Architecture & Detailed Design Report

Satellite Locator Application

Version: 1.0

Date: 8 November 2025

Contents

1. Introduction

1.1 Purpose of the Document

This document provides a comprehensive architectural overview and detailed design for the `SatelliteLocator.html` single-page HTML application. The application simulates and visualises the real-time position of an Earth-orbiting satellite, along with the positions of the Sun and Moon, on a 2D world map.

This report details the system's components, their interactions, the core logic and algorithms, external dependencies, and key data flows. It is intended to provide an understanding of the application's internal workings.

1.2 Application Overview

The Satellite Locator is a client-side web application with no backend. All logic, rendering, and calculation occurs within the user's browser.

Core Features:

- **Satellite Tracking:** Plots a satellite's ground track on a 2D map based on user-provided Two-Line Element (TLE) data.
- **Simulation Control:** Allows the user to set a simulation epoch, control the playback speed (from real-time to 5000x), and reset the simulation.
- **Real-time "Catch up":** A feature to fast-forward the simulation from a past epoch to the current real time.
- **TLE Handling:**
  - Allows manual entry of TLE data.
  - Provides buttons to fetch TLEs for common satellites (ISS, Tiangong, etc.) from a public API.
- **Orbital Data Display:** Calculates and displays key orbital parameters like altitude, period, apogee, perigee, inclination, and eccentricity.
- **Celestial Visualization:** Renders the real-time sub-solar point (Sun's position) and sub-lunar point (Moon's position) with its correct phase.
- **Map Visualization:**
  - Renders world coastlines and a lat/lon grid.
  - Displays the day/night terminator and twilight zones.
  - Renders a configurable-length ground track for the satellite.

2. System Architecture

2.1 High-Level Overview

The application follows a simple, monolithic, client-side architecture. It can be broken down into three logical layers contained within the single `SatelliteLocator.html` file:

1. **View (Presentation Layer):** The HTML structure and CSS styling. This layer is responsible for all UI elements, including the control panel, map container, and styling for rendered SVG elements.
2. **Controller (Application Logic Layer):** The core JavaScript code that acts as the "brains" of the application. It handles user events, manages the simulation state, and coordinates updates between the View and the Model.
3. **Model (Data & Services Layer):** This layer consists of the global state variables (e.g., `currentSimDate`, `satrec`) and the external libraries (`satellite.js`, `d3.js`) that provide the specialised services for orbital mechanics and map rendering.

2.2 Core Components & Interfaces

2.2.1 View (HTML/CSS)

- **HTML:** The DOM is split into two main parts:
  - `#panel-container`: Contains the main control panel (`#info`), the collapsible "mini-panel" (`#mini-panel`), and the toggle button (`#toggle-button`).
  - `#map-container`: A fullscreen container that holds the `#map-svg` element.
- **CSS:** A single `<style>` block defines all application styles. It uses a combination of ID-based selectors for major components and class-based selectors for reusable elements (`.sim-button`, `.display-item`, `.satellite`, `.sun-marker`, etc.).

2.2.2 Controller (JavaScript Application)

This is the central script running the application. It is event-driven and state-based.

- **State Management:** Global variables (e.g., `currentSimDate`, `satrec`, `isCatchingUp`) hold the entire application state.
- **Event Handling:** The `setupUI()` function attaches listeners to all interactive elements (buttons, sliders, inputs).
- **Simulation Engine:** The `simulationLoop()` function, driven by requestAnimationFrame, forms the application's heartbeat, constantly advancing time and recalculating positions.

2.2.3 Model (Data & External Libraries)

The application relies on several external interfaces:

- **`satellite.js` (External Library):**
  - **Purpose:** Provides all core orbital mechanics calculations.
  - **Interface:**
    - `satellite.twoline2satrec(tle1, tle2)`: Parses TLE strings into a satrec

