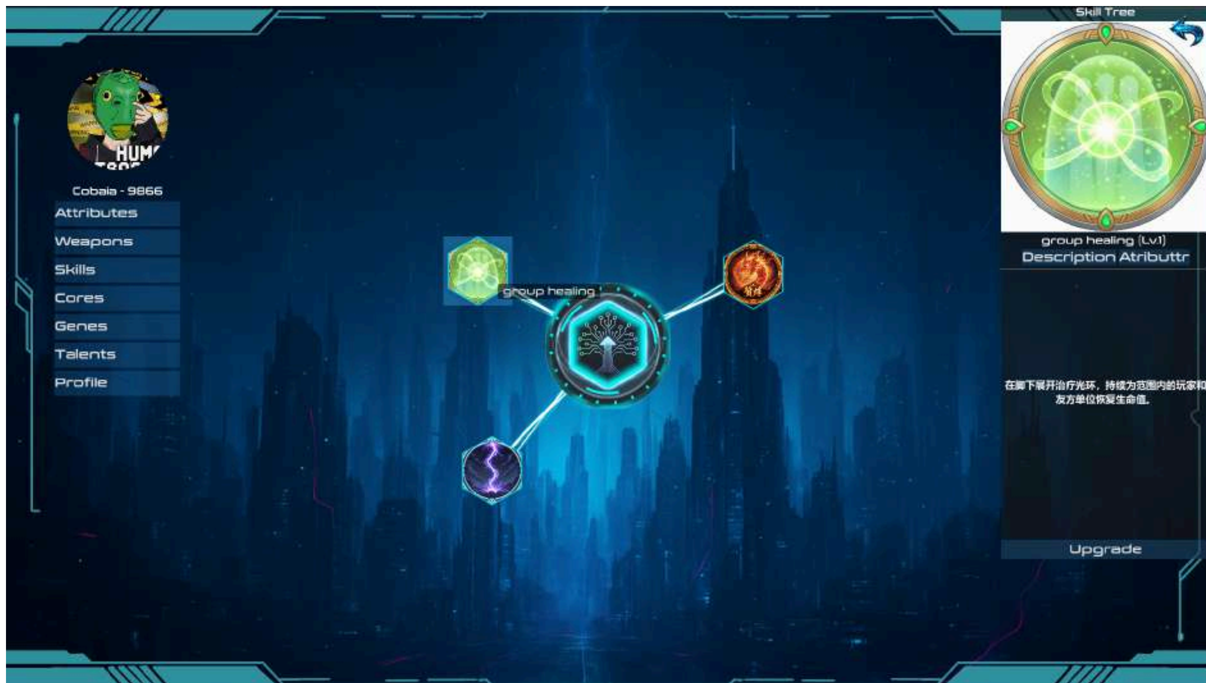
8.1.4.1 Interfaz de información del personaje

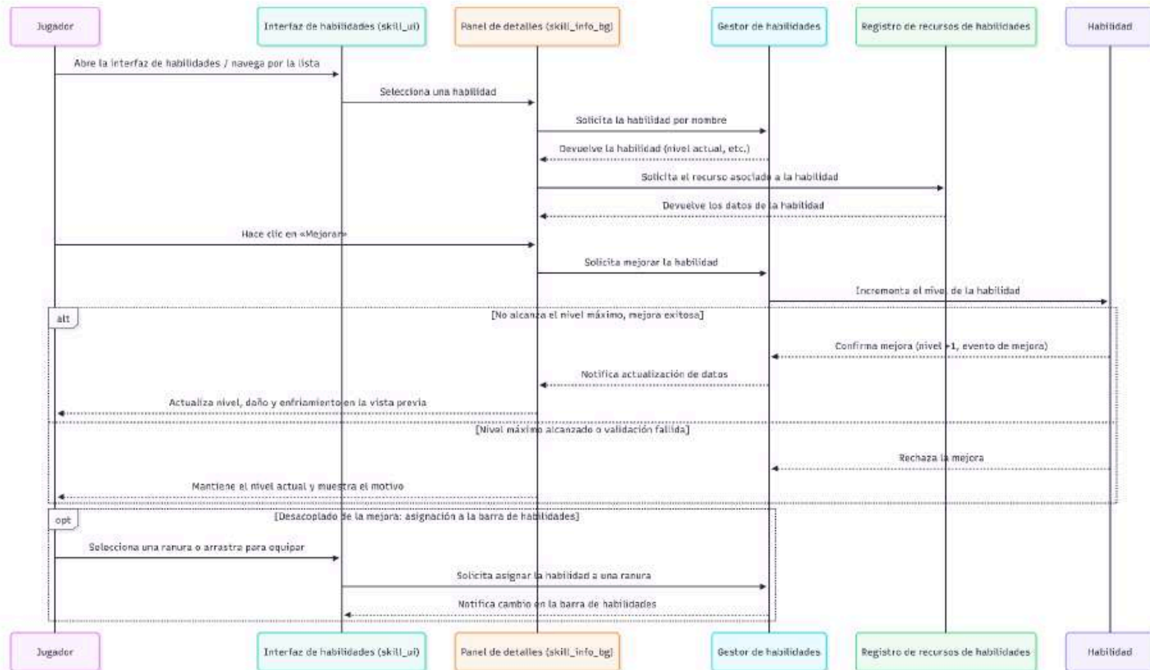
(1) Interfaz de atributos
La interfaz de atributos muestra la composición del poder del personaje, incluyendo atributos base y derivados, así como sus fuentes (equipamiento, habilidades y estados).

También permite acciones como asignación de puntos, reinicio o mejora, incorporando confirmaciones y visualización de costes para evitar errores. La estructura se organiza en áreas de atributos, fuentes de bonificación y acciones, complementadas con tooltips informativos.

(2) Interfaz de habilidades
La interfaz de habilidades presenta información clave como descripción, enfriamiento, coste, nivel y requisitos de desbloqueo. Además, permite equipar habilidades en la barra rápida.

El diseño prioriza la coherencia entre interfaz y sistema, mostrando claramente los cambios tras mejoras o desbloqueos y evitando discrepancias entre la visualización y la lógica de combate.
(Figura 8-5: Diagrama de flujo de interacción de la interfaz de habilidades)

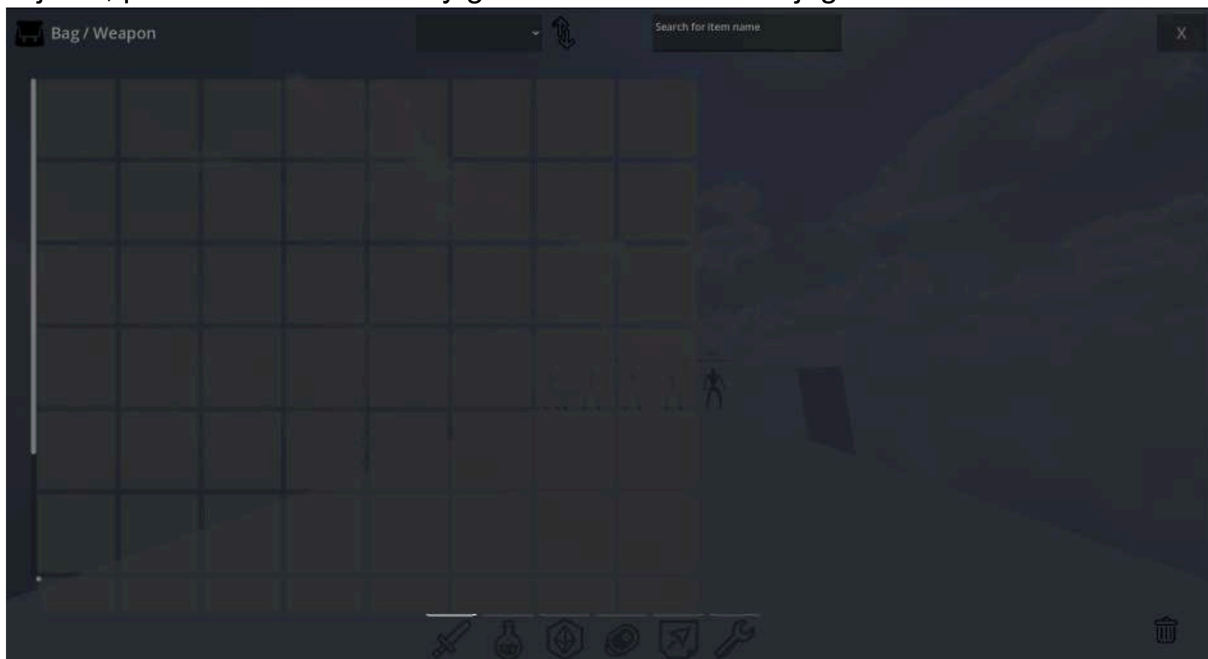




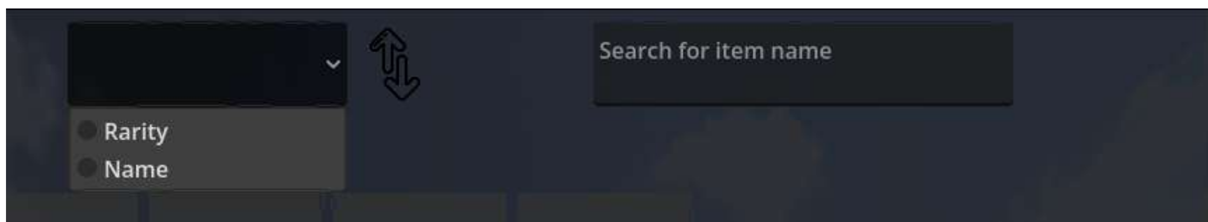
(3) Estado de desarrollo
Otros módulos de interfaz aún se encuentran en desarrollo y serán ampliados progresivamente en futuras iteraciones.

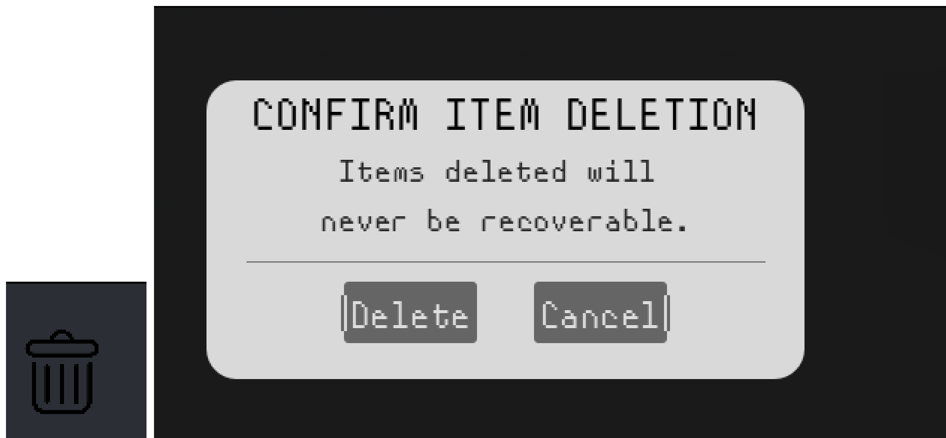
8.1.4.2 Inventario

La interfaz de inventario actúa como el punto de acceso principal al sistema de objetos, permitiendo visualizar y gestionar los ítems del jugador.



Incluye clasificación por tipo, rareza y cantidad, así como operaciones como arrastrar, usar o descartar objetos. Para reducir errores, se implementan mecanismos de confirmación.





A nivel técnico, la interfaz se sincroniza con el modelo de datos mediante binding unidireccional o eventos, evitando modificaciones directas desde la UI y garantizando la consistencia del sistema.

Acción del usuario → Desencadenar evento → Modificar datos → Actualizar interfaz de usuario

```
172 func _on_tool_pressed() -> void:
173     inventory_manager.switch_bag(InventoryManager.BAG_TOOL)
174
175 # 搜索物品
176 func _on_text_edit_text_changed():
177     var search_text = text_edit.text
178     inventory_manager.set_name_filter(search_text)
179
180 # 物品排序功能
181 func _on_classify_item_selected(index: int):
182     match index:
183         0:
184             inventory_manager.set_sort_mode(InventoryManager.SortMode
185         1:
186             inventory_manager.set_sort_mode(InventoryManager.SortMode
187         _:
188             inventory_manager.set_sort_mode(InventoryManager.SortMode
189
190 func _on_sort_pressed():
191     # 切换排序顺序
192     inventory_manager.toggle_sort_order()
```

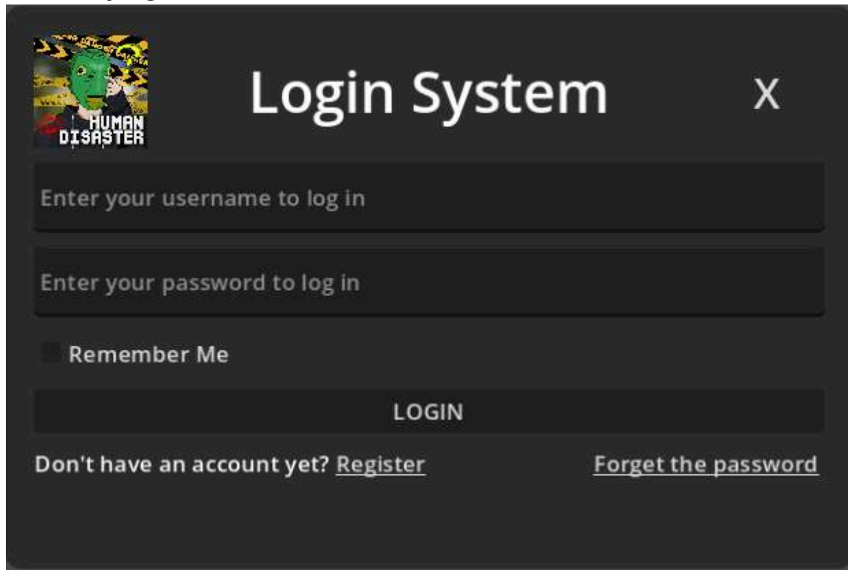
8.1.4.3 Interfaz de inicio de sesión

El sistema utiliza un modelo basado en validación local con datos principales almacenados en el backend. Los nuevos usuarios deben registrarse mediante correo electrónico, introduciendo email, contraseña y un código de verificación enviado al correo, tras lo cual sus datos se almacenan en el sistema.

Los usuarios existentes, tras iniciar sesión, acceden a la selección de personaje y posteriormente al mundo del juego. Antes de permitir el acceso, el sistema consulta

los datos del personaje (como estadísticas) para verificar el progreso, por ejemplo, si el tutorial ha sido completado, aplicando así mecanismos de control de acceso.

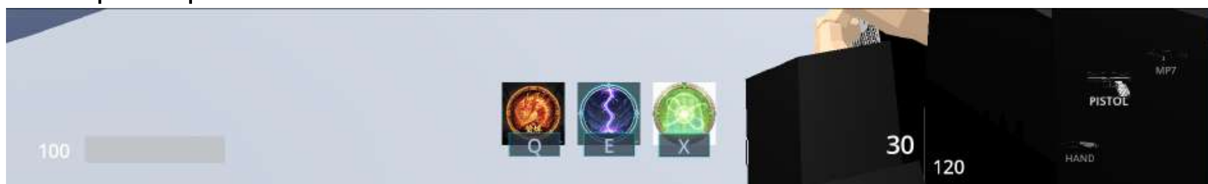
Este flujo garantiza la coherencia entre los datos del usuario y el estado del juego.



8.1.4.4 Interfaz de combate

La interfaz de combate proporciona información en tiempo real esencial para la toma de decisiones, como salud, escudo, munición, estado de habilidades y datos del objetivo.

A nivel visual, incluye elementos como daño flotante, indicadores de crítico y efectos de impacto para mejorar la experiencia del jugador. Debido a la alta frecuencia de actualización, se emplean técnicas de actualización incremental o mecanismos de caché para optimizar el rendimiento.



8.1.4.5 Teclado y ratón virtuales

El teclado y ratón virtuales se definen como una extensión del sistema de entrada, utilizados principalmente para depuración y soporte del tutorial.

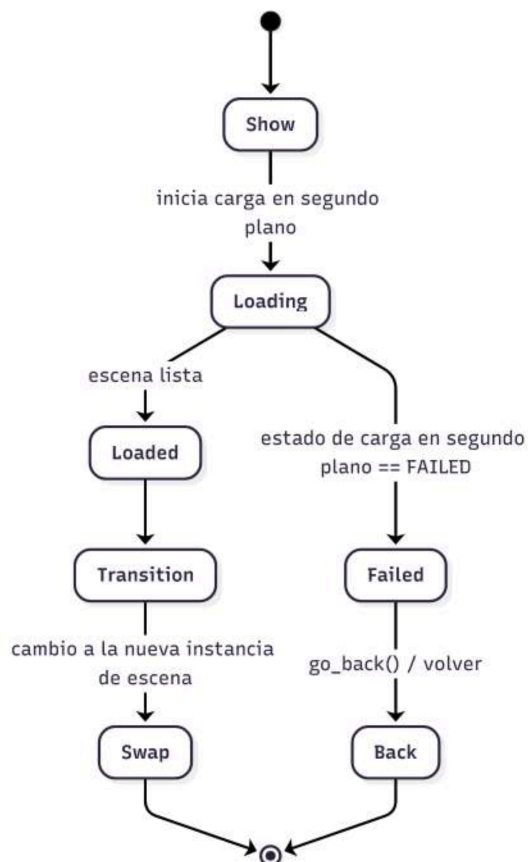
Permiten simular entradas para verificar el correcto funcionamiento del sistema de control y, en el contexto del tutorial, sirven como herramienta visual para indicar al jugador las acciones que debe realizar en cada fase.

8.1.4.6 Pantalla de carga

La interfaz de carga muestra el estado de procesos como carga de recursos, solicitudes de red y restauración de datos. El progreso se obtiene a partir de la carga de recursos, peticiones al servidor y deserialización de datos.

Incluye mecanismos de retroalimentación como barra de progreso, mensajes informativos y opciones de reintento o retorno en caso de error, garantizando la continuidad del sistema.

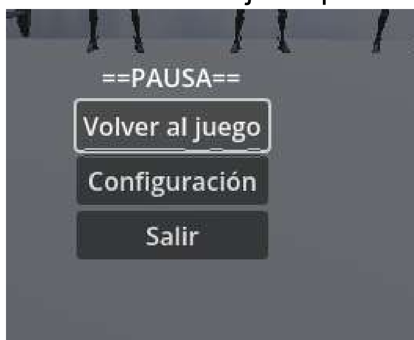
(Figura 8-6: Máquina de estados de la interfaz de carga)



8.1.4.7 Ajustes, pausa y popups

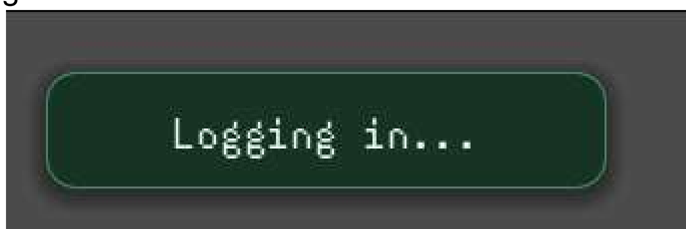
La interfaz de ajustes centraliza la configuración del sistema, incluyendo volumen, calidad gráfica, sensibilidad y controles, con persistencia al salir.

El menú de pausa está vinculado al estado del juego, deteniendo la ejecución mediante escalado temporal, desactivando entradas de combate y manteniendo la coherencia en la jerarquía de interfaces.

