Detailed Design Report:
Satellite Coverage Visualiser

Version: 1.0

Date: 16 October 2025

1. Introduction

1.1. Purpose

This document provides a detailed technical design of the single-page HTML Satellite Coverage Visualiser application within the file `SatelliteCoverage.html`, which includes HTML for structure, CSS for styling, and JavaScript for all logic, including an embedded Web Worker. It deconstructs the system architecture, data models, component interactions, and core algorithms.

1.2. System Overview

The Satellite Coverage Visualiser is a real-time simulation tool that models the orbits of up to three user-defined satellites. It calculates their orbital paths, including J_2 perturbations, and determines their ground coverage based on a configurable Field of View (FOV).

The system visualises this data in real-time through:

1. A 2D equirectangular map (using D3.js and TopoJSON) displaying coastlines, satellite markers, and ground tracks.
2. A dynamic heatmap (using HTML Canvas) showing the number of satellites (0, 1, 2, or 3) covering any point on Earth.
3. A statistical panel (using a Web Worker) that calculates the total percentage of the Earth's surface covered by 0, 1, 2, or 3 satellites.

The application is fully interactive, allowing users to modify all orbital parameters and simulation settings, with the visualisation updating instantly.

2. System Architecture

2.1. High-Level Architecture

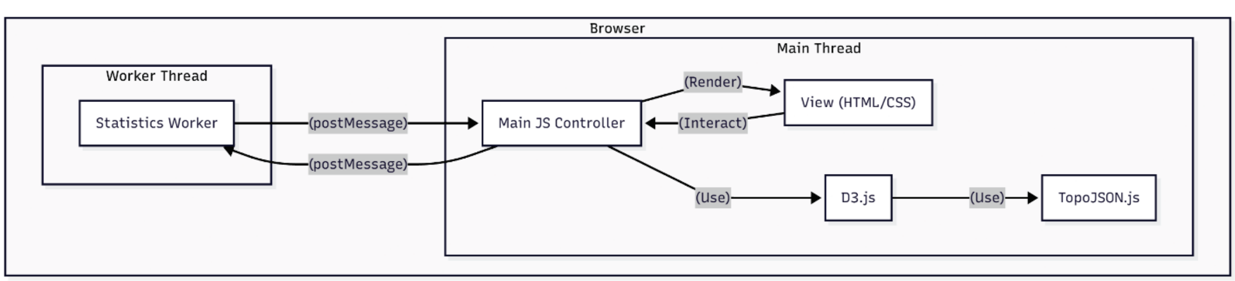
The application operates on a 3-component architecture, separating concerns between the main user interface (UI) thread and a background computation thread.

1. **View (DOM/CSS):** The HTML structure and CSS styling. It provides all user controls (sliders, toggles, buttons) and display areas (map, heatmap, text readouts).
2. **Main Controller (Main JS Thread):** Runs the primary `simulationLoop` using `requestAnimationFrame`. This thread is responsible for:
   - Handling all user input.
   - Propagating satellite orbits (physics calculations).
   - Rendering the SVG map (satellite markers, ground tracks).
   - Calculating and rendering the coverage heatmap.
   - Coordinating with the Statistics Worker.
3. **Statistics Service (Web Worker):** An independent, background thread. Its sole purpose is to perform the computationally expensive Monte Carlo simulation for global coverage statistics. This prevents the main UI from freezing.

2.2. UML Component Diagram

This diagram shows the main components and their dependencies.

```
flowchart TB
  subgraph subGraph0["Main Thread"]
        id1["Main JS Controller"]
        id2["View (HTML/CSS)"]
        id3["D3.js"]
        id4["TopoJSON.js"]
  end
  subgraph subGraph1["Worker Thread"]
        id5["Statistics Worker"]
  end
  subgraph Browser["Browser"]
        subGraph0
        subGraph1
  end
    id2 -- (Interact) --> id1
    id1 -- (Render) --> id2
    id1 -- (Use) --> id3
    id3 -- (Use) --> id4
    id1 -- (postMessage) --> id5
    id5 -- (postMessage) --> id1
```



3. Data Model and State

3.1. Global State Variables

The main `JavaScript` controller maintains several key global variables to manage the simulation state.

