Manual Técnico del Backend

Sistema: Batalla Naval Algebraica

Basado en los módulos `logic.py` y `state.py`

Contents

1 Introducción **2**

2 Arquitectura del Backend **3**

2.1 Diagrama de Componentes (Textual) 3

3 Documentación del Módulo `state.py` **4**

3.1 Descripción General 4

3.2 Estructura completa del estado global 4

3.3 Función `reset_state()` 4

4 Documentación del Módulo `logic.py` **7**

4.1 Responsabilidades principales 7

4.2 Parser algebraico: _parse_input_function() 7

4.3 Detección de impacto: _curve_hits_cell() 8

4.4 Función principal: `fire_shot()` 8

4.5 Nuevo juego: `new_game()` 9

4.6 Puntaje final 9

5 Diagramas Textuales **10**

5.1 Secuencia: **Jugador dispara una función** 10

6 Ejemplos y Casos de Prueba **11**

6.1 Disparo válido 11

6.2 Disparo inválido 11

6.3 Hundimiento 11

7 Mejoras Recomendadas **12**

Introducción

Este documento describe el backend del videojuego **Batalla Naval Algebraica**, compuesto principalmente por los módulos:

- **logic.py**: núcleo de reglas, evaluación algebraica, disparos, detección de impacto, puntuación, pistas y nuevo juego.
- **state.py**: contenedor del estado global durante toda la ejecución.

Ambos módulos forman la capa lógica que se conecta con:

- La **UI en Pygame** (entrada de usuario y renderizado),
- **helpers** (utilidades matemáticas),
- **ships** (generación de barcos),
- **config** (parámetros por dificultad).

—

Arquitectura del Backend

Diagrama de Componentes (Textual)

- **UI (pygame)** Envía input del usuario, renderiza salidas, llama a `fire_shot()`, `new_game()`, `use_hint()`, etc.
- **logic.py** Ejecuta reglas, matemáticas, detecciones y actualiza el estado.
- **state.py** Contiene el estado global mutado por `logic.py`.
- **config.py** Provee ranges, vidas, umbrales, tolerancias, etc.
- **helpers.py** Utilidades matemáticas (cuadrantes, distancias, pistas).
- **ships.py** Genera barcos y asigna imágenes.

—

Documentación del Módulo `state.py`

Descripción General

El módulo **state.py** define y administra una única estructura global de estado llamada:

`state` : diccionario global modificado por la lógica del juego.

Todo el backend se basa en este **estado centralizado**.

—

Estructura completa del estado global

La Tabla 3.1 resume todas las claves del estado según el código real.

—

Función `reset_state()`

Esta función restaura completamente el estado, configurando vidas, entradas, mensajes, flags y pistas.

```
def reset_state():
    cfg = DIFFICULTY_CONFIG[state["difficulty"]]
    state["ships"] = []
    state["hits"] = []
    state["misses"] = []
    state["attempts"] = 0
    state["hints_used"] = 0
    state["active_hint_index"] = 0
    state["selected_ship_index"] = 0
    state["max_lives"] = cfg["lives"]
    state["lives"] = cfg["lives"]
    state["input_x"] = ""
```

Clave	Descripción
screen_state	"menu", "howto", "playing". Controla la pantalla actual.
difficulty	Dificultad activa. Usada por `config.py`.
ships	Lista de barcos generados automáticamente.
hits	Lista de celdas impactadas exitosamente.
misses	Lista de funciones fallidas (texto).
placing_ships	Modo avanzado para colocar barcos manualmente.
ships_to_place	Número de barcos por colocar manualmente.
lives	Vidas restantes.
max_lives	Máximo de vidas según dificultad.
attempts	Intentos realizados.
hints_used	Número de pistas utilizadas.
active_hint_index	Índice actual del tipo de pista.
selected_ship_index	Barco objetivo de las pistas.
animating	Lista de animaciones activas.
flash_effects	Lista de efectos visuales.
cursor_show	Control del cursor parpadeante.
input_x, input_y	Entradas numéricas (modo avanzado).
input_function	Función ingresada por el jugador.
active_func	Última función confirmada.
active_field	Campo de entrada activo.
msg	Último mensaje del sistema (UI lo muestra).
victoria	Indica si se ganó la partida.
derrota	Indica si se perdió la partida.
reveal_ships	Debug: muestra posiciones reales.
show_ships	Debug: visualización de barcos.