(satellite record) object.
    - `satellite.propagate(satrec, date)`: Propagates the satrec to a specific Date object and returns position/velocity.
    - `satellite.eciToGeodetic(eciPos, gmst)`: Converts Earth-Centred Inertial (ECI) coordinates to Geodetic (lat, lon, alt).
    - `satellite.gstime(date)`: Calculates the Greenwich Mean Sidereal Time for a given Date.
- **`d3.js` & `topojson.js` (External Libraries):**
    - **Purpose:** Renders the SVG map and its geographic features.
    - **Interface:**
        - `d3.select()`: Selects DOM elements.
        - `d3.json()`: Loads the world coastline data.
        - `topojson.feature()`: Parses the TopoJSON map data.
        - `d3.geoEquirectangular()`: The map projection.
        - `d3.geoPath(projection)`: The SVG path generator.
        - `d3.geoGraticule()`: The lat/lon grid generator.
        - `d3.geoCircle()`: Used to draw the terminator/shadow zones.
- **TLE API (External Service):**
    - **Purpose:** Fetches TLE data for pre-selected satellites.
    - **Interface:**
        - `fetch('https://tle.ivanstanojevic.me/api/tle/{NORAD_ID}')`: A simple REST API that returns TLE data in JSON format.

3. Detailed Design - Controller (JavaScript)

3.1 Global State

The application's state is managed by a set of global variables:

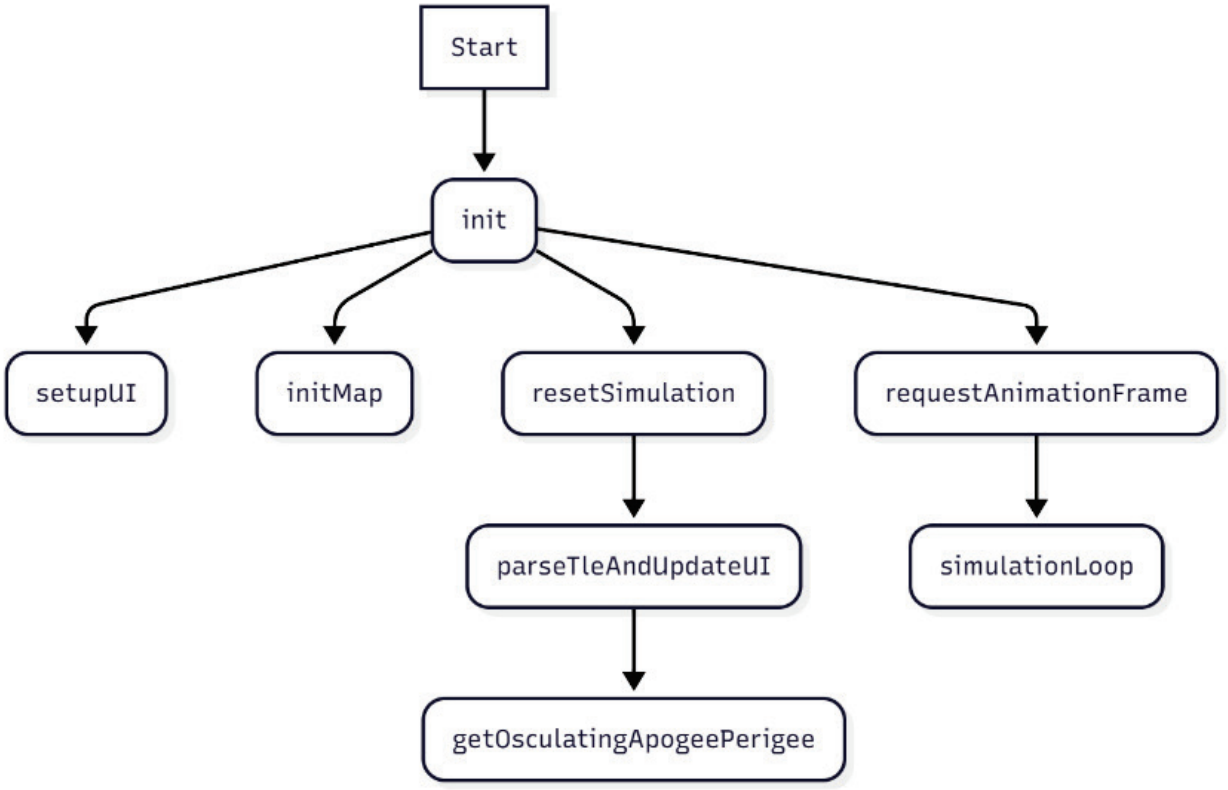
Variable	Type	Purpose
`svg, projection, path, graticule`	`Object`	`D3.js` objects for map rendering.
`lastTimestamp`	`Number`	Timestamp (ms) of the last animation frame.
`epochDate`	`Date`	The “zero point” of the simulation, set by the user.
`currentSimDate`	`Date`	The current “in-game” time, advanced by the loop.
`elapsedSimSeconds`	`Number`	Total sim seconds passed since `epochDate`.
`isCatchingUp`	`Boolean`	Flag set to true when “Catch up” mode is active.
`satelliteConfigs`	`Array`	Array holding the parsed `satrec` object.
`propagatedStates`	`Array`	Array holding the latest calculated state (lat, lon, alt).
`groundTracks`	`Array`	Array of `[lon, lat, time]` points for the ground track.
`sunPos`	`Object`	`{latitude, longitude}` of the sub-solar point.
`moonData`	`Object`	`{latitude, longitude, phase}` of the Moon.
`dom`	`Object`	A cached map of all key DOM elements.
`civilCircle, nauticalCircle, etc.`	`Object`	`D3.js` geo-circle generators for shadows.

