Design & Architecture Report

Lambert Solver

Version: 1.0

Date: 22 November 2025

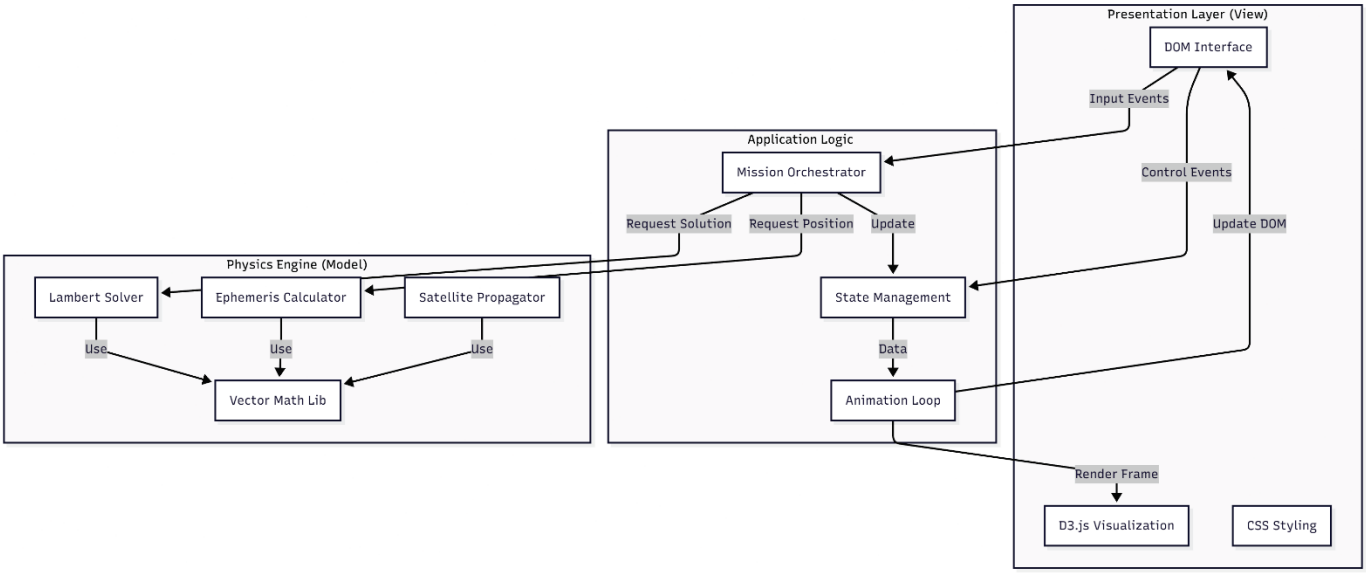
Contents

1. Introduction

This document provides a comprehensive overview for the `LambertSolver.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). The SPA calculates and visualise optimal transfer orbits between inner solar system bodies (Mercury, Venus, Earth, Mars). It uses Keplerian orbital mechanics for planetary ephemerides and a Universal Variable Lambert Solver to determine trajectory solutions. The application features a reactive UI for simulation control and a D3.js-based SVG rendering engine for 2D visualisation of 3D orbital data.

2. High-Level Architecture

The application follows a Model-View-Controller (MVC) variant adapted for a functional JavaScript environment. It is monolithic (contained in one file) but logically stratified into three distinct layers.




```
graph TD
    subgraph "Presentation Layer (View)"
        UI[DOM Interface]
        D3[D3.js Visualization]
        CSS[CSS Styling]
    end

    subgraph "Application Logic (Controller)"
        State[State Management]
        Orchestrator[Mission Orchestrator]
        Loop[Animation Loop]
    end

    subgraph "Physics Engine (Model)"
        Ephemeris[Ephemeris Calculator]
        Lambert[Lambert Solver]
        Propagator[Satellite Propagator]
        MathLib[Vector Math Lib]
    end

    UI -->|Input Events| Orchestrator
    UI -->|Control Events| State
    Orchestrator -->|Request Solution| Lambert
    Orchestrator -->|Request Position| Ephemeris
    Orchestrator -->|Update| State
    State -->|Data| Loop
    Loop -->|Render Frame| D3
    Loop -->|Update DOM| UI
    Lambert -->|Use| MathLib
    Ephemeris -->|Use| MathLib
    Propagator -->|Use| MathLib
```

