TecnoCampus
Escola Superior
Politècnica

Centre adscrit a la
upf. Universitat Pompeu Fabra Barcelona

Grado en Ingeniería Informática de Gestión y Sistemas de Información

Creación de un 4X por turnos simultáneos para móvil

Memoria

Roger Perales Mares

TUTOR: Dr. David Ródenas Picó

2021-2022

TecnoCampus
Mataró-Maresme

Abstract

Strategy videogames have been pushed into a niche over the years. The 4X subgenre (Explore, Expand, Exploit and Exterminate) is formed of particularly complex games that require a large investment of time to be enjoyed. The objective of this project is to develop a casual 4X game prototype, for mobile platforms, with asynchronous turns. In this way, players can take turns without the need to invest a lot of time or effort.

Resumen

Los videojuegos del género de la estrategia han sido desplazados a un nicho a lo largo de los años. El subgénero de los 4X (E**x**plore, E**x**pand, E**x**ploit and E**x**terminate) son juegos especialmente complejos que requieren de una gran inversión de tiempo para poder ser disfrutados. El objetivo de este proyecto es desarrollar un prototipo de juego 4X casualizado, para plataformas móviles, con turnos asíncronos, de manera que sus jugadores puedan realizar sus turnos sin la necesidad de invertir mucho tiempo ni esfuerzo.

Resum

Els videojocs del gènere de l'estratègia han estat desplaçats a un nínxol al llarg dels anys. El subgènere dels 4X (Explore, Expand, Exploit and Exterminate) són jocs especialment complexos que requereixen una gran inversió de temps per poder ser gaudits. L'objectiu d'aquest projecte és desenvolupar un prototip de joc 4X casualitzat, per a plataformes mòbils, amb torns asíncrons, de manera que els seus jugadors puguin fer els torns sense la necessitat d'invertir molt de temps ni esforç.

Índice

Abstract I

Resumen I

Resum I

1. Introducción 1

2. Marco Teórico 3

2.1. Modelo de Comunicación vía Red 3

2.1.1. Peer-to-Peer 3

2.1.2. Client as a Host 4

2.1.3. Client-Server 5

2.2. Framework del Servidor 6

2.2.1. Java con Spring Boot 6

2.2.2. JavaScript con NodeJS 6

2.2.3. Ruby con Ruby on Rails 7

2.2.4. Python con Django 7

2.2.5. C# con .NET 7

2.2.6. PHP con Laravel 7

2.2.6. C++ con Boost 8

2.2. Game Engine 9

3. Objetivos y Abasto 11

4. Análisis de Referentes 13

5. Metodología.. 15

6. Definición de Requerimientos Funcionales y Tecnológicos 17

7. Iteraciones del Modelo Espiral.. 19

7.1. Primer Ciclo.. 19

7.2. Segundo Ciclo... 19

7.3. Tercer Ciclo 20

7.4. Cuarto Ciclo.. 21

7.5. Quinto Ciclo 21

8. Desarrollo 23

8.1. Game Design Document... 23

8.2. Lenguaje del Servidor... 25

8.2.1. El Entorno de Conexión del Servidor 25

8.2.2. La Arquitectura del Backend... 27

8.2.3. TypeScript 29

8.3. Base de Datos 30

8.3.1. Elección de la Base de Datos... 30

8.3.2. Arquitectura de la Base de Datos 31

8.4. Sistema de Autenticación 33

8.5. Generación de Partidas 37

8.6. Cargar Estados de Partidas 40

8.6.1. Cargar Partidas no Empezadas 41

8.6.2. Cargar Partidas Empezadas 42

8.7. Generación del Mapa Estelar 45

8.7.1. Algoritmo de Generación Procedural 46

8.7.2. Representación Visual del Mapa Estelar 51

8.8. Generación del Mapa Planetario 52

8.9. Turnos de Juego y Reglas de Juego 54

8.10. Tecnologías y Edificios 56

8.11. Las Estrellas visualmente 57

9. Conclusiones 59

10. Posibles Ampliaciones 61

11. Bibliografía 63

Índice de Figuras

Figura 1. Diagrama Peer-to-Peer. Fuente: Elaboración propia. 3

Figura 2. Diagrama Client as a Host. Fuente: Elaboración propia. 4

Figura 3. Diagrama Client-Server. Fuente: Elaboración propia. 5

Figura 4. Modelo en espiral. Fuente: [8] 16

Figura 5. Diagrama de la arquitectura del proyecto. Fuente: Elaboración propia. 25

Figura 6. Diseño de la Base de Datos. Fuente: Elaboración propia. 31

Figura 7. Encriptación de la contraseña usando Argon2. Fuente: Elaboración propia. 34

Figura 8. Pantalla de inicio de sesión. Fuente: Elaboración propia. 35

Figura 9. Pantalla de creación de cuenta. Fuente: Elaboración propia. 35

Figura 10. Menú de creación de partidas. Fuente: Elaboración propia. 38

Figura 11. Menú de unión a partida existente. Fuente: Elaboración propia. 38

Figura 12. Generación de un código aleatorio. Fuente: Elaboración propia. 39

Figura 13. Menú de partidas en curso. Fuente: Elaboración propia 40

Figura 14. Menú de partida en espera. Fuente: Elaboración propia. 41

Figura 15. Petición a la Base de Datos de los datos de una partida. Fuente: Elaboración propia. 43

Figura 16. Tiempo de espera al cargar partida a lo largo de las iteraciones. Fuente: Elaboración propia. 44

Figura 17. Métodos estáticos de Vector2. Fuente: Elaboración propia. 45

Figura 18. Ruido Simplex. Fuente: Elaboración propia. 46

Figura 19. Ruido generando filamentos. Fuente: Elaboración propia................................ 47

Figura 20. Ruido procesado. Fuente: Elaboración propia... 48

Figura 21. Estrellas colocadas en el mapa de ruido. Fuente: Elaboración propia.............. 49

Figura 22. Estrellas orgánicamente colocadas en el mapa de ruido. Fuente: Elaboración propia.. 50

Figura 23. Representación visual del mapa estelar. Fuente: Elaboración propia............... 51

Figura 24. Captura del sistema planetario. Fuente: Elaboración propia. 53

Figura 25. Carta de planeta. Fuente: Elaboración propia.. 53

Figura 26. ActionInterface. Fuente: Elaboración propia... 55

Figura 27. El Shader de la estrella en Shader Graph. Fuente: Elaboración propia. 57

Figura 28. Shader de la estrella. Fuente: Elaboración propia... 58

1. Introducción

Este proyecto pretende desarrollar el prototipo de un videojuego móvil multijugador de género 4X con turnos simultáneos, llevando a la modernidad un género que ya hace tiempo que se popularizó en los años previos a internet y que cada vez tiene menos público.

El objetivo de este trabajo es desarrollar un juego para dispositivos Android con estructura de cliente-servidor: Una aplicación móvil que se conecta a un servidor remoto, donde se procesan los datos de los distintos usuarios, y se devuelven en forma de estrellas y planetas en los que jugar.

El servidor tiene una forma de administrar usuarios y partidas en curso, así como de enviar y recibir información a los clientes. Cada usuario puede disponer de varias partidas en curso, y cada partida puede tener más de un usuario.

Todos los usuarios pueden ver el estado actual de cada una de sus partidas, y tienen la oportunidad de jugar en aquellas partidas en las que aún no hayan realizado su turno. Una vez que todos los usuarios de una partida han ejecutado su turno, se pasa a la siguiente ronda y todos los usuarios disponen de la posibilidad de volver a jugar.

El juego, por su parte, es una implementación del género 4X, hecho de manera que los turnos sean simultáneos: todos los jugadores pueden, o no, jugar a la vez.

2. Marco Teórico

2.1. Modelo de Comunicación vía Red

A la hora de ofrecer un servicio multijugador en un juego, hay varios acercamientos posibles respecto a la conectividad entre usuarios:

2.1.1. Peer-to-Peer

La intencionalidad del modelo Peer-to-Peer es la de transmitir datos directamente de cliente a cliente, donde el equipo que requiere de comunicación hace la petición directamente al equipo que tiene la información deseada [1]. Puesto que este modelo no tiene un nodo central, requiere que cada usuario realice sus propios procesos, aumentando la complejidad de la lógica, así como los riesgos de desincronización, mientras que compromete la seguridad y la integridad de los datos. Definitivamente, Peer-to-peer no es el modelo más óptimo para realizar este proyecto.

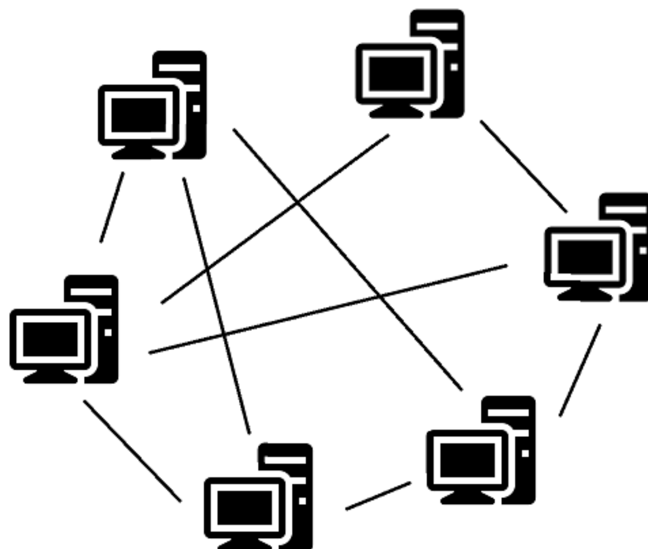

Figura 1. Diagrama Peer-to-Peer. Fuente: Elaboración propia.

2.1.2. Client as a Host

Este modelo usa a uno de los clientes involucrados en la red (uno de los usuarios de la partida) como intermediario entre el resto de clientes. Este 'host' se encarga de procesar todas las peticiones que el resto de clientes realizan, así como toda la información que le llega del resto de equipos. Dado que el juego desarrollado para este proyecto no tendrá como requisito que todos los usuarios estén conectados a la vez, hacer funcionar este modelo resulta muy enrevesado, cuando hay opciones objetivamente más sencillas y eficientes. La gran ventaja que tiene este modelo respecto al siguiente, Cliente-Servidor, es el ahorro de presupuesto por la parte del servidor, pues este modelo no requiere de uno, pero a su vez, es poco escalable y muy limitado en varios aspectos.

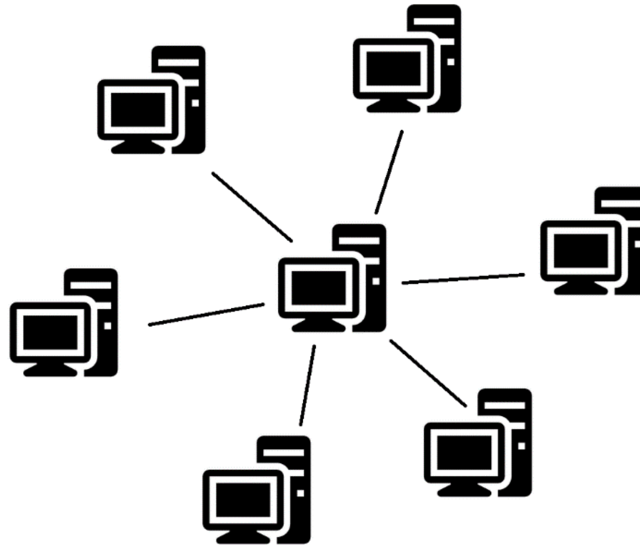

Figura 2. Diagrama Client as a Host. Fuente: Elaboración propia.

2.1.3. Client-Server

En el modelo cliente-servidor, se utiliza una lógica parecida al modelo de Client as a Host, donde la principal diferencia es que el encargado del intercambio de información, así como de realizar los procesos lógicos requeridos, es un servidor remoto en vez de un cliente elegido previamente [2]. Este servidor está disponible en todo momento, por lo que todos los clientes tienen acceso rápido a él.

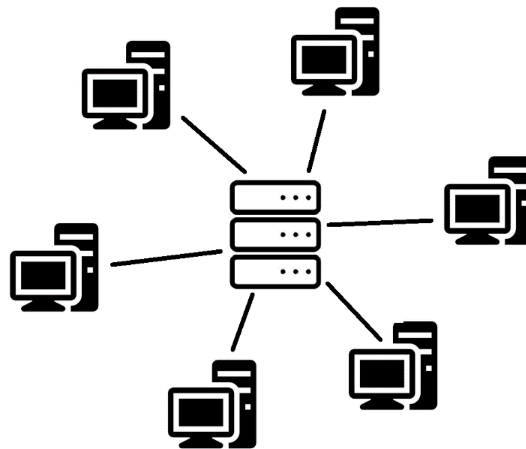

Figura 3. Diagrama Client-Server. Fuente: Elaboración propia.

2.2. Framework del Servidor

Es importante tener en cuenta las fortalezas y debilidades de todos los lenguajes y frameworks disponibles para desarrollo móvil con la finalidad de escoger el que mejor se adapta al proyecto en cuestión.

A continuación, se expone un breve resumen del análisis hecho con algunos de los frameworks de desarrollo web más populares.

2.2.1. Java con Spring Boot

Java, soporta multi-threading, lo que le permite utilizar todo el potencial de las unidades de procesamiento del servidor, si es que este tiene varios núcleos. Además, le permite efectuar los procesados complejos que requerirían análisis de datos, de imágenes, etc. Todo, sin perder eficiencia en la comunicación con los usuarios.

Es un framework pesado, con mucha capacidad de escalabilidad que facilita la comunicación con clientes y bases de datos, o el trabajo con documentos XML.

2.2.2. JavaScript con NodeJS

Java Script ofrece mucha facilidad de prototipado, gracias a su velocidad de compilado, que ayuda al desarrollo ágil y acelera todo el proceso de desarrollo del backend. El trabajo con documentos JSON está muy optimizado y puede ayudar mucho a ejecutar una buena arquitectura en conjunción con una Base de Datos basada en documentos. Además, al ser uno de los lenguajes de programación más usados alrededor del mundo, es fácil encontrar frameworks, información y documentación sobre cualquier caso de uso. La baja utilización de memoria y el alto rendimiento evitan bloqueos entre conexiones, ya sea con clientes o con la base de datos. Todo esto baja los costes del Servidor, consiguiendo mejor eficiencia. En conclusión, Node.js es ligero y potente.

2.2.3. Ruby con Ruby on Rails

Ruby on Rails es un framework usado para desarrollar backend para aplicaciones web. Su principal virtud es el prototipado, puesto que su entorno de trabajo está pensado para trabajar con tal objetivo, utilizando el Modelo Vista Controlador (MVC), y, por lo tanto, rápidamente se puede obtener una aplicación web convencional funcionando.