3.2 Core Logic & Function Hierarchy

3.2.1 Initialization Flow

The application begins when the main `init()` function is called at the end of the script.

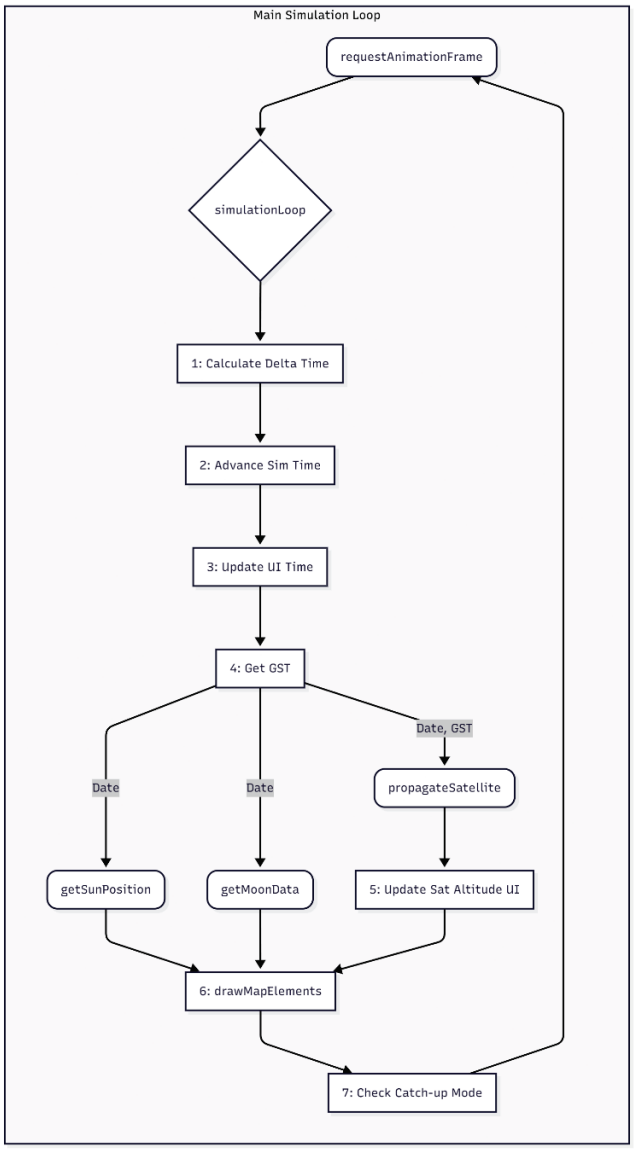
```
graph TD
    A[Start] --> B(init);
    B --> C(setupUI);
    B --> D(initMap);
    B --> E(resetSimulation);
    E --> F(parseTleAndUpdateUI);
    F --> G(getOsculatingApogeePerigee);
    B --> H(requestAnimationFrame);
    H --> I(simulationLoop);
```



