Battle for the Oceans - Game Bible v2.0

Vision & Heritage

Battle for the Oceans modernizes the classic paper Battleship game of the 1930s into a sophisticated multiplayer naval strategy experience. The game transforms the simple 10x10 grid into diverse maritime theaters with varying grid sizes, terrain types, and fleet compositions across different historical eras.

The core vision extends beyond traditional turn-based gameplay to support:

- **Multi-era naval combat** (Traditional Battleship, Midway Island, Pirates of the Gulf)
- **Alliance-based multiplayer** with dynamic team formation
- **Intelligent AI opponents** with adaptive strategies
- **Responsive cross-platform design** (desktop, mobile, tablet)
- **Real-time and turn-based modes** with configurable rules

Architectural Philosophy

Synchronous State Machine Core

The game employs a **deterministic state machine** as the single source of truth, avoiding race conditions from asynchronous UI operations. State transitions execute business logic **before** React state updates, ensuring consistency.

Dual-Layer Architecture

1. **Game Logic Layer** (synchronous) - immediate access via `gameLogic` object
2. **React UI Layer** (asynchronous) - updates triggered by `forceUIUpdate()`

This separation allows:

- Instant AI decision-making without React delays
- Reliable turn management in multiplayer scenarios
- Consistent state across all UI components

Object Hierarchy

```
Game (orchestrator)
├── Alliance[] (team management)
|   └── Player[] (human/AI instances)
├── Board (spatial logic, terrain, hit resolution)
├── Visualizer (visual state management)
├── PlayerFleets Map<playerId, Fleet>
|   └── Fleet (ship collection per player)
|       └── Ship[] (individual vessels)
└── StateMachine (state transitions)
```

Core Architecture Components

GameContext.js - The Central Nervous System

Purpose: Synchronous game state management with React integration **Key Features**:

- Direct `gameLogic` object manipulation (no React delays)
- Business logic execution before state transitions
- UI update triggering via `forceUIUpdate()`
- Player, fleet, and game instance lifecycle management

Critical Design: GameContext stores game state in a synchronous `gameLogic` object, not React state, enabling instant AI turns and reliable multiplayer coordination.

Game.js - The Orchestrator

Purpose: Core game mechanics, turn management, and AI coordination **Key Features**:

- Unified hit resolution with alliance rules
- Automatic AI turn triggering
- Ship placement registration and validation
- Game rule enforcement (friendly fire, ship capture, turn progression)
- Real-time UI notification system

Architecture Decision: Game class manages both human and AI players through the same interface, with automatic turn progression and AI move execution.

StateMachine.js - State Flow Control

States: launch → login → era → placement → play → over **Events**: LOGIN, SELECTERA, PLACEMENT, PLAY, OVER, ERA

Design Pattern: Business logic executes synchronously before state transitions, ensuring UI consistency and preventing invalid state combinations.

Board.js - Spatial Logic Engine

Purpose: Terrain management, ship placement validation, hit resolution **Key Features**:

- NOAA Chart 1 terrain system (deep, shallow, shoal, marsh, land, rock, excluded)
- Spatial ship tracking via cell mapping
- Shot history with persistent visualization
- Player-specific board views for UI rendering

Critical Function: Board handles all spatial queries and terrain validation, supporting irregular map shapes through excluded cells.

Visualizer.js - Visual State Manager

Purpose: Visual state computation and management separate from game logic **Key Features**:

- Damage ring calculations (red for enemy, blue for own ships)
- Skull indicators for sunk ships
- Shot result tracking for animations
- Clean separation from Board spatial logic

Design Decision: Visualizer provides pre-computed visual data to rendering layer, eliminating complex calculations in UI components.

Player Hierarchy

- **Player.js** - Base class with stats, scoring, elimination logic
- **HumanPlayer.js** - Input handling, notifications, session management
- **AiPlayer.js** - Strategy implementation, memory system, adaptive difficulty

AI Strategy System: Multiple AI personalities (aggressive, defensive, methodical_hunting, opportunistic) with configurable difficulty and adaptive learning.