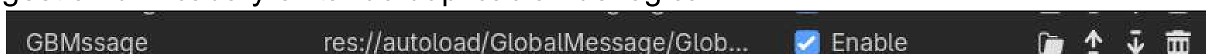


8.1.4.8 Sistema global de mensajes

El sistema de mensajes global permite mostrar información clave mediante elementos como notificaciones (Toast), cuadros de confirmación, recompensas y guías del tutorial.



Se implementa como un componente global accesible por scripts, asegurando una gestión unificada y evitando duplicación de lógica.



8.1.4.9 Gestión y adaptación de UI

La gestión de la UI se basa en garantizar coherencia en la navegación, jerarquía y control de estados. Se adopta una estrategia simplificada de tipo “pila con instancia única”:

- Una pila (ui_stack) gestiona el estado de las interfaces abiertas

- current_ui referencia la interfaz activa
- Métodos como open_ui, toggle_ui y close_current_ui controlan las transiciones

```
signal ui_opened(ui_name: String)
signal ui_closed(ui_name: String)

## UI栈，只保存当前打开的UI
var ui_stack: Array = []
## 当前UI
var current_ui: Control = null
```

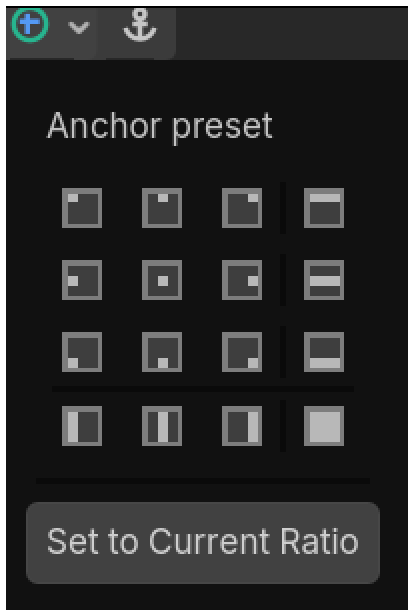
El mapeo entre acciones y UI se gestiona de forma centralizada, y la entrada del usuario se procesa en un único punto.
Además, se implementan restricciones de entrada:

- Se bloquea la interacción de UI en la pantalla de inicio de sesión
- Se limita durante fases críticas del tutorial

```
func close_current_ui():
	if not current_ui:
		push_warning("没有打开的UI")
		return

	var ui_name = ui_stack.pop_back() if ui_stack.size() > 0 else "Unkno
	_close_ui_instance(current_ui)
	current_ui = null
	ui_closed.emit(ui_name)
	print_debug("UI已关闭: ", ui_name, " | 栈深度: ", ui_stack.size())
```

8.1.4.10 Adaptación de resolución

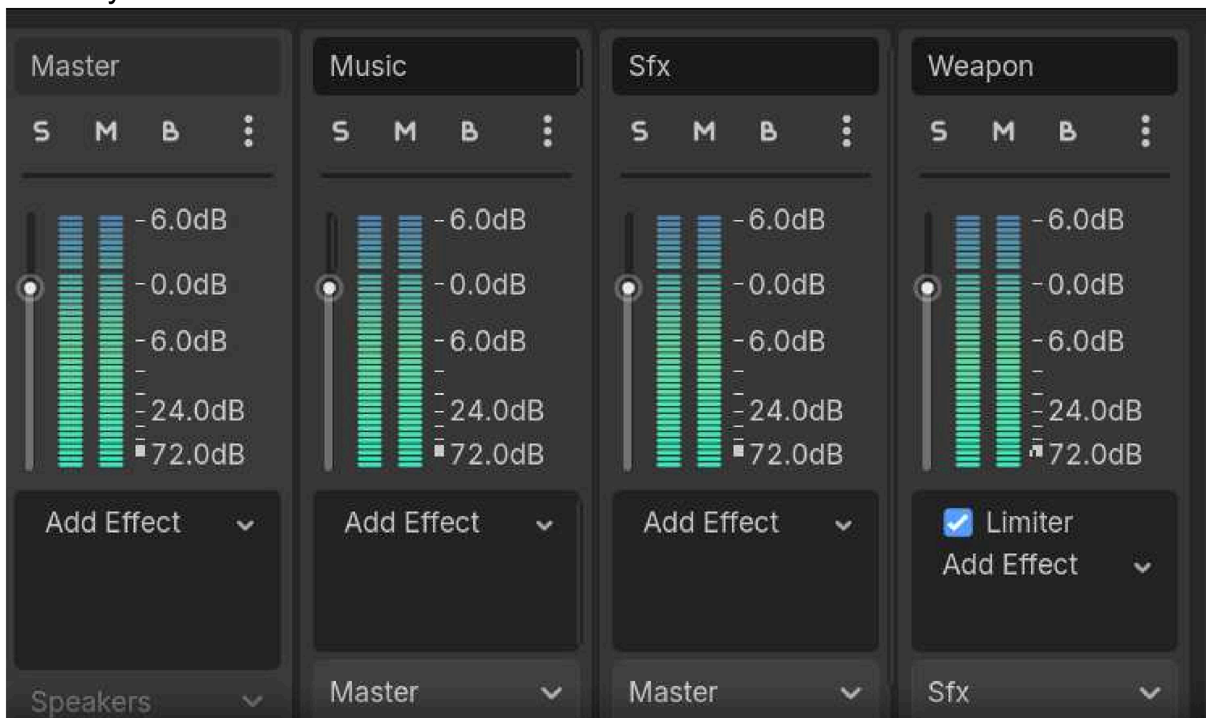
Para garantizar la compatibilidad entre dispositivos, se utilizan estrategias de anclaje y escalado uniforme, asegurando una correcta visualización y evitando la superposición de elementos críticos.



8.1.5 Audio y efectos de sonido

El sistema de sonido se utiliza para mejorar la retroalimentación del juego y la inmersión del jugador, además de proporcionar señales de estado como advertencias de vida baja o disponibilidad de habilidades.

El sistema de audio se divide en cuatro categorías principales: efectos de personaje, habilidades, armas y música de fondo, organizados en grupos para facilitar su control y mezcla.



8.1.5.1 Sonidos del personaje

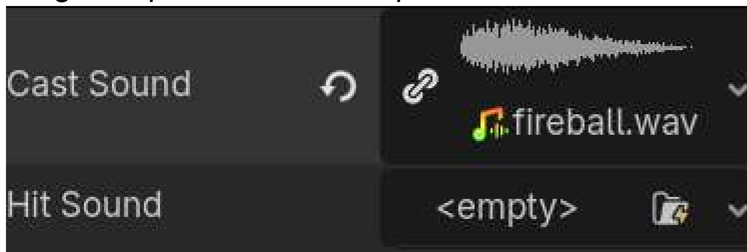
Incluyen sonidos de pasos, impactos, muerte e interacciones. Estos efectos están vinculados a eventos de animación o cambios de estado para evitar desincronización entre audio y visuales.

A nivel de implementación, los sonidos se gestionan desde un nodo central del personaje que controla la carga y reproducción de los recursos de audio.

8.1.5.2 Sonidos de habilidades

Incluyen sonidos de lanzamiento, impacto, explosiones y efectos persistentes. Se vinculan al ciclo de vida de la habilidad (inicio, impacto y finalización).

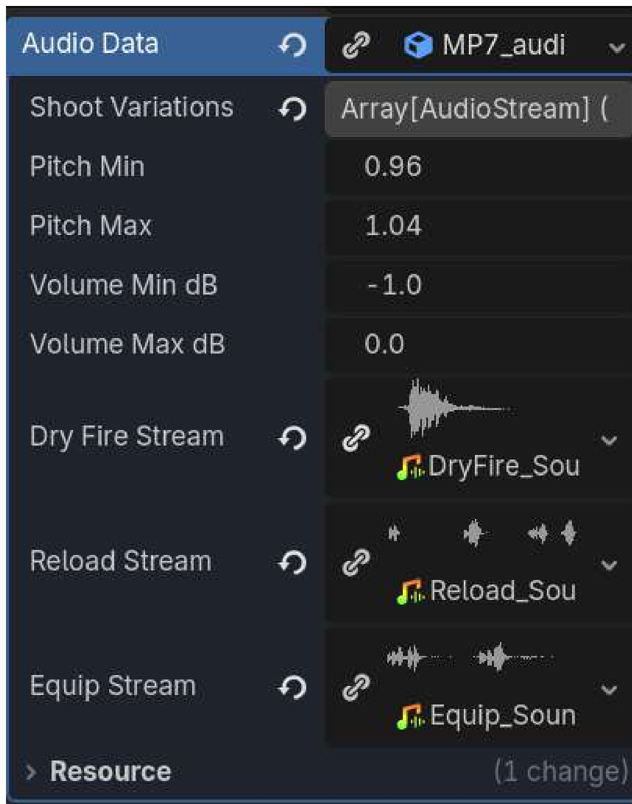
Cada tipo de habilidad tiene prioridades y reglas de mezcla específicas para asegurar que los efectos importantes no sean enmascarados por otros sonidos.



8.1.5.3 Sonidos de armas

Incluyen disparos, recarga, disparo en vacío e impactos.

Se utiliza un enfoque basado en datos para asignar recursos de audio y momentos de activación según el tipo de arma, lo que permite un sistema flexible y extensible.



8.1.5.4 Música de fondo y mezcla

La música de fondo cambia según la escena o el estado del juego, utilizando transiciones suaves mediante fundidos de entrada y salida.

El sistema de mezcla controla el balance general de audio, regulando la relación entre música y efectos para mantener claridad y jerarquía sonora.

8.1.6 Sistema de habilidades

El sistema de habilidades se basa en dos principios fundamentales: arquitectura basada en datos y consistencia en la resolución de impactos. Todas las habilidades se definen mediante recursos y tablas de configuración, incluyendo multiplicadores de daño, área de efecto, tiempo de recarga, coste de recursos y tipo de atributo, logrando un desacoplamiento total entre lógica y código.

En futuras versiones, el flujo de ejecución se formalizará con fases como pre-animación, post-animación e interrupción, además de distintos tipos de selección de objetivo (direccional, área o bloqueo).

```
class_name SkillResource
extends Resource

## 与 skills.json 的关联 ID（0 = 未关联）
@export var skill_id: int = 0

## 基础属性
@export var skill_name: String = "UNAMED"
@export_multiline var description: String = ""
@export var icon: Texture2D
@export var max_level: int = 10

## 基础数值（等级1时的值）
@export_group("基础属性")
@export var base_damage: float = 100.0
@export var base_attack_power: float = 50.0
@export var base_cooldown: float = 5.0
@export var base_range: float = 10.0
# 持续时间（DOT技能用）
@export var base_duration: float = 0.0
```

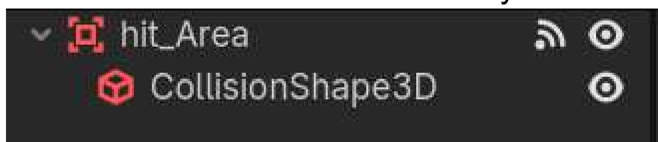
8.1.6.1 Sistema de impacto

La detección de impacto se implementa mediante hitboxes, raycasts o proyectiles.

El sistema sigue un principio clave:

Los efectos visuales representan, pero la lógica determina el resultado

Es decir, el impacto real se calcula mediante colisiones o lógica del sistema, mientras que los efectos visuales solo responden a eventos, evitando inconsistencias entre visualización y mecánicas.



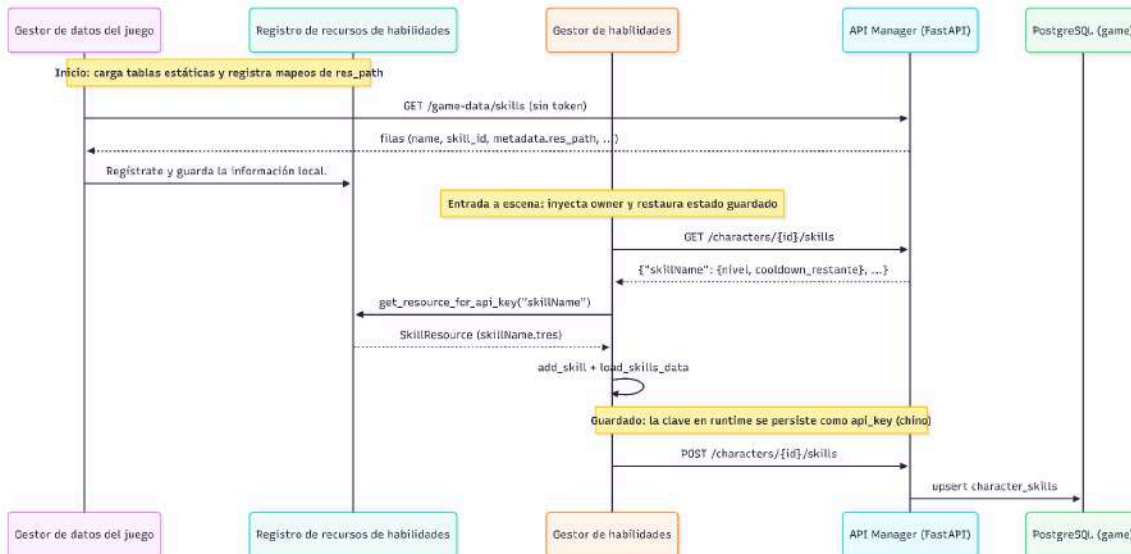
```
func _on_hit_area_body_entered(body: Node3D) -> void:
	if not body.is_in_group("enemy"):
		return
	_hit_target(body)


func _on_hit_area_area_entered(area: Area3D) -> void:
	if not area.is_in_group("enemy"):
		return
	_hit_target(area)
```

8.1.6.2 Arquitectura de datos y ejecución

El sistema sigue un flujo estandarizado:

datos estáticos → registro de recursos → instancia en runtime → persistencia/sincronización

GameDataManager carga las configuraciones estáticas
SkillResourceRegistry gestiona el mapeo de recursos
SkillManager controla las instancias en tiempo de ejecución (cooldown, nivel, barra de habilidades)
(Figura 8-7: Diagrama de secuencia de lanzamiento de habilidades)



8.1.6.3 Estructura de datos y mapeo

Cada habilidad se compone de tres niveles:

Definición estática (valores y efectos)

Ruta de recurso (.tres)
Clave de persistencia (API / backend)

Para resolver inconsistencias entre nombres locales y claves del servidor, se introduce SkillResourceRegistry como capa de mapeo unificada:

Clave runtime (SkillResource.skill_name)
Clave API (game.skills.name)
ID de recurso (skills.json.skill_id)

Este sistema permite conversión bidireccional entre guardado y carga.

```
func register_mapping(api_display_name: String, skill_res: SkillResource
	if skill_res == null:
		push_warning("[SkillResourceRegistry] register_mapping: 收到空资源
		return
	var runtime: String = skill_res.skill_name.strip_edges()
	if runtime.is_empty():
		push_warning("[SkillResourceRegistry] register_mapping: 资源 skil
		return
	SkillLookup.register_synonym(runtime, runtime)
	if not api_display_name.is_empty():
		SkillLookup.register_synonym(api_display_name, runtime)
	if skill_res.skill_id > 0:
		SkillLookup.register_synonym(str(skill_res.skill_id), runtime)
	_resources_by_runtime_name[runtime] = skill_res
	if not api_display_name.is_empty():
		_api_key_to_runtime[api_display_name] = runtime
		_runtime_to_api_key[runtime] = api_display_name
	elif not _runtime_to_api_key.has(runtime):
		## 无 API 名时退化为自映射，保证 save_skills_data 至少能回写 runtime k
		_runtime_to_api_key[runtime] = runtime
```

8.1.6.4 Arquitectura del sistema

SkillManager gestiona las habilidades en runtime, incluyendo registro, barra de habilidades y actualización de estado. Todas las habilidades se almacenan mediante claves normalizadas para garantizar consistencia.

```
func add_skill(skill_resource: SkillResource, initial_level: int = 1) ->
	if skill_resource == null:
		push_warning("[SkillManager] add_skill 收到 null 资源")
		return null

	## 自动回注册：本地测试技能没经过 /game-data/skills 也能走存档路径
	if not SkillResourceRegistry.is_registered(skill_resource):
		SkillResourceRegistry.register_mapping(skill_resource.skill_name

	var key: String = _skill_dict_key(skill_resource.skill_name)
	if skills.has(key):
		var existing: Skill = skills[key]
		## 场景切换或重生后 character 会指向新节点，这里必须刷新，避免 Skill 持有
		if character != null:
			existing.owner_node = character
		return existing
```

La barra de habilidades utiliza un sistema de slots fijo para asegurar estabilidad en la entrada del usuario.

Gracias a SkillResourceRegistry, el sistema permite:

Sistema de habilidades basado en datos
Soporte para desarrollo local y pruebas
Consistencia entre cliente y backend
Evitar fallos por discrepancias de nombres en la recuperación de datos

Clave API (game.skills.name)	**skill_id (skills.json)**	**Ruta de recurso (.tres)**	**Clave runtime (SkillResource.skill_name)**	**Observaciones**
Fireball	3002002	res://resource/skill/Fireball.tres	FireBall	Habilidad de proyectil; admite "3002002" como alias
Lightning	3003008	res://resource/skill/Lightning.tres	Lightning	Habilidad AOE con daño continuo
Group Healing	3005009	res://resource/skill/GroupHealingSkill.tres	group healing	Habilidad de soporte/curación

8.1.7 Sistema de inventario

El sistema de inventario gestiona la obtención, almacenamiento y uso de objetos, actuando como un puente entre el sistema de combate, el sistema de recompensas y la progresión del jugador. Cada objeto se define mediante atributos como tipo, rareza, límite de apilamiento y acciones permitidas (usar, equipar, inspeccionar o mover).

Los objetos pueden obtenerse a través de enemigos, recompensas, cofres o tiendas. El sistema debe definir reglas claras como capacidad máxima, lógica de apilamiento, descarte, descomposición y mecanismos de ordenación y filtrado.

Implementación en el proyecto

El inventario está gestionado por un módulo central que utiliza un sistema de capacidad fija (por defecto 60 espacios). La entrada de objetos se realiza mediante un sistema de drops independiente.

A nivel de datos, el inventario forma parte del snapshot del personaje y se sincroniza con el backend mediante API, garantizando consistencia entre cliente y servidor. Se distingue claramente entre la estructura local y los DTOs utilizados en la comunicación.

En la capa de interfaz, el inventario utiliza un diseño basado en componentes y sincronización mediante eventos o binding, evitando modificaciones directas sobre los datos.

8.1.8 Sistema de armas

El sistema de armas define las diferencias funcionales entre armas y su integración con el sistema de combate, sonido y UI. Los atributos principales incluyen cadencia de disparo, retroceso, dispersión, capacidad del cargador y recarga.
El sistema debe contemplar:

- Flujo de ataque (entrada → detección → cálculo de daño → feedback)
- Cambio de arma
- Gestión de munición y recarga
- Sincronización con UI y efectos de sonido

En caso de incluir mejoras o progresión, es necesario definir reglas de escalado, estructura de datos y puntos de interacción en la interfaz.