2.1 Layer Definitions

1. **Physics Engine (Model):** Contains the mathematical core. It stores planetary constants, calculates state vectors (position/velocity) from orbital elements, and solves the boundary-value problem (Lambert's problem).
2. **Application Logic (Controller):** Manages the application state (`simTime`, `speed`, `mission` object), handles user input (calculation requests, animation toggles), and orchestrates the optimisation algorithms.
3. **Presentation Layer (View):** Handles the DOM (HTML inputs/outputs) and the SVG Canvas. It uses D3.js to map astronomical units (au) to screen pixels.

3. Detailed Design: Data Structures & State

The application relies on several global data structures to maintain physics constants and runtime state.

3.1 Physics Constants

The `CONSTANTS` object defines the immutable laws of the simulation.

- **au:** 1.496×10^8 km.
- **μ (MU):** 1.327×10^{11} km³/s² (Sun's Standard Gravitational Parameter).
- **J2000:** The reference epoch (Jan 1, 2000, 12:00 TT) used for orbital element propagation.

3.2 Planetary Data (`BODIES`)

Each planet is defined by its Mean Orbital Elements with respect to the J2000 Ecliptic and their centennial rates of change.

- Structure:
  - `els`: Base elements ($a, e, i, L, \varpi, \Omega$) at J2000.
  - `rates`: Change per Julian Century.
  - `r`: Visualisation radius.
  - `color`: Hex code for rendering.

3.3 Runtime State (`State`)

This object serves as the "Source of Truth" for the simulation.

- `simTime` (Date): Current simulation timestamp.
- `speed` (Number): Time multiplier (e.g., 100,000x real-time).
- `mission` (Object): Stores the computed solution (Departure/Arrival dates, Δv vectors, path points).
- `svg`: Stores viewport dimensions and current scaling factor.

4. Astrodynamics Engine: Logic and Mathematics

This section details the mathematical models implemented in the `getBodyState`, `solveLambert`, and `getSatState` functions.

4.1 Ephemeris Calculation (`getBodyState`)

This function calculates the position (r) and velocity (v) of a planet at a specific time.

Logic Flow:

1. **Time Calculation:** Calculate T, the number of Julian centuries since J2000.
2. **Element Propagation:** Apply linear rates: $Element_{current} = Element_{base} + (Rate \times T)$.
3. **Kepler's Equation**: Solve for Eccentric Anomaly (E) using Newton-Raphson iteration:

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

   Iteration: $E_{n+1} = E_n - \frac{E_n - e\sin(E_n) - M}{1 - e\cos(E_n)}$

4. **Perifocal Coordinates**: Calculate position/velocity in the 2D orbital plane (PQW frame).

$$r = a(1 - e\cos E)$$
$$x_p = r\cos\nu\,, \qquad y_p = r\sin\nu$$

5. **3D Rotation (ECI):** Rotate the Perifocal vectors to the Heliocentric Ecliptic frame using 3D rotation matrices derived from the Longitude of Ascending Node (Ω), Argument of Periapsis (ω), and Inclination (i).

4.2 Lambert Solver (`solveLambert`)

This function solves the boundary-value problem: finding the trajectory between two vectors r_1 and r_2 in time Δt. It uses the Universal Variable Formulation.

Mathematical Model:

1. **Transfer Angle:** Calculated via dot product $r_1 \cdot r_2$. Corrected for prograde/retrograde motion using the cross-product z-component.
2. **Universal Variables**: Uses Stumpff functions $C(z)$ and $S(z)$ to handle elliptical, parabolic, and hyperbolic orbits uniformly.

$$C(z) = \frac{1 - \cos\sqrt{z}}{z}, \qquad S(z) = \frac{\sqrt{z} - \sin\sqrt{z}}{\sqrt{z^3}}$$

