Game Bible v3.4 - Battle for the Oceans

Comprehensive Architecture Documentation

Document Overview

This Game Bible consolidates all architectural decisions, implementation details, and design patterns for Battle for the Oceans. Version 3.4 incorporates the **debug system architecture** and **enhanced development standards**.

Key Updates in v3.4:

- Debug system (debug.js v1.0) with category-based filtering
- Enhanced critical instructions for version control and logging
- CanvasBoard.js architectural decision (unified battle/placement component)
- React useCallback dependency fixes for proper closure handling

Previous Updates (v3.0-3.3):

- CSS architecture modernization with BEM methodology (v3.3)
- Turn-based gameplay scope confirmed (v3.3)
- Guest player system finalized (v3.2)
- Statistics standardization fully integrated (v3.0)

Table of Contents

Vision & Scope

Core Vision

Battle for the Oceans is a **turn-based strategic naval combat game** that modernizes the classic 1930s Battleship experience into sophisticated multiplayer scenarios across different historical eras.

Confirmed Scope - Rich 30-Minute Sessions:

- **Turn-based gameplay only** - Realtime multiplayer removed from scope
- **Deep strategic experience** in 20-30 minute sessions
- **Quality over quantity** - Focus on polished, engaging tactical gameplay
- **Cross-platform responsive design** (desktop, mobile, tablet)

Multi-era naval combat:

- Traditional Battleship (10x10, classic gameplay) - **Free**
- Midway Island (12x12, WWII Pacific theater) - **Premium**
- Pirates of the Gulf (30x20+, irregular maps, alliance battles) - **Future**

Gameplay Modes:

- **Human vs AI** (primary focus)
- **Alliance-based scenarios** with strategic team formation
- **Intelligent AI opponents** with distinct personalities and strategies
- **Guest play support** for immediate access without registration

Architectural Philosophy

Turn-Based Strategic Focus: The game emphasizes thoughtful decision-making over reaction time. Each move matters, creating tension through careful planning rather than rapid execution.

30-Minute Rich Experience:

- **Placement Phase** (3-5 minutes): Strategic ship positioning
- **Combat Phase** (15-20 minutes): Tactical engagement
- **Resolution** (2-5 minutes): Results, statistics, replay options

---

Core Architecture

Architectural Philosophy

Synchronous Core Principle: The game employs a deterministic, synchronous core as the single source of truth, avoiding race conditions from asynchronous UI operations. State transitions execute business logic before UI updates, ensuring consistency.

Key Design Decisions:

1. **Single Source of Truth**: CoreEngine holds all game state synchronously
2. **Turn-Based Logic**: Action queue manages turn progression and timing
3. **Instant AI Intelligence**: AI decision-making is immediate (no artificial delays)
4. **Presentation Layer Timing**: Visual/audio feedback controlled by action queue
5. **Observer Pattern**: UI subscribes to state changes rather than polling
6. **Service Separation**: Database operations isolated in dedicated service classes
7. **ID-Based User Types**: User type determined by ID prefix pattern (no boolean flags)
8. **CSS Architecture**: Modular, BEM-methodology styling system
9. **Unified Canvas Component**: CanvasBoard handles both placement and battle modes

Object Hierarchy

```
CoreEngine (singleton orchestrator)
├── State Machine (integrated)
│   ├── States & Events
│   └── Transition Logic
├── Game Logic Data
│   ├── Era Configuration
│   ├── Players (Human/AI/Guest)
│   ├── Game Instance
│   └── Board & Visualizer
├── Service Instances
│   ├── UserProfileService
│   ├── GameStatsService
│   ├── LeaderboardService
│   ├── RightsService
│   └── EraService
└── Observer Pattern
    └── UI Subscribers

Game Instance (battle orchestrator)
├── Alliance[] (team management)
│   └── Player[] (human/AI/guest instances)
├── Board (spatial logic, terrain)
├── Visualizer (visual state management)
├── PlayerFleets Map<playerId, Fleet>
│   └── Fleet (ship collection per player)
│       └── Ship[] (individual vessels)
├── Message (communication system)
└── Action Queue (turn-based timing control)
```

Debug System

Architecture (v3.4)

Purpose: Provide filterable, categorized console logging with automatic version tracking and minimal developer overhead.