2.2.4. Python con Django

Django, de manera similar a Ruby on Rails, es un framework conceptualizado para el desarrollo rápido, por lo que también implementa un modelo estándar, el Modelo Vista Template (MVT), por lo que la velocidad de desarrollo que otorga se ve mitigada por la inflexibilidad del propio framework.

2.2.5. C# con .NET

.NET es un framework sencillo de usar, con una capacidad de procesamiento muy alta, y unos subsistemas capaces de integrarse a muchos módulos distintos. Este framework disfruta de mucha sencillez para trabajar con una gran cantidad de datos, para manipular documentos XML, o para controlar grandes accesos a las bases de Datos.

2.2.6. PHP con Laravel

Laravel, similar a Ruby on Rails o a Django, está construido sobre la suposición de que el proyecto se va a desarrollar usando un modelo en concreto, en este caso, el modelo vista controlador. El motor de ‘templates’ promueve el desarrollo por capas, de manera que los distintos módulos no se pisen entre ellos. Este framework utiliza componentes de otros frameworks, y no puede funcionar como Standalone. Además, la velocidad de

procesamiento de las peticiones es sorprendentemente lenta, comparándola a otros frameworks y lenguajes.

2.2.7. C++ con Boost

C++ es un lenguaje muy verboso, con un tiempo de desarrollo promedio muy largo. En contraposición a esto, es el lenguaje más rápido, y si se trabaja con cuidado y conocimiento, también el más personalizable. Boost son una serie de librerías que ayudan a realizar algunas tareas especialmente complejas, de manera más simple y eficaz, pero no ofrece un entorno de trabajo full-stack como la mayoría de los demás frameworks.

2.3. Game Engine

A la hora de desarrollar un juego, es cierto que no hay una metodología única a seguir, pero también es cierto que es una práctica muy común el usar un motor de juego que apoye al desarrollador.

Casi todos los juegos, en la industria del videojuego actual, utilizan alguna clase de motor de juego. Los motores de juego no dejan de ser frameworks que incluyen funcionalidades comunes entre la mayoría de los videojuegos, como un motor de render, uno de sonido, un motor de físicas, entre otras, de manera que cada vez que se quiere hacer un juego nuevo, se evite empezar desde cero. Además, otorga al desarrollador herramientas para hacer el desarrollo del producto más fluido y organizado.

Los motores de juego han evolucionado mucho desde sus inicios, cuando no eran más que una porción de código que se llevaba de un proyecto a otro. Actualmente, hay varios motores accesibles al público, cada uno con su propia licencia de desarrollo.

Con el fin de escoger el motor que mejor se adecúe a la naturaleza de este proyecto, se han analizado y comparado algunos de los motores públicos más usados en la industria, así como Cocos2d-x, Game Maker Studio, Swift, Unity3d y Unreal Engine.

La primera decisión ha sido descartar Swift de entre las opciones, pues solo es viable para los productos de Apple, los cuales utilizan iOS, y no Android.

Unity3d es sin duda el más popular de los motores, pues más de dos terceras partes de los juegos móviles utilizan Unity como motor [3]. Es potente, a la vez que ligero, fácil de emplear, dado que la documentación está muy bien detallada, y tiene soporte tanto para 3d, como para 2d. Comparándolo con Cocos2d-x, un motor de naturaleza similar, no tiene nada que envidiarle. Cocos es algo más ligero, pero Unity gana en opciones, características y escalabilidad.

Game Maker Studio es un motor que lo que consigue es simplificar enormemente el proceso de desarrollo de un juego, con su forma de programar con cajas y nodos, en vez de con un lenguaje de programación tradicional. Es ligero y flexible, aunque tiene muchas

limitaciones, como, por ejemplo, que está basado en gráficos en 2d, por lo que sufre mucho cuando tratas de añadir una tercera dimensión.

Unreal Engine también usa esas cajas visuales para programar, en vez de usar C++, pero al contrario que Game Maker, esto no simplifica la tarea, simplemente la hace más visual. Siendo este el motor más complejo de utilizar que hay en el mercado, también es de los más utilizados, justo por debajo de Unity. Aunque Unreal Engine 5 ya está disponible al público, dado que la versión disponible es solo una Beta, para la comparación se tiene en cuenta a Unreal Engine 4. En cuanto a aspectos Técnicos, Unreal Engine está por encima de todo el resto de motores comparados, incluido Unity3d [4]. No obstante, para el desarrollo de un juego para plataformas móviles tiene la gran desventaja de ser un motor muy pesado, que requiere de mucha potencia gráfica por parte del dispositivo que lo utiliza. Esto impide que el juego funcione correctamente en una gran gama de dispositivos.

3. Objetivos y Abasto

El objetivo final de este proyecto es tener un juego funcional con una conexión a servidor que le permita crear, jugar y terminar partidas multijugador con turnos simultáneos.

Con respecto al diseño, este juego se ha diseñado para ser un 4X accesible para no requerir de mucho tiempo durante una sesión de juego, a la vez que mantiene la profundidad del género.

El público objetivo son los usuarios que frecuentan juegos de estilos similares, como otros subgéneros de la estrategia, y que puedan sentirse atraídos por la accesibilidad del título. La gran mayoría de este público son varones jóvenes (entre 25 y 44 años) [5]. No por eso hay que dejar al resto del público fuera del foco: El juego debe ser accesible para toda clase de públicos.

Una vez aclarado lo que se quiere conseguir, se van a marcar los objetivos en bloques de trabajo.

- Diseñar e implementar el backend del juego, así como toda la infraestructura que este necesite para funcionar: Servidor, base de datos, etc.
- Diseñar e implementar el frontend del juego, así como la comunicación con el backend. El juego debe mostrar el estado de tus partidas, a la vez que te permite jugar los turnos de las partidas activas.
- Diseñar el juego en sí. El juego debe seguir los pasos de los 4X anteriores a él, mientras que innova en algún aspecto para diferenciarse del resto, todo manteniendo la accesibilidad de entrada a nuevos jugadores.

Además de estos objetivos principales, este proyecto tiene los siguientes objetivos secundarios:

- Implementar un sistema de Matchmaking que permita a los usuarios crear partidas con desconocidos, así como darles herramientas fuera de las partidas para darle cuerpo a la aplicación.
- Generar un apartado artístico atractivo para el juego (modelos, texturas, sonidos, música, etc.), de modo que el estilo del proyecto tenga vida propia, siendo así capaz de atraer a más público. No debe ser un estilo complejo, pero si coherente consigo mismo.
- Diseñar e implementar una Experiencia de Usuario que apoye la accesibilidad del diseño del juego, de modo que cada sesión de juego sea agradable y, por lo tanto, la retención del título sea mayor.

4. Análisis de Referentes

Los juegos 4X, abreviación para "E**x**plore, E**x**pand, E**x**ploit and E**x**terminate" en inglés, son un género de juegos, con énfasis en el jugador contra jugador asimétrico, en el cual varios participantes tratan de superar a los demás mediante varias estrategias posibles, de entre ellas el control de recursos, el control del mapa, etc.

Hace años, este género se volvió muy popular entre los usuarios de ordenadores domésticos. Ejemplos de títulos remarcables son: VGA Planets, Heroes of Might and Magic o Age of Wonders.

Pero este género ha pasado a un segundo plano durante la última década, transformándose en un nicho. El género de la estrategia forma un 3,7% de los ingresos de videojuegos en Estados Unidos [6]. Todos los subgéneros que conforman el género de la estrategia comparten ese 3,7% de las ventas, incluyendo a los 4X. Además, los 4X son el subgénero más complicado y complejo en el género, lo cual los hace aún más propensos a quedarse en un nicho. Puede decirse que el subgénero no está en su mejor momento.

Aun así, hay excepciones. Sid Meier's Civilization es una saga de videojuegos 4X extremadamente popular. Estos juegos son producciones de mucha envergadura y calidad en todos los aspectos (jugable, sonoro, visual, etc.), que simplifican las mecánicas de los 4X para que no se requiera de un proceso de aprendizaje lento y sea accesible a prácticamente todos los públicos, a la vez que profundizan en las dinámicas, para que los más expertos tengan la posibilidad de desenvolver todo su potencial.

La hipótesis propuesta consiste en que el hecho de que los 4X no sean populares hoy en día, es porque mantienen la fórmula original del subgénero, sin adaptarse a las necesidades de las personas.

Por lo tanto, a la hora de llevar este subgénero al mercado móvil, hay que tener en consideración varios factores. La mayor parte del público objetivo tiene entre 25 y 44 años, edades en las que el tiempo no es un recurso que sobre. El tiempo que dura una sesión de juego usual para esta clase de usuarios es de treinta minutos [5]. Si aspiramos a que varios jugadores habituales de juegos de estrategia prueben el producto, debemos ofrecerles una

experiencia complementaria a la que estén acostumbrados, y que sea flexible con la duración de sus sesiones. Por lo tanto, usando un juego basado en turnos simultáneos asíncronos, podemos resolver los problemas de tiempo de sesión y de sincronización con los demás jugadores a la vez.

5. Metodología

Para completar un proyecto de este calibre, es necesario organizar el tiempo y los recursos acorde a una metodología. Según las características de los requerimientos se han comparado los pros y los contras de varias metodologías aplicadas al desarrollo del proyecto [7]. Dada la extensión de este proyecto, junto a la inexperiencia acerca de varios campos de desarrollo, se ha decidido usar el "modelo espiral" [8].

El "modelo espiral", desarrollado por Barry W. Boehm es un modelo que se basa en ciclos de desarrollo flexibles, centrado en minimizar los factores de riesgo. Cada ciclo tiene cuatro fases:

- Determinar Objetivos. Se esclarecen los objetivos a cumplir al final del ciclo y se determinan, para cada punto, condiciones de éxito y de fracaso.
- Identificar y resolver riesgos. Se hace un análisis de cada punto para determinar los posibles riesgos que puedan desarrollarse. Entonces se evalúan todos los posibles acercamientos a los distintos problemas, y se escoge al que menos riesgos conlleve.
- Desarrollar y testear. Se implementan todos los puntos de la primera fase, siguiendo la metodología escogida en la segunda, resolviendo todos los riesgos de la manera más segura posible.
- Planificar la siguiente iteración. Se aprueba el final del ciclo actual, y se proponen metas para el siguiente.

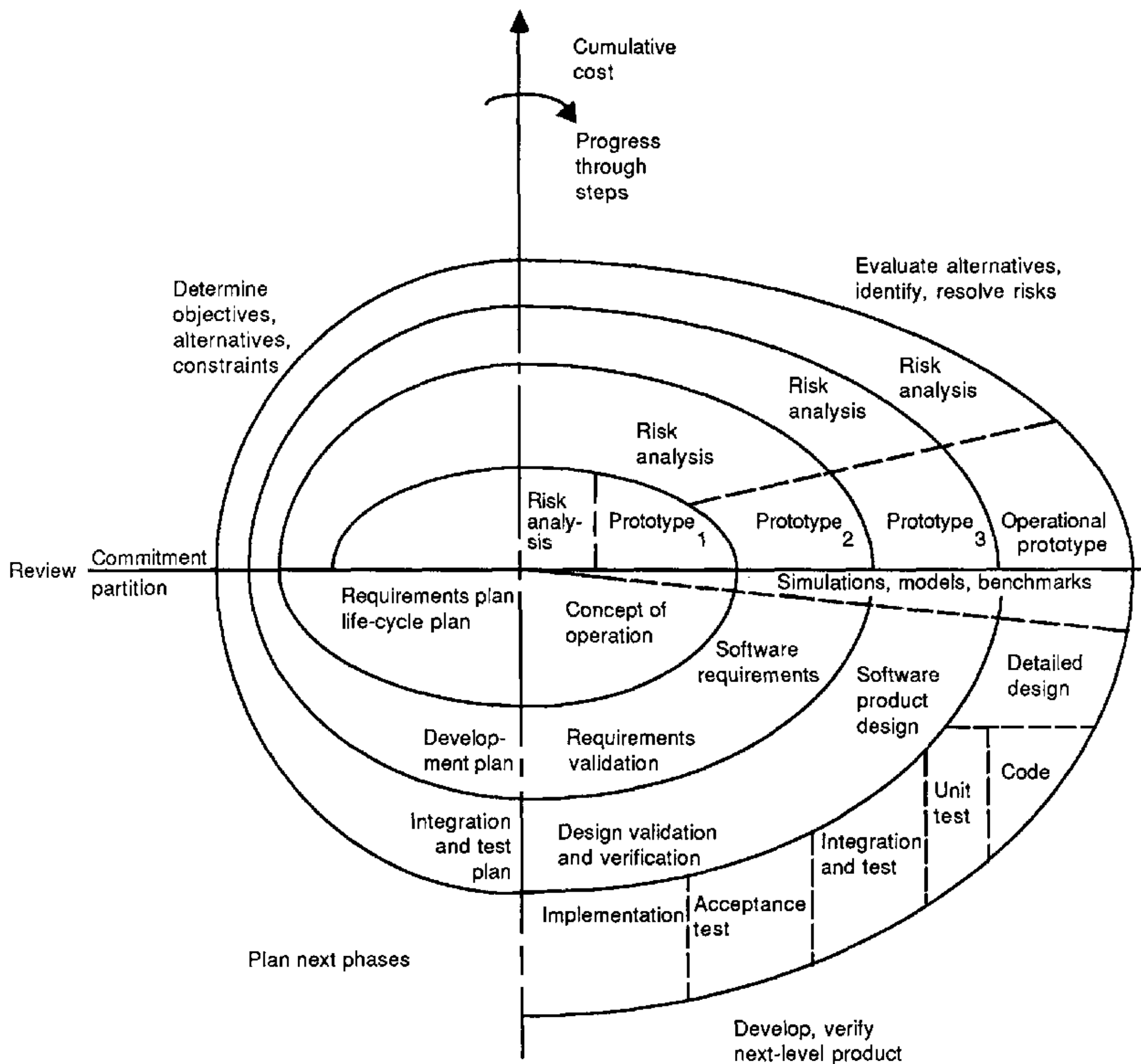

Figura 4. Modelo en espiral. Fuente: [8]

Cada ciclo tiene una duración indeterminada, y usa una metodología distinta al anterior. Esto permitirá a las distintas partes del proyecto, que requiere de disciplinas muy dispares, avanzar sin pisarse unas a las otras. A cambio, esta metodología requiere que el análisis de riesgos sea lo más profundo y preciso posible, o podría terminar dañando todo el proyecto.