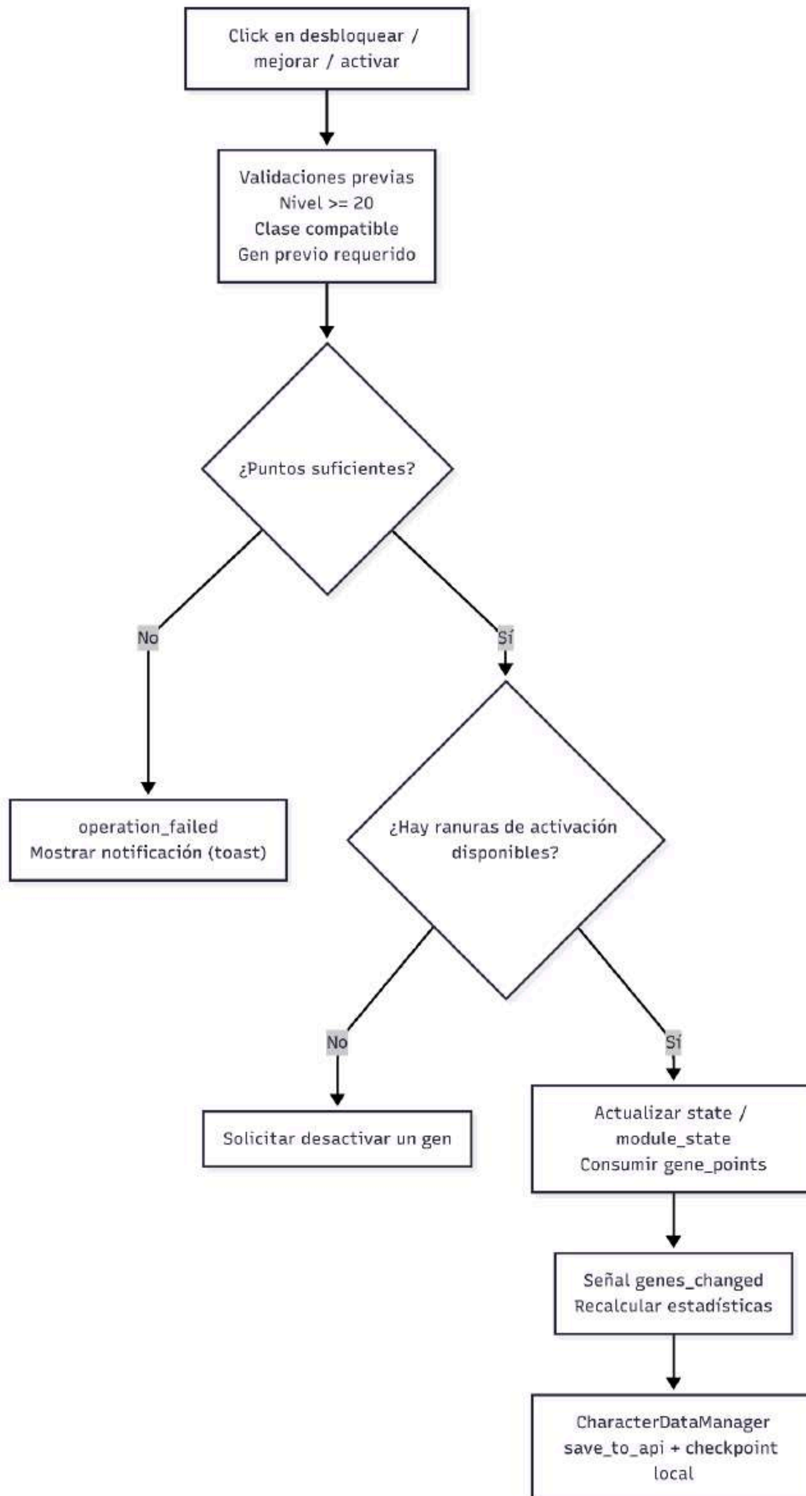
Implementación en el proyecto

La lógica de armas está centralizada en un gestor que controla el disparo, cambio de armas y actualización de estados. Las armas, proyectiles y objetos recogibles se gestionan como recursos instanciables.
Para garantizar la consistencia, el estado de armas y objetos recogidos se integra en el sistema de persistencia, permitiendo restaurar correctamente el estado del jugador tras cambios de escena o carga de partida.

8.1.9 Sistema genético

El sistema genético es uno de los mecanismos centrales del proyecto, diseñado para ampliar las posibilidades de progresión del personaje y diversificar las estrategias de combate. Los jugadores pueden obtener secuencias genéticas mediante inyección de genes o procesos de despertar, que al activarse modifican atributos, habilidades y efectos especiales (como probabilidad de drop o inmunidades).

(Figura 8-8: Diagrama de flujo de desbloqueo, mejora y activación de genes)



8.1.9.1 Efectos y relación con otros sistemas

Los genes afectan múltiples subsistemas:

Sistema de atributos
Sistema de combate (crítico, daño, etc.)
Reglas especiales (inmunidad, regeneración, etc.)

El GeneManager centraliza todos los efectos activos y los aplica durante el recálculo de estadísticas, garantizando coherencia y trazabilidad.

8.1.9.2 Mecanismos de evolución

El sistema incorpora tres mecanismos principales:

Selección: activación de genes según condiciones específicas
Crossover: combinación de genes para generar nuevos efectos
Mutación: variaciones aleatorias controladas por probabilidad

Estos mecanismos introducen variabilidad sin perder control estructural.

8.1.9.3 Flujo de desbloqueo y activación

El ciclo de vida de los genes incluye:

Verificación de requisitos (nivel, clase, prerrequisitos)
Consumo de recursos (gene points)
Limitación por slots activos
Aplicación de efectos y recálculo de estadísticas

El sistema utiliza eventos (como genes_changed) para garantizar actualización coherente.

8.1.9.4 Implementación y consistencia

El sistema está gestionado por un módulo central que define:

Nivel de desbloqueo del sistema
Límite de slots activos (incluyendo límite global)
Costes de desbloqueo y mejora según rareza

El cliente y el backend comparten las mismas reglas, y todas las operaciones críticas son validadas en el servidor, garantizando:

consistencia entre cliente y servidor

Esto evita errores derivados de concurrencia o manipulación de datos.

8.1.9.5 Balance y control del sistema

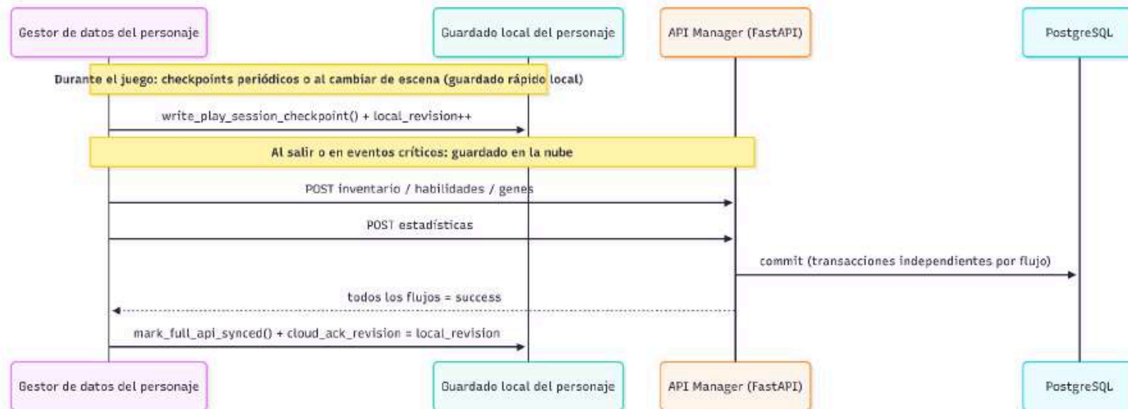
Para evitar que el sistema se convierta en una fuente de aleatoriedad incontrolable, se aplican varias estrategias:

Limitación de efectos mediante caps y slots
Escalado por rareza, vinculando poder con coste
Control de aleatoriedad mediante modelos probabilísticos acotados
Mecanismos de control del jugador, permitiendo decisiones estratégicas

Este enfoque equilibra la variabilidad con la previsibilidad, asegurando una experiencia jugable estable y ajustable.

8.1.10 Sistema de guardado

El sistema de almacenamiento tiene como objetivo garantizar la recuperabilidad de los datos y la consistencia entre dispositivos. Para ello, se adopta una estrategia híbrida basada en almacenamiento local y almacenamiento en la nube.
(Figura 8-9 Diagrama de secuencia archivado)



8.1.10.1 Almacenamiento local

El almacenamiento local permite guardar el estado del juego de forma rápida y frecuente, con las siguientes características:

- Lectura y escritura de baja latencia
- Mecanismos básicos de validación y cifrado
- Uso de versiones para seguimiento del estado

El sistema realiza guardados periódicos o en eventos clave, como cambios de escena, asegurando la recuperación ante fallos.

8.1.10.2 Sincronización con la nube

El almacenamiento en la nube se utiliza para la persistencia a largo plazo y sincronización entre dispositivos.
El sistema define estrategias claras de sincronización:

- Sobrescritura, fusión o resolución de conflictos
- Uso de versiones (por ejemplo, local_revision)
- Sincronización en eventos críticos o al salir del juego

Los datos del personaje (atributos, inventario, habilidades, genes y estado de escena) se envían al servidor mediante API.

8.1.10.3 Control de versiones y consistencia

Se utilizan dos indicadores principales:

- local_revision: versión local
- cloud_ack_revision: versión confirmada por el servidor

Cuando la versión local supera a la de la nube, se marca como pendiente de sincronización. Tras una sincronización exitosa, ambas versiones se alinean. Durante el inicio de sesión, los datos de la nube se consideran la fuente de verdad principal.

8.1.10.4 Compatibilidad y evolución

El sistema incluye mecanismos para soportar cambios en la estructura de datos:

- Valores por defecto para nuevos campos
- Migración de campos antiguos
- Tolerancia en la deserialización

Esto garantiza la compatibilidad con versiones anteriores.

8.1.10.5 Implementación y flujo del sistema

El sistema se organiza en varios módulos:

- Gestión de datos en memoria
- Almacenamiento local
- Comunicación con el backend

Se soportan los siguientes flujos:

- Guardado periódico local
- Guardado en cambio de escena
- Guardado forzado al salir
- Manejo de errores y reintentos en la nube

8.1.10.6 Separación de responsabilidades de datos

El sistema distingue claramente entre:

- Datos en la nube: atributos, inventario, habilidades, genes y estado del mundo
- Datos locales: configuraciones del cliente y datos temporales

Esta separación garantiza consistencia global y flexibilidad local.

Módulo	Campos (ejemplo)	API / DTO en la nube	Snapshot en cliente	Almacenamiento local (.lcs)	Descripción
Atributos (stats)	max_health, current_health, attack, defense, critical_rate, gene_point	POST/GET /characters/{id}/stats	_stats_snapshot	stats_snapshot	Fuente principal de datos; se sincroniza periódicamente

	s, experience				
Tutorial	tutorial_completed	Incluido en stats	_stats_snapshot["tutorial_completed"]	Incluido en stats	Permite consistencia entre dispositivos
Inventario	slots: [{id, qty} | null]	POST/GET /characters/{id}/inventory	_inventory_snapshot	inventory_snapshot	Soporta apilamiento y slots vacíos
Habilidades	skills: { api_key: {level, cooldown_remaining} }	POST/GET /characters/{id}/skills	_skills_snapshot	skills_snapshot	api_key identifica habilidades mediante mapeo
Genes	genes[], gene_modules[]	POST/GET /characters/{id}/genes	_genes_snapshot	genes_snapshot	Validación crítica en servidor
Estado de escena	scene_path, position, rotation_y, collected_pickables	GET/POST /characters/{id}/scene_state	_scene_state_snapshot	scene_state_snapshot	Permite restaurar estado del mundo
Extensión cliente	client_blob (preferencias, progreso UI)	Opcional	Opcional	client_blob	Datos locales no dependientes del servidor

8.1.11 Sistema de usuario

El sistema de usuario gestiona la identidad, los personajes y la configuración básica, definiendo claramente **el modelo de identidad, los límites de datos y el ciclo de vida de la sesión.**
En este proyecto, el sistema se basa en autenticación backend mediante JWT, siguiendo el flujo:

Login → emisión de token → validación en peticiones → selección de personaje → carga de datos

El ciclo de vida de la sesión incluye:
- Inicio de sesión: obtención del token de acceso
- Uso: autenticación en cada petición
- Expiración: renovación o nuevo login

En cuanto al manejo de errores:

- Fallo de autenticación → respuesta controlada
- Error en carga de datos → reintento o uso de datos locales

Además, el sistema está integrado con el almacenamiento en la nube:

- Carga automática de datos tras login
- Sincronización en eventos clave
- Consistencia entre dispositivos

Extensibilidad:

- Integración futura con OAuth2
- Soporte para múltiples personajes
- Identidad compartida con sistema de comunidad

8.1.12 Combate y daño

El sistema de combate constituye el núcleo del juego y se implementa como un **pipeline determinista de procesamiento de datos.**

8.1.12.1 Flujo de resolución

El orden estándar es:

Impacto → Cálculo de daño → Aplicación → Estados → Muerte → Recompensas

Este orden se mantiene estrictamente para evitar inconsistencias lógicas.

8.1.12.2 Modelo de daño

El daño depende de distintos factores como los atributos del personaje, la habilidad utilizada, el equipamiento y las características del enemigo. Este modelo permite un cálculo comprensible y facilita el ajuste de valores durante el desarrollo.

8.1.12.3 Arquitectura basada en eventos

El sistema de combate se apoya en una arquitectura basada en eventos para desacoplar la lógica interna de la capa de presentación. Cada interacción relevante genera eventos estandarizados, tales como:

- OnHit (impacto registrado)
- OnDamage (daño aplicado)
- OnDeath (muerte del objetivo)

Este enfoque permite que el núcleo del sistema se centre exclusivamente en el cálculo y resolución del combate, mientras que otros subsistemas reaccionan a dichos eventos de manera independiente. Entre sus principales ventajas destacan el desacoplamiento entre lógica y representación, la facilidad para registrar información de depuración y la escalabilidad del sistema ante nuevas funcionalidades sin modificar la lógica base.

8.1.12.4 Feedback de combate

La retroalimentación del combate se diseña para reforzar la claridad del estado del juego y mejorar la percepción del impacto de las acciones del jugador. Se estructura en cuatro capas complementarias:

- **Feedback físico**: incluye retroceso, empujes e interrupciones de animación
- **Feedback de interfaz (UI)**: visualización de números de daño, críticos y estados
- **Feedback visual**: efectos de partículas, flashes y cambios de estado en el entorno
- **Feedback sonoro**: efectos de impacto, golpes, habilidades y eventos críticos

La arquitectura mantiene una separación estricta entre lógica y presentación: la lógica del combate únicamente genera eventos, mientras que la capa de presentación consume dichos eventos para producir la retroalimentación correspondiente.

8.1.12.5 Casos límite

El sistema contempla varios escenarios críticos para garantizar estabilidad y consistencia en situaciones de alta concurrencia o eventos simultáneos:

- **Impactos múltiples simultáneos**: se gestionan mediante colas de eventos para asegurar un procesamiento ordenado y determinista.
- **Muerte en el mismo frame**: se evita la duplicación de estados mediante validaciones de consistencia antes de aplicar el estado final.
- **Invulnerabilidad temporal**: los impactos entrantes se filtran y se ignoran mientras el estado de invulnerabilidad esté activo.

Estos mecanismos aseguran que el sistema mantenga un comportamiento estable incluso en condiciones extremas de ejecución.

8.1.13 Clases y estados

El sistema de clases define el estilo de juego de cada personaje, mientras que el sistema de estados regula los efectos dinámicos que afectan temporalmente su comportamiento en combate. Ambos sistemas están diseñados para complementarse y mantener coherencia en la progresión y la jugabilidad.

8.1.13.1 Sistema de clases

El sistema de clases determina la identidad funcional del personaje dentro del juego. Cada clase define:

- Crecimiento de atributos principales
- Conjunto de habilidades disponibles
- Restricciones de armas o equipamiento
- Habilidades pasivas específicas

El objetivo principal es ofrecer variedad de estilos de juego sin comprometer el equilibrio global del sistema.

8.1.13.2 Sistema de estados

El sistema de estados gestiona efectos temporales aplicados a los personajes o entidades. Su ciclo de vida está claramente definido como:

Add → Refresh → Stack → Remove → Expire

Este flujo garantiza un comportamiento consistente para todos los efectos, independientemente de su origen.

8.1.13.3 Reglas clave del sistema de estados

El sistema sigue un conjunto de reglas estructuradas para asegurar coherencia:

- Apilamiento: ciertos estados pueden acumularse según su tipo
- Reemplazo: algunos estados sustituyen versiones anteriores
- Acumulación controlada: límites definidos para evitar abusos
- Reinicio de duración: refresco de duración en condiciones específicas

Además, el sistema define una jerarquía de prioridad, donde los estados de control tienen precedencia sobre otros efectos, y un sistema de inmunidades basado en etiquetas que evita interacciones no deseadas.

8.1.13.4 Diseño técnico

La implementación se basa en una arquitectura centralizada y orientada a datos. Los estados no dependen de lógica dispersa en múltiples sistemas, sino que se gestionan desde un controlador unificado. Esto facilita la mantenibilidad, la extensibilidad y la depuración del sistema.

8.1.14 Sistema de atributos

El sistema de atributos constituye la base numérica del juego y define todas las interacciones de progresión, combate y balance. Su diseño se basa en un modelo composable de estadísticas que permite combinar múltiples fuentes de modificación de forma controlada.

8.1.14.1 Estructura del sistema

Los atributos se dividen en dos niveles:

- Atributos base: vida, ataque, defensa
- Atributos derivados: probabilidad de crítico, velocidad, resistencias, entre otros

Esta separación permite mantener claridad entre valores fundamentales y efectos calculados.

8.1.14.12 Flujo de cálculo

El proceso de cálculo sigue una estructura secuencial:

Base → modificadores → cálculo derivado → valor final

Este modelo garantiza una dependencia unidireccional, lo que facilita la depuración y permite la implementación de mecanismos de cacheo para optimizar el rendimiento.

(3) Reglas de acumulación

Los modificadores de atributos pueden aplicarse mediante dos tipos principales de operaciones:

- Aditiva: suma directa de valores
- Multiplicativa: escalado porcentual del resultado

Por ejemplo:

Ataque final = (base + bonus) × multiplicador

Este esquema permite un control preciso del crecimiento de poder del personaje.

(4) Sistema de crecimiento

El crecimiento de atributos se produce a través de múltiples fuentes:

- Subida de nivel
- Equipamiento
- Estados activos
- Sistema genético

Cada fuente se integra dentro del mismo sistema de cálculo para mantener coherencia global.

(5) Balance del sistema

El equilibrio numérico se controla mediante diferentes mecanismos:

- Límites máximos (caps)
- Soft caps para crecimiento progresivo
- Configuración basada en datos

Este enfoque permite realizar ajustes de balance sin necesidad de recompilar el sistema, facilitando iteraciones rápidas durante el desarrollo.

(6) Integración con el sistema de combate

Todos los atributos convergen finalmente en el sistema de combate, donde se utilizan para el cálculo de daño, defensa y efectos secundarios. De esta forma, se establece un ciclo completo que conecta progresión, estadísticas y resolución de combate de manera coherente y consistente.

