UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

Curso:

Programación 2

Documento Técnico Proyecto Final: Programación 2

Integrantes:

Cárdenas Carreras, Dahlia Carina

Cárdenas Soto, Tadeo Joaquín

Rodenas Rocha, Oscar Daniel

Yañez Encalada, Aldrian Alejandro

Profesor:

Rivas Medina, Ruben Demetrio

28 de noviembre del 2025

Índice

Descripción de arquitectura:

El proyecto Rival Frontiers se construyó siguiendo un modelo de Arquitectura Modular y Orientada a Objetos (POO). Este modelo divide el juego en tres capas lógicas (Dominio, Contexto y Control), garantizando que las reglas del juego estén completamente separadas de la lógica que lo ejecuta y de la entrada del usuario.

El código está organizado en una estructura de directorios estándar de C++ para separar las interfaces de las implementaciones de las clases, lo cual es vital para la modularidad.

- Carpeta include: Esta carpeta contiene todas las cabeceras (.h) del proyecto. Cada archivo .h actúa como un contrato que declara las clases, sus atributos y los métodos que la clase promete implementar.
- Carpeta src: Esta carpeta contiene los archivos fuente (.cpp). Aquí se escribe la lógica real de cada método declarado en las cabeceras. Esta separación permite que el código sea compilado y mantenido de forma independiente por módulos.
- El archivo main.cpp es el punto de inicio del juego. El archivo CMakeLists.txt gestiona la compilación, indicando cómo enlazar todos los archivos de src con main.cpp y dónde buscar las cabeceras.

El sistema está organizado en tres capas lógicas. Cada capa usa sus propias cabeceras (.h) y archivos de implementación (.cpp) para cumplir funciones específicas.

A. Dominio: Modelo del Mundo

El Dominio es la capa que define todas las entidades fundamentales del juego. Su objetivo es modelar el mundo usando POO (Herencia, Polimorfismo y Encapsulación) para representar terrenos, unidades, edificios, recursos y posiciones. Aquí se definen las reglas base del juego, de forma independiente al motor o a los controles.

Archivos del Dominio y su función

- **Terreno.h**
  Define toda la estructura responsable de modelar los distintos tipos de terrenos del mapa en el juego. Esta parte del sistema es esencial porque determina cómo afecta el entorno al movimiento, defensa y estrategia de las unidades. En este archivo se

establece la clase base Terreno, que funciona como un molde para todos los terrenos existentes. Contiene atributos como un código de dos letras que identifica cada tipo de terreno dentro del mapa, y define dos funciones virtuales puras: costo_movimiento y bono_defensa

- **Unidad.h / Unidad.cpp**

  Unidad.h define la clase Unidad y sus derivadas (Soldado, Mago, Ingeniero).

  Unidad.cpp contiene las implementaciones de los métodos específicos de cada tipo de unidad (mover, atacar, habilidad especial) y las interacciones con el resto del motor (Contexto, Mapa, Celda, Recursos).

- **Edificio.h / Edificio.cpp**

  Edificio.h define la estructura general de todos los edificios del juego. Contiene la clase base Edificio, que establece los atributos esenciales (tipo y propietario) y declara dos funciones clave que todos los edificios.

  Edificio.cpp implementa los comportamientos definidos en el encabezado. Cada edificio aporta recursos diferentes a la facción que lo controla a través del método efecto_turno

- **Recursos.h**

  Define la estructura encargada de manejar los diferentes recursos que intervienen en la economía del juego. Aquí se encuentra la clase Recursos, que representa la cantidad de comida, metal y energía disponibles para un jugador o un edificio. Estos recursos juegan un papel fundamental, ya que determinan si una unidad puede activar una habilidad, si un edificio puede ser construido, o si una acción estratégica puede llevarse a cabo durante un turno.

- **Coordenada.h**

  Representa posiciones del mapa usando operadores sobrecargados para comparaciones y salidas.

- **Celda.h**

  Representa cada espacio del mapa.

  Usa punteros inteligentes (shared_ptr) para almacenar Terreno, Unidad o Edificio de forma segura.

- **Faccion.h**

  Modela a cada jugador: recursos, moral y zonas controladas.

B. Contexto: Gestor Global de Estado

El Contexto es la capa encargada de mantener el estado global de la partida. Sirve como un motor central que reúne información del Dominio y coordina los datos necesarios para ejecutar la partida.

Archivos del Contexto

- **Mapa.h / Mapa.cpp**

  Mapa.h define la estructura fundamental que representa el escenario principal del juego. En esta clase se administra el mapa completo como una matriz de objetos Celda, cada una conteniendo información del terreno, las unidades, edificios u otros elementos presentes.
  Mapa.cpp contiene la implementación completa del comportamiento descrito en el encabezado. Uno de los elementos más importantes es el constructor del mapa. Aquí se generan todas las celdas que lo forman y se decide, celda por celda, qué tipo de terreno tendrá cada una. En lugar de asignar terrenos de manera aleatoria, el constructor utiliza condiciones específicas (como las coordenadas exactas de la celda) para colocar distintos tipos de terreno: pantanos, bosques, montañas, agua o llanuras.

- **Contexto.h / Contexto.cpp**
  Contexto.h declara la estructura principal del juego: mapa, facciones y lista de eventos.
- **Contexto.cpp** actúa como el cerebro del sistema, sincronizando los módulos del Dominio con la lógica del Control.

C. Control: Manejo de Fases y Eventos

La capa de Control maneja el flujo del tiempo y las acciones. Su misión es alternar entre las acciones del jugador y las rutinas automáticas del sistema (la IA).