6. Definición de Requerimientos Funcionales y Tecnológicos

A continuación, se listan los requerimientos funcionales y tecnológicos que el producto final debe cumplir.

Requerimientos funcionales:

- Identificar al usuario.
- Mantener un registro de las partidas pasadas.
- Mostrar las partidas en curso de cada jugador.
- Mostrar por separado las partidas pendientes del turno.
- Crear partidas.
- Invitar a otros usuarios a una partida.
- Acceder a partidas en curso.
- El juego debe funcionar y ejecutarse según las normas establecidas en su diseño.

Requerimientos Tecnológicos:

- Capacidad de acceso al contenido del juego cómodamente desde una variedad de dispositivos Android.
- Mantener una base de datos con la información de todos los usuarios y partidas necesarios para el correcto funcionamiento del juego.
- Ofrecer seguridad en el almacenamiento de datos.
- Ofrecer la opción de alterar distintas opciones gráficas para adaptar el juego a las necesidades del dispositivo usado.
- Computar el resultado de una ronda de forma determinística según las normas del juego.

7. Iteraciones del Modelo Espiral

A continuación, se detallan los distintos ciclos por los que pasa el proyecto, así como las decisiones tomadas en cada uno.

7.1. Primer Ciclo

El primer ciclo de desarrollo sienta las bases del proyecto, de modo que todos los ciclos posteriores puedan construirse a partir de lo creado durante esta iteración.

Por eso mismo, hay que determinar qué factores son fundamentales para que el proyecto pueda erigirse con seguridad y eficacia.

Dado que el apartado artístico se ha considerado un objetivo secundario, ha sido pospuesto por el momento.

En cuanto al diseño, es necesario tener claro cuáles son los requerimientos funcionales que deben implementarse, antes de iniciar el desarrollo. Por lo tanto, durante este ciclo habrá un componente importante en desarrollar y finalizar una primera versión de un Documento de Diseño del videojuego.

Por último, en el aspecto más técnico del desarrollo, se requiere una base sobre la que trabajar. Es un objetivo de este ciclo generar un proyecto, e implementar una conexión cliente-servidor segura.

7.2. Segundo Ciclo

El segundo ciclo sirve para definir la arquitectura tanto del servidor como la del cliente, así como la de la Base de Datos. Durante este ciclo se da forma a los tres sectores con cuidado para que no se pisen unos a otros en el futuro.

Se ha desarrollado un sistema de autenticación seguro desde todos los extremos. Con ello, se ha empezado a generar la forma de todos los apartados de este proyecto.

El cliente muestra formularios de creación de cuentas y de autenticación al usuario, a prueba de inyecciones de código, y mostrando la información que el usuario necesita ver. Tanto nombres de usuario como contraseñas están controlados por sus respectivas normas.

El servidor recibe la información, ya tratada para que no se puedan enviar consultas sin sentido, y compara los resultados contra la base de datos, usando un algoritmo de encriptación en la contraseña.

La base de datos contiene ahora una colección de cuentas, cada una con sus propiedades, que se ha llenado con cuentas de prueba.

Durante este ciclo, se ha detectado que la magnitud del proyecto es mucho mayor a lo planeado, y se ha reducido el foco de producir un juego entero, a generar los sistemas principales para que luego pueda llenarse de contenido fácilmente. El objetivo es crear el prototipo base sobre el que construir el producto final.

7.3. Tercer Ciclo

Este ciclo sirve para generar las partidas y empezarlas a visualizar. El objetivo es crear un sistema de invitaciones donde un jugador pueda generar una partida e invitar a otros jugadores a unirse a dicha partida. Cuando la partida se llena, el servidor automáticamente genera el escenario de la partida y los jugadores pueden empezar.

Como siempre, esta tarea se puede partir en dos grandes grupos: Backend, y Frontend.

En el frontend se crea un formulario que permita al jugador la creación de partidas, así como unirse a ellas.

El servidor deberá controlar la creación, unión y borrado de partidas, así como de mantener la cuenta de los jugadores que van entrando. Cuando la partida está llena, este debe empezar la generación del mapa.

Se debe implementar la generación de un mapa de estrellas procedural que tenga las propiedades necesarias para que sea jugable.

Por último, el cliente debe mostrar ese mapa estelar.

7.4. Cuarto Ciclo

Durante esta iteración, se debe desarrollar los planetas:

El cliente debe permitir la acción de pulsar sobre una de las estrellas del mapa estelar y abrir una escena que muestre una representación del sistema solar: La estrella con sus datos, y cada uno de los planetas.

Los planetas los debe generar el servidor al momento de generar la partida, proceduralmente, siguiendo el documento de diseño de juego. Cada planeta tiene sus características, así como sus tecnologías, edificios y naves.

El cliente, a su vez, debe proporcionar al usuario una forma de interactuar con los planetas, así como de mostrar la información que el jugador requiera.

7.5. Quinto Ciclo

Para el quinto ciclo, el juego base ya está establecido, y lo que ahora queda por desarrollar es el contenido.

Durante esta iteración toca darle forma al prototipo, generando los módulos base de tecnologías para que el juego sea funcional, así como los edificios correspondientes.

Aparte, debe ser posible generar naves espaciales, así como destacamentos para moverlos de estrella a estrella.

Este ciclo ha sido interrumpido debido a la escasez de tiempo. Debido a esto, las naves espaciales no han podido ser desarrolladas para el prototipo del proyecto.

8. Desarrollo

En este apartado se exponen todas las decisiones tomadas durante el desarrollo del prototipo, así como la arquitectura y el diseño tenidos en cuenta.

8.1. Game Design Document

Dentro de la industria del videojuego, al documento que contiene el diseño íntegro del producto en desarrollo, se le denomina 'Game Design Document'.

En su forma más básica, es un documento tradicional dividido en secciones que diseccionan el comportamiento de cada aspecto del juego, mientras que su forma más moderna en equipos de diseño grandes es generar un espacio interconectado (usando alguna herramienta especializada) y un método de desarrollo 'Agile'.

Uno de los puntos más relevantes concernientes al desarrollo del documento, es la herramienta y/o plataforma en las que se va a desarrollar. El editor de texto es una herramienta versátil con un interés especial en este análisis, puesto que será la base sobre la que se van a comparar el resto de herramientas. Plataformas como las Wikis, precisan de un poco más de tiempo de desarrollo, pero otorgan una estructuración, facilidad para enlazar información, y una escalabilidad que el editor de Texto no tiene. Por último, herramientas especializadas, como Nuclino o Articy, a cambio de una licencia y el tiempo de aprendizaje y habituamiento, te ofrecen multitud de herramientas dentro de una plataforma unificada que permite a un equipo entero trabajar en el diseño del juego, con información actualizada, y sin pisarse unos a otros.

Puesto que el equipo de desarrollo está formado por un solo individuo, no parece necesario la inversión en licencia ni el tiempo requerido para aprender a usar una herramienta especializada. El método tradicional del editor de texto es tan bueno como siempre, pero la idea de usar una wiki también tiene sus ventajas. Por desgracia, también tiene sus riesgos. Generar una Wiki, aunque en menor medida, también requiere habituamiento. Además, requiere de más tiempo de desarrollo. Dado que la escalabilidad del documento

es un riesgo muy pequeño, puesto que, en comparación a grandes títulos, el juego a desarrollar no va a ser demasiado extenso, la opción con menos riesgos es también la más básica. El documento de Diseño del Juego se va a desarrollar con un editor de texto, ayudado de hipervínculos y hojas de cálculo.

En cuanto a metodología, teniendo en cuenta que el proyecto consiste en un videojuego multijugador donde los jugadores se baten entre ellos para determinar un vencedor, esta tiene que adaptarse a las necesidades. No se requiere de un diseño que empiece por el primer nivel y termine en el último. Se necesitan revisiones cíclicas y tener todos los sistemas en cuenta. Por lo tanto, se va a utilizar una metodología ágil y un acercamiento Top-Down.

El documento de diseño del juego puede encontrarse en los anexos de este trabajo.

8.2. Lenguaje del Servidor

El proyecto de este juego se puede partir en tres partes: Clientes, servidor y Base de Datos

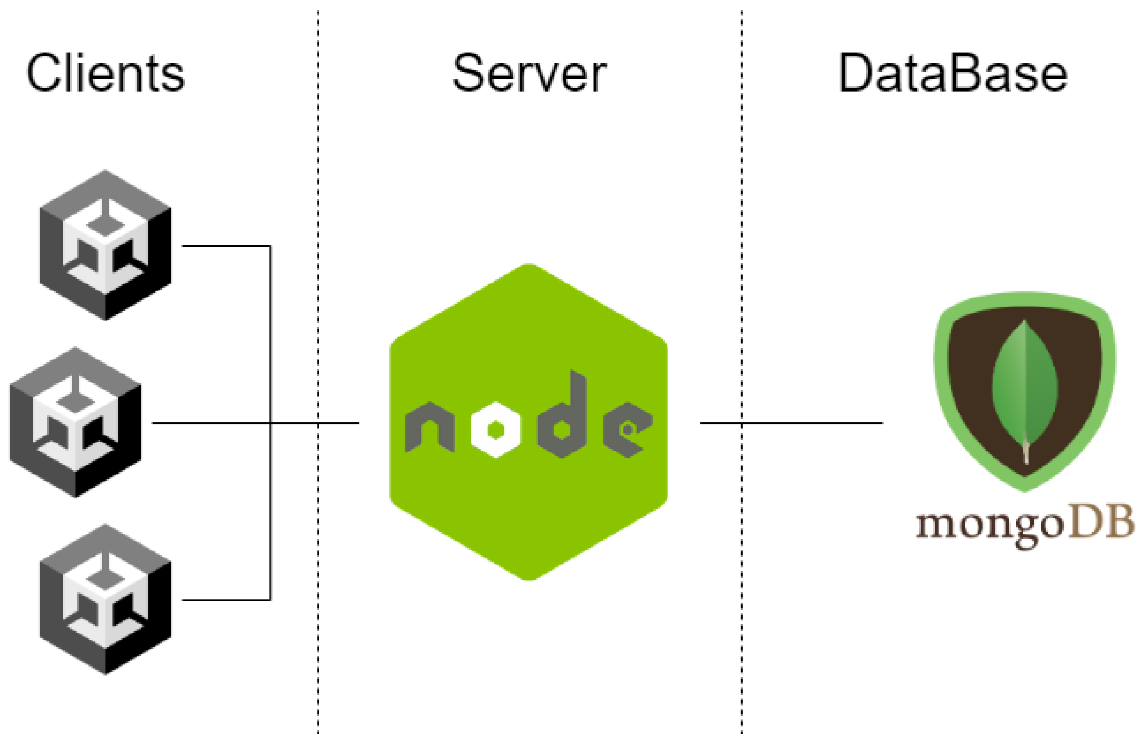

Figura 5. Diagrama de la arquitectura del proyecto. Fuente: Elaboración propia.

El frontend, desarrollado en Unity, es un cliente dependiente de un servidor. Este cliente hace peticiones al servidor web, desarrollado usando Node.js. El servidor procesa la llamada, y pide los datos necesarios a la base de datos MongoDB. Cuando estos datos han sido procesados, se los envía al cliente, y este se los muestra al usuario.

8.2.1. El Entorno de Conexión del Servidor

Tras el análisis de las distintas arquitecturas de conexión entre clientes, el modelo de conexión escogido es el de Cliente-Servidor, pues se adapta perfectamente al proyecto al permitir a todos los jugadores conectarse asíncronamente al juego, y ofrece muy buena capacidad de escalabilidad.

Unity3d ha sido escogido para desarrollar el cliente de este proyecto. Unity es el motor que mantiene un perfecto equilibrio entre complejidad y potencia, mientras que es el que más herramientas da a disposición del desarrollador y el que mejor documentación posee.

Por su parte, el servidor debe ser robusto y seguro, de manera que no pueda comprometer la seguridad de los usuarios, a la vez que ligero, puesto que el servidor de un videojuego como el de este proyecto no necesita realizar cálculos complejos, ni comunicarse con una multitud de servicios distintos, pero si comunicarse con miles de usuarios al mismo tiempo.

Con estas características por delante, se han comparado varias tecnologías populares en el desarrollo de web y móvil, también usadas en desarrollo de videojuegos.

Tanto Ruby on Rails como Django, así como Laravel, son frameworks que imponen un modelo a cambio de un desarrollo fluido y simple, asumiendo que la aplicación a desarrollar es una aplicación web convencional. Por ese motivo, han sido descartadas rápidamente para el uso en este proyecto.

C++ es uno de los lenguajes menos útiles para el desarrollo de prototipos. C++ es un lenguaje desconocido para el desarrollador, a la vez que es muy complejo y verboso para ser usado en un proyecto de tan corta duración. El riesgo de no llegar a un prototipo funcional es demasiado alto.

Java y Spring Boot son las tecnologías que se han trabajado en el grado académico, por lo que la facilidad que aporta el desarrollo, es un riesgo menos a tener en cuenta. Aparte, su potencia y la capacidad de usar multi-threading pueden ser aliados muy poderosos a la hora de procesar los turnos de juego.

.NET tiene ventajas muy similares a Spring, con el valor añadido de que es mucho más ligero, y sencillo de usar.

Por su parte, Node.js es un framework veloz y eficaz, que es capaz de administrar muchos procesos utilizando paralelismo. Es simple de usar y es muy sencillo de acoplar a módulos externos.

Valorando entre .NET, Spring y Node.Js, la decisión final ha sido trabajar con Java Script:

Los tres candidatos son muy eficaces y potentes. La estabilidad y seguridad de Java son características a tener en cuenta, pero la naturaleza de este proyecto se beneficia más de la facilidad de prototipado de JavaScript. Además, la ligereza de JavaScript permite una mayor flexibilidad de las capacidades de la máquina que ejecute el servidor, sin perder eficiencia significativamente en el proceso.

8.2.2. La Arquitectura del Backend

La decisión que incumbe a este apartado es la de definir quién hará el procesamiento de los turnos, si el mismo módulo de Node.js, o un subprograma que será lanzado por el módulo principal.

Desarrollar las funcionalidades del juego en JavaScript comparado al entorno de Unity puede ser mucho más obtuso: El cliente y el servidor siempre han tenido que compartir código, de manera que desarrollar una aplicación web siempre ha obligado a desarrollar las funcionalidades dos veces. Pero el entorno de Unity te permite programar las funcionalidades una sola vez, y luego, a la hora de generar las builds del juego y el servidor, escoger cuál es la función que va a realizar. Esta herramienta resulta realmente útil si se utiliza un modelo de comunicación 'Client as a Host', puesto que solo hay que adaptar el código moderadamente para que todo funcione como es debido.