- `svg`, `projection`, `path`, `graticule`: D3.js objects that control the SVG map rendering.
- `heatmapCtx`, `heatmapWidth`, `heatmapHeight`: HTML Canvas context and dimensions for the heatmap.
- `lastTimestamp`: `DOMHighResTimeStamp` used to calculate the delta time between `simulationLoop` frames.
- `epochDate`: A JavaScript `Date` object representing the simulation's "`T=0`". Set by the "Epoch (UTC)" input.
- `currentSimDate`: The advancing simulation time, calculated as `epochDate` + `elapsedSimSeconds`.
- `epochGST`: The Greenwich Sidereal Time (in radians) calculated once for the `epochDate`.
- `elapsedSimSeconds`: A number representing the total simulation seconds passed since `epochDate`.
- `satelliteStates`: An array `[{}, {}, {}]` storing the most recent propagated state for each satellite.
- `groundTracks`: An array `[ [], [], [] ]` where each sub-array stores points [`lon`, `lat`, `timestamp`] for the satellite's ground track.
- `lastStatsUpdateTime`: A timestamp used to throttle messages to the Stats Worker.
- `statsWorker`: The `Worker` object instance.
- `pixelToGeoCache`: A performance-critical array. It stores a pre-calculated mapping of every heatmap canvas pixel to its corresponding ECEF coordinate.
    - **Structure:** `[ null, {ecef: {x,y,z}}, {ecef: {x,y,z}}, null, ... ]`
    - `null` is stored for pixels that are off the map projection.

3.2. Key Data Structures

3.2.1. Satellite State Object

This object is the primary output of `propagateSatellite()` and is stored in the `satelliteStates` array.

Property	Type	Description
`eci`	`{x, y, z}`	ECI (Inertial) position vector (km).
`ecef`	`{x, y, z}`	ECEF (Earth-Fixed) position vector (km).
`latLon`	`[lon, lat]`	Geographic coordinates (degrees).
`period`	`Number`	Orbital period (seconds).
`nu_rad`	`Number`	True Anomaly (radians).
`alt_km`	`Number`	Altitude above Earth's surface (km).
`current_raan_rad`	`Number`	Propagated RAAN (radians).
`current_aop_rad`	`Number`	Propagated Argument of Perigee (radians).
`raan_dot_deg_day`	`Number`	RAAN J_2 perturbation rate (deg/day).
`aop_dot_deg_day`	`Number`	AoP J_2 perturbation rate (deg/day).

3.2.2. Satellite Coverage Parameters Object

This object is created in the `simulationLoop` and passed to both `updateHeatmap()` and the `statsWorker`. It contains pre-calculated values needed for coverage checks.

Property	Type	Description
`satPos`	`{x, y, z}`	The satellite's ECEF position (same as `state.ecef`).
`r_sat`	`Number`	The satellite's radius from Earth's centre (km).
`fov_rad`	`Number`	The sensor's FOV half-angle (radians).
`nadir`	`{x, y, z}`	The nadir vector (points from sat to Earth's centre).
`nadirMag`	`Number`	Magnitude of the nadir vector (same as `r_sat`).
`horizon_angle_rad`	`Number`	The max angle from nadir to the Earth's limb (radians).

4. Interfaces

4.1. User Interface (UI)

The UI is divided into two main sections:

1. **Control Panel (`#panel-container`):**
   - **Tabs:** Allows switching between controls for `Sat 1`, `Sat 2`, `Sat 3`, and `Sim`.
   - **Satellite Tabs (`#sat-1`, ...):**
     - **Orbital Sliders:** 6 sliders for Keplerian elements (`a, e, i, raan_0, aop_0, m0`).
     - **Coverage Sliders:** 1 slider for `FOV`.
     - **Toggles:** Checkboxes for `Show Ground Track` and `Show Coverage Area`.
     - **State Display:** Text readouts for real-time `True Anomaly, Altitude, Current RAAN`, etc.
   - **Sim Tab (`#sim`):**
     - **Epoch Control:** A `datetime-local` input to set the simulation start time.
     - **Speed Control:** A slider to control the simulation speed multiplier.
     - **Reset Button:** Resets the simulation to the epoch time.
   - **Statistics Panel (`#coverage-stats`):**
     - Displays the percentage of Earth's surface covered by 0, 1, 2, or 3 satellites.
2. **Map View (`#map-container`):**
   - **Heatmap (`#heatmap-canvas`):** A canvas layer displaying coverage density.
   - **Map (`#map-svg`):** An SVG layer displaying coastlines, graticules, satellite markers, and ground tracks.

4.2. Programmatic Interfaces (Worker API)

4.2.1. Main Thread -> Stats Worker

- **Call:** `statsWorker.postMessage(satCoverageParams)`
- **Data:** `satCoverageParams` - An *array* of **Satellite Coverage Parameter Objects** (see 3.2.2). This array contains one object for each satellite that has "Show Coverage Area" toggled on.

