SonarAtlas Architecture & Scope

Interactive Sonar Simulation & WebAssembly Integration Plan

Prepared for Jake • Atlas Universalis

Part 1: Project Scope & Vision

The objective is to build a highly interactive, browser-based technical simulator—a "serious game" presenting real-life scenarios that a sonar operator would encounter on board a ship. The simulation will feature a full suite of controls, gauges, and acoustic outputs, alongside a 3D sandbox with true-to-life models of ships and submarines executing historical encounter missions.

1. The Console: Gauges, Switches, and Output

The console interface does not require a 3D engine. Blazor's C# component model excels at orchestrating exact 2D replicas of console interfaces, dials, switches, and keypads using standard HTML and CSS. For acoustic readouts (broadband and narrowband waterfalls), the C++ engine will process FFT data and stream it to HTML5 `<canvas>` elements, which can seamlessly paint high-fidelity scrolling data at 60 frames per second.

2. The 3D Sandbox and True-to-Life Models

To render actual 3D models of ships, submarines, and the ocean environment, the architecture will utilize WebGL. This can be achieved through a C++ graphics engine compiled via Emscripten, or via JavaScript interop in Blazor using a library like Three.js. This allows for loading industry-standard 3D models (.gltf/.obj) into a 3D Cartesian coordinate system, with vectors, speeds, and depths managed by C# state machines.

3. The Acoustic Physics Engine

C++ WebAssembly integration acts as the core physics engine. It will continuously calculate environmental variables—water temperature gradients, thermal layers, convergence zones, bottom bounce, and the Doppler shift of moving vessels. As vessels move within the 3D sandbox, the C++ engine calculates how specific acoustic signatures (propeller cavitation, plant noise) travel through the simulated water column to the hydrophones.

4. Navigating Realism vs. OPSEC

Because actual modern console layouts and true-to-life acoustic signatures of active vessels are classified, the simulation will focus on historical encounters (e.g., Cold War-era diesel engagements, WWII convoy defenses). This allows for highly technical, historically accurate missions using genericized,

open-source acoustic models that demonstrate the exact physics of sound propagation without exposing sensitive operational data.

Part 2: Technical Migration Plan

> **Strategic Objective:** Transition computational workloads (DSP, audio canvasing, FFTs) from JavaScript workers to near-native WebAssembly speeds utilizing the existing standalone Blazor WebAssembly PWA architecture.

Phase 1: Toolchain Installation

Enable the .NET SDK to compile raw C/C++ files into the WebAssembly bundle by installing the WASM build tools, which provides the underlying Emscripten compiler:

```
dotnet workload install wasm-tools
```

Phase 2: Proof of Concept (The C Bridge)

Establish a functional pipeline with a minimal C function before porting heavy workloads. Create `signal_math.c` in the core project directory:

```
// signal_math.c
int calculate_signal_floor(int base_value) {
    return base_value * 2;
}
```

Phase 3: Wiring into Blazor

Instruct the Blazor project compiler to include the native C file:

```
<ItemGroup>
    <NativeFileReference Include="signal_math.c" />
</ItemGroup>
```

Execute from a Razor Component using `DllImport`:

```
@using System.Runtime.InteropServices

<button @onclick="TestNativeCode">Test C Integration</button>

@code {
    [DllImport("signal_math")]
    static extern int calculate_signal_floor(int base_value);

    private void TestNativeCode()
    {
        int result = calculate_signal_floor(42);
        Console.WriteLine($"Result from native C: {result}");
    }
}
```

Phase 4: DSP & Audio Migration Target

- **Adopt a C/C++ FFT Library:** Integrate a lightweight library like `KissFFT` directly into the WASM bundle.
- **Memory Management:** Pass memory pointers of raw audio buffers from C# directly to the C++ processing loop.
- **Rendering:** Stream the processed spectrum data back to C# to drive UI visualization components efficiently.