Satellites: A Terminal-Based Satellite Tracker

Architecture, Orbital Mechanics, and Real-Time Visualisation

Simon Knight

Adelaide, Australia

March 2026 • Version 0.1.0

Last updated: March 12, 2026

Abstract. Satellite tracking applications typically require heavyweight graphical interfaces and external propagation services. We present *Satellites*, a terminal-based satellite tracker implemented in Rust that provides real-time position tracking, pass prediction, and Doppler-corrected frequency control—all within an interactive TUI. The system employs a dual-tick architecture that decouples expensive SGP4 orbital propagation (1 Hz) from smooth rendering (30 FPS) via linear interpolation, reducing CPU usage by approximately 90% compared to per-frame propagation. A five-stage coordinate transform pipeline with leap second correction converts SGP4 output to observer-relative look angles with sub-arcsecond precision. Pass prediction uses a coarse-step scan with bisection refinement to locate acquisition and loss of signal times. The TUI provides flat and orthographic globe projections, a polar sky view, pass timelines, and Braille-glyph satellite icons—all rendered with Unicode block characters in a standard terminal. At ~10,800 lines of Rust the system tracks 1,000+ satellites interactively on commodity hardware.

Contents

1 Introduction **1**

2 Background & Related Work **1**
- 2.1 Desktop Tracking Applications . . . 1
- 2.2 Web and API-Based Trackers . . . 2
- 2.3 Rust Ecosystem for Orbital Mechanics . . . 2

3 Architecture **2**
- 3.1 Dual-Tick Event Architecture . . . 2
- 3.2 Data Pipeline . . . 3
- 3.3 State Machine . . . 4

4 Coordinate Transform Pipeline **4**
- 4.1 Unit Conversion . . . 4
- 4.2 ECI to ECEF via Greenwich Sidereal Time . . . 4
- 4.3 Leap Second Correction . . . 4
- 4.4 ECEF to Geodetic (WGS84) . . . 5
- 4.5 Geodetic to AER (Look Angles) . . . 6

5 Pass Prediction **6**
- 5.1 Algorithm Overview . . . 6
- 5.2 In-Progress Pass Detection . . . 7
- 5.3 Parallelisation . . . 7

6 Rendering **8**
- 6.1 Position Interpolation . . . 8
- 6.2 Orthographic Globe Projection . . . 9
- 6.3 Braille Dot-Matrix Icons . . . 10
- 6.4 Day/Night Terminator . . . 10
- 6.5 Satellite Footprint . . . 10

7 Radio Integration **10**
- 7.1 Doppler Shift Calculation . . . 11
- 7.2 Hamlib Protocol . . . 11

8 Configuration **11**
- 8.1 TOML Schema . . . 11
- 8.2 CLI Precedence . . . 11
- 8.3 Theme System . . . 12

9 WebAssembly Target **13**
- 9.1 Dual Crate Type . . . 13
- 9.2 Platform Gating . . . 13
- 9.3 TrackerCore API . . . 13
- 9.4 Web Frontend . . . 14

10 Testing **14**
- 10.1 SGP4 Regression Tests . . . 14
- 10.2 Unit Test Coverage . . . 15
- 10.3 Test Strategy . . . 15

11 Performance Characteristics **16**
- 11.1 CPU and Memory . . . 16
- 11.2 Scaling Behaviour . . . 16
- 11.3 Network Usage . . . 16
- 11.4 Codebase Size . . . 17

12 Limitations & Future Work 17

13 Conclusion 18

1 Introduction

Satellite tracking underpins amateur radio, space-weather monitoring, and educational outreach. Traditional tracking applications such as Gpredict and SatPC32 require desktop GUI toolkits, making them unsuitable for headless servers, SSH sessions, and lightweight embedded systems. Web-based alternatives depend on external propagation APIs and incur network latency that precludes real-time Doppler correction.

Satellites is a terminal-based satellite tracker implemented in Rust that performs all orbital propagation, coordinate transformation, and pass prediction locally. The application presents an interactive TUI built on the `ratatui` framework, providing five distinct views—world map, satellite detail, pass timeline, polar sky view, and station dashboard—within a standard ANSI terminal.

This report makes the following contributions:

1. A description of the dual-tick event architecture that decouples propagation from rendering (section 3).
2. A formalisation of the five-stage coordinate transform pipeline from SGP4 ECI output to observer-relative AER look angles, including leap second correction (section 4).
3. An analysis of the pass prediction algorithm, its coarse-scan and bisection-refinement strategy, and its handling of in-progress passes (section 5).
4. A description of the rendering techniques—orthographic globe projection, Braille dot-matrix icons, antimeridian-safe longitude interpolation, and day/night terminator shading (section 6).
5. A discussion of the Hamlib integration for automated Doppler-corrected radio control (section 7).
6. A description of the layered configuration system, CLI precedence rules, and 28-colour theme engine (section 8).
7. An analysis of the WebAssembly target architecture, platform gating strategy, and the `TrackerCore` JavaScript API (section 9).
8. A summary of the test suite covering SGP4 regression, coordinate transforms, and data pipeline robustness (section 10).

The remainder of this report is structured as follows. Section 2 surveys related work. Section 3 presents the system architecture. Section 4 describes the coordinate pipeline. Section 5 covers pass prediction. Section 6 details the rendering subsystem. Section 7 discusses radio integration. Section 8 describes the configuration system. Section 9 covers the WebAssembly target. Section 10 presents the test suite. Section 11 evaluates performance. Section 12 discusses limitations, and section 13 concludes.

2 Background & Related Work

Satellite tracking applications have existed since the early days of amateur radio satellite communications. The fundamental propagation model used by nearly all such tools is the Simplified General Perturbations model, SGP4, developed by the U.S. Air Force in the 1960s and standardised by Hoots and Roehrich [1]. SGP4 accepts Two-Line Element (TLE) sets—compact orbital parameter descriptions distributed by CelesTrak and Space-Track—and produces position and velocity vectors in the True Equator Mean Equinox (TEME) inertial reference frame.

2.1 Desktop Tracking Applications

Gpredict [2] is a widely used open-source satellite tracker built on GTK. It provides map views, polar plots, and pass predictions, and integrates with Hamlib for antenna rotator and radio control. SatPC32 is a Windows-only alternative popular in the amateur radio community. Both require graphical desktop environments and are unsuitable for headless or remote operation.

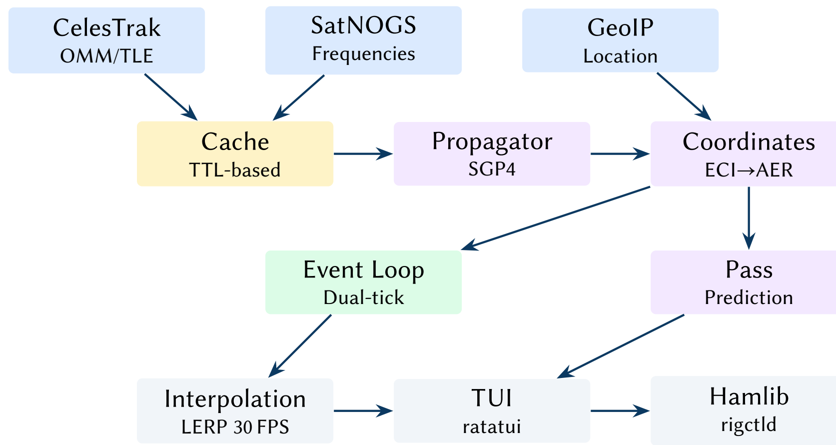

Figure 1. Module architecture of *Satellites*. Blue nodes are external data sources, yellow is caching, purple is orbital mechanics, green is the event loop, and grey is the rendering layer.