Esto provoca que desarrollar en Java Script sea una pérdida de tiempo, y una fuente de posibles errores de desincronización. Pero en contraparte, para hacer funcionar esta tecnología usando el modelo Cliente-Servidor, se requiere de una solución de arquitectura que resuelva los problemas de acoplamiento entre tecnologías.

Usar esta arquitectura, requiere que cuando el servidor detecte que se han ejecutado todos los turnos, se ejecute un subproceso que resuelva todos los conflictos, para acto seguido devolver el turno al servidor, y entonces a la base de datos.

Si las aplicaciones de frontend y backend fueran desarrolladas en este mismo entorno (Unity), la comunicación entre ellos no requiere de la generación de diccionarios

adicionales en el entorno de backend. De este modo, la serialización de objetos es casi automática.

Por el contrario, la ejecución de un proceso C# utilizando el Unity Engine por parte del servidor es muy costoso en recursos de procesamiento, y requiere de un servidor mucho más potente de lo esperado. Además, se deben idear sistemas para que el servidor no procese demasiadas partidas al mismo tiempo. Dado que este proceso no es de alta prioridad, se puede generar una cola de partidas por procesar.

A su vez, si se desarrolla el procesador de turnos en un módulo JavaScript, opuesto a la automatización de la serialización de objetos de Unity, tendremos la oportunidad de crear una arquitectura en la base de datos lo más limpia y escalable posible. Esto ofrece un riesgo a un diseño obtuso más bajo, puesto que existen millares de ejemplos de servicios parecidos, y de documentación al respecto.

Como síntesis, se puede pensar en el desarrollo en el entorno de Unity como rápido y a prueba de errores, y el desarrollo en un módulo JavaScript como eficiente y escalable.

Dada la naturaleza del proyecto, un juego en el que las funcionalidades secundarias son lo que dan variedad y riqueza a las partidas, que el código sea escalable es altamente importante, y no se puede correr el riesgo de ir entorpeciendo el desarrollo a medida que este avanza. Aparte, uno de los riesgos identificados de usar MongoDB como base de datos es la posibilidad de generar una mala arquitectura, y con ello llenar el proyecto de errores derivados de la base de datos. La solución a este riesgo y la decisión tomada durante el primer ciclo es la de tener especial cuidado en el diseño de la arquitectura de la base de datos.

En conclusión, la decisión es continuar usando únicamente JavaScript en el lado del servidor.

8.2.3. TypeScript

Durante los dos primeros ciclos de desarrollo se ha usado JavaScript como lenguaje para desarrollar el servidor, usando Node.js como framework. Pero la necesidad de una estructura más rígida, tanto por la parte de la base de datos, como por la lógica que en ese momento iba a empezar a desarrollarse, el lenguaje ha sido migrado a TypeScript.

TypeScript es un superconjunto de JavaScript. Es un lenguaje que añade los tipos estáticos a la escritura, añadiendo rigidez y organización al código intrínsecamente. Permite al programador definir interfaces, y saber en todo momento con lo que se está tratando gracias al compilador de TypeScript que provee de información sobre las variables, las clases, los objetos, así como de autocompletado y sugerencias.

Además, el compilador de TypeScript compila el código en archivos JavaScript que se ejecutan como tal, por lo que, virtualmente, TypeScript es equivalente a JavaScript en tiempo de ejecución.

8.3. Base de Datos

8.3.1. Elección de la Base de Datos

Para elegir la base de datos, hay que tener en cuenta qué clase de datos necesitamos guardar, y cómo se necesita acceder a estos. Claramente, tenemos que guardar la información de la cuenta de cada usuario, así como sus partidas y el estado de estas. El problema viene al tratar de prever cuáles van a ser las necesidades del estado de las partidas, dado que el Documento de Diseño del Juego aún no está desarrollado.

Teniendo esto en cuenta y sumándole que el diseño del juego va a variar a lo largo del desarrollo y testeo, se necesita una base de datos flexible, que se adapte a las necesidades del proyecto. De las bases de datos comparadas, destacan las no relacionales, en especial las basadas en documentos. Este tipo de base de datos facilita la tarea de guardado de estados de partida, puesto que estos estados se suelen tratar de una cantidad considerable de datos simples relacionados únicamente con su jugador, datos que pueden ser convertidos a un documento de forma sencilla.

Estos datos de partidas van a ser creados, solicitados, sobrescritos y eliminados con frecuencia, por lo que MongoDB, un sistema de base de datos NoSQL orientado a documentos optimizado para contenido muy transitorio, va a ser el sistema designado para el desarrollo de la aplicación.

Se debe tener en cuenta que los riesgos de MongoDB frente a una base de datos relacional como las trabajadas durante el curso académico son muy distintos. Una base de datos relacional como Oracle no ha sido elegida dado que los riesgos pueden hacer que el diseño de la base de datos colisione con el del juego en sí, lo cual puede llevar a casos en los que se deban tomar decisiones drásticas en uno de los dos diseños, sino ambos. Los riesgos de MongoDB, aunque menores, requieren que se sea muy organizado y riguroso con el diseño de la Base de Datos, puesto que MongoDB prácticamente no impone restricciones, y ello puede conllevar a una mala organización que ocasione errores e imprevistos en el proyecto.

8.3.2. Arquitectura de la Base de Datos

El siguiente diagrama muestra el diseño que se ha seguido en el momento de montar los esquemas de la base de datos y, por lo tanto, su forma actual.

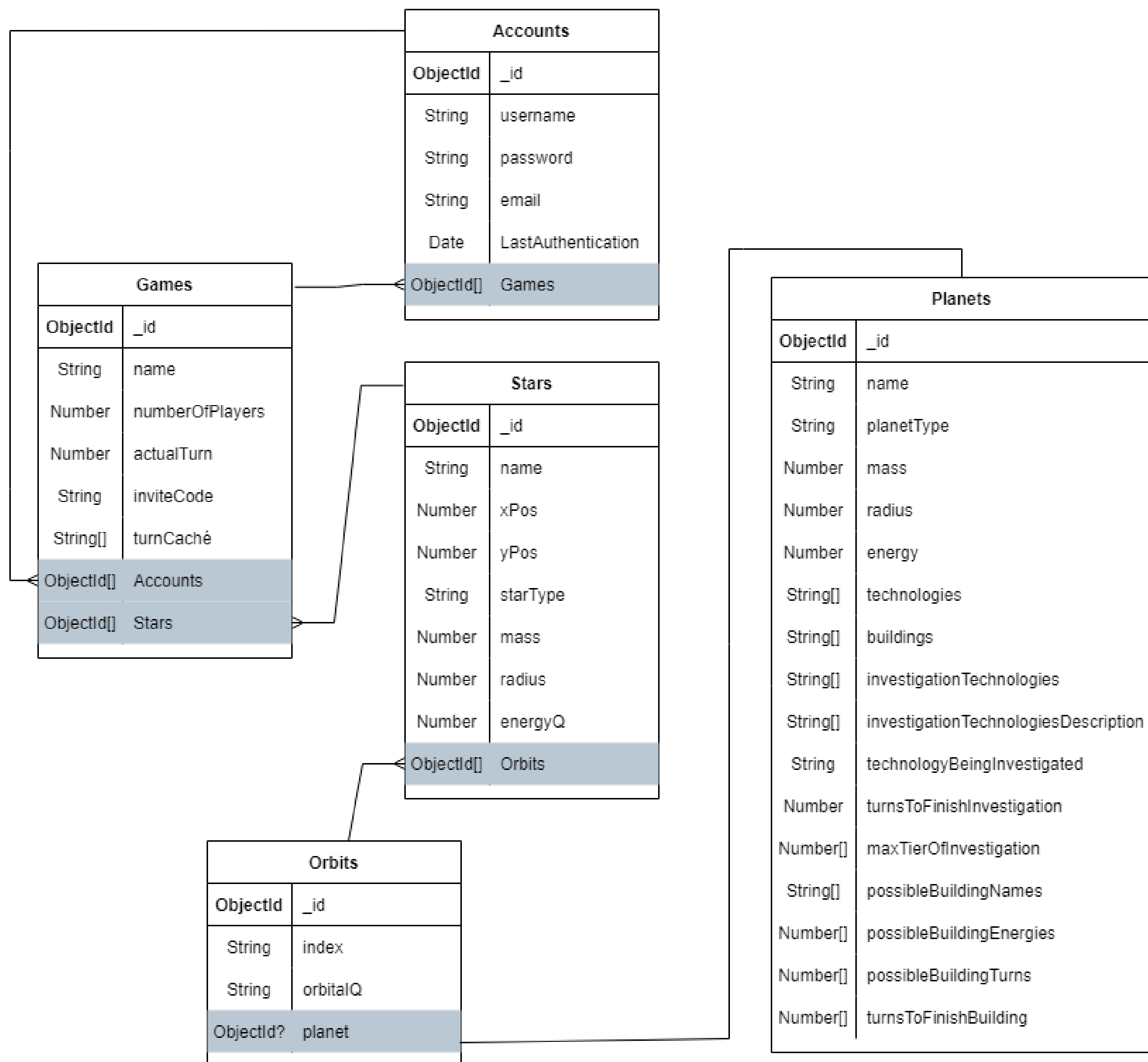

Figura 6. Diseño de la Base de Datos. Fuente: Elaboración propia.

Como se puede observar, las partidas contienen una tabla de referencias a las cuentas, mientras que éstas también tienen una tabla de referencias a las partidas. Esto se debe a

que una cuenta debe mantener el registro de las partidas a las que puede acceder, mientras que una partida debe conocer los jugadores que están implicados en ella.

La decisión de tener una colección para las órbitas en vez de colocar la poca información que éstas almacenan en la colección de los planetas (como se ha hecho con muchos otros datos) es debido a que es una buena forma de mantener la cuenta de cuántos planetas puede llegar a tener una estrella, independientemente de los que tenga. Por lo tanto, todas las estrellas del mismo tipo tienen el mismo número de referencias a documentos de órbitas, pero no todas tienen el mismo número de planetas.

El documento de los planetas está lleno de información, puesto que este tipo de objeto es con el que el jugador va a interactuar de manera continua.

8.4. Sistema de Autenticación

El sistema de autenticación está formado por dos acciones muy distintas: registrar una cuenta nueva y entrar con una cuenta existente.

Puesto que durante esta sección ha sido la primera en la que se ha trabajado con Node.js, se ha necesitado documentación sobre muchos aspectos diferentes del desarrollo.

Se han usado los ejemplos de varias fuentes [9] [10] para entender rápidamente el funcionamiento de las tecnologías a usar, y así poder construir una solución óptima para el proyecto en cuestión.

Primero de todo se ha construido un método en el backend para registrarse, pidiendo un formulario. Se piden los datos de entrada, como el nombre de usuario y la contraseña, así como la dirección de e-mail. Este dato puede ser importante para el producto cuando esté en un estado de producción, puesto que otorga la capacidad al usuario de comunicar que ha perdido las credenciales, por ejemplo, o nos permite enviar información a todos los usuarios, de entre otras funcionalidades.

La contraseña, que nos llega como una cadena de caracteres, se encripta usando una implementación del algoritmo ‘Hash’ de Argon2 [11], utilizando como ‘salt’ un dato pseudoaleatorio generado por la librería ‘Crypto’ de Node.js. Este dato lo guardamos en una cadena de caracteres en la base de datos, junto al ‘salt’.

```
crypto.randomBytes(32, function(err, salt) {
    argon.hash(password, salt).then(async (hash: any) => {

        var newAccount = new AccountModel({
            username: username,
            password: hash,
            email: email,
            lastAuthentication: Date.now(),
        });
        await newAccount.save();
        response.code = 0;
        response.msg = "Sing Up Successful";
        response.gameAccount = { username: username };
        res.send(response);
        return;
    });
});
```

Figura 7. Encriptación de la contraseña usando Argon2. Fuente: Elaboración propia.

De esta manera, en el método de identificación, cuando el usuario introduce el nombre de usuario y su contraseña, la librería Argon verifica si el valor que ha sido guardado en la base de datos como contraseña del usuario cuyo nombre equivale al introducido por el cliente, coincide con la combinación del 'salt' también guardada en la base de datos, y la cadena de caracteres que el usuario ha introducido como contraseña.

Es importante denotar que, durante todo el proceso autenticación, el servidor devuelve la información de forma limitada, para complicar mucho un posible caso de infracción de la seguridad.

Teniendo ya el backend listo, el frontend ha sido desarrollado construyendo encima del servidor. Se ha diseñado un formulario visual que se muestra al usuario, que varía

dependiendo de si este desea registrar un nuevo usuario, o identificarse con uno ya existente.

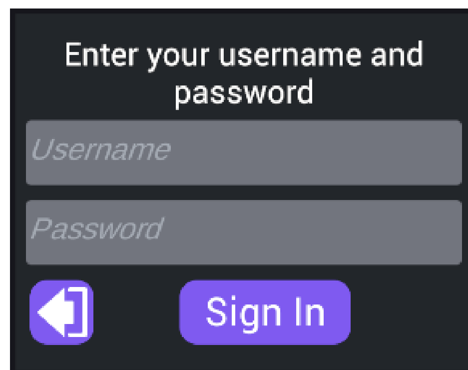

Figura 8. Pantalla de inicio de sesión. Fuente: Elaboración propia.

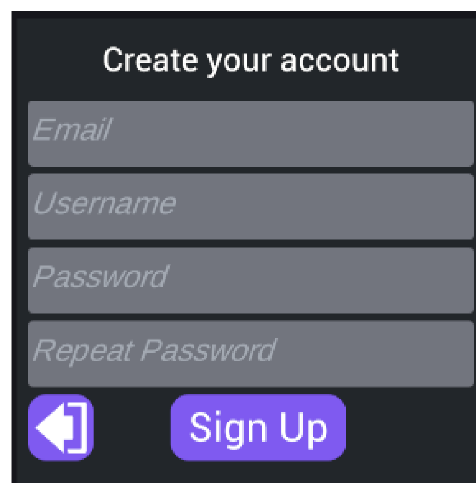

Figura 9. Pantalla de creación de cuenta. Fuente: Elaboración propia.

Todos los campos se comprueban para que no se envíen datos prematuramente inválidos al servidor. Como caso a destacar, la contraseña usa un 'Regex' bastante complejo: La contraseña debe ser un texto de entre cinco y treinta y dos caracteres, dónde se pueden usar letras, números y caracteres especiales. Para ser válida, esta debe contener al menos un número y una letra.

El cliente envía los datos al servidor y transmite la respuesta al usuario, ya sea intrínsecamente, mostrándole una respuesta en forma de texto, o de formas alternativas, como cambios de escena, iconos de carga o botones no funcionales temporalmente.