3. Time of Flight Iteration: Uses a numerical solver (Secant/Newton method) to find the auxiliary variable z (related to the semi-major axis) that satisfies the flight time equation:

$$\sqrt{\mu}\Delta t = x^3 S(z) + A\sqrt{y}$$

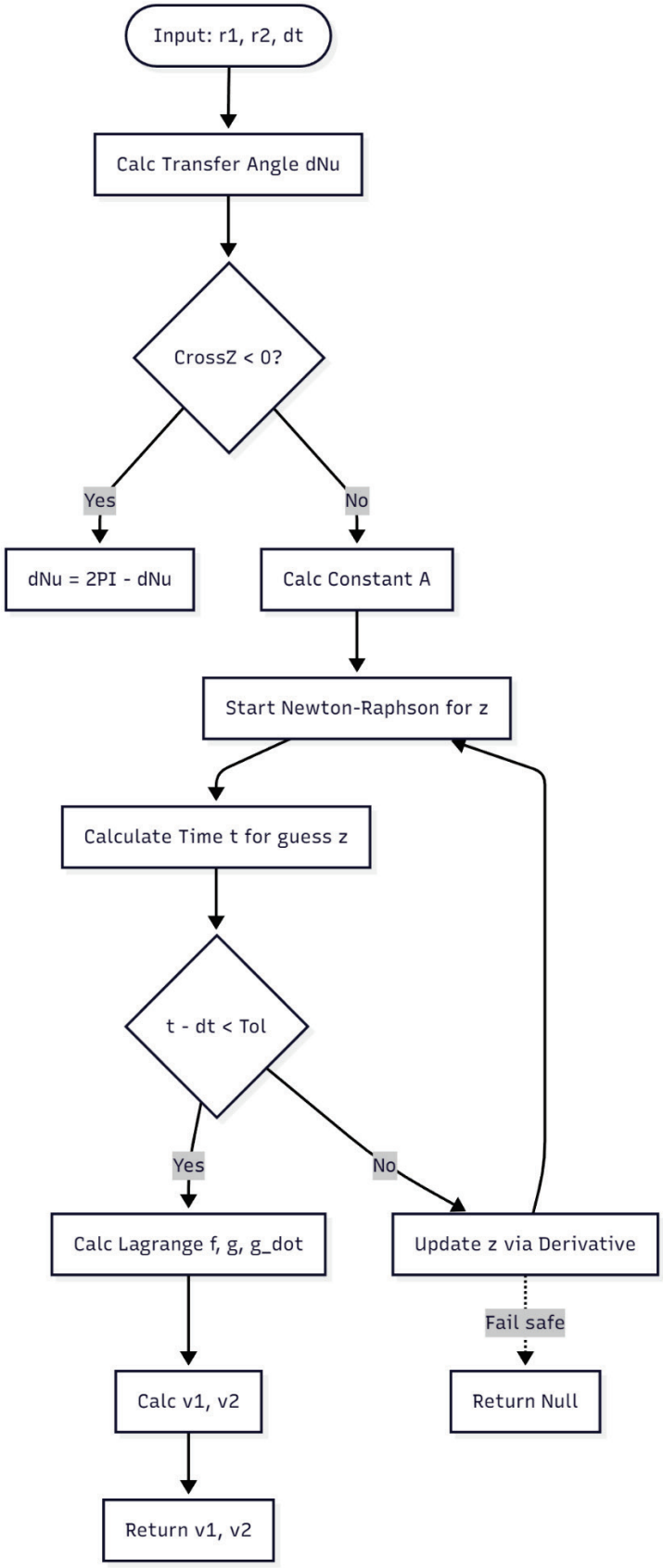
   where y and x are geometric parameters derived from r_1, r_2, A, z.