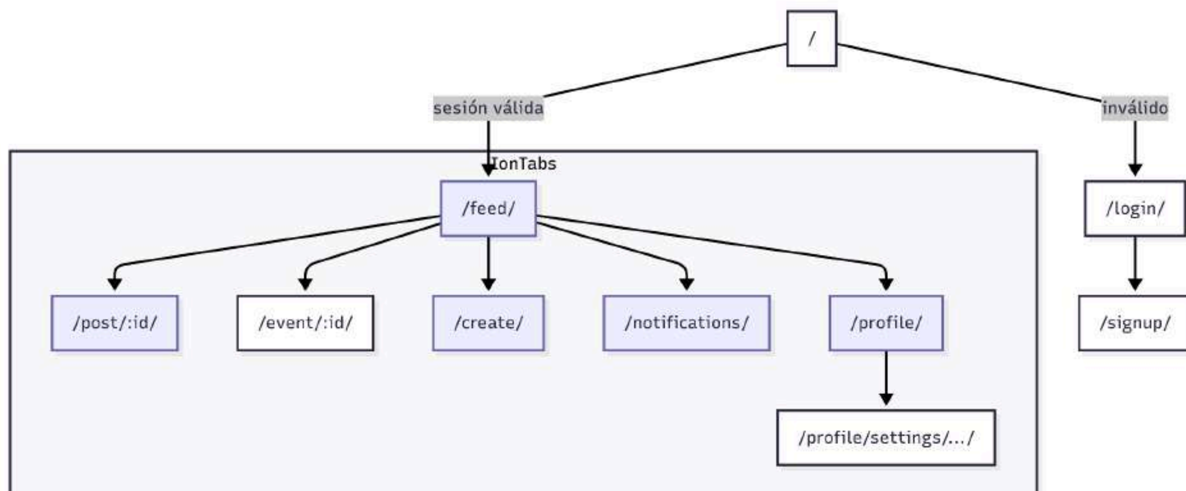
8.2 Herramientas de depuración y pruebas

8.2.1 Requisitos y arquitectura de la información

Las principales tareas del usuario en la aplicación comunitaria incluyen: registro e inicio de sesión, navegación del flujo de contenido (Feed), publicación de posts (incluyendo imágenes), interacción social (comentarios, “me gusta” y seguimiento), recepción de notificaciones y gestión del perfil personal.

El diseño de la arquitectura de la información se centra en dos aspectos clave: en primer lugar, mantener una estructura de navegación coherente entre dispositivos móviles y entorno web; en segundo lugar, permitir una transición fluida entre usuarios no autenticados y autenticados, evitando acciones indebidas en estado anónimo, como la publicación o interacción sobre contenido. Para ello, el sistema incorpora mecanismos de control tanto a nivel de rutas como de interfaz, garantizando una experiencia de usuario consistente y segura.

（Figura 8-10: Arquitectura de la aplicación y árbol de rutas）



8.2.2 Estructura del frontend y organización de rutas

El frontend se organiza en una arquitectura de tres capas: Page (páginas), Component (componentes) y Service (servicios). La capa de páginas gestiona la composición de vistas y la estructura general; la capa de componentes se encarga de los elementos reutilizables y la lógica de interacción; y la capa de servicios centraliza las peticiones API, el almacenamiento en caché y la gestión del estado. Esta organización reduce el acoplamiento y mejora la mantenibilidad y escalabilidad del sistema.

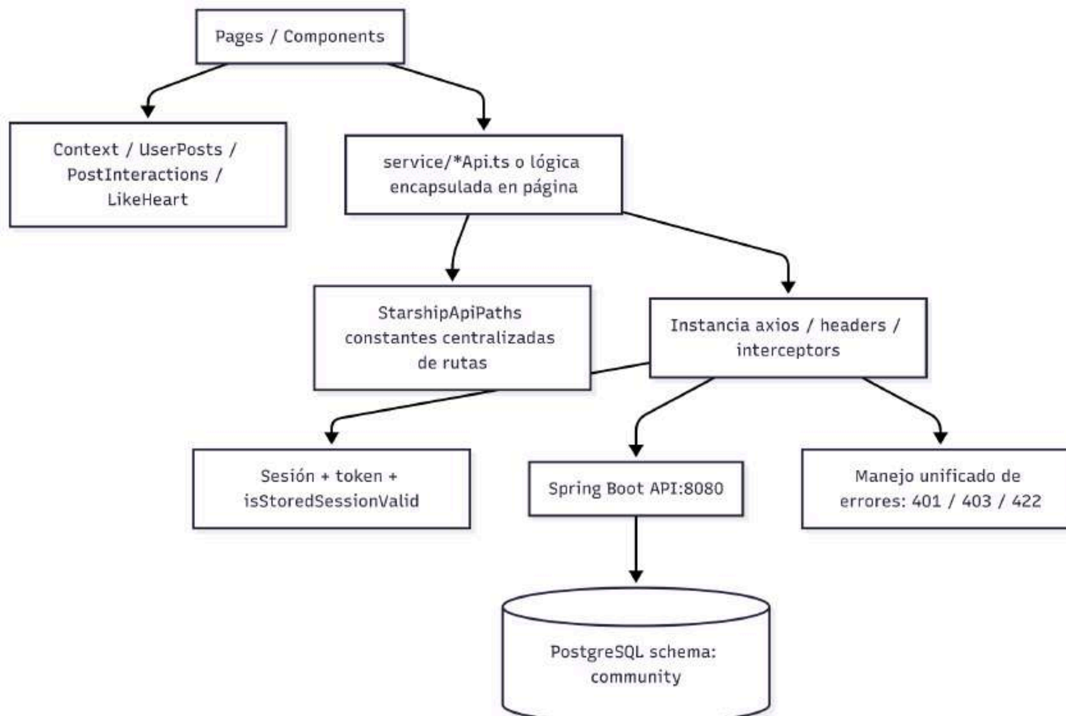
La estructura de rutas se define por dominios funcionales como Feed, Post, Profile, Auth y Settings. Las rutas que requieren autenticación se protegen mediante guardias de navegación o una capa centralizada de autorización, y utilizan parámetros dinámicos (como postId o userId) para la carga de contenido. En el contexto de Ionic, es necesario gestionar adecuadamente la interacción entre el navigation stack y la navegación por pestañas (tabs), evitando inconsistencias derivadas de rutas anidadas.

En la implementación, el punto de entrada del frontend se encuentra en StarShipDH/src/App.tsx. El sistema utiliza IonTabs junto con IonRouterOutlet para construir la navegación inferior basada en pestañas, y emplea ProtectedRoute para proteger rutas como /feed, /post/:id, /create, /notifications, /profile y sus subrutas.

Como restricción de diseño, las páginas de inicio de sesión y registro se sitúan fuera de la estructura IonTabs, evitando que el sistema de navegación de Ionic mezcle rutas como /signup dentro del historial de una pestaña, lo que podría provocar errores de navegación tras el registro. Además, la ruta raíz / utiliza isStoredSessionValid() para verificar la validez de la sesión y redirigir automáticamente al usuario a /feed o /login, centralizando así la lógica de autenticación en un único punto y garantizando un flujo de navegación coherente.

8.2.3 Encapsulación de la capa de red y estrategia de alineación de API

Las aplicaciones comunitarias requieren invocar con frecuencia los servicios del backend, por lo que es necesario centralizar la gestión de rutas API, cabeceras de solicitud, manejo de errores y estrategias de reintento, evitando así problemas de mantenimiento y falta de consistencia derivados de llamadas dispersas.
(Figura 8-11: Diagrama de la estructura de encapsulación de la capa de red frontal)



En este proyecto, la capa de red se diseña siguiendo tres principios fundamentales:

(1) Centralización de rutas API
Se definen constantes de rutas (por ejemplo, StarshipApiPaths) en un único punto, lo que reduce el riesgo de desalineación entre frontend y backend y facilita la evolución del sistema.

(2) Encapsulación unificada de solicitudes
Se implementa un cliente HTTP basado en axios que gestiona automáticamente la inclusión del token JWT, la interceptación de errores (401/403/422), la validación de sesión y las notificaciones globales.

(3) Optimización de carga de datos
Para datos de tipo lista (como feed o comentarios), se incorporan mecanismos de paginación y caché para evitar solicitudes redundantes y mejorar el rendimiento.

La estructura general del flujo de red puede representarse como:

UI → capa de servicio → constantes de rutas → cliente HTTP → backend → base de datos

Para garantizar la coherencia entre frontend y backend, el sistema establece una correspondencia entre rutas, controladores y requisitos de autenticación, siguiendo el principio de “lectura anónima y escritura autenticada”.

En la implementación, todas las rutas API se centralizan en StarShipDH/src/lib/starshipApiPaths.ts, manteniendo correspondencia directa con los controladores del backend. Este enfoque evita la dispersión de rutas, reduce errores de sincronización y mejora la mantenibilidad del sistema.

Tabla de rutas

Módulo	Método	Ruta (Frontend)	Controlador	Autenticación	Observaciones
Autenticación	POST	/api/auth/login	AuthController	Anónimo	Devuelve token JWT y guarda sesión
Autenticación	POST	/api/auth/register	AuthController	Anónimo	Registro devuelve JWT directamente
Publicaciones	GET	/api/posts	PostController	Lectura anónima	Permite acceso sin autenticación

Publicaciones	GET	/api/posts/{id}	PostController	Lectura anónima	Datos personalizados si hay usuario
Publicaciones	GET	/api/posts/mine	PostController	Requiere login	Debe tener prioridad en rutas
Publicaciones	POST	/api/posts	PostController	Requiere login	Crear publicación
Likes	POST/DELETE	/api/posts/{postId}/like	PostController	Requiere login	Like / unlike
Comentarios	GET	/api/comments/post/{postId}	CommentController	Lectura anónima	Se usa en detalle del post
Comentarios	POST	/api/comments	CommentController	Requiere login	Crear comentario
Likes de comentarios	POST/DELETE	/api/comments/{commentId}/like	CommentController	Requiere login	Like / unlike
Perfil de usuario	GET/PUT	/api/me/profile	MeProfileController / MeSettingsController	Requiere login	Gestión de perfil
Seguimientos	GET/POST/DELETE	/api/follows/*	FollowController	Requiere login	Estructura preparada
Notificaciones	GET/POST	/api/notifications/*	NotificationController	Requiere login	Lectura y estado de notificación
Actividades	GET	/api/activities	ActivityController	Requiere login	Protegido por JWT

8.2.4 Autenticación y gestión de sesión

El sistema de autenticación no solo debe permitir el inicio de sesión, sino también definir claramente los límites entre acceso anónimo y operaciones autenticadas.

En este proyecto, el sistema se basa en JWT e incluye los siguientes aspectos:

(1) Gestión del ciclo de vida de la sesión

El token JWT se almacena en el cliente y se valida en cada solicitud mediante funciones como isStoredSessionValid(). El sistema contempla control de expiración e invalidación de sesión.

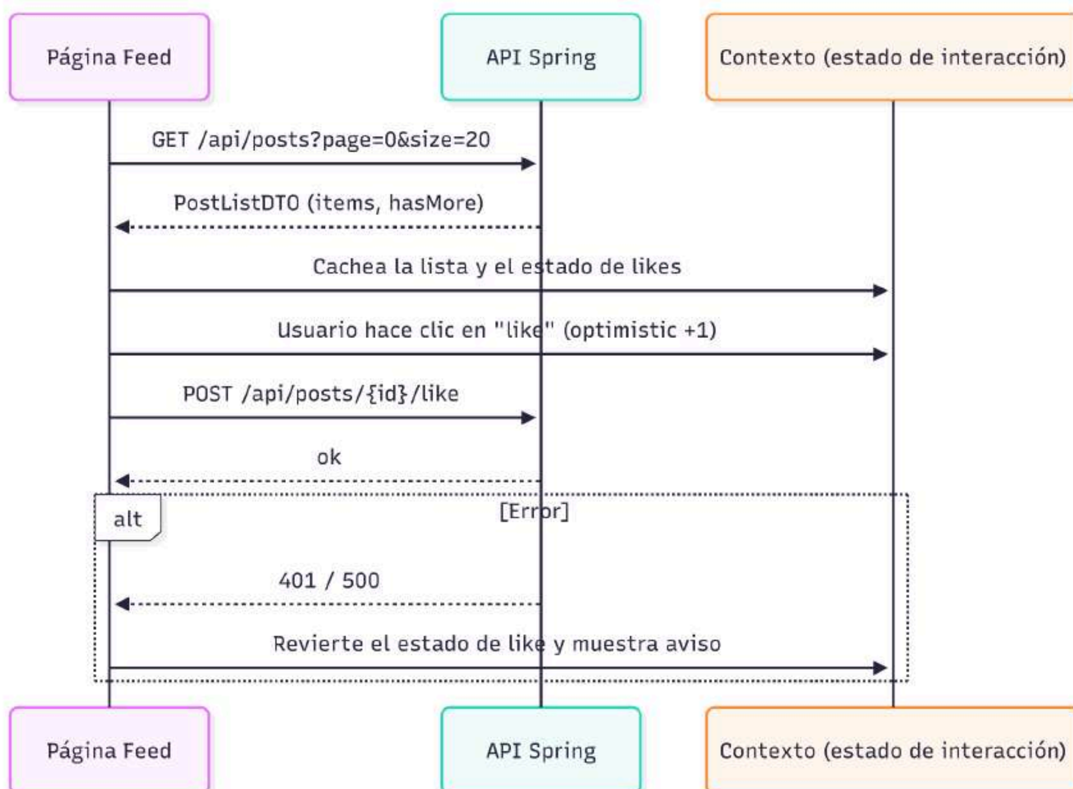
(2) Separación entre lectura y escritura
Se establece una política clara: el acceso a contenidos es permitido de forma anónima, mientras que las operaciones de escritura (publicar, comentar, dar "like") requieren autenticación.

(3) Manejo de errores de autenticación
En caso de fallo de autenticación, el sistema notifica al usuario, redirige al flujo de inicio de sesión y, cuando es posible, preserva el contexto actual para facilitar la continuidad tras autenticarse.

La separación entre páginas públicas y protegidas se implementa a nivel de enrutamiento: las páginas de login y registro se sitúan fuera de IonTabs, mientras que las páginas principales se protegen mediante ProtectedRoute. Además, se optimiza la interacción del usuario ocultando o deshabilitando acciones de escritura cuando no hay sesión válida, y gestionando de forma uniforme la expiración de la sesión.

8.2.5 Feed y mecanismos de interacción

El diseño del feed se centra en la optimización del rendimiento de listas y la consistencia de las interacciones. El sistema implementa un mecanismo estable de sincronización de estado para gestionar actualizaciones frecuentes de datos.
(Figura 8-12: Diagrama de secuencia de interacción de la lista de alimentación)



(1) Paginación y actualización por “pull-to-refresh”
El sistema controla la carga de datos mediante parámetros de paginación (page/size) e implementa un mecanismo de actualización manual, evitando la sobrecarga inicial de datos.

(2) Actualización optimista
El sistema actualiza inmediatamente el estado en la interfaz cuando el usuario realiza acciones como “like” o seguimiento, mientras envía la solicitud al backend en paralelo. En caso de error, el sistema revierte automáticamente el estado y notifica al usuario.

(3) Carga progresiva de comentarios
El sistema implementa carga por niveles mediante paginación o expansión bajo demanda, reduciendo la carga inicial de renderizado. Además, gestiona el foco de entrada para garantizar continuidad en la interacción.

(4) Estrategia de sincronización tras publicación
El sistema inserta el nuevo contenido en la parte superior del feed o recarga la lista completa tras una publicación exitosa, garantizando la consistencia de datos entre frontend y backend.

8.2.6 Subida de imágenes y gestión de recursos multimedia

El sistema implementa un flujo completo de gestión de imágenes que abarca procesamiento en cliente, transmisión de red y almacenamiento en servidor.

(1) Origen y preprocesamiento
El sistema permite seleccionar imágenes desde galería o cámara, y aplica compresión, recorte y validación de formato antes de la subida.

(2) Protocolo de subida
El sistema utiliza multipart/form-data como protocolo de transmisión, integrando archivos y datos de formulario en una sola solicitud para mejorar la eficiencia.

(3) Restricciones del backend
El sistema aplica en el frontend las restricciones definidas por el backend en cuanto a tamaño, cantidad y formato de archivos, garantizando la validez de los datos.

(4) Manejo de errores
El sistema implementa mecanismos de detección de errores y permite reintentos de subida, mejorando la estabilidad del proceso.

El flujo general de procesamiento es el siguiente:

selección de imagen → preprocesamiento → construcción de solicitud → subida → almacenamiento → persistencia de metadatos → retorno de URL

8.2.7 Notificaciones y perfil de usuario

El sistema de notificaciones proporciona retroalimentación sobre interacciones de usuario, incluyendo comentarios, “likes” y seguimiento.

El sistema obtiene las notificaciones mediante mecanismos de consulta periódica o actualización manual, y mantiene estados de lectura y no lectura de forma consistente.

El módulo de perfil de usuario permite la gestión de información personal y configuración del sistema, incluyendo edición de datos, actualización de avatar y control de privacidad, manteniendo coherencia con el sistema de autenticación.

8.2.8 Construcción y despliegue (Web / Capacitor)

El sistema soporta despliegue multiplataforma en Web y dispositivos móviles mediante Capacitor, con un flujo de construcción unificado y reproducible.

El proceso de construcción se define como:

entorno de desarrollo (Vite) → compilación TypeScript → empaquetado frontend → despliegue web o empaquetado móvil

El sistema gestiona permisos y compatibilidad entre plataformas, incluyendo acceso a cámara, galería y sistema de archivos.

El flujo de construcción y pruebas se define en package.json, incluyendo desarrollo, construcción, pruebas unitarias, pruebas end-to-end y análisis de código. El proyecto utiliza @capacitor/*, @ionic/react y axios, garantizando un flujo completo de desarrollo, pruebas y despliegue consistente.

8.3 Backend y base de datos (FastAPI + Spring Boot + PostgreSQL)

Este apartado describe la arquitectura del backend dividida en dos subsistemas principales: el backend del juego basado en FastAPI y el backend de la comunidad implementado con Spring Boot. Ambos sistemas comparten una única instancia de PostgreSQL, pero utilizan esquemas separados para garantizar una separación clara de dominios y una mejor mantenibilidad del sistema.

8.3.1 Backend del juego (FastAPI): organización de API y arquitectura basada en datos

El backend del juego gestiona la autenticación, la persistencia de personajes, la sincronización de guardado en la nube y la distribución de datos estáticos del sistema de juego (objetos, armas, habilidades, enemigos y genes). La arquitectura organiza las rutas por dominios funcionales como auth, character, inventory, skills y game-data.

El sistema define los modelos de entrada y salida mediante **Pydantic**, lo que establece un contrato estricto entre cliente y servidor. Los datos estáticos se exponen a través de rutas de solo lectura (/game-data/*), lo que garantiza consistencia y evita modificaciones en tiempo de ejecución desde el cliente.

El punto de entrada del backend se encuentra en StarshipBackend/PSQL_DH/main.py. Este archivo define claramente la estructura de rutas del sistema:

- Autenticación: /register, /login, /me, /send_verification, /verify_email
- Gestión de personajes: /characters y subrecursos como /inventory, /skills, /stats, /genes, /scene_state
- Datos estáticos: /game-data/items, /weapons, /skills, /genes, /enemies
- Operaciones de genes: /gene-modules/unlock, /gene-modules/upgrade

El archivo APIManager.gd del cliente implementa una correspondencia directa con estas rutas, lo que establece una arquitectura coherente de tres capas: cliente, API y base de datos.

Los esquemas **Pydantic** definidos en schemas.py establecen la estructura formal de los datos. CharacterStatsRequest y CharacterStatsResponse definen el estado del personaje; InventorySaveRequest utiliza estructuras de slots fijos; **skills** se modela mediante diccionarios indexados por **api_key**; y el sistema de genes se estructura como entidades independientes con módulos asociados. La autenticación se implementa mediante JWT utilizando OAuth2PasswordBearer, con expiración estándar de 24 horas.

8.3.2 Consistencia de guardado: sincronización local y en la nube

El sistema de guardado utiliza un modelo híbrido compuesto por guardado local rápido y sincronización en la nube. El guardado local garantiza la recuperación ante fallos del sistema, mientras que el guardado en la nube asegura la persistencia entre dispositivos.

El sistema mantiene tres variables de control de versión: local_revision, cloud_ack_revision y pending_cloud_sync, lo que permite detectar diferencias entre estados locales y remotos.

