ChatGPT

Advanced Procedural Map Generation in Star Throne

To eliminate crossing warp lanes and territory overlaps, we propose a new `MapGenerator.js` using physics-based and geometric techniques. First, we **generate territory positions** according to the selected layout (Organic, Clusters, Spiral, Core, Rings, Binary). For example, *Organic* starts with jittered random points (or Poisson disk samples) to avoid clumping, while *Clusters* spawns points around several cluster centers, and *Spiral* arranges points along spiral arms. After initial placement, we apply **force-directed relaxation**: each node repels others (like electrically charged particles) and edges (if any) act like springs. This spreads out points evenly and minimizes potential lane crossings [1] [2]. In practice we simulate repulsive forces (e.g. Coulomb-like) among all points and optional attractive forces to adjust spacing to the map size. This yields a clear, well-spaced distribution of stars.

Next, we use **Delaunay triangulation and Voronoi partitioning** to build connections. A Delaunay triangulation on the point set produces a planar graph (no crossing edges) [3]. Its dual is the Voronoi diagram, partitioning space into “territories” around each point [4]. We compute the Delaunay graph (using Delaunator or a similar library) to get all candidate edges. We then compute a **Minimum Spanning Tree (MST)** of that graph (e.g. via Kruskal), ensuring connectivity with minimal total length. Since the Euclidean MST is guaranteed to be a subgraph of the Delaunay graph [3] [5], its edges inherit planarity. We include those MST edges as warp lanes. We may optionally add a few extra Delaunay edges (short non-crossing chords) for redundancy without creating crossings.

To **avoid edges cutting through star circles**, we check each candidate edge against other territories. If a straight line between two stars comes within a star’s radius, we skip that edge. By construction (points well-spaced and edges planar), this is rare; and our collision check ensures no lane intersects any star’s circle. This guarantees warp lanes are clear of other territories.

Finally, we output the map data as a list of territory objects `{id, x, y, connections: [...], owner:null, armies:0, ...}`. The `connections` array lists neighbor IDs (bidirectionally). This format matches the existing game’s `Territory` interface. Hooks accept `mapSize`, `layout`, and `numPlayers` (for example, we might use `numPlayers` to choose cluster counts). The overall spatial bounds (`MapGenerator.mapWidth`, `mapHeight`) are also set so the camera centers correctly.

Below is the full implementation of **MapGenerator.js** following these principles:

```
// MapGenerator.js
// Generates non-overlapping, planar star maps for Star Throne

import Delaunator from 'delaunator';  // Delaunay library (or include its code)

/**
```

```
 * Helper: returns true if segment AB intersects circle centered at C with
radius R.
 */
function lineIntersectsCircle(A, B, C, R) {
    // Project C onto line AB, clamp to segment, check distance
    const vx = B.x - A.x, vy = B.y - A.y;
    const wx = C.x - A.x, wy = C.y - A.y;
    const proj = (wx*vx + wy*vy) / (vx*vx + vy*vy);
    const t = Math.max(0, Math.min(1, proj));
    const px = A.x + t * vx, py = A.y + t * vy;
    const dx = px - C.x, dy = py - C.y;
    return (dx*dx + dy*dy) < (R*R);
}

export default class MapGenerator {
    // Map dimensions (set after generateMap)
    static mapWidth = 0;
    static mapHeight = 0;

    /**
     * Generate a map with given number of territories, layout, and player
count.
     * Returns array of territory objects: {id, x, y, connections: [...],
owner:null, armies:0, ...}.
     */
    static generateMap(mapSize, layout, numPlayers=1) {
        // 1. INITIAL POINT PLACEMENT BASED ON LAYOUT
        let points = [];
        const width = 2000, height =
2000; // base map area (tunable or based on mapSize)
        // Create initial points differently per layout
        if (layout === 'clusters') {
            // e.g. 4 clusters by default (or numPlayers clusters)
            const numClusters = Math.max(2, Math.min(numPlayers, 8));
            let centers = [];
            for (let i = 0; i < numClusters; i++) {
                centers.push({
                    x: Math.random() * width * 0.8 + width*0.1,
                    y: Math.random() * height * 0.8 + height*0.1
                });
            }
            for (let i = 0; i < mapSize; i++) {
                const c = centers[i % centers.length];
                // Gaussian cluster around center
                const theta = Math.random()*2*Math.PI;
                const r = Math.sqrt(Math.random()) * 300;
                points.push({ x: c.x + r*Math.cos(theta), y: c.y +
r*Math.sin(theta) });
```

```
            }
        }
        else if (layout === 'spiral') {
            // Place points along a spiral
            const arms = 3;
            for (let i = 0; i < mapSize; i++) {
                const angle = i * (2 * Math.PI / (mapSize / arms));
                const r = 50 +  (i / mapSize) * Math.min(width, height) / 2;
                // add some noise
                const x = width/2 + r*Math.cos(angle) + (Math.random()*50 - 25);
                const y = height/2 + r*Math.sin(angle) + (Math.random()*50 -
25);
                points.push({ x, y });
            }
        }
        else if (layout === 'core') {
            // Dense core: points near center
            for (let i = 0; i < mapSize; i++) {
                const r = Math.sqrt(Math.random()) * (Math.min(width, height)/
3);
                const theta = Math.random() * 2 * Math.PI;
                points.push({
                    x: width/2 + r * Math.cos(theta),
                    y: height/2 + r * Math.sin(theta)
                });
            }
        }
        else if (layout === 'rings') {
            // Concentric rings
            const rings = Math.ceil(mapSize / 20);
            for (let i = 0; i < mapSize; i++) {
                const ring = i % rings;
                const r = (50 + 100 * ring) + Math.random()*20;
                const theta = Math.random() * 2 * Math.PI;
                points.push({
                    x: width/2 + r * Math.cos(theta),
                    y: height/2 + r * Math.sin(theta)
                });
            }
        }
        else if (layout === 'binary') {
            // Two main clusters far apart
            const offset = 400;
            const centers = [
                { x: width/2 - offset, y: height/2 },
                { x: width/2 + offset, y: height/2 }
            ];
            for (let i = 0; i < mapSize; i++) {
                const c = centers[i % 2];
                const theta = Math.random()*2*Math.PI;
                const r = Math.sqrt(Math.random()) * 200;
                points.push({ x: c.x + r*Math.cos(theta), y: c.y +
r*Math.sin(theta) });
            }
        }
        else { // 'organic' or default
            // Jittered random (or Poisson)
            for (let i = 0; i < mapSize; i++) {
                points.push({
                    x: Math.random() * width,
                    y: Math.random() * height
                });
            }
        }

        // 2. RELAXATION / FORCE-DIRECTED REPULSION
        // Use simple repulsion loop to improve spacing (Lloyd or Coulomb)
        for (let iter = 0; iter < 5; iter++) {
            const forces = points.map(p => ({x:0, y:0}));
            for (let i = 0; i < mapSize; i++) {
                for (let j = i+1; j < mapSize; j++) {
                    const A = points[i], B = points[j];
                    const dx = B.x - A.x, dy = B.y - A.y;
                    const dist2 = dx*dx + dy*dy + 0.01;
                    const dist = Math.sqrt(dist2);
                    // Repulsive force magnitude (inverse-square)
                    const rep = 1000 / dist2;
                    // Unit direction
                    const ux = dx / dist, uy = dy / dist;
                    // Apply forces
                    forces[i].x -= rep * ux;
                    forces[i].y -= rep * uy;
                    forces[j].x += rep * ux;
                    forces[j].y += rep * uy;
                }
            }
            // Update positions with small damping factor, keep within bounds
            for (let i = 0; i < mapSize; i++) {
                points[i].x += Math.max(-10, Math.min(10, forces[i].x));
                points[i].y += Math.max(-10, Math.min(10, forces[i].y));
                points[i].x = Math.max(0, Math.min(width, points[i].x));
                points[i].y = Math.max(0, Math.min(height, points[i].y));
            }
        }

        // 3. COMPUTE DELAUNAY TRIANGULATION
        const coords = [];
        for (let p of points) { coords.push(p.x, p.y); }
        const delaunay = Delaunator.from(points, p => p.x, p => p.y);
        const edges = new Set();
        // Collect unique edges from Delaunay
        for (let e = 0; e < delaunay.triangles.length; e += 3) {
            const tri = [
                delaunay.triangles[e],
                delaunay.triangles[e+1],
                delaunay.triangles[e+2]
            ];
            // For each triangle, add its 3 edges
            [[0,1],[1,2],[2,0]].forEach(([i,j]) => {
                const a = Math.min(tri[i], tri[j]);
                const b = Math.max(tri[i], tri[j]);
                edges.add(`${a},${b}`);
            });
        }
        // Convert edge set to array of [i,j]
        let edgeList = Array.from(edges).map(str => {
            const [a,b] = str.split(',').map(Number);
            return {u:a, v:b, length: Math.hypot(points[a].x - points[b].x,
points[a].y - points[b].y)};
        });

        // 4. MINIMUM SPANNING TREE (Kruskal)
        edgeList.sort((e1,e2) => e1.length - e2.length);
        const parent = Array(mapSize).fill(0).map((_,i)=>i);
        function find(i){ return parent[i]===i ? i :
(parent[i]=find(parent[i])); }
        function unite(a,b){ parent[find(a)] = find(b); }
        const connections = Array.from({length: mapSize}, () => []);
        let edgesUsed = 0;
        for (let edge of edgeList) {
            if (edgesUsed >= mapSize-1) break;
            const ru = find(edge.u), rv = find(edge.v);
            if (ru !== rv) {
                unite(ru, rv);
                // Before adding, check it does not cut through any circle
                let valid = true;
                const A = points[edge.u], B = points[edge.v];
                // Assume minimal radius (e.g. 20) to test intersection
                const R = 20;
                for (let k = 0; k < mapSize; k++) {
                    if (k===edge.u || k===edge.v) continue;
                    const C = {x: points[k].x, y: points[k].y};
                    if (lineIntersectsCircle(A, B, C, R)) { valid = false;
break; }
```

```
                }
                if (!valid) continue;
                // Add to connections (bidirectional)
                connections[edge.u].push(edge.v);
                connections[edge.v].push(edge.u);
                edgesUsed++;
            }
        }

        // 5. OPTIONAL: Add some extra non-crossing Delaunay edges for more
connectivity
        // (skip for brevity or implement as needed)

        // 6. BUILD TERRITORY OBJECTS
        const territories = [];
        for (let i = 0; i < mapSize; i++) {
            territories.push({
                id: i,
                x: points[i].x,
                y: points[i].y,
                connections: connections[i],
                owner: null,
                armies: 0,
                isColonizable: true,
                isThroneStar: false,
                isFlashing: false
            });
        }

        // Compute bounding box for map dimensions
        const xs = points.map(p=>p.x), ys = points.map(p=>p.y);
        const minX = Math.min(...xs), maxX = Math.max(...xs);
        const minY = Math.min(...ys), maxY = Math.max(...ys);
        MapGenerator.mapWidth = maxX - minX + 200;  // add margin
        MapGenerator.mapHeight = maxY - minY + 200;

        return territories;
    }
}
```

This implementation uses **force-directed relaxation** for spacing [1], **Voronoi/Delaunay methods** for planar connectivity [4] [3], and an MST to finalize warp lanes. It returns an array of territories with `connections` listing neighbor IDs (bidirectional edges). The static `mapWidth`/`mapHeight` fields record the map's size for camera centering.

Integration into StarThrone.js

We replace the old `GameMap.generateMap()` call with our `MapGenerator`. For example, in `StarThrone.js`:

```
 import GameMap from './GameMap.js';
+ import MapGenerator from './MapGenerator.js';

 ...

 async initSinglePlayer() {
     ...
-    this.territories = this.gameMap.generateMap(mapSize, layout);
+    // Use new MapGenerator instead of GameMap
+    this.territories = MapGenerator.generateMap(mapSize, layout,
this.config.aiCount + 1);
+    // Update map dimensions for camera
+    this.gameMap.width = MapGenerator.mapWidth;
+    this.gameMap.height = MapGenerator.mapHeight;
     ...
 }
```

We add an import of `MapGenerator` and call `MapGenerator.generateMap(mapSize, layout, numPlayers)`. After generation, we set `gameMap.width/height` from `MapGenerator.mapWidth` so the camera centers correctly. The returned territory array is assigned to `this.territories` just as before. (Optionally, we may deprecate or remove `GameMap.generateMap` entirely.)

Integration and Testing Plan

1. **Add MapGenerator Module**: Place `MapGenerator.js` in the game code directory. Ensure Delaunator (or equivalent) is available (e.g. via npm or script tag).
2. **Import and Replace**: In `StarThrone.js`, import `MapGenerator` and replace all calls to `GameMap.generateMap()` with `MapGenerator.generateMap(...)`, passing `mapSize`, `layout`, and the player count. Update `this.gameMap.width/height` to match `MapGenerator.mapWidth/height`.
3. **Maintain Data Schema**: The new territories have the same structure (id, x, y, connections, etc.), so AI, camera, and supply systems should work unchanged. Verify properties like `owner`, `armies`, `isColonizable` are initialized appropriately (e.g. all `null` owner at start).
4. **Test Layouts Individually**: Run the game in all six layouts (Organic, Clusters, Spiral, Core, Rings, Binary). Ensure stars appear in the intended pattern and are well-spaced.
5. **Verify Warp Lanes**: Check on-screen that warp lanes do not cross or cut through stars. Manually inspect (or write a test) to confirm no edge intersects another or any territory circle.
6. **Edge Cases and Stress Tests**: Test with various `mapSize` and `numPlayers` values (min, max). Ensure performance is acceptable (Delaunay + MST are $O(n \log n)$ [3]).

7. **Gameplay Tests**: Play through to confirm AI behavior, supply route, and camera work normally (they use the same territory graph). Confirm no regressions in colonization or combat due to map changes.
8. **Automated Checks**: Optionally write automated validation (e.g. assert graph connectivity, no crossings using geometry checks).
9. **Review and Iterate**: Adjust relaxation iterations or clustering parameters if maps feel too sparse or dense. Ensure visual clarity and fairness are improved over the old algorithm.

This new modular approach should eliminate overlapping warp lanes and star circle intersections, improving clarity without altering gameplay logic (territory and connection schemas remain the same). By leveraging **force-directed layout and Voronoi/Delaunay techniques**, we ensure visually appealing, fair maps [1] [4] [3].

Sources: Force-directed and Voronoi methods in procedural map layout [1] [4]; using Delaunay triangulation for planar MST graphs [3] [5].

---

[1] Force-directed graph drawing - Wikipedia
https://en.wikipedia.org/wiki/Force-directed_graph_drawing

[2] [4] Voronoi maps tutorial
https://www.redblobgames.com/x/2022-voronoi-maps-tutorial/

[3] [5] Euclidean minimum spanning tree - Wikipedia
https://en.wikipedia.org/wiki/Euclidean_minimum_spanning_tree