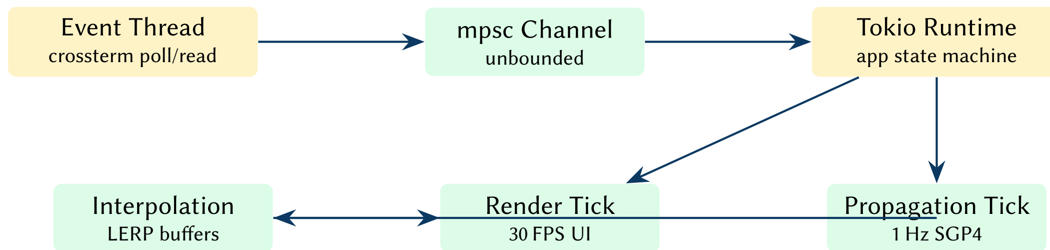

Figure 2. Dual-tick and thread architecture. A blocking event thread feeds an async runtime through a channel; propagation updates are decoupled from rendering via interpolation.

2.2 Web and API-Based Trackers

N2YO and SatNOGS [3] provide web-based tracking interfaces with server-side propagation. These approaches trade local computation for network dependency, introducing latency that makes real-time Doppler correction impractical. SatNOGS additionally operates a distributed ground station network and maintains a comprehensive transmitter database that our system queries for frequency data.

> **Design Decision 1: Why a TUI?**
>
> A terminal-based interface runs anywhere an SSH session exists: headless Raspberry Pi ground stations, remote contest stations, cloud VMs. The Braille dot-matrix rendering achieves sub-cell positioning (2×4 dots per character cell), providing visual fidelity approaching that of pixel-based GUIs while requiring zero graphical dependencies.

2.3 Rust Ecosystem for Orbital Mechanics

The Rust crate `sgp4` [4] provides an SGP4/SDP4 implementation with TLE validation. The `map_3d` crate offers coordinate transform functions (ECI→ECEF→geodetic→AER) based on the `map_3d` MATLAB library. The `ratatui` framework [5] provides immediate-mode TUI rendering with canvas, table, and chart widgets. *Satellites* builds on these crates, adding the dual-tick interpolation layer, pass prediction, and multi-view TUI that none of them provides individually.

3 Architecture

Satellites is organised into five subsystems: *data* (API clients and caching), *tracking* (propagation, coordinates, passes), *location* (observer geolocation), *interaction* (input state machines), and *ui* (rendering). Figure 1 shows the module structure.

3.1 Dual-Tick Event Architecture

The central design decision is the *dual-tick* event loop. SGP4 propagation for N satellites is computationally expensive—each evaluation involves Keplerian orbit integration with perturbation

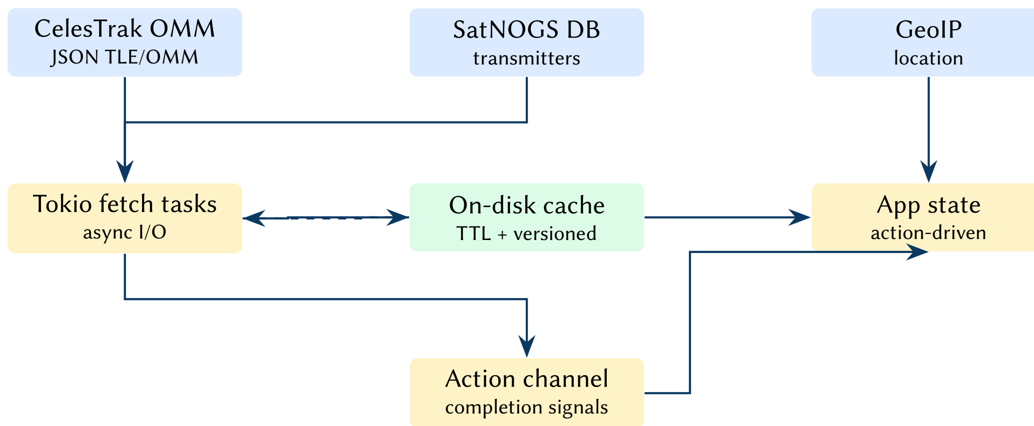

Figure 3. Data pipeline and caching lifecycle. Background fetch tasks populate a TTL-based on-disk cache and notify the app via actions so the UI remains responsive.

corrections. A naive approach that propagates every satellite on every frame would limit either frame rate or satellite count.

Satellites separates propagation from rendering with two independent tick rates:

1. **Propagation tick (1 Hz)** — runs full SGP4 evaluation for all tracked satellites, producing a position snapshot.
2. **Render tick (30 FPS)** — interpolates between the two most recent propagation snapshots and redraws the TUI.

Listing 1. Event type enumeration showing the dual-tick events.

```
pub enum Event {
    /// Propagation tick (1 Hz - triggers SGP4 position updates)
    Tick,
    /// Render tick (30 FPS - triggers UI redraws)
    Render,
    Key(KeyEvent),
    Mouse(MouseEvent),
    Resize(u16, u16),
    Action(Action),
}
```

The propagation tick runs on a dedicated OS thread (via `std::thread::spawn`) that polls `crossterm` events and emits both tick types through a Tokio `mpsc::UnboundedChannel`. The main Tokio runtime consumes these events and drives state transitions in the application state machine.

> **Thread Architecture**
>
> The event handler runs on a bare OS thread rather than a Tokio task because `crossterm`'s `poll`/`read` functions are blocking and must not share a thread with the async runtime. The `mpsc` channel bridges the synchronous event thread to the async application loop.

3.2 Data Pipeline

Satellite orbital elements are fetched from CelesTrak's OMM JSON API and cached locally with a configurable TTL (default 24 hours). Transmitter frequency data is fetched from the SatNOGS database API and cached separately (default 48 hours). The cache uses a versioned file-based strategy stored under `~/.cache/satellites/`, resolved via the `dirs` crate for cross-platform compatibility.

Background data fetches are dispatched as Tokio tasks and do not block the event loop. Completion is signalled via the action channel, allowing the UI to remain responsive during network operations.

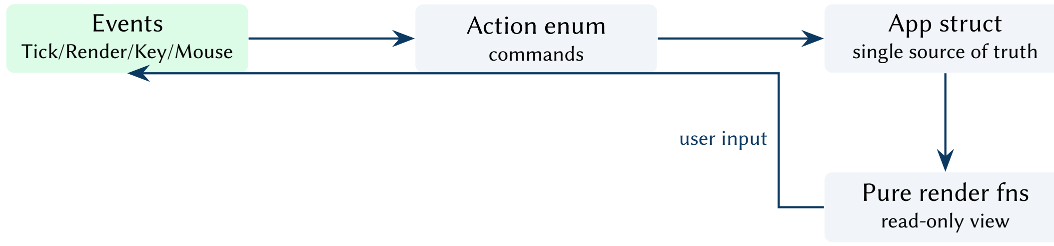

Figure 4. State transition discipline. All mutations flow through actions; UI rendering is a pure function of state, keeping behavior testable and traceable.

