Detailed Design Report: Orbit 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 Orbit Visualiser application. It outlines the system architecture, core logic, astrodynamical calculations, and user interface implementation based on the provided `OrbitVisualiser.html` file.

1.2 Project Overview

The Orbit Visualiser is an interactive, web-based tool designed to demonstrate the principles of orbital mechanics. It renders a 3D, true-to-scale model of an Earth-orbiting satellite. Users can manipulate the six classical orbital elements (COEs) via sliders and observe the real-time effects on the orbit's size, shape, and orientation. The application displays key derived orbital parameters and the satellite's real-time state, providing an educational experience for understanding astrodynamics.

2. System Architecture

2.1 Architectural Style

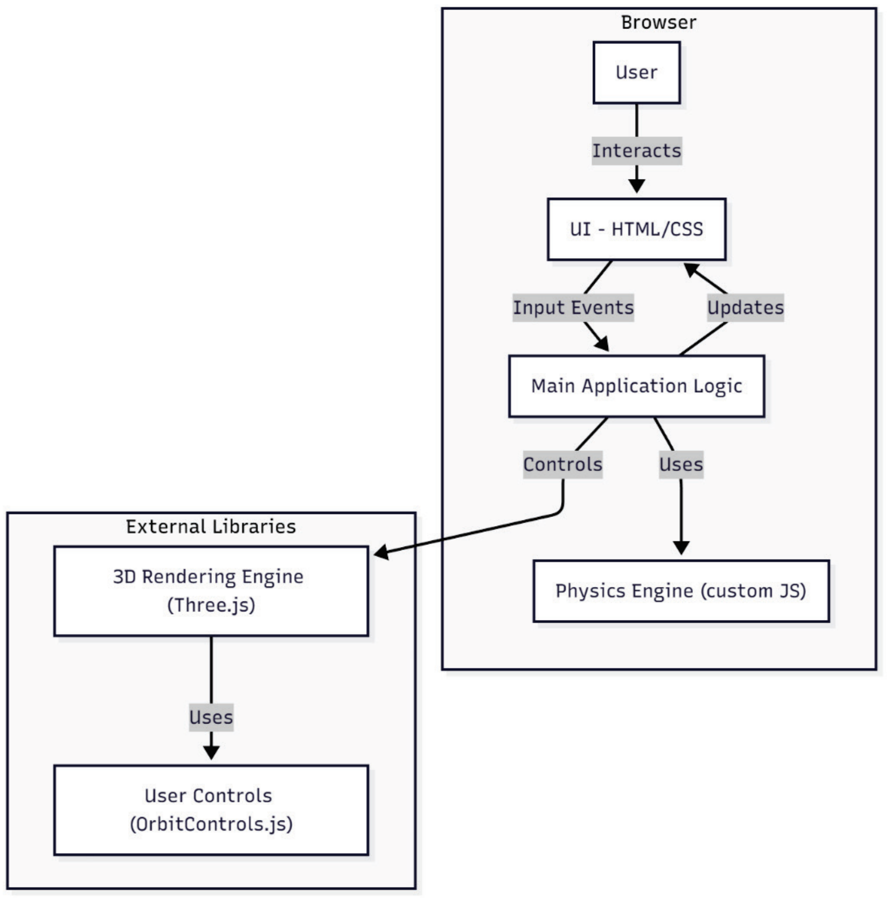
The application is a **single-page application (SPA)** contained within a single HTML file. This file encapsulates three distinct layers:

- **Presentation Layer (HTML/CSS):** Defines the structure and appearance of the user interface, including the control panel and the 3D scene container.
- **Logic Layer (JavaScript):** Manages the application state, handles user input, performs all physics calculations, and updates the 3D scene and UI panels.
- **Visualization Layer (Three.js):** An external library integrated into the logic layer, responsible for WebGL rendering of the 3D objects (Earth, satellite, orbit path).