Fleet & Ship Management

- **Fleet.js** - Ship collection per player, defeat detection, health calculations
- **Ship.js** - Individual vessel with health system, terrain restrictions, placement state

Health System: Ships use floating-point health arrays (0.0-1.0 per cell) supporting partial damage and future enhancement systems.

Alliance System

Purpose: Multi-player team coordination with dynamic membership **Features**:

- Friendly fire prevention
- Alliance ownership and management
- Dynamic team formation during gameplay
- Era-specific alliance configurations

UI Component Architecture

Hooks System

- **useGameState.js** - React integration layer for game state access
- **useBattleBoard.js** - Canvas-based battle visualization with animations

Components

- **FleetPlacement.js** - Ship placement with touch/mouse drag interface
- **FleetBattle.js** - Battle visualization with real-time feedback
- **LoginDialog.js** - Authentication with guest/registered user support

Pages (State Machine Mapping)

- **LaunchPage** → login transition
- **LoginPage** → era selection transition
- **SelectEraPage** → placement transition
- **PlacementPage** → play transition
- **PlayingPage** → over transition
- **OverPage** → era transition (restart)

Era Configuration System

Eras are JSON configurations stored in Supabase defining:

```json
{
  "name": "Traditional Battleship",
  "rows": 10,
  "cols": 10,
  "max_players": 2,
  "terrain": [[...]], // 2D array of terrain types
  "ships": [...], // Fleet composition
  "ai_captains": [...], // Available AI opponents
  "alliances": [...], // Team configurations
  "game_rules": {
    "friendly_fire": false,
    "ship_capture": false,
    "turn_required": true,
    "turn_on_hit": false,
    "turn_on_miss": true
  },
  "messages": {...} // Context-specific flavor text
}
```

Future Eras:

- **Midway Island**: 12x12 grid, WWII Pacific theater
- **Pirates of the Gulf**: 30x20 or 40x30, Gulf of Mexico shape, multiplayer alliances

Configuration Reference

Terrain Types (NOAA Chart 1 Convention)

- **Deep water** - white #FFFFFF - All ships allowed
- **Shallow water** - light blue #E6F3FF - Most ships allowed
- **Shoal water** - medium blue #CCE7FF - Small ships only
- **Marsh** - green #E6F7E6 - Shallow draft vessels
- **Land** - buff #F5F5DC - No ships allowed
- **Rock** - grey #D3D3D3 - No ships allowed
- **Excluded** - transparent - Unplayable cells for irregular map shapes

Game Rules Configuration

- **friendly_fire**: Allow attacking alliance members
- **ship_capture**: Enable capturing sunk enemy ships
- **turn_required**: Enable turn-based vs rapid-fire mode
- **turn_on_hit**: Continue turn when hitting target
- **turn_on_miss**: Continue turn when missing target
- **placement_restriction**: Ship placement area limits for large grids

Ship Configuration Template

```json
{
  "name": "Carrier",
  "size": 5,
  "terrain": ["deep"],
  "view_template": "ships/traditional/carrier/{index}.png"
}
```

Technology Stack

Frontend

- **React 18** with hooks and context
- **HTML5 Canvas** for battle board visualization
- **CSS Grid/Flexbox** for responsive layouts
- **Netlify** deployment with CDN

Backend

- **Supabase** for database (eras, users, scores)
- **PostgreSQL** for relational data
- **Supabase Auth** for user management
- **Brevo** for email communications

Payments & Features

- **Stripe** integration for premium eras
- **Voucher system** for influencer access
- **Feature gating** based on payment status

Game Flow Architecture

State Transitions

1. **Launch** → **Login**: User authentication or guest access
2. **Login** → **Era**: Era selection and opponent choice
3. **Era** → **Placement**: Game initialization and ship placement
4. **Placement** → **Play**: Battle phase with turn management
5. **Play** → **Over**: Game completion and statistics
6. **Over** → **Era**: Restart with new era or replay