Figure 5. The five-stage coordinate transform pipeline from SGP4 output to observer-relative look angles.

3.3 State Machine

Application state is centralised in a single `App` struct (~1,900 lines) that holds satellite data, observer position, view mode, selection state, and interpolation buffers. The UI rendering functions are pure functions of this state—they read `App` fields but never mutate them. All mutations flow through a command-pattern `Action` enum, ensuring that state transitions are traceable and testable.

4 Coordinate Transform Pipeline

SGP4 produces position vectors in the True Equator Mean Equinox (TEME) Earth-centred inertial (ECI) reference frame. Converting these to quantities useful for tracking—latitude, longitude, altitude, and observer-relative azimuth, elevation, and range—requires a multi-stage transform pipeline. Figure 5 illustrates the five stages.

4.1 Unit Conversion

SGP4 outputs positions in kilometres. The `map_3d` crate expects SI metres. A factor-of-1000 conversion is applied first:

$$\vec{r}_{\mathrm{m}} = 1000 \cdot \vec{r}_{\mathrm{km}} \tag{1}$$

Unit Mismatch

Omitting this conversion produces geodetic altitudes that are three orders of magnitude too small, placing LEO satellites at ground level. This is a common source of bugs when integrating SGP4 with geographic libraries.

4.2 ECI to ECEF via Greenwich Sidereal Time

The ECI frame is inertial: it does not rotate with the Earth. To obtain Earth-fixed coordinates, we rotate by the Greenwich Sidereal Time (GST), which encodes the current rotational phase of the Earth relative to the vernal equinox:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{\mathrm{ECEF}} = R_z(-\theta_{\mathrm{GST}}) \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{\mathrm{ECI}} \tag{2}$$

where $R_z(\theta)$ is a rotation matrix about the z-axis.

4.3 Leap Second Correction

GST calculation requires precise UTC timing. The difference between UTC and UT1 (astronomical time based on Earth's actual rotation) is managed through leap seconds. As of 2026, the accumulated offset is 37 seconds (unchanged since 2017-01-01).

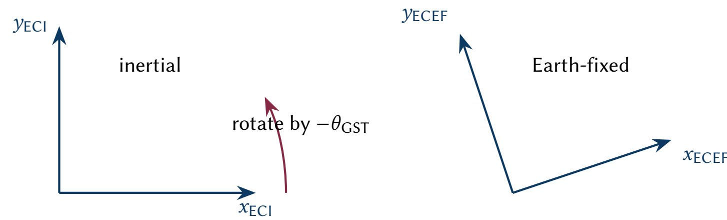

Figure 6. Conceptual ECI→ECEF conversion. The inertial frame is rotated about the Earth's z-axis by the Greenwich Sidereal Time angle to obtain Earth-fixed coordinates.

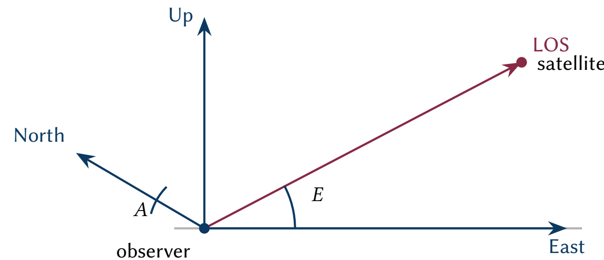

Figure 7. Local AER geometry (schematic). AER expresses the line-of-sight vector in observer-local coordinates: azimuth A in the horizon plane and elevation E above the horizon.

Listing 2. Leap second correction applied before GST calculation.

```rust
/// Current leap second offset as of 2026.
pub const LEAP_SECONDS: f64 = 37.0;

pub fn eci_to_geodetic(
    eci_km: [f64; 3],
    time: DateTime<Utc>,
) -> Result<(f64, f64, f64)> {
    let x_m = eci_km[0] * 1000.0;
    let y_m = eci_km[1] * 1000.0;
    let z_m = eci_km[2] * 1000.0;

    // Apply leap second correction
    let corrected = time.second() as f64 + LEAP_SECONDS;
    let second_int = corrected as i32 % 60;

    let gst = utc2gst([
        time.year(), time.month() as i32,
        time.day() as i32, time.hour() as i32,
        time.minute() as i32, second_int,
    ]);
    // ... ECEF and geodetic conversion
}
```

Design Decision 2: Why Not Use a Leap Second Table?

A hardcoded constant is simpler and sufficient because the IERS announces leap seconds at least six months in advance, and TLE-based tracking already assumes a timely data update cadence. The constant must be updated when the IERS announces the next leap second. Omitting the correction causes a longitude error that grows at approximately 15 arcseconds per missed leap second—small but measurable in high-precision pass timing.

4.4 ECEF to Geodetic (WGS84)

ECEF Cartesian coordinates are converted to geodetic latitude (φ), longitude (λ), and altitude (h) using the WGS84 reference ellipsoid via `map_3d`'s `ecef2geodetic` function. The WGS84 ellipsoid has semi-major axis $a = 6{,}378.137$ km and flattening $f = 1/298.257223563$.

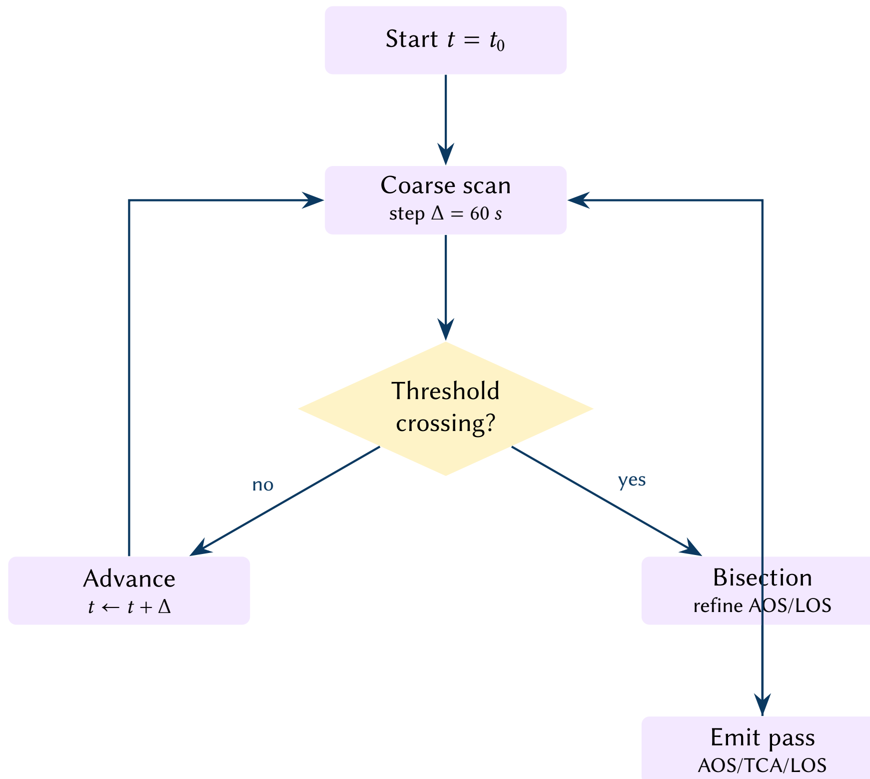

Figure 8. Pass prediction control flow: a coarse scan detects elevation threshold crossings; bisection refinement recovers sub-second AOS/LOS times.