La finalidad de esta sección es la de que el usuario pueda conectarse a su cuenta desde cualquier dispositivo, y acceder a su contenido, sus partidas, y sus opciones, en cualquier momento.

8.5. Generación de Partidas

Las partidas son espacios en la base de datos que unen a unos cuantos usuarios con un estado de una partida. Cuando se llegan a ciertos requisitos, se genera un mapa estelar: un lienzo en el que los jugadores pueden ejercer todas las acciones que el juego les permita.

Durante el diseño se han propuesto distintas formas de conseguir que varios jugadores puedan juntarse en una misma partida a voluntad. La más común, dentro del ámbito de los videojuegos, es por mensaje de invitación, usando un sistema de amistades y mensajes que proporciona el propio servicio. La decisión final para el prototipo es, de hecho, una solución muy popular en prototipos de juegos, invitación por código.

El funcionamiento es el siguiente: Cuando un jugador crea una partida, el sistema le da al usuario un código único que está ligado a la partida en sí. Más adelante, otro jugador puede unirse a esa misma partida si introduce ese código en cuestión.

El cliente ha sido desarrollado primero, añadiendo las opciones de crear partida y de unirse a una partida existente en el menú principal.

Cuando un jugador entra en el menú de creación de partida, se le pide que introduzca el nombre con el que será guardada la partida, así como el número de jugadores que van a participar. Este número puede variar desde dos a dieciséis jugadores. Cuando el usuario pulsa el botón de crear partida, el servidor recibe la petición y devuelve el código único de ocho dígitos. Este código se muestra en el cliente, junto a un botón que lo copia al portapapeles del usuario, como se muestra en la Figura 10, de forma que dicho usuario pueda compartirlo con los demás participantes de la partida.

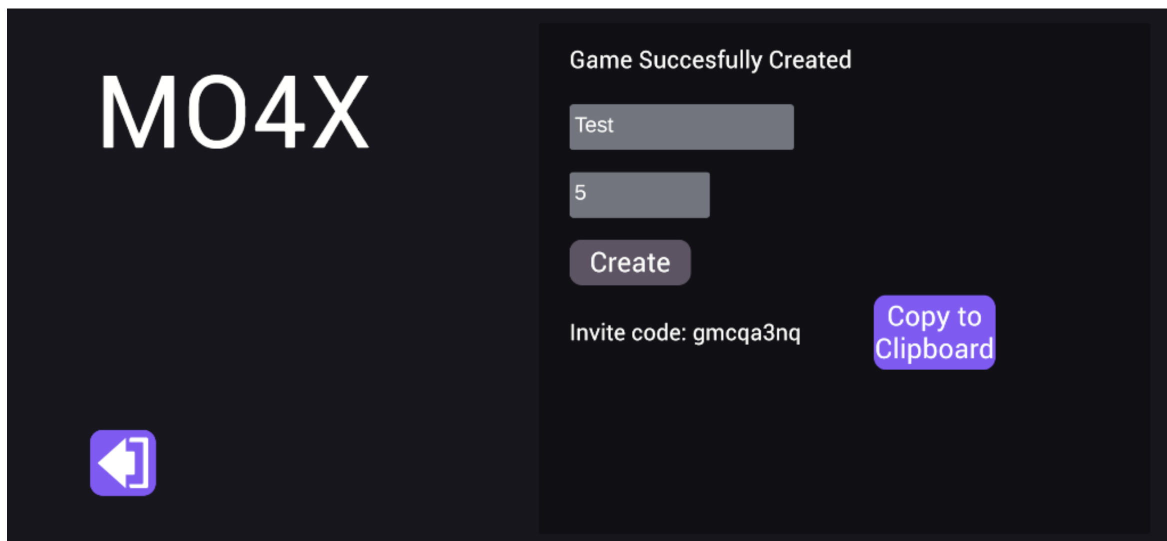

Figura 10. Menú de creación de partidas. Fuente: Elaboración propia.

Por su lado, cuando un usuario quiere unirse a una partida, simplemente accede al menú de unión a partida existente, introduce el código de invitación (o pulsa el botón que copia el contenido del portapapeles), y pulsa el botón de unirse a la partida. El servidor devolverá la respuesta apropiada.

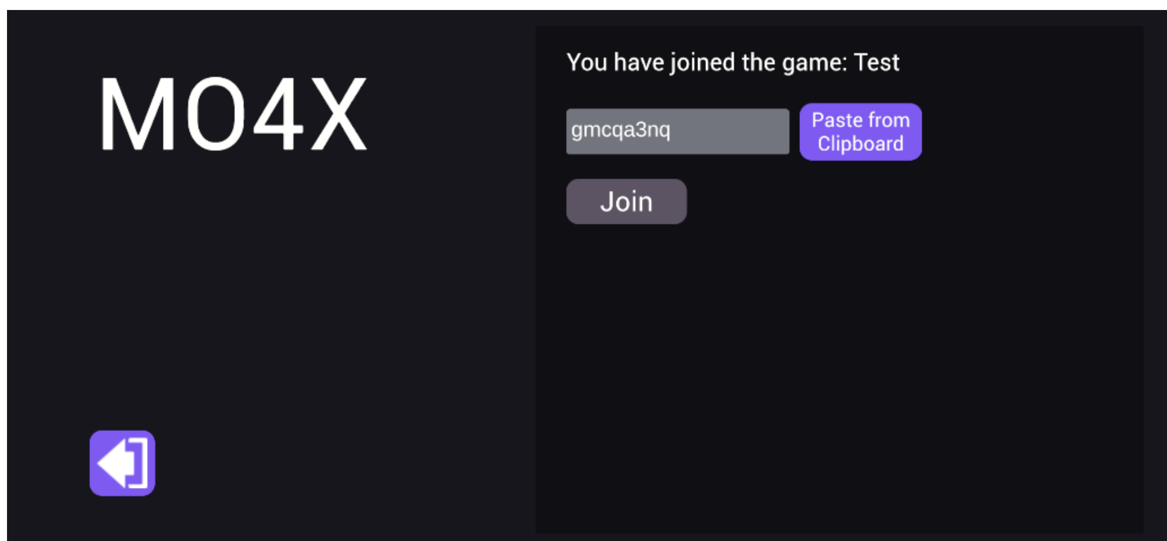

Figura 11. Menú de unión a partida existente. Fuente: Elaboración propia.

En cuanto al diseño del backend se ha partido en dos métodos distintos. El primero, el de crear partida, después de comprobar que los datos de usuario que el cliente envía son correctos, y que los datos del formulario también lo son, genera un número aleatorio que luego es transformado en una cadena de caracteres que es cortada para que se trate de un texto de ocho caracteres. Entonces, el servidor hace una petición a la base de datos para que devuelva cualquier partida aún no empezada con ese mismo código de invitación. Si la base de datos no encuentra ninguna coincidencia, se puede saber que ese código no ha sido usado por una partida que siga en espera de empezar, por lo que dicho código es válido.

```
var inviteCode;
var searchBool = false;

while (!searchBool) {
    inviteCode = Math.random().toString(36).substring(2, 10);
    var a = await GameModel.findOne({actualTurn: -1, inviteCode: inviteCode}) as gameSchemaInterface;
    if (a == null) searchBool = true;
}
```

Figura 12. Generación de un código aleatorio. Fuente: Elaboración propia.

Una vez el código es validado, se crea una nueva partida, que se guarda en el servidor, se le asigna el código de invitación, y se devuelve al cliente el resultado.

El segundo método, es el que permite al usuario unirse a partidas existentes. El método simplemente valida al usuario, y luego hace una petición a la base de datos buscando una partida cuyo código de invitación coincida con el introducido por el usuario. Si este coincide, el usuario es introducido automáticamente en esa partida.

Si después de unirse satisfactoriamente, el número de usuarios dentro de la partida es el número que el creador definió desde un principio, el servidor ejecuta la inicialización de la partida.

8.6. Cargar Estados de Partidas

Una vez que un usuario ha creado una partida, puede ver el estado en que se encuentra, o en su defecto, si todos los jugadores están listos, empezar a jugar.

Ambos escenarios se tratan de forma diferente, tanto desde el cliente como desde el servidor.

Cuando un jugador accede al menú de partidas en curso, el cliente le muestra una pantalla como la de la Figura 13, donde se muestra una carta por cada una de las partidas en las que es partícipe.

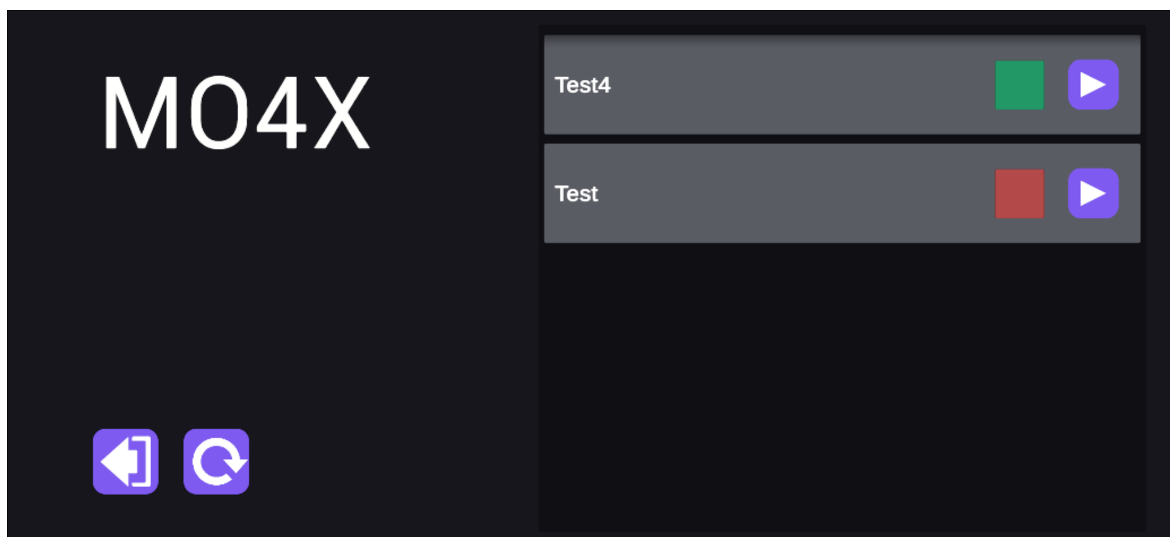

Figura 13. Menú de partidas en curso. Fuente: Elaboración propia

El monitor rojo o verde indica si la partida está disponible para que el usuario efectúe su turno (Verde), o si aún no puede jugarlo (Rojo).

Cuando el menú se abre, se hace una petición al servidor, y este devuelve todas las partidas en las que el jugador está participando. Solo devuelve algunos valores, como el nombre, los jugadores que participan, o el código de invitación, pero no el estado de la partida en sí.

El usuario puede pulsar el botón de refrescar para que la petición se efectúe de nuevo, y la información se actualice.

Una vez, el jugador pulsa el botón de entrar en la partida, el cliente, según si la partida ha empezado o no, muestra al usuario una pantalla diferente.

8.6.1. Cargar Partidas no Empezadas

Si la partida aún no ha empezado, se muestra un menú que informa al jugador sobre el estado de esa partida. Se indica qué usuarios están dentro de esa partida y cuántos faltan para que dé comienzo. A su vez, se muestra el código de invitación, así como un botón para copiarlo al portapapeles. Por último, hay dos botones más: uno para salir de la partida, y uno para cerrar ese menú emergente.

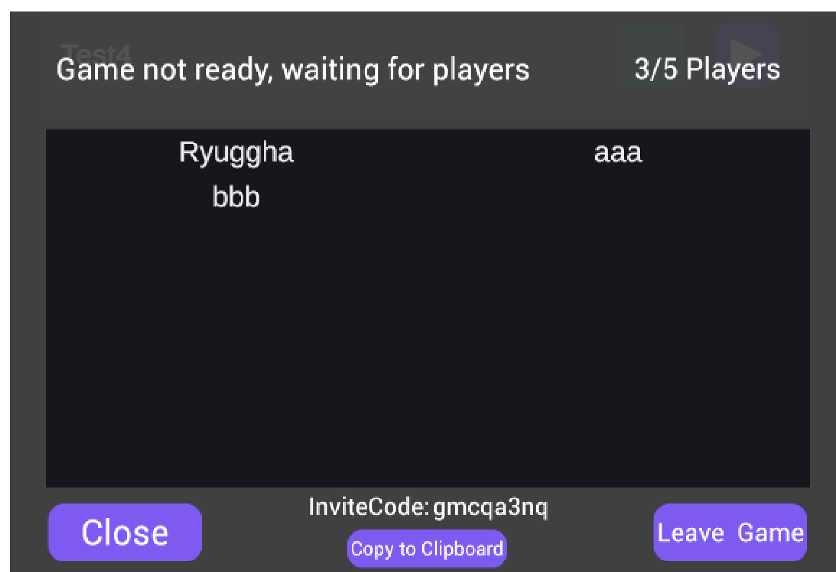

Figura 14. Menú de partida en espera. Fuente: Elaboración propia.

Este menú utiliza la información que el servidor ha enviado en la petición anterior y no requiere de una nueva. El único elemento que puede generar otra petición, es el botón de salir de la partida.

Este botón ejecuta otra función en el servidor que saca al jugador de entre los usuarios de dicha partida, y si este es el único jugador de dicha partida, borra la partida entera de la base de datos.

8.6.2. Cargar Partidas Empezadas

Por su parte, si la partida está empezada, el servidor envía el estado de la partida al completo al jugador. La idea detrás de esto, es que el jugador pueda jugar sin interrupciones durante la duración de su turno, inclusive si este pierde la conexión a internet en medio de la sesión de juego.

El servidor, en este momento, debe recopilar todos los datos acerca de una partida, y enviarlos al cliente. Estos datos incluyen las distintas estrellas del mapa, sus órbitas y planetas, así como las tecnologías, los edificios, y los estados de estos. En total, eso son unos 1.200 documentos que sacar de la base de datos.

Los requisitos que tiene el servidor, son que se consigan los documentos de forma secuencial para procesar los datos, y a su vez, que las peticiones no tarden demasiado tiempo, puesto que, de otra manera, estamos acaparando la base de datos para un solo cliente creando barreras para los demás, y haciendo esperar al usuario un tiempo de carga inapropiado.