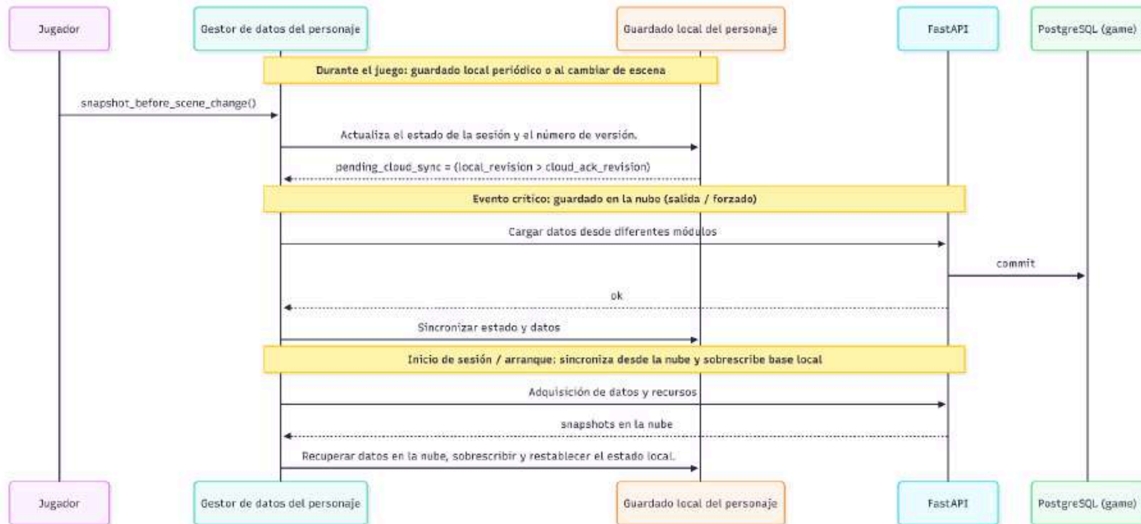
El sistema local está implementado en LocalCharacterSave.gd, donde los datos se almacenan mediante archivos cifrados en user://local_character_save/. El sistema incrementa local_revision en cada checkpoint.

Cuando local_revision > cloud_ack_revision, el sistema activa la sincronización pendiente. Tras una escritura exitosa en el servidor, cloud_ack_revision se actualiza y se sincroniza con el estado local.

En el arranque del sistema, los datos de la nube sobrescriben el estado local como fuente de verdad autoritativa, garantizando consistencia entre sesiones.

El backend aplica reglas de progresión mediante niveles efectivos y sistemas de “breakthrough”, limitando el progreso en función de umbrales definidos en DEFAULT_SYNC_BREAKTHROUGH_GATE_LEVELS.

(Figura 8-13: Diagrama de tiempos de archivo local y en la nube)



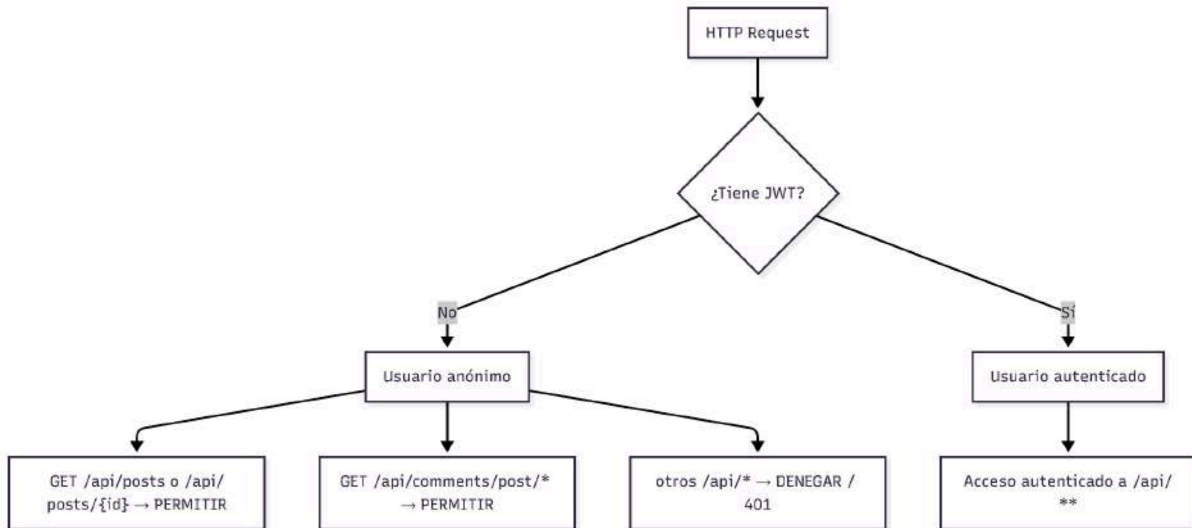
8.3.3 Backend de comunidad (Spring Boot): arquitectura y reglas de seguridad

El backend de comunidad implementa un sistema basado en Spring Boot con arquitectura en tres capas: Controller, Service y Repository. Este diseño separa claramente la lógica de presentación, negocio y persistencia.
El sistema implementa una política de seguridad basada en JWT y un modelo de acceso dual: lectura anónima y escritura autenticada. Esta separación garantiza escalabilidad y control de acceso consistente.
Se define además un sistema de respuesta uniforme mediante ApiResponse, lo que simplifica la integración con el frontend.

La configuración de seguridad se encuentra en SecurityConfig.java. El sistema utiliza SessionCreationPolicy.STATELESS, lo que garantiza que no existe estado de sesión en el servidor.
Las reglas de acceso establecen:

- Acceso anónimo: /api/auth/**, GET /api/posts, GET /api/posts/*, GET /api/comments/post/**
- Acceso autenticado: todas las demás rutas /api/**

(Figura 8-14: Diagrama de reglas para la visualización anónima y el inicio de sesión)



El sistema incluye una restricción crítica de orden de rutas, donde /api/posts/mine se evalúa antes de /api/posts/* para evitar colisiones de reglas.
Los controladores principales incluyen:

- **AuthController**: autenticación y registro
- **PostController**: gestión de publicaciones y "likes"
- **CommentController**: comentarios y reacciones
- **FollowController**: sistema de seguimiento
- **NotificationController**: notificaciones
- **ActivityController**: registro de actividad

8.3.4 Diseño de base de datos: separación por esquemas y modelo relacional

La base de datos PostgreSQL se organiza mediante múltiples esquemas para separar dominios funcionales. El esquema **game** gestiona la lógica del juego, mientras que el esquema **community** gestiona la interacción social.
El sistema utiliza claves primarias normalizadas y relaciones uno-a-muchos y muchos-a-muchos para representar usuarios, personajes, inventarios, habilidades, genes, publicaciones y comentarios.

El archivo base de la base de datos se encuentra en DesastreHuman.sql, el cual inicializa todos los esquemas del sistema.
El proyecto utiliza migraciones incrementales ubicadas en /migrations, lo que permite la evolución controlada del esquema sin romper compatibilidad.

(Figura 8-15(a): Diagrama entidad-relación del esquema del juego)

(Figura 8-15(b): Diagrama entidad-relación del esquema de la comunidad)

8.3.5 Migración y estrategia de inicialización

El sistema utiliza un modelo de doble capa compuesto por script base y migraciones incrementales. El script base permite la inicialización completa de un entorno vacío, mientras que las migraciones permiten la evolución progresiva del esquema.

Las migraciones garantizan la compatibilidad hacia atrás y la trazabilidad de cambios estructurales.
El sistema distingue entre:

- Entornos nuevos: inicialización mediante script base
- Entornos existentes: actualización mediante migraciones incrementales

8.4 Partes transversales del sistema (testing, depuración, optimización, despliegue y gestión de versiones)

Este proyecto integra un frontend de comunidad (StarShipDH basado en Ionic/React/Capacitor), un backend de comunidad (Spring Boot), un backend de juego (FastAPI) y un cliente en Godot, formando una arquitectura multiplataforma. Por ello, el desarrollo implica problemas transversales como la consistencia de contratos de API, la eficiencia de integración, la estabilidad del rendimiento y la reproducibilidad del despliegue. Esta sección sintetiza de forma unificada los mecanismos de testing, depuración, optimización, seguridad y gestión de versiones desde una perspectiva de ingeniería de sistemas.

8.4.1 Estrategia de pruebas (unitarias, integración y contrato)

Debido a que el proyecto integra una aplicación frontend (StarShipDH basada en Ionic + React + Capacitor), un backend de servicios comunitarios (Spring Boot), un backend de lógica de juego (FastAPI) y un cliente en Godot, se han identificado problemas transversales de ingeniería como la consistencia de contratos API, la eficiencia de integración, la estabilidad de rendimiento y la reproducibilidad del despliegue. Por ello, esta sección consolida los mecanismos comunes del sistema para evitar redundancia en capítulos funcionales y reflejar el diseño a nivel arquitectónico.

8.4.1 Estrategia de pruebas (unitarias, integración y contractuales)

El sistema de pruebas se estructura en tres niveles, garantizando la coherencia desde la lógica interna hasta el comportamiento global del sistema.

(1) Pruebas unitarias
Las pruebas unitarias validan lógica determinista, incluyendo fórmulas de daño, progresión de experiencia, cálculos de probabilidad, validaciones de permisos y reglas de negocio básicas.

(2) Pruebas de integración
Las pruebas de integración verifican la interacción entre componentes, incluyendo autenticación, controladores REST, acceso a base de datos y flujos CRUD completos.

(3) Pruebas contractuales
Las pruebas contractuales garantizan la coherencia entre cliente, frontend y backend, validando rutas API, estructuras de datos, tipos y formatos de respuesta.

El frontend (StarShipDH) integra **Vitest** y **Cypress** para pruebas **unitarias** y **E2E**.

El backend en Spring Boot implementa pruebas con MockMvc para controladores de autenticación, publicaciones, comentarios, seguidores, notificaciones y perfil de usuario.
FastAPI utiliza pytest junto con validación de OpenAPI para garantizar la coherencia de los contratos API.
El cliente en Godot no dispone aún de un framework formal de testing, aunque incluye scripts de verificación funcional para lógica básica.

8.4.2 Depuración y mecanismos de diagnóstico

El sistema implementa un modelo unificado de observabilidad para facilitar la detección y reproducción de errores.

(1) Depuración en frontend
Se utiliza Vite Dev Server y herramientas de navegador para inspección de UI, red y estado de la aplicación.

(2) Depuración en backend
Spring Boot utiliza un manejador global de excepciones que devuelve respuestas estructuradas.
FastAPI expone OpenAPI para depuración interactiva y validación de contratos.

(3) Depuración en cliente (Godot)
El cliente implementa un sistema global de notificaciones (Toast) para mostrar errores de ejecución, fallos de carga y eventos relevantes del sistema en tiempo real.

El sistema **GlobalMessage** centraliza la salida de eventos críticos en Godot, incluyendo errores de guardado, fallos de red y eventos de tutorial.

8.4.3 Optimización de rendimiento y compatibilidad

La optimización del sistema se aplica en cuatro niveles: red, renderizado, carga y acceso a datos.

(1) Optimización en tiempo de ejecución
El frontend reduce re-renderizados innecesarios mediante virtualización de listas.
El backend optimiza consultas mediante índices en la base de datos.
El cliente utiliza object pooling para reducir asignaciones frecuentes.

(2) Optimización de carga
Se aplican estrategias de lazy loading y segmentación de recursos en frontend.
El backend utiliza paginación para reducir la carga de transferencia de datos.
El cliente en Godot emplea carga asíncrona de escenas.

(3) Compatibilidad
El sistema soporta ejecución en Web y dispositivos móviles, con adaptaciones de interfaz y estrategias de degradación para dispositivos de bajo rendimiento.

8.4.4 Seguridad y control de acceso

El sistema utiliza autenticación JWT sin estado como base de seguridad.

Spring Boot implementa reglas de acceso basadas en permisos diferenciando rutas públicas y privadas mediante SecurityConfig.
FastAPI valida tokens mediante inyección de dependencias, asegurando el aislamiento de datos de usuario.

Se aplican validaciones estrictas de entrada y respuestas de error estandarizadas para reducir la complejidad en el cliente.

8.4.5 Construcción, despliegue y gestión de versiones

El sistema adopta una arquitectura multi-módulo con aislamiento de entornos para garantizar reproducibilidad.

(1) Construcción
El frontend utiliza Vite y Capacitor para despliegue multiplataforma.
Spring Boot utiliza Maven con perfiles de entorno diferenciados.
FastAPI se integra en pipelines CI mediante pytest.
Godot se exporta como cliente multiplataforma.

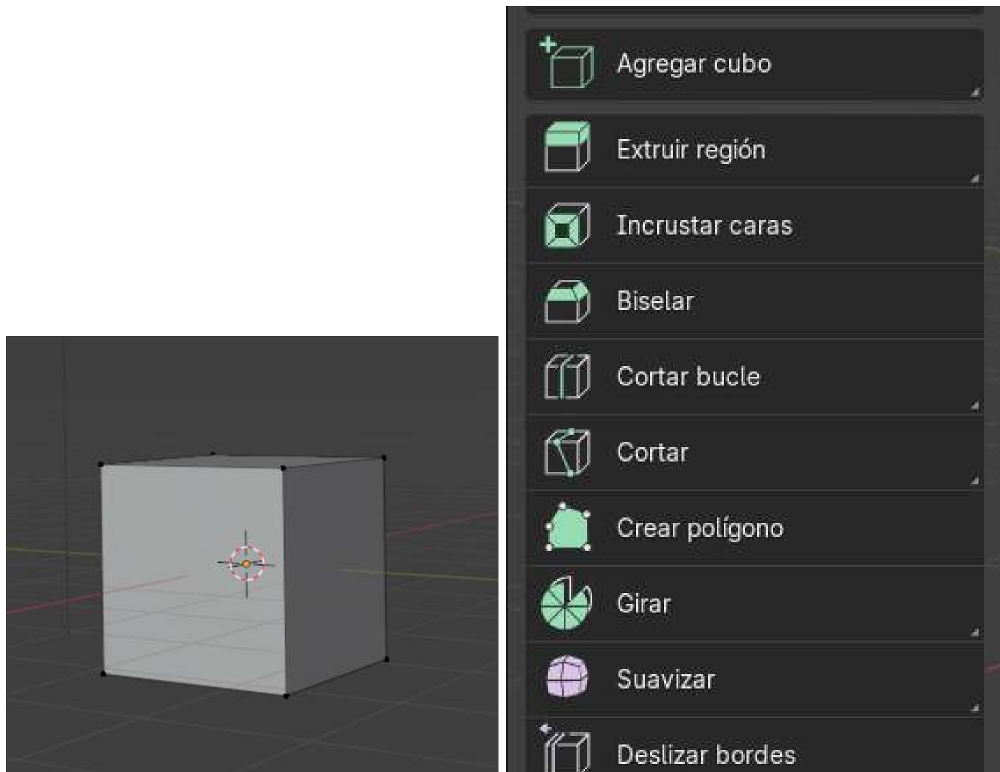
(2) Despliegue
El sistema utiliza despliegue basado en contenedores o hosting estático.
La base de datos evoluciona mediante scripts de migración controlados a partir de un script base único.

(3) Gestión de versiones
Se utiliza control de versiones de esquema para garantizar compatibilidad entre cliente, backend y base de datos, asegurando coherencia entre almacenamiento local y en la nube.

9.2 Proceso de implementación en Blender

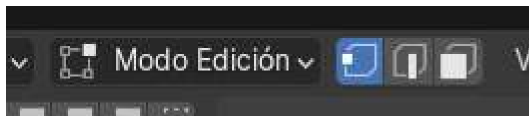
9.2.1 Modelado

Al comenzar con el modelado, normalmente empezamos con un cubo. Puedes añadir un cubo usando el atajo **Shift + A** y seleccionando "Malla" en el menú. Después, ingresamos al modo de edición presionando **Tab**. En el modo de edición, utilizamos las herramientas de la barra lateral izquierda, como extruir, mover y cortar, para darle forma al modelo que queremos crear.

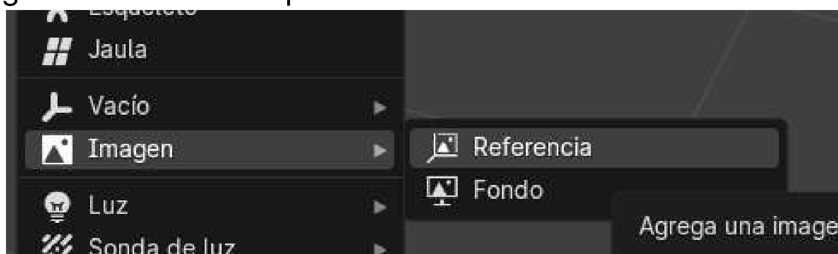


En el modo de edición, hay tres tipos de selección: modo vértice, modo arista y modo cara. Estos modos se pueden alternar dependiendo de cómo queramos manipular el modelo:

- Modo vértice: Permite seleccionar y mover puntos individuales.
- Modo arista: Permite seleccionar y mover bordes del modelo.
- Modo cara: Permite seleccionar y manipular caras enteras del modelo.



También podemos añadir una imagen de referencia usando el atajo Shift + A para guiarnos durante el proceso de modelado.



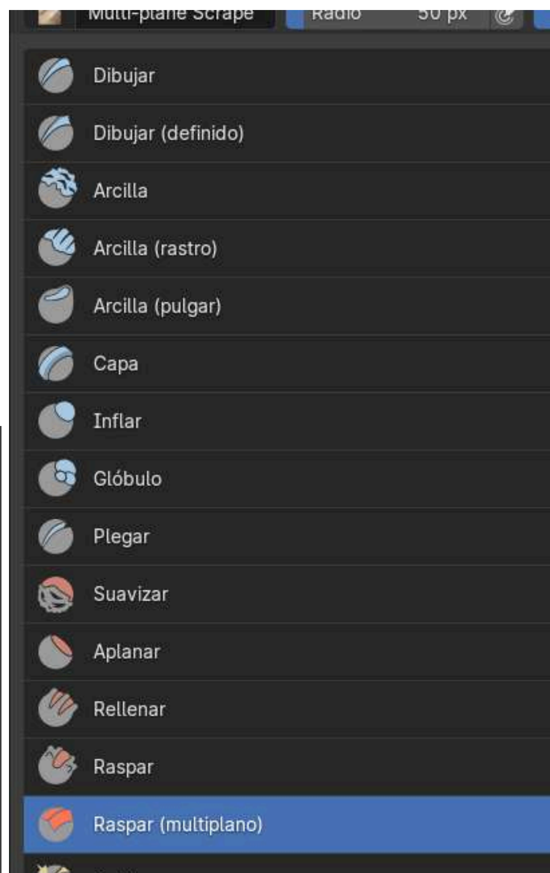
9.2.2 Esculpido

Blender tiene muchas herramientas de esculpido que nos permiten añadir detalles finos a nuestros modelos. Antes de usar estas herramientas, es crucial asegurarnos de que nuestro modelo tenga suficientes caras. Si el modelo es de baja resolución, las herramientas de esculpido no funcionarán bien.

Para aumentar la cantidad de caras, podemos aplicar un modificador de "**subdivisión de superficie**". Este modificador divide cada cara del modelo en partes más pequeñas, haciendo que el modelo sea más suave y detallado.

En el modo de esculpido, tenemos acceso a una variedad de pinceles que nos permiten realizar diferentes tipos de esculpido, como:

- **Dibujar**: Añade volumen al modelo.
- **Suavizar**: Suaviza la superficie del modelo.
- **Agarrar**: Permite agarrar y mover partes del modelo.