4.2.2. Stats Worker -> Main Thread

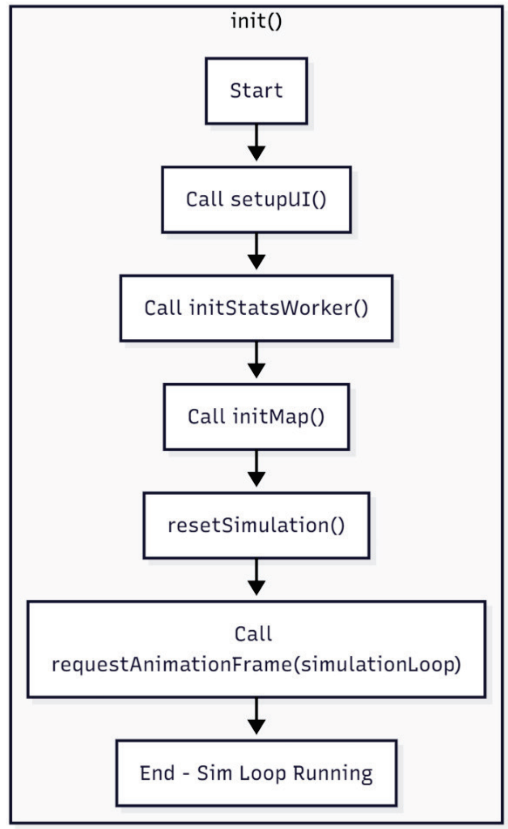
- **Event:** `statsWorker.onmessage`
- **Data:** `{ counts, numSamples }`
  - `counts`: An array `[count_0, count_1, count_2, count_3]` where `count_N` is the number of random samples that were covered by `N` satellites.
  - `numSamples`: The total number of random samples taken (e.g., 50000).

5. Detailed Component Design

5.1. Initialization (`init` function)

The `init()` function orchestrates the application start-up.

```
flowchart TB
 subgraph subGraph0["init()"]
        id1["Start"]
        id2["Call setupUI()"]
        id3["Call initStatsWorker()"]
        id4["Call initMap()"]
        id5["resetSimulation()"]
        id6["Call requestAnimationFrame(simulationLoop)"]
        id7["End - Sim Loop Running"]
  end
    id1 --> id2
    id2 --> id3
    id3 --> id4
    id4 --> id5
    id5 --> id6
    id6 --> id7
```