Turn Management

- **Turn-based**: Traditional alternating turns with configurable rules
- **Rapid-fire**: Continuous shooting with real-time response
- **Simultaneous**: Future multiplayer mode with sync resolution

AI Integration

AI players integrate seamlessly into the turn system:

- Automatic turn detection and execution
- Strategy-based targeting with memory system
- Configurable reaction times for realistic pacing
- Learning algorithms for adaptive gameplay

Key Design Decisions

Single Board Architecture

Both placement and battle phases use the same Board instance, eliminating data synchronization issues and ensuring visual consistency.

Synchronous Game Logic

Game state lives outside React's asynchronous system, enabling instant AI responses and preventing race conditions in multiplayer scenarios.

Position Data Architecture

Ship placement passes position data as parameters rather than reading from ship objects, maintaining separation of concerns and enabling flexible placement validation.

Enhanced Hit Resolution

The Game class centralizes hit resolution with alliance rules, damage calculation, and ship capture mechanics, supporting future combat enhancements.

Separated Visual State Management

Visualizer class manages visual effects separate from Board spatial logic, enabling simplified rendering and consistent visual feedback.

Canvas-Based Battle Visualization

HTML5 Canvas provides smooth animations, real-time feedback, and scalable rendering across device sizes while maintaining 60fps performance.

Era-Driven Configuration

All game parameters (terrain, rules, ships, AI) are era-configurable, enabling rapid deployment of new game modes without code changes.

Development Standards

Version Management

All files include version numbers and copyright headers for tracking changes and maintaining code consistency across the development team.

Error Handling

Defensive programming with graceful degradation:

- Missing game instance checks
- Terrain validation before ship placement
- Network failure recovery for Supabase operations

Performance Optimization

- Minimal React re-renders through strategic state separation
- Canvas rendering optimizations for smooth animations
- Efficient spatial queries using Map-based cell indexing

Mobile Responsiveness

- Touch-friendly ship placement with drag gestures
- Responsive CSS Grid layouts adapting to screen orientation
- Performance optimization for mobile device constraints

Future Enhancements

Planned Features

- **Real-time multiplayer** with WebSocket synchronization
- **Tournament system** with bracket management
- **Ship upgrade mechanics** with experience systems
- **Environmental effects** (storms, fog, mine fields)
- **Advanced AI** with machine learning adaptation

Technical Roadmap

- **WebRTC** for peer-to-peer multiplayer
- **Redis** for real-time game state synchronization
- **AI/ML** integration for adaptive opponent behavior
- **3D visualization** option with Three.js integration

Ship Visual System

Implementation: Template URL system for ship section images stored in Supabase

Era Config Structure:

```json
{
  "ships": [
    {
      "name": "Carrier",
      "size": 5,
      "terrain": ["deep"],
      "view_template": "ships/traditional/carrier/{index}.png"
    }
  ]
}
```

File Structure: `/ships/{era}/{shiptype}/{index}.png` where index 0 = stern (tap-anchor point)

Architecture Components:

- **Ship.js**: Add `view_template` string property from era config
- **AssetManager**: New class for image loading, caching, and URL resolution
- **ShipRenderer**: New class for compositing Ship data with loaded images for sidebar display

Responsive Display:

- Large screens: Full PNG ship visualization with damage overlays
- Small screens: Compact text format "[5] 40%" for performance
- Post-game: Full visual damage review regardless of screen size