Design Philosophy:

- **Zero friction**: Works with existing `console.log()` calls
- **Category-based filtering**: Enable/disable specific debug categories
- **Automatic timestamps**: All logs include execution time
- **Runtime control**: Toggle categories without code changes

Implementation

debug.js v1.0:

javascript

```javascript
// src/utils/debug.js
const version = "v1.0";

// Control which categories are logged
const control = {
  PLACEMENT: true,
  SOUND: false,
  AI: true,
  CANVAS: true,
  NETWORK: false,
  STATS: false,
  DEBUG: true
};

// Store original console methods
const originalLog = console.log;
const originalError = console.error;
const originalWarn = console.warn;

// Override console.log to filter by category
console.log = (...args) => {
  const firstArg = args[0];

  // Check if first argument starts with a category tag like [PLACEMENT]
  if (typeof firstArg === 'string') {
    const categoryMatch = firstArg.match(/^\[(\w+)\]/);

    if (categoryMatch) {
      const category = categoryMatch[1];

      // Only log if this category is enabled
      if (control[category] === true) {
        const timestamp = new Date().toLocaleTimeString();
        originalLog(`[${timestamp}] [${category}]`,
          ...args.slice(0, 1).map(arg => arg.replace(/^\[\w+\]\s*/, '')),
          ...args.slice(1));
      }
      return;
    }
  }

  // No category tag, log normally (backwards compatible)
  originalLog(...args);
};
```

```
// Keep error and warn unfiltered (always show)
console.error = (...args) => {
  const timestamp = new Date().toLocaleTimeString();
  originalError(`[${timestamp}] [ERROR]`, ...args);
};

console.warn = (...args) => {
  const timestamp = new Date().toLocaleTimeString();
  originalWarn(`[${timestamp}] [WARN]`, ...args);
};

// Export control for runtime modification
export const debugControl = control;

export const enableCategory = (category) => {
  if (category in control) {
    control[category] = true;
    originalLog(`[DEBUG] Enabled logging for: ${category}`);
  }
};

export const disableCategory = (category) => {
  if (category in control) {
    control[category] = false;
    originalLog(`[DEBUG] Disabled logging for: ${category}`);
  }
};

console.log('[DEBUG] Debug system initialized', version);
// EOF
```

Usage Patterns

In Application Code:

```javascript
// Tagged logs are filtered by category
console.log('[PLACEMENT]', version, 'Mouse down at:', row, col);
console.log('[SOUND]', version, 'Playing audio:', soundFile);
console.log('[AI]', version, 'Strategy selected:', strategyName);
console.log('[CANVAS]', version, 'Drawing cells:', cellCount);

// Untagged logs always show (backwards compatible)
console.log('Game started');
console.error('Critical error:', error);
console.warn('Warning message');
```

Output Example (PLACEMENT enabled, SOUND disabled):

```
[3:45:22 PM] [PLACEMENT] v0.1.11 Mouse down at: 5 3
[3:45:23 PM] [PLACEMENT] v0.1.11 Ship placed successfully
Game started
[3:45:24 PM] [ERROR] Critical error: Network timeout
```

Runtime Control:

```javascript
// In browser console or application code
import { enableCategory, disableCategory } from './utils/debug.js';

disableCategory('PLACEMENT'); // Turn off placement logs
enableCategory('SOUND');      // Turn on sound logs
```

Standard Debug Categories

PLACEMENT - Ship placement interactions

- Mouse/touch event handling
- Preview calculation and validation
- Drag direction detection
- Placement confirmation

CANVAS - Canvas rendering operations

- Draw cycle execution
- Layer rendering (terrain, ships, overlays)
- Animation frame updates
- Performance tracking

SOUND - Audio system operations

- Sound file loading
- Playback events
- Volume control
- Audio context state

AI - AI decision-making

- Strategy selection
- Target calculation
- Move evaluation
- Hunt mode transitions

STATS - Statistics tracking

- Stat updates during gameplay
- Database sync operations
- Leaderboard calculations
- Profile updates

NETWORK - Server communication

- API requests/responses
- WebSocket events
- Data synchronization
- Error recovery

DEBUG - General development logging

- System initialization
- Configuration loading
- State transitions
- Development utilities

Integration

Setup (one-time):

```javascript
// src/index.js
import './utils/debug.js'; // Initialize debug system
import App from './App';
```