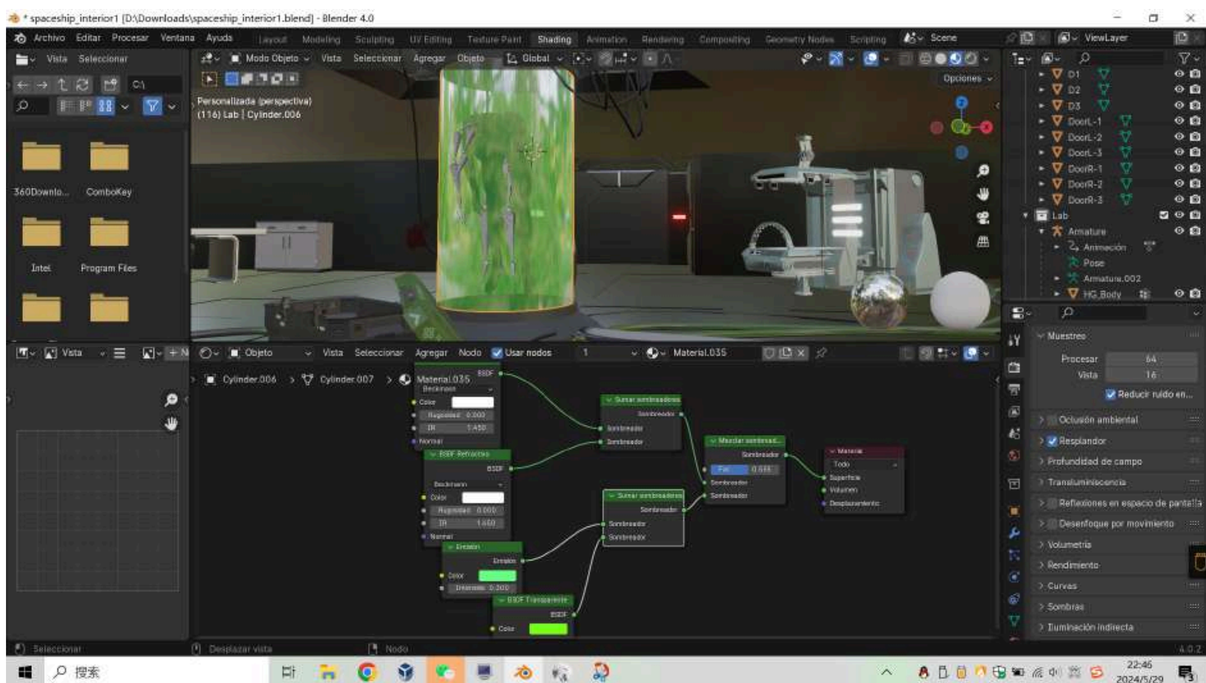


9.2.3 Sombreado

El sombreado en Blender implica la creación de materiales y texturas que se aplican a la superficie de los modelos 3D. En el Editor de Sombreado, podemos crear materiales complejos conectando diferentes nodos. Algunos de los nodos más utilizan incluyen:

- **Principled BSDF:** Un nodo versátil que combina varios tipos de sombreado en uno solo.
- **Image Texture:** Permite aplicar una imagen como textura en el modelo.
- **Mix Shader:** Combina dos shaders diferentes.

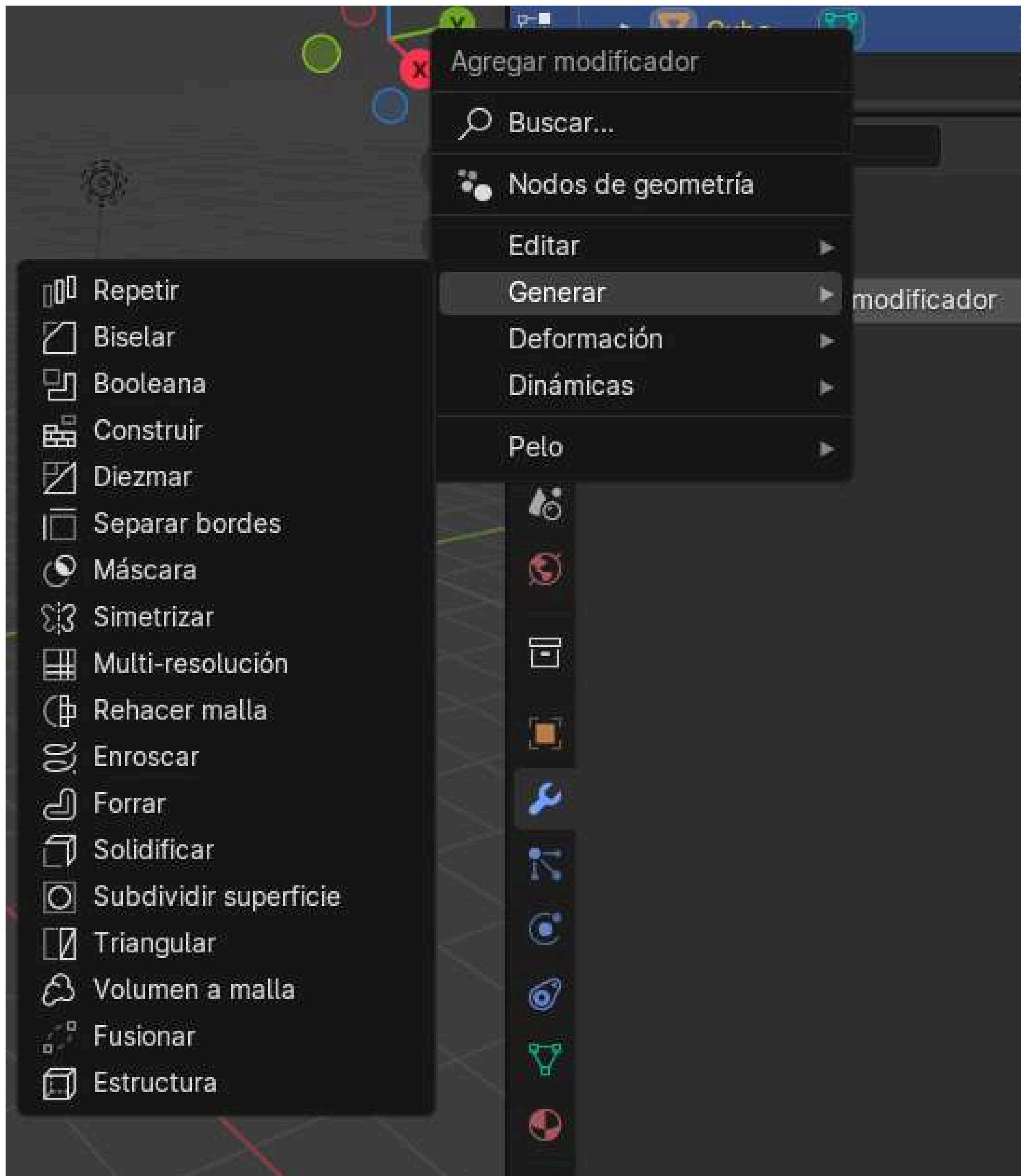
Mediante la conexión de estos nodos, podemos definir cómo se verá la superficie del modelo bajo diferentes condiciones de iluminación. Este proceso nos permite crear materiales realistas y detallados.



9.2.4 Modificador

Los modificadores en Blender son herramientas que nos ayudan a modificar y optimizar nuestros modelos de manera no destructiva. Algunos de los modificadores más útiles incluyen:

- **Espejo**: Crea una copia simétrica del modelo a lo largo de un eje. Para modelar objetos simétricos, como personajes o vehículos, ya que solo necesitamos modelar una mitad y el modificador se encargará de la otra.
- **Subdivisión de superficie**: Añade más subdivisiones a la geometría del modelo, haciéndolo más suave y detallado.
- **Repetir**: Duplica el modelo en una matriz, sirve para crear objetos repetitivos como columnas o ruedas.



9.2.5 UV

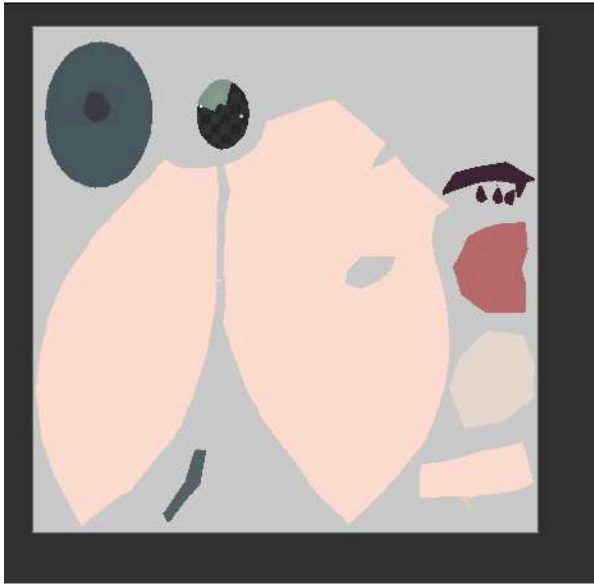
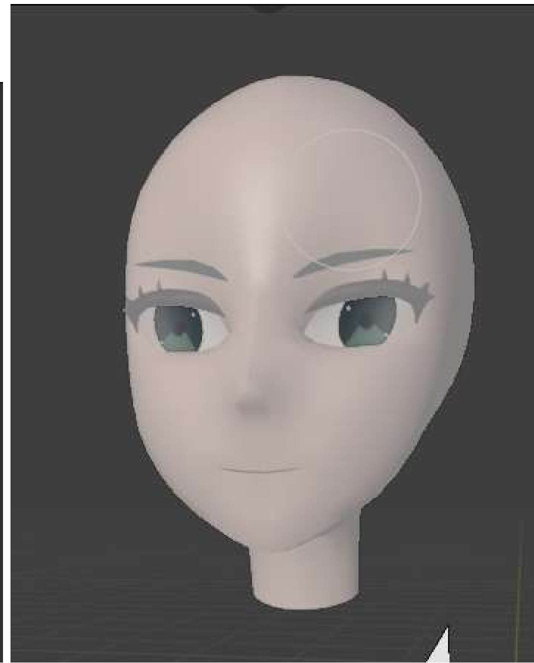
El proceso de UV Mapping se encarga de definir cómo las texturas se proyectan sobre la superficie del modelo 3D. En esta fase se realiza el desplegado (unwrap) de la malla, transformando la geometría tridimensional en un espacio bidimensional donde se pueden aplicar texturas de forma controlada.

Durante este proceso se ajustan las islas UV para minimizar deformaciones, reducir estiramientos y optimizar el uso del espacio de textura. También se reorganizan manualmente las zonas más relevantes del modelo (como rostro, manos o elementos principales del personaje) para facilitar una correcta lectura visual de las texturas en el motor de renderizado.

El objetivo principal de esta etapa es garantizar una correcta correspondencia entre la geometría del modelo y su representación visual, evitando errores de distorsión en el resultado final dentro de Godot.

9.2.6 Paint

La fase de pintura de texturas se utiliza principalmente para añadir información de color, detalles visuales y definir el estilo artístico del modelo. En este proyecto, se emplearon principalmente las herramientas de pintura integradas en Blender para la creación y ajuste de las texturas.
Durante este proceso, el modelo se pinta de forma manual o semiautomática, incluyendo la definición del color base, variaciones de luz y sombra, adición de detalles y tratamientos estilizados, con el objetivo de lograr una coherencia visual acorde al estilo general del proyecto.
En este caso concreto, este proceso se utilizó principalmente para el coloreado de un nuevo modelo de personaje. A partir de un mapeado UV correctamente desplegado, se realizó la pintura por zonas, lo que permitió mejorar la eficiencia y el control del trabajo. En la práctica, se utilizó con mayor frecuencia la herramienta airbrush (aerógrafo) para obtener transiciones suaves y degradados, junto con herramientas de máscara (mask) para controlar de forma más precisa las distintas áreas del modelo y gestionar la superposición de colores y detalles.

10. Pruebas y ejecución

10.1 Metodología de pruebas y selección de herramientas

Las pruebas unitarias se utilizan principalmente para verificar el comportamiento lógico de las unidades mínimas testeables, como funciones puras, validaciones o métodos aislados, sin depender de I/O externo ni de entornos complejos de ejecución. Algunos ejemplos representativos del proyecto incluyen:

- **test_experience_level_math.py**: valida la curva de experiencia y asegura la coherencia entre el sistema de niveles de Godot y la lógica de cálculo del backend.
- **test_resistance_clamp_logic.py**: verifica la lógica de limitación de resistencias y su consistencia con la ruta de guardado save_character_stats.
- **test_schemas_stats.py** y **test_schemas_game.py**: validan los límites y estructuras de los modelos Pydantic principales.
- **test_auth_jwt**.py: comprueba la generación, validación y configuración de JWT.
- **AuthApplicationServiceTest** y **CommentApplicationServiceTest**: validan ramas lógicas de los servicios de aplicación en Spring Boot.

Las pruebas de integración se enfocan en la colaboración entre múltiples componentes reales. Por ejemplo:

- **test_integration_db.py**: al habilitar RUN_INTEGRATION=1 y configurar DATABASE_URL apuntando a PostgreSQL inicializado, verifica la conexión a la base de datos y la carga de la aplicación.

- **AuthApiIntegrationTest**: inicia un contexto completo de Spring Boot mediante puertos aleatorios y realiza validaciones HTTP de extremo a extremo.

Las pruebas de interfaz y contrato se centran principalmente en la estabilidad de métodos HTTP, rutas, códigos de estado y estructuras de respuesta unificadas, sin profundizar en cada rama de negocio. Algunos ejemplos incluyen:

- **test_openapi_contract.py**: fija el subconjunto de rutas FastAPI utilizado por **Godot APIManager**.
- **test_http_validation_unauth.py**: valida el orden de retorno entre 401 Unauthorized y 422 Validation Error.
- **test_app_smoke.py**: realiza pruebas smoke sobre /openapi.json y /docs.
- Los distintos ***ControllerTest** de Spring Boot utilizan MockMvc para validar URLs, códigos HTTP y la estructura JSON de **ApiResponse**.
- **test_migrations_repo.py:** verifica la existencia de scripts base SQL y archivos de migración.

10.2 Motivos de selección de herramientas

En el backend del juego se eligió **pytest** como framework principal debido a que es el estándar de facto dentro del ecosistema Python. Sus mecanismos como **fixture**, **parametrize** y **markers** permiten separar fácilmente pruebas sin base de datos y pruebas opcionales con PostgreSQL. Además, **FastAPI TestClient** puede ejecutar pruebas HTTP dentro del mismo proceso, facilitando regresiones frecuentes de APIs.

En Spring Boot se optó por **MockMvc + JUnit 5**. MockMvc permite simular peticiones HTTP sin necesidad de navegador, y mediante **standaloneSetup** junto con el manejador global de excepciones se puede mantener estable la estructura JSON de errores y respuestas **ApiResponse**. Adicionalmente, **MockMvcUserPrincipal** resuelve diferencias relacionadas con la inyección de **Security Principal** durante pruebas segmentadas.
Para Godot, las pruebas se apoyan principalmente en:

- api_test.tscn
- test/unit/run_unit_tests.gd

Además de pequeños escenarios de prueba y validaciones manuales de escenas, scripts y sistemas interactivos dentro del juego.
Actualmente, la parte frontend de la aplicación comunitaria sigue dependiendo principalmente de pruebas manuales y de la verificación de compilación correcta del proyecto.
Cabe señalar que el proyecto aún no incluye pruebas de rendimiento a gran escala, pruebas de carga ni pruebas prolongadas de tipo **soak test**. Por ello, no existen métricas cuantitativas relacionadas con **QPS**, latencia **P99** o límites del pool de conexiones. Esta limitación se reconoce explícitamente y se considera una posible línea de mejora futura.

10.3 Resumen de archivos de prueba

10.3.1 FastAPI

Archivo	Enfoque principal
test_openapi_contract.py	Contrato OpenAPI de rutas y métodos
test_http_validation_unauth.py	Orden entre autenticación y validación
test_auth_jwt.py	JWT y autenticación
test_schemas_stats.py	DTO de estadísticas de personaje
test_schemas_game.py	DTO de inventario, habilidades, genes y escenas
test_experience_level_math.py	Matemática de experiencia y niveles
test_resistance_clamp_logic.py	Lógica de resistencias
test_main_get_character.py	Consulta de personajes mediante mocks
test_app_smoke.py	Smoke test de /openapi.json y /docs
test_migrations_repo.py	Verificación de scripts SQL y migraciones
test_integration_db.py	Integración con PostgreSQL
conftest.py	Configuración de entorno y fallback SQLite

10.3.2 Springboot

Clase de prueba	Enfoque
AuthControllerTest	Registro, login y contrato HTTP
PostControllerTest	Lectura/escritura de posts y acceso anónimo
CommentControllerTest	Comentarios
FollowControllerTest	Sistema de seguimiento
NotificationControllerTest	Notificaciones

MeProfileControllerTest / MeSettingsControllerTest	Perfil y configuración
ActivityControllerTest	Actividades
AuthApplicationServiceTest	Servicio de autenticación
CommentApplicationServiceTest	Servicio de comentarios
StarshipApplicationTests	Smoke test de contexto con H2
AuthApiIntegrationTest	Integración end-to-end con base de datos real

10.3.3 Juego y la aplicación de comunidad

10.3.3.1 Cliente de juego（Godot）

Las pruebas del cliente de juego se componen principalmente de pruebas headless mediante scripts, pruebas de integración API y validaciones manuales dentro de escenas reales.
Entre ellas:

- **run_unit_tests.gd** se utiliza para ejecutar pruebas de lógica pura del cliente y validar parte de los sistemas básicos.
- **api_test.tscn** se utiliza para realizar pruebas de integración con el backend FastAPI, incluyendo registro, login, JWT, datos de personaje y llamadas a /game-data/*.
- Las pruebas manuales cubren principalmente creación de personajes, cambio de escenas, sincronización de guardado, sistema de habilidades, sistema genético y flujo del tutorial.

Actualmente el cliente todavía no dispone de un sistema completo de automatización para combate, IA enemiga y validación de todos los niveles, por lo que parte importante de las pruebas sigue realizándose manualmente durante la partida. Por ejemplo:

- flujo y restricciones del tutorial;
- reproducción y transición de audio;
- shaders, efectos visuales y renderizado;
- animaciones UI y feedback visual;
- sincronización entre guardado local cifrado y control de versiones.

Dentro de las pruebas relacionadas con consistencia de guardado y sincronización en la nube se verifican aspectos como:

- correcta generación de checkpoints locales;

- actualización de local_revision y cloud_ack_revision;
- recuperación de sincronización tras cierres inesperados;
- fallback correcto al caché local cuando falla la nube.

En términos generales, la estrategia actual de pruebas del cliente Godot se centra en una combinación de “automatización para lógica crítica + validación manual del gameplay principal”.

10.3.3.2 Aplicación de comunidad

La aplicación de comunidad（StarShipDH）se apoya principalmente en pruebas manuales y validación de builds. Las pruebas automatizadas existen como soporte opcional.
El proyecto ya incluye:

- npm run test.unit（Vitest）
- npm run test.e2e（Cypress）

para pruebas de componentes y flujos end-to-end, aunque actualmente no forman parte obligatoria del pipeline CI.
Las pruebas principales realizadas en esta etapa incluyen:

- flujo de login y registro;
- carga de lista y detalle de publicaciones;
- autenticación JWT y validación de permisos;
- verificación en entorno móvil mediante Capacitor.

Debido a que el frontend soporta tanto entorno Web como Capacitor, algunos problemas no pueden detectarse únicamente mediante pruebas unitarias y requieren validación manual en entorno real, por ejemplo:

- configuración correcta de **VITE_API_URL**;
- funcionamiento del proxy de Vite en desarrollo;
- problemas de CORS o permisos tras empaquetado con Capacitor;
- persistencia del estado de login en móvil;
- comportamiento de rutas y refresh de páginas.

Además, actualmente el proceso de build de producción y empaquetado con Capacitor todavía no está integrado completamente en CI, por lo que antes de cada release se ejecutan manualmente:

- npm run build
- Capacitor sync/build
- Pruebas en dispositivo físico o emulador.