3.2.2 Main Simulation Loop

The `simulationLoop` is the application's engine. It's a recursive function that drives all continuous updates.

```
graph TD
    subgraph simulationLoop [Main Simulation Loop]
        A(requestAnimationFrame) --> B{simulationLoop};
        B --> C[1: Calculate Delta Time];
        C --> D[2: Advance Sim Time];
        D --> E[3: Update UI Time];
        E --> F[4: Get GST];
        F -- Date --> G(getSunPosition);
        F -- Date --> H(getMoonData);
        F -- Date, GST --> I(propagateSatellite);
        I --> J[5: Update Sat Altitude UI];
        G & H & J --> K[6: drawMapElements];
        K --> L[7: Check Catch-up Mode];
        L --> A;
    end
```



3.3 Function Definitions

3.3.1 Initialisation & Setup

- **`init()`**
  - **Description:** The main entry point. Caches DOM elements, sets up UI, initialises the map, resets the simulation to the initial state, and starts the simulation loop.
  - **Calls:** `setupUI()`, `initMap()`, `resetSimulation()`, `requestAnimationFrame(simulationLoop)`
- **`setupUI()`**
  - **Description:** Attaches all event listeners to the DOM elements (buttons, sliders, inputs). Sets the initial state of UI controls.
  - **Calls:** `resetSimulation()`, `syncSpeed()`, `updateTrackLengthDisplay()`, `startCatchUp()`, `loadIssTle()`, `parseTleAndUpdateUI()`
- **`initMap()`**
  - **Description:** Configures the `D3.js` projection and path generator. Loads the TopoJSON world map data. Appends all necessary SVG layers (<g> elements) for coastlines, shadows, celestial bodies, and the satellite.
  - **Calls:** `onWindowResize()`, `drawMapElements()`

3.3.2 Event Handlers

- **`loadTle(tleId, buttonId, buttonText)`**
  - **Description:** An async function that fetches TLE data from the `tle.ivanstanojevic.me` API. It disables all TLE buttons, shows a "Loading..." state, and re-enables them on success or failure.
  - **Calls:** `fetch()`, `parseAndApplyTle()`
- **`parseAndApplyTle(line1, line2)`**
  - **Description:** Called by `loadTle`. Parses TLE's epoch to set simulation `epochDate`. Sets TLE text in text areas and calls `resetSimulation()` to apply the new satellite.
  - **Calls:** `resetSimulation()`
- **`startCatchUp()`**
  - **Description:** Triggered by the “Catch up” button. Checks if the `currentSimDate` is in the past. If so, it sets `isCatchingUp = true`, sets the speed to 5000, and disables the speed sliders and “Catch up” button.
- **`resetSimulation()`**
  - **Description:** Resets the simulation to the `epochDate`. Clears the `groundTracks` array, re-parses the TLE, and cancels “Catch up” mode if active.
  - **Calls:** `parseTleAndUpdateUI()`
- **`onWindowResize()`**
  - **Description:** Recalculates the SVG dimensions and D3 projection scale to fit the new window size. Redraws all geographic and dynamic map elements.
  - **Calls:** `drawMapElements()`

3.3.3 Core Simulation & Calculation Engine

- **`simulationLoop(timestamp)`**
  - **Description:** The core recursive loop. Calculates time delta, advances `currentSimDate`, calls all calculation functions (`getSunPosition`, `getMoonData`, `propagateSatellite`), and triggers the redraw (`drawMapElements`).
  - **Calls:** `getSunPosition()`, `getMoonData()`, `propagateSatellite()`, `drawMapElements()`, `requestAnimationFrame()`
- **`propagateSatellite(satrec, currentDate, gst)`**
  - **Description:** The primary satellite calculation function.
  - **Interface:** `satellite.propagate()` (to get ECI position), `satellite.eciToGeodetic()` (to get lat/lon/alt).
  - **Returns:** An object `{ latLon: [lon, lat], alt_km, period }`.
- **`getOsculatingApogeePerigee(satrec)`**
  - **Description:** A computationally intensive function called only by `parseTleAndUpdateUI` (not in the main loop). It simulates one full orbit at 360 steps to find the true min/max altitude (perigee/apogee) by sampling the instantaneous altitude at each step.
  - **Interface:** `satellite.propagate()`, `satellite.gstime()`, `satellite.eciToGeodetic()`
  - **Returns:** `{ perigeeAltitude, apogeeAltitude }`.
- **`getSunPosition(date, gmst_rad)`**
  - **Description:** Calculates the sub-solar point (the latitude/longitude where the Sun is directly overhead). See section 5.3 for algorithm.
  - **Returns:** `{ latitude, longitude }`.
- **`getMoonData(date, gmst_rad)`**
  - **Description:** Calculates the Moon's phase and its approximate sub-lunar point (latitude/longitude). See section 5.4 for algorithm.
  - **Returns:** `{ latitude, longitude, phase }`.