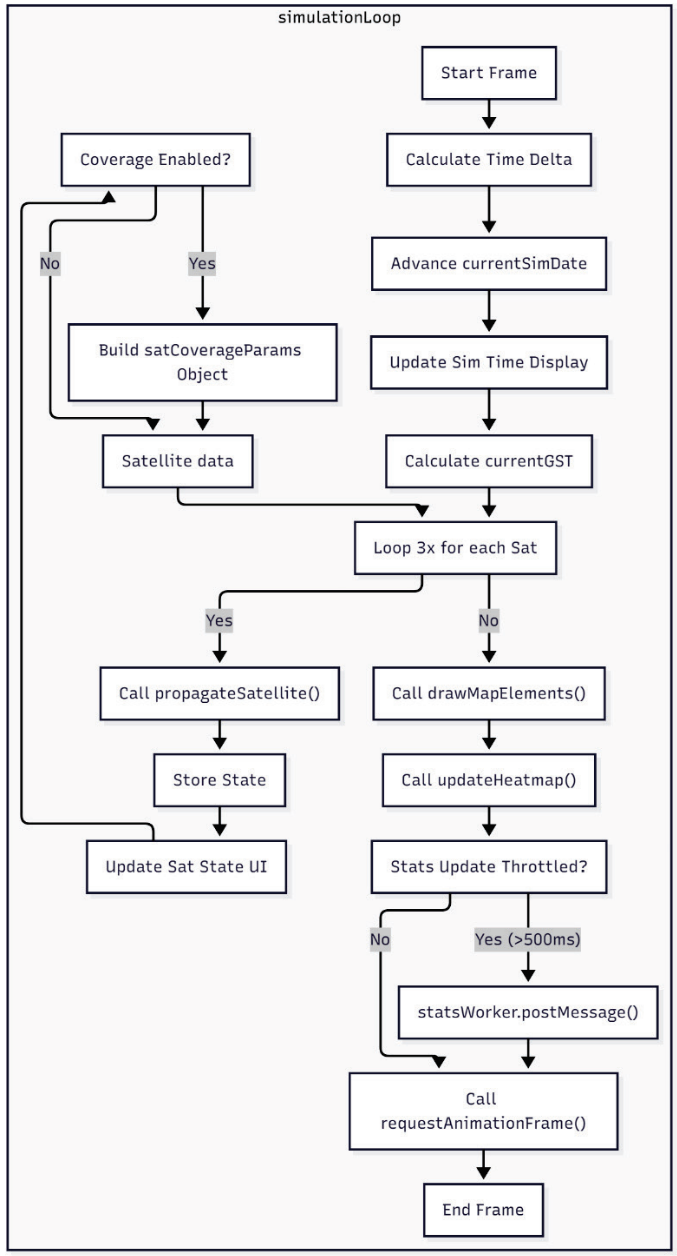


- `setupUI()`: Attaches all event listeners to sliders, buttons, and toggles.
- `initStatsWorker()`: Creates the worker from the inline script and sets up the `onmessage` listener to receive statistics.
- `initMap()`: Sets up the D3 projection, SVG layers, and loads the TopoJSON world map.
- `resetSimulation()`: Sets the initial `epochDate` and calculates `epochGST`.
- `requestAnimationFrame()`: Kicks off the main `simulationLoop`.

5.2. Main Simulation Loop (`simulationLoop`)

This is the application. It's responsible for advancing the simulation and updating all visuals every frame.

```
flowchart TB
 subgraph subGraph0["simulationLoop"]
        ida["Start Frame"]
        idb["Calculate Time Delta"]
        idc["Advance currentSimDate"]
        idd["Update Sim Time Display"]
        ide["Calculate currentGST"]
        idf["Loop 3x for each Sat"]
        idg["Call propagateSatellite()"]
        idh["Store State"]
        idi["Update Sat State UI"]
        idj["Coverage Enabled?"]
        idk["Build satCoverageParams
Object"]
        idl["Satellite data"]
        idm["Call drawMapElements()"]
        idn["Call updateHeatmap()"]
        ido["Stats Update Throttled?"]
        idp["statsWorker.postMessage()"]
        idq["Call requestAnimationFrame()"]
        idr["End Frame"]
  end
    ida --> idb
    idb --> idc
    idc --> idd
    idd --> ide
    ide --> idf
    idf -- Yes --> idg
    idg --> idh
    idh --> idi
    idi --> idj
    idj -- Yes --> idk
    idk --> idl
    idj -- No --> idl
    idl --> idf
    idf -- No --> idm
    idm --> idn
    idn --> ido
    ido -- Yes (>500ms) --> idp
    idp --> idq
    ido -- No --> idq
    idq --> idr
```



5.3. Orbital Mechanics Engine

This engine is a set of pure functions that handle all physics calculations.

5.3.1. `propagateSatellite(index, elapsedSimSeconds, currentGST)`

This is the core function for determining a satellite's position.

1. **Get Parameters:** Reads `a, e, i, raan_0, aop_0, m0` from the `dom.sats[index]` controls.
2. **Validate Input:** Checks if perigee `(a * (1 - e))` is inside the Earth. If so, displays a warning and returns `{}`.
3. **Calculate J_2 Perturbation Rates:**
   - Mean Motion: $n = \sqrt{\mu/a^3}$
   - Semi-latus Rectum: $p = a(1 - e^2)$
   - J2 Factor: $J_2^{factor} = -(3/2)\, n\, J_2 (R_E/p)^2$
   - RAAN Rate (Nodal Precession): $\dot{\Omega} = J_2^{factor} \cos i$
   - AoP Rate (Apsidal Precession): $\dot{\omega} = J_2^{factor}\ (5/2\ (\sin i)^2 - 2)$
4. **Propagate Elements:**
   - Current Mean Anomaly: $M = (M_0 + n\, t)$
   - Current RAAN: $\Omega = \left(\Omega_0 + \dot{\Omega}\, t\right)$
   - Current AoP: $\omega = (\omega_0 + \dot{\omega}\, t)$
   - where t= `elapsedSimSeconds`.
5. **Solve Kepler's Equation:** Calls `solveKepler(M, e)` to get the Eccentric Anomaly, E.
6. **Find Perifocal Coordinates:**
   - $x_{orb} = a\,(\cos(E) - e)$
   - $y_{orb} = a\sqrt{1 - e^2}\, sin(E)$
7. **Rotate to ECI Frame:** Applies a 3D rotation matrix (a $R_z(-\Omega) \cdot R_x(-i) \cdot R_z(-\omega)$ transformation).
   - `Rxx = cos(Ω)cos(ω) - sin(Ω)sin(ω)cos(i)`
   - `Rxy = -cos(Ω)sin(ω) - sin(Ω)cos(ω)cos(i)`
   - `Ryx = sin(Ω)cos(ω) + cos(Ω)sin(ω)cos(i)`
   - `Ryy = -sin(Ω)sin(ω) + cos(Ω)cos(ω)cos(i)`
   - `Rzx = sin(ω)sin(i)`
   - `Rzy = cos(ω)sin(i)`
   - $x_{eci} = R_{xx}\ x_{orb} + R_{xy}\, y_{orb}$; $y_{eci} = R_{yx}\ x_{orb} + R_{yy}\, y_{orb}$; $z_{eci} = R_{zx}\ x_{orb} + R_{zy}\, y_{orb}$
8. **Rotate to ECEF Frame:** Applies a 2D rotation around the Z-axis based on Earth's rotation (`currentGST`).
   - $x_{ecef} = x_{eci} \cos GST + y_{eci} \sin GST$
   - $y_{ecef} = -x_{eci} \sin GST + y_{eci} \cos GST$
   - $z_{ecef} = z_{eci}$
9. **Convert to Lat/Lon:** Calls `ecefToLatLon(ecef)`.
10. **Return State Object:** (See section 3.2.1).

