Design & Architecture Report

Hohmann Transfer Calculator

Version: 1.0

Date: 14 November 2025

Contents

1. Introduction

1.1 Document Purpose

This document provides a comprehensive overview for the `HohmannTransfer.html` single-page HTML application. It outlines the structure, data flow, core logic, and visualisation engine. The application is contained within a single HTML file, embedding its structure (HTML), style (CSS), and logic (JavaScript).

1.2 System Overview

The Hohmann Transfer Calculator is a “vanilla” JavaScript (ES6+) application with no external dependencies. Its primary purpose is to:

1. **Accept user inputs** for an initial orbit, a target orbit, and rocket parameters (mass, I_{sp}) around a selected central body.
2. **Calculate the optimal bi-tangential (Hohmann-style) transfer** between these orbits.
3. **Display the results** of the calculation, including delta-V (Δv), transfer time, and required propellant.
4. **Provide a dynamic SVG visualization** of the initial orbit, target orbit, and the calculated transfer ellipse.
5. **Offer advanced warnings** when a bi-elliptic transfer might be more fuel-efficient.

1.3 Technology Stack

- **Structure**: HTML5
- **Styling**: CSS3
- **Logic:** JavaScript (ES6+ Strict Mode)
- **Visualisation:** Scalable Vector Graphics (SVG)

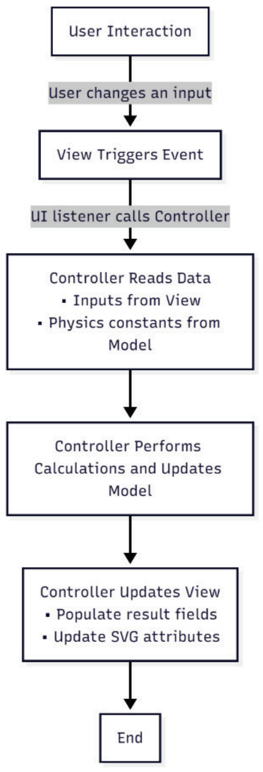
2. System Architecture

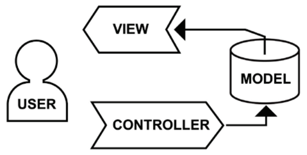
2.1 Architectural Model

The application follows a monolithic frontend architecture, with all code self-contained. Conceptually, its operation mimics a **Model-View-Controller (MVC)** pattern:

- **Model:** The application's state. This includes:
  - Static physics data (e.g., `CENTRAL_BODIES`, `G_CONST`).
  - User-provided input values from the DOM.
  - Global state variables (e.g., `g_scale`, `CURRENT_GM_M3_S2`).
  - The calculated result object (`g_transferData`).
- **View:** The user interface, composed of:
  - **HTML DOM:** The control panel (`#info`) with all its inputs, labels, and result fields.
  - **SVG DOM:** The visualization canvas (`#viz-svg`) containing `<ellipse>` and `<circle>` elements.
- **Controller:** The JavaScript functions that act as the "engine". They respond to user events (e.g., button clicks, window resizing), update the Model, perform calculations, and update the View in response. Key controller functions include `calculateAndDraw`, `updateCentralBody`, and `setupUI`.

2.2 Architectural Diagram





1. The **User** interacts with the **View** (e.g., changes an input).
2. The **View** triggers an event, which is handled by the **Controller** (e.g., `setupUI` listener calls `calculateAndDraw`).
3. The **Controller** reads data from the View (inputs) and the **Model** (physics constants).
4. The **Controller** performs calculations and updates the **Model** (populates `g_transferData`).
5. The **Controller** then pushes updates back to the **View** (populates result fields and updates SVG attributes).

```
flowchart TD
 A[User Interaction] -- User changes an input --> B[View Triggers Event]
 B -- UI listener calls Controller --> C[Controller Reads Data<br/>•
Inputs from View<br/>• Physics constants from Model]
 C --> D[Controller Performs Calculations and Updates Model]
 D --> E[Controller Updates View<br/>• Populate result fields<br/>• Update
SVG attributes]
 E --> F[End]
```