Archivos del Control

- **Controlador.h / Controlador.cpp**
  Controlador.h define la clase base Controlador y sus derivadas: ControladorJugador y

ControladorSistema.
Controlador.cpp implementa la lógica de la IA: ejecutar_patrullaje(), ejecutar_ofensiva(), ejecutar_defensa().

- **Evento.h / Evento.cpp**
  Evento.h define la clase base Evento para crear efectos temporizados.
  Evento.cpp implementa ejecutar(), donde se definen efectos como:
  - EventoRefuerzo: Crea unidades
  - EventoClima: Afecta moral o atributos del Contexto)

- **main.cpp**
  Coordina el flujo principal: ejecuta las fases del jugador, luego la fase del sistema, y finalmente llama a Contexto::procesar_eventos().

2. Sistemas de Eventos

Se conoce que el proyecto debe incluir un motor de eventos con cronogramas, disparadores y rutinas que reaccionen al estado del mapa. El código desarrollado implementa este sistema dentro del **contexto** y de las fases automáticas del sistema, logrando un comportamiento que impacta directamente en el curso de la partida.

2.1. Modelo de eventos en el juego

- procesar_eventos_turno(ctx)
- ctx.procesar_eventos()
- crtl_sistema.resolver_fase(ctx)
- generar_mision_aleatoria() y verificar_misiones_secundarias()

2.2. Tipos de eventos

- Eventos ambientales y globales

- Eventos económicos
- Eventos del sistema
- Eventos narrativos (misiones secundarias)

2.3. Flujo de ejecución de eventos

1. Fin de acciones del jugador
2. Reset de habilidades
3. Mantenimiento y penalizaciones de moral
4. Producción de recursos
5. Captura de neutrales
6. Procesamiento de eventos del mundo (procesar_eventos_turno)
7. Rutinas del sistema
8. Reactivación de unidades
9. Verificación de misiones y objetos
10. Evaluación de victoria o derrota

2.4. Lógica interna del motor de eventos

1. Eventos programados: clima, scripts generales
2. Eventos económicos automáticos: producción, mantenimiento
3. Eventos reactivos: capturas automáticas, misiones
4. Eventos de la IA enemiga: movimiento, refuerzos, ataques
5. Actualizaciones finales: reactivación, habilidades, objetivos

3. Balance

3.1. Economía inicial y progresión

El jugador inicia con:

- Comida = 18
- Metal = 12
- Energía = 7

Luego, aumenta la producción por turno:

- Granja: +2 comida

- Forja: +1 metal
- Torre: +1 energía, visión
- Cuartel: habilita reclutamiento

3.2. Economía de acciones

El sistema impone 2 puntos de acción por turno que se consumen en mover unidades, reclutar, construir, usar habilidades, desbloquear mejoras y la construcción rápida del ingeniero. Mientras tanto, la economía mantiene una presión constante mediante los costos de unidades (comida + metal), costos de edificios (metal + energía) y el sistema de mantenimiento (comida).

3.3. Balance de combate

El combate usa una fórmula equilibrada entre ataque base, mejoras, moral y defensa:

dano_final = max(1, dano_con_moral - (def_base + bono_terreno))

3.4. Balance de crecimiento

- Capturar neutrales otorga ventaja pero arriesga recursos.
- El mantenimiento limita el exceso de unidades.
- Las rutinas de guardianes contrarresten la expansión del jugador.
- El dominio se convierte en condición real de victoria o derrota.

3.5. Balance de Sistemas del Mundo (IA)

- Rutinas del Guardián **(**Visto en Controlador.cpp): El ControladorSistema ejecuta rutinas de ejecutar_ofensiva, ejecutar_patrullaje y ejecutar_defensa.

Justificación: Estas rutinas sirven como el principal contrapeso a la expansión del jugador. Al ejecutarse en la Fase del Sistema (fuera de los 2 PA del jugador), la IA garantiza que el jugador siempre estará bajo presión para defender sus fronteras, evitando que la partida se convierta en una victoria fácil y forzando la inversión de recursos en defensa.

4. Casos de prueba

Demostramos que las características clave de la Programación Orientada a Objetos (POO) y los sistemas del juego están funcionando según lo previsto en tu código.

4.1. Polimorfismo en el Terreno

- **Objetivo:** Verificar que el costo de movimiento cambie según la implementación del Terreno de destino.
- **Escenario:** Mover un Ingeniero desde una posición de Llanura a una de Montaña.
- **Resultado Esperado:** El método Ingeniero::mover debe llamar a la versión de costo_movimiento del objeto Montaña. Si la Montaña tiene un costo de 99 (o un valor muy alto), la unidad debe ser incapaz de moverse, demostrando que el mecanismo polimórfico eligió la implementación correcta.

4.2. Flujo de Turno y Efectos de Edificio

- **Objetivo:** Verificar que los efectos de los edificios se ejecuten automáticamente en la Fase del Mundo.
- **Escenario:** El jugador tiene 10 Metal. Construye una Forja y luego finaliza su turno.
- **Resultado Esperado:** Al entrar al Turno 2, los recursos del jugador deben reflejar el aumento de +2 Metal proporcionado por la Forja (total 12 Metal), probando que el callback de producción se ejecutó correctamente en la fase automática.

4.3. Eventos del Sistema

- **Objetivo:** Verificar que la lógica de los eventos (como refuerzos) se ejecute e interactúe dinámicamente con el mapa.
- **Escenario:** El ControladorSistema activa un evento de Refuerzo para la facción J2 (Guardián).
- **Resultado Esperado:** La función EventoRefuerzo::ejecutar debe instanciar nuevas unidades (ej. Soldado) usando std::make_shared y colocarlas en celdas

libres del mapa. Se verifica que el número de unidades de J2 ha aumentado en el mapa.