3.3.4 Rendering & Display

- **`drawMapElements()`**
  - **Description:** The main rendering function called at the end of every `simulationLoop`. It takes the latest calculated data from the global state (`propagatedStates`, `sunPos`, `moonData`) and updates the `cx`, `cy`, `transform`, and `d` attributes of the corresponding SVG elements.
  - **Logic:**
    1. **Satellite & Track:** Updates the satellite's `cx/cy`. Filters the `groundTracks` array based on the "Track Length" slider and redraws the ground track `<path>`.
    2. **Sun & Shadows:** Updates the Sun's `cx/cy`. Calculates the Sun's antipode and redraws the four shadow circles centred on it.
    3. **Moon:** Updates the Moon's `<g>` element transform. Calculates the SVG path `d` attribute for the lit portion based on the phase and applies it to the `moon-lit-side` path.

- **`parseTleAndUpdateUI(index)`**
  - **Description:** Reads the TLE strings from the `textareas`.
  - **Interface:** `satellite.twoline2satrec()`
  - **Logic:** On success, it calls `getOsculatingApogeePerigee()` and updates the "Satellite State" display in the UI. On failure (e.g., invalid TLE), it displays an error in the `#tle_warning` box and clears the data.
  - **Calls:** `getOsculatingApogeePerigee()`

4. Data Models & Algorithms

4.1 Satellite Propagation (SGP4)

The application uses the SGP4 algorithm provided by `satellite.js` to model the satellite's orbit.

- **Input:** TLE (Two-Line Element) set.
- **Process:**
  1. **Initialisation:** The TLE is parsed into a satrec object via `satellite.twoline2satrec()`. This is done once when the TLE is changed or the simulation is reset.
  2. **Propagation:** In each `simulationLoop`, the `satellite.propagate(satrec, currentSimDate)` function is called. This calculates the satellite's position and velocity in ECI (Earth-Centred Inertial) coordinates for the given time.
  3. **Coordinate Conversion:** The ECI coordinates are not useful for plotting on a 2D map. They must be converted to Geodetic coordinates (latitude, longitude, altitude). This requires the Greenwich Mean Sidereal Time (GMST), which accounts for the Earth's rotation.
     - `gst = satellite.gstime(currentSimDate)`
     - `geodetic = satellite.eciToGeodetic(eciPosition, gst)`
- **Output:** The satellite's `latitude`, `longitude` (in radians, converted to degrees), and `height` (in km).

4.2 Osculating Apogee/Perigee Algorithm

The apogee and perigee listed in the TLE are mean elements. To find the true (osculating) min/max altitude, this application simulates one full orbit.

- **Algorithm (in `getOsculatingApogeePerigee`):**
  1. Get the satellite's orbital period (in minutes) from the `satrec: periodMin = (2 * PI / satrec.no)`.
  2. Get the TLE epoch Date object.
  3. Define 360 simulation steps: `stepMs = (periodMin / 360) * 60 * 1000`.
  4. Initialise `minAlt = Infinity, maxAlt = -Infinity`.
  5. Loop `i` from 0 to 360:
     a. Calculate `stepDate = new Date(epochDate.getTime() + i * stepMs)`.
     b. Propagate the satellite to `stepDate` to get its ECI position.
     c. Calculate `gst` for `stepDate`.
     d. Convert ECI to Geodetic to get `alt_km`.
     e. if `(alt_km < minAlt) minAlt = alt_km`.
     f. if `(alt_km > maxAlt) maxAlt = alt_km`.
  6. Return `{ perigeeAltitude: minAlt, apogeeAltitude: maxAlt }`.

4.3 Sun Position Calculation

The Sun's position is calculated using a simplified astronomical model.