3. Data and State Management

3.1 Global State

The application's state is managed by a few key global variables:

- `g_scale`: (Number) The current visualization scale in pixels-per-kilometre. It is dynamically recalculated on window resize or new calculations.
- `g_transferData`: (Object | null) A "result" object that stores all calculated parameters for the transfer. It acts as the single source of truth for the visualization. It is `null` if no valid calculation has been performed.
- `CURRENT_BODY_RADIUS_KM`: (Number) The radius (in km) of the currently selected central body.
- `CURRENT_GM_M3_S2`: (Number) The standard gravitational parameter (μ or GM) of the selected body, in m^3/s^2. This is a critical constant for all physics calculations.

3.2 Key Data Structures

3.2.1 `CENTRAL_BODIES` (Constant)

A read-only object mapping body names to their physical properties.

JavaScript

```javascript
// Example Structure
"earth": {
    name: "Earth",
    mass_kg: 5.972e24,
    radius_km: 6378.137,
    peri_val: 0.983, // Perihelion (AU) or Perigee (km)
    apo_val: 1.017,  // Aphelion (AU) or Apogee (km)
    soi_km: 925000,
    color: "#336699"
}
```

3.2.2 `g_transferData` (Dynamic State Object)

This object is populated by `calculateAndDraw` and is the core "Model" for the results and visualisation.

JavaScript

```javascript
// Example Structure for a valid calculation
g_transferData = {
    r_p1_km: 8000.0,    // Initial orbit periapsis radius (km)
    r_a1_km: 8000.0,    // Initial orbit apoapsis radius (km)
    r_p2_km: 42164.0,   // Target orbit periapsis radius (km)
    r_a2_km: 42164.0,   // Target orbit apoapsis radius (km)

    // Transfer ellipse radii
    r_p_t_km: 8000.0,   // Transfer periapsis radius (km)
    r_a_t_km: 42164.0,  // Transfer apoapsis radius (km)

    // Burn 1 details
    burn1_r_km: 8000.0,         // Radius of burn 1 (km)
    burn1_isApoapsis: false,    // Burn is at periapsis
    burn1_dv: 2450.5,           // Delta-V for burn 1 (m/s)

    // Burn 2 details
    burn2_r_km: 42164.0,        // Radius of burn 2 (km)
    burn2_isApoapsis: true,     // Burn is at apoapsis
    burn2_dv: 1475.0            // Delta-V for burn 2 (m/s)
};
```

3.2.3 dom (Constant)

A single cache object, populated at startup, that holds references to all necessary DOM elements for fast access.

JavaScript

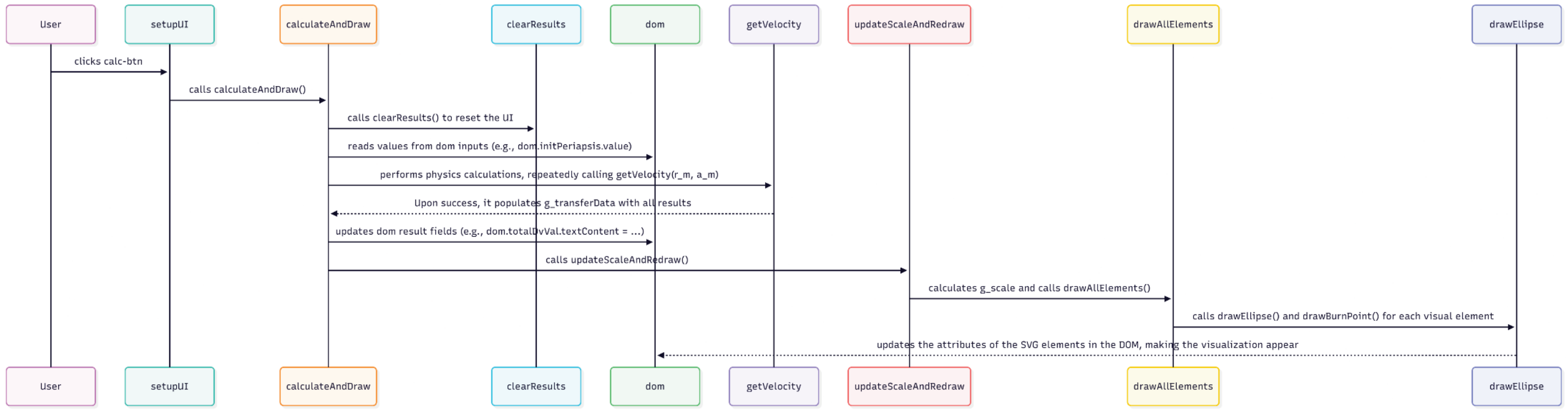
```javascript
// Abridged example
const dom = {
    infoDiv: document.getElementById('info'),
    vizSvg: document.getElementById('viz-svg'),
    initPeriapsis: document.getElementById('init-periapsis'),
    totalDvVal: document.getElementById('total-dv-val'),
    orbitInitial: document.getElementById('orbit-initial'),
    // ... all other element references };
```

3.3 Data Flow

This diagram (next page) illustrates the sequence of events for the application's main use case: clicking “Calculate Transfer”.

Sequence Breakdown:

1. **User** clicks `calc-btn`.
2. The `setupUI` **listener** calls `calculateAndDraw()`.
3. `calculateAndDraw()` calls `clearResults()` to reset the UI.
4. It then reads values from `dom` **inputs** (e.g., `dom.initPeriapsis.value`).
5. It performs physics calculations, repeatedly calling `getVelocity(r_m, a_m)`.
6. Upon success, it **populates** `g_transferData` with all results.
7. It updates `dom` **result fields** (e.g., `dom.totalDvVal.textContent = ...`).
8. It calls `updateScaleAndRedraw()`.
9. `updateScaleAndRedraw()` calculates `g_scale` and calls `drawAllElements()`.
10. `drawAllElements()` calls `drawEllipse()` and `drawBurnPoint()` for each visual element.
11. `drawEllipse()` and `drawBurnPoint()` update the attributes of the **SVG elements** in the DOM, making the visualisation appear.


User
setupUI
calculateAndDraw
clearResults
dom
getVelocity
updateScaleAndRedraw
drawAllElements
drawEllipse
clicks calc-btn
calls calculateAndDraw()
calls clearResults() to reset the UI
reads values from dom inputs (e.g., dom.initPeriapsis.value)
performs physics calculations, repeatedly calling getVelocity(r_m, a_m)
Upon success, it populates g_transferData with all results
updates dom result fields (e.g., dom.totalDvVal.textContent = ...)
calls updateScaleAndRedraw()
calculates g_scale and calls drawAllElements()
calls drawEllipse() and drawBurnPoint() for each visual element
updates the attributes of the SVG elements in the DOM, making the visualization appear

```
sequenceDiagram
  participant User as User
  participant setupUI as setupUI
  participant calculateAndDraw as calculateAndDraw
  participant clearResults as clearResults
  participant dom as dom
  participant getVelocity as getVelocity
  participant updateScaleAndRedraw as updateScaleAndRedraw
  participant drawAllElements as drawAllElements
  participant drawEllipse as drawEllipse
  User ->> setupUI: clicks calc-btn
  setupUI ->> calculateAndDraw: calls calculateAndDraw()
  calculateAndDraw ->> clearResults: calls clearResults() to reset the UI
  calculateAndDraw ->> dom: reads values from dom inputs (e.g., dom.initPeriapsis.value)
  calculateAndDraw ->> getVelocity: performs physics calculations, repeatedly calling getVelocity(r_m, a_m)
  getVelocity -->> calculateAndDraw: Upon success, it populates g_transferData with all results
  calculateAndDraw ->> dom: updates dom result fields (e.g., dom.totalDvVal.textContent = ...)
  calculateAndDraw ->> updateScaleAndRedraw: calls updateScaleAndRedraw()
  updateScaleAndRedraw ->> drawAllElements: calculates g_scale and calls drawAllElements()
  drawAllElements ->> drawEllipse: calls drawEllipse() and drawBurnPoint() for each visual element
  drawEllipse -->> dom: updates the attributes of the SVG elements in the DOM, making the visualization appear
```

4. Core Logic and Physics

The application's logic is cantered in the `calculateAndDraw` function, which relies on several fundamental physics equations. All calculations are performed in **meters (m), kilograms (kg), and seconds (s)**, and are only converted to km or min for final display.

4.1 Physics Equations

4.1.1 Gravitational Parameter (μ)

This is calculated once in `updateCentralBody` and stored globally.

$$\mu = GM$$

- `CURRENT_GM_M3_S2 = G_CONST * body.mass_kg`

4.1.2 Vis-viva Equation (Orbital Velocity)

This is the most-used equation, implemented in `getVelocity(r_m, a_m)`. It calculates the velocity (v) of an object at a specific radius (r) in an orbit with a semi-major axis (a).

$$v = \sqrt{\mu\left(\frac{2}{r} - \frac{1}{a}\right)}$$

- `return Math.sqrt(CURRENT_GM_M3_S2 * ((2 / r_m) - (1 / a_m)))`

4.1.3 Transfer Time

The time of flight for the transfer is half the orbital period (T) of the transfer ellipse.

$$t_{transfer} = \frac{T}{2} = \frac{1}{2}\,2\pi\sqrt{\frac{a_t^3}{\mu}} = \pi\sqrt{\frac{a_t^3}{\mu}}$$

`transfer_time_s = Math.PI * Math.sqrt(Math.pow(a_t_m, 3) / CURRENT_GM_M3_S2)`

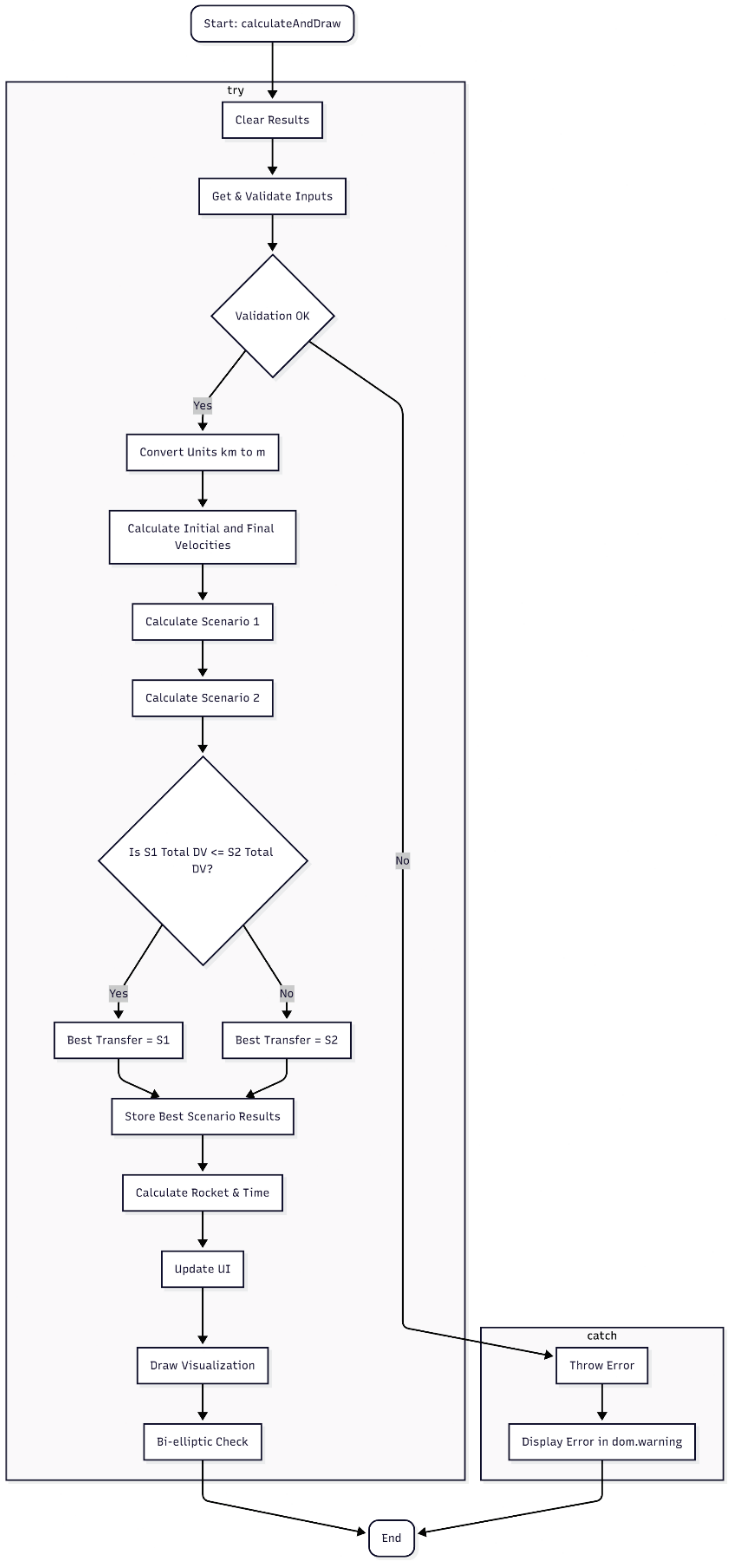
4.1.4 Tsiolkovsky Rocket Equation

Used to find the required propellant mass.

1. **Exhaust Velocity** (v_e): `ve = isp_s * G0`
2. **Mass Ratio (R):** `massRatio = Math.exp(best_total_dv / ve)`
3. **Initial Mass (m_i):** `initialMass_kg = finalMass_kg * massRatio`
4. **Propellant Mass (m_p):** `propellantMass_kg = initialMass_kg - finalMass_kg`

4.2 Hohmann Transfer Logic

The core `calculateAndDraw` function follows the displayed logical flow



```
graph TD
    A(Start: calculateAndDraw) --> B[Clear Results];

    subgraph try
        B --> C[Get & Validate Inputs];
        C --> D{Validation OK};
        D -- Yes --> F[Convert Units km to m];
        F --> G[Calculate Initial and Final Velocities];
        G --> H[Calculate Scenario 1];
        H --> I[Calculate Scenario 2];
        I --> J{Is S1 Total DV <= S2 Total DV?};
        J -- Yes --> K[Best Transfer = S1];
        J -- No --> L[Best Transfer = S2];
        K --> M[Store Best Scenario Results];
        L --> M;
        M --> N[Calculate Rocket & Time];
        N --> O[Update UI];
        O --> P[Draw Visualization];
        P --> Q[Bi-elliptic Check];
    end

    D -- No --> S[Throw Error];

    subgraph catch
        S --> T[Display Error in dom.warning];
    end

    Q --> R(End);
    T --> R;
```

Flowchart Explanation:

1. **Start:** `calculateAndDraw` is called.
2. **Clear Results:** All UI fields are reset.
3. **Get & Validate Inputs**:
    a. Read all 6 inputs from the `dom`.
    b. Check for `NaN`.
    c. Check `apoapsis >= periapsis`.
    d. Check `altitude > 100km`.
    e. If any check fails, **throw an Error**.
4. **Convert Units:** Convert all 4 orbit altitudes (km) to radii (meters) from the centre of the body.
5. **Calculate Initial/Final Velocities:** Use `getVelocity` to find $v_{i,p1}, v_{i,a1}, v_{f,p2}, v_{f,a2}$.
6. **Calculate Scenario 1 (S1)**:
    a. Transfer from Initial Periapsis (r_{p1}) to Target Apoapsis (r_{a2}).
    b. Calculate $a_{t1} = (r_{p1} + r_{a2})/2$.
    c. Calculate Δv_1 and Δv_2.
    d. Calculate $\Delta v_{total,S1}$.
7. **Calculate Scenario 2 (S2)**:
    a. Transfer from Initial Apoapsis (r_{a1}) to Target Periapsis (r_{p2}).
    b. Calculate $a_{t2} = (r_{a1} + r_{p2})/2$.
    c. Calculate Δv_1 and Δv_2.
    d. Calculate $\Delta v_{total,S2}$.

8. **Find Best Transfer**:
    a. Check: Is $\Delta v_{total,S1} \leq \Delta v_{total,S2}$?
    b. **If Yes:** The best transfer is S1.
    c. **If No:** The best transfer is S2.
    d. (If neither is possible, an error is thrown).
9. **Store Results:** Populate the `g_transferData` object with all parameters from the best scenario.
10. **Calculate Rocket & Time:** Use the Rocket Equation and Transfer Time formula.
11. **Update UI:** Set text content for all result fields (e.g., `dom.totalDvVal`).
12. **Draw Visualisation:** Call `updateScaleAndRedraw().`
13. **Bi-elliptic Check:** Perform the advanced bi-elliptic transfer analysis.
14. **End/Catch:** If any error was thrown, the catch block displays it in `dom.warning`.

4.3 Bi-Elliptic Transfer Check

This is an advanced feature that runs after the Hohmann calculation. It checks if a 3-burn bi-elliptic transfer could be more efficient.

1. It first calculates the $\Delta\, v$ for a **bi-parabolic** transfer (the theoretical limit of a bi-elliptic where the intermediate apoapsis $r_b = \infty$).
2. **If** `dv_BiParabolic < dv_Hohmann`**:** A bi-elliptic transfer is guaranteed to be more efficient at some r_b.
3. The code then performs a **bisection search** to find the “breakeven” radius r_b where `getBiEllipticDV(r_b) == dv_Hohmann`.
4. A warning is displayed to the user (`dom.biEllipticWarning`) informing them of the more efficient, but much slower, bi-elliptic option, and the altitude at which it becomes advantageous.

5. Visualisation (UI/SVG)

The visualisation is generated dynamically by manipulating SVG elements.

5.1 Dynamic Scaling (`updateScaleAndRedraw`)

The visualisation must handle orbits of 100km and 100,000,000km on the same screen.

1. It gets the viewport dimensions (`w`, `h`).
2. It finds the largest orbital radius to be drawn from `g_transferData (max_r_km)`.
3. It calculates a `g_scale` (pixels per km) to make this largest orbit fit within 45% of the smallest screen dimension.

   - `max_dim = Math.min(w, h) * 0.45`
   - `g_scale = max_dim / max_r_km`

4. It then calls `drawAllElements()` to redraw everything at the new scale.

5.2 Drawing Orbits (`drawEllipse`)

This function draws an orbit (a vertical ellipse) with the central body at the southern focus (0,0).

1. **Inputs:** r_p (periapsis) and r_a (apoapsis) in km.
2. **Scale:** All values are multiplied by `g_scale` to get pixels.
3. **Semi-major axis (a_{px}):** The vertical radius of the ellipse.
   - $a_{px} = (r_{p,px} + r_{a,px})/2$
4. **Focal distance (c_{px}):** The distance from the ellipse's centre to its focus.
   - $c_{px} = (r_{a,px} - r_{p,px})/2$
5. **Semi-minor axis (b_{px}):** The horizontal radius of the ellipse.
   - $b_{px} = \sqrt{a_{px}^2 - c_{px}^2}$
6. **Centre Offset (cy_{px}):** The SVG ellipse is drawn from its centre `(cx, cy)`. To place the focus at (0,0), the ellipse's centre must be shifted *up* by c_{px}.
   - $cy_{px} = -c_{px}$
7. **Set Attributes:** The function sets the following attributes on the SVG `<ellipse>` element:
   - `cx = 0`
   - `cy = -c_px`
   - `rx = b_px`
   - `ry = a_px`

5.3 Drawing Burn Points (`drawBurnPoint`)

This function draws a circle at the burn location.

1. **Location:** The Y-coordinate is set based on the scaled radius, with a negative sign if it's at apoapsis.
   - `cy_px = isApoapsis ? -r_px : r_px`
2. **Size:** The radius (`r`) of the circle itself is scaled proportionally to the burn's Δv. This provides a clear visual cue for the “size” of the burn.
   - `radius_px = minRadius + (dv / dvScaleFactor)`

6. Function Definitions

This section details the application's “API” and call hierarchy.

6.1 Initialisation & Setup

- `init()`
    - **Description:** The main entry point for the application.
    - **Calls:** `setupUI()`, `updateCentralBody()`, `onWindowResize()`, `calculateAndDraw()`.
    - **Called By:** (Self-invoked at end of script).

- `setupUI()`
    - **Description:** Attaches all event listeners to the DOM elements.
    - **Calls:** `calculateAndDraw()` (on click), `updateCentralBody()` (on change), `onWindowResize()` (on resize), `clearResults()` (on input).
    - **Called By**: `init()`.

6.2 State & Body Management

- `updateCentralBody()`
    - **Description:** Reads the central body dropdown. Updates global physics constants (`CURRENT_GM_M3_S2`, `CURRENT_BODY_RADIUS_KM`). Handles logic for the “Custom” body inputs. Updates the UI display for body info.
    - **Calls:** `clearResults()`.
    - **Called By:** `init()`, `setupUI` (on change listener).

- `clearResults()`
    - **Description:** Resets all result fields to ‘N/A’, hides all warnings, and hides all SVG visual elements. `Sets g_transferData` to null.
    - **Called By:** `updateCentralBody()`, `calculateAndDraw()`, `setupUI` (on input listener).

6.3 Core Calculation

- calculateAndDraw()
    - **Description:** The main engine. Performs the entire validation, calculation, result-storage, and UI-update workflow as described in Section 4.2.
    - **Calls:** clearResults(), getVelocity(), updateScaleAndRedraw(), getBiEllipticDV(), getBiEllipticTime_s().
    - **Called By:** init(), setupUI (on click listener).

6.4 Physics Subroutines

- `getVelocity(r_m, a_m)`
  - **Description:** Implements the Vis-viva equation, also referred to as the orbital-energy-invariance law.
  - **Parameters:** `r_m` (current radius in meters), `a_m` (semi-major axis in meters).
  - **Returns:** (Number) Velocity in m/s.
  - **Called By:** `calculateAndDraw()`.
- `getBiEllipticDV(r_A, v_A_initial, r_D, v_D_final, rb, mu)`
  - **Description:** Calculates the total 3-burn Δv for a bi-elliptic transfer with an intermediate apoapsis of `rb`.
  - **Returns:** (Number) Total Δv in m/s (or `Infinity` if invalid).
  - **Called By:** `calculateAndDraw()` (for the advanced check).
- `getBiEllipticTime_s(r_A, r_D, rb, mu)`
  - **Description:** Calculates the total time for a bi-elliptic transfer.
  - **Returns:** (Number) Total time in seconds.
  - **Called By:** `calculateAndDraw()` (for the advanced check).

6.5 Visualisation & Drawing

- `onWindowResize()`
  - **Description:** Adjusts the SVG canvas size and re-centers the main `<g>` element. Calls for a rescale and redraw.
  - **Calls:** `updateScaleAndRedraw()`.
  - **Called By:** `init()`, `setupUI` (on resize listener).
- `updateScaleAndRedraw()`
  - **Description:** Calculates the new `g_scale` global variable based on the largest orbit radius and current viewport size.
  - **Calls:** `drawAllElements()`.
  - **Called By:** `onWindowResize()`, `calculateAndDraw()`.
- `drawAllElements()`
  - **Description:** The master drawing function. It reads all data from `g_transferData` and applies it to the SVG elements using the current `g_scale`.
  - **Calls:** `drawEllipse()`, `drawBurnPoint()`.
  - **Called By:** `updateScaleAndRedraw()`.
- `drawEllipse(el, r_p_km, r_a_km, scale)`
  - **Description:** Calculates and sets the `cx`, `cy`, `rx`, `ry` attributes for an `<ellipse>` element as described in Section 5.2.
  - **Called By:** `drawAllElements()`.
- `drawBurnPoint(el, r_km, scale, isApoapsis, dv)`
  - **Description:** Sets `cx`, `cy`, and `r` attributes for a `<circle>` element as per Section 5.3.
  - **Called By:** `drawAllElements()`.

7. Standards and Dependencies

- **Standards:** The code adheres to JavaScript ES6 ‘strict mode’.
- **Dependencies:** The application has no external dependencies. It relies only on standard Web APIs (DOM, SVG).