Adding New Categories:

```javascript
// In debug.js, add to control object:
const control = {
  PLACEMENT: true,
  SOUND: false,
  AI: true,
  CANVAS: true,
  NETWORK: false,
  STATS: false,
  DEBUG: true,
  ANIMATION: true,  // New category
  DATABASE: false   // New category
};
```

Development Standards

Version Management

File Headers:

```javascript
// src/classes/CoreEngine.js
// Copyright(c) 2025, Clint H. O'Connor

const version = "v0.3.3";
```

Tracking:

- Version in code content
- Version in artifact titles
- Version in console logs
- **Always increment versions when making changes**

Critical Instructions for Development

ALWAYS when modifying or creating code files:

1. **Version Control:**
   - Include version number as `const version = "vX.Y.Z";` for JS files
   - Add version logging: `console.log("[FILENAME]", version);` at initialization
   - For non-JS files use header comment: `/* src/utils/debug.js (v0.1.0) */`
   - End files with `// EOF` or `/* EOF */`
   - **Increment version for ANY changes between chat messages**
2. **Debug Logging:**
   - Use category tags for ALL console.log statements: `console.log('[CATEGORY]', version, ...)`
   - Verify category exists in `src/utils/debug.js` control object
   - Add new categories to debug.js if needed
   - Include version in debug statements for traceability
3. **Available Debug Categories:**
   - `PLACEMENT` - Ship placement interactions
   - `SOUND` - Audio operations
   - `AI` - AI decision-making
   - `CANVAS` - Canvas rendering
   - `NETWORK` - Server communication
   - `STATS` - Statistics tracking
   - `DEBUG` - General development logging
4. **Code Quality:**
   - Ask for existing code before modifying
   - Preserve working functionality
   - Understand interfaces before changing
   - Use BEM CSS methodology
   - No inline styles - all styling in CSS files
   - Check ID prefix for guest/AI detection
   - Add proper dependency arrays to React hooks

CSS Standards

BEM Methodology Required:

```css
/* ✅ Correct */
.btn--primary
.modal__header
.era-item--selected

/* ❌ Incorrect */
.btn-primary
.modalHeader
.era_item_selected
```

No Inline Styles:

```javascript
// ❌ Incorrect
<div style={{padding: '1rem', color: 'blue'}}>

// ✅ Correct
<div className="content-pane text-primary">
```

Responsive Design:

```css
/* ✅ Mobile-first approach */
.btn { font-size: 0.875rem; }
@media (min-width: 768px) {
  .btn { font-size: 1rem; }
}
```

Error Handling

Defensive Programming:

```javascript
// Check for required data
if (!this.eraConfig || !this.humanPlayer || !this.selectedOpponent) {
  throw new Error(`Missing: ${missing.join(', ')}`);
}

// Validate state transitions
const nextState = this.states[this.currentState]?.on[event];
if (!nextState) {
  throw new Error(`No transition for ${this.currentState} with ${event}`);
}
```

React Hook Dependencies

Critical: Proper Dependency Arrays

React's `useCallback` and `useMemo` require accurate dependency arrays to prevent stale closures. When a function is used before it's defined, you'll get a "lexical declaration" error.

Correct Pattern:

```javascript
// 1. Define helper functions first
const isValidShipPlacement = useCallback((cells) => {
  // validation logic
}, [currentShip, eraConfig, gameBoard]);

// 2. Then use them in dependent functions
const handleMouseDown = useCallback((e) => {
  // can safely call isValidShipPlacement
  setIsValidPlacement(isValidShipPlacement(defaultCells));
}, [isValidShipPlacement, /* other deps */]);
```

Common Mistakes:

- Calling functions before they're defined
- Missing dependencies in useCallback arrays
- Stale closures from incomplete dependencies

Code Quality Principles

1. **Single Source of Truth** - Statistics in Player.js, nowhere else
2. **Turn-Based Architecture** - No realtime complexity
3. **No Mapping Functions** - Services use Player stats directly
4. **Simpler is Better** - Avoid overcomplicated solutions
5. **BEM CSS Methodology** - Consistent, maintainable styling
6. **No Inline Styles** - All styling in CSS files
7. **DRY Principle** - Identical code suggests need for class
8. **Clear Interfaces** - Well-defined component contracts
9. **Observer Pattern** - Subscribe to changes, don't poll
10. **ID-Based Types** - Use ID prefix for guest/AI detection
11. **Debug Categories** - Tag all console.log statements
12. **Version Everything** - Track changes across codebase