Table 3.1: Estructura del estado global del juego

```
    state["input_y"] = ""
    state["input_function"] = ""
    state["active_func"] = ""
    state["active_field"] = None
    state["msg"] = "Juego reiniciado."
    state["victoria"] = False
    state["derrota"] = False
    state["flash_effects"] = []
    state["animating"] = []
    state["cursor_show"] = True
    state["reveal_ships"] = False
    state["placing_ships"] = False
```

```
    state["ships_to_place"] = 0
```

—

Documentación del Módulo `logic.py`

Responsabilidades principales

- Parseo de funciones matemáticas ingresadas por el jugador.
- Evaluación vectorizada con NumPy.
- Detección geométrica curva–celda.
- Administración de vidas, impactos, hundimientos y fin de partida.
- Generación de pistas.
- Generación de nuevo juego.

—

Parser algebraico: _parse_input_function()

Características

- Soporta $\sin, \cos, \tan, \sqrt{}, \log, e^x$.
- Conversión automática: `sen → sin`, `raiz → sqrt`, ß**.
- `Multiplicación implícita: 2x → 2*x, xsin → x*sin.`
- `Maneja funciones inversas: arcsen → asin.`
- `Vectorizada vía lambdify (NumPy).`

```
f = lambdify(x, sym, modules=["numpy"])
test = f(0)   # Validaci n
return f
```

—

Detección de impacto: _curve_hits_cell()

Evalúa si la curva pasa por una celda usando:

- Comparación directa $f(cx) \approx cy$
- Muestreo denso de puntos
- Detección de intersección segmento-rectángulo

—

Función principal: `fire_shot()`

Flujo

1. Toma el texto en `state["input_function"]`.
2. Lo parsea; si falla, mensaje y return.
3. Genera puntos de graficación para la UI.
4. Prueba impacto contra todos los barcos.
5. Si acierta:
    - Añade a `hits`.
    - Crea animaciones.
    - Revisa hundimiento.
6. Si falla:
    - Añade a `misses`, resta vidas.
    - Calcula distancia al barco más cercano.
    - Crea animación de fallo.
7. Verifica victoria o derrota.

—

Nuevo juego: `new_game()`

```
reset_state()
state["ships"] = generate_ships()
asignar_imagenes_barcos(cell_size)
setup_grid_params()
```

Genera barcos nuevos y reinicia toda la sesión.

—

Puntaje final

$$\text{Score} = \max(0, 100 + 10 \cdot \text{lives} - 5 \cdot \text{attempts} - 2 \cdot |\text{misses}|)$$

—

Diagramas Textuales

Secuencia: Jugador dispara una función

1. Jugador escribe función en UI.
2. UI actualiza `state["input_function"]`.
3. Usuario presiona “Disparar”.
4. UI llama a `fire_shot()`.
5. `fire_shot()`:
   - parsea la función,
   - evalúa puntos,
   - detecta impactos,
   - modifica `state`.
6. UI lee `state["msg"]`, `hits`, `animating`, etc., y renderiza.

—

Ejemplos y Casos de Prueba

Disparo válido

Entrada:

$$f(x) = 2x + 1$$

Resultados esperados:

- Si interseca un barco → impacto.
- Animación naranja.

Disparo inválido

Entrada:

$$f(x) = 2x+$$

Resultado:

- Mensaje: "Función inválida o vacía."
- No se altera el estado del juego.

Hundimiento

Si todas las celdas de un barco aparecen en `hits`, entonces:

- `ship["sunk"] = True`
- Mensaje: "¡Hundiste el barco!"

—

Mejoras Recomendadas

- Reemplazar muestreo fijo (0.1) por muestreo adaptativo.
- Validaciones adicionales de seguridad en parseo.
- Separar estado en una clase con métodos (menos riesgo de mutación accidental).
- Registrar historial detallado para debugging.

—