UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Desarrollo de un juego de estrategia multijugador por turnos

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Wang, Yuntao

Tutor/a: López Patiño, José Enrique

CURSO ACADÉMICO: 2023/2024

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

Agradecimientos especiales

Quisiera expresar mi más profundo agradecimiento a todas las personas que me han brindado su apoyo y asistencia durante el desarrollo de este proyecto. En particular, me gustaría agradecer a CrookedHead, el creador de TBS Framework, por su invaluable ayuda y asesoramiento en Discord. Su conocimiento y disposición para ayudar han sido fundamentales para el éxito de este proyecto.

Resumen

El objetivo del proyecto es desarrollar un juego de estrategia por turnos para dos jugadores a través de una plataforma web. Dicha plataforma gestionará la posibilidad de que múltiples usuarios puedan conectarse a través de ella e iniciar y desarrollar varias partidas al mismo tiempo. La disputa de las partidas no será en tiempo real, sino que los jugadores dispondrán de turnos alternativos en los que gestionarán la disposición de las unidades que se le han proporcionado. Por ello será necesario que la plataforma utilice una base de datos que permita guardar el estado de las partidas dentro de un turno y entre turnos alternativos.

El juego planteará diferentes escenarios, en los que inicialmente los jugadores decidirán qué unidades utilizar para luego, en cada ronda, desplegarlas y ordenarlas según los recursos de que dispongan para lograr el objetivo de victoria de destruir la base del oponente. Cada escenario dispondrá, por tanto, de un número de recursos limitado por el que los jugadores deberán competir para lograr el objetivo final, de manera que tengan que plantear una estrategia personalizada.

Cada jugador dispondrá de unidades de diferente tipo, con características propias en cuanto a capacidad de ataque, capacidad de defensa, movilidad y consumo de recursos. Las tecnologías a emplear en el desarrollo de la plataforma serán todas las relacionadas con desarrollos web dinámicos, incluyendo la de base de datos ya comentada.

RESUMEN EJECUTIVO

La memoria del TFG del Desarrollo de un juego de estrategia multijugador por turnos debe desarrollar en el texto los siguientes conceptos, debidamente justificados y discutidos, centrados en el ámbito de la telematica

CONCEPT (ABET)	CONCEPTO (traducción)	¿Cumple? (S/N)	¿Dónde? (páginas)
1. IDENTIFY:	1. IDENTIFICAR:		
1.1. Problem statement and opportunity	1.1. Planteamiento del problema y oportunidad	si	8
1.2. Constraints (standards, codes, needs, requirements & specifications)	1.2. Toma en consideración de los condicionantes (normas técnicas y regulación, necesidades, requisitos y especificaciones)	si	12-29
1.3. Setting of goals	1.3. Establecimiento de objetivos	si	10
2. FORMULATE:	2. FORMULAR:		
2.1. Creative solution generation (analysis)	2.1. Generación de soluciones creativas (análisis)	si	12-29
2.2. Evaluation of multiple solutions and decision-making (synthesis)	2.2. Evaluación de múltiples soluciones y toma de decisiones (síntesis)	si	12-29
3. SOLVE:	3. RESOLVER:		
3.1. Fulfilment of goals	3.1. Evaluación del cumplimiento de objetivos	si	60-69
3.2. Overall impact and significance (contributions and practical recommendations)	3.2. Evaluación del impacto global y alcance (contribuciones y recomendaciones prácticas)	si	60-69

Índice

Índice de figuras

Índice de código

Capítulo 1. Introducción

En el exigente mundo del desarrollo de videojuegos y las tecnologías de conectividad, la innovación constante y la capacidad de integrar diversas tecnologías son cruciales para mantenerse competitivo. La creación de videojuegos no solo implica creatividad y diseño, sino también una sólida comprensión de las tecnologías subyacentes que permiten su funcionamiento, especialmente en lo que respecta a la comunicación y la interactividad en tiempo real.

El desarrollo de un juego de estrategia por turnos presenta numerosos desafíos técnicos, incluyendo la gestión de recursos, la sincronización de datos entre jugadores y la creación de una experiencia de usuario envolvente. La implementación de estos elementos requiere una combinación de habilidades en programación, diseño de sistemas y una comprensión profunda de los principios de conectividad.

Un ejemplo clásico que ilustra bien estos desafíos es **Panzer Corps**, un juego de estrategia por turnos que permite a los jugadores revivir batallas históricas y desarrollar tácticas militares detalladas. Este juego se centra en la planificación estratégica y el control de unidades en un campo de batalla virtual, ofreciendo una experiencia de juego profundamente inmersiva.

Por otro lado, juegos como **Hearts of Iron** llevan el concepto de los juegos de estrategia por turnos a un nuevo nivel al incorporar la conectividad global. Los jugadores no solo interactúan con una inteligencia artificial sofisticada, sino que también pueden conectarse y competir con otros jugadores a nivel global, haciendo que la experiencia sea más dinámica y competitiva. Este tipo de conectividad permite partidas multijugador en tiempo real, donde la estrategia y la táctica se combinan con la interacción social y la competición.

Ejemplo escena del juego Heart of Iron 4

Ejemplo escena del juego Panzer Corps

El objetivo de este Trabajo de Fin de Grado (TFG) es desarrollar un juego de estrategia por turnos que incorpore un sistema de producción de unidades y gestión de recursos, utilizando plataformas y tecnologías avanzadas como Unity para el desarrollo del juego y Nakama para la gestión de servidores. Estas herramientas no solo proporcionan una base sólida para la creación del juego, sino que también facilitan la implementación de características complejas como la interacción en tiempo real entre jugadores.

A lo largo de este TFG, se abordarán diversos aspectos del desarrollo del juego, incluyendo:

1. **Diseño del sistema de producción de unidades**: Implementación de un sistema flexible y dinámico que permita a los jugadores producir y gestionar sus unidades en función de los recursos disponibles.
2. **Gestión de recursos y puntos de control**: Creación de un sistema de captura de puntos de control en el mapa que proporcione recursos a los jugadores, añadiendo un elemento estratégico clave al juego.
3. **Implementación del sistema de turnos**: Desarrollo de una mecánica de juego que permita a los jugadores tomar turnos de manera ordenada y estratégica, asegurando un equilibrio entre ataque y defensa.
4. **Interfaz de usuario y experiencia de juego**: Diseño de una interfaz intuitiva y atractiva que mejore la experiencia del jugador y facilite la interacción con el juego.

El éxito de este proyecto dependerá de la capacidad de integrar estas diversas áreas de desarrollo y de superar los desafíos técnicos asociados con la sincronización de datos y la gestión de recursos en tiempo real. A través de este TFG, se demostrará la aplicabilidad de los principios de la conectividad en el desarrollo de videojuegos, destacando la importancia de una sólida base técnica para la creación de experiencias de juego innovadoras y atractivas.

En conclusión, este TFG no solo se propone desarrollar un juego de estrategia por turnos, sino también explorar y demostrar cómo las tecnologías de conectividad pueden mejorar significativamente la interactividad y la experiencia del usuario en los videojuegos. La combinación de estos campos ofrece una oportunidad única para innovar y crear productos de entretenimiento de alta calidad en un mercado cada vez más competitivo y tecnológicamente avanzado.

Capítulo 2. Objetivos

Este Trabajo de Fin de Grado se centra en el desarrollo de un juego de estrategia por turnos para dos jugadores, accesible a través de una plataforma web. El objetivo principal de este proyecto es demostrar cómo se pueden combinar técnicas de desarrollo de videojuegos con tecnologías de conectividad para crear una experiencia de usuario envolvente y técnicamente robusta.

Este objetivo principal se puede desglosar en los siguientes objetivos específicos:

1. **Introducción a la Importancia de la Integración de Tecnologías de Conectividad en los Videojuegos**:
    - Destacar cómo las tecnologías de conectividad pueden mejorar la interacción y la experiencia del usuario en los videojuegos, permitiendo una comunicación fluida y en tiempo real entre los jugadores.
2. **Diseño del Sistema de Producción de Unidades y Gestión de Recursos**:
    - Describir el proceso de diseño e implementación de un sistema de producción de unidades y gestión de recursos dentro del juego, asegurando que sea dinámico y balanceado para mantener el interés del jugador.
3. **Desarrollo de la Mecánica de Turnos**:
    - Explicar cómo se diseñará e implementará la mecánica de turnos, garantizando que sea intuitiva y permita una estrategia compleja, equilibrando el juego para ambos jugadores.
4. **Implementación y Configuración de Nakama para la Gestión de Servidores**:
    - Mostrar cómo configurar y utilizar Nakama para la gestión de servidores, permitiendo la sincronización de datos en tiempo real y la interacción entre jugadores a través de una infraestructura sólida.
5. **Diseño e Implementación de la Interfaz de Usuario (UI)**:
    - Detallar el proceso de diseño e implementación de una interfaz de usuario intuitiva y atractiva, que facilite la interacción del jugador con el juego y mejore la experiencia general.
6. **Pruebas Funcionales y Evaluación de la Experiencia de Usuario**:
    - Explicar cómo se diseñarán y ejecutarán las pruebas funcionales para asegurar que todas las características del juego funcionen correctamente y cómo se evaluará la experiencia del usuario a través de pruebas con jugadores reales.
7. **Documentación Detallada del Proceso de Desarrollo**:
    - Proporcionar una documentación exhaustiva de cada fase del desarrollo del juego, incluyendo ejemplos, capturas de pantalla y explicaciones detalladas, para facilitar la replicación del proyecto y el aprendizaje de futuros desarrolladores.
8. **Énfasis en la Calidad y la Seguridad de la Aplicación**:

- Subrayar la importancia de mantener altos estándares de calidad y seguridad durante todo el proceso de desarrollo, implementando mejores prácticas para la codificación, pruebas y despliegue del juego.

9. **Evaluación de Resultados y Propuestas de Mejora**:
   - Evaluar la efectividad de las estrategias implementadas y proponer posibles mejoras y futuras expansiones del proyecto, basadas en la retroalimentación de los usuarios y en el análisis de las pruebas realizadas.

Capítulo 3. Herramientas y tecnología

En este capítulo se explicarán las diferentes herramientas utilizadas a lo largo del proyecto, detallando el propósito de cada una y la razón por la cual se eligieron. Este proyecto se divide en varias partes, cada una de las cuales requiere herramientas específicas para cumplir sus objetivos:

- Plataforma de desarrollo del juego
- Gestión y sincronización de servidores
- Contenerización y despliegue del entorno

Teniendo en cuenta estos tres puntos, comenzaremos por el primero.

3.1 Plataforma de desarrollo

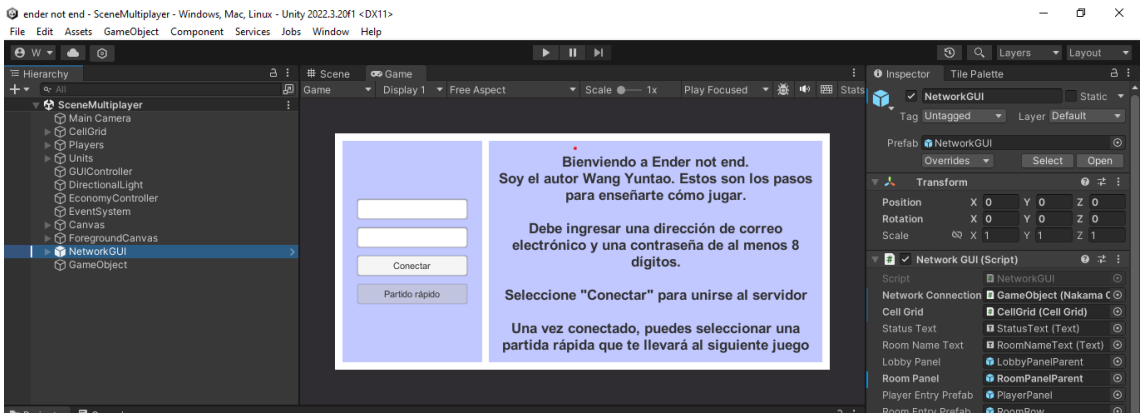
En esta primera parte, es esencial definir las herramientas que se utilizarán para el desarrollo del juego, desde el motor de juego hasta las herramientas de gestión de servidores.

3.1.1 Unity

Unity es una plataforma de desarrollo de juegos ampliamente utilizada, conocida por su potente soporte multiplataforma y sus extensos recursos comunitarios. La elección de Unity para este proyecto se basa en varias razones clave:

- **Soporte Multiplataforma:** Unity permite el desarrollo de juegos que pueden ejecutarse en múltiples plataformas, incluyendo PC, consolas y dispositivos móviles.
- **Recursos Abundantes:** La tienda de Unity ofrece una vasta colección de recursos 2D y 3D, que facilitan el desarrollo de gráficos y otros assets del juego.
- **Facilidad de Uso:** Unity proporciona una interfaz de usuario intuitiva y herramientas de desarrollo robustas, lo que acelera el proceso de creación y prueba del juego.

Unity será la base sobre la cual se desarrollará todo el contenido interactivo del juego, incluyendo la lógica de juego, la interfaz de usuario y las interacciones de los jugadores.