Critical Instructions for Claude

NEVER:

- Initialize statistics outside Player.js constructor
- Use inline styles (style={{...}})
- Create new files without seeing current version
- Replace working code without understanding it
- Make architectural changes without seeing implementation
- Add realtime complexity to turn-based game
- Use legacy CSS class names in new code
- Forget to add category tags to console.log statements
- Forget to increment version numbers

ALWAYS:

- Ask for existing code before modifying
- Increment version numbers properly
- Use BEM CSS methodology
- Preserve working functionality
- Understand interfaces before changing
- Choose simpler approaches
- Use CSS classes for all styling
- Check ID prefix for guest/AI detection
- Focus on 30-minute rich gameplay sessions
- Add category tags to all console.log statements
- Verify categories exist in debug.js
- Include version in debug statements
- End files with EOF comment
- Add proper React hook dependencies

UI Integration

CanvasBoard Architecture (v3.4)

Purpose: Unified canvas component handling both placement and battle modes

Design Decision: Rather than maintaining separate FleetPlacement and FleetBattle components, CanvasBoard provides a single, mode-aware canvas renderer. This reduces code duplication and simplifies the rendering pipeline.

Mode Switching:

```javascript
// Placement mode
<CanvasBoard
  mode="placement"
  currentShip={selectedShip}
  onShipPlaced={handleShipPlaced}
  humanPlayer={humanPlayer}
/>

// Battle mode
<CanvasBoard
  mode="battle"
  gameState={gameState}
  onShotFired={handleShotFired}
/>
```

Benefits:

- Single canvas rendering engine
- Shared terrain layer caching
- Consistent coordinate system
- Unified mouse/touch handling
- Reduced maintenance burden

Key Implementation Details:

1. **Layer Rendering:**
   - Layer 1: Cached terrain + grid
   - Layer 2: Ships (mode-dependent logic)
   - Layer 3: Overlays (preview/hits)
   - Layer 4: Animations
2. **Mouse Handling:**
   - Placement: drag-and-drop with preview
   - Battle: click to fire shots
3. **React Hook Dependencies:**
   - Helper functions defined before handlers
   - Complete dependency arrays to prevent stale closures
   - Fixed lexical scoping issues

Appendix A: File Inventory

Core Classes

CoreEngine.js v0.3.3

- Singleton orchestrator, state machine, service coordination
- Key Methods: `dispatch()`, `processEventData()`, `transition()`, `handleStateTransition()`, `initializeForPlacement()`, `startGame()`, `handleGameOver()`, `registerShipPlacement()`, `handleAttack()`, `getUIState()`, `subscribe()`

Game.js v0.3.0

- Turn-based battle orchestrator, combat mechanics, turn management
- Key Methods: `addPlayer()`, `receiveAttack()`, `canAttack()`, `calculateDamage()`, `registerShipPlacement()`, `processPlayerAction()`, `handleTurnProgression()`, `checkAndTriggerAITurn()`, `executeAITurnQueued()`, `checkGameEnd()`, `endGame()`, `getGameStats()`

Player.js v0.3.0

- Base player class, statistics initialization
- Key Properties: `hits`, `misses`, `sunk`, `hitsDamage`, `score`
- Key Getters: `shots`, `accuracy`, `averageDamage`, `damagePerShot`

HumanPlayer.js v0.1.0

- Extends Player for human players
- Key Methods: `selectTarget()` (stub)

AiPlayer.js v0.2.3

- Extends Player for AI opponents, implements turn-based strategies
- Key Methods: `makeMove()`, `selectTarget()`, `processAttackResult()`, strategy-specific methods

Board.js v0.2.1

- Spatial logic, terrain system, collision detection
- Key Methods: `registerShipPlacement()`, `getShipDataAt()`, `isValidCoordinate()`, `recordShot()`, `getShipCells()`, `clear()`

Fleet.js v0.1.6

- Ship collection management per player
- Key Methods: `fromEraConfig()` (static factory), `addShip()`, `removeShip()`, `isDefeated()`, `getStats()`

Ship.js v0.1.9

- Individual vessel, health system, overkill damage
- Key Methods: `place()`, `reset()`, `receiveHit()`, `isSunk()`, `getHealth()`