4. Velocity Reconstruction: Once z converges, Lagrange coefficients ($f, g, \dot{g}$) are computed to find

$$v_1 = \frac{\overrightarrow{r_2} - f\,\overrightarrow{r_1}}{g}, \ \overrightarrow{v_2} = \frac{\dot{g}\,\overrightarrow{r_2} - \overrightarrow{r_1}}{g}.$$

Flowchart of Lambert Solver Logic

```
flowchart TD
    Start([Input: r1, r2, dt]) --> Geometry[Calc Transfer Angle dNu]
    Geometry --> Prograde Check{CrossZ < 0?}
    Prograde Check -- Yes --> AdjustAngle[dNu = 2PI - dNu]
    Prograde Check -- No --> CalcConst[Calc Constant A]
    CalcConst --> Iteration[Start Newton-Raphson for z]
    Iteration --> CalcT[Calculate Time t for guess z]
    CalcT --> Converged{t - dt < Tol}
    Converged -- No --> UpdateZ[Update z via Derivative]
    UpdateZ --> Iteration
    Converged -- Yes --> Lagrange[Calc Lagrange f, g, g_dot]
    Lagrange --> Velocity[Calc v1, v2]
    Velocity --> Return[Return v1, v2]

    UpdateZ -.->|Fail safe| Divergence[Return Null]
```


Input: r1, r2, dt
Calc Transfer Angle dNu
CrossZ < 0?
Yes
No
dNu = 2PI - dNu
Calc Constant A
Start Newton-Raphson for z
Calculate Time t for guess z
t - dt < Tol
Yes
No
Calc Lagrange f, g, g_dot
Update z via Derivative
Fail safe
Calc v1, v2
Return Null
Return v1, v2

4.3 Satellite Propagation (`getSatState`)

Used to draw the transfer arc and animate the satellite. It reverses the ephemeris process:

1. **State to Elements:** Converts $r, \vec{v}$ into orbital elements $(a, e, i, \Omega, \omega, M_0)$.
2. **Propagate Mean Anomaly:** $M(t) = M_0 + n \cdot dt$.
3. **Solve Kepler:** Finds new E.
4. **Elements to State:** Converts back to Cartesian coordinates.

5. Logic & Optimisation Design

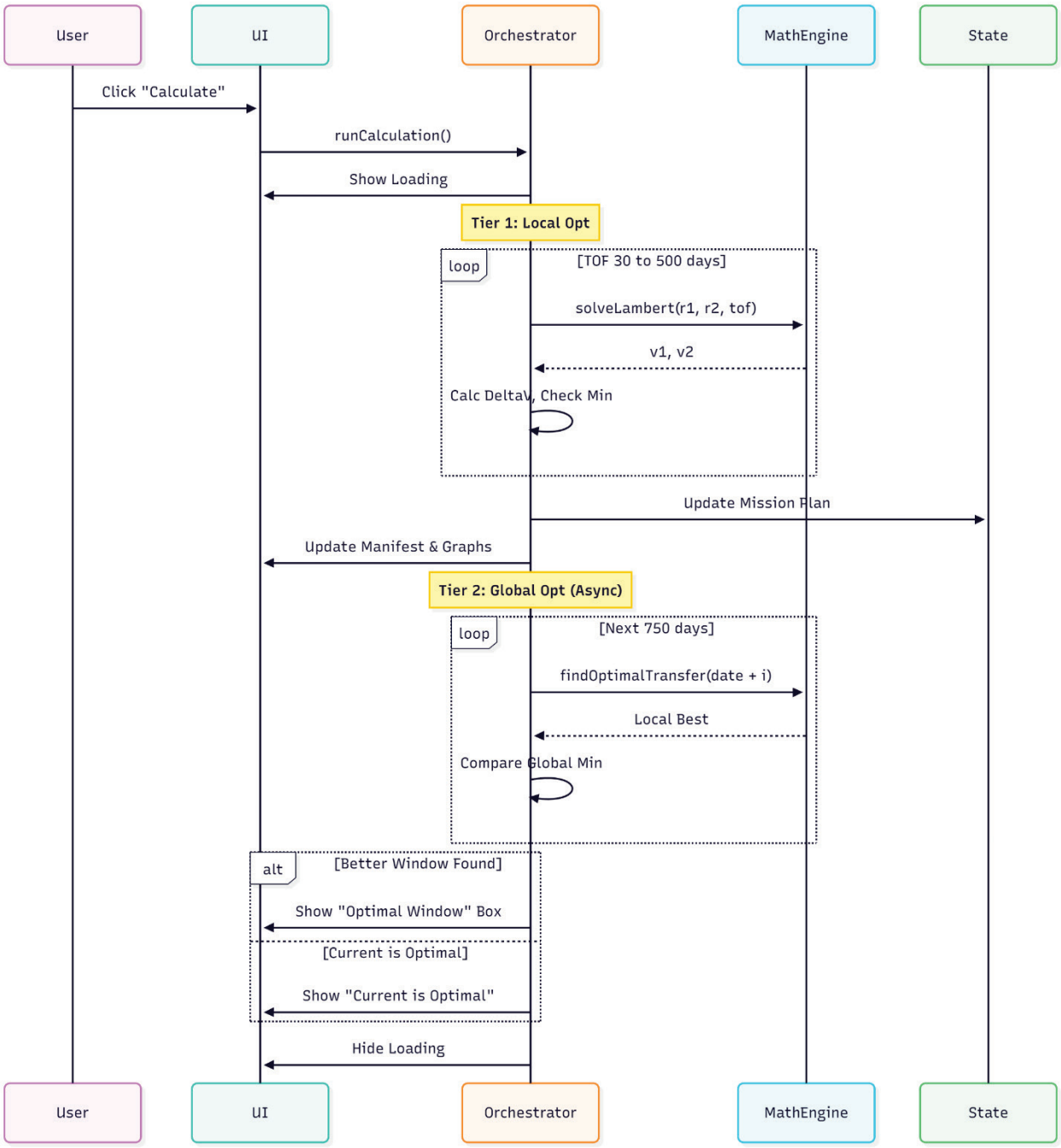
The application uses a two-tiered optimisation strategy to find the minimum Δv launch window.