En conjunto, las pruebas actuales del frontend de comunidad se enfocan principalmente en la validación de interfaces, comportamiento de autenticación y consistencia de datos entre frontend y backend.

11. Riesgos, retos, innovaciones y perspectivas de la investigación

11.1 Riesgos que se pueden encontrar en el proyecto

La siguiente fichero de excel contiene una descripción general y una evaluación de los riesgos que podemos encontrar en nuestros proyectos:
Riesgos de DH - Jiayi Chen

11.2 Desafíos y soluciones de la investigación

Problema	**Manifestación o causa raíz**	**Solución implementada (según el repositorio)**
Conflictos de concurrencia en el guardado en la nube	Peticiones paralelas de stats, scene_state y loadout pueden sobrescribirse entre sí, provocando pérdida de datos (p. ej., slots de armas).	Unificación de stats y scene_state en una única transacción POST /characters/{id}/stats. Otros módulos (inventario, habilidades, genes) se mantienen independientes para evitar colisiones de campos JSON.
Inconsistencia entre cliente, API y base de datos	Desalineación entre ORM, schemas.py, migraciones SQL y serialización en Godot (save_to_dict) genera campos faltantes o truncados.	Uso de models.py y migraciones como fuente de verdad. Se mantiene una tabla de trazabilidad del sistema completo. Validaciones defensivas: clamp en cliente y límites en backend.
Bloqueo de interacción por UI (Toast)	MOUSE_FILTER no se hereda en la jerarquía, impidiendo interacción con UI inferior.	Aplicación recursiva de MOUSE_FILTER_IGNORE en todo el subárbol del Toast.
Duración incorrecta de animaciones Toast	Combinación de Tween.chain() y set_parallel() provoca inconsistencias en el tiempo de visualización.	Separación de responsabilidades: Tween para animación y Timer para el tiempo de permanencia.
Duplicación de lógica de experiencia y nivel	La lógica de progresión se implementa tanto en UI como en backend, generando inconsistencias.	Centralización en Stats mediante una API única (get_level_experience_segment).

Modelo de atributos incompatible (jugador vs enemigos)	Jugadores usan progresión por experiencia, enemigos niveles fijos, lo que dispersa la lógica.	Diseño dual en Stats: modo derivado por experiencia o nivel fijo (fixed_combat_level).
Conflicto de arquitectura con WeaponManager	Uso simultáneo como Autoload y nodo de escena rompe referencias a cámara y jugador.	Eliminación de Autoload; el sistema se integra como nodo hijo del jugador con auto-detección en _ready().
Problemas de despliegue de red	Uso de localhost impide conexión entre cliente y backend en VM.	Uso de IP de red local de la máquina virtual y documentación explícita en NETWORK_DEPLOYMENT.md.
Desalineación de esquema SQL	No ejecutar migraciones completas provoca campos faltantes o errores silenciosos.	DesastreHuman.sql definido como script base único de inicialización de base de datos.
Deriva entre documentación y código	Cambios en rutas o scripts no reflejados en documentación.	Uso de Issue Log y TODO centralizados como fuente de sincronización continua.
Deuda técnica en scripts y señales	Existencia de scripts duplicados (WeaponViewModel vs view_model.gd) y uso parcial de SignalBus.	Migración progresiva controlada con auditoría de referencias antes de eliminar o consolidar.
Persistencia incompleta de cooldowns	El backend no almacena cooldown_remaining, causando reinicio de estados.	Decisión de diseño: simplificación del backend; posibilidad de extensión futura del esquema.
Problemas de renderizado en feed social	URLs inválidas o blob: no convertidos generan imágenes rotas.	Validación de URLs en frontend y conversión blob → data URL antes de enviar al servidor.
Desalineación de rutas API	Endpoints hardcodeados en múltiples archivos generan inconsistencias tras cambios.	Centralización en starshipApiPaths.ts como única fuente de rutas.
Sistema de colisiones inconsistente	Uso de capas y máscaras sin convención clara provoca fallos en detección de rayos e interacciones.	Estandarización de 9 capas en project.godot y definición de máscaras (MASK_INTERACT_RAY, MASK_BULLET, etc.). Separación de CharacterBody3D y Area3D.

Rayos físicos sin filtrado adecuado	PhysicsRayQuery sin máscara específica detecta objetos no deseados.	Definición explícita de máscaras (MASK_AIM_TARGET, MASK_SKILL_MOUSE_RAY).
Acceso prematuro a sistemas del menú	Usuarios sin tutorial completado acceden a funciones avanzadas.	Control de acceso basado en /stats mediante fetch_stats_snapshot_for_menu y deshabilitación dinámica de UI.
Duplicidad de fuentes de datos estáticos	Inconsistencia entre JSON (game_data) y recursos .tres.	GameDataManager como fuente central de datos en tiempo de ejecución, evitando divergencias.
Corrupción de escena y código	Los archivos del proyecto se corrompieron debido a que no se les dio el mantenimiento adecuado.	Descarga el archivo intacto de Github y sobrescríbelo.

11.3 Puntos de Innovación

11.3.1. Coordinación por dominios en un sistema multi-backend heterogéneo

El sistema adopta una arquitectura de **doble backend desacoplado**, donde:

- El sistema de juego se implementa con FastAPI (gestión de personajes, guardado y datos estáticos)
- El sistema de comunidad se implementa con Spring Boot (posts, comentarios, relaciones sociales y notificaciones)

Ambos servicios operan sobre una base de datos compartida estructurada mediante **múltiples esquemas (auth, game, community, chat)**, lo que permite reutilizar el sistema de autenticación manteniendo aislamiento entre dominios.

11.3.2. Arquitectura de capacidades transversales basada en Autoload (Godot)

En el cliente de juego se define un conjunto de singletons globales (Autoload) que encapsulan funcionalidades transversales:

- Red, datos estáticos, guardado, sistema de tutoriales
- Gestión de UI, escenas, audio, habilidades y sistema genético

Este enfoque reduce el acoplamiento entre escenas y evita pérdida de estado, consolidando una infraestructura reutilizable a nivel global.
Adicionalmente, se implementa una **capa unificada de acciones de entrada**, permitiendo compartir semántica entre combate, UI y sistema de tutoriales.

11.3.3. Estrategia de consistencia: “guardado local cifrado + persistencia en la nube”

El sistema de guardado combina:

- **Snapshots** locales cifrados para tolerancia a fallos
- **Persistencia remota** como fuente de verdad

Se introduce un modelo basado en tres variables:

- local_revision
- cloud_ack_revision
- pending_cloud_sync

Este esquema permite detectar divergencias, gestionar sincronización y habilitar reintentos controlados.

Además, se implementa un **guardado forzado al cerrar la aplicación**, reduciendo el riesgo de pérdida de progreso.

11.3.4. Sistema de datos estáticos dirigido por backend con fallback local

Los datos estáticos del juego (items, habilidades, enemigos, etc.) Se distribuyen mediante endpoints /game-data, permitiendo una arquitectura data-driven.
El cliente utiliza:

- Indexación en memoria para acceso O(1)
- Caché local cifrada como mecanismo de recuperación ante fallo

Esto mejora la resiliencia del sistema en condiciones de red inestables o uso offline.

11.3.5. Sistema de habilidades con mapeo de tres niveles

Se introduce una arquitectura de mapeo entre:

1. Claves del backend (IDs o nombres)
2. Rutas de recursos (res_path)
3. Claves internas en tiempo de ejecución

El componente intermedio permite desacoplar datos remotos y recursos locales, soportando alias, internacionalización y evolución del sistema sin romper compatibilidad.

11.3.6. Sistema genético modular con restricciones consistentes

El sistema de genes se estructura en:

- Genes principales (main genes)
- Subgenes modulares con dependencias y costes progresivos

Se aplican restricciones en dos niveles:

- Límite por clase
- Límite global del sistema

Estas reglas están sincronizadas entre cliente y backend, evitando inconsistencias. Además, se introducen extensiones semánticas como modificadores dependientes del tipo de enemigo (vs_targets), permitiendo pasar de bonificaciones numéricas a reglas dinámicas.

11.3.7. Sistema de enemigos basado en etiquetas (tag-driven design)

Los enemigos se definen mediante plantillas que incluyen **etiquetas de combate (combat_tags).**
Esto permite:

- Implementar mecánicas de ventaja/desventaja sin lógica hardcodeada
- Facilitar la extensión de comportamientos mediante configuración

El modelo reduce la complejidad del código y mejora la escalabilidad del diseño de combate.

11.3.8. Modelo de seguridad de mínimos privilegios en la comunidad

El sistema de comunidad implementa un modelo de acceso basado en:

- Lectura anónima
- Escritura autenticada

Se presta especial atención al orden de reglas en la configuración de seguridad, evitando vulnerabilidades por coincidencias ambiguas.
Además, se define un formato de respuesta unificado (ApiResponse<T>), lo que simplifica el manejo de errores en el frontend.

11.3.9. La testabilidad como principio transversal del sistema

El sistema integra mecanismos de validación en múltiples capas:

- **FastAPI**: pruebas de contrato basadas en OpenAPI
- **Spring Boot**: pruebas de controladores con inyección de autenticación
- **Frontend/cliente**: combinación de pruebas unitarias y validación de escenarios API

Este enfoque reduce inconsistencias entre componentes, mejora la estabilidad del sistema y facilita la detección temprana de errores en integración.

11.4 Limitaciones

1. **Falta de optimización para alta concurrencia**:

   El sistema no ha sido sometido a optimizaciones como ajuste del pool de conexiones, estrategias de limitación de tasa ni pruebas sistemáticas de carga. En consecuencia, no se dispone de métricas cuantificables como QPS, latencia P99 o distribución del tiempo de fotogramas, lo que dificulta evaluar su rendimiento bajo condiciones de alta concurrencia.

2. **Ausencia de soporte para multijugador en tiempo real**:

   No se han implementado mecanismos como WebSocket, sincronización por frames, predicción de estado ni rollback. El módulo de multijugador se encuentra en una fase preliminar y no cumple con los requisitos de un entorno productivo.

3. **Complejidad en el uso de JSONB**:

   Aunque JSONB proporciona flexibilidad en el modelado de datos, las consultas complejas, agregaciones y generación de informes implican un alto coste computacional y requieren el uso de índices avanzados (como GIN o índices de expresión). Actualmente, el sistema se basa principalmente en operaciones de lectura/escritura por entidad completa, sin abordar escenarios de análisis a gran escala.

4. **Acoplamiento parcial entre interfaz y lógica**:

   La interfaz de usuario (menús, HUD) mantiene cierto nivel de acoplamiento con los módulos globales (Autoload y señales), sin adoptar patrones como MVVM o mecanismos de data binding, lo que puede afectar la mantenibilidad a medida que el sistema crece.

5. **Lógica autoritativa dependiente del cliente**:

   Las mecánicas clave, como el combate o la recolección de objetos, no son validadas por el servidor. Esto introduce riesgos en escenarios competitivos o en entornos donde la integridad del sistema es crítica.

6. **Fragmentación del sistema de identidad**:

   El sistema de juego y el sistema de comunidad utilizan mecanismos de autenticación independientes basados en JWT, sin integración mediante estándares como OIDC o identidad federada. Esto dificulta la unificación de la identidad del usuario entre distintos subsistemas.

7. **Deuda técnica y componentes provisionales**:

   Existen elementos incompletos o redundantes, como la definición de `SignalBus` sin uso efectivo, duplicación en la implementación de

`WeaponViewModel`, o la simplificación de atributos como los tiempos de recarga. Estos aspectos han sido identificados como decisiones temporales que requieren refactorización futura.

8. **Ausencia de una capa de caché**:

   El sistema no incorpora mecanismos de caché (como Redis) para optimizar el acceso a datos frecuentes, lo que puede generar una carga innecesaria sobre la base de datos en escenarios de uso intensivo.

9. **Falta de procesamiento asíncrono**:

   No se han implementado sistemas de colas de tareas ni procesamiento asíncrono para operaciones como envío de correos, gestión de logs o análisis de datos, lo que puede afectar el rendimiento del sistema bajo alta carga.

10. **Limitaciones en logging y monitorización**:

    El sistema carece de una infraestructura unificada de logging, trazabilidad de errores y monitorización del rendimiento (por ejemplo, mediante ELK o Prometheus), lo que dificulta la observabilidad y el mantenimiento en entornos productivos.

11. **Escalabilidad de la base de datos no validada**:

    Aunque se emplea una separación lógica mediante múltiples esquemas, no se han implementado estrategias como particionado, replicación o separación de lectura/escritura, y la capacidad de escalar el sistema no ha sido verificada en escenarios de gran volumen de datos.

12. **Ausencia de integración y despliegue continuo (CI/CD)**:

    El proceso de despliegue depende de configuraciones manuales y entornos locales, sin la incorporación de pipelines de integración y despliegue continuo (como GitHub Actions o Docker Compose), lo que limita la automatización y la reproducibilidad del sistema.

13. **Limitaciones del sistema de navegación de enemigos (atascos y oscilación de ruta)**

    En la implementación del sistema de navegación de los enemigos se identifican ciertas limitaciones, principalmente el “atasco en obstáculos” y la “oscilación de la ruta (path jittering)”. En entornos estrechos o complejos, los enemigos pueden cambiar de dirección de forma inestable debido a la corrección local de la ruta, o incluso intentar moverse repetidamente hacia objetivos no alcanzables en los límites del sistema de navegación.

    Esta problemática se origina principalmente en una lógica de navegación local simplificada y en un tratamiento limitado de obstáculos dinámicos y

casos de borde, lo que reduce la estabilidad del comportamiento de movimiento en ciertos escenarios.

14. **Limitaciones del motor de renderizado y elección tecnológica futura**

    Durante el desarrollo de este proyecto, se identificaron ciertas limitaciones relacionadas con el rendimiento de Godot en la renderización de escenas a gran escala y mapas abiertos. Estas limitaciones se hacen más evidentes al manejar terrenos extensos, modelos de alta densidad y sistemas de iluminación complejos, afectando principalmente la eficiencia de carga y la estabilidad del renderizado.

    En caso de que estas limitaciones no puedan resolverse adecuadamente mediante optimizaciones (como streaming de escenas, sistema LOD o segmentación de mundos), se considerará la migración del proyecto a Unreal Engine en futuras fases de desarrollo, aprovechando sus capacidades superiores en renderizado de grandes entornos y su pipeline gráfico de alto rendimiento.

11.5 Trabajo futuro y evolución del sistema

11.5.1 Proyecto Macro

Los siguientes elementos representan trayectorias de evolución específicas basadas en componentes y patrones arquitectónicos ampliamente utilizados en la industria, que optimizará según las necesidades de mi proyecto:

1. **Introducción de una capa de caché (Redis)**

Se recomienda integrar Redis u otros sistemas de caché en memoria para almacenar datos de acceso frecuente, como las rutas /game-data/* y los resultados de paginación del feed comunitario. Mediante TTL y mecanismos de invalidación activa, se puede reducir la carga de lectura sobre PostgreSQL y mejorar el rendimiento de respuesta.

2. **Comunicación en tiempo real (WebSocket / gRPC Stream)**

Sobre la arquitectura REST existente, se propone incorporar WebSocket o gRPC streaming para habilitar actualizaciones en tiempo real de publicaciones, comentarios y notificaciones, reduciendo la sobrecarga del polling.

En caso de implementar multijugador, se evaluará una arquitectura basada en servidores por salas, combinada con sincronización por entradas o modelos de sincronización de estado con predicción y rollback.

3. **Control de concurrencia en el sistema de guardado**

Para evitar conflictos de escritura concurrente, se introducirán mecanismos de control de versiones (version / ETag) o validación mediante timestamps (updated_at), junto con estrategias de reintento en el cliente.

4. **Optimización de la arquitectura interna (ECS / separación lógica-render)**

En escenarios con alta densidad de entidades, se evaluará la adopción del modelo ECS (Entity-Component-System) o la separación entre lógica de actualización y renderizado mediante interpolación, optimizando el rendimiento con herramientas de profiling.

5. **Almacenamiento de objetos y CDN**

El sistema de imágenes evolucionará hacia un modelo basado en subida mediante URLs prefirmadas y almacenamiento en servicios de objetos, manteniendo únicamente referencias en la base de datos (post_images) para reducir la carga del backend.

6. **CI/CD y aseguramiento de calidad**

Se implementará una pipeline de integración y despliegue continuo que incluirá:

Pruebas automatizadas con bases de datos Dockerizadas (pytest)
Control de cobertura de código (coverage.py / JaCoCo)
Pruebas de carga reproducibles con k6 o Locust

Estos resultados podrán utilizarse como evidencia de evaluación del sistema.

7. **Unificación de identidad y observabilidad**

Se adoptará OAuth2 / OIDC para unificar la identidad de usuario entre sistemas, junto con OpenTelemetry o logging estructurado para mejorar la observabilidad.

Se unificarán identificadores como request_id (FastAPI), MDC (Spring Boot) y códigos de error en el cliente, permitiendo trazabilidad completa del sistema.

11.5.2 Orientado a juego

El desarrollo futuro del sistema de juego se orienta a mejorar la jugabilidad, la coherencia del mundo, el rendimiento y la expresividad visual, transformando el prototipo actual en una experiencia más completa y robusta.

1. **Mejora de la física y control del personaje**

Se plantea optimizar el modelo de gravedad y movimiento, ajustando parámetros físicos y la respuesta del controlador para lograr una experiencia más natural y precisa.
Asimismo, se prevé refinar la máquina de estados del personaje, incorporando nuevas transiciones y animaciones, con el objetivo de enriquecer la expresividad y fluidez del control.

2. **Sistema narrativo y de misiones**

Se introducirá un sistema narrativo que permita estructurar el contenido del juego mediante:

- **Historia principal**
- **Sistema de misiones**
- **Eventos guiados por narrativa**

El sistema de tutorial evolucionará hacia un enfoque basado en narrativa, permitiendo que el jugador aprenda mecánicas a través de la historia.

3. **Sistema de interacción y diálogos**

Se ampliará el sistema de diálogo para soportar:

- Conversaciones estructuradas
- Posibles ramificaciones
- Integración con misiones y eventos

Esto permitirá una mayor profundidad en la interacción con NPCs.

4. **Mejora visual y de interfaz**

Se optimizarán los siguientes elementos visuales:

- Modelos de armas y su representación
- Sistema de tercera persona (cámara y animación)
- Interfaz de combate (HUD)
- Panel de personaje (estadísticas, equipo, progresión)

Además, se mejorarán elementos artísticos como iconos, fondos y coherencia visual general.

5. **Evolución del sistema de progresión**

Se ampliarán los sistemas existentes para mejorar la profundidad del juego:

- Mejora del sistema genético, ampliando los efectos sobre atributos y mecánicas
- Introducción de un sistema de moneda
- Implementación de un sistema de logros (achievements)

Estos elementos permitirán una progresión más rica y motivadora.

6. **Gestión del mundo y rendimiento**

Para soportar entornos más complejos, se implementarán:

- Carga por bloques del mapa (chunk loading)
- Expansión del mapa del juego
- Optimización del uso de recursos en tiempo de ejecución

Esto permitirá escalar el mundo del juego sin comprometer el rendimiento.

7. **Coherencia global del sistema**

El objetivo final es integrar todos estos elementos en un sistema coherente donde:

- Narrativa, misiones y mecánicas estén conectadas

- La progresión del jugador tenga impacto tangible
- La experiencia sea consistente tanto a nivel visual como funcional

12. Conclusión

Este proyecto no se concibe como un programa de videojuego de un solo módulo, sino como un prototipo de aplicación distribuida compuesta por varios subsistemas. El sistema general está formado por un cliente en Godot, un servicio de juego en FastAPI, un servicio de comunidad en Spring Boot y una aplicación frontend para la comunidad. Todos ellos se apoyan en una base de datos PostgreSQL común para el almacenamiento y la gestión de datos. Cada subsistema se comunica mediante interfaces HTTP y contratos de datos, lo que permite un desacoplamiento lógico, pero mantiene la coherencia a nivel de información, dando lugar a una arquitectura de “multi-cliente y multi-servicio”.