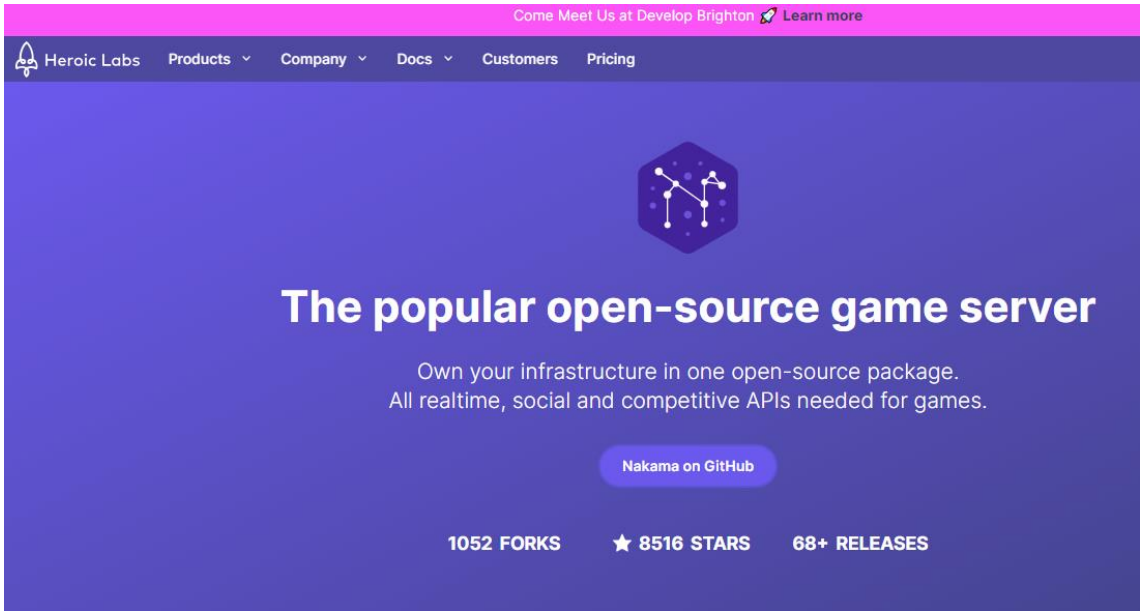


Escena de trabajo de Unity

3.1.2 Nakama y Docker

Nakama es una solución de backend para juegos que proporciona una serie de funcionalidades esenciales para la gestión de juegos multijugador. La integración de Nakama con Docker permite una implementación eficiente y escalable. Las características destacadas de Nakama y Docker son:

- **Gestión de Jugadores:** Nakama permite la autenticación y gestión de jugadores, así como la creación y administración de sesiones de juego.
- **Sincronización en Tiempo Real:** Facilita la sincronización de datos en tiempo real entre jugadores, crucial para la experiencia multijugador.
- **Contenerización con Docker:** Docker permite empaquetar Nakama y todas sus dependencias en un contenedor, asegurando que se ejecuten de manera consistente en cualquier entorno.
- **Facilidad de Despliegue:** Docker simplifica el proceso de despliegue y gestión de aplicaciones, haciendo que la configuración sea más manejable y reproducible.
- **Escalabilidad:** Ambas herramientas juntas facilitan la escalabilidad del juego, permitiendo manejar un gran número de jugadores simultáneamente.



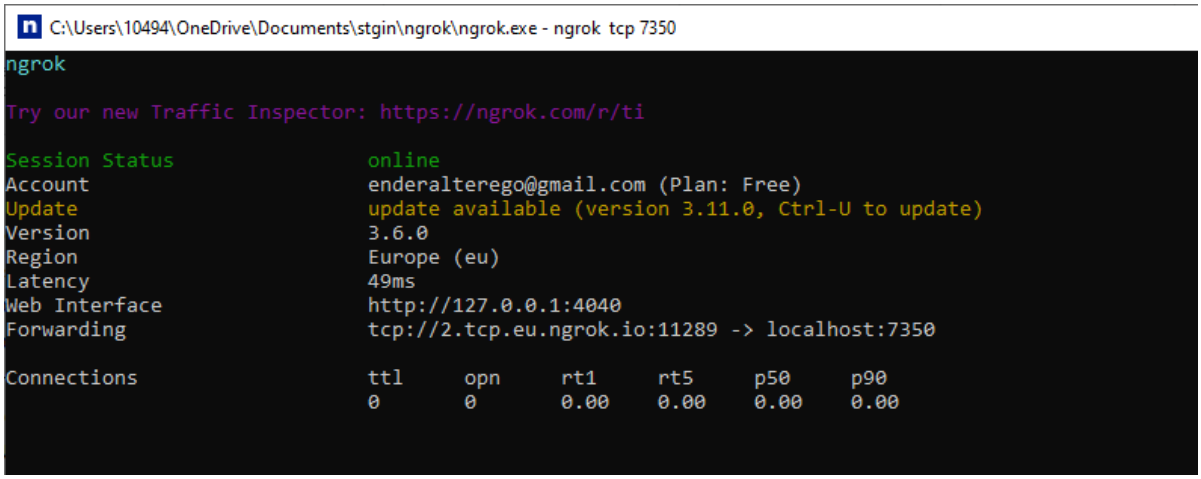
Nakama server

3.1.3 Ngrok

Ngrok es una herramienta que permite crear túneles seguros hacia servicios locales, facilitando el acceso a aplicaciones en desarrollo desde cualquier lugar. Su inclusión en este proyecto se debe a las siguientes ventajas:

- **Acceso Remoto**: Ngrok permite exponer el servidor local Nakama, que corre en `localhost:7350`, a través de internet. Esto facilita que cualquier persona, desde cualquier lugar, pueda acceder y jugar al juego mediante una URL pública proporcionada por Ngrok.
- **Seguridad**: Proporciona túneles seguros, protegiendo la comunicación y los datos transferidos, asegurando que las pruebas y el acceso remoto se realicen de manera segura.
- **Facilidad de Uso**: Ngrok es fácil de configurar y usar, lo que acelera el proceso de desarrollo y pruebas al permitir una rápida exposición de servicios locales al internet sin necesidad de configuraciones complicadas.
- **Gratuito**: Ngrok ofrece un plan gratuito que permite su uso sin costo alguno. Aunque cada vez que se reinicia el servidor se requiere configurar una nueva dirección, su estabilidad y fluidez durante el uso son sobresalientes.

Al utilizar Ngrok, se logra que el juego desarrollado pueda ser accesible en cualquier momento y desde cualquier lugar, facilitando tanto las pruebas como la colaboración con otros desarrolladores y jugadores.



escenario de ejecución de ngrok

3.2 Recursos gráficos

En esta sección se explicarán los recursos gráficos utilizados para el desarrollo del juego, incluyendo la obtención y la integración de estos recursos en el proyecto. Estos recursos son fundamentales para crear una experiencia visual atractiva y coherente.

3.2.1 Tienda de Unity

La Tienda de Unity (Unity Asset Store) es una plataforma donde los desarrolladores pueden comprar o descargar gratuitamente una amplia variedad de assets, incluyendo gráficos, sonidos, scripts y más. La elección de la Tienda de Unity se debe a varias razones:

- **Variedad de Recursos:** Ofrece una extensa colección de assets 2D y 3D que cubren una amplia gama de estilos y necesidades.
- **Calidad:** Los assets disponibles en la Tienda de Unity son de alta calidad y están listos para ser integrados en los proyectos, lo que ahorra tiempo y esfuerzo en el desarrollo.
- **Licencias:** Los assets adquiridos en la Tienda de Unity vienen con licencias claras que permiten su uso legal en proyectos comerciales, garantizando el cumplimiento de las leyes de derechos de autor.

Durante el desarrollo de este juego de estrategia por turnos, se utilizarán principalmente recursos 2D para crear personajes, escenarios y otros elementos gráficos que sean necesarios para la experiencia de juego.

Recursos que compré

3.2.2 Integración de assets

La integración de assets, que en el contexto del desarrollo de videojuegos se refiere a cualquier recurso digital utilizado en el juego, como gráficos, sonidos, modelos 3D, texturas y animaciones, es un proceso crucial para asegurar que todos los elementos gráficos se alineen con la visión del juego y funcionen correctamente dentro del motor de Unity.

Este proceso incluye:

- **Importación de Assets:**
  - Los assets descargados desde la Tienda de Unity se importarán al proyecto de Unity. Esto se hace fácilmente a través de la interfaz del motor, permitiendo a los desarrolladores arrastrar y soltar assets directamente en la escena del juego.
- **Configuración y Adaptación:**
  - Una vez importados, los assets pueden necesitar ajustes para adaptarse al estilo y la mecánica del juego. Esto puede incluir la modificación de sprites, la configuración de animaciones y la adaptación de propiedades físicas.
  - **Modificación de Sprites:** Ajustar las imágenes 2D para que se alineen correctamente con otros elementos del juego.
  - **Configuración de Animaciones:** Establecer cómo se mueven y se comportan los modelos y sprites dentro del juego.
  - **Adaptación de Propiedades Físicas:** Configurar elementos como colisiones, gravedad y otros aspectos físicos para que se comporten de manera realista o adecuada dentro del entorno del juego.
- **Optimización:**
  - Para asegurar un rendimiento óptimo, los assets serán optimizados. Esto incluye la reducción del tamaño de los archivos gráficos sin perder calidad visual, la organización eficiente de los recursos y la gestión de la memoria.
  - **Reducción de Tamaño:** Minimizar el tamaño de los archivos gráficos y de sonido para mejorar los tiempos de carga y el rendimiento general.
  - **Organización de Recursos:** Mantener una estructura de carpetas y nomenclatura claras y organizadas para facilitar la gestión de los assets.
  - **Gestión de la Memoria:** Asegurarse de que los assets se carguen y descarguen de manera eficiente para no agotar los recursos del sistema.
- **Testing y Refinamiento:**
  - Después de integrar los assets, se realizarán pruebas para asegurar que se comporten correctamente en el juego. Cualquier problema detectado será corregido, y los assets serán refinados según sea necesario.
  - **Pruebas de Integración:** Verificar que los assets funcionen correctamente dentro del juego, interactuando bien con otros elementos y sistemas.

- **Corrección de Problemas:** Resolver cualquier conflicto o error que surja durante las pruebas de integración.
- **Refinamiento:** Ajustar los assets y su implementación para mejorar la calidad visual y el rendimiento del juego.

En resumen, la integración de assets no solo implica añadir recursos gráficos y sonoros al proyecto de Unity, sino también asegurarse de que estos recursos se ajusten y funcionen de manera óptima dentro del juego, proporcionando una experiencia de usuario fluida y visualmente coherente.

3.3 Servidores y bases de datos

En esta sección se explicará la configuración del servidor Nakama y cómo se maneja la comunicación entre el cliente (el juego desarrollado en Unity) y el servidor, utilizando varios archivos de código esenciales para la implementación del backend del juego.

3.3.1 Configuración de Nakama

Nakama es un backend para juegos que permite gestionar jugadores, sesiones de juego y la sincronización en tiempo real. Para configurar Nakama y asegurarnos de que funcione correctamente con nuestro juego, seguiremos los siguientes pasos:

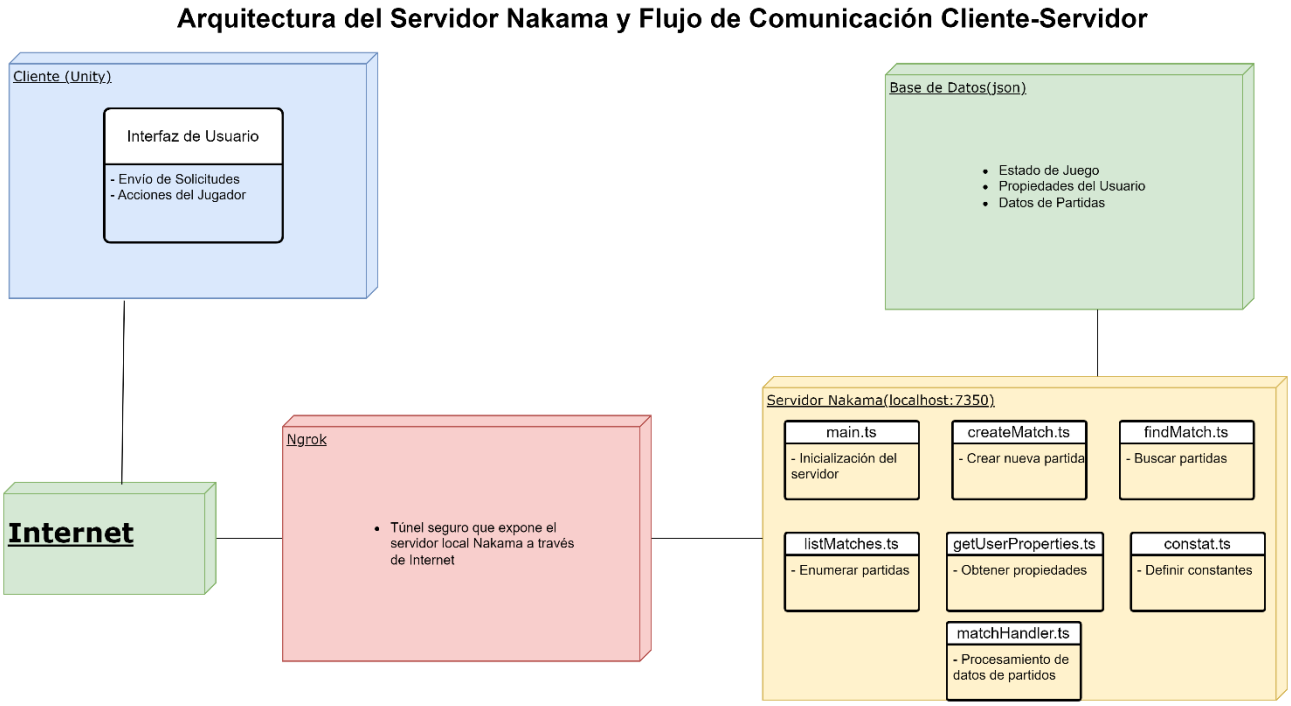


Diagrama de Flujo del Servidor Nakama y Comunicación Cliente-Servidor

Flujo de Datos y Pasos:

1. **Inicialización del Cliente:**
   - El usuario inicia el juego en el cliente de Unity, ingresa su correo electrónico y contraseña, y se conecta al servidor Nakama mediante un botón.
2. **Servidor Nakama (main.ts):**
   - **Inicialización del Servidor:**
     - El servidor Nakama se inicializa, configurando los módulos y controladores necesarios para manejar las solicitudes del cliente.
     - En `main.ts`, se inicializa el módulo y se registran las llamadas RPC y las funciones de manejo de partidas.
3. **Crear y Unirse a Partidas:**
   - **Crear Partida (createMatch.ts):**
     - El jugador solicita la creación de una nueva partida desde el cliente, el servidor recibe la solicitud y llama a la función en `createMatch.ts`.
     - El servidor inicializa los datos de la partida en la base de datos y devuelve el ID de la partida al cliente.
   - **Buscar Partida (findMatch.ts):**
     - El jugador puede buscar partidas existentes mediante una solicitud desde el cliente, el servidor llama a `findMatch.ts` para manejar la solicitud y devuelve el ID de la partida correspondiente.
   - **Enumerar todas las partidas (listMatches.ts):**
     - El cliente solicita una lista de todas las partidas disponibles, el servidor llama a `listMatches.ts` y devuelve la lista de partidas.
4. **Gestión de Partidas (matchHandler.ts):**
   - **Unirse a la partida (matchJoinAttempt, matchJoin):**
     - Un nuevo jugador solicita unirse a una partida, el servidor llama a `matchJoinAttempt` para verificar la solicitud y, si es válida, llama a `matchJoin` para agregar al jugador a la partida.
   - **Bucle de la partida (matchLoop):**
     - El servidor mantiene el estado de la partida y, mediante `matchLoop`, maneja cada tick del juego, recibiendo y procesando las acciones de los jugadores.
   - **Dejar la partida (matchLeave):**
     - Un jugador deja la partida, el servidor llama a `matchLeave` para actualizar el estado de la partida y manejar la lógica después de la salida del jugador.
   - **Terminar la partida (matchTerminate):**
     - Al finalizar la partida, el servidor llama a `matchTerminate` para limpiar los datos de la partida.
5. **Gestión de Jugadores y Propiedades:**
   - **Obtener propiedades del usuario (getUserProperties.ts):**

        - El servidor lee las propiedades de un usuario específico en una partida específica desde la base de datos y las devuelve al cliente.
    - **Manejo del estado del jugador (handlePlayerReadyChanged, handlePlayerNumberChanged):**
        - El servidor maneja los cambios en el estado de preparación y número de los jugadores, actualiza el estado de la partida y lo transmite a los otros jugadores.
6. **Almacenamiento y Recuperación de Datos:**
    - **Definición de Constantes (constat.ts):**
        - Define las constantes globales necesarias para la configuración del servidor, como identificadores únicos y nombres de colecciones.
    - **Almacenamiento de Datos:**
        - Los datos de las partidas, jugadores y estados se almacenan en la base de datos y se gestionan mediante la API de almacenamiento de Nakama.

7. **Integración de ngrok:**

- **Uso de ngrok para Exponer el Servidor Nakama:**
    - Ngrok se utiliza para exponer el servidor Nakama local al Internet, facilitando que jugadores remotos puedan acceder al servidor.
    - El servidor Nakama, que se ejecuta en localhost:7350, se expone a través de una URL pública proporcionada por ngrok.
    - Los jugadores pueden conectarse a esta URL pública para interactuar con el servidor Nakama desde cualquier ubicación, mejorando la accesibilidad y permitiendo pruebas remotas.

Código de Ejemplo:

Constantes Globales (constat.ts): Este archivo contiene las constantes necesarias para la configuración del servidor, tales como identificadores únicos y nombres de colecciones. Definir estas constantes ayuda a mantener la configuración organizada y fácil de modificar.

```
/**
 * Nakama服务器脚本的全局常量。
 *
 * UidSystem: 用于系统级操作的全局唯一标识符。
 * CollectionRoomNameToMatchId: 用于将房间名称映射到比赛ID的集合名称。
 * CollectionMatchIdToRoomName: 用于将比赛ID映射到房间名称的集合名称。
 * CollectionMatchUserProperties: 用于存储比赛中用户属性的集合名称。
 */

const UidSystem = "00000000-0000-0000-0000-000000000000";
const CollectionRoomNameToMatchId = "RoomNameToMatchId";
const CollectionMatchIdToRoomName = "MatchIdToRoomName";
const CollectionMatchUserProperties = "matchUserProperties";
```

Inicialización del Módulo Principal (main.ts): Este archivo es el punto de entrada principal para la aplicación del servidor. Aquí se inicializa el servidor Nakama y se configuran los controladores necesarios para manejar las conexiones y las solicitudes de los clientes.

```typescript
let InitModule: nkruntime.InitModule = function (ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, initializer:
  nkruntime.Initializer) {
  // 注册比赛模块，'lobby' 为比赛的名称
  initializer.registerMatch('lobby', {
      matchInit,       // 比赛初始化
      matchJoinAttempt,// 尝试加入比赛
      matchJoin,       // 加入比赛
      matchLeave,      // 离开比赛
      matchLoop,       // 比赛循环
      matchSignal,     // 比赛信号
      matchTerminate,  // 终止比赛
  });

  // 注册匹配成功后的处理函数
  initializer.registerMatchmakerMatched(matchmakerMatched);

  // 注册远程过程调用（RPC）函数
  initializer.registerRpc("rpcCreateCustomMatch", rpcCreateCustomMatch);  // 创建自定义比赛的RPC
  initializer.registerRpc("rpcFindCustomMatch", rpcFindCustomMatch);      // 查找自定义比赛的RPC
  initializer.registerRpc("rpcListMatches", rpcListMatches);              // 列出所有比赛的RPC
  initializer.registerRpc("rpcGetUserProperties", rpcGetUserProperties);  // 获取用户属性的RPC
}
```

Crear una Partida Personalizada (createMatch.ts): Este archivo maneja la lógica para crear una nueva partida en el servidor. Es responsable de inicializar los datos de la partida y registrar la partida en el servidor

```typescript
/**
 * 在Nakama服务器中创建自定义比赛。
 *
 * 预期的负载格式如下：
 * {
 *   "roomName": string,       // 要创建的房间名称
 *   "maxPlayers": number,     // 房间允许的最大玩家数
 *   "isPrivate": boolean      // 房间是否为私有
 * }
 * 返回创建的比赛ID或失败时的错误消息。
 */
function rpcCreateCustomMatch(ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, payload: string): string {
  // 错误代码枚举
  enum ErrorCodes {
    InvalidInput = 0,          // 无效输入
    RoomExists = 1,            // 房间已存在
    MatchCreationFailed = 2,   // 比赛创建失败
    InternalServerError = 3    // 服务器内部错误
  };

  try {
    // 解析负载
    const parsedPayload = JSON.parse(payload);
    const { roomName, maxPlayers, isPrivate } = parsedPayload;

    // 检查房间名称是否有效
    if (!roomName || typeof roomName !== 'string') {
      return JSON.stringify({ "error": "Invalid room name", "code": ErrorCodes.InvalidInput });
    }
```

```
// 检查房间是否已存在
const existingRoom = nk.storageRead([{ collection: CollectionRoomNameToMatchId, key: roomName, userId: UidSystem }]);
if (existingRoom.length > 0) {
  return JSON.stringify({ "error": "Room already exists", "code": ErrorCodes.RoomExists });
}

// 创建新的比赛
const matchId = nk.matchCreate("lobby", { "roomName": roomName, "maxPlayers": maxPlayers, "isPrivate": isPrivate, "host": ctx.userId });
if (!matchId) {
  return JSON.stringify({ "error": "Match creation failed", "code": ErrorCodes.MatchCreationFailed });
}

// 将房间名称和比赛ID映射关系写入存储
nk.storageWrite([
  {
    collection: CollectionRoomNameToMatchId,
    key: roomName,
    value: { "matchId": matchId },
    userId: UidSystem,
    permissionRead: 2
  },
  {
    collection: CollectionMatchIdToRoomName,
    key: matchId,
    value: { "roomName": roomName },
    userId: UidSystem,
    permissionRead: 2
  }
]);
```

```
    // 返回创建的比赛ID
    return JSON.stringify({ "matchId": matchId });
  } catch (e) {
    logger.error(`Failed to create custom match: ${e}`);
    return JSON.stringify({ "error": "Internal Server Error", "code": ErrorCodes.InternalServerError });
  }
}
```

Buscar coincidencias personalizadas (findMatch.ts): este archivo maneja la lógica para buscar coincidencias existentes en el servidor. Busca una coincidencia según el nombre de la sala proporcionado y devuelve el ID de la coincidencia.

```
/**
 * 在Nakama服务器中根据房间名称查找自定义比赛。
 *
 * 预期的负载格式如下：
 * {
 *   "roomName": string   // 要搜索的房间名称
 * }
 *
 * 如果找到比赛，则返回比赛ID，否则返回错误消息。
 */
function rpcFindCustomMatch(ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, payload: string): string {
  // 错误代码枚举
  enum ErrorCodes {
    InvalidInput = 0,          // 无效输入
    RoomNotFound = 1,          // 房间未找到
    InternalServerError = 2    // 服务器内部错误
  };

  try {
    // 解析负载
    const parsedPayload = JSON.parse(payload);
    const { roomName } = parsedPayload;

    // 检查房间名称是否有效
    if (!roomName || typeof roomName !== 'string') {
      return JSON.stringify({ "error": "Invalid room name", "code": ErrorCodes.InvalidInput });
    }
```

```
    // 如果比赛ID不存在，则返回房间未找到的错误
    if (!matchId) {
      return JSON.stringify({ "error": "Room not found", "code": ErrorCodes.RoomNotFound });
    }

    // 返回找到的比赛ID
    return JSON.stringify({ "matchId": matchId });
  } catch (e) {
    logger.error(`Failed to find custom match: ${e}`);
    return JSON.stringify({ "error": "Internal Server Error", "code": ErrorCodes.InternalServerError });
  }
}
```

Enumerar todas las coincidencias personalizadas (listMatches.ts): este archivo maneja la lógica para enumerar todas las coincidencias existentes en el servidor. Devuelve una lista de partidos para que los jugadores los vean y se unan.

```
/**
 * 根据提供的条件列出Nakama服务器上可用的比赛。
 *
 * 预期的负载包含用于筛选比赛列表的参数，格式如下：
 * {
 *   "limit": number,          // 返回的比赛数量上限。
 *   "authoritative": boolean, // 是否仅列出权威（服务器管理的）比赛。
 *   "label": string,          // 用于筛选比赛的标签。
 *   "minSize": number,        // 比赛中参与者的最小数量。
 *   "maxSize": number         // 比赛中参与者的最大数量。
 * }
 *
 * 返回符合给定条件的比赛列表，排除私人比赛。
 */
function rpcListMatches(ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, payload: string): string {
    // 解析负载
    const parsedPayload = JSON.parse(payload);
    const { limit, authoritative, label, minSize, maxSize } = parsedPayload;

    // 获取所有比赛列表
    const allMatches = nk.matchList(limit, authoritative, label, minSize, maxSize);

    // 筛选公共比赛，排除私人比赛
    const publicMatches = allMatches.filter(function (match) {
        const labelData = JSON.parse(match.label);
        return !labelData.isPrivate;
    });

    // 返回公共比赛的JSON字符串
    return JSON.stringify(publicMatches);
}
```

GetUserProperties.ts:recupera atributos para un concurso y usuario específicos del servidor Nakama.

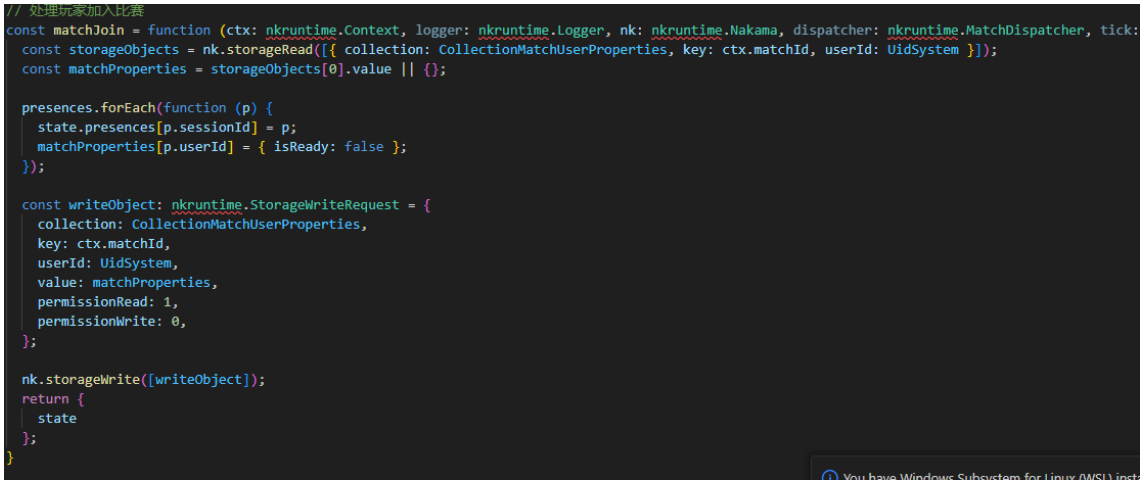
```
/**
 * 从Nakama服务器中检索特定比赛和用户的属性。
 *
 * 预期的负载格式如下：
 * {
 *   "matchId": string,   // 比赛ID
 *   "userId": string     // 要检索其属性的用户ID
 * }
 *
 * 返回指定比赛中指定用户的属性。
 */
function rpcGetUserProperties(ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, payload: string): string {
  // 解析负载
  const parsedPayload = JSON.parse(payload);
  const { matchId, userId } = parsedPayload;

  // 从存储中读取比赛用户属性
  const matchUserProperties = nk.storageRead([{ collection: CollectionMatchUserProperties, key: matchId, userId: UidSystem }]);
  const userProperties = matchUserProperties[0].value;

  // 返回指定用户的属性
  return JSON.stringify(userProperties[userId]);
};
```

matchHandler.ts: el archivo contiene múltiples funciones para procesar eventos de partido, incluida la inicialización del partido, la entrada y salida del jugador, el bucle del partido, la señal y terminación del partido, etc. Cada función maneja una lógica de juego específica para garantizar el progreso fluido del juego y la actualización del estado. El archivo también define una enumeración de código de operación (OpCode) y un conjunto correspondiente de funciones de controlador (ActionHandlers) para procesar diferentes tipos de mensajes de carrera.

Unirse a la partida (matchJoinAttempt, matchJoin):

Un nuevo jugador solicita unirse a una partida, el servidor llama a matchJoinAttempt para verificar la solicitud y, si es válida, llama a matchJoin para agregar al jugador a la partida.

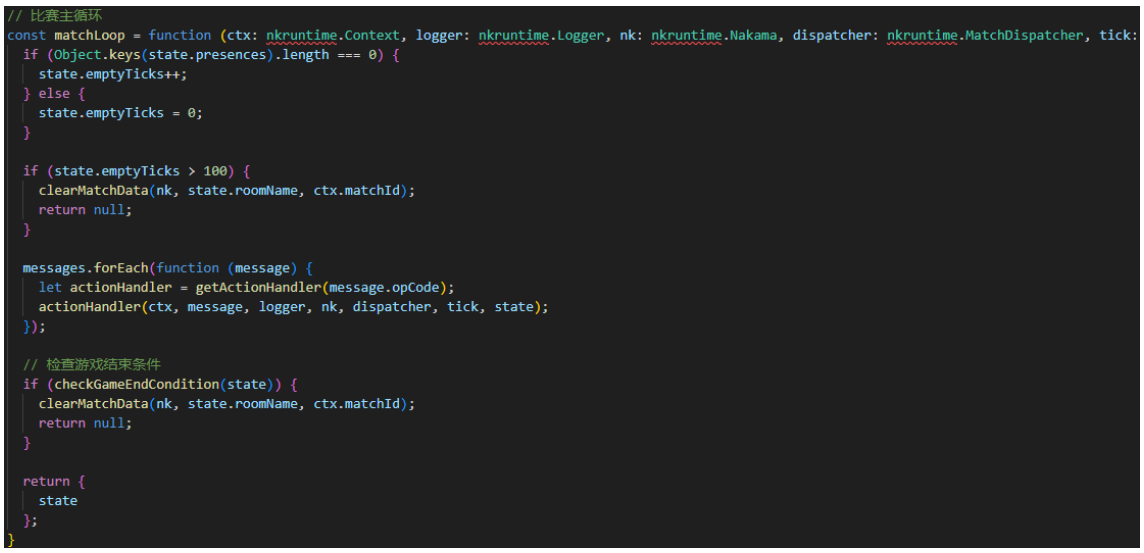
```
// 处理玩家尝试加入比赛
const matchJoinAttempt = function (ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, dispatcher: nkruntime.MatchDispatcher
  logger.debug('%q attempted to join Lobby match', ctx.userId);
  const { maxPlayers } = JSON.parse(ctx.matchLabel);
  const currentPlayers = Object.keys(state.presences).length;

  if (currentPlayers >= maxPlayers) {
    return {
      state,
      accept: false,
      rejectMessage: 'The room is full'
    };
  }
  return {
    state,
    accept: true
  };
}
```



```
// 处理玩家加入比赛
const matchJoin = function (ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, dispatcher: nkruntime.MatchDispatcher, tick:
  const storageObjects = nk.storageRead([{ collection: CollectionMatchUserProperties, key: ctx.matchId, userId: UidSystem }]);
  const matchProperties = storageObjects[0].value || {};

  presences.forEach(function (p) {
    state.presences[p.sessionId] = p;
    matchProperties[p.userId] = { isReady: false };
  });

  const writeObject: nkruntime.StorageWriteRequest = {
    collection: CollectionMatchUserProperties,
    key: ctx.matchId,
    userId: UidSystem,
    value: matchProperties,
    permissionRead: 1,
    permissionWrite: 0,
  };

  nk.storageWrite([writeObject]);
  return {
    state
  };
}
```

Bucle de la partida (matchLoop):

El servidor mantiene el estado de la partida y, mediante matchLoop, maneja cada tick del juego, recibiendo y procesando las acciones de los jugadores.



```
// 比赛主循环
const matchLoop = function (ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, dispatcher: nkruntime.MatchDispatcher, tick:
  if (Object.keys(state.presences).length === 0) {
    state.emptyTicks++;
  } else {
    state.emptyTicks = 0;
  }

  if (state.emptyTicks > 100) {
    clearMatchData(nk, state.roomName, ctx.matchId);
    return null;
  }

  messages.forEach(function (message) {
    let actionHandler = getActionHandler(message.opCode);
    actionHandler(ctx, message, logger, nk, dispatcher, tick, state);
  });

  // 检查游戏结束条件
  if (checkGameEndCondition(state)) {
    clearMatchData(nk, state.roomName, ctx.matchId);
    return null;
  }

  return {
    state
  };
}
```

Dejar la partida (matchLeave):

Un jugador deja la partida, el servidor llama a matchLeave para actualizar el estado de la partida y manejar la lógica después de la salida del jugador.

```
// 处理玩家离开比赛
const matchLeave = function (ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, dispatcher: nkruntime.MatchDispatcher, tick:
  const matchLabel = JSON.parse(ctx.matchLabel);
  const hostId = matchLabel.host;

  presences.forEach(p => {
    delete state.presences[p.sessionId];
    logger.debug('%q left the match', p.userId);
  });

  // 更新存储以反映存在的变化
  const storageObjects = nk.storageRead([{ collection: CollectionMatchUserProperties, key: ctx.matchId, userId: UidSystem }]);
  const matchProperties = storageObjects[0].value || {};

  presences.forEach(p => delete matchProperties[p.userId]);

  const writeObject: nkruntime.StorageWriteRequest = {
    collection: CollectionMatchUserProperties,
    key: ctx.matchId,
    userId: UidSystem,
    value: matchProperties,
    permissionRead: 1,
    permissionWrite: 0,
  };
  nk.storageWrite([writeObject]);
```

Terminar la partida (matchTerminate):

Al finalizar la partida, el servidor llama a matchTerminate para limpiar los datos de la partida.

```
// 处理比赛终止
const matchTerminate = function (ctx: nkruntime.Context, logger: nkruntime.Logger, nk: nkruntime.Nakama, dispatcher: nkruntime.MatchDispatcher,
  logger.debug('Lobby match terminated');

  return {
    state
  };
}
```

3.3.2 Comunicación cliente-servidor

Para garantizar una comunicación eficiente y segura entre el cliente y el servidor, es necesario implementar varios aspectos clave:

1. **Protocolos de Comunicación:**
   - **WebSocket:** Utilizado para la comunicación en tiempo real entre el cliente y el servidor. Permite la transferencia bidireccional de datos, esencial para la actualización instantánea de las acciones del jugador y el estado del juego.
   - **HTTP/HTTPS:** Utilizado para operaciones de solicitud-respuesta, como la autenticación de usuarios y la recuperación de datos no críticos en tiempo real. Este protocolo garantiza una comunicación segura y encriptada, protegiendo los datos sensibles durante la transferencia.
2. **Autenticación y Gestión de Sesiones:**
   - **Autenticación de Usuarios:** Implementada para verificar la identidad de los jugadores al iniciar sesión en el servidor. Esto asegura que solo usuarios autorizados puedan acceder al juego y sus funcionalidades.
   - **Gestión de Sesiones:** Asegura que cada sesión de juego sea única y segura, utilizando tokens de sesión que se verifican en cada solicitud. Esto previene el acceso no autorizado y mantiene la integridad de las sesiones de juego.
3. **Sincronización de Datos:**
   - **Estado del Juego:** Utiliza JSON para almacenar y transferir el estado del juego entre el cliente y el servidor. Esto incluye la posición de los jugadores, el estado de los recursos y otros datos relevantes del juego. La estructura en JSON permite una fácil manipulación y transferencia de datos.
   - **Mensajes en Tiempo Real:** Implementados a través de WebSocket, permiten la actualización instantánea de las acciones de los jugadores y el estado del juego. Esto asegura que todos los jugadores vean los mismos eventos y cambios en tiempo real, mejorando la experiencia de juego.
4. **Manejo de Eventos del Juego:**
   - **Eventos de Juego:** Como el inicio y fin de los turnos, el uso de habilidades y cambios en el estado de los jugadores. Estos eventos se manejan en el servidor y se sincronizan con el cliente para asegurar una experiencia de juego coherente y fluida.
   - **Lógica del Juego:** Implementada en el servidor para asegurar la integridad del juego. El servidor valida todas las acciones de los jugadores antes de aplicarlas al estado del juego. Esto previene trampas y asegura que el juego se desarrolle según las reglas establecidas.

Detalles Específicos:

1. **WebSocket:**
   - **Implementación:** Se establece una conexión WebSocket entre el cliente y el servidor al iniciar el juego. Esta conexión se utiliza para transmitir datos en tiempo real, como movimientos de unidades, ataques y otros eventos del juego.
   - **Ventajas:** La comunicación bidireccional y en tiempo real permite que las acciones de los jugadores se reflejen instantáneamente en el juego, mejorando la experiencia de usuario.
2. **HTTP/HTTPS:**
   - **Autenticación:** Cuando un jugador intenta iniciar sesión, se envía una solicitud HTTP/HTTPS al servidor con las credenciales del usuario. El servidor verifica estas credenciales y, si son correctas, inicia una sesión para el usuario.
   - **Recuperación de Datos:** Operaciones como cargar configuraciones de usuario o recuperar estadísticas del juego se realizan a través de solicitudes HTTP/HTTPS, asegurando que estos datos se transfieran de manera segura.
3. **Gestión de Sesiones:**
   - **Tokens de Sesión:** Cada vez que un usuario inicia sesión, el servidor genera un token de sesión único. Este token se utiliza para identificar y validar al usuario en todas las solicitudes posteriores, garantizando que cada acción provenga de un usuario autenticado.
   - **Seguridad:** Los tokens de sesión se encriptan y se verifican en cada solicitud, previniendo accesos no autorizados y asegurando la integridad de la sesión.
4. **Sincronización de Datos:**
   - **Formato JSON:** El estado del juego se almacena y transfiere en formato JSON, que es ligero y fácil de manipular tanto en el cliente como en el servidor. Esto incluye datos como la posición de las unidades, el estado de los recursos y otras métricas del juego.
   - **Actualizaciones en Tiempo Real:** Utilizando WebSocket, se envían mensajes en tiempo real cada vez que ocurre un evento en el juego, como un movimiento de unidad o un ataque. Esto asegura que todos los jugadores vean el mismo estado del juego en todo momento.
5. **Manejo de Eventos del Juego:**
   - **Validación de Acciones:** Cada vez que un jugador realiza una acción, esta se envía al servidor, que la valida antes de aplicarla al juego. Esto asegura que todas las acciones sean legales y que el juego se desarrolle de acuerdo a las reglas.
   - **Sincronización:** Una vez que una acción es validada, el servidor envía actualizaciones a todos los clientes conectados para reflejar el nuevo estado del juego, manteniendo a todos los jugadores sincronizados.

En resumen, la comunicación cliente-servidor en este juego de estrategia por turnos se basa en protocolos de comunicación eficientes y seguros, una gestión robusta de la autenticación y las sesiones, y una sincronización precisa de los datos del juego para asegurar una experiencia de usuario fluida y coherente.

Capítulo 4. Diseño y desarrollo del juego

4.1 Diseño del sistema de producción de unidades

En esta sección se detallará el diseño del sistema de producción de unidades dentro del juego de estrategia por turnos. Este sistema es crucial para permitir a los jugadores crear y gestionar sus unidades en función de los recursos disponibles y las necesidades estratégicas.

4.1.1 Lógica de producción

La lógica de producción define las reglas y los mecanismos mediante los cuales los jugadores pueden crear nuevas unidades en el juego. Este sistema debe ser equilibrado y permitir una variedad de estrategias para los jugadores. A continuación, se presenta un ejemplo de código que ilustra cómo se puede implementar la lógica de producción de unidades en el juego:

CustomUnitGenerator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using TbsFramework.Cells;
using TbsFramework.Units;
using UnityEngine;

namespace TbsFramework.Grid.UnitGenerators
{
    0 references
    public class CustomUnitGenerator : MonoBehaviour, IUnitGenerator
    {
        2 references
        public Transform UnitsParent; // 父对象，用于存放所有单位
        0 references
        public Transform CellsParent; // 父对象，用于存放所有单元格

        /// <summary>
        /// 返回UnitsParent对象的子对象中的单位。
        /// </summary>
```

```
public List<Unit> SpawnUnits(List<Cell> cells)
{
    List<Unit> ret = new List<Unit>();
    for (int i = 0; i < UnitsParent.childCount; i++)
    {
        var unit = UnitsParent.GetChild(i).GetComponent<Unit>();
        if (unit != null)
        {
            ret.Add(unit); // 将单位添加到列表中
        }
        else
        {
            Debug.LogError("Units Parent游戏对象中存在无效对象");
        }
    }
    return ret; // 返回包含所有单位的列表
}
```

Funcionamiento General

El `CustomUnitGenerator` se utiliza para inicializar y posicionar las unidades en el juego. Durante la fase de producción de unidades, el generador:

- Busca las unidades definidas como hijos de `UnitsParent`.
- Verifica y recopila estas unidades.
- (Opcionalmente) Ajusta las unidades a la grilla del juego utilizando el método `SnapToGrid`.

Este sistema permite a los jugadores generar unidades de manera dinámica y las posiciona correctamente en la grilla del juego, asegurando que el proceso de producción de unidades sea eficiente y organizado.

4.1.2 Integración con sistemas de recursos

Para gestionar y asignar mejor los recursos, el sistema de control económico del juego es crucial. El siguiente ejemplo de código muestra cómo implementar la lógica básica de control económico.

EconomyController.cs

```
using System.Collections.Generic;
using TbsFramework.Grid;
using UnityEngine;
using UnityEngine.Assertions;

namespace TbsFramework.Example4
{
    0 references
    public class EconomyController : MonoBehaviour
    {
        // 字典，存储每个玩家的账户金额
        5 references
        private Dictionary<int, int> Account = new Dictionary<int, int>();
        1 reference
        public int StartingAmount = 0; // 每个玩家的初始金额

        0 references
        public void Awake()
        {
            // 注册GameStarted事件，当游戏开始时调用OnGameStarted方法
            FindObjectOfType<CellGrid>().GameStarted += OnGameStarted;
```

```
        // 游戏开始时初始化每个玩家的账户金额
        1 reference
        private void OnGameStarted(object sender, System.EventArgs e)
        {
            foreach (var player in (sender as CellGrid).Players)
            {
                Account.Add(player.PlayerNumber, StartingAmount); // 初始化每个玩家的账户金额
            }
        }

        // 获取指定玩家的账户金额
        0 references
        public int GetValue(int playerNumber)
        {
            if (Account.ContainsKey(playerNumber))
            {
                return Account[playerNumber];
            }
            return 0; // 如果账户不存在，返回0
        }
```

```
        // 更新指定玩家的账户金额
        0 references
        public void UpdateValue(int playerNumber, int delta)
        {
            Assert.IsTrue(Account.ContainsKey(playerNumber), string.Format("The Account of player number {0} wa
            Account[playerNumber] += delta; // 更新账户金额
        }
    }
}
```

Funcionamiento General

El `EconomyController` se encarga de gestionar los recursos económicos de los jugadores durante la partida. Este controlador:

- Inicializa las cuentas de los jugadores con una cantidad de recursos definida al comenzar la partida.
- Permite consultar la cantidad de recursos de un jugador específico.
- Permite actualizar la cantidad de recursos de un jugador, ya sea aumentando o disminuyendo su valor.

Este sistema asegura que los jugadores puedan administrar sus recursos de manera efectiva, lo cual es crucial para desarrollar estrategias y tomar decisiones en el juego.

4.2 Gestión de recursos y puntos de control

En este capítulo, analizaremos más de cerca la gestión de recursos y los sistemas de puntos de control del juego, que son partes vitales de los juegos de estrategia por turnos.

4.2.1 Diseño de mapas y recursos

En esta sección, discutimos el diseño del mapa y su distribución de recursos. El diseño de mapas es crucial para el equilibrio y la jugabilidad del juego. Un buen diseño de mapa puede brindar igualdad de oportunidades y desafíos para todos los jugadores, mejorando la profundidad estratégica y el interés del juego.

Descripción general del diseño de mapas

Cada jugador tiene su propia base en el mapa. La base proporciona ingresos de recursos básicos pero no puede producir soldados. Los jugadores deben depender de los cuarteles para producir soldados. Cada jugador comienza con 4 cuarteles. También hay algunos puntos de recursos neutrales repartidos por el mapa. Los jugadores pueden aumentar sus ingresos ocupando estos puntos de recursos.

diseño básico

Base: la base de cada jugador está ubicada en la esquina o borde del mapa y proporciona ingresos de recursos básicos. La base en sí no puede producir soldados y necesita depender de cuarteles para la expansión militar.

diseño de cuarteles

Cuarteles: Cada jugador tiene inicialmente 4 cuarteles en el mapa. Los cuarteles son edificios clave para producir soldados. Los jugadores deben proteger sus propios cuarteles y al mismo tiempo intentar ocupar los cuarteles del enemigo para debilitar la capacidad de producción del oponente.

Puntos de recursos neutrales

Puntos de recursos neutrales: hay varios puntos de recursos neutrales distribuidos en el mapa. Los jugadores pueden aumentar sus ingresos por ronda ocupando estos puntos de recursos. Estos puntos de recursos se distribuyen en el centro y los bordes del mapa, lo que lleva a los jugadores a expandirse y competir estratégicamente.

Administracion de recursos

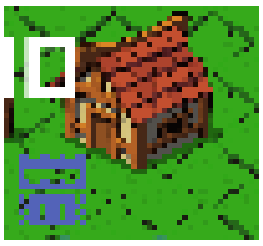
La gestión de recursos es un aspecto muy importante del juego. A través del sistema de control económico del juego, los jugadores pueden gestionar sus ingresos y gastos de recursos.

Recursos iniciales: Cada jugador tiene 2500 monedas de oro al comienzo del juego. Esto proporciona a los jugadores una base de recursos inicial para la producción temprana de unidades y el despliegue estratégico.

Crecimiento de ingresos: al comienzo de cada ronda, los ingresos de recursos de un jugador aumentarán automáticamente según la cantidad de bases, cuarteles y puntos de recursos neutrales que controle. Cada punto de recurso neutral aumentará los ingresos del jugador en 500 monedas de oro por ronda.

La mapa

Base



Cuarteles

Puntos de recursos neutrales

4.2.2 Sistema de captura de puntos de control

El sistema de puntos de control añade una capa extra de estrategia al juego. Al controlar puntos clave del mapa, los jugadores pueden obtener recursos adicionales y ventajas estratégicas. Cada punto de control tiene su propio valor y ubicación únicos, y los jugadores deben desarrollar estrategias para competir y defender estos puntos clave.

CaptureAbility.cs

Descripción del Funcionamiento

La clase `CaptureAbility` gestiona la capacidad de una unidad para capturar puntos de control en el juego. Esta habilidad es fundamental para que los jugadores obtengan ventajas estratégicas y recursos adicionales. A continuación, se presentan y analizan las partes más importantes del código:

Análisis del Código

①.Declaración de Variables:

```
4 references
public Button ActivationButton; // 激活按钮
12 references
private Unit CapturingStructure; // 正在占领的结构
```

- `ActivationButton`: Botón que el jugador debe presionar para activar la habilidad de captura.

- `CapturingStructure`: La unidad que está siendo capturada.

②.Método Act:

```
public override IEnumerator Act(CellGrid cellGrid, bool isNetworkInvoked = false)
{
    if (CanPerform(cellGrid)) // 检查是否可以执行占领操作
    {
        // 获取当前单元格中所有可占领的单位
        var capturable = GetComponent<Unit>().Cell.CurrentUnits.Select(u => u.GetComponent<Capturable
                                                                .OfType<CapturableAbility>()
                                                                .ToList();

        // 计算占领量
        var captureAmount = (int)Mathf.Ceil(GetComponent<Unit>().HitPoints * 10f / GetComponent<Unit>

        // 执行占领
        CapturingStructure = capturable[0].GetComponent<Unit>();
        capturable[0].Capture(captureAmount, GetComponent<Unit>().PlayerNumber);
        UnitReference.ActionPoints -= 1; // 减少单位的行动点数
    }

    yield return null;
```

- **Propósito:** Este método se llama cuando la habilidad se activa. Gestiona la lógica de captura.
- **Proceso:**

  - Verifica si la habilidad puede ejecutarse (`CanPerform`).
  - Calcula el `captureAmount` basado en los puntos de vida de la unidad.
  - Captura la estructura utilizando el `PlayerNumber` de la unidad.
  - Reduce los puntos de acción de la unidad (`ActionPoints`).

③.Método Display:

```
// 显示占领按钮
0 references
public override void Display(CellGrid cellGrid)
{
    if (CanPerform(cellGrid))
    {
        ActivationButton.gameObject.SetActive(true);
    }
}
```

- **Propósito:** Muestra el botón de activación si la habilidad se puede realizar.
- **Proceso:**

  - Verifica si la habilidad puede ejecutarse (`CanPerform`).
  - Activa el botón de activación (`ActivationButton`).

④.Método OnAbilitySelected:

```
// 当能力被选择时调用
0 references
public override void OnAbilitySelected(CellGrid cellGrid)
{
    if (CapturingStructure != null)
    {
        if (!UnitReference.Cell.CurrentUnits.Contains(CapturingStructure))
        {
            // 重置被占领结构的忠诚度
            CapturingStructure.GetComponent<CapturableAbility>().Loyality = CapturingStructure.GetComp
            CapturingStructure.GetComponent<CapturableAbility>().UpdateLoyalityUI();
            CapturingStructure = null;
        }
    }
}
```

- **Propósito:** Se llama cuando se selecciona la habilidad.
- **Proceso:**

  - Verifica si la estructura capturada ya no está en la misma celda.

- Restablece la lealtad de la estructura capturada.

⑤.Método CanPerform:

```
// 检查是否可以执行占领操作
3 references
public override bool CanPerform(CellGrid cellGrid)
{
    var capturable = GetComponent<Unit>().Cell.CurrentUnits.Select(u => u.GetComponent<CapturableAbility
                                                .OfType<CapturableAbility>()
                                                .ToList();

    return capturable.Count > 0 && capturable[0].GetComponent<Unit>().PlayerNumber != GetComponent<Unit
}
```

- **Propósito:** Verifica si la habilidad puede realizarse.
- **Proceso:**

  - Comprueba si hay unidades capturables en la misma celda.
  - Verifica que la unidad capturable no pertenezca al mismo jugador.
  - Asegura que la unidad tiene suficientes puntos de acción (`ActionPoints`).

Funcionamiento General

La clase `CaptureAbility` permite a las unidades capturar estructuras en el juego, añadiendo una dimensión estratégica importante. La habilidad:

- Verifica si la captura es posible.
- Gestiona la lógica de captura y actualización del estado de la estructura capturada.
- Permite a los jugadores activar esta habilidad a través de la interfaz de usuario, mostrando un botón cuando es posible realizar la acción.

Este sistema asegura que los jugadores puedan interactuar estratégicamente con el entorno del juego, capturando puntos de control para obtener ventajas y recursos adicionales.

4.3 Implementación del sistema de turnos

En esta sección, describimos en detalle la implementación del sistema por turnos. El sistema por turnos es uno de los mecanismos centrales de los juegos de estrategia, que gestiona la secuencia de acciones del jugador y los cambios en el estado del juego.

4.3.1 Mecánica de turnos

1. Clase CellGrid

CellGrid es la clase principal que gestiona todo el proceso del juego por turnos. Realiza un seguimiento de las celdas, unidades y objetos de los jugadores en el juego, inicia el juego y maneja las transiciones de turnos. Cuando el usuario interactúa con una unidad o célula, ésta reacciona y genera eventos relacionados con el progreso del juego.

Las características clave incluyen:

Comenzar el juego

Manejar la entrada del usuario

Gestionar transiciones de giro

CellGrid.cs:

Explicación de las Partes Principales

①. Método `InitializeAndStart`:

```
public void InitializeAndStart()
{
    Initialize();
    StartGame();
}
```

- **Propósito**: Inicializa y comienza el juego llamando a los métodos `Initialize` y `StartGame`.
- **Proceso**:
  - Llama a `Initialize` para configurar jugadores, celdas y unidades.
  - Llama a `StartGame` para iniciar el ciclo del juego.

②.Método `Initialize`:

```
public void Initialize()
{
    if (LevelLoading != null)
        LevelLoading.Invoke(this, EventArgs.Empty);

    GameFinished = false;
    Players = new List<Player>();
    for (int i = 0; i < PlayersParent.childCount; i++)
    {
        var player = PlayersParent.GetChild(i).GetComponent<Player>();
        if (player != null && player.gameObject.activeInHierarchy)
        {
            player.Initialize(this);
            Players.Add(player);
        }
    }
```

```
Cells = new List<Cell>();
for (int i = 0; i < transform.childCount; i++)
{
    var cell = transform.GetChild(i).gameObject.GetComponent<Cell>();
    if (cell != null)
    {
        if (cell.gameObject.activeInHierarchy)
        {
            Cells.Add(cell);
        }
    }
    else
    {
        Debug.LogError("Invalid object in cells parent game object");
    }
}
```

```
foreach (var cell in Cells)
{
    cell.CellClicked += OnCellClicked;
    cell.CellHighlighted += OnCellHighlighted;
    cell.CellDehighlighted += OnCellDehighlighted;
    cell.GetComponent<Cell>().GetNeighbours(Cells);
}

Units = new List<Unit>();
var unitGenerator = GetComponent<IUnitGenerator>();
if (unitGenerator != null)
{
    var units = unitGenerator.SpawnUnits(Cells);
    foreach (var unit in units)
    {
        AddUnit(unit.GetComponent<Transform>());
    }
```

- **Propósito**: Inicializa jugadores, celdas y unidades.
- **Proceso**:
  - Invoca eventos de carga de nivel.
  - Configura jugadores y los añade a la lista de jugadores.
  - Configura celdas, añade eventos de clic y selecciona celdas.
  - Genera unidades iniciales usando `IUnitGenerator`.

③.Método `StartGame`:

```
public void StartGame()
{
    TransitionResult transitionResult = GetComponent<TurnResolver>().ResolveStart(this);
    PlayableUnits = transitionResult.PlayableUnits;
    CurrentPlayerNumber = transitionResult.NextPlayer.PlayerNumber;

    GameStarted?.Invoke(this, EventArgs.Empty);

    PlayableUnits().ForEach(u => { u.GetComponents<Ability>().ToList().ForEach(a => a.OnTurnStart(this)
    CurrentPlayer.Play(this);
    Debug.Log("Game started");
}
```

- **Propósito**: Inicia el juego y establece el primer turno.
- **Proceso**:
  - Resuelve el inicio del turno usando `TurnResolver`.
  - Configura las unidades jugables y establece el jugador actual.
  - Invoca eventos de inicio del juego.
  - Ejecuta el método `Play` del jugador actual.

④.Método `EndTurn`:

```
public void EndTurn(bool isNetworkInvoked=false)
{
    _cellGridState.EndTurn(isNetworkInvoked);
}
```

- **Propósito**: Finaliza el turno actual y realiza la transición al siguiente turno.
- **Proceso**:
    - Llama a `EndTurnExecute` para manejar la lógica de finalización de turno.

⑤.Método `EndTurnExecute`:

```
private void EndTurnExecute(bool isNetworkInvoked=false)
{
    cellGridState = new CellGridStateBlockInput(this);
    bool isGameFinished = CheckGameFinished();
    if (isGameFinished)
    {
        return;
    }

    var playableUnits = PlayableUnits();
    for (int i = 0; i < playableUnits.Count; i++)
    {
        var unit = playableUnits[i];
        if (unit == null)
        {
            continue;
        }
```

```
            unit.OnTurnEnd();
            var abilities = unit.GetComponents<Ability>();
            for (int j = 0; j < abilities.Length; j++)
            {
                var ability = abilities[j];
                ability.OnTurnEnd(this);
            }
        }
        TransitionResult transitionResult = GetComponent<TurnResolver>().ResolveTurn(this);

        PlayableUnits = transitionResult.PlayableUnits;
        CurrentPlayerNumber = transitionResult.NextPlayer.PlayerNumber;

        if (TurnEnded != null)
            TurnEnded.Invoke(this, isNetworkInvoked);

        Debug.Log(string.Format("Player {0} turn", CurrentPlayerNumber));
```

```
        playableUnits = PlayableUnits();
        for (int i = 0; i < playableUnits.Count; i++)
        {
            var unit = playableUnits[i];
            if (unit == null)
            {
                continue;
            }

            var abilities = unit.GetComponents<Ability>();
            for (int j = 0; j < abilities.Length; j++)
            {
                var ability = abilities[j];
                ability.OnTurnStart(this);
            }
            unit.OnTurnStart();
        }
        CurrentPlayer.Play(this);
    }
```

- **Propósito**: Ejecuta la lógica de finalización de turno y determina el siguiente jugador en turno.
- **Proceso**:
    - Verifica si el juego ha terminado.
    - Actualiza el estado de las unidades y sus habilidades al final del turno.
    - Resuelve el siguiente turno usando `TurnResolver`.
    - Invoca eventos de finalización de turno.
    - Inicia el turno del siguiente jugador llamando a `Play`.

Funcionamiento General

La clase `CellGrid` es fundamental para el funcionamiento del sistema de turnos del juego. Maneja la inicialización del juego, la interacción del usuario y las transiciones de turno, asegurando que el flujo del juego sea coherente y que todos los eventos relevantes sean gestionados adecuadamente. La combinación de esta lógica con los eventos definidos proporciona una base sólida para el desarrollo de un juego de estrategia por turnos.

2. Turn Resolver

Las transiciones de turno se implementan a través del mecanismo TurnResolver, que permite seleccionar el siguiente jugador y las unidades disponibles para usar en un turno determinado.

SubsequentTurnResolver.cs

Explicación de las Partes Principales

1. Método `ResolveStart`:

```
public override TransitionResult ResolveStart(CellGrid cellGrid)
{
    // 找到玩家编号最小的玩家作为下一个玩家
    var nextPlayerNumber = cellGrid.Players.Min(p => p.PlayerNumber);
    var nextPlayer = cellGrid.Players.Find(p => p.PlayerNumber == nextPlayerNumber);
    // 找到所有属于该玩家的单位
    var allowedUnits = cellGrid.Units.FindAll(u => u.PlayerNumber == nextPlayerNumber);

    // 返回下一个玩家和允许的单位
    return new TransitionResult(nextPlayer, allowedUnits);
}
```

- **Propósito**: Determina el jugador que comenzará el juego.
- **Proceso**:
    - Selecciona el jugador con el número más bajo (indicando el primer jugador).
    - Encuentra todas las unidades pertenecientes a ese jugador.
    - Devuelve un `TransitionResult` con el siguiente jugador y las unidades permitidas.

2. Método `ResolveTurn`:

```
public override TransitionResult ResolveTurn(CellGrid cellGrid)
{
    // 计算下一个玩家的编号
    var nextPlayerNumber = (cellGrid.CurrentPlayerNumber + 1) % cellGrid.NumberOfPlayers;
    // 如果下一个玩家没有单位，则继续找下一个玩家
    while (cellGrid.Units.FindAll(u => u.PlayerNumber.Equals(nextPlayerNumber)).Count == 0)
    {
        nextPlayerNumber = (nextPlayerNumber + 1) % cellGrid.NumberOfPlayers;
    }

    // 找到下一个玩家和其对应的单位
    var nextPlayer = cellGrid.Players.Find(p => p.PlayerNumber == nextPlayerNumber);
    var allowedUnits = cellGrid.Units.FindAll(u => u.PlayerNumber == nextPlayerNumber);

    // 返回下一个玩家和允许的单位
    return new TransitionResult(nextPlayer, allowedUnits);
}
```

- **Propósito**: Determina el siguiente jugador en el turno.
- **Proceso**:
    - Calcula el número del siguiente jugador en la secuencia, avanzando en el orden.
    - Si el siguiente jugador no tiene unidades, pasa al siguiente jugador.
    - Encuentra todas las unidades pertenecientes al siguiente jugador.
    - Devuelve un `TransitionResult` con el siguiente jugador y las unidades permitidas.

Funcionamiento General

En este juego de estrategia por turnos para dos jugadores, el SubsequentTurnResolver gestiona las transiciones de turno de manera sencilla y eficiente. Selecciona al siguiente jugador en orden y verifica que tenga unidades disponibles para jugar. Este mecanismo garantiza un flujo de juego justo y alternado entre los dos jugadores, permitiendo una experiencia de juego equilibrada y organizada.

3. Game State Management

Mecánicas de interacción del usuario, transiciones de turnos y manejo de condiciones de final de juego:

Interacción del usuario: la selección de unidades, el movimiento y los ataques del usuario son administrados por subclases de la clase CellGridState. CellGridState es una clase abstracta que contiene métodos de devolución de llamada que se llaman cuando se seleccionan, deseleccionan o se hace clic en celdas.

Transición de giro: cuando finaliza un giro, se llama al método CellGrid.EndTurn. Los jugadores usan botones GUI para finalizar sus turnos.

CellGridState.cs

Explicación de las Partes Principales

①. Método `OnCellDeselected`:

```
public virtual void OnCellDeselected(Cell cell)
{
    cell.UnMark();
}
```

- **Propósito**: Maneja la lógica cuando se deselecciona una celda.
- **Proceso**:
  - Llama al método `UnMark` en la celda, que desmarca la celda visualmente.
  - Este método se llama cuando el mouse sale del colisionador de una celda.

②. Método `OnCellSelected`:

```
public virtual void OnCellSelected(Cell cell)
{
    cell.MarkAsHighlighted();
}
```

- **Propósito**: Maneja la lógica cuando se selecciona una celda.
- **Proceso**:
  - Llama al método `MarkAsHighlighted` en la celda, que resalta visualmente la celda.
  - Este método se llama cuando el mouse entra en el colisionador de una celda.

❸. Método `EndTurn`:

```
public virtual void EndTurn(bool isNetworkInvoked)
{
    _cellGrid.EndTurnExecute(isNetworkInvoked);
}
```

- **Propósito**: Finaliza el turno actual y desencadena las transiciones de turno.
- **Proceso**:
    - Llama a `EndTurnExecute` en `CellGrid` para manejar la lógica de finalización del turno.
    - Este método se llama cuando el jugador finaliza su turno, ya sea a través de la GUI o mediante una llamada de red.

Funcionamiento General

En la gestión del estado del juego, la clase `CellGridState` y sus subclases juegan un papel crucial en la administración de la interacción del usuario, las transiciones de turnos y las condiciones del final del juego. Estas clases encapsulan la lógica de cómo el juego debe reaccionar a las acciones del usuario, asegurando que el flujo del juego sea coherente y que las transiciones de turno se manejen de manera eficiente.

4.3.2 Estrategias de ataque y defensa

En esta sección, discutiremos las estrategias de ataque y contraataque en el juego. Aunque no existe un mecanismo de defensa específico, las unidades pueden defenderse después de ser atacadas.

AttackAbility.cs

Explicación de las Partes Principales

①. Método `Act`:

```
public override IEnumerator Act(CellGrid cellGrid, bool isNetworkInvoked = false)
{
    // 检查是否可以执行攻击操作以及目标单位是否在攻击范围内
    if (CanPerform(cellGrid) && UnitReference.IsUnitAttackable(UnitToAttack, UnitReference.Cell))
    {
        // 执行攻击目标单位的操作
        UnitReference.AttackHandler(UnitToAttack);
        yield return new WaitForSeconds(0.5f); // 攻击动画的等待时间

        // 检查目标单位是否可以进行反击
        if (UnitToAttack.ActionPoints > 0 && UnitToAttack.IsUnitAttackable(UnitReference, UnitToAttack.
        {
            // 执行反击操作
            UnitToAttack.AttackHandler(UnitReference);
            yield return new WaitForSeconds(0.5f); // 反击动画的等待时间
        }
    }
    yield return null;
}
```

- **Propósito**: Maneja la lógica de ataque de la unidad y la posibilidad de contraataque.
- **Proceso**:
    - Verifica si la unidad puede realizar un ataque y si el objetivo es atacable.
    - Llama al método `AttackHandler` en la unidad objetivo.
    - Si la unidad objetivo tiene suficientes puntos de acción, verifica si puede contraatacar y realiza el contraataque llamando a `AttackHandler` nuevamente.

②. Método `Display`:

```
public override void Display(CellGrid cellGrid)
{
    var unit = GetComponent<Unit>();
    var enemyUnits = cellGrid.GetEnemyUnits(cellGrid.CurrentPlayer);
    // 查找所有在攻击范围内的敌方单位
    inAttackRange = enemyUnits.Where(u => UnitReference.IsUnitAttackable(u, UnitReference.Cell)).ToList
    // 将在攻击范围内的敌方单位标记为可攻击的敌人
    inAttackRange.ForEach(u => u.MarkAsReachableEnemy());
}
```

- **Propósito**: Muestra las unidades enemigas que están dentro del rango de ataque.
- **Proceso**:
  - Obtiene todas las unidades enemigas dentro del rango de ataque de la unidad.
  - Marca estas unidades enemigas como alcanzables para proporcionar una indicación visual al jugador.

③. Método `OnUnitClicked`:

```
public override void OnUnitClicked(Unit unit, CellGrid cellGrid)
{
    // 如果点击的单位在攻击范围内，执行攻击
    if (UnitReference.IsUnitAttackable(unit, UnitReference.Cell))
    {
        UnitToAttack = unit;
        UnitToAttackID = UnitToAttack.UnitID;
        StartCoroutine(HumanExecute(cellGrid));
    }
    // 如果点击的是当前玩家的单位，进入该单位的能力选择状态
    else if (cellGrid.GetCurrentPlayerUnits().Contains(unit))
    {
        cellGrid.cellGridState = new CellGridStateAbilitySelected(cellGrid, unit, unit.GetComponents<Ab
    }
}
```

- **Propósito**: Maneja la lógica cuando se hace clic en una unidad.
- **Proceso**:
  - Si la unidad es atacable, asigna la unidad objetivo y llama a `HumanExecute` para realizar el ataque.
  - Si la unidad pertenece al jugador actual, cambia el estado de `CellGrid` a `CellGridStateAbilitySelected`.

④. Método `CleanUp`:

```
public override void CleanUp(CellGrid cellGrid)
{
    // 清理攻击范围内单位的标记
    inAttackRange.ForEach(u =>
    {
        if (u != null)
        {
            u.UnMark();
        }
    });
}
```

- **Propósito**: Limpia las marcas visuales después de la finalización de la habilidad.
- **Proceso**:

- Desmarca todas las unidades que estaban marcadas como alcanzables.

⑤. Método `CanPerform`:

```
public override bool CanPerform(CellGrid cellGrid)
{
    // 检查单位是否还有行动点数
    if (UnitReference.ActionPoints <= 0)
    {
        return false;
    }

    var enemyUnits = cellGrid.GetEnemyUnits(cellGrid.CurrentPlayer);
    // 查找所有在攻击范围内的敌方单位
    inAttackRange = enemyUnits.Where(u => UnitReference.IsUnitAttackable(u, UnitReference.Cell)).ToList(

    // 如果有任何敌方单位在攻击范围内，则返回true
    return inAttackRange.Count > 0;
}
```

- **Propósito**: Verifica si la habilidad de ataque puede realizarse.
- **Proceso**:
  - Verifica si la unidad tiene suficientes puntos de acción.
  - Obtiene todas las unidades enemigas dentro del rango de ataque.
  - Devuelve verdadero si hay unidades enemigas atacables dentro del rango.

Funcionamiento General

La clase `AttackAbility` maneja la lógica de ataque y contraataque en el juego. Proporciona métodos para ejecutar ataques, verificar si un ataque es posible y gestionar la visualización de unidades enemigas alcanzables. Esta clase es crucial para implementar estrategias de combate efectivas, permitiendo a las unidades no solo atacar sino también responder a los ataques enemigos, añadiendo una capa de complejidad estratégica al juego.

4.4 Interfaz de usuario

En esta sección, describimos en detalle el diseño de la interfaz de usuario del juego. El diseño de interfaz de usuario es la creación de elementos visuales y diseños que interactúan con los usuarios. Una buena interfaz de usuario no sólo debe ser hermosa, sino también fácil de usar, para que los jugadores puedan comprender fácilmente las funciones y operaciones del juego.

interfaz de inicio de sesión

Función:

Los jugadores inician sesión en el juego utilizando su correo electrónico y contraseña. Dado que no existe una conexión real con el servidor de buzones de correo, sólo se necesitan un buzón de correo virtual y una contraseña para iniciar sesión.

Paso:

Ingrese la dirección de correo electrónico: los jugadores ingresan su dirección de correo electrónico en el cuadro de entrada designado.

Ingresar contraseña: el jugador ingresa la contraseña en el cuadro de entrada de contraseña.

Haga clic en el botón Conectar: El jugador hace clic en el botón "Conectar" para conectarse al servidor.

Interfaz principal

Función:

Los jugadores pueden realizar operaciones en la sala en la interfaz principal, incluida la creación de salas, unirse a salas y partidas rápidas.

Paso:

Crear sala: los jugadores hacen clic en el botón "Crear sala" para crear una nueva sala de juegos.

Unirse a una sala: los jugadores hacen clic en el botón "Unirse a la sala" para unirse a una sala de juegos existente.

Partida rápida: los jugadores hacen clic en el botón "Partida rápida" para emparejar rápidamente a un oponente para un juego.

Interfaz de sala

Función:

Después de ingresar a la sala, los jugadores pueden seleccionar su número de jugador y prepararse para comenzar el juego.

Paso:

Seleccionar número de jugador: los jugadores seleccionan su número de jugador (0 o 1) en la sala.

Listo para comenzar el juego: Después de que todos los jugadores hayan seleccionado sus números, haga clic en el botón "Iniciar juego" para comenzar el juego.

NetworkGUI.cs

Explicación de las Partes Principales

①. Método `ConnectToServer`:

```
// 连接到服务器
0 references
public void ConnectToServer()
{
    SetStatus("Connecting...");
    var userName = _usernameInput.text;
    var password = _passwordInput.text; // 获取密码输入
    var customParams = new Dictionary<string, string> { { "password", password } };
    _networkConnection.ConnectToServer(userName, customParams);
}
```

- **Propósito**: Conectar al jugador al servidor Nakama utilizando el correo electrónico y la contraseña ingresados.
- **Proceso**:
  - Obtiene el correo electrónico y la contraseña del jugador desde los campos de entrada.
  - Utiliza `_networkConnection` para conectarse al servidor con las credenciales proporcionadas.

②. Método `CreatePlayerPanel`:

```
private GameObject CreatePlayerPanel(NetworkUser user, int userIndex, int maxUserCount, string playerNumber, bool isReady)
{
    Assert.IsNotNull(_localUser, $"{nameof(_localUser)} field is not set up");

    var playerSelectionPanelInstance = Instantiate(_playerEntryPrefab, _playerEntryPrefab.transform.parent);
    playerSelectionPanelInstance.transform.Find("Player#").GetComponent<Text>().text = string.Format("#{0}", userIndex.ToString());
    playerSelectionPanelInstance.transform.Find("PlayerName").GetComponent<Text>().text = user.UserName;

    playerSelectionPanelInstance.transform.Find("PlayerNumber").GetComponentInChildren<InputField>().text = playerNumber;
    playerSelectionPanelInstance.transform.Find("PlayerNumber").GetComponent<InputField>().interactable = user.UserID.Equals(_localUser.UserID);
    if (user.UserID.Equals(_localUser.UserID))
    {
        playerSelectionPanelInstance.transform.Find("PlayerNumber").GetComponent<InputField>().onValueChanged.AddListener((value) =>
        {
            playerSelectionPanelInstance.transform.Find("IsReady").GetComponent<Toggle>().interactable = value != string.Empty && user.UserID.Equ
            _localPlayerNumber = int.Parse(value);
```

```
            playerSelectionPanelInstance.transform.Find("IsReady").GetComponent<Toggle>().interactable = value != string.Empty && user.UserID.Equ
            _localPlayerNumber = int.Parse(value);
            var actionParams = new Dictionary<string, string>
            {
                { "user_id", _localUser.UserID },
                { "player_number", value.ToString() }
            };
            _networkConnection.SendMatchState((long)OpCode.PlayerNumberChanged, actionParams);
        });
    }
```

```
    playerSelectionPanelInstance.transform.Find("IsReady").GetComponent<Toggle>().isOn = isReady;
    playerSelectionPanelInstance.transform.Find("IsReady").GetComponent<Toggle>().interactable = playerNumber != string.Empty && user.UserID.Equa
    if (user.UserID.Equals(_localUser.UserID))
    {
        playerSelectionPanelInstance.transform.Find("IsReady").GetComponent<Toggle>().onValueChanged.AddListener((value) =>
        {
            playerSelectionPanelInstance.transform.Find("PlayerNumber").GetComponent<InputField>().interactable = !value;

            var actionParams = new Dictionary<string, string>
            {
            { "user_id", _localUser.UserID },
            { "is_ready", value.ToString() }
            };
            _networkConnection.SendMatchState((long)OpCode.IsReadyChanged, actionParams);
            _readyCount += value ? 1 : -1;
            if (_readyCount == maxUserCount)
            {
                StartCoroutine(SetupMatch());
            }
        });
```

- **Propósito**: Crear y configurar el panel de selección de jugador en la interfaz de la sala.
- **Proceso**:
    - Instancia un nuevo panel de selección de jugador.
    - Configura el panel con la información del usuario, como el nombre de usuario y el número de jugador.
    - Agrega listeners para manejar cambios en el número de jugador y el estado de preparación.

③. Método `OnRoomJoined`:

```
public class NetworkGUI : MonoBehaviour
    private void OnRoomJoined(object sender, RoomData roomData)
        _quickMatchButton.interactable = false;

        _playerPanels = new Dictionary<string, GameObject>();
        _localUser = roomData.LocalUser;

        _lobbyPanel.SetActive(false);
        _roomPanel.SetActive(true);
        _roomNameText.text = roomData.RoomName;
        int userIndex = 1;
        foreach (var networkUser in roomData.Users)
        {
            var playerNumber = networkUser.CustomProperties.ContainsKey("playerNumber") ? networkUser.CustomPropertie
            var isReady = networkUser.CustomProperties.ContainsKey("isReady") && bool.Parse(networkUser.CustomPropert
            var playerSelectionPanelInstance = CreatePlayerPanel(networkUser, userIndex, roomData.MaxUsers, playerNum
            playerSelectionPanelInstance.SetActive(true);

            _playerPanels.Add(networkUser.UserID, playerSelectionPanelInstance);
            userIndex += 1;
        }
        SetStatus("Sala unida");
```

- **Propósito**: Manejar la lógica cuando un jugador se une a una sala.
- **Proceso**:
  - Configura la interfaz para mostrar la sala.
  - Crea paneles de selección de jugador para cada jugador en la sala.
  - Actualiza el estado de la interfaz y muestra el nombre de la sala.

④. Método `OnPlayerReadyChanged`:

```
// 玩家准备状态改变时调用
1 reference
private void OnPlayerReadyChanged(Dictionary<string, string> actionParams)
{
    var userID = actionParams["user_id"];
    if (userID.Equals(_localUser.UserID))
    {
        return;
    }

    var isReady = bool.Parse(actionParams["is_ready"]);
    _readyCount += isReady ? 1 : -1;

    if (_readyCount == _maxPlayers)
    {
        StartCoroutine(SetupMatch());
    }

    _playerPanels[userID].transform.Find("IsReady").GetComponent<Toggle>().isOn = bool.Parse(actionParams["is_re
}
```

- **Propósito**: Actualizar la interfaz y la lógica del juego cuando el estado de preparación de un jugador cambia.
- **Proceso**:
  - Verifica si el jugador está listo y actualiza el contador de jugadores listos.
  - Si todos los jugadores están listos, inicia la configuración del juego.

⑤. Método `SetupMatch`:

```
private IEnumerator SetupMatch()
{
    for (int i = 0; i < _playersParent.childCount; i++)
    {
        var playerGO = _playersParent.GetChild(i).gameObject;
        if (!playerGO.activeInHierarchy)
        {
            continue;
        }

        var player = playerGO.GetComponent<Player>();
        var playerNumber = player.PlayerNumber;

        if (!playerNumber.Equals(_localPlayerNumber) && player is HumanPlayer || player is AIPlayer && !_network
        {
            Destroy(player);
            var remotePlayer = playerGO.AddComponent<RemotePlayer>();
            remotePlayer.NetworkConnection = _networkConnection;
            remotePlayer.PlayerNumber = playerNumber;
        }
    }
```

```
    yield return new WaitForEndOfFrame();
    gameObject.SetActive(false);
    _cellGrid.InitializeAndStart();
}
```

- **Propósito**: Configurar y comenzar el juego cuando todos los jugadores están listos.
- **Proceso**:
    - Reemplaza los jugadores locales por jugadores remotos si es necesario.
    - Inicializa y comienza el juego utilizando el objeto `CellGrid`.

Funcionamiento General

La interfaz de usuario (UI) es un componente esencial para la experiencia de juego, proporcionando a los jugadores una manera intuitiva y eficiente de interactuar con el juego. A lo largo de esta sección, hemos detallado las diferentes interfaces que componen nuestro juego de estrategia por turnos, desde la interfaz de inicio de sesión hasta la interfaz principal y la de la sala.

La implementación del código en `NetworkGUI.cs` ha sido diseñada para asegurar una experiencia de usuario fluida y sin complicaciones. A través de métodos bien estructurados, hemos logrado:

1. **Conexión al Servidor**: Permitir a los jugadores conectarse al servidor Nakama utilizando su correo electrónico y contraseña, simplificando el proceso de autenticación.

2. **Gestión de Salas**: Facilitar la creación y unión a salas, permitiendo a los jugadores organizar partidas de manera eficiente.
3. **Selección de Jugador y Preparación**: Implementar una interfaz clara y accesible para la selección de número de jugador y estado de preparación, asegurando que todos los jugadores estén listos antes de comenzar el juego.
4. **Configuración del Juego**: Configurar y lanzar la partida de manera automática cuando todos los jugadores están listos, garantizando una transición suave al juego real.

A través de estos componentes, hemos construido una interfaz que no solo es funcional sino también accesible, mejorando significativamente la experiencia del usuario y facilitando la interacción con el juego. La integración con el servidor Nakama asegura que el juego sea robusto y capaz de manejar múltiples jugadores de manera eficiente, mientras que el diseño de la UI asegura que los jugadores puedan concentrarse en la estrategia y el disfrute del juego sin distracciones técnicas.

Capítulo 5. Pruebas y evaluación

5.1 Pruebas funcionales

5.1.1 Casos de prueba

En esta sección, detallamos los diversos escenarios de prueba que realizamos para garantizar que el juego funcione correctamente. A continuación se muestran algunos casos de prueba principales:

Prueba de función de inicio de sesión:

Pasos: Ingrese la dirección de correo electrónico virtual y la contraseña, y haga clic en el botón Conectar.

Resultados esperados: inicie sesión correctamente y acceda a la interfaz del lobby.

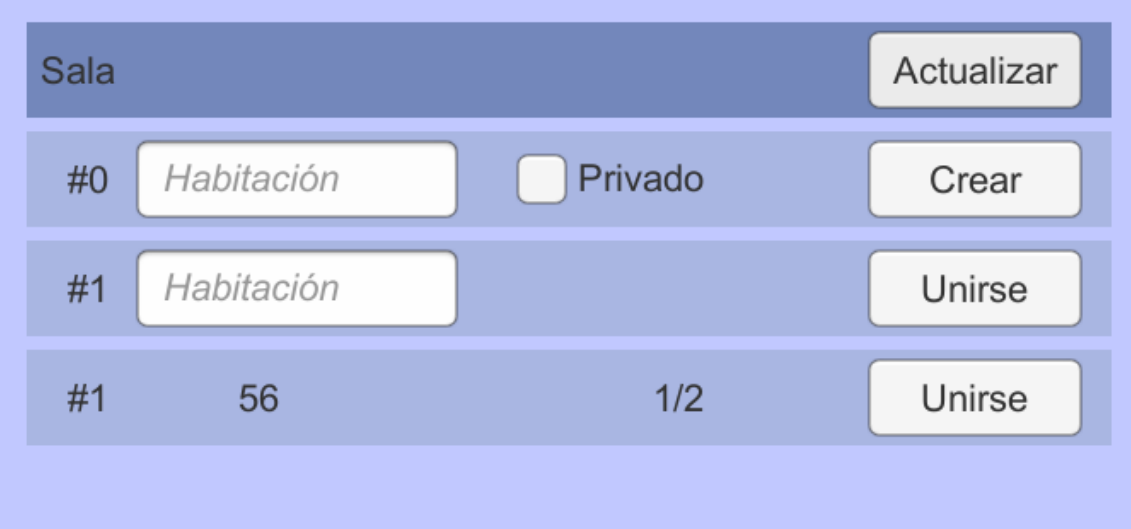
Resultado: Aprobado.

Crea una prueba de sala:

Pasos: Ingrese el nombre de la sala en la interfaz del lobby y haga clic en el botón Crear sala.

Resultados esperados: la sala se crea correctamente y se ingresa a la interfaz de la sala.

Resultado: Aprobado.



Únase a la prueba de la sala:

Pasos: Ingrese el nombre de la habitación existente en la interfaz del lobby y haga clic en el botón Agregar habitación.

Resultados esperados: Únase exitosamente a la sala e ingrese a la interfaz de la sala.

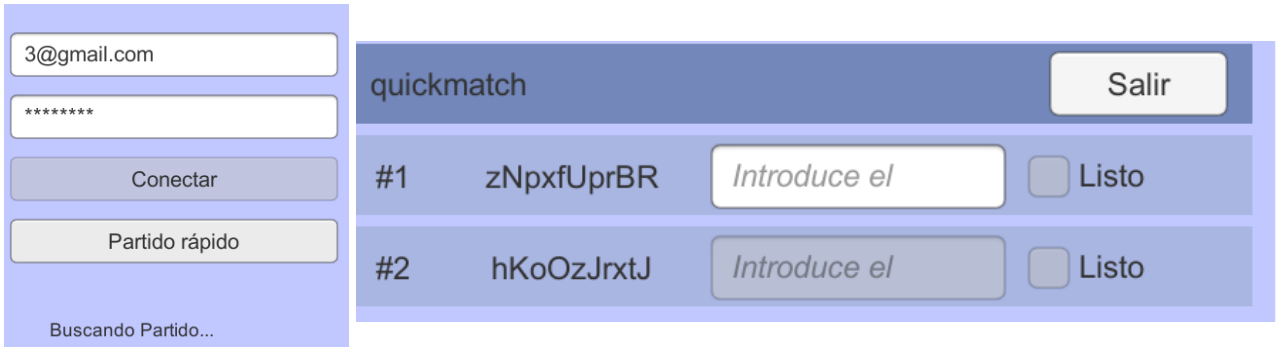
Resultado: Aprobado.

Prueba de partida rápida:

Pasos: haga clic en el botón Partida rápida.

Resultados esperados: Se hizo coincidir con éxito y se ingresó a la interfaz de la sala.

Resultado: Aprobado.

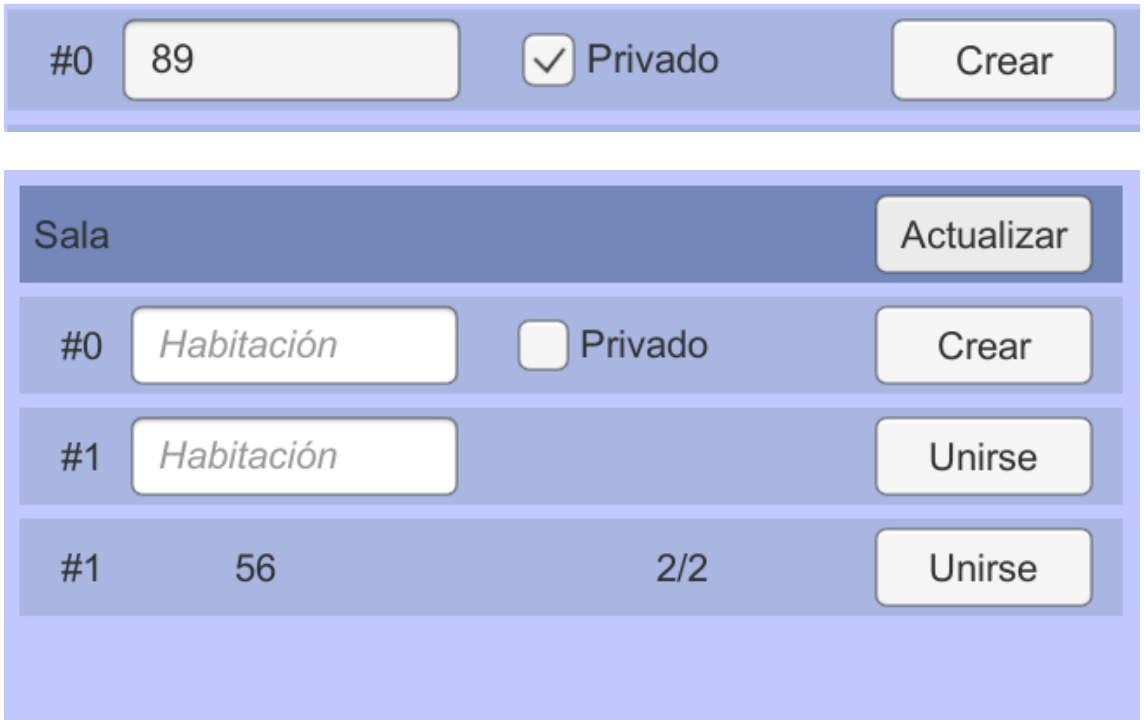


Prueba de habitación privada:

Pasos: seleccione la opción "Habitación privada" al crear una habitación, ingrese el nombre de la habitación y haga clic en el botón Crear habitación. El otro jugador ingresa el nombre de la sala y hace clic en el botón Unirse a la sala.

Resultados esperados: solo los jugadores que conocen el nombre de la sala pueden unirse a una sala privada, otros jugadores no pueden verla ni unirse.

Resultado: Aprobado.



El juego comienza a probar:

Pasos: Una vez que todos los jugadores estén listos, comienza el juego.

Resultados esperados: se ingresa correctamente a la interfaz del juego y el jugador puede iniciar el juego.

Resultado: Aprobado.

Prueba de paso de turnos:

- **Pasos:** Iniciar una partida, realizar acciones y finalizar el turno.
- **Resultados esperados:** Verificar que el control pasa al siguiente jugador sin problemas.
- **Resultado:** Aprobado.

Prueba de apilación de recursos:

- **Pasos:** Realizar acciones que generen y consuman recursos.
- **Resultados esperados:** Comprobar que los valores de los recursos se actualizan correctamente.
- **Resultado:** Aprobado.

DAY 1					
	UNITS	LOST	BASES	INCOME	FUNDS
PLAYER 0	0	0	5	2500	5000
PLAYER 1	0	0	5	2500	2500
NEUTRAL BASES	10				

DAY 2					
	UNITS	LOST	BASES	INCOME	FUNDS
PLAYER 0	0	0	5	2500	7500
PLAYER 1	1	0	5	2500	4000
NEUTRAL BASES	10				

Prueba de movimiento y combate:

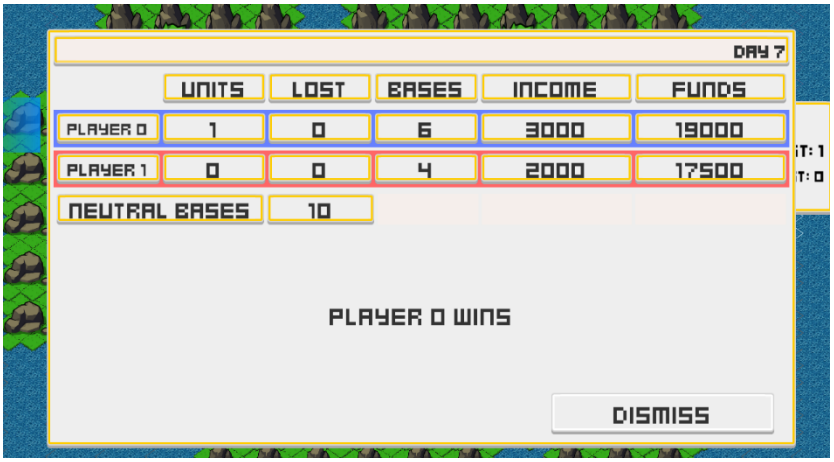
- **Pasos:** Iniciar el movimiento de las unidades y luego atacar a otras unidades.
- **Resultados esperados:** Verificar que las unidades se muevan correctamente, mostrar correctamente el rango de movimiento y los objetivos que se pueden atacar, y asegurar que las unidades respondan adecuadamente según las reglas del juego.
- **Resultado:** Aprobado

Prueba de victoria del juego:

Pasos: Un jugador gana al capturar la base del oponente.

Resultados esperados: el juego determina correctamente las condiciones de victoria y muestra la interfaz de victoria.

Resultado: Aprobado**.**

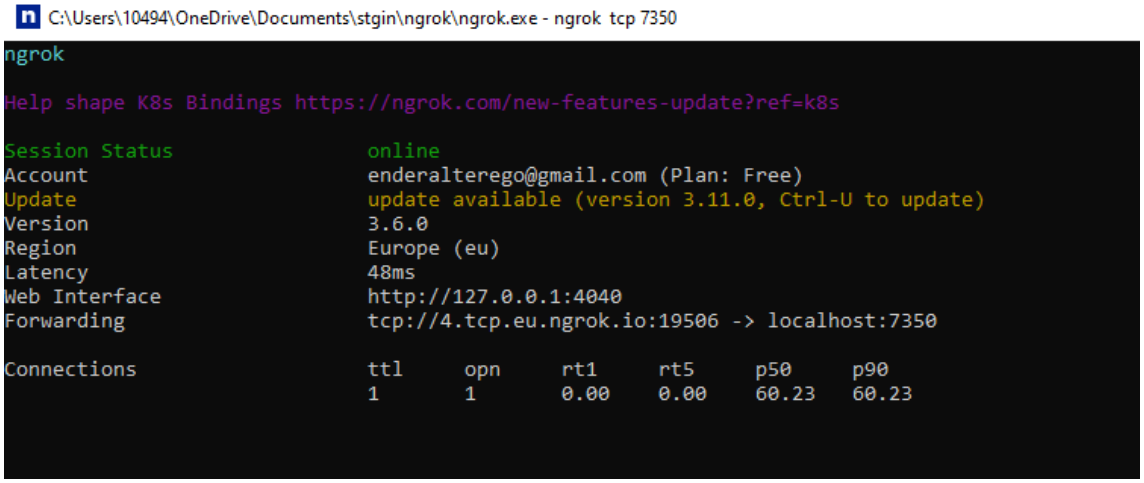


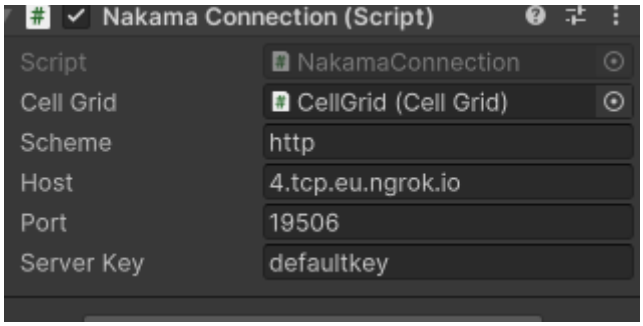
prueba ngrok:

Pasos: use ngrok para exponer el servidor local y los jugadores se conectarán al servidor del juego a través de la URL proporcionada por ngrok.

Resultados esperados: los jugadores pueden conectarse con éxito a servidores expuestos a través de ngrok y jugar normalmente.

Resultado: Aprobado.





5.1.2 Resultados de las pruebas

Todas las pruebas funcionales han pasado y las funciones principales del juego, como iniciar sesión, crear salas, unirse a salas y partidas rápidas, pueden funcionar con normalidad. Además, las funciones de creación y unión de salas privadas, así como el juicio de las condiciones de victoria del juego, también funcionaron normalmente sin mayores errores.

5.2 Evaluación y problemas conocidos

En esta sección, enumeraremos los problemas conocidos descubiertos durante el proceso de prueba y propondremos los planes de mejora correspondientes.

5.2.1 Problemas conocidos

Los jugadores no pueden volver a conectarse a juegos anteriores después de salir de un juego:

Descripción: Cuando un jugador sale durante el juego, no puede volver a conectarse al juego anterior, lo que tiene cierto impacto en la experiencia del jugador.

Impacto: Alto.

Sólo hay un mapa, no más opciones de mapas:

Descripción: Actualmente, el juego solo proporciona un mapa, que carece de diversidad y no puede satisfacer las necesidades de los jugadores de diferentes mapas.

Impacto: Medio.

5.2.2 Plan de mejoras

En respuesta a los problemas conocidos anteriormente, hemos desarrollado los siguientes planes de mejora:

Función de reconexión:

Plan: Desarrollar un mecanismo de reconexión que permita a los jugadores volver a conectarse a su juego anterior después de salir.

Prioridad: Alta.

Agregar selección de mapa:

Planifique: diseñe y agregue múltiples mapas para que los jugadores elijan para mejorar la diversidad y la jugabilidad del juego.

Prioridad: Media.

A través de las mejoras anteriores, esperamos mejorar la experiencia de juego general de los jugadores y resolver los principales problemas actuales.

Capítulo 6. Conclusiones

6.1 Evaluación del cumplimiento de objetivos

Para evaluar el cumplimiento de los objetivos, tomaremos como referencia lo establecido en el capítulo 2 de este proyecto. El objetivo principal de este proyecto fue diseñar y desarrollar un juego de estrategia por turnos, implementando funciones clave como la gestión de recursos, producción de unidades, captura de puntos de control y un sistema de turnos. Además, el proyecto buscaba proporcionar una interfaz de usuario intuitiva y realizar pruebas funcionales para asegurar la calidad del juego.

Este objetivo principal se desglosa en los siguientes puntos, que nos permiten evaluar de manera más precisa el cumplimiento del mismo:

- **Diseño y desarrollo del sistema de producción de unidades**:
  - Se ha detallado y desarrollado un sistema de producción de unidades dentro del juego, permitiendo a los jugadores crear y gestionar sus unidades según los recursos disponibles y las necesidades estratégicas. La lógica de producción ha sido implementada y probada exitosamente.
- **Gestión de recursos y captura de puntos de control**:
  - Se ha implementado un sistema de gestión de recursos que permite a los jugadores controlar sus ingresos y gastos durante el juego. Además, se ha desarrollado un sistema de captura de puntos de control que influye en el control de recursos y la estrategia del juego.
- **Implementación del sistema de turnos**:
  - Se ha desarrollado un sistema de turnos que gestiona la secuencia de acciones de los jugadores, asegurando que cada jugador tenga su turno para realizar movimientos y acciones. Este sistema ha sido probado y funciona de acuerdo con las expectativas.
- **Interfaz de usuario y experiencia de juego**:
  - Se ha diseñado y desarrollado una interfaz de usuario que permite a los jugadores interactuar con el juego de manera intuitiva. La funcionalidad de inicio de sesión, creación de salas, unirse a salas y selección de jugadores ha sido implementada y probada.
- **Pruebas funcionales**:
  - Se han realizado pruebas funcionales exhaustivas, incluyendo pruebas de inicio de sesión, creación de salas, unirse a salas, partidas rápidas, inicio de juego, y condiciones de victoria. Todos estos casos de prueba han sido documentados y los resultados indican que el juego funciona correctamente en todos los aspectos críticos.

En resumen, se han cumplido todos los objetivos marcados y el proyecto ha sido una experiencia gratificante, ya que considero que lo realizado en este proyecto puede ser de gran utilidad en el desarrollo de juegos de estrategia por turnos.

6.2 Propuestas para futuros trabajos

Como propuesta para futuros trabajos, se presentan diversas oportunidades de mejora y expansión del proyecto actual:

- **Implementación de la funcionalidad de reconexión**:
  - Desarrollar una funcionalidad que permita a los jugadores reconectarse a una partida en curso en caso de desconexión, mejorando así la experiencia del usuario y la continuidad del juego.
- **Añadir más mapas y opciones de personalización**:
  - Diseñar y desarrollar múltiples mapas para aumentar la diversidad y la rejugabilidad del juego. Además, permitir la personalización de los mapas y las condiciones de victoria podría atraer a una audiencia más amplia.
- **Optimización del rendimiento**:
  - Revisar y optimizar el código y los procesos del juego para asegurar un rendimiento óptimo en diversas plataformas y dispositivos.

Estas mejoras y expansiones no solo aumentarán la calidad y el atractivo del juego, sino que también proporcionarán una base sólida para futuros desarrollos y adaptaciones en el ámbito de los juegos de estrategia por turnos.

Capítulo 7. Bibliografía y referencias

[1] 2D Isometric Village https://assetstore.unity.com/packages/2d/environments/2d-isometric-village-178345

[2] 2D Isometric Tile Starter Pack https://assetstore.unity.com/packages/2d/environments/2d-isometric-tile-starter-pack-27944

[3] Turn Based Strategy Framework https://assetstore.unity.com/packages/templates/systems/turn-based-strategy-framework-50282

[4] 2D Pixel Unit Maker https://assetstore.unity.com/packages/2d/characters/2d-pixel-unit-maker-spum-188715