5.3.2. `solveKepler(M, e)`

- **Purpose:** Solves Kepler's Equation $M = E - e \sin E$ for E.
- **Algorithm:** Newton-Raphson iteration.
- **Logic:**
    1. Start with guess $E_0 = M$.
    2. Define $f(E) = E - e \sin(E) - M$.
    3. Define $f'(E) = E - e \cos(E)$.
    4. Iterate: $E_{n+1} = E_n - (f(E_n)/f'(E_n))$
    5. Stop when $\Delta E < 10^{-6}$ or after 100 iterations.
    6. Return E.

5.3.3. `getGST(date)`

- **Purpose:** Calculates Greenwich Sidereal Time for a given `Date`.
- **Logic:**
    1. Convert `date` to Julian Date (JD).
    2. Calculate centuries since J2000: $T_{ut1} = (JD - 2451545.0) / 36525.0$.
    3. Calculate GMST (degrees) using the standard IAU formula:
    $$GMST = 280.46... + 360.98...(JD - 2451545.0) + ...$$
    4. Normalize $GMST$ to $[0, 360)$ range.
    5. Convert to radians and return.

5.4. Map and Visualization Engine

5.4.1. `drawMapElements()`

- **Purpose:** Updates the SVG markers and ground tracks.
- **Logic:**
    1. Loops `forEach` satellite in `satelliteStates`.
    2. **Marker:**
        - Gets `[lon, lat]` from the state.
        - Projects to screen coordinates: `coords = projection([lon, lat])`.
        - Selects `svg.select("#sat-marker-i")`.
        - Sets `.attr("cx", coords[0])` and `.attr("cy", coords[1])`.
    3. **Ground Track:**
        - If `track` is checked:
        - Filters `groundTracks[i]` to remove points older than 2 orbits.
        - Pushes the new `[lon, lat, timestamp]` to `groundTracks[i]`.
        - Creates a GeoJSON `LineString` object from the filtered track points.
        - Selects `svg.select("#track-i")`.
        - Binds the GeoJSON: `.datum(geoJsonLine)`.
        - Redraws the path: `.attr("d", path)`.

5.4.2. `updateHeatmap(satParams)`

- **Purpose:** Renders the coverage heatmap to the canvas.
- **Logic:**
  1. Gets a new `ImageData` buffer from `heatmapCtx`.
  2. Loops `for (let i = 0; i < pixelToGeoCache.length; i++)`.
  3. `geo = pixelToGeoCache[i]`.
  4. If geo is `null`, set `coverageCount = 0`.
  5. If geo is valid:
     - `pointECEF = geo.ecef.`
     - `coverageCount = 0`.
     - Loop `forEach (params in satParams)`.
     - If `isPointVisible(pointECEF, params...): coverageCount++`.
  6. `color = heatmapColors[coverageCount]`.
  7. Set RGBA values in the buffer:
     - `data[i*4 + 0] = color[0]` (R)
     - `data[i*4 + 1] = color[1]` (G)
     - `data[i*4 + 2] = color[2]` (B)
     - `data[i*4 + 3] = color[3]` (A)
  8. After loop, calls `heatmapCtx.putImageData(imgData, 0, 0)`.