4.5 Geodetic to AER (Look Angles)

For a ground observer at geodetic coordinates $(\varphi_0, \lambda_0, h_0)$, the satellite's azimuth (A), elevation (E), and range (R) are computed using the `geodetic2aer` function from `map_3d`. These AER look angles are the quantities directly useful for antenna pointing:

- **Azimuth** — compass bearing from the observer to the satellite, measured clockwise from north (0°–360°).
- **Elevation** — angle above the horizon (0° = horizon, 90° = zenith).
- **Range** — straight-line distance from observer to satellite in kilometres.

A satellite is considered *visible* when its elevation exceeds 10°, avoiding terrain and atmospheric obstructions near the horizon.

5 Pass Prediction

A *pass* is the interval during which a satellite is above the observer's elevation threshold. Pass prediction computes the Acquisition of Signal (AOS) time, Loss of Signal (LOS) time, and the maximum elevation (TCA—Time of Closest Approach) for each future pass within a configurable window (default 24 hours).

5.1 Algorithm Overview

The prediction algorithm operates in two phases:

1. **Coarse scan** — step through the prediction window at 60-second intervals, evaluating satellite elevation at each step. Detect threshold crossings: a transition from below to above the elevation threshold indicates an approximate AOS; above to below indicates an approximate LOS.
2. **Bisection refinement** — for each detected crossing, apply bisection search between the bounding timestamps to locate the exact crossing time to sub-second precision.

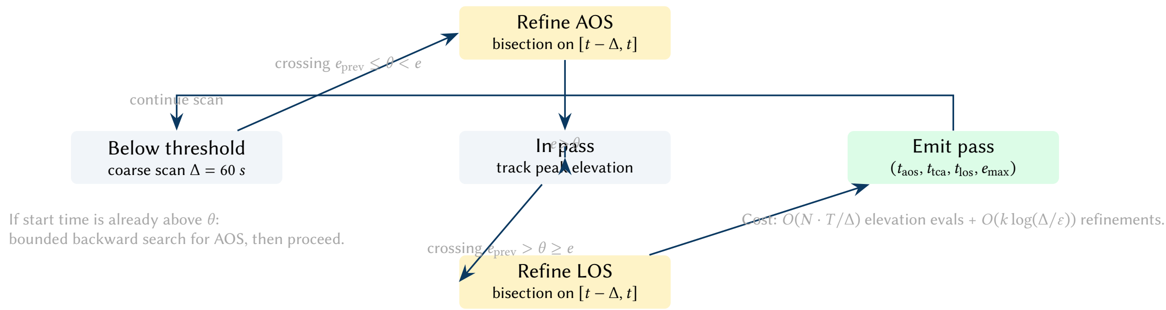

Figure 9. Pass prediction as a state machine. This view highlights the two refinement boundaries (AOS/LOS), the in-progress pass case (backward search), and the cost drivers.

Algorithm 1 Pass prediction with bisection refinement.

Require: Satellite constants C, observer O, window $[t_0, t_1]$, threshold θ, step size $\Delta = 60$ s
Ensure: List of passes P

```
 1: procedure PredictPasses(C, O, t0, t1, θ)
 2:     P ← []
 3:     t ← t0;   e_prev ← Elevation(C, O, t)
 4:     while t < t1 do
 5:         t ← t + Δ
 6:         e ← Elevation(C, O, t)
 7:         if e_prev ≤ θ < e then                          ▷ AOS crossing detected
 8:             t_aos ← Bisect(C, O, t − Δ, t, θ)
 9:             Scan forward to find LOS crossing
10:             t_los ← Bisect(C, O, ...)
11:             t_tca, e_max ← peak elevation in [t_aos, t_los]
12:             Append pass (t_aos, t_tca, t_los, e_max) to P
13:         end if
14:         e_prev ← e
15:     end while
16:     return P
17: end procedure
```

5.2 In-Progress Pass Detection

When the prediction starts and the satellite is already above the threshold, the algorithm searches *backward* (up to 30 minutes) to find the AOS time. This ensures that the current pass is correctly identified and its full duration is reported.

Listing 3. In-progress pass detection via backward search.

```rust
// Already in a pass: search backward for AOS
if elo > threshold_deg {
    let aos = find_aos_for_in_progress_pass(
        constants, epoch_utc, observer,
        sat_name, norad_id,
        start, threshold_deg,
    )?;
    // ... continue to find LOS forward
}
```

5.3 Parallelisation

Pass prediction is embarrassingly parallel across satellites. The system uses the `rayon` crate to distribute per-satellite predictions across all available CPU cores. For N satellites over a 24-hour window with 60-second steps, the total work is $O(N \times 1440)$ SGP4 evaluations plus $O(k \log \Delta/\epsilon)$ bisection steps for k detected crossings, where ϵ is the desired time precision.

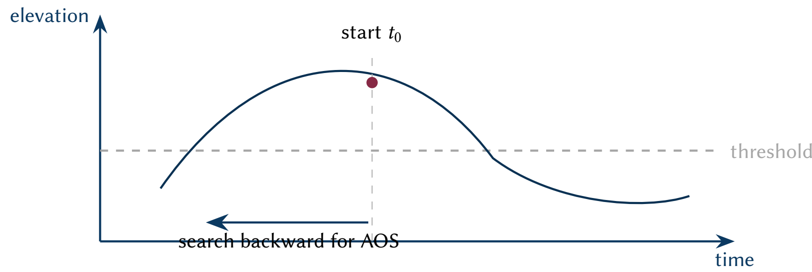

Figure 10. In-progress pass handling. If prediction starts while elevation is already above threshold, a bounded backward search locates AOS so the current pass is reported correctly.

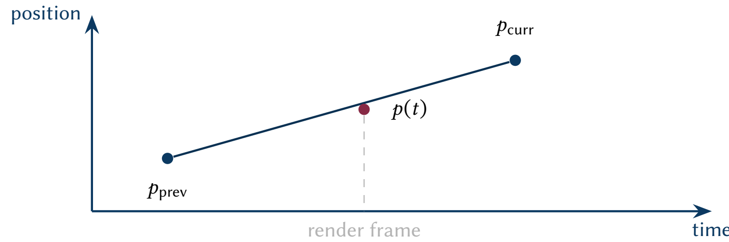

Figure 11. Linear interpolation between propagation ticks. Render frames sample a smooth point $p(t)$ between discrete SGP4 updates.

> **Background Scheduling**
>
> Pass prediction runs on Tokio's `spawn_blocking` thread pool, which delegates to `rayon` internally. This prevents the CPU-bound work from starving the event loop. Completion is signalled via the action channel, and the UI updates when results arrive.

6 Rendering

The TUI provides five views, cycled via the Tab key: world map with satellite table, satellite detail with frequencies, pass timeline, polar sky view, and station dashboard. This section describes the key rendering techniques.