5.1 Optimisation Hierarchy

1. **Local Optimisation** (`findOptimalTransfer`):
   - **Input:** Origin Body, Target Body, Specific Date.
   - **Process:** Iterates through flight durations (Time of Flight - TOF) from 30 to 500 days in 10-day steps.
   - **Cost Function:** Total $\Delta v = \left|\left|\overrightarrow{v_{dep}} - \overrightarrow{v_{planet1}}\right|\right| + \left|\left|\overrightarrow{v_{arr}} - \overrightarrow{v_{planet2}}\right|\right|$.
   - **Result:** The TOF that minimizes Δv for that specific departure date.
2. **Global Optimisation** (`runCalculation`):
   - **Input:** User selected date.
   - **Process:** Scans 750 days into the future (approx. one synodic period for Mars/Earth) in 30-day increments. It calls `findOptimalTransfer` for each increment.
   - **Result:** Suggests a "Better Window" to the user if a significantly lower Δv exists in the future.

5.2 Orchestration Sequence Diagram

```
sequenceDiagram
    participant User
    participant UI
    participant Orchestrator
    participant MathEngine
    participant State

    User->>UI: Click "Calculate"
    UI->>Orchestrator: runCalculation()
    Orchestrator->>UI: Show Loading

    Note over Orchestrator: Tier 1: Local Opt
    loop TOF 30 to 500 days
        Orchestrator->>MathEngine: solveLambert(r1, r2, tof)
        MathEngine-->>Orchestrator: v1, v2
        Orchestrator->>Orchestrator: Calc DeltaV, Check Min
    end

    Orchestrator->>State: Update Mission Plan
    Orchestrator->>UI: Update Manifest & Graphs

    Note over Orchestrator: Tier 2: Global Opt (Async)
    loop Next 750 days
        Orchestrator->>MathEngine: findOptimalTransfer(date + i)
        MathEngine-->>Orchestrator: Local Best
        Orchestrator->>Orchestrator: Compare Global Min
    end

    alt Better Window Found
        Orchestrator->>UI: Show "Optimal Window" Box
    else Current is Optimal
        Orchestrator->>UI: Show "Current is Optimal"
    end
    Orchestrator->>UI: Hide Loading
```


User
UI
Orchestrator
MathEngine
State
Click "Calculate"
runCalculation()
Show Loading
Tier 1: Local Opt
loop
[TOF 30 to 500 days]
solveLambert(r1, r2, tof)
v1, v2
Calc DeltaV, Check Min
Update Mission Plan
Update Manifest & Graphs
Tier 2: Global Opt (Async)
loop
[Next 750 days]
findOptimalTransfer(date + i)
Local Best
Compare Global Min
alt
[Better Window Found]
Show "Optimal Window" Box
[Current is Optimal]
Show "Current is Optimal"
Hide Loading
User
UI
Orchestrator
MathEngine
State

6. Visualisation & Rendering

The visualisation is handled by `renderFrame`, driven by `requestAnimationFrame`. It uses a dynamic scaling system to ensure the relevant orbits fill the screen.