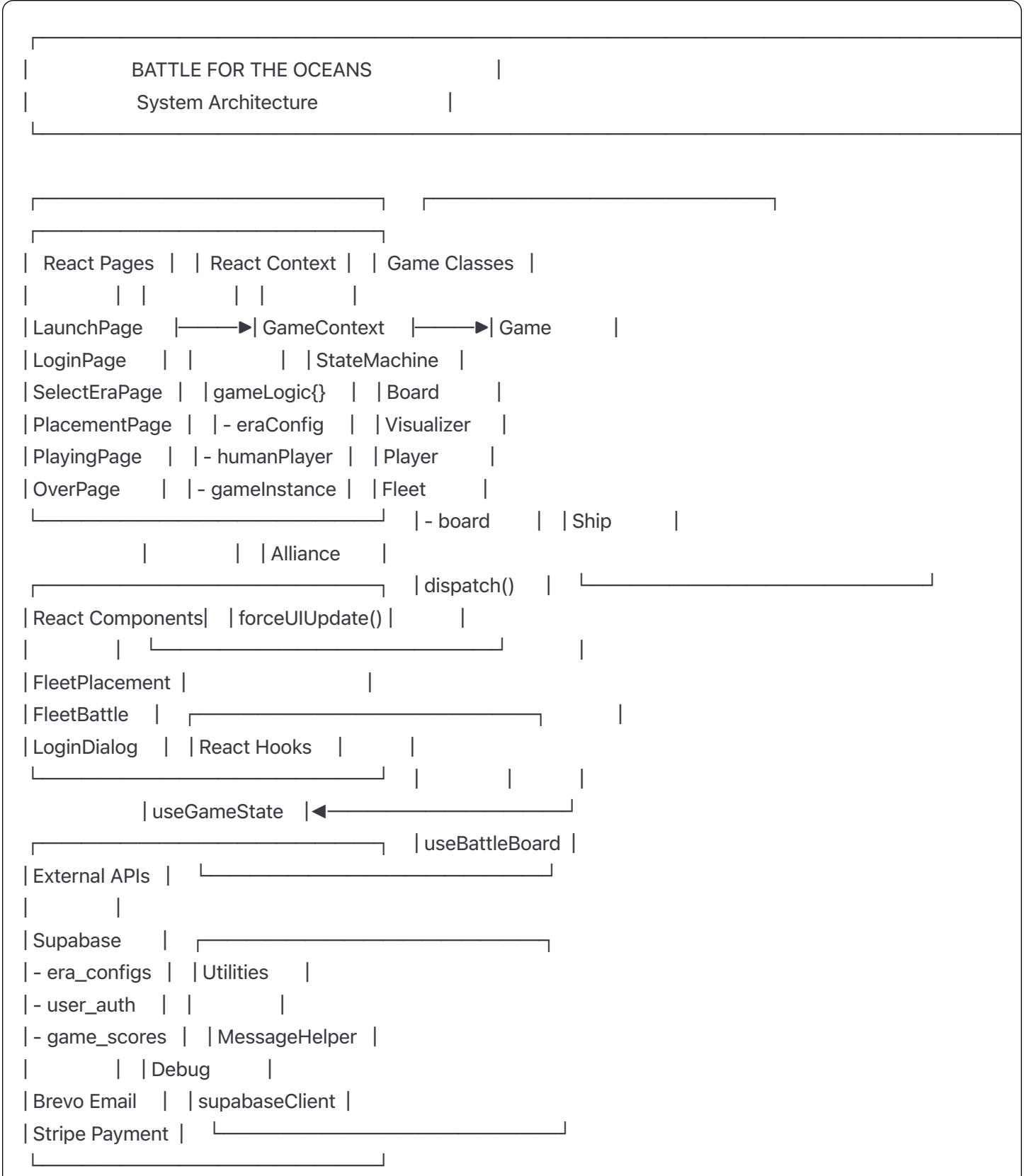
Integration Points:

- Sidebar fleet boxes with ship health visualization
- Damage overlay system for individual ship sections
- Era-specific ship artwork for visual variety

UI Enhancements

- **Fleet Status Sidebar**: Responsive fleet health visualization with expandable alliance groupings
- **3D Layer Toggle**: Optional perspective view of fleet layers for premium visualization
- **Adaptive Board Indicators**: Simplified damage indicators (red/blue circles, skulls) for reduced visual complexity

System Architecture Diagram



Data Flow:

1. Pages trigger events via dispatch()
2. GameContext executes business logic synchronously
3. React hooks provide UI integration layer
4. Game classes handle core mechanics
5. External APIs provide configuration and persistence

Class Architecture & File Specifications

Core Classes (/src/classes/)

Game.js - Game Orchestrator

Purpose: Central game coordinator managing turn flow, hit resolution, and AI integration

Key Data:

- `players[]` - Array of Player instances
- `playerFleets` - Map<playerId, Fleet>
- `alliances` - Map<allianceId, Alliance>
- `cellContents` - Map<"row,col", shipData[]>
- `shipOwnership` - Map<shipId, playerId>
- `gameRules` - Era-specific rule configuration
- `state` - Game phase ('setup', 'placement', 'playing', 'finished')

Key Methods:

- `addPlayer(id, type, name, strategy, difficulty)` - Add human/AI player
- `receiveAttack(row, col, firingPlayer, damage)` - Unified hit resolution
- `registerShipPlacement(ship, shipCells, orientation, playerId)` - Ship placement
- `processPlayerAction(action, data)` - Handle player moves
- `checkAndTriggerAITurn()` - Automatic AI turn management
- `canAttack(firingPlayerId, targetPlayerId)` - Alliance rule validation

Architecture Notes: Handles both human and AI players through unified interface, automatically manages turn progression, provides real-time UI notifications.

Board.js - Spatial Logic Engine

Purpose: Terrain management, ship placement validation, spatial queries

Key Data:

- `terrain[][]` - 2D array of terrain types (deep, shallow, land, etc.)
- `cellContents` - Map<"row,col", shipData[]> for spatial queries
- `shotHistory[]` - Array of attack records with timestamps
- `rows`, `cols` - Board dimensions

Key Methods:

- `canPlaceShip(shipCells, shipTerrain)` - Validate ship placement
- `registerShipPlacement(ship, shipCells)` - Register ship location
- `getShipDataAt(row, col)` - Get ships at coordinates
- `isValidAttackTarget(row, col)` - Validate attack position
- `getPlayerView(playerId, playerFleets, shipOwnership)` - UI board state
- `recordShot(row, col, attacker, result)` - Log attack history

Architecture Notes: Supports irregular map shapes via excluded terrain, maintains persistent shot history, provides player-specific views for UI rendering.

Visualizer.js - Visual State Manager

Purpose: Visual effect computation and management

Key Data:

- `cells[][]` - 2D array of visual state objects
- Cell properties: `redRingPercent`, `blueRingPercent`, `showSkull`, `lastShotResult`

Key Methods:

- `updateCellVisuals(row, col, hitResults, firingPlayer, attackResult)` - Update visual state
- `getVisualState()` - Get complete visual grid for rendering
- `clearAll()` - Reset all visual effects
- `getStats()` - Visual state statistics

Architecture Notes: Provides pre-computed visual data to eliminate complex UI calculations, maintains separation from Board spatial logic.

Player.js - Base Player Class

Purpose: Common player functionality and statistics

Key Data:

- `id`, `name`, `type` - Player identification
- `score`, `shotsFired`, `shotsHit` - Game statistics
- `isEliminated`, `isActive` - Player state
- `color` - Visual identification

Key Methods:

- `onAttacked(attacker, attackResult)` - React to incoming attacks
- `getStats(gameInstance)` - Get player statistics
- `canPlay(gameInstance)` - Check if player can continue
- `reset()` - Reset for new game
- `serialize(gameInstance)` - Export player data

HumanPlayer.js - Human Player Implementation

Purpose: Human player input handling and session management

Key Data:

- `sessionId`, `isOnline`, `lastSeen` - Session tracking
- `pendingAction` - Current input request
- `preferences` - User settings (sounds, animations, etc.)