Después de varias iteraciones, se ha llegado a la siguiente solución:

El servidor hace una llamada por cada tipo de documento, con pausas entre ellas para procesar la información por capas, en vez de recursivamente. Como ejemplo, creamos un vector con todos los identificadores de las estrellas de la partida, y hacemos una llamada pidiendo todas esas estrellas. Acto seguido, se procesan las estrellas, generando un vector que contiene todos los identificadores de los documentos de las órbitas que luego se piden a la base de datos, etc.

```
let auxStars = await StarModel.find({_id: {$in: game.stars}}) as StarSchemaInterface[];

let orbitIdList: ObjectId[] = [];

for (const star of auxStars) {
    orbitIdList = orbitIdList.concat(star.orbits);
}

let orbitInterfaces = await OrbitModel.find({_id: {$in: orbitIdList}}) as OrbitSchemaInterface[];

let orbitMap = new Map<string, OrbitSchemaInterface>(
     orbitInterfaces.map(o => {
         return [o._id.toString(), o]
     }));
let planetIdList: ObjectId[] = [];

for (const orbit of orbitMap.values()) {
    if (orbit.planet != null) planetIdList.push(orbit.planet);
}

let planetMap = new Map<string, PlanetSchemaInterface>(
    (await PlanetModel.find({_id: {$in: planetIdList}}) as PlanetSchemaInterface[]).map(o => {
        return [o._id.toString(), o]
    }));
```

Figura 15. Petición a la Base de Datos de los datos de una partida. Fuente: Elaboración propia.

Una vez el servidor tiene todos los datos necesarios, genera un árbol de objetos que simbolizan la partida entera, y la envía al cliente de vuelta.

Como se muestra en la Figura 16, el tiempo de espera medio desde que el usuario pulsa el botón de empezar partida, y que el cliente le muestra la partida en curso es de 3,25s. A lo largo de las iteraciones, este valor ha descendido drásticamente. Este valor se ha tomado al cargar diez partidas distintas y hacer la media del tiempo que tarda en cargar. El error de estos datos nunca supera el 15% de error.

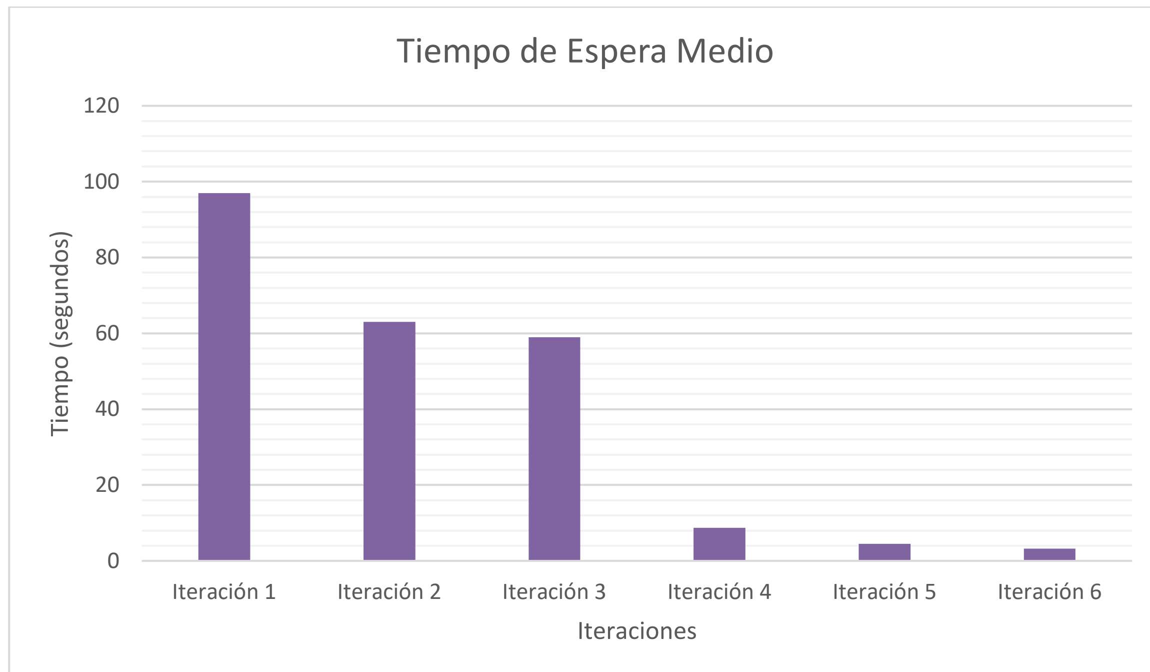

Figura 16. Tiempo de espera al cargar partida a lo largo de las iteraciones. Fuente: Elaboración propia.

Estos datos se han tomado con el servidor en local y la base de datos en la nube, por lo que el valor real variará ligeramente.

Una vez el cliente recibe la partida, regenera el árbol de clases, y lo asigna a una variable en la clase administradora, un objeto que usa el patrón 'singleton' [12] para mantenerse presente durante todas las escenas en las que el juego puede llevarse a cabo. Acto seguido, el cliente carga la escena de juego principal: El mapa estelar.

8.7. Generación del Mapa Estelar

El mapa estelar es un conjunto de estrellas que, entre sus características, se incluyen un par de coordenadas, 'x' e 'y'. Estas coordenadas se utilizan para localizar la estrella en un espacio cartesiano bidimensional, que abstrae la idea de espacio interestelar.

La generación sucede en el servidor. Una vez el último usuario se une a una partida, el servidor cierra las puertas a más jugadores y ejecuta la generación del mapa.

Para este trabajo, se han desarrollado unos cuantos módulos que se encargarán de trabajos distintos durante el proceso.

Uno de estos módulos es una clase auxiliar que se ha llamado 'Vector2', inspirada en UnityEngine.Vector2, de Unity. Esta clase permite trabajar con un tipo Vector bidimensional, en vez de con dos valores individuales 'x' e 'y'. Además, esta clase está cargada con varios métodos para operar con vectores en un espacio euclidiano, que facilitan el trabajo más adelante. Algunos de estos métodos son los siguientes:

- Magnitude: Devuelve el módulo del vector.
- Normalized: Devuelve el vector dirección con módulo 1.
- Scaled: Devuelve el vector multiplicado por un valor.
- Minus: Resta el vector a otro vector.
- Plus: Suma el vector a otro vector.

También se han incluido algunos métodos estáticos que devuelven vectores usados frecuentemente.

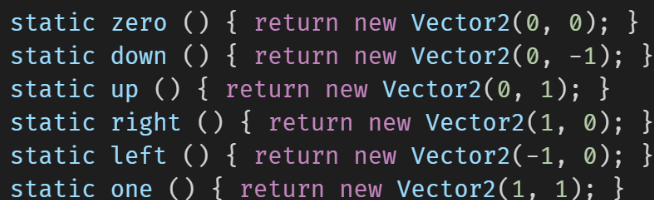

Figura 17. Métodos estáticos de Vector2. Fuente: Elaboración propia.

8.7.1. Algoritmo de Generación Procedural

El algoritmo de generación procedural del mapa estelar se basa en el procesamiento de un mapa de ruido Simplex [13] generado pseudoaleatoriamente a partir de una semilla. De este modo, el mapa es distinto cada vez que es generado. El proceso es el siguiente:

Primero, se establece un espacio circular dónde las estrellas se pueden generar. Este espacio crece en proporción de raíz cuadrada dependiendo del número de estrellas con el que se vaya a jugar, debido a que nos encontramos ante un espacio bidimensional. Acto seguido se genera un ruido que encaje con las dimensiones de este.

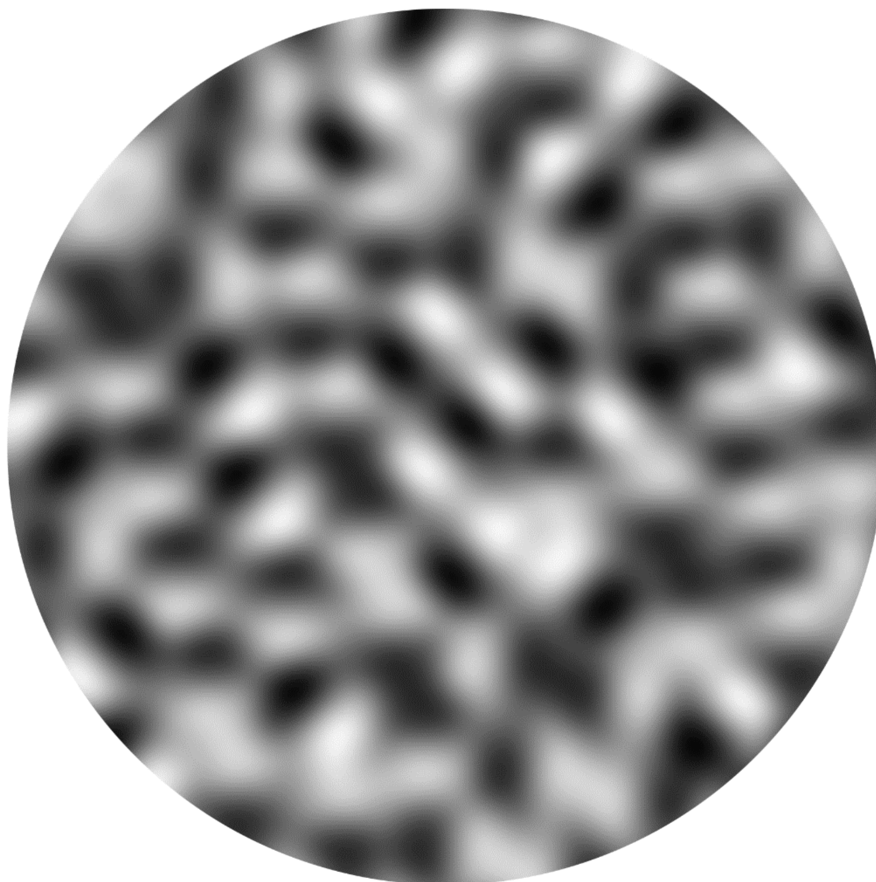

Figura 18. Ruido Simplex. Fuente: Elaboración propia.

Según se puede ver en la representación del ruido de la Figura 18, se trata de un mapa bidimensional circular donde cada punto del espacio tiene asignado un valor comprendido entre el 0 y el 1. En la representación visual, el 0 está representado por el color negro, y el uno, por el color blanco.

Puesto que tanto si se programan las estrellas para generarse en los espacios blancos, como si se hace para hacerlo en los espacios negros, pueden quedar burbujas de estrellas inalcanzables sin métodos ortodoxos, este ruido aún debe ser procesado.

El objetivo final es conseguir un filamento de estrellas que parezca orgánico, pero que a su vez sea entretenido como mapa de juego. Por lo tanto, se ha usado el espacio de cambio entre el color blanco y el color negro.

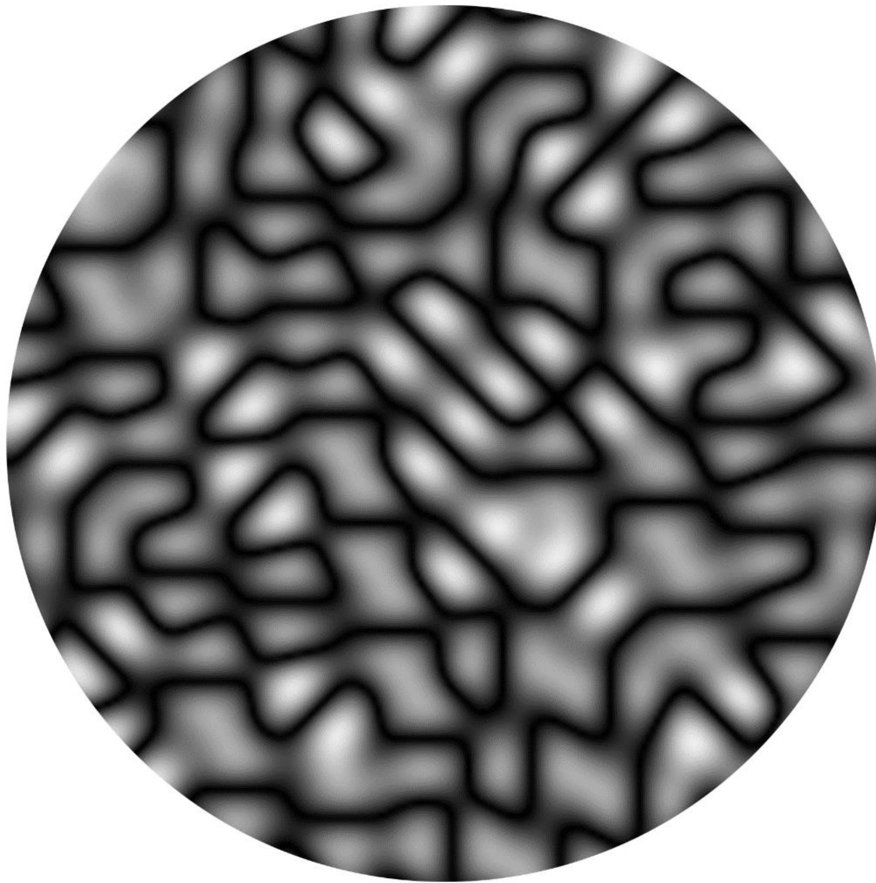

Figura 19. Ruido generando filamentos. Fuente: Elaboración propia.

Como se puede observar en la Figura 19, donde hubo color blanco o negro, ahora hay tonos claros y, en medio de ambos valores, ahora tenemos una zona estable de color negro. Si se procesa más esta imagen, y siguiendo la analogía de los colores blanco y negro, cambiando el umbral en el que se dibuja el color negro, conseguimos algo como el siguiente diagrama.