5.4.3. `isPointVisible(pointECEF, ...params)`

- **Purpose:** Checks if a single ECEF point is visible to a single satellite.
- **Logic:**
  1. **Horizon Check:**
     - Calculates the angle θ_{centre} between the satellite's position vector and the point's position vector.
     - $\boldsymbol{V}_{sat} = satPos$
     - $\boldsymbol{V}_{point} = pointECEF$
     - $\cos(\theta_{centre}) = \boldsymbol{V}_{sat} \cdot \boldsymbol{V}_{point} / |\boldsymbol{V}_{sat}||\boldsymbol{V}_{point}|$
     - Calculates the pre-computed `horizon_angle_rad`.
     - If $\boldsymbol{V}_{sat} > horizon\_angle\_rad$, return `false` (point is behind Earth).
  2. **FOV Check:**
     - Calculates the angle θ_{foc} between the satellite's nadir vector and the vector from the satellite to the point.
     - $\boldsymbol{V}_{nadir} = nadir$
     - $\boldsymbol{V}_{satToPoint} = pointECEF - satPos$
     - $\cos(\theta_{fov}) = \boldsymbol{V}_{nadir} \cdot \boldsymbol{V}_{satToPoint} / |\boldsymbol{V}_{nadir}||\boldsymbol{V}_{satToPoint}|$
     - If $\theta_{fov} > fov\_rad$ (the FOV *half-angle*), `return false` (point is outside the sensor's view).
  3. If both checks pass, `return true`.

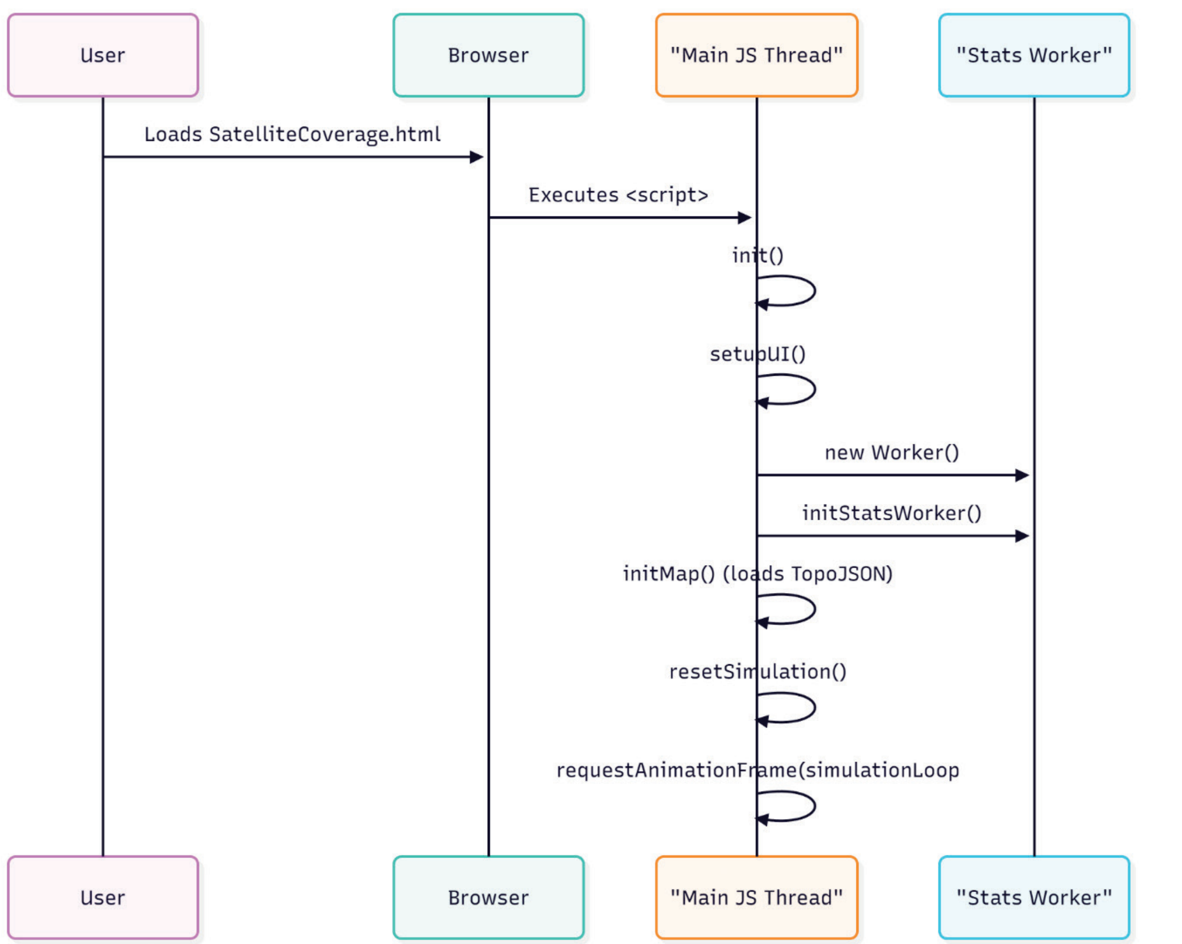
5.5. Statistics Worker (`stats-worker`)