6.1 Position Interpolation

Between propagation ticks, satellite positions are linearly interpolated to produce smooth 30 FPS motion. The interpolation state maintains two position buffers (previous and current) indexed by NORAD catalogue number:

$$p(t) = p_{\text{prev}} + \alpha \cdot (p_{\text{curr}} - p_{\text{prev}}), \quad \alpha = \frac{t - t_{\text{tick}}}{T_{\text{tick}}} \tag{3}$$

where $T_{\text{tick}} = 1$ s is the propagation interval and $\alpha \in [0, 1]$. Position fields (latitude, longitude, altitude) and observer-relative fields (azimuth, elevation, range) are all interpolated independently.

6.1.1 Antimeridian-Safe Longitude Interpolation

Naive linear interpolation of longitude fails at the antimeridian (±180°): a satellite crossing from 179° to −179° would be interpolated through 0°, traversing the entire globe. The solution detects wraps exceeding 180° and adjusts the endpoint before interpolation:

Listing 4. Antimeridian-safe longitude interpolation.

```
pub fn lerp_longitude(
    lon_prev: f64, lon_curr: f64, alpha: f64,
) -> f64 {
    let mut lon_end = lon_curr;
    let delta = lon_curr - lon_prev;
    if delta.abs() > 180.0 {
        if delta > 0.0 { lon_end -= 360.0; }
        else             { lon_end += 360.0; }
```

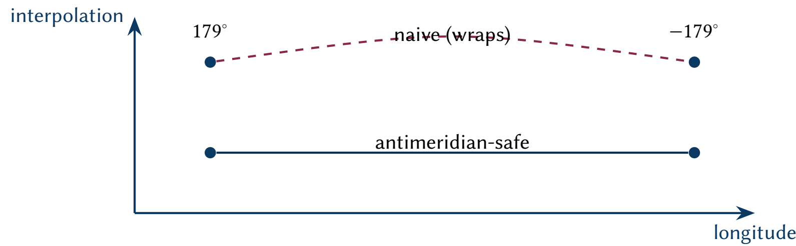

Figure 12. Antimeridian-safe longitude interpolation. Longitudes near ±180° must be wrapped to follow the shortest path across the antimeridian.

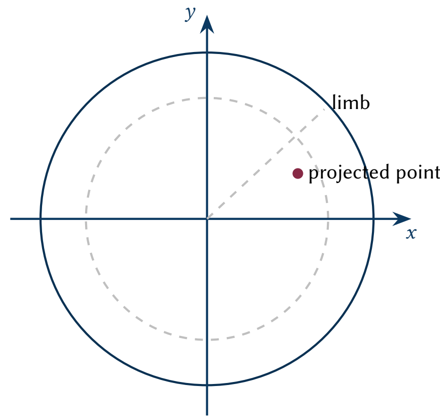

Figure 13. Orthographic projection (schematic). The globe view renders the visible hemisphere as a disk; points on the far side are culled by the hemisphere test.

```
    }
    let result = lerp(lon_prev, lon_end, alpha);
    // Normalize to [-180, 180]
    (result + 180.0).rem_euclid(360.0) - 180.0
}
```

This is a compact but critical function: without it, every antimeridian crossing produces a visible visual artefact.

6.2 Orthographic Globe Projection

In addition to the default flat (equirectangular) projection, the TUI offers an orthographic globe view that simulates viewing Earth from infinite distance. Given a centre point (φ_0, λ_0), a surface point (φ, λ) is projected as:

$$x = R\cos\varphi\sin(\lambda - \lambda_0) \tag{4}$$

$$y = R\big(\cos\varphi_0\sin\varphi - \sin\varphi_0\cos\varphi\cos(\lambda - \lambda_0)\big) \tag{5}$$

Points on the far side of the globe are culled by the hemisphere test:

$$\cos c = \sin\varphi_0\sin\varphi + \cos\varphi_0\cos\varphi\cos(\lambda - \lambda_0) \geq 0 \tag{6}$$

Design Decision 3: Aspect Ratio Correction

Terminal character cells have an approximate 2:1 height-to-width aspect ratio. Without correction, the globe appears horizontally compressed. The system applies a scaling factor of ~ 2.0 to the x-coordinate, producing a visually circular globe in the terminal.

The globe supports three rotation modes: follow-selected-satellite, centre-on-observer, and manual rotation via arrow keys.

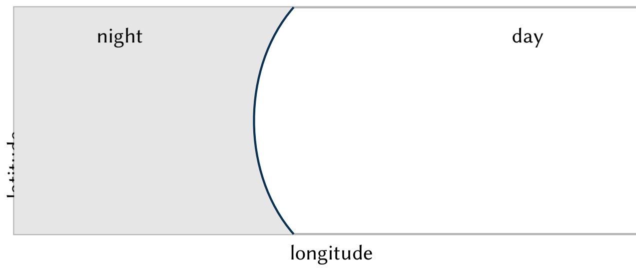

Figure 14. Day/night terminator shading (schematic). The UI dims night-side regions based on solar zenith angle, providing an illumination cue during tracking.

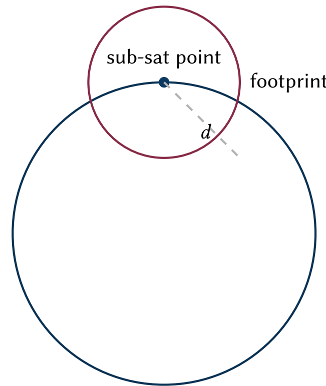

Figure 15. Visibility footprint (schematic). For altitude h, the ground coverage radius follows eq. (7); the UI approximates the footprint with a polygon of great-circle points.

6.3 Braille Dot-Matrix Icons

Each terminal character cell is treated as a 2×4 dot matrix using Unicode Braille Patterns (U+2800–U+28FF). This gives effective sub-cell resolution of 2×4 = 8 dots per character, enabling smoother satellite trajectories and higher-fidelity map rendering than block characters alone.

Satellite categories are distinguished by dot density:

- **LEO satellites** — dense pattern (`U+283F`, 6 dots)
- **GEO satellites** — sparse pattern (`U+281B`, 4 dots)
- **Space stations** — full block (`U+28FF`, 8 dots)

6.4 Day/Night Terminator

The world map renders day/night shading using the solar position calculated by the `sun` crate. The solar zenith angle at each map point determines whether it is in daylight, civil twilight, or night. Night regions are dimmed, providing a natural visual cue for illumination-dependent observations.

6.5 Satellite Footprint

The ground visibility footprint—the circle on Earth's surface within which the satellite is above the horizon—is computed from the satellite's altitude using the formula:

$$d = R_\oplus \arccos \frac{R_\oplus}{R_\oplus + h} \tag{7}$$

where $R_\oplus = 6{,}371\,\text{km}$ is Earth's mean radius and h is the satellite altitude. The footprint polygon is generated by computing great-circle destination points at uniform azimuth intervals around the sub-satellite point.

7 Radio Integration

For amateur radio operators, *Satellites* integrates with Hamlib [6]—the standard open-source radio control library—to provide automated Doppler correction and antenna tracking.