Alliance.js v0.1.2

- Team coordination, friendly fire prevention
- Key Methods: `addPlayer()`, `removePlayer()`, `changeOwner()`, `isPlayerInAlliance()`

Visualizer.js v0.2.2

- Pre-computed visual state for UI rendering
- Key Methods: `updateCellVisuals()`, `updateShipSunk()`, `getCellVisuals()`, `clearAll()`

Message.js v0.1.1

- Game message system, era integration
- Key Methods: `post()`, `get()`, `clear()`, `getEraMessage()`

Services

UserProfileService.js v0.1.0

- Profile CRUD, game name validation
- Key Methods: `getUserProfile()`, `createUserProfile()`, `validateGameName()`, `checkGameNameAvailability()`

GameStatsService.js v0.3.0

- Statistics calculation, database updates
- Key Methods: `updateGameStats()`, `calculateGameResults()`, `recordGameCompletion()`

LeaderboardService.js v0.1.1

- Rankings, champion tracking, guest filtering
- Key Methods: `getLeaderboard()`, `getRecentChampions()`, `getPlayerRanking()`, `getPlayerPercentile()`

RightsService.js v0.1.0

- Era access, voucher redemption
- Key Methods: `hasEraAccess()`, `grantEraAccess()`, `redeemVoucher()`, `getUserRights()`

EraService.js v0.1.2

- Era configuration, message system
- Key Methods: `getAllEras()`, `getEraById()`, `getPromotableEras()`, `getMessage()`, `getGameStateMessage()`, `validateEraConfig()`

StripeService.js v0.1.0

- Price fetching, formatting
- Key Methods: `fetchPrice()`, `formatPrice()`, `clearCache()`, `getCachedPrice()`

UI Components

App.js v0.2.3

- Root component, scene rendering
- Key Components: `SceneRenderer` (state-based page routing)

GameContext.js v0.3.0

- React context provider, CoreEngine wrapper
- Key Exports: `GameProvider`, `useGame()`, `GameContext`

CanvasBoard.js v0.1.11

- Unified canvas component for placement and battle
- Key Methods: `drawCanvas()`, `handleMouseDown()`, `handleMouseMove()`, `handleMouseUp()`, `isValidShipPlacement()`
- Modes: `placement`, `battle`

LoginDialog.js v0.1.35

- Authentication UI, guest login
- Key Methods: `handleLogin()`, `handleSignUp()`, `handleGuest()`, `handleForgotPassword()`

ProfileCreationDialog.js v0.1.1

- Game name creation
- Key Methods: `validateGameName()`, `handleSubmit()`

PromotionalBox.js v0.2.0

- Era promotion after games
- Key Methods: `findPromotionalEra()`, `fetchPriceInfo()`, `handleLearnMore()`

Utilities

debug.js v1.0

- Category-based console logging
- Key Exports: `debugControl`, `enableCategory()`, `disableCategory()`

MessageHelper.js v0.1.0

- Message templating, variable substitution
- Key Methods: `getRandomMessage()`, `replaceVariables()`, `getMessage()`, `getGameMessage()`, `formatCell()`

supabaseClient.js v0.1.6

- Supabase initialization
- Key Exports: `supabase` (client instance)

Conclusion

Game Bible v3.4 documents the **complete implemented architecture** of Battle for the Oceans with emphasis on **developer productivity** through the debug system and **code quality** through enhanced standards.

Key Architectural Achievements:

- **Debug system** with category-based filtering and zero-friction usage
- **CanvasBoard unification** reducing component duplication
- **Enhanced critical instructions** for consistent development practices
- **Turn-based gameplay** confirmed as core focus
- **30-minute rich gameplay sessions** with strategic depth
- **CSS architecture modernized** with BEM methodology
- **Guest player system** with ID prefix pattern
- **Service layer separation** with guest/AI filtering

Development Standards Established:

- **Category-tagged logging** for filterable debug output
- **Version tracking** in all code files
- **EOF markers** for file completeness
- **No inline styles** - All styling in CSS files
- **BEM methodology** - Consistent `.block__element--modifier` naming
- **React hook dependencies** - Proper closure management

This architecture provides a **solid foundation for strategic turn-based naval combat** while maintaining **developer productivity** through clear standards and useful debugging tools.

End of Game Bible v3.4 // EOF