6.1 Scaling Logic

- **Default:** Scales to fit Mars' orbit (~1.7 au).
- **Mission Active**: Scans the mission path points. Finds the maximum distance from the sun ($maxPath$) and scales the SVG transform to fit that distance with 10% padding.

$$Scale = \frac{\min(width, height)/2}{1.1 \max au}.$$

6.2 Rendering Pipeline

1. **Clear Canvas:** Remove previous SVG elements.
2. **Draw Static Elements:** Sun, Planetary Orbits (traced by propagating 100 points over one period).
3. **Draw Dynamic Elements:** Planets (at `simTime`). Labels are scaled inversely ($12/scale$) so they remain readable regardless of zoom.
4. **Draw Mission**:
    - Transfer trajectory (dashed line).
    - Satellite Marker:
        - If $t < t_{dep}$: Hidden.
        - If $t_{dep} \leq t < t_{arr}$: Calculated via `getSatState`.
        - If $t \geq t_{arr}$: Locked to Target Planet coordinates (rendezvous).

7. Interface Definitions

7.1 Core Functions

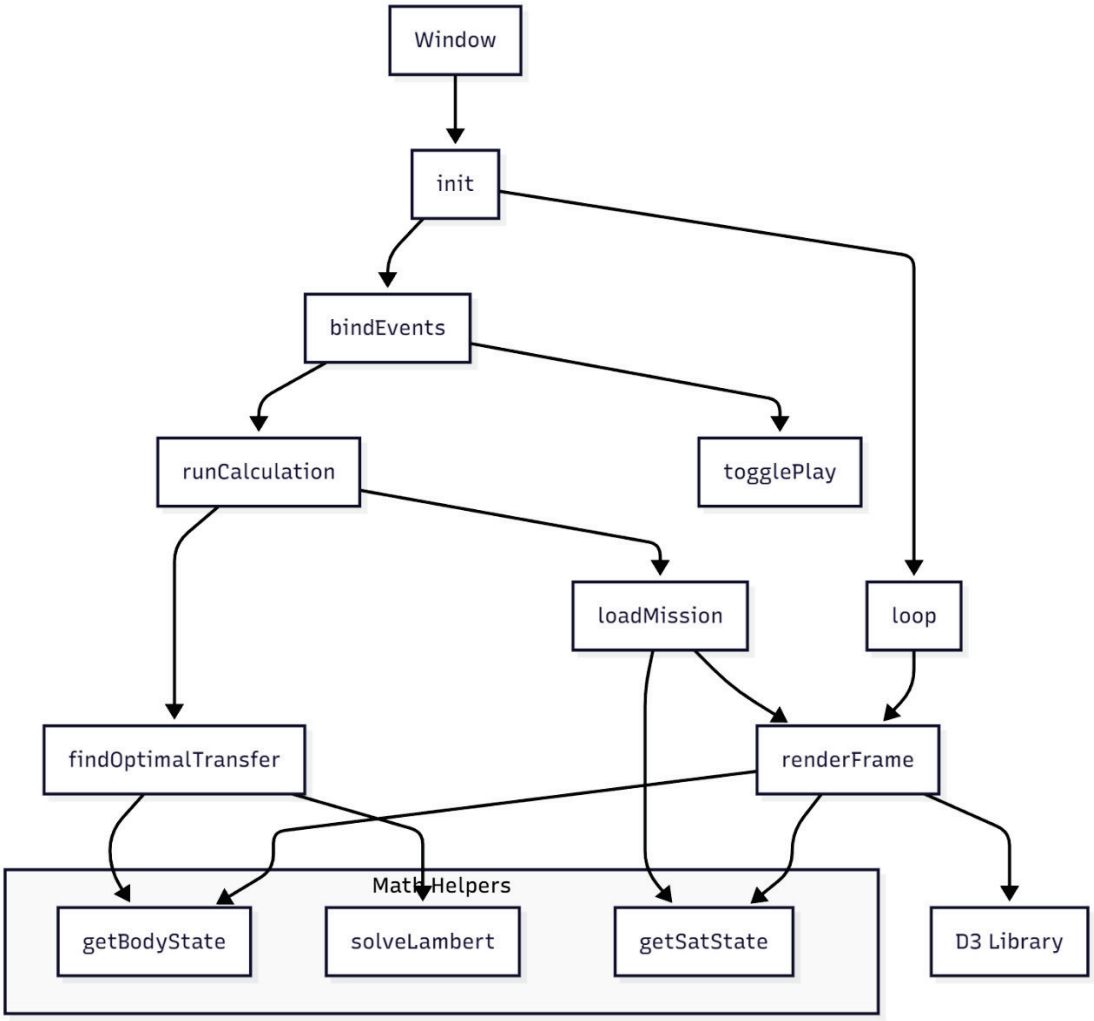
Function Name	Scope	Inputs	Outputs	Description
`getBodyState`	Engine	`bodyKey` (string), `date` (Date)	`{r: [x,y,z], v: [x,y,z]}`	Calculates planet ephemeris.
`solveLambert`	Engine	`r1`, `r2` (Vectors), `dt` (sec)	`{v1, v2}` or `null`	Solves the orbital boundary value problem.
`getSatState`	Engine	`r0`, `v0` (Vectors), `dt` (sec)	`[x,y,z]`	Propagates satellite position.
`findOptimalTransfer`	Logic	`origin`, `target`, `date`	`Object` (Mission Data)	Finds best TOF for a specific date.
`runCalculation`	Logic	None (Reads DOM)	Void	Orchestrates calculation and UI updates.
`renderFrame`	View	None	Void	Main D3.js drawing routine.