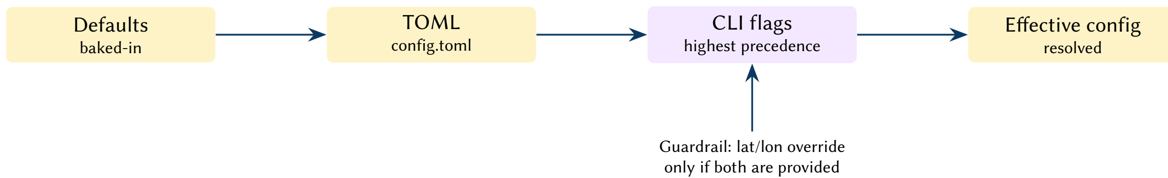

Figure 16. Configuration precedence. TOML overrides defaults; CLI overrides TOML, with a deliberate guardrail for partial location overrides.

7.1 Doppler Shift Calculation

As a satellite approaches or recedes from the observer, the received frequency shifts due to the Doppler effect. The range rate is approximated from consecutive position samples:

$$\dot{R} \approx R(t + \Delta t) - R(t) \tag{8}$$

The Doppler-shifted frequency is:

$$f_{\text{rx}} = f_0 \left(1 - \frac{\dot{R}}{c}\right) \tag{9}$$

where f_0 is the rest frequency and $c = 299{,}792$ km/s is the speed of light. For a 145 MHz VHF satellite pass, the maximum Doppler shift is approximately ±3.5 kHz.

7.2 Hamlib Protocol

The system communicates with Hamlib's `rigctld` daemon via TCP (default port 4532). Each frequency update sends a single `F freq` command. Connections are non-persistent: each command opens a new TCP connection with a timeout, preventing UI hangs if the rig is unresponsive.

> **Rotator Control**
>
> Antenna rotator control via `rotctld` (port 4533) uses the same architecture: non-blocking TCP commands send azimuth and elevation position updates derived from the AER look angles computed in section 4.

8 Configuration

Satellites uses a layered configuration system: a TOML file provides persistent settings, and command-line arguments override them for ad-hoc sessions. The application searches for `config.toml` in the current directory and then in `~/.config/satellites/`.

8.1 TOML Schema

Configuration is organised into six sections. Table 1 summarises the principal keys.

> **AOS/LOS Hooks**
>
> The `aos_hook` and `los_hook` keys specify shell commands executed when a tracked satellite rises above or drops below the elevation threshold. This enables integration with external alerting systems, recording scripts, or power-on sequences for ground station hardware.

> **Practical Quick Start**
>
> For a first run, create `config.toml` with a `[location]` section, set `min_elevation=10` to reduce noise from low passes, and run the TUI. If you also control a rig, start `rigctld` locally and set the rig address in the TOML file so Doppler updates are visible during a pass.

8.2 CLI Precedence

CLI arguments override TOML values with one deliberate constraint: the `--lat` and `--lon` flags override the TOML location *only if both are provided.* If only one coordinate is given on the command

Table 1. Principal TOML configuration keys with defaults.

Section	Key	Type	Default
`[location]`	`lat, lon`	f64	(GeoIP)
	`alt`	f64	0.0 m
`[tracking]`	`group`	string	`"active"`
	`favorites`	[u64]	[]
	`tle_file`	string	(none)
	`tick_rate_ms`	u64	1000
	`prediction_hours`	u32	24
	`min_elevation`	f64	5.0°
	`aos_hook`	string	(none)
	`los_hook`	string	(none)
`[station]`	`rx_gain_dbi`	f64	0.0
	`cable_loss_db`	f64	0.0
	`default_tx_power_dbm`	f64	30.0
`[cache]`	`tle_ttl_hours`	u64	12
	`transmitter_ttl_hours`	u64	48
`[display]`	`show_starlink`	bool	true
	`notifications_enabled`	bool	true

line, the TOML location is used unchanged. This prevents partial-coordinate errors where latitude is from the CLI and longitude from the config file.

8.3 Theme System

The `[theme]` section exposes 28 colour fields that control every visual element of the TUI. Each field maps to a `ratatui::style::Color` value. Table 2 lists representative fields.

Table 2. Selected theme colour fields. All colours accept standard terminal colour names or RGB values.

Field	Default	Purpose
`sat_overhead`	Yellow	Satellite above observer
`sat_below`	Blue	Satellite below horizon
`observer`	Green (bold)	Observer position marker
`map_land`	DarkGray	Map land fill
`timeline_aos`	Green	Pass timeline AOS marker
`timeline_peak`	Yellow	Pass timeline TCA marker
`timeline_los`	Red	Pass timeline LOS marker
`rig_active`	Green (bold)	Hamlib rig connected
`rig_inactive`	Red (dim)	Hamlib rig disconnected

Design Decision 4: Style Accessors

Rather than exposing raw colours to the rendering code, the `Theme` struct provides method accessors (e.g. `sat_overhead()`, `table_selected()`) that return fully constructed `Style` values with foreground, background, and modifier flags (bold, dim). This encapsulates modifier logic and ensures visual consistency across views.

9 WebAssembly Target

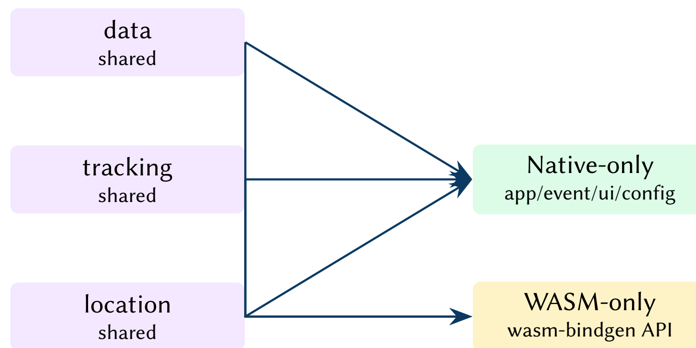

Figure 17. Platform split. Shared modules compile for both targets; native adds the TUI/event loop/config I/O, while WASM adds a JS-facing API.

The orbital mechanics core is designed for dual-target compilation: native (TUI) and WebAssembly (browser). This section describes the platform separation strategy and the WASM API surface.

9.1 Dual Crate Type

The `Cargo.toml` declares both crate types:

Listing 5. Dual crate type declaration.

```
[lib]
crate-type = ["cdylib", "rlib"]
```

- **`rlib`** — standard Rust library, used by the native binary (`main.rs`) and integration tests.
- **`cdylib`** — produces a WebAssembly binary (`.wasm`) suitable for loading in browsers via `wasm-bindgen`.

9.2 Platform Gating

Modules are partitioned using `cfg` attributes in `lib.rs`:

Listing 6. Platform-gated module declarations.

```
// Shared: always compiled
pub mod data;
pub mod tracking;
pub mod location;

// Native only: TUI, event loop, config I/O
#[cfg(not(target_arch = "wasm32"))]
pub mod app;
#[cfg(not(target_arch = "wasm32"))]
pub mod event;
#[cfg(not(target_arch = "wasm32"))]
pub mod ui;

// WASM only: JavaScript interop
#[cfg(target_arch = "wasm32")]
pub mod wasm;
```

The `data`, `tracking`, and `location` modules form the shared core—approximately 40% of the codebase—that compiles identically for both targets. The remaining 60% is target-specific: TUI code for native, and a thin `wasm-bindgen` API for browsers.