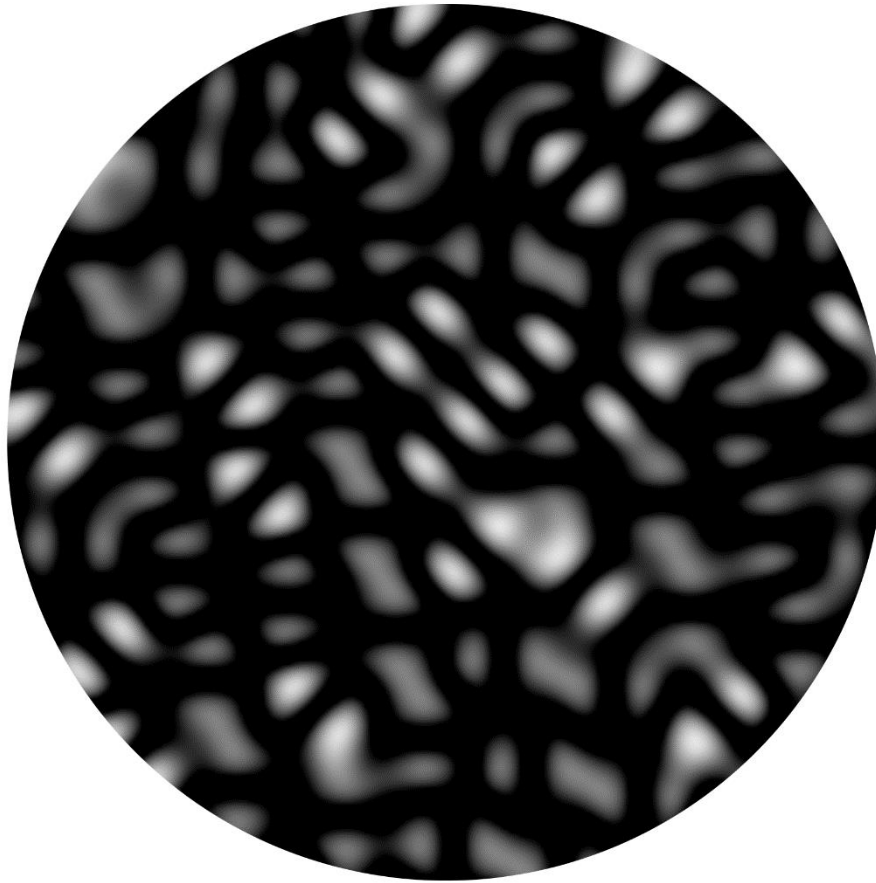

Figura 20. Ruido procesado. Fuente: Elaboración propia.

Una vez se tiene un grafo bidimensional con estas características, el siguiente paso es empezar a generar las posiciones donde las estrellas se deben encontrar. Para ello, se escoge un punto al azar dentro del espacio en el que se trabaja, y si en ese punto, el grafo tiene 'un color negro', allí se genera una estrella. De forma contraria, se prueba con otro punto aleatorio. Se continúa con el proceso hasta que se han generado el número de estrellas deseadas.

Antes de que se genere una estrella, el programa comprueba que ninguna otra estrella esté demasiado cerca de la pretendiente, para que no haya superposición. Si esto ocurre, la estrella por generar, se descarta.

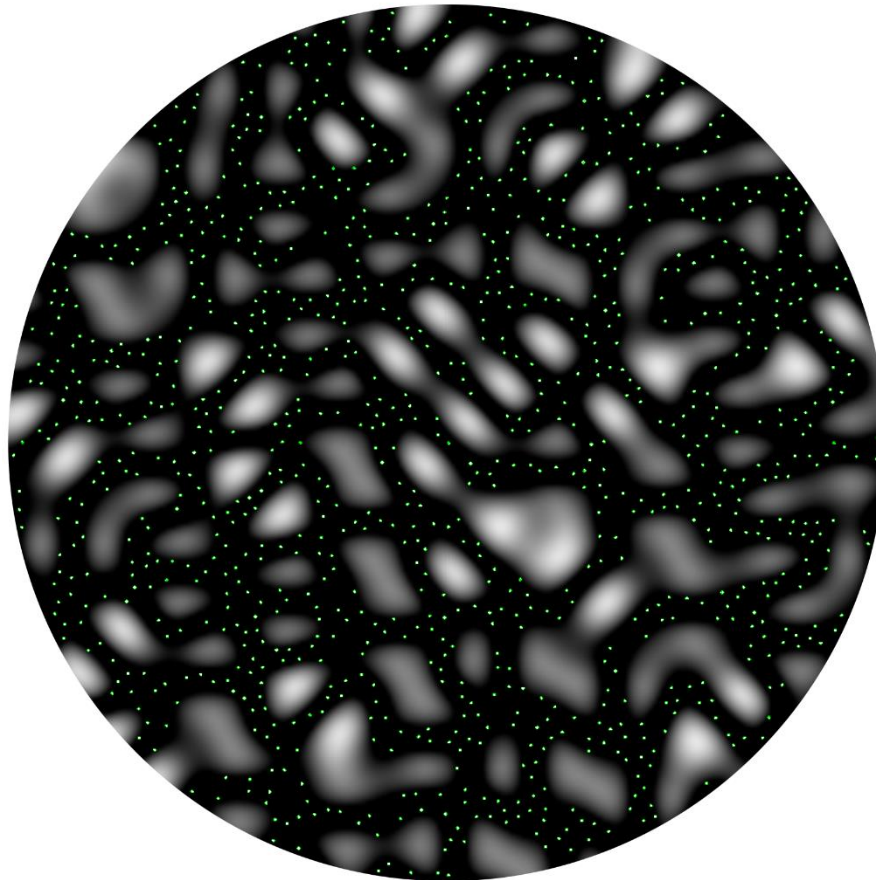

Figura 21. Estrellas colocadas en el mapa de ruido. Fuente: Elaboración propia.

La Figura 21 es una composición donde se ha superpuesto el anterior grafo (Figura 20) con las estrellas generadas usando esa misma distribución. El color de estas se ha variado para mejor visibilidad.

El resultado es cercano al deseado. Por último, para crear un mapa algo más orgánico, se ha aleatorizado un poco el descarte de estrellas en posiciones blancas. Se ha utilizado el degradado que otorga el ruido simplex y se ha creado la posibilidad de que las estrellas se generen en zonas donde el valor del ruido sea distinto a cero, de manera que ocasionalmente aparecerán estrellas donde no deberían según el mapa de ruido.

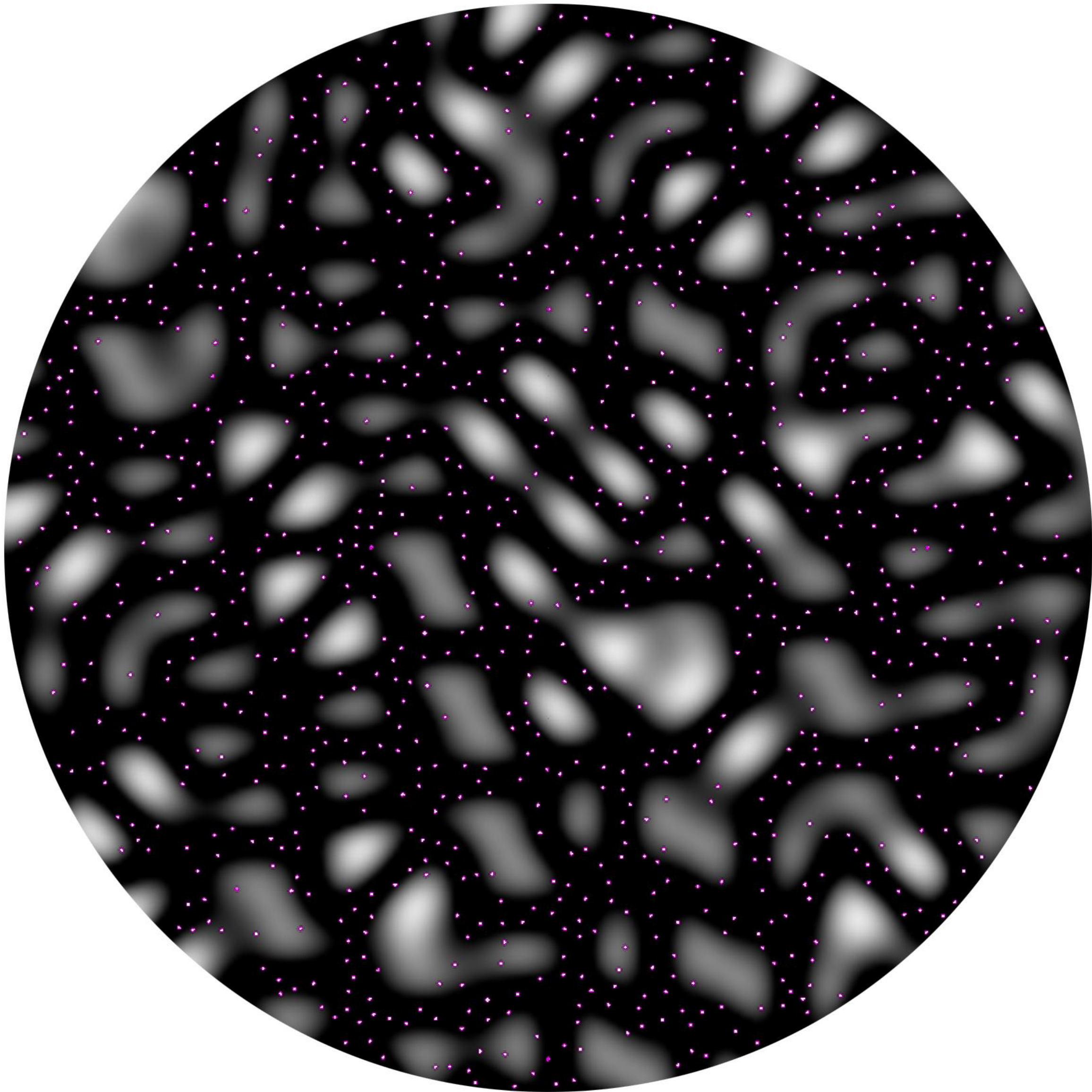

Figura 22. Estrellas orgánicamente colocadas en el mapa de ruido. Fuente: Elaboración propia.

Como se puede observar en la Figura 22, las estrellas forman estructuras filamentosas, visualmente placenteras, de carácter orgánico, e interesantes para crear un mapa dónde las conexiones abren varias posibilidades de triangularidad al jugador.

8.7.2. Representación Visual del Mapa Estelar

Una vez el servidor envía el mapa estelar al cliente cada vez que este carga una partida, el programa carga la escena donde el mapa es visualizado y coloca estrella a estrella en un plano. Estas estrellas son esferas tridimensionales con un material no afectado por la luz, de forma que la escena no necesita de iluminación. En un futuro, este material puede substituirse por un shader que varíe y titile para crear una sensación de cielo nocturno.

La cámara es colocada en la posición en la que el jugador la dejó por última vez, o, en su defecto, en una posición intermedia entre las estrellas que este puede observar en pantalla.

El resultado final es el que se puede observar en la Figura 23.

Figura 23. Representación visual del mapa estelar. Fuente: Elaboración propia.

8.8. Generación del Mapa Planetario

La generación del mapa planetario es una función en la que el frontend es mucho más importante.

Por la parte del servidor, los planetas se generan a la vez que lo hace el mapa estelar. De forma similar a las estrellas, existe una clase planeta, cuyo constructor genera proceduralmente si este no recibe parámetros de entrada. Como solo existen dos tipos de planetas, los parámetros para la generación procedural son más simples que los de las estrellas.

Al momento de su creación, a cada estrella se le asigna un número de objetos órbita dependiendo de los parámetros de su tipo de estrella. Este número oscila de uno en los agujeros negros, a doce en las enanas rojas. Cada una de estas órbitas tiene una probabilidad de una entre tres de generar en ella un planeta.

La clase planeta como tal, genera las siguientes variables proceduralmente: nombre, tipo de planeta, masa y radio. Estos son asignados a su órbita, y ésta a la vez a su estrella.

Todos estos planetas se guardan en la base de datos, generando en el proceso muchos documentos enlazados entre sí mediante listas de identificadores.

Por parte del cliente, este debe mostrar la representación de los planetas alrededor de su estrella, así como el menú del planeta, que contiene información de este, junto a las herramientas para acceder al contenido del planeta, tales como la lista de tecnologías disponibles y por investigar, o los distintos edificios, como el hangar, o los generadores de energía.

Dentro del cliente, en la escena del mapa estelar, cuando se pulsa encima de una estrella, el cliente cambia de escena a la vista del sistema planetario. El juego muestra una representación de la estrella, rodeada de los planetas que giran a su alrededor. El usuario puede mover la cámara rotándola en el eje vertical, de la misma manera que rotan los planetas, para poder ver claramente todos estos cuerpos, incluyendo los que quedan detrás de la estrella. Para facilitar al usuario el reconocimiento de tales cuerpos celestes,

se dibuja en pantalla una representación de la órbita circular por donde el planeta viaja, alrededor de su sol.

Figura 24. Captura del sistema planetario. Fuente: Elaboración propia.

Una vez en esta escena, el usuario puede pulsar encima de cada uno de los planetas, para que le muestre la información necesaria.

En el caso de los planetas deshabitados, se muestra únicamente la información básica, como la masa o la energía. Contrariamente, en los planetas con tecnologías, edificios y/o naves, se muestra mucha más información, como se muestra en la Figura 25.

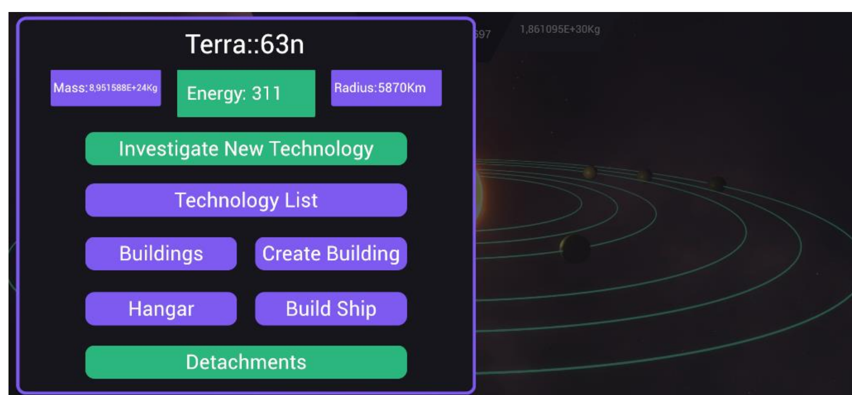

Figura 25. Carta de planeta. Fuente: Elaboración propia.

Los detalles sobre a dónde llevan los distintos botones se cubren en la sección 8.10, acerca de las tecnologías y los edificios.

8.9. Turnos de Juego y Reglas de Juego

Los turnos de juego son el fundamento de la jugabilidad de este proyecto. Por lo tanto, la implementación ha sido desarrollada pensando en la sencillez y la escalabilidad.

Como ha sido planeado junto a la base de datos, cuando un usuario termina su turno, lo envía al servidor, y este lo almacena en la base de datos, serializado en forma de cadena de caracteres. Cuando todos los jugadores han terminado su turno, el servidor procesa los turnos de todos los usuarios, y actualiza el estado del juego para que cuando los jugadores vuelvan a entrar al juego, puedan realizar su siguiente turno.

Cuando el servidor recibe un turno, no lo lee. Lo almacena directamente en la base de datos, a menos que ya haya tantos turnos como jugadores. En ese caso, los descarga todos, los procesa, y los elimina de la base de datos.

