Battle for the Oceans - Game Bible v2.1

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
- Alliance-based game-over detection
- Automatic AI turn triggering with proper synchronization
- 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.

Recent Updates:

- Fixed turn progression logic to prevent AI double-firing
- Implemented alliance-based victory conditions
- Enhanced Visualizer integration for simplified UI rendering

StateMachine.js - State Flow Control

States: `launch → login → era → placement → play → over`

Events: `LOGIN`, `SELECTERA`, `PLACEMENT`, `PLAY`, `OVER`, `ERA`, `REPLAY`

Recent Updates:

- Added `REPLAY` event for "Battle Again" functionality
- `over → placement` transition for same-era rematch
- `over → era` transition for new era selection

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 *(NEW)*

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 (red/blue/gray)
- Shot result tracking for animations
- Clean separation from Board spatial logic
- Pre-computed visual data for simplified UI rendering

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

Simplified Visual System:

- **Gray dots**: Player misses (always visible)
- **Blue rings**: Hits on player ships (transparent background)
- **Red rings**: Hits on enemy ships (transparent background)
- **Blue skulls**: Player ships sunk
- **Red skulls**: Enemy ships sunk
- **Gray skulls**: Multiple ships sunk in same cell

Player Hierarchy

Player.js - Base Class

Key Features: Stats, scoring, elimination logic, UUID-based identification

Recent Updates:

- Enhanced UUID support throughout player system
- Improved color generation for visual identification

HumanPlayer.js - Human Player Implementation

Key Features: Input handling, notifications, session management

- Uses Supabase UUID for consistent identification
- Session tracking and online status management

AiPlayer.js - AI Player Implementation

Key Features: Strategy implementation, memory system, adaptive difficulty

- Multiple AI personalities (aggressive, defensive, methodical_hunting, opportunistic)
- Locally-generated UUIDs (no Supabase profiles needed)
- Configurable reaction times for realistic pacing

Turn Management System *(UPDATED)*

Synchronous Turn Logic

The game uses a fully synchronous turn system that prevents race conditions:

1. **Human Turn**: Player action → immediate processing → turn progression
2. **AI Turn**: Turn detection → immediate AI execution → turn progression
3. **Turn Rules**: Era-configurable (turn_on_hit, turn_on_miss, turn_required)

Recent Fixes

- **Eliminated AI double-firing**: Fixed `handleTurnProgression()` to only check for AI turns when turn actually advances
- **Proper turn alternation**: Ensured `checkAndTriggerAITurn()` only executes after confirmed turn changes
- **Alliance-based victory**: Game ends when last alliance is defeated, not last player

AI Delay Strategy

AI delays are character-based, not UX-based:

- **Turn-based games**: Minimal delay for user experience
- **Real-time games**: Character-based delays to balance against human reaction times
- **Multiplayer**: Essential for fair play in rapid-fire modes

Fleet & Ship Management

Fleet.js - Ship Collection Manager

Key Features: Ship collection per player, defeat detection, health calculations

Ship.js - Individual Vessel

Key Features: 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
- Alliance-based victory conditions *(NEW)*

UI Component Architecture

State Machine Integration *(UPDATED)*

All pages now follow consistent state machine dispatch patterns:

- **LaunchPage** → dispatches `LOGIN`
- **LoginPage** → dispatches `SELECTERA`
- **PlacementPage** → dispatches `PLAY`
- **PlayingPage** → dispatches `OVER` *(FIXED)*
- **OverPage** → dispatches `ERA` or `REPLAY` *(UPDATED)*

Key Fix: PlayingPage now automatically dispatches `OVER` when game ends, maintaining consistency with other pages that dispatch state transitions directly rather than using React conditional rendering.

Hooks System

useGameState.js - React Integration Layer

Recent Updates:

- Enhanced to support all PlayingPage requirements
- Simplified game state management
- Improved error handling and loading states

useBattleBoard.js - Canvas Battle Visualization

Key Features: Canvas-based battle visualization with animations **Recent Updates**:

- Simplified rendering using Visualizer pre-computed data
- Performance optimizations for mobile devices
- Fixed skull rendering with proper color coding

Components

FleetPlacement.js - Ship Placement Interface

Purpose: Touch/mouse ship placement with drag gestures

FleetBattle.js - Battle Board Component

Purpose: Canvas-based battle visualization wrapper

LoginDialog.js - Authentication Interface

Purpose: User authentication with Supabase integration

Pages (State Machine Mapping)

OverPage.js *(UPDATED)*

Recent Changes:

- Added "Battle Again" functionality using `REPLAY` event
- Maintains era and opponent settings for immediate rematch
- "Choose New Era" option for complete restart

Transitions:

- `REPLAY` event → `placement` state (same settings)
- `ERA` event → `era` state (new settings)

Era Configuration System

Eras are JSON configurations stored in Supabase defining:

Current Eras:

- **Traditional Battleship**: 10x10 grid, classic naval combat

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 *(UPDATED)*

```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": {
    "ship_capture": false,
    "turn_required": true,
    "turn_on_hit": true,
    "turn_on_miss": false
  },
  "messages": {...} // Context-specific flavor text
}
```

Game Rules:

- `ship_capture`: Enable capturing sunk enemy ships
- `turn_required`: Enable turn-based vs rapid-fire mode
- `turn_on_hit`: Continue turn when hitting target (true = get another shot)
- `turn_on_miss`: Continue turn when missing target (false = turn ends)
- `placement_restriction`: Ship placement area limits for large grids

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 *(UPDATED)*

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 *(UPDATED)*
7. **Over → Placement**: Battle again with same settings *(NEW)*

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
- Character-based 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 *(REINFORCED)*

Game state lives outside React's asynchronous system, enabling instant AI responses and preventing race conditions in multiplayer scenarios. Recent fixes ensure all critical game flow (turn progression, game-over detection, AI execution) happens synchronously.

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 *(UPDATED)*

The Game class centralizes hit resolution with alliance rules, damage calculation, and ship capture mechanics. Recent updates include proper alliance-based victory detection and improved Visualizer integration.

Separated Visual State Management *(NEW)*

Visualizer class manages visual effects separate from Board spatial logic, enabling simplified rendering and consistent visual feedback. The UI layer now consumes pre-computed visual data instead of performing complex analysis.

Canvas-Based Battle Visualization *(UPDATED)*

HTML5 Canvas provides smooth animations, real-time feedback, and scalable rendering across device sizes while maintaining 60fps performance. Recent optimizations include simplified rendering pipeline using Visualizer data.

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 *(UPDATED)*

- Minimal React re-renders through strategic state separation
- Canvas rendering optimizations for smooth animations
- Efficient spatial queries using Map-based cell indexing
- Visualizer pre-computation eliminates UI calculation overhead

Mobile Responsiveness

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

Recent Architectural Updates Summary

State Machine Enhancements

- Added `REPLAY` event for seamless rematches
- Fixed page transition consistency across all components
- Eliminated React conditional rendering in favor of state machine dispatch

Game Logic Improvements

- Fixed AI turn progression to prevent double-firing
- Implemented alliance-based victory conditions
- Enhanced synchronous turn management
- Improved UUID consistency throughout player system

Visual System Overhaul

- Introduced Visualizer class for simplified UI rendering
- Implemented color-coded skull system for sunk ships
- Optimized canvas rendering performance
- Separated visual computation from UI presentation

Turn Management Fixes

- Resolved AI synchronization issues
- Clarified character-based vs UX-based delay purposes
- Strengthened synchronous architecture principles

These updates maintain the core architectural philosophy while addressing performance issues, improving user experience, and preparing the foundation for future multiplayer expansion.

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

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.