2.2 UML Component Diagram

The logical architecture of the application can be represented by the following components:

```
flowchart TB
  subgraph subGraph0["Browser"]
        id1["User"]
        id2["UI - HTML/CSS"]
        id3["Main Application Logic"]
        id4["Physics Engine (custom JS)"]
  end
  subgraph subGraph1["External Libraries"]
        id5["3D Rendering Engine (Three.js)"]
        id6["User Controls (OrbitControls.js)"]
  end
    id1 -- Interacts --> id2
    id2 -- Input Events --> id3
    id3 -- Updates --> id2
    id3 -- Uses --> id4
    id3 -- Controls --> id5
    id5 -- Uses --> id6
```



2.3 Component Descriptions

Component	Description	Technology
UI (User Interface)	Renders the control panel, sliders, and data displays. Captures user interactions (e.g., slider movements, button clicks).	HTML, CSS
Main Application Logic	The core orchestrator. Initializes the application (init), handles UI events, calls the physics engine, and updates the 3D scene.	JavaScript
Physics Engine	A set of custom JavaScript functions that implement astrodynamical equations (e.g., solveKepler, calculatePositionVector).	JavaScript
3D Rendering Engine	Manages the WebGL scene, camera, lighting, and rendering of all 3D objects.	Three.js
User Controls	A Three.js add-on that enables mouse-based camera controls (orbit, pan, zoom).	OrbitControls.js

3. Core Logic and Function Interfaces

The application's logic is event-driven and centred around an animation loop.

3.1 Key Functions

Function	**Parameters**	**Returns**	**Description**
`init()`	None	`void`	Initializes the entire application: sets up the scene, camera, renderer, objects, UI listeners, and starts the animation loop.
`setupUI()`	None	`void`	Sets initial slider values and attaches event listeners to them. Manages the collapsible panel logic.
`updateOrbit()`	None	`void`	Reads slider values, recalculates and redraws the static orbit line, updates derived parameters, and resets physics constants like meanMotion.
`solveKepler(M, e)`	`M` (Number), `e` (Number)	`Number`	Solves Kepler's Equation for the Eccentric Anomaly (E) using the Newton-Raphson iterative method.
`calculatePositionVector(nu)`	`nu` (Number)	`Vector3`	Calculates the 3D position vector of the satellite in the ECI frame for a given true anomaly.
`updateSatellite(dt)`	`dt` (Number)	`void`	Calculates the satellite's new position for the current frame based on elapsed time and orbital physics.
`animate()`	None	`void`	The main animation loop. Called once per frame, it updates controls, object rotations, and calls updateSatellite.
`onWindowResize()`	None	`void`	Handles browser window resizing to keep the 3D scene correctly proportioned.

3.2 Global Constants and Variables

- **Constants:** `EARTH_RADIUS`, `MU_EARTH` (gravitational parameter), `SCALE` (for rendering).
- **State Variables:** `meanMotion` (orbital speed), `meanAnomaly` (current position in a simplified model).
- **Three.js Objects:** `scene`, `camera`, `renderer`, `controls`, `earth`, `satellite`, etc.

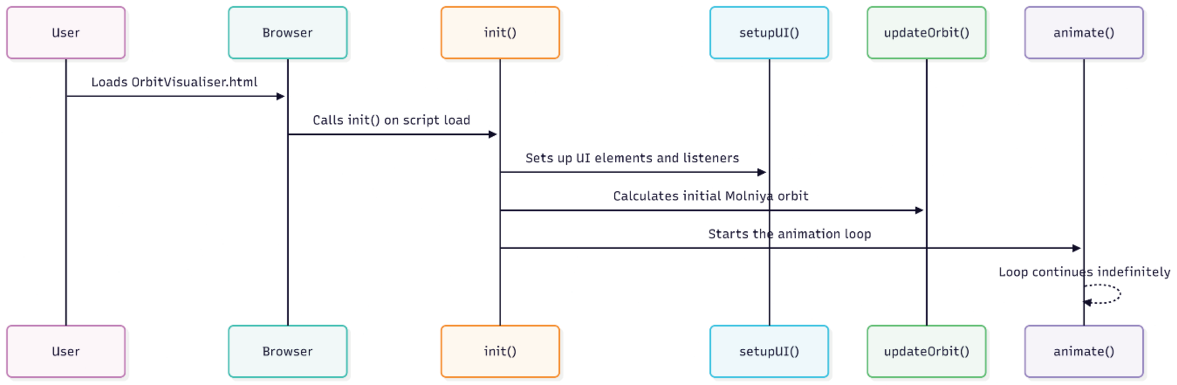
4. Dynamic Behaviour (Sequence Diagrams)

4.1 Application Initialization

This diagram shows the sequence of events when the page first loads.

```
sequenceDiagram
    participant User
    participant Browser
    participant init()
    participant setupUI()
    participant updateOrbit()
    participant animate()

    User->>Browser: Loads OrbitVisualiser.html
    Browser->>init(): Calls init() on script load
    init()->>setupUI(): Sets up UI elements and listeners
    init()->>updateOrbit(): Calculates initial Molniya orbit
    init()->>animate(): Starts the animation loop
    animate()-->>animate(): Loop continues indefinitely
```


User
Browser
init()
setupUI()
updateOrbit()
animate()
Loads OrbitVisualiser.html
Calls init() on script load
Sets up UI elements and listeners
Calculates initial Molniya orbit
Starts the animation loop
Loop continues indefinitely

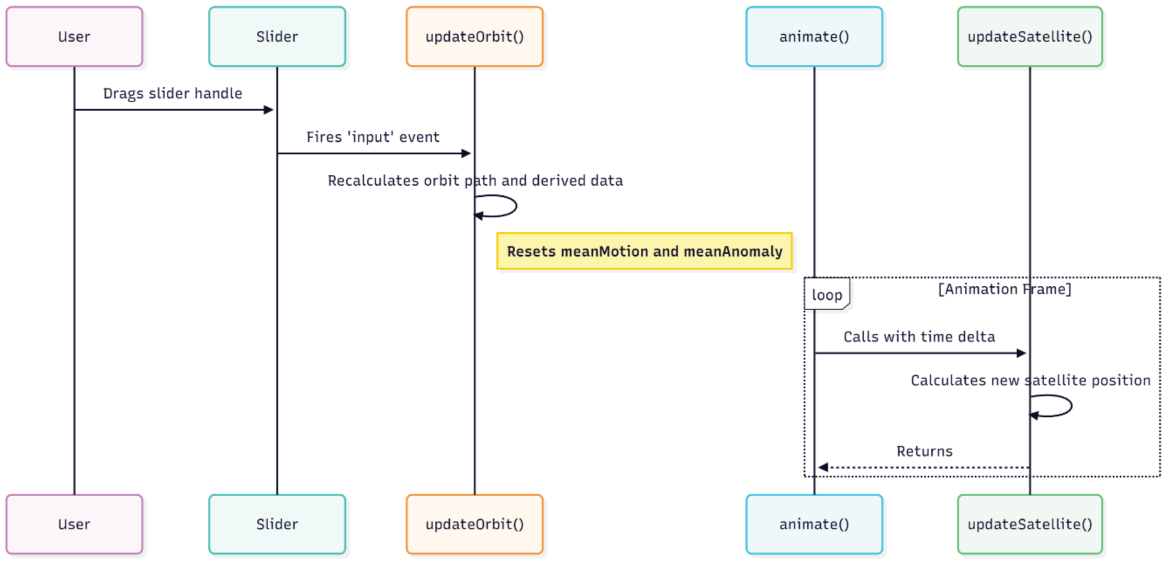
4.2 User Input and Animation Loop

This diagram shows what happens when a user interacts with a slider.

```
sequenceDiagram
    participant User
    participant Slider
    participant updateOrbit()
    participant animate()
    participant updateSatellite()

    User->>Slider: Drags slider handle
    Slider->>updateOrbit(): Fires 'input' event
    updateOrbit()->>updateOrbit(): Recalculates orbit path and derived data
    Note right of updateOrbit(): Resets meanMotion and meanAnomaly

    loop Animation Frame
        animate()->>updateSatellite(): Calls with time delta
        updateSatellite()->>updateSatellite(): Calculates new satellite position
        updateSatellite()-->>animate(): Returns
     end
```



5. Astrodynamics Implementation

The core of the simulation is the correct application of orbital mechanics equations.

5.1 Coordinate Systems

The simulation uses two primary coordinate systems:

1. **Orbital (Perifocal) Frame:** A 2D frame where the orbit lies on the x-y plane, with the x-axis pointing towards the perigee (point of closest approach). Calculations are simplest in this frame.
2. **Earth-Centered Inertial (ECI) Frame:** A 3D frame with the Earth at the origin. The Z-axis points to the North Pole, and the X-axis points towards the vernal equinox. This is the standard frame for representing orbits in 3D space.

The function `calculatePositionVector` is responsible for converting coordinates from the Orbital Frame to the ECI Frame.

5.2 Fundamental Equations

5.2.1 Mean Motion (n)

The mean motion is the average angular speed of the satellite.

Equation:

$$n = \sqrt{\frac{\mu}{a^3}}$$

where:

- μ (MU_EARTH) is the gravitational parameter of the Earth.
- a is the semi-major axis.

Implementation (`updateOrbit`):

```
const T_seconds = 2 * Math.PI * Math.sqrt(Math.pow(a, 3) / MU_EARTH);
// ...
meanMotion = 2 * Math.PI / T_seconds;
```

5.2.2 Kepler's Equation and Solver

It's impossible to get the satellite's position directly from time. We must first solve Kepler's Equation, which relates mean anomaly (a proxy for time) to eccentric anomaly (a geometric parameter).

Equation:

$$M = E - e\sin(E)$$

where:

- M (`meanAnomaly`) is the mean anomaly.

- E is the eccentric anomaly.
- e is the eccentricity.

This equation has no closed-form solution for E. The code uses the Newton-Raphson method to solve it iteratively.

Implementation (`solveKepler`):

```
function solveKepler(M, e) {
    let E = M; // Initial guess
    const tolerance = 1e-6;
    for (let i = 0; i < 100; i++) {
        const f = E - e * Math.sin(E) - M;
        const f_prime = 1 - e * Math.cos(E);
        const delta = f / f_prime;
        E -= delta;
        if (Math.abs(delta) < tolerance) return E;
    }
    return E;
}
```

5.2.3 True Anomaly (v) from Eccentric Anomaly (E)

Once E is known, the true anomaly ν (the actual angle of the satellite from perigee) can be found.

Equation:

$$\tan\left(\frac{\nu}{2}\right) = \sqrt{\frac{1+e}{1-e}}\tan\left(\frac{E}{2}\right)$$

The code uses a more numerically stable `atan2` implementation.

Implementation (`updateSatellite`):

```
const sin_nu = (Math.sqrt(1 - e * e) * Math.sin(E_rad)) / (1 - e *
Math.cos(E_rad));
const cos_nu = (Math.cos(E_rad) - e) / (1 - e * Math.cos(E_rad));
const nu_rad = Math.atan2(sin_nu, cos_nu);
```

5.2.4 Position in the Orbital Frame

The satellite's distance r from Earth and its position (x_orb, y_orb) in the 2D orbital plane are calculated.

Equation:

$$r = \frac{a(1-e^2)}{1+e\cos(\nu)}$$
$$x_{orb} = r\cos(\nu)$$
$$y_{orb} = r\sin(\nu)$$

Implementation (`calculatePositionVector`):

```
const r_mag = a * (1 - e * e) / (1 + e * Math.cos(true_anomaly_rad));
const x_orb = r_mag * Math.cos(true_anomaly_rad);
const y_orb = r_mag * Math.sin(true_anomaly_rad);
```

5.2.5 Transformation to ECI Frame

The 2D orbital coordinates are rotated into the 3D ECI frame using a series of three rotations based on the orientation parameters: inclination (i), RAAN (Ω), and argument of perigee (ω).

Implementation (`calculatePositionVector`): The code pre-calculates the components of the rotation matrix (R_{xx}, R_{xy}, etc.) and applies them to transform [x_{orb}, y_{orb}, 0] into [x_{eci}, y_{eci}, z_{eci}].

5.3 Derived Orbital Parameters

The UI displays several parameters calculated directly from the primary orbital elements. Key examples include:

- **Orbital Period (T):**

$$T = 2\pi\sqrt{a^3/\mu}$$

- **Specific Orbital Energy (ε):**

$$\varepsilon = -\mu/2a$$

- **Perigee Altitude:**

$$r_p = a(1-e) - R_{Earth}$$

- **Apogee Altitude:**

$$r_a = a(1+e) - R_{Earth}$$

These are calculated within the `updateOrbit` function whenever a slider is changed.

6. User Interface (UI)

6.1 Layout and Styling

The UI is built with standard HTML and styled using CSS Flexbox to create a two-column layout: the control panel (`#info`) and the 3D canvas (`#scene-container`). The visual appearance (fonts, colors, blur effects) is defined in the `<style>` block.

6.2 Controls and Data Display

- **Controls:** `<input type="range">` elements are used for all user-adjustable parameters.
- **Data:** `<span>` elements within a flexbox layout (`.display-item`) are used to show parameter names and their real-time values.

6.3 UI Interactivity

JavaScript `addEventListener` is used to link UI elements to functions:

- The `input` event on each slider is tied to the `updateOrbit` or `updateSpeedDisplay` functions.
- The `click` event on the `#toggle-button` adds or removes the `.collapsed` class from the `#info` panel, triggering a CSS transition to hide or show it. The button itself is positioned relative to the panel to ensure it remains accessible.