Para procesarlos primero hay que parsear el texto que han enviado los distintos clientes, en formato json, a distintas clases serializables. La primera clase almacena información global, como el usuario al cual pertenece el turno, y una colección de acciones.

Dichas acciones se han guardado usando 'actionInterface' (Figura 26), una interfaz de la cual extienden otras varias interfaces, una por cada tipo de acción realizable. Esta interfaz padre contiene una variable llamada 'code' que indica el tipo de acción realizada mediante un código numérico.

Una vez este sistema está organizado, procesar cada turno es simple. El programa solo tiene que iterar por los distintos turnos hechos por todos los jugadores, comprobar que los datos son correctos, y realizar la lógica de estos.

Por último, deberá realizar cálculos generales de final de turno, como por ejemplo la generación de energía por turno.

```
export interface actionListInterface {
    gameId: string,
    userId: string,
    actionList: actionInterface[],
};

export interface actionInterface {
    code: number,
};

export interface moveShip extends actionInterface {
    shipId: string,
    targetStarId: string,
};

export interface changeStarName extends actionInterface {
    starId: string,
    newName: string,
};
```

Figura 26. ActionInterface. Fuente: Elaboración propia.

Desde el cliente, se ha creado un objeto usando el patrón 'singleton' que servirá de controlador de las acciones que toma el jugador. Este objeto contiene una lista con todas las acciones que el usuario va ejecutando durante su turno.

Estas acciones se guardan usando 'ActionInterface', una clase abstracta que mimetiza las interfaces que llevan al mismo nombre en el backend.

Usando estas clases abstractas, el controlador de acciones llena la lista a la vez que controla que las acciones que puede ejecutar el jugador son legales dentro de las normas del juego.

Cuando el jugador pulsa el botón de finalizar el turno, el cliente serializa la lista en formato json, y la envía al servidor, para ser almacenada en la base de datos hasta que sea necesaria.

El servidor no permite enviar otro turno si el usuario en cuestión ya ha enviado un turno durante la ronda actual.

8.10. Tecnologías y Edificios

El trato del juego para las tecnologías y los edificios va de la mano. Las tecnologías son el contenido de este juego. Aunque en el prototipo solo las más básicas están presentes, esta base ha sido desarrollado de forma escalable, para que sea sencillo implementar más tecnologías en cualquier momento.

Cada tecnología equivale a un edificio o módulo que ofrece una ventaja activa o pasiva al jugador. Por lo tanto, todas las tecnologías tienen una función.

Desde el servidor se ha creado una colección de clases que extienden unas de otras, de modo que las funcionalidades de las clases padres puedan usarse por las hijas sin riesgo a incoherencias, sin la necesidad de repetir código, y sin perder rendimiento en el proceso. Además, algunas clases sirven para identificar grandes grupos de tecnologías.

Se ha desarrollado un método que implementa el patrón factoría [12], para gestionar la creación de tecnologías, edificios o módulos. Este método puede ser llamado por dos razones distintas, principalmente.

La primera es la reconstrucción de una tecnología. En ese caso, el método se puede llamar con el nombre de la tecnología en los parámetros, y este devuelve la tecnología solicitada.

La segunda es la generación de tecnologías pseudoaleatoria. Los parámetros en este caso son dos, opcionales ambos, el nivel de la tecnología y la disciplina de esta. El método devuelve una tecnología aleatoria que cumpla los requisitos solicitados en los parámetros.

En el almacenamiento de naves y planetas, las tecnologías, módulos y edificios se representan con una simple cadena de caracteres, que indican el nombre de la tecnología que representan. Cuando los datos de la tecnología necesitan ser usados, como, por ejemplo, durante el procesamiento de turnos, se usa a la factoría de tecnologías para crear un objeto con esos datos.

8.11. Las Estrellas visualmente

Las estrellas usan un material basado en un shader. Este shader está generado juntando y procesando varias instancias de ruido.

Este shader ha sido desarrollado utilizando el Shader Graph de Unity.

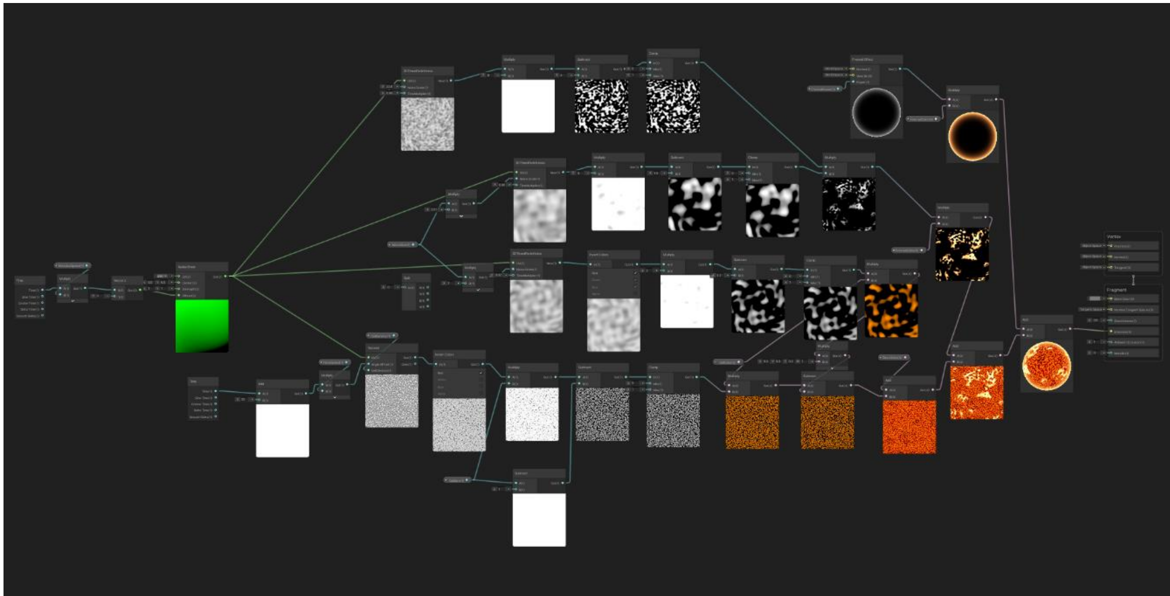

Figura 27. El Shader de la estrella en Shader Graph. Fuente: Elaboración propia.

La base del shader es un ruido de voronoi [14]. Este ruido está formado por un número arbitrario de puntos negros en un espacio blanco. El ángulo varía levemente con el tiempo, por lo que el ruido es dinámico.

Este ruido es invertido para transformarlo en puntos blancos en un fondo gris, y se trata para que ese fondo sea negro. Entonces se multiplica por un color base, por defecto anaranjado, que tiñe los puntos blancos de naranja. Este efecto simula las manchas solares más pequeñas.

A esto se le substrae otro mapa de ruido, esta vez simple, pero tridimensional, donde las primeras dos dimensiones se asignan a las UV del material, y la tercera al tiempo. De esta manera parece que el ruido evolucione con el tiempo.

Antes de la substracción, este ruido ha sido tratado para que sea, en su mayoría, un fondo negro, con manchas blancas que aparecen, se mueven un poco, y desaparecen al poco. Estas manchas son mucho más grandes que los puntos del ruido de voronoi.

Al substraer este mapa, se están generando zonas oscuras en la primera textura, de modo que las manchas solares ahora pueden tener más de una dimensión de profundidad.

Aparte de esto, se crean dos mapas más de ruido simple, tratados de manera similar al anterior, pero con la diferencia en el tamaño de las manchas. En uno de ellos se hacen manchas mucho más grandes, y en el otro mucho más pequeñas. Además, el tiempo varía estos dos mapas con más lentitud. Cuando se multiplican ambos mapas, la combinación restante es una donde hay grandes manchas blancas formadas de otras manchas más pequeñas. A esta textura se le asigna un segundo color, más brillante que el anterior, y es sumado a la primera textura.

El resultado de la combinación es un mapa que abstrae tanto las manchas solares, como las diferencias de temperatura de la estrella, así como destellos puntuales. Este mapa se asigna a las UVs del material, no sin antes tratarlos para que tengan una estructura radial.

Por último, se usa un nodo de efecto fresnel, que crea una zona alrededor del objeto según la vista de la cámara, y, tiñendo ese efecto de un amarillo, se crea un efecto visual de brillo alrededor de la estrella.

El shader está preparado para variar los ritmos y los colores según el tipo de estrella con la que se trate.

El resultado final es el de la Figura 28.

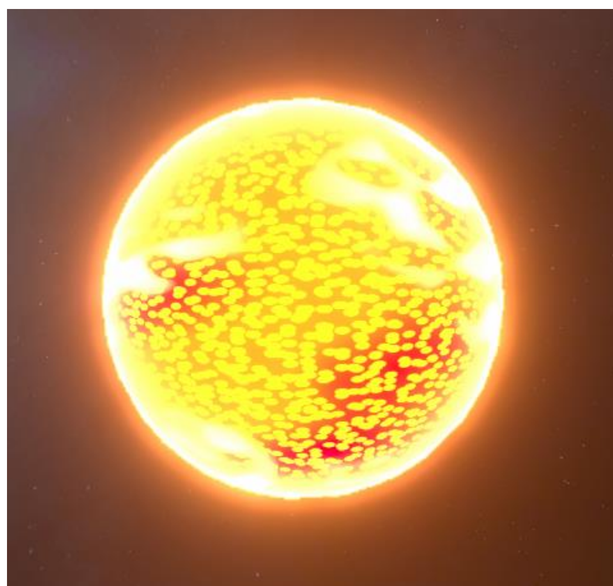

Figura 28. Shader de la estrella. Fuente: Elaboración propia.

9. Conclusiones

A continuación, se exponen las conclusiones a las que se ha llegado después del desarrollo del proyecto.

El primer dato a destacar es que la metodología en espiral ha demostrado funcionar correctamente para lo que se escogió: La evasión de riesgos. Por otro lado, los puntos negativos de esta decisión también tienen su peso. El proyecto no ha llegado a un estado de producto mínimo viable. El estado actual del proyecto es el de una base rígida sobre la que construir el juego deseado, implementado el resto del contenido con facilidad.

Node.js junto a JavaScript han resultado tener una eficacia incluso por encima de la esperada, y la facilidad de desarrollo que han proporcionado, han permitido la experimentación e iteración de las funcionalidades a implementar. Además, su uso junto al de la base de datos de MongoDB ha terminado siendo sencillo e intuitivo, a la vez que perfectamente funcional y eficaz.

Unity, como motor de juego, ha permitido el desarrollo sencillo y organizado de, tanto la interfaz de usuario, como de elementos estéticos del juego, así como las estrellas o los planetas. Cabe destacar que la extensa documentación ha ayudado en gran medida a que no se haya detenido el desarrollo en ningún momento por desconocimiento del medio.

Los objetivos principales han sido completados. El proyecto resultante es un juego básico de turnos asíncronos, con generación de partidas, estrellas y planetas, tecnologías y edificios. El estado del juego es muy prototípico, por lo que no puede usarse para medir la viabilidad del producto en el mercado, algo que quizás otra metodología hubiera conseguido, a cambio de la robustez y escalabilidad de las cuales goza el proyecto.

Finalmente, los objetivos secundarios no han sido alcanzados, debido a restricciones temporales.

10. Posibles Ampliaciones

Como se ha expuesto en el capítulo anterior, el proyecto se encuentra en un estado prototípico. Esto genera que las posibles ampliaciones de este puedan variar mucho, dependiendo del avance de unas posibles futuras iteraciones de la metodología en espiral.

La primera de las ampliaciones fundamentales es la de terminar de implementar los elementos que forman las naves espaciales en el proyecto. Estos no son más que elementos básicos del juego, parecidos a los edificios, pero que deben tener su propia colección en la base de datos, pues estas no siempre se mantienen en las mismas coordenadas.

La otra ampliación fundamental es la regla de juego que amplía la restricción de información que el servidor otorga a cada cliente. Esta característica se ha dejado para una futura implementación, puesto que depende enormemente del contenido que se quiera añadir al juego.

Por último, y el elemento más variable y extenso de todos, es el contenido del juego. El proyecto se ha dejado en un estado en el que implementar contenido nuevo es extremadamente sencillo. Nuevas tecnologías, edificios y módulos solo requieren que el desarrollador implemente dichos elementos en la factoría de tecnologías, y en las reglas de ronda, o de turno. Además, los módulos auxiliares pueden facilitar el proceso significativamente.

Aparte de los elementos funcionales, el proyecto se puede ampliar desde disciplinas distintas a las trabajadas, como la artística, o la social.

11. Bibliografía

[1] J. Cope, «What's a Peer-to-Peer (P2P) Network?,» Computerworld, 8 Abril 2002. [En línea]. Available: https://www.computerworld.com/article/2588287/networking-peer-to-peer-network.html. [Último acceso: 14 Enero 2022].

[2] H. S. Oluwatosin, «Client-Server Model,» *IOSR Journal of Computer Engineering,* vol. 16, nº 1, pp. 67-71, 2014.

[3] O. Sotamaa y J. Svelch, Game Production Studies, 2021.

[4] S. X. Eleftheria Christopoulou, «Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices,» *International Journal of Serious Games,* vol. IV, nº 4, pp. 21-36, 2017.

[5] A. Knezovic, «Mobile Strategy Games 2020 Report — Including Ads, Business & Monetization Tactics,» Udonis.blog, 2020.

[6] ESA, «2019 Essential Facts About the Computer and Video Game Industry,» Entretainment Software Association, 2019.

[7] Centers for Medicare & Medicaid Services, «Selecting a development approach,» 2008.

[8] B. W. Boehm, «A Spiral Model of Software Develpment and Enhancement,» *Computer,* vol. 21, nº 5, pp. 61-72, 1988.

[9] quill18creates, «"Powerful but Simple Game Server with Unity & Node.JS", Youtube,» 28 Mayo 2021. [En línea]. Available: https://www.youtube.com/c/quill18creates. [Último acceso: 11 Junio 2022].

[10] Epitome, «"Build a working Login page with Unity, Node.js and Mongo", Youtube,» 17 Julio 2021. [En línea]. Available: https://www.youtube.com/c/EpitomeGames. [Último acceso: 11 Junio 2022].

[11] D. K. Daniel Dinu, «phc-winner-argon2,» 6 Diciembre 2015. [En línea]. Available: https://github.com/p-h-c/phc-winner-argon2. [Último acceso: 29 Abril 2022].

[12] E. Gamma, Design patterns : elements of reusable object-oriented software, 1995.

[13] K. Perlin, «Noise hardware,» p. 26, 2001.

[14] P. Cozzi y C. Riccio, «Worley Noise,» de *OpenGL Insights*, 2012, p. 113–115.