- **Algorithm (in `getSunPosition`):**
  1. Calculate **Julian Date** from the `currentSimDate`.
  2. Calculate **days since J2000 epoch** (`daysSince2000`).
  3. Calculate Sun's **Mean Longitude (L)** and **Mean Anomaly (g)** using standard formulae:
     - $L = (280.460 + 0.9856474 \times d) \bmod 360$
     - $g = (357.528 + 0.9856003 \times d) \bmod 360$
  4. Calculate **Ecliptic Longitude (λ)** by correcting for eccentricity:
     - $\lambda = L + 1.915 \times \sin(g) + 0.020 \times \sin(2g)$
  5. Calculate **Obliquity of the Ecliptic (ϵ)**:
     - $\epsilon = 23.439 - 0.0000004 \times d$
  6. Convert Ecliptic coordinates (λ, ϵ) to Equatorial coordinates (**Right Ascension** α, **Declination** δ):
     - $\alpha = \tan^{-1}(\cos\varepsilon \times \sin\lambda, \cos\lambda)$
     - $\delta = \sin^{-1}(\sin\varepsilon \times \sin\lambda)$
  7. Convert Equatorial to Geodetic (Lat, Lon):
     - `latitude = Declination (`δ`)`
     - `longitude = Right Ascension (`α`) - GMST`
  8. Return `{ latitude, longitude }` in degrees.

4.4 Moon Position & Phase Calculation

The Moon's calculation is a simplified model that derives its position from its phase relative to the Sun.

- **Algorithm (in `getMoonData`):**
  1. **Calculate Phase:**
     - Find milliseconds since a **Known New Moon** (`KNOWN_NEW_MOON_MS`).
     - Divide by the Moon's **Synodic Period** (`SYNODIC_PERIOD_DAYS`) and take the modulus.
     - `phase = (daysSinceNewMoon / SYNODIC_PERIOD_DAYS) % 1`
     - This gives a value from 0 (New Moon) to 0.5 (Full Moon) to 1.0 (New Moon).
  2. **Calculate Position:**
     - This model approximates the Moon's position by assuming its orbital plane is the same as the Sun's (the ecliptic) and its angular separation from the Sun is determined by its phase.
     - Get the Sun's **Ecliptic Longitude (λ_{sun})** (from steps 1-4 in 4.3).
     - Calculate the Moon's approximate **Ecliptic Longitude (λ_{moon})**:
       - $\lambda_{moon} = \big(\lambda_{sun} + (phase \times 360)\big) \bmod 360$
     - Convert λ_{moon} to **Right Ascension (α_{moon})** and **Declination (δ_{moon})** using the same equations as in 4.3 (step 6).
     - Convert to Geodetic (Lat, Lon):

        - latitude = \delta_{moon}
        - longitude = \alpha_{moon} - GMST
    3. Return { latitude, longitude, phase }.

4.5 Moon Phase SVG Path Generation

The Moon's phase is drawn using a clever SVG path. A <g> element contains two parts:

1. A base circle (.moon-dark-side) that is always dark grey.
2. A <path> (.moon-lit-side) drawn on top, representing the lit portion.

The d attribute of this path is dynamically calculated in drawMapElements based on the phase:

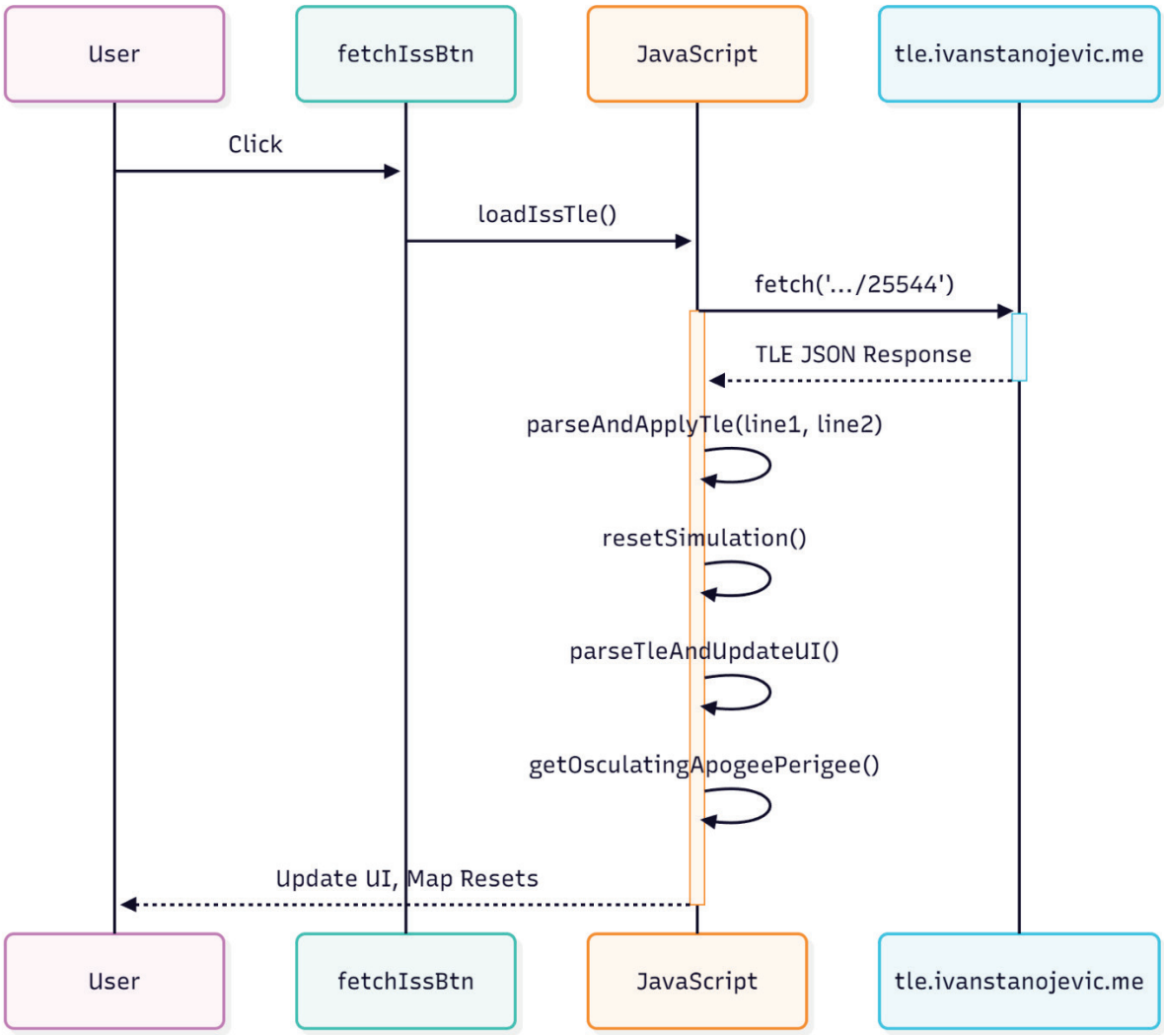
- A **Waxing** phase (0 to 0.5) is drawn as a combination of the Moon's right-side arc and an elliptical arc for the terminator.
- A **Waning** phase (0.5 to 1.0) is drawn as a combination of the Moon's left-side arc and an elliptical arc for the terminator.
- The "bulge" of the elliptical arc is controlled by $x_{bulge} = -r \times \cos(phase \times 2\pi)$, which simulates the crescent and gibbous shapes.