En la primera versión del proyecto, el trabajo se centró principalmente en la construcción del framework básico del cliente en Godot, incluyendo la estructura de escenas, el sistema de entrada y un esbozo de la lógica de combate y del sistema de personaje. Esta fase tenía como objetivo validar la estructura principal del gameplay, por lo que todavía no se había incorporado un backend independiente ni un sistema completo de persistencia de datos.

En la versión actual, el proyecto se amplía de forma significativa al introducir una arquitectura completa de servicios y una capa de datos más formalizada. FastAPI se utiliza para implementar la lógica principal del sistema de juego y la gestión de guardado de personajes, mientras que Spring Boot se encarga de la interacción social y el sistema de cuentas. Ambos servicios tienen responsabilidades bien definidas y funcionan de forma independiente, aunque comparten un mismo sistema de identidad. El cliente se conecta a estos servicios mediante APIs, lo que permite evolucionar desde una lógica local hacia una arquitectura basada en servicios.

En el nivel de datos, PostgreSQL actúa como núcleo central del sistema. Se organiza mediante esquemas (schemas) para separar dominios funcionales y se combinan estructuras relacionales con campos JSONB para manejar datos más dinámicos. Los datos de progreso del juego, el contenido social y la información de autenticación se gestionan en dominios distintos, pero todos comparten una identidad unificada, lo que garantiza la coherencia global del sistema.

En el lado del cliente, Godot no solo se encarga del renderizado y la interacción, sino también de la sincronización del estado en tiempo de ejecución. El sistema combina almacenamiento local en caché con sincronización en la nube para asegurar la recuperabilidad de los datos. Además, se utiliza un enfoque basado en datos estáticos y carga dinámica para reducir el acoplamiento y mejorar la extensibilidad del sistema. En esta versión del proyecto también se han añadido múltiples sistemas y mecánicas nuevas.

En el diseño del backend, FastAPI se centra en la lógica del juego y la consistencia del guardado, utilizando contratos de datos definidos mediante DTOs. Spring Boot, por su parte, se enfoca en las funcionalidades sociales y el control de permisos,

utilizando JWT para autenticación sin estado y una arquitectura en capas para facilitar la escalabilidad. Ambos forman parte del sistema backend, pero mantienen límites claros entre dominios.

En cuanto al enfoque de ingeniería, se introducen mecanismos de estandarización de interfaces y centralización de rutas API, lo que reduce inconsistencias entre clientes y servidores. Además, se incluyen pruebas básicas y validaciones de contrato para disminuir riesgos durante la refactorización, mejorando así la mantenibilidad del sistema.

Sin embargo, durante el desarrollo también se ha hecho un uso intensivo de herramientas de inteligencia artificial. En este proyecto, la IA se ha utilizado principalmente para generar código base, aunque debido a que el resultado no siempre es estable o alineado con la intención del sistema, gran parte del código ha sido posteriormente revisado y ajustado manualmente. También se han generado algunos recursos gráficos mediante IA para pruebas iniciales, como texturas temporales.

Finalmente, desde el punto de vista del progreso general, el desarrollo actual se considera satisfactorio, ya que se han construido las bases estructurales de varios subsistemas importantes. Al tratarse de un proyecto individual, el ritmo de desarrollo no es uniforme, pero en futuras iteraciones se espera seguir ampliando la arquitectura, incorporando mejores soluciones técnicas y completando el sistema de juego en su totalidad.

En conjunto, este trabajo representa una evolución estructural sobre la base del cliente en Godot, pasando de un prototipo centrado únicamente en el juego a una arquitectura distribuida con múltiples servicios. El objetivo principal no ha sido aumentar la complejidad de un único módulo, sino establecer límites claros entre sistemas, definir rutas de datos coherentes y garantizar mecanismos de consistencia entre plataformas, sentando así una base sólida para futuras extensiones y mantenimiento a largo plazo.

13. Bibliografía y referencias

Blender:

https://www.bilibili.com/video/BV1w9T4z2EuY/?spm_id_from=333.788.videopod.sections&vd_source=0f1a51d177f4d011ed777e0e0601b2a1&p=5
https://www.bilibili.com/video/BV1JEivB9EV9/?spm_id_from=333.337.search-card.all.click

Godot:

https://docs.godotengine.org/en/stable/index.html
https://docs.godotengine.org/en/stable/tutorials/scripting/scene_tree.html
https://docs.godotengine.org/en/stable/classes/class_httprequest.html
https://docs.godotengine.org/en/stable/classes/class_fileaccess.html
https://docs.godotengine.org/en/stable/tutorials/physics/physics_introduction.html

https://www.bilibili.com/video/BV1bv4bedEdM/?spm_id_from=333.788.videopod.sections
https://www.bilibili.com/video/BV1Hm41167mP?spm_id_from=333.788.videopod.episodes&p=9
https://www.youtube.com/@Le_x_Lu

Python:

https://fastapi.tiangolo.com/
https://uvicorn.dev/
https://docs.sqlalchemy.org/en/20/
https://www.psycopg.org/docs/
https://passlib.readthedocs.io/en/stable/
https://github.com/Kludex/python-multipart
https://github.com/JoshData/python-email-validator
https://python-jose.readthedocs.io/en/latest/
https://github.com/theskumar/python-dotenv
https://pydantic.dev/docs/validation/latest/get-started/
https://docs.pytest.org/en/stable/
https://www.python-httpx.org/
https://docs.python.org/3/library/index.html
https://fastapi.tiangolo.com/tutorial/cors/
https://fastapi.tiangolo.com/tutorial/dependencies/
https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/
https://starlette.dev/

Java:

https://docs.spring.io/spring-framework/reference/
https://docs.spring.io/spring-framework/reference/web/webmvc.html
https://docs.spring.io/spring-security/reference/
https://docs.spring.io/spring-data/jpa/reference/
https://jakarta.ee/specifications/bean-validation/
https://hibernate.org/validator/documentation/
https://jdbc.postgresql.org/
https://www.h2database.com/html/main.html
https://github.com/jwtk/jjwt
https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/argon2/Argon2PasswordEncoder.html
https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/filter/OncePerRequestFilter.html
https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api/2.3.1
https://mvnrepository.com/artifact/org.glassfish.jaxb/jaxb-runtime/2.3.6
https://projectlombok.org/
https://docs.spring.io/spring-boot/reference/testing/
https://docs.junit.org/6.0.3/overview.html
https://maven.apache.org/tools/wrapper/
https://maven.apache.org/plugins/maven-compiler-plugin/
https://maven.apache.org/surefire/maven-surefire-plugin/
https://maven.apache.org/guides/

Postgres:

https://www.postgresql.org/
https://www.postgresql.org/docs/current/indexes-intro.html
https://www.postgresql.org/docs/current/datatype-uuid.html
https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/current/datatype-net-types.html
https://www.postgresql.org/docs/current/datatype-datetime.html
https://www.postgresql.org/docs/current/ddl-schemas.html
https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-CHECK-CONSTRAINTS
https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-FK
https://www.postgresql.org/docs/current/ddl-constraints.html
https://www.postgresql.org/docs/current/pgtrgm.html
https://www.postgresql.org/docs/current/datatype-enum.html
https://www.postgresql.org/docs/current/datatype-numeric.html#DATATYPE-SERIAL
https://www.postgresql.org/docs/current/datatype-numeric.html

Otros:

https://desahuman.wuaze.com/
https://github.com/danielgatis/rembg

14. Anexos

[1] Narrador del inicio

https://docs.google.com/document/d/15sjgRcXTrBouwsKlPURlaPrf9v6ZKSjyvdpa74LF7hg/edit?pli=1&tab=t.xvug2xl8c4on

[2] Escena de carga

https://drive.google.com/file/d/1XzHVWyLY9nk1UJqwAsMOv_5DJIHp2So-/view?usp=drive_link

[3] Nuestro web

https://desahuman.wuaze.com o http://desahuman.wuaze.com

[4] Lista de abreviaturas y glosario de términos

https://docs.google.com/document/d/15sjgRcXTrBouwsKlPURlaPrf9v6ZKSjyvdpa74LF7hg/edit?pli=1&tab=t.gzi7q4eshud2#heading=h.zd2cdwk4olb6

[5] Seguiment Setmanal

https://docs.google.com/spreadsheets/d/18qmzZbEwSGx6b4wRDTXEIIYGoAb1dM9MOgCMcq-22mo/edit?pli=1&gid=0#gid=0

[6] Análisis

https://docs.google.com/spreadsheets/d/1tWEucjCFIZSGWIGDX7r1M2OWnU7KVmkJBZ_cJMSKdMc/edit?gid=0#gid=0

[7] Diagrama

https://drive.google.com/drive/folders/1bLTbfx0Dqw9LuP8Sr4i8qIdEP47J2csG?usp=sharing

Lista de abreviaturas y glosario de términos

Las abreviaturas se presentan en orden alfabético latino.
Los términos corresponden a conceptos frecuentes dentro del proyecto, con el objetivo de facilitar la lectura a evaluadores sin experiencia previa en motores de videojuegos.

Abreviaturas

Abreviatura	Nombre completo / origen	Explicación
3D	Three-Dimensional	Espacio y renderizado tridimensional
ADS	Aim Down Sights	Apuntado por mira (mencionado en algunos diseños y TODOs del proyecto)
API	Application Programming Interface	Interfaz de programación de aplicaciones; en este proyecto principalmente APIs HTTP REST
Argon2	Argon2 password hashing	Familia de algoritmos de hash para contraseñas; utilizado mediante passlib
ASGI	Asynchronous Server Gateway Interface	Interfaz asíncrona de gateway para Python; utilizada por FastAPI y uvicorn
CDN	Content Delivery Network	Red de distribución de contenido; utilizada para recursos estáticos y multimedia
CI	Continuous Integration	Integración continua; ejecución automática de pruebas y verificaciones
CORS	Cross-Origin Resource Sharing	Política de acceso entre distintos orígenes en navegadores
CRUD	Create, Read, Update, Delete	Operaciones básicas de creación, lectura, actualización y eliminación
DDL	Data Definition Language	Lenguaje de definición de datos; SQL relacionado con tablas y estructuras
DTO	Data Transfer Object	Objeto de transferencia de datos; utilizado en Pydantic y DTO de Java

E2E	End-to-End	Pruebas end-to-end que cubren el flujo completo
ECS	Entity–Component–System	Arquitectura Entidad–Componente–Sistema utilizada en videojuegos
ETag	Entity Tag	Identificador de versión HTTP para validación de caché y bloqueo optimista
ER	Entity–Relationship	Modelo entidad–relación de bases de datos
FPS	First-Person Shooter	Shooter en primera persona
GDD	Game Design Document	Documento de diseño del videojuego
GDScript	Godot Script	Lenguaje de scripting integrado de Godot
GIN	Generalized Inverted Index	Tipo de índice de PostgreSQL utilizado en JSONB y arrays
H2	H2 Database Engine	Base de datos en memoria utilizada en pruebas de Spring
HTTP(S)	Hypertext Transfer Protocol (Secure)	Protocolo de transferencia de hipertexto
IDE	Integrated Development Environment	Entorno de desarrollo integrado, como IntelliJ IDEA o VS Code
IoC	Inversion of Control	Inversión de control; principio utilizado por Spring
JSON	JavaScript Object Notation	Formato de intercambio de datos
JSONB	JSON Binary	Tipo JSON binario de PostgreSQL con soporte de índices
JPA	Java Persistence API	API de persistencia utilizada en Spring Data JPA
JWT	JSON Web Token	Token compacto utilizado para autenticación sin estado

LAN	Local Area Network	Red de área local
LOD	Level of Detail	Nivel de detalle para optimización gráfica
MDC	Mapped Diagnostic Context	Contexto de diagnóstico para logs estructurados
MVC	Model–View–Controller	Patrón arquitectónico Modelo–Vista–Controlador
NAT	Network Address Translation	Traducción de direcciones de red
NPC	Non-Player Character	Personaje no jugable
ORM	Object–Relational Mapping	Mapeo objeto-relacional; utilizado por SQLAlchemy y JPA
P99	99th Percentile	Métrica de rendimiento basada en percentiles
POJO	Plain Old Java Object	Objeto Java simple
REST	Representational State Transfer	Estilo arquitectónico para APIs
RFC	Request for Comments	Sistema de documentos estándar de Internet
SMTP	Simple Mail Transfer Protocol	Protocolo de envío de correo electrónico
SQL	Structured Query Language	Lenguaje estructurado de consultas
SSO	Single Sign-On	Inicio de sesión único
TTL	Time To Live	Tiempo de vida de caché
UA	User Agent	Cadena identificadora del cliente
UI	User Interface	Interfaz de usuario
UUID	Universally Unique Identifier	Identificador único universal

VM	Virtual Machine	Máquina virtual

Nota:
El token utilizado en el encabezado HTTP Authorization: Bearer <token> corresponde a un JWT.

Escena 1: Escuchando en las sombras

Lugar: Nave científica alienígena · Laboratorio genético

El laboratorio brillaba con una luz fría y artificial.
Las paredes metálicas estaban cubiertas de conductos luminosos y pantallas de datos que parpadeaban sin descanso. En el aire flotaba un leve olor a desinfectante.

Un asistente de laboratorio observaba un panel holográfico suspendido sobre sus manos. En él giraban cadenas de ADN humano que se separaban y recomponían constantemente.

—Doctor… —dijo con entusiasmo contenido—. La capacidad de modificación genética de estos bípedos es increíble. Son material experimental perfecto.

En la pantalla, el modelo genético de un ser humano era fragmentado y reorganizado una y otra vez.

El doctor permanecía de pie en el centro de la sala, con las manos cruzadas detrás de la espalda. Su voz era grave, tranquila.

—Debemos acelerar la investigación. El consejo central vendrá pronto a inspeccionar el proyecto.
Si tenemos éxito… nuestra civilización encontrará un nuevo camino evolutivo.

La cámara se alejaba lentamente.

Al fondo del laboratorio, varias camas experimentales se alineaban en perfecto orden. Sobre ellas descansaban cuerpos humanos inconscientes, conectados a tubos transparentes y dispositivos de monitorización. Un líquido azul pálido recorría lentamente las conducciones.

Frente al doctor se alzaba un enorme tanque cilíndrico.

Dentro flotaba un hombre sumergido en un líquido amarillento. Varias conexiones neuronales penetraban directamente en su columna vertebral.

El doctor lo observaba en silencio.

A un lado, sobre una mesa metálica, reposaba un recipiente de cristal más pequeño.

En su interior nadaba una criatura extraña, una especie de pez alienígena de aspecto deformado.

El asistente desvió la mirada hacia él.

—Doctor… si combináramos la capacidad adaptativa de esta especie con el ADN humano…

El doctor no respondió.
Solo permaneció inmóvil, pensativo.

Entonces—

La cámara descendió bruscamente, atravesando la estructura del laboratorio.

Debajo del piso principal se extendía el nivel técnico de la nave.
La iluminación era tenue. Grandes tuberías de refrigeración y conductos de aire cruzaban el espacio mientras las máquinas emitían un zumbido grave y constante.

Junto a una rejilla de ventilación, un joven permanecía inmóvil, conteniendo la respiración.

Era el protagonista.

Vestía un traje de combate parcialmente destruido. La armadura estaba cubierta de arañazos y quemaduras; la insignia del hombro apenas podía distinguirse.

El casco colgaba de su cintura.

Una pequeña luz roja parpadeaba débilmente en el panel de su traje.

El protagonista se acercó lentamente a la rejilla de ventilación y observó el laboratorio a través de las ranuras metálicas.

Cuando vio a los humanos utilizados como experimentos, sus pupilas se contrajeron ligeramente.

Su mano se cerró lentamente en un puño.

—Primero tengo que entender la estructura de este lugar… —murmuró en voz baja—. Luego enviaré toda la información.

A lo lejos, una luz roja de emergencia se encendió repentinamente.

Bip…
Bip…

Pesados pasos metálicos resonaron desde la oscuridad del corredor.

El protagonista reaccionó de inmediato y colocó el casco sobre su cabeza.

El sistema del traje se activó.

El HUD iluminó el interior del visor.

«Se aproxima una señal de vida.»

El protagonista contuvo la respiración.

La imagen se oscureció lentamente.

Escena 2: Ruta de infiltración

Lugar: Nave científica alienígena · Nivel técnico

La oscuridad desapareció poco a poco.

El HUD del casco volvió a iluminarse.

De repente, una transmisión se abrió en el canal de comunicación.

Una voz femenina sonó en sus oídos, ligera y burlona, aunque hablaba en voz baja.

—¿Hola? ¿Me recibes?
No me digas que ya terminaste flotando en uno de esos tanques raros.

El protagonista se apoyó contra una tubería y respondió en susurros.

—Sigo vivo.
Pero realmente están experimentando con humanos.

Al otro lado hubo un breve silencio.

Después, ella soltó un pequeño suspiro.

—…Ya sabía que estos alienígenas escondían algo turbio.
Pero ahora mismo no juegues al héroe.

Una línea verde apareció sobre el HUD.

—A unos veinte metros a tu derecha viene una patrulla.
Te marqué una ruta segura.

El mapa táctico se actualizó automáticamente.

—Confía en mí.
He hackeado las cámaras, los escáneres térmicos y hasta los detectores biométricos de la nave.

El protagonista se incorporó lentamente.

A lo lejos, una patrulla alienígena cruzó un pasillo metálico. Sus pasos resonaban por toda la estructura.

> Objetivo actualizado
> Dirígete al punto de encuentro

El protagonista avanzó por estrechos corredores llenos de conductos y maquinaria industrial.

Las señales del HUD parpadeaban constantemente.

—Gira a la derecha —indicó ella—.
Hay un conducto de mantenimiento.

Frente a él apareció una rejilla de ventilación.

El protagonista se agachó y comenzó a desmontarla.

—Sí, por ahí.
Métete dentro. Créeme… es mucho más elegante que entrar disparando por la puerta principal.

El protagonista avanzó arrastrándose por el conducto.

El viento silbaba débilmente entre las paredes metálicas.

A través de las rejillas inferiores podían verse patrullas moviéndose bajo luces rojas intermitentes.

—Sigue avanzando —dijo ella entre risas—.
La salida está cerca.
Y si te quedas atascado, avísame. Quiero sacar una foto.

El protagonista empujó la tapa del conducto y saltó al suelo.

Almacén

La habitación estaba iluminada apenas por luces amarillentas.

Cajas y suministros ocupaban casi todo el espacio.

En una esquina, una mujer desmontaba tranquilamente un arma sentada sobre un contenedor militar.

Al escuchar el ruido, levantó el rifle de inmediato.

Pero al reconocerlo, sonrió.

—Vaya…
Saliste vivo.

Se levantó y le lanzó un arma.

El protagonista la atrapó en el aire.

—Toma.
No pienso dejarte discutir con alienígenas usando solo los puños.

Ella se puso de pie y golpeó suavemente la armadura de su hombro.

Por un instante, su expresión se volvió seria.

—Pero hablando en serio…
Si lo que viste ahí arriba es real…

Miró hacia la puerta metálica del almacén.

—Entonces esta nave esconde algo mucho peor de lo que decía la misión.

El HUD volvió a iluminarse.

> Objetivo actualizado
> Escapa de la zona de investigación junto a tu compañera

La pantalla se fundió lentamente en negro.

El juego comenzaba.