- **Purpose:** Calculates global coverage statistics without blocking the UI.
- **`onmessage` Logic:**
  1. Receives `satParams` array from the main thread.
  2. Initializes `counts = [0, 0, 0, 0]`.
  3. **Monte Carlo Loop:** Runs `for (let i = 0; i < 50000; i++)`.
     - **Generate Random Point:**
       - $u = \text{Math.random}(), v = \text{Math.random}()$
       - $\text{lon} = 360u - 180$
       - $\text{lat} = \cos^{-1}(2v - 1)\,(180/\pi) - 90$ (Ensures uniform spherical distribution)
     - `pointECEF = latLonToECEF(lat, lon, EARTH_RADIUS_KM)`.
     - `coverageCount = 0`.
     - Loop `forEach (params in satParams)`.
     - If `isPointVisible(pointECEF, params...): coverageCount++`.
     - `counts[coverageCount]++`.
  4. After loop, `postMessage({ counts, numSamples: 50000 })` back to the main thread.

6. Key Sequence Diagrams

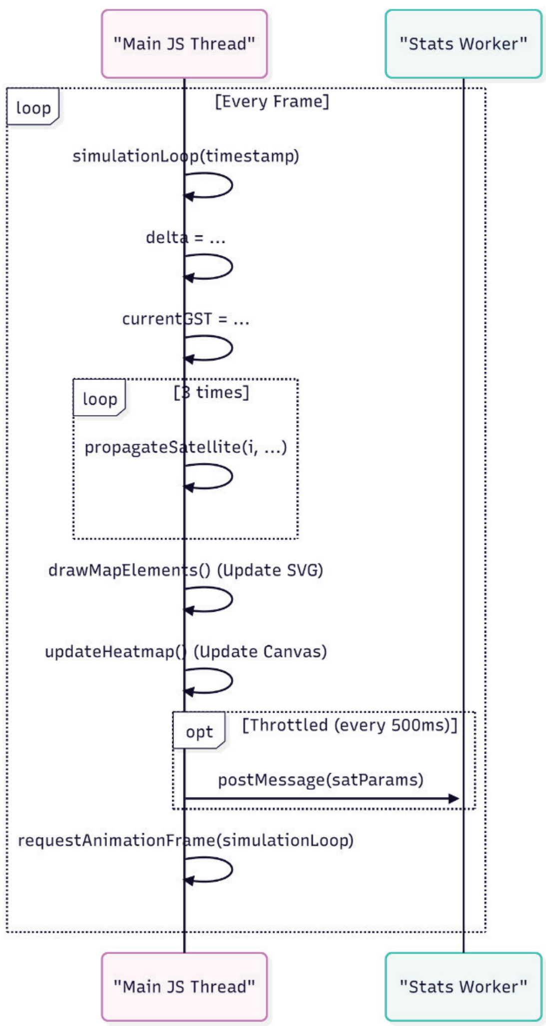
6.1. Application Load and Initialisation

```
sequenceDiagram
    participant User
    participant Browser
    participant MainThread as "Main JS Thread"
    participant WorkerThread as "Stats Worker"

    User->>Browser: Loads SatelliteCoverage.html
    Browser->>MainThread: Executes <script>
    MainThread->>MainThread: init()
    MainThread->>MainThread: setupUI()
    MainThread->>WorkerThread: new Worker()
    MainThread->>WorkerThread: initStatsWorker()
    MainThread->>MainThread: initMap() (loads TopoJSON)
    MainThread->>MainThread: resetSimulation()
    MainThread->>MainThread: requestAnimationFrame(simulationLoop
```



