Final Project Report
Real-Time Satellite Tracker Simulation

Submitted by:
Roni Michaeli 209233873
Neta Cohen 25195774
Matan ziv   208235796

June 2025

Table of Contents

Introduction

In this project, we built a simple Python simulation that follows the International Space Station (ISS) in real time and shows how a satellite would point its camera at moving targets on Earth. Our code uses the Skyfield library every five seconds to get the ISS's current latitude, longitude, and altitude. Then it calculates two viewing angles:

- **Heading** the compass direction from the satellite toward the chosen ground point.
- **Tilt** how much the camera must tilt down from the horizontal to keep that point in view.

We implemented two modes:

1. **Orbit Mode**: the camera looks straight down (heading = 0°, tilt = 0°) and you see the ISS ground track in Google Earth and a small GUI.
2. **Focus Mode**: the program finds each update's nearest target, recalculates heading and tilt, and smