Índice general

1 Memoria Descriptiva del Juego Nexa **1**
1.1 Introducción . . . 1
1.2 Arquitectura y Plataforma Tecnológica . . . 1
1.2.1 Arquitectura del Sistema . . . 1
1.3 Funcionalidades del Sistema . . . 1
1.4 Proceso de Instalación . . . 2
1.5 Roles de Usuario y Permisos . . . 2
1.6 Gestión de Energía y Recursos . . . 2
1.7 Condiciones de Victoria . . . 2
1.8 Sistema de Colisiones y Conflictos . . . 3
1.9 Arquitectura Técnica Detallada . . . 3
1.9.1 GameController (Infrastructure) . . . 3
1.9.2 TickService (Application) . . . 3
1.9.3 CollisionService (Application) . . . 3
1.9.4 CaptureService (Application) . . . 3
1.9.5 VictoryService (Application) . . . 3
1.9.6 GameStateManagerService (Application) . . . 4
1.10 Tipos de Nodos Especializados . . . 4
1.11 Implementación de Testing . . . 4
1.12 Configuración y Constantes . . . 4
1.13 Sistema de Path Aliases . . . 4
1.14 Despliegue y Producción . . . 5
1.15 Limitaciones del Sistema . . . 5
1.16 Estructura de Directorios . . . 5
1.17 Mantenimiento y Versionado . . . 6
1.18 Conclusiones . . . 6

1 Memoria Descriptiva del Juego Nexa

1.1 Introducción

La presente memoria descriptiva documenta el funcionamiento, instalación y uso del juego Nexa, desarrollado bajo el framework Phaser 3 y utilizando TypeScript con arquitectura limpia. Este juego es una herramienta de entretenimiento estratégico que permite a dos jugadores competir por el control de nodos en un grafo mediante la gestión de recursos de energía. Los usuarios principales son jugadores que buscan experiencias de estrategia en tiempo real con mecánicas innovadoras basadas en teoría de grafos.

1.2 Arquitectura y Plataforma Tecnológica

Nexa es un juego web desarrollado en TypeScript sobre el framework de juegos Phaser 3.90. Utiliza Vite como build tool y está diseñado para funcionar en navegadores web modernos con soporte para Canvas HTML5. El entorno de ejecución incluye Node.js 18.x, pnpm 8.x como gestor de paquetes y TypeScript 5.7 con configuración estricta.

1.2.1 Arquitectura del Sistema

El proyecto implementa Clean Architecture con cuatro capas claramente definidas:

Core Layer (Dominio): - Entidades: Node, Edge, Player, Graph, EnergyPacket - Value Objects: ID, NodeType, Color - Sin dependencias externas

Application Layer (Casos de Uso): - Servicios: TickService, CollisionService, CaptureService, VictoryService - Interfaces y contratos entre capas - Lógica de negocio del juego

Infrastructure Layer (Adaptadores): - GameController: Orquestador principal - GameRenderer: Adaptador de Phaser - GameFactory: Inyección de dependencias

Presentation Layer (UI): - Escenas de Phaser: Boot, MainMenu, Game, GameOver - Manejo de eventos de usuario

1.3 Funcionalidades del Sistema

Nexa permite la gestión completa de partidas de estrategia en tiempo real sobre grafos. Sus principales funcionalidades incluyen:

- Registro y gestión de dos jugadores en partida 1v1.
- Generación procedural de grafos con diferentes topologías.
- Sistema de energía conservativa con distribución entre nodos y aristas.
- Cuatro tipos de nodos especializados (Básico, Ataque, Defensa, Energía).
- Sistema de colisiones con resolución en tiempo real.
- Detección automática de tres condiciones de victoria diferentes.
- Sistema de fragmentación de grafo mediante detección de nodos de articulación.
- Envío de paquetes de energía con intervalos configurables por tipo de nodo.

1.4 Proceso de Instalación

La instalación de Nexa requiere la configuración de un entorno de desarrollo con Node.js y pnpm. Posteriormente, se deben realizar los siguientes pasos:

1. Instalar Node.js 18.x o superior y pnpm 8.x en el sistema operativo.
2. Clonar el repositorio desde GitHub.
3. Instalar las dependencias del proyecto con pnpm install.
4. Configurar las variables de entorno si es necesario.
5. Ejecutar el servidor de desarrollo con pnpm run dev.
6. Acceder al juego vía navegador web en http://localhost:8080.
7. Para producción, compilar con pnpm run build y servir la carpeta dist/.

1.5 Roles de Usuario y Permisos

Los usuarios del sistema tienen dos roles principales:

- Jugador 1: controla nodos de color azul, inicia en un nodo específico del grafo, gestiona la distribución de su energía total entre sus nodos controlados.
- Jugador 2: controla nodos de color rojo, inicia en un nodo opuesto del grafo, gestiona su pool de energía independiente del jugador contrario.

Ambos jugadores tienen las mismas capacidades: seleccionar nodos propios, asignar energía de ataque a aristas hacia nodos vecinos, defender nodos automáticamente con energía no asignada, y capturar nodos enemigos cuando su ataque supera la defensa.

1.6 Gestión de Energía y Recursos

Los jugadores pueden distribuir su energía total entre sus nodos controlados. Cada nodo tiene propiedades específicas según su tipo: intervalos de ataque (frecuencia de emisión de paquetes), intervalos de defensa (frecuencia de regeneración), multiplicadores de ataque y defensa, y energía inicial que aporta al ser capturado.

El sistema implementa conservación de energía: la suma de energía en todos los nodos más la energía en tránsito por aristas se mantiene constante. Los paquetes de energía viajan por las aristas hasta su destino, donde pueden colisionar con paquetes enemigos o atacar nodos enemigos si superan su defensa.

1.7 Condiciones de Victoria

El sistema verifica continuamente tres condiciones de victoria:

Victoria por Dominación: Un jugador debe controlar al menos el 70% de los nodos del grafo de forma sostenida durante 10 segundos consecutivos. El sistema trackea el tiempo de dominación de cada jugador y reinicia el contador si pierde la mayoría.

Victoria por Tiempo: Si transcurren 3 minutos (180 segundos) sin que ningún jugador alcance otra condición, gana quien controle la mayor cantidad de nodos al finalizar el tiempo. En caso de empate, la partida termina en empate.

Victoria por Eliminación: Si un jugador pierde su nodo inicial (base), pierde inmediatamente la partida. Esta es una condición de derrota automática que termina el juego sin importar otros factores.

1.8 Sistema de Colisiones y Conflictos

El sistema implementa un algoritmo determinístico para resolver conflictos entre paquetes de energía:

Colisión en aristas: Cuando dos paquetes enemigos se encuentran en la misma arista, si tienen igual magnitud se destruyen ambos. Si tienen diferente magnitud, el mayor continúa con energía igual a la diferencia.

Ataque a nodos: Cuando un paquete de energía alcanza un nodo enemigo, si la energía de ataque es mayor que la defensa actual del nodo, el nodo es capturado y la energía sobrante se convierte en su nueva defensa. Si son iguales, el nodo queda sin propietario (neutral). Si el ataque es menor, se destruye y la defensa se reduce.

Fragmentación de grafo: Al capturarse un nodo de articulación (nodo crítico que conecta componentes del grafo), el grafo puede fragmentarse. El jugador solo conserva los nodos en el componente conexo que contiene su nodo inicial. Los nodos en componentes desconectados quedan sin propietario.

1.9 Arquitectura Técnica Detallada

1.9.1 GameController (Infrastructure)

Actúa como Facade entre la capa de presentación y la capa de aplicación. Coordina el game loop principal impulsado por Phaser, procesa eventos de entrada del usuario, sincroniza TickService, VictoryService y GameStateManager, y maneja el inicio, pausa y finalización de partidas.

1.9.2 TickService (Application)

Implementa el game loop del juego ejecutándose en cada frame. Actualiza la defensa de nodos cada 30ms, emite paquetes de energía según intervalos de cada nodo, avanza paquetes en las aristas, detecta y resuelve colisiones, procesa llegadas a nodos destino, y ejecuta capturas cuando corresponde.

1.9.3 CollisionService (Application)

Detecta colisiones entre paquetes de energía agrupándolos por arista. Resuelve conflictos según las reglas definidas: destrucción mutua en empate, sobrevivencia del mayor con diferencia, y advertencia en caso de colisión entre paquetes aliados opuestos.

1.9.4 CaptureService (Application)

Maneja la captura de nodos verificando si el ataque supera la defensa. Aplica efectos especiales del nodo capturado (energía adicional, multiplicadores). Detecta nodos de articulación usando el algoritmo de Tarjan con complejidad O(V + E). Gestiona la fragmentación eliminando nodos desconectados del nodo inicial del jugador.

1.9.5 VictoryService (Application)

Verifica las tres condiciones de victoria en cada tick. Trackea el tiempo de dominación acumulado por cada jugador. Calcula el porcentaje de nodos controlados. Genera resultados de victoria con estadísticas finales cuando se cumple alguna condición.

1.9.6 GameStateManagerService (Application)

Mantiene el estado inmutable del juego. Genera snapshots para la capa de presentación sin exponer entidades mutables. Calcula estadísticas derivadas como porcentajes de control y energía total. Gestiona contadores de tiempo, ticks y trackers de dominación.

1.10 Tipos de Nodos Especializados

Nodo Básico: Intervalo de ataque 2000ms, intervalo de defensa 3000ms, multiplicadores 1x para ataque y defensa, energía inicial 20 unidades. Funcionalidad estándar sin bonificaciones.

Nodo de Ataque: Intervalo de ataque reducido a 1000ms, intervalo de defensa 3000ms, multiplicador de ataque 2x (duplica energía enviada), multiplicador de defensa 1x, energía inicial 20 unidades. Ideal para ofensivas agresivas.

Nodo de Defensa: Intervalo de ataque 2000ms, intervalo de defensa reducido a 1500ms, multiplicador de ataque 1x, multiplicador de defensa 2x (duplica protección), energía inicial 20 unidades. Ideal para posiciones defensivas.

Nodo de Energía: Intervalos estándar de 2000ms ataque y 3000ms defensa, multiplicadores 1x, pero energía inicial de 50 unidades. Capturarlo otorga un boost significativo de recursos al jugador.

1.11 Implementación de Testing

El proyecto incluye una suite completa de tests unitarios con Vitest 4.0. Se implementan tests para todos los servicios principales: GameStateManager con 20 tests de estado y snapshots, CaptureService con tests de articulación y fragmentación, VictoryService con verificación de las tres condiciones, CollisionService con múltiples escenarios de conflicto.

Los tests se ejecutan con pnpm test y garantizan la estabilidad del sistema. Se utiliza ESLint 9 con reglas estrictas de TypeScript para mantener calidad de código. La cobertura de tests supera el 75% del código base.

1.12 Configuración y Constantes

El sistema define constantes críticas en GAME_CONSTANTS:

- DOMINANCE_PERCENT: 70 (porcentaje requerido para victoria por dominación)
- DOMINANCE_DURATION_MS: 10000 (10 segundos sostenidos de dominación)
- TIME_LIMIT_MS: 180000 (3 minutos de límite temporal)
- DEFAULT_SPEED: 0.002 (velocidad de paquetes de energía)
- ASSIGNMENT_AMOUNT: 10 (cantidad base de asignación de energía)

La configuración de Phaser establece resolución de 1024x768, modo de escalado FIT con centrado automático, física arcade sin gravedad, y cuatro escenas principales.

1.13 Sistema de Path Aliases

El proyecto utiliza path aliases de TypeScript para mejorar la legibilidad:

- @/ apunta a src/
- @/core apunta a src/core/
- @/application apunta a src/application/
- @/infrastructure apunta a src/infrastructure/
- @/presentation apunta a src/presentation/

Esto permite imports limpios sin rutas relativas complejas y facilita refactorizaciones futuras.

1.14 Despliegue y Producción

Para desplegar en producción se ejecuta pnpm run build, lo que genera archivos optimizados en la carpeta dist/. Estos archivos pueden servirse desde cualquier servidor web estático: GitHub Pages, Netlify, Vercel, o servidor propio con Apache/Nginx.

El build de producción aplica minificación, tree-shaking, code-splitting y optimización de assets. El sistema de logging se configura automáticamente según el entorno (detallado en desarrollo, mínimo en producción).

1.15 Limitaciones del Sistema

El sistema actual tiene las siguientes limitaciones conocidas:

- Soporte limitado a 2 jugadores (partidas 1v1).
- El grafo es estático y se genera al inicio de la partida.
- No hay persistencia automática de partidas.
- Rendimiento óptimo con 15-30 nodos en el grafo.
- Requiere resolución mínima de 1280x720 para experiencia completa.
- No incluye sistema de matchmaking o juego en línea.

1.16 Estructura de Directorios

```
nexa/
  src/
      core/                    # Domain Layer
          entities/           # Node, Edge, Player, Graph, EnergyPacket
          types/              # ID, NodeType, Color
      application/             # Application Layer
          services/           # TickService, CollisionService, etc.
          interfaces/         # Contratos entre capas
          strategies/         # Algoritmos intercambiables
          constants/          # Constantes del sistema
      infrastructure/          # Infrastructure Layer
          game/               # GameController, GameFactory
          renderer/           # GameRenderer (Phaser adapter)
          implementations/    # Implementaciones concretas
          logging/            # Sistema de logs
      presentation/            # Presentation Layer
           scenes/            # BootScene, MainMenuScene, etc.
  public/                      # Assets estáticos
  tests/                       # Suite de tests unitarios
  docs/                        # Documentación del proyecto
  vite/                        # Configuración de Vite
```

1.17 Mantenimiento y Versionado

El proyecto sigue Semantic Versioning (SemVer): versión mayor para cambios incompatibles, versión menor para nuevas funcionalidades compatibles, versión patch para correcciones de bugs.

El workflow de Git utiliza rama main para producción, rama dev para desarrollo activo, ramas feature/* para nuevas funcionalidades, y ramas fix/* para correcciones. Se requiere Pull Request y revisión de código antes de merge a main.

Las dependencias se actualizan periódicamente verificando con pnpm outdated y aplicando pnpm update. Se mantiene documentación actualizada en el directorio docs/ y se siguen Conventional Commits para mensajes de commit claros.

1.18 Conclusiones

Nexa es una solución integral para juegos de estrategia en tiempo real basados en grafos, con un sistema escalable y accesible desde cualquier navegador moderno. Su implementación con Clean Architecture permite separación clara de responsabilidades, alta testabilidad y facilidad de mantenimiento.

El uso de TypeScript con tipado estricto garantiza robustez del código y prevención de errores en tiempo de compilación. La suite completa de tests proporciona confianza en la estabilidad del sistema y facilita la integración continua de nuevas características.

La arquitectura en capas con inversión de dependencias permite que cada capa sea independiente y testeable, facilitando la evolución futura del sistema sin comprometer la estabilidad. El sistema de colisiones determinístico y la detección de fragmentación de grafos demuestran la aplicación práctica de algoritmos avanzados en un contexto de entretenimiento.

Nexa representa un ejemplo académico de aplicación de principios de ingeniería de software en el desarrollo de juegos, demostrando que las buenas prácticas de arquitectura son aplicables más allá de aplicaciones empresariales tradicionales.