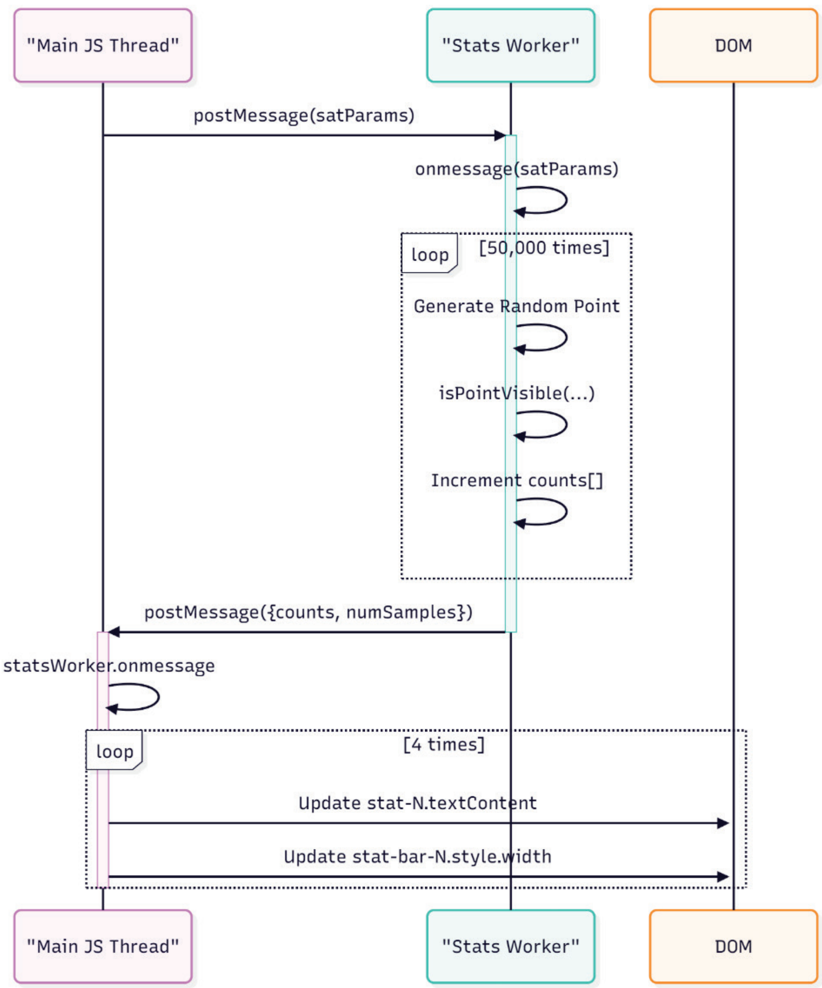
6.2. Main Simulation Tick

```
sequenceDiagram
    participant MainThread as "Main JS Thread"
    participant WorkerThread as "Stats Worker"

    loop Every Frame
        MainThread->>MainThread: simulationLoop(timestamp)
        MainThread->>MainThread: delta = ...
        MainThread->>MainThread: currentGST = ...

        loop 3 times
            MainThread->>MainThread: propagateSatellite(i, ...)
        end

        MainThread->>MainThread: drawMapElements() (Update SVG)
        MainThread->>MainThread: updateHeatmap() (Update Canvas)

        opt Throttled (every 500ms)
            MainThread->>WorkerThread: postMessage(satParams)
        end

        MainThread->>MainThread: requestAnimationFrame(simulationLoop)
    end
```



6.3. Statistics Calculation (Async)

```
sequenceDiagram
    participant MainThread as "Main JS Thread"
    participant WorkerThread as "Stats Worker"
    participant DOM

    MainThread->>WorkerThread: postMessage(satParams)

    activate WorkerThread
    WorkerThread->>WorkerThread: onmessage(satParams)
    loop 50,000 times
        WorkerThread->>WorkerThread: Generate Random Point
        WorkerThread->>WorkerThread: isPointVisible(...)
        WorkerThread->>WorkerThread: Increment counts[]
    end
    WorkerThread->>MainThread: postMessage({counts, numSamples})
    deactivate WorkerThread

    activate MainThread
    MainThread->>MainThread: statsWorker.onmessage
    loop 4 times
        MainThread->>DOM: Update stat-N.textContent
        MainThread->>DOM: Update stat-bar-N.style.width
    end
    deactivate MainThread
```



6.4. User Interaction (Slider Change)

```
sequenceDiagram
    participant User
    participant DOM
    participant MainThread as "Main JS Thread"

    User->>DOM: Drags 'Semi-major Axis' slider
    DOM->>MainThread: 'input' event

    activate MainThread
    MainThread->>MainThread: updateSatelliteUI(index)
    MainThread->>DOM: Update a-1_val.textContent
    MainThread->>DOM: Update perigee_warning-1
    deactivate MainThread

    Note over MainThread: On next animation frame...

    MainThread->>MainThread: simulationLoop()
    MainThread->>DOM: Reads new slider value
    MainThread->>MainThread: propagateSatellite(1, ...)
    Note over MainThread: ...uses new 'a' value
    MainThread->>MainThread: drawMapElements()
    MainThread->>MainThread: updateHeatmap()
    Note over MainThread: Visuals are now updated
```