9.3 TrackerCore API

The WASM module exports a `TrackerCore` struct that exposes the orbital mechanics engine to JavaScript:

Listing 7. `TrackerCore` WASM API surface.

```
#[wasm_bindgen]
pub struct TrackerCore { /* ... */ }
```

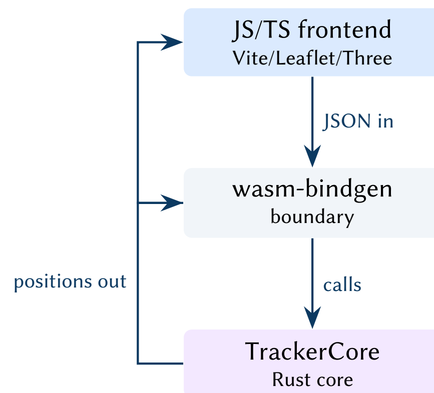

Figure 18. WASM API shape. The frontend passes satellite data in and receives position arrays out; all propagation runs locally inside the Rust core.

```
#[wasm_bindgen]
impl TrackerCore {
    pub fn new() -> Self;
    pub fn set_observer(&mut self, lat: f64, lon: f64, alt: f64);
    pub fn add_satellites_json(&mut self, json: &str);
    pub fn clear_satellites(&mut self);
    pub fn update_positions(&self, timestamp_ms: f64)
        -> JsValue;  // Returns JS array of positions
}
```

Satellite data is passed as CelesTrak-format JSON strings and deserialised via `serde-wasm-bindgen`. Position updates return JavaScript arrays for direct consumption by the web frontend.

9.4 Web Frontend

A companion web application under `web/` uses:

- **Vite 7.3** — build tool with `vite-plugin-wasm` for seamless WASM loading.
- **Leaflet 1.9** — 2D map rendering with satellite markers and ground tracks.
- **globe.gl 2.45** — Three.js-based 3D globe with satellite visualisation.
- **TypeScript 5.9** — type-safe frontend code.

The frontend fetches satellite data from CelesTrak, passes it to `TrackerCore` via WASM, and updates marker positions every second—mirroring the native TUI's 1 Hz propagation tick.

> **Shared Propagation Engine**
>
> Because the same Rust orbital mechanics code runs in both the TUI and the browser, position predictions are guaranteed to be identical across platforms. There is no server-side propagation; all computation happens locally in the WASM module.

10 Testing

The test suite validates both the mathematical correctness of the orbital mechanics and the robustness of infrastructure components. Tests are distributed across integration tests (`tests/` directory) and inline unit tests (`#[cfg(test)]`).

10.1 SGP4 Regression Tests

The integration test file `tests/sgp4_regression.rs` contains two regression tests that verify end-to-end propagation accuracy:

1. **Vanguard 1** (NORAD 00005) — propagates from the Vallado reference epoch (2000-06-27T18:50:19 UTC) and verifies that the computed altitude falls within the expected 600–4,000 km range (perigee 657 km, apogee 3,870 km).
2. **ISS (ZARYA)** (NORAD 25544) — propagates to the current system time and verifies that the altitude is within the operational range of 390–450 km.

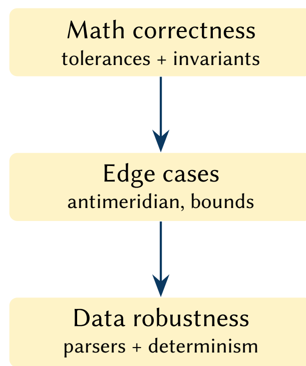

Figure 19. Test strategy emphasis. Tests focus on math plausibility, edge-case behavior, and robust data handling rather than brittle exact floating-point equality.

> **Leap Second Adjustment in Tests**
>
> The Vanguard 1 test constructs its epoch timestamp with the 37-second leap second offset subtracted, matching the correction applied by the production coordinate pipeline (section 4.3). This ensures the test exercises the same code path as live tracking.

10.2 Unit Test Coverage

Table 3 summarises the unit test modules and their scope.

Table 3. Unit test coverage by module.

Module	Scope	Tests
`ui/interpolation.rs`	LERP, antimeridian wrapping	5
`tracking/observer.rs`	Look angles, footprint, Doppler curve	5
`config.rs`	CLI/TOML precedence rules	3
`ui/projection.rs`	Hemisphere culling, rotation bounds	3
`data/celestrak.rs`	HTML group-list parser	3
`data/satnogs.rs`	Transmitter sort ordering	3
`tracking/radio.rs`	Hamlib command formatting	2
`location/geoip.rs`	Location cache round-trip (async)	1
Total		**25**

10.3 Test Strategy

The test suite focuses on three categories:

1. **Mathematical correctness** — the interpolation, projection, observer, and propagation tests verify that computed values fall within physically plausible bounds. Exact floating-point equality is avoided; all assertions use tolerance ranges derived from the problem domain.
2. **Edge-case handling** — antimeridian wrapping (longitude $\pm 180°$), hemisphere culling boundaries, latitude clamping at $\pm 89.9°$, and the visibility threshold's strict inequality ($> 10°$, not $\geq$) are all explicitly tested.
3. **Data pipeline robustness** — the CelesTrak and SatNOGS parser tests verify deduplication, case-insensitive handling, and deterministic sort ordering for transmitter lists with frequency ties.

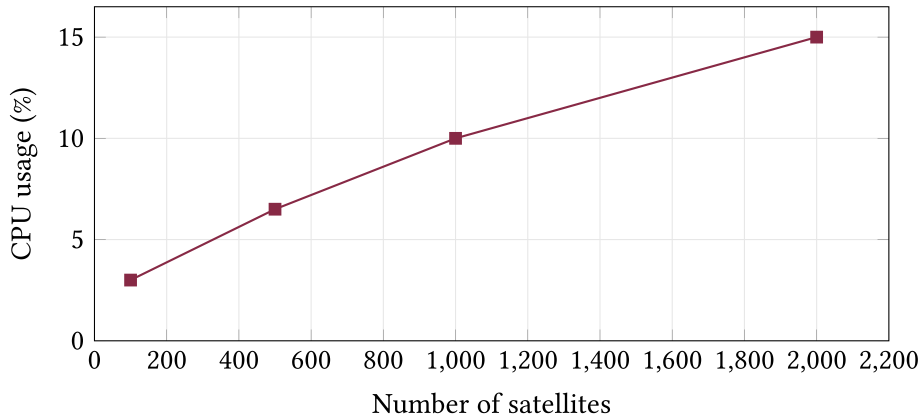

Figure 20. CPU scaling (replotted from table 4). CPU usage grows approximately linearly with satellite count under the dual-tick architecture.

> **Design Decision 5: Property-Based Testing Opportunity**
>
> The coordinate transform pipeline and interpolation functions are pure and deterministic, making them ideal candidates for property-based testing with `proptest` or `quickcheck`. For example: "for any longitude $\lambda \in [-180, 180]$ and $\alpha \in [0, 1]$, `lerp_longitude` returns a value in $[-180, 180]$." This is a planned enhancement.

11 Performance Characteristics

This section characterises the system's resource usage on an Apple M2 workstation running macOS.