Key Methods:

- `selectTarget(gameState, availableTargets, timeoutMs)` - Async target selection
- `handleUserInput(inputType, data)` - Process UI interactions
- `showNotification(notification)` - Display user alerts
- `setOnlineStatus(isOnline)` - Handle connection state
- `isResponsive()` - Check if player is active

AiPlayer.js - AI Player Implementation

Purpose: Intelligent AI opponent with strategy and memory

Key Data:

- `strategy` - AI personality ('aggressive', 'methodical_hunting', etc.)
- `difficulty` - Skill level (0.0-1.0+)
- `memory` - AI memory system with target tracking
- `reactionTime` - Calculated thinking delay

Key Methods:

- `makeMove(gameInstance)` - Execute AI turn
- `selectTarget(gameInstance, availableTargets)` - Choose target
- `pickTarget(availableTargets, gameInstance)` - Strategy-based targeting
- `selectCoordinates(target, boardSize, gameInstance)` - Choose attack position
- `processAttackResult(target, row, col, result)` - Update AI memory
- `simulateThinking()` - Realistic delay simulation

Architecture Notes: Multiple AI strategies with adaptive difficulty, memory system for improved targeting, seamless integration with turn management.

Fleet.js - Ship Collection Manager

Purpose: Manage ship collections per player

Key Data:

- `owner` - Player ID
- `ships[]` - Array of Ship instances
- `createdAt` - Fleet creation timestamp

Key Methods:

- `addShip(ship)` - Add ship to fleet
- `addShips(config)` - Bulk add from era config
- `isDefeated()` - Check if all ships sunk
- `isPlaced()` - Check if all ships positioned
- `getHealth()` - Overall fleet health percentage
- `fromEraConfig(owner, eraConfig)` - Create from configuration

Ship.js - Individual Vessel

Purpose: Individual ship with health and placement state

Key Data:

- `name`, `size`, `terrain[]` - Ship characteristics
- `isPlaced` - Placement status
- `health[]` - Per-cell health array (0.0-1.0)
- `sunkAt` - Timestamp when ship was destroyed
- `view_template` - Image template string for visual rendering

Key Methods:

- `place()` - Mark ship as placed
- `receiveHit(index, damage)` - Apply damage to specific cell
- `getHealth()` - Overall health percentage
- `isSunk()` - Check if ship is destroyed
- `reset()` - Reset to initial state
- `fromConfig(config)` - Create from era configuration

Alliance.js - Team Management

Purpose: Multi-player alliance coordination

Key Data:

- `name`, `owner`, `avatar` - Alliance identity
- `players[]` - Array of Player instances
- `createdAt` - Alliance creation time

Key Methods:

- `addPlayer(player)` - Add player to alliance
- `removePlayer(player)` - Remove player from alliance
- `changeOwner(newOwnerId)` - Transfer ownership
- `isDefeated()` - Check if alliance eliminated
- `getName()` - Get display name (substitutes for single-player)

StateMachine.js - State Flow Controller

Purpose: Deterministic state transitions

Key Data:

- `currentState` - Current game state
- `states{}` - State transition map
- `event{}` - Event symbol definitions

Key Methods:

- `transition(event)` - Execute state change
- `getCurrentState()` - Get current state
- `getLastEvent()` - Get last transition event

Context & Hooks (/src/context/, /src/hooks/)

GameContext.js - Central State Manager

Purpose: Synchronous game state with React integration

Key Data:

- `gameLogic{}` - Synchronous game state object
- `uiVersion` - React state version trigger
- `gameStateMachine` - State machine instance

Key Methods:

- `dispatch(event, eventData)` - Execute business logic + state transition
- `updateEraConfig(config)` - Set era configuration
- `updateHumanPlayer(playerData)` - Set authenticated user
- `initializeGame(gameMode)` - Create game instance
- `registerShipPlacement(ship, shipCells, orientation, playerId)` - Ship placement

Architecture Notes: Executes business logic before state transitions, provides immediate access to game state, triggers UI updates via forceUIUpdate().

useGameState.js - Game State Hook

Purpose: React integration layer for game state access

Key Data:

- `gameState` - UI-friendly game state object
- Game instance references and accessors

Key Methods:

- `handleAttack(row, col)` - Process player attacks
- `updateGameState(game)` - Sync with Game instance
- `isValidAttack(row, col)` - Validate attack position
- `getPlayerView(playerId)` - Get board view for rendering

useBattleBoard.js - Canvas Battle Visualization

Purpose: HTML5 Canvas battle board with animations

Key Data:

- `canvasRef` - Canvas element reference
- `animations[]` - Active animation effects
- `shotHistory` - Attack visualization state

Key Methods:

- `drawCanvas()` - Render complete battle board
- `handleCanvasClick(e, onShotFired)` - Process canvas clicks
- `recordOpponentShot(row, col, result)` - Show AI attacks
- `showShotAnimation(result, shooter)` - Animate attack effects

Components (/src/components/)

FleetPlacement.js - Ship Placement Interface

Purpose: Touch/mouse ship placement with drag gestures

Key Methods:

- `handleCellMouseDown/Move/Up()` - Ship placement gestures
- `calculateShipCells()` - Calculate ship position from drag
- `isValidFleetPlacement()` - Validate placement attempt
- `getCellClass()` - CSS class calculation for cell rendering

FleetBattle.js - Battle Board Component

Purpose: Canvas-based battle visualization wrapper

Key Methods:

- Canvas event handling and AI shot notification integration

LoginDialog.js - Authentication Interface

Purpose: User authentication with Supabase integration

Key Methods:

- `handleLogin()` - Process user login
- `handleSignUp()` - Process new user registration
- `handleGuest()` - Guest user authentication

Pages (/src/pages/)

LaunchPage.js - Entry Point

Purpose: Game entry with video intro **Transition**: LOGIN event to login state

LoginPage.js - Authentication Page

Purpose: User authentication portal **Transition**: SELECTERA event with user data to era state

SelectEraPage.js - Era Selection Interface

Purpose: Era and opponent selection **Key Methods**:

- `fetchEras()` - Load era configurations from Supabase
- `handleEraSelect()` - Process era selection
- `handleOpponentSelect()` - Choose AI opponent **Transition**: PLACEMENT event to placement state

PlacementPage.js - Ship Placement Phase

Purpose: Fleet placement interface **Key Methods**:

- `handleShipPlaced()` - Process ship placement
- `handlePlacementComplete()` - Complete placement phase **Transition**: PLAY event to play state

PlayingPage.js - Battle Phase

Purpose: Active battle interface with dual rendering modes **Key Features**:

- CSS Grid battle board for compatibility
- Canvas battle board for enhanced visualization
- Real-time turn management
- Fleet status sidebar with responsive design **Transition**: OVER event to over state

OverPage.js - Game Completion

Purpose: Results display and game restart **Key Methods**:

- `formatGameLog()` - Format battle log for export
- `copyGameLog()` - Copy results to clipboard
- `emailGameLog()` - Email results **Transition**: ERA event to restart, or close application

Utilities (/src/utils/)

MessageHelper.js - Dynamic Message System

Purpose: Era-specific message generation with templates

Key Methods:

- `getRandomMessage(messages)` - Random selection from arrays
- `replaceVariables(message, variables)` - Template variable substitution
- `getGameMessage(eraConfig, context, variables)` - Context-specific messages
- `formatCell(row, col)` - Coordinate formatting (A1, B2, etc.)

Debug.js - Logging Utility

Purpose: Versioned logging system **Methods**: `log()`, `error()`, `warn()` with version prefixes

supabaseClient.js - Database Client

Purpose: Supabase integration with environment validation **Features**: Era configuration loading, user authentication, score persistence

This comprehensive architecture provides a robust foundation for scaling from simple 1v1 battles to complex multiplayer naval warfare while maintaining the strategic depth that made the original Battleship game timeless.