7.2 Dependencies

- **D3.js (v7):** Used for SVG creation, path generation (`d3.line`), and selection manipulation.
- **Google Fonts:** Loads 'Anta' and 'Roboto Mono'.

8. Class/Function Hierarchy Diagram

Although functional, the code structure implies a hierarchy of dependency.




```
graph TD
    Window --> Init[init]
    Init --> Loop[loop]
    Init --> Events[bindEvents]
    Events --> RunCalc[runCalculation]
    Events --> TogglePlay[togglePlay]
    RunCalc --> FindOpt[findOptimalTransfer]
    RunCalc --> LoadMission[loadMission]
    FindOpt --> GetBody[getBodyState]
    FindOpt --> Lambert[solveLambert]
    LoadMission --> GetSat[getSatState]
    LoadMission --> Render[renderFrame]
    Loop --> Render
    Render --> GetBody
    Render --> GetSat
    Render --> D3[D3 Library]
    subgraph "Math Helpers"
        GetBody
        Lambert
        GetSat
    end
```

9. Results Generation & Data Flow

How the displayed results are generated:

1. **C3 (Characteristic Energy)**: Derived from the magnitude of the hyperbolic excess velocity at departure.

$$C_3 = v_\infty^2 = \left|\left|\overrightarrow{v_{transfer}} - \overrightarrow{v_{planet}}\right|\right|^2$$

2. **Hyperbolic Excess Velocity (V_∞)**:

$$V_\infty = \sqrt{C_3}$$

3. **Angles (Declination/Right Ascension):** Calculated from the V_∞ vector components relative to the Ecliptic J2000 frame.
   - $\text{Declination} = \sin^{-1}(v_z/|v|)$
   - $\text{Right Ascension} = \tan^{-1}(v_y \quad v_x)$
4. **Total Delta-V**: The scalar sum of the departure burn and the arrival capture burn

$$\Delta v_{total} = \left|\left|\overrightarrow{v_{dep}} - \overrightarrow{v_{p1}}\right|\right| + \left|\left|\overrightarrow{v_{arr}} - \overrightarrow{v_{p2}}\right|\right|.$$