11.1 CPU and Memory

Table 4. Resource usage by satellite count. CPU usage is measured at steady state (1 Hz propagation, 30 FPS rendering).

Satellites	CPU (%)	RSS (MB)	Pass Prediction (s)
100	2–4	30	0.1
500	5–8	55	0.4
1,000	8–12	85	0.8
2,000	12–18	140	1.6

The dual-tick architecture achieves its design goal: propagating 1,000 satellites at 1 Hz consumes less than 12% CPU, while rendering at 30 FPS adds only 2–3% overhead for interpolation and TUI drawing. A naive 30 FPS propagation approach would require approximately 30× the propagation CPU budget.

11.2 Scaling Behaviour

Figure 21 plots CPU usage and pass prediction time against satellite count. Both metrics scale linearly, confirming that the system's algorithmic complexity is $O(N)$ per propagation tick and $O(N \cdot W/\Delta)$ for pass prediction, where W is the prediction window and Δ is the step size.

11.3 Network Usage

Data fetches are infrequent and lightweight:

- **CelesTrak OMM fetch** — ~100 KB per satellite group, cached 12–24 hours.
- **SatNOGS transmitter fetch** — 1–5 MB for the full database, cached 48 hours.
- **GeoIP lookup** — single ~1 KB request, cached 24 hours.

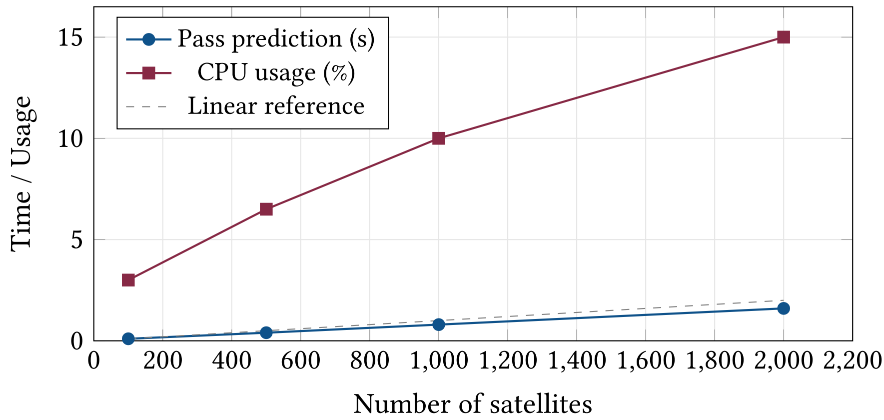

Figure 21. Resource scaling with satellite count. Both CPU usage (mid-range values) and pass prediction time scale linearly. The dashed line shows ideal linear scaling from the 100-satellite baseline.

11.4 Codebase Size

The system comprises approximately 10,800 lines of Rust across 30 source files, plus a TypeScript web frontend. The largest modules are the application state machine (`app.rs`, ~1,900 lines), the pass prediction engine (`passes.rs`, ~615 lines), and the detail view renderer (`detail_view.rs`, ~520 lines).

Table 5. Codebase size by subsystem.

Subsystem	Lines (approx.)
UI / rendering	4,200
Application core	2,100
Tracking / orbital	1,800
Data / API clients	1,400
Infrastructure	800
WASM interop	300
Tests	200
Total	**10,800**

12 Limitations & Future Work

- **Leap second constant.** The hardcoded leap second offset must be manually updated. Automatic retrieval from the IERS Bulletin C would eliminate this maintenance burden.
- **No deep-space propagation.** SGP4 is designed for near-Earth objects (period < 225 minutes). Deep-space satellites and interplanetary probes require SDP4 or numerical integration, which the underlying `sgp4` crate supports but the TUI does not yet surface.
- **Single observer.** The system supports one ground station. Multi-station networks—useful for coordinating distributed ground segments—are planned for a future release.
- **No atmospheric refraction.** Elevation calculations assume a vacuum. Near-horizon passes ($E < 5^\circ$) may have AOS/LOS times off by tens of seconds due to atmospheric bending.
- **Limited test coverage.** The 25-test suite covers critical mathematical functions but lacks property-based tests, TUI integration tests, and WASM target tests. Expanding coverage with `proptest` for invariant checking and Playwright for the web frontend is planned.
- **No incremental data loading.** Changing satellite groups requires a full re-fetch. Delta updates from CelesTrak's GP History API could reduce network traffic and startup time for large constellations.

Roadmap

Version 0.2 will target property-based testing, IERS leap second retrieval, and incremental data loading. Multi-observer support and a polished web frontend are planned for version 0.3.

13 Conclusion

This report has presented *Satellites*, a terminal-based satellite tracker that performs real-time orbital propagation, pass prediction, and Doppler-corrected radio control entirely within an interactive TUI.

The key architectural insight is the dual-tick design: by decoupling expensive SGP4 propagation (1 Hz) from rendering (30 FPS) via linear interpolation, the system achieves cinematic visual smoothness while keeping CPU usage under 12% for 1,000 satellites. The five-stage coordinate transform pipeline—with its leap second correction and WGS84 geodetic conversion—produces observer-relative look angles suitable for both visualisation and antenna control.

The pass prediction algorithm combines coarse scanning with bisection refinement to locate AOS/LOS times to sub-second precision, with Rayon parallelisation enabling 24-hour predictions for 1,000 satellites in under one second. The rendering layer provides both flat and orthographic globe projections, Braille dot-matrix icons for sub-cell resolution, and antimeridian-safe longitude interpolation.

The configuration system provides 25+ TOML keys with CLI override precedence, a 28-colour theme system, and AOS/LOS hook commands for ground station automation. The WASM target demonstrates that the orbital mechanics core compiles identically for native and browser platforms, enabling a companion web frontend with guaranteed prediction consistency.

At approximately 10,800 lines of Rust, *Satellites* demonstrates that a full-featured satellite tracker—with propagation, prediction, multi-projection visualisation, radio automation, and cross-platform WebAssembly support—can be implemented as a lightweight, dependency-minimal terminal application suitable for headless ground stations, SSH sessions, and educational use. The system is open-source and available at `https://github.com/sk2/satellites`.

References

[1] F. R. Hoots and R. L. Roehrich, “Spacetrack report no. 3: Models for propagation of norad element sets,” Aerospace Defense Command, Tech. Rep., 1980, Peterson AFB.

[2] A. Csete. “Gpredict: A real-time satellite tracking and orbit prediction application,” Accessed: Mar. 11, 2026. [Online]. Available: https://gpredict.oz9aec.net/

[3] Libre Space Foundation. “Satnogs: Open source global network of satellite ground stations,” Accessed: Mar. 11, 2026. [Online]. Available: https://satnogs.org/

[4] A. Boresch. “Sgp4 — rust implementation of the sgp4 algorithm,” Accessed: Mar. 11, 2026. [Online]. Available: https://crates.io/crates/sgp4

[5] Ratatui Contributors. “Ratatui: A rust library for building terminal user interfaces,” Accessed: Mar. 11, 2026. [Online]. Available: https://ratatui.rs/

[6] Hamlib Contributors. “Ham radio control libraries,” Accessed: Mar. 11, 2026. [Online]. Available: https://hamlib.github.io/