Appendix A: Development To-Do List

High Priority (Current Sprint)

Visual System Enhancement

- **Visualizer Integration**: Add Visualizer instance to Game.js constructor and update receiveAttack() to call visualizer.updateCellVisuals()
- **Board Rendering Simplification**: Update useBattleBoard to consume pre-computed visual state from Visualizer instead of complex analysis
- **Simplified Board Indicators**: Implement red/blue circles and skull indicators for basic damage visualization

Fleet Status Sidebar

- **Responsive Fleet Display**: Create sidebar component showing fleet health with adaptive layout (landscape sidebar, portrait scroll-down)
- **Compact Fleet Format**: Implement "[5] 40%" text format for small screens during active gameplay
- **Fleet Box Expansion**: Add collapsible fleet boxes with alliance grouping for multiplayer scenarios

Medium Priority (Next Sprint)

Ship Visual System

- **Ship.js Enhancement**: Add `view_template` property to Ship class from era configuration
- **AssetManager Class**: Create new class for image loading, caching, and URL resolution
  - Template URL processing: `ships/{era}/{shiptype}/{index}.png`
  - Caching system for loaded ship images
  - Error handling for missing assets
- **ShipRenderer Class**: Create compositor for Ship data with loaded images
  - PNG section rendering with damage overlays
  - Ship orientation handling (horizontal/vertical)
  - Health visualization overlay system

Era Configuration Updates

- **Ship Image Templates**: Update era configs in Supabase with `view_template` fields
- **Asset Storage Structure**: Organize ship PNG assets in Supabase storage following `/ships/{era}/{shiptype}/{index}.png` convention
- **Stern Indexing**: Ensure all ship assets follow index 0 = stern convention for placement consistency

Low Priority (Future Sprints)

3D Visualization

- **Layer Toggle**: Optional 3D perspective view showing fleet layers with sunken ships
- **Three.js Integration**: Implement 3D rendering option for premium visualization
- **Performance Optimization**: Ensure 3D mode doesn't impact core gameplay performance

Advanced Fleet Management

- **Post-Game Ship Review**: Full PNG ship visualization with detailed damage analysis after game completion
- **Battle Damage History**: Visual timeline of ship damage progression
- **Fleet Health Analytics**: Advanced statistics and damage pattern analysis

Multiplayer Enhancements

- **Alliance Sidebar**: Visual alliance management with member fleet status
- **Fleet Communication**: Alliance-based fleet status sharing
- **Spectator Mode**: Observer view with all fleet visualizations

Technical Infrastructure

- **Asset Pipeline**: Automated ship image processing and optimization
- **CDN Integration**: Optimize asset delivery for global performance
- **Mobile Asset Optimization**: Responsive image loading based on device capabilities

Code Quality & Testing

Unit Tests

- **Visualizer Tests**: Comprehensive testing of visual state calculations
- **AssetManager Tests**: Image loading and caching functionality
- **ShipRenderer Tests**: PNG composition and overlay systems

Performance Optimization

- **Mobile Performance**: Optimize sidebar rendering for mobile devices
- **Memory Management**: Efficient PNG caching and cleanup
- **Render Pipeline**: Minimize re-renders during active gameplay

Documentation

- **Component Documentation**: JSDoc for new visual system components
- **Asset Guidelines**: Documentation for creating ship PNG assets
- **Integration Guide**: How to add new ship types and visual assets

Configuration Tasks

Supabase Updates

- **Era Schema**: Add `view_template` fields to ship configurations
- **Asset Storage**: Set up organized folder structure for ship images
- **Migration Scripts**: Update existing era configs with image templates

Asset Creation

- **Traditional Era Ships**: Create PNG sets for carrier, battleship, cruiser, submarine, destroyer
- **Damage Overlays**: Create damage/sunk overlay assets for ship visualization
- **Era-Specific Assets**: Plan ship artwork for Pirates of the Gulf and Midway Island eras

This to-do list serves as the development roadmap, prioritizing visual enhancements and fleet management features that directly improve gameplay experience while maintaining architectural integrity.