5. Sequence Diagrams

5.1 Use Case: User Loads a New TLE (e.g., “International Space Station”)

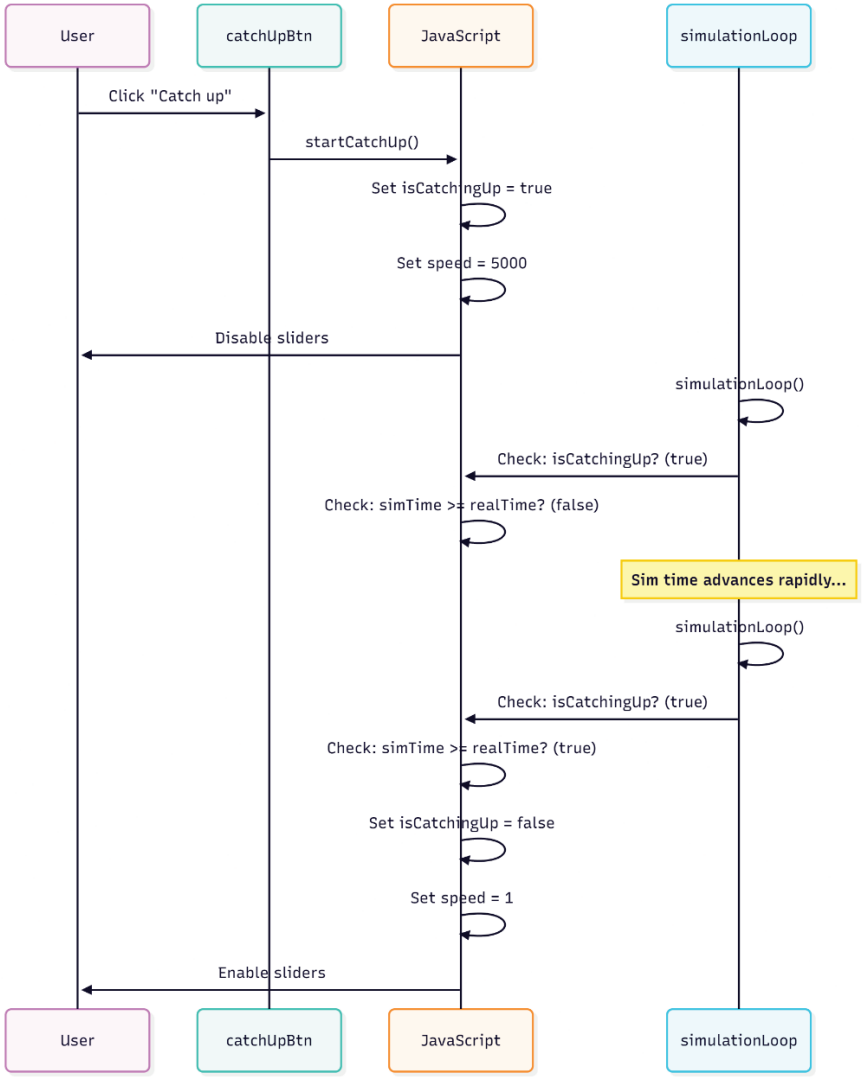
```
sequenceDiagram
    participant User
    participant Button as fetchIssBtn
    participant App as JavaScript
    participant API as tle.ivanstanojevic.me

    User->>Button: Click
    Button->>App: loadIssTle()
    App->>API: fetch('.../25544')
    activate App
    activate API
    API-->>App: TLE JSON Response
    deactivate API
    App->>App: parseAndApplyTle(line1, line2)
    App->>App: resetSimulation()
    App->>App: parseTleAndUpdateUI()
    App->>App: getOsculatingApogeePerigee()
    App-->>User: Update UI, Map Resets
    deactivate App
```



5.2 Use Case: User clicks "Catch up"

```
sequenceDiagram
    participant User
    participant Button as catchUpBtn
    participant App as JavaScript
    participant SLoop as simulationLoop
    User->>Button: Click "Catch up"
    Button->>App: startCatchUp()
    App->>App: Set isCatchingUp = true
    App->>App: Set speed = 5000
    App->>User: Disable sliders
    SLoop->>SLoop: simulationLoop()
    SLoop->>App: Check: isCatchingUp? (true)
    App->>App: Check: simTime >= realTime? (false)
    Note over SLoop: Sim time advances rapidly...
    SLoop->>SLoop: simulationLoop()
    SLoop->>App: Check: isCatchingUp? (true)
    App->>App: Check: simTime >= realTime? (true)
    App->>App: Set isCatchingUp = false
    App->>App: Set speed = 1
    App->>User: Enable sliders
```



6. Standards & Dependencies

Type	Name	Version	Purpose
Standard	HTML5	-	Document structure.
Standard	CSS3	-	Document styling.
Standard	JavaScript (ES6)	-	Application logic.
Library	`satellite.js`	6.0.1	SGP4 propagation and coordinate transformations.
Library	`d3.js`	7.x	SVG map rendering, projections, and element manipulation.
Library	`topojson.js`	3.x	Parsing of the TopoJSON world map file.
Service	TLE API	-	`tle.ivanstanojevic.me` for fetching TLE data.