Technical Design Document 04: Fluid Simulation Engine

Version: 1.0.0
Role: WebMMO Architect
Context: This module powers the "Toxicity" mechanics in Project Reclamation.

1. Architectural Challenge

We must simulate a 2D fluid grid (100×100 cells per room) for 100+ concurrent players.

- **Constraint:** JavaScript Objects (class Cell { fluid: 10 }) are too heavy for GC.
- **Constraint:** Bandwidth cannot handle sending 10,000 floats at 20Hz.

2. Server-Side Simulation (The Authority)

2.1 Data Structure: TypedArrays & Double Buffering

We use flat Float32Array buffers to store fluid depth. We use **Double Buffering** to prevent directional flow bias during iteration.

```
// server/systems/FluidSystem.ts
export class FluidGrid {
    readonly width: number;
    readonly height: number;

    // Flat arrays: index = y * width + x
    private terrain: Int8Array;      // Static height (-128 to 127)
    private currentFluid: Float32Array; // Current tick state
    private nextFluid: Float32Array;    // Next tick state (Writing target)

    constructor(w: number, h: number) {
        this.width = w; this.height = h;
        const size = w * h;
        this.terrain = new Int8Array(size);
        this.currentFluid = new Float32Array(size);
        this.nextFluid = new Float32Array(size);
    }

    /**
     * Cellular Automata Step (Simplified Hydrodynamics)
     */
    tick(deltaTime: number) {
```

```
        // 1. Reset next buffer to current
        this.nextFluid.set(this.currentFluid);

        // 2. Iterate and spread
        for (let i = 0; i < this.currentFluid.length; i++) {
            const fluid = this.currentFluid[i];
            if (fluid <= 0.01) continue; // Optimization: Skip dry cells

            this.spreadToNeighbors(i, fluid);
        }

        // 3. Swap Buffers
        const temp = this.currentFluid;
        this.currentFluid = this.nextFluid;
        this.nextFluid = temp;
    }
}
```

2.2 Optimization: Bitboard "Dirty Chunks"

To avoid iterating 10,000 cells every tick, we divide the map into 10×10 chunks.

- **Bitmask:** An array of Booleans (or a Bitset).
- **Logic:** If a cell in Chunk A changes, mark Chunk A (and neighbors) as "Dirty".
- **Loop:** Only iterate cells inside Dirty Chunks.

3. Network Synchronization (Delta Compression)

Sending raw floats is bandwidth suicide. We use a **Quantized Binary Patch**.

3.1 The Protocol

1. **Quantization:** Server converts float (0.0 - 10.0 depth) to byte (0 - 255).
2. **Diffing:** Server compares currentByteGrid vs lastSentByteGrid.
3. **Run-Length Encoding (RLE):**
    - Raw: 0, 0, 0, 0, 0, 5, 5, 0, 0
    - RLE: [5, 0], [2, 5], [2, 0] (5 zeros, 2 fives, 2 zeros).
4. **Payload:** room.broadcast("fluidPatch", binaryBuffer);

4. Client-Side Rendering (Phaser 3)

We do **not** use tilemap.putTileAt (CPU heavy). We use a **Dual-Layer Masking Shader**.

4.1 The Visual Stack

1. **Layer Bottom:** The "Clean" world (Grass/Flowers). Always rendered.
2. **Layer Top:** The "Toxic" world (Sludge textures). Rendered on top.
3. **Mask:** A RenderTexture controlled by the Fluid Grid data.

4.2 The "Dissolve" Shader

This shader runs on the GPU. It uses a noise texture to make the transition between Clean and Toxic look organic (like burning paper or drying water), rather than a hard pixel edge.

```
// assets/shaders/dissolve.frag
precision mediump float;

uniform sampler2D uMainSampler; // The Sludge Layer
uniform sampler2D uMaskSampler; // The Fluid Data (Red channel = depth)
uniform sampler2D uNoiseSampler; // Perlin Noise texture

varying vec2 outTexCoord;

void main() {
    vec4 texColor = texture2D(uMainSampler, outTexCoord);
    float fluidDepth = texture2D(uMaskSampler, outTexCoord).r;
    float noise = texture2D(uNoiseSampler, outTexCoord).r;

    // Organic Threshold:
    // As fluidDepth decreases, the noise eats away at the texture.
    float threshold = fluidDepth + (noise * 0.2);

    // Hard cutoff for transparency
    float alpha = step(0.1, threshold);

    gl_FragColor = vec4(texColor.rgb, texColor.a * alpha);
}
```

4.3 Client Interpolation

Since network updates arrive at ~10Hz but render at 60Hz, the client must Linear Interpolate (Lerp) the mask values.
renderTexture.value = Phaser.Math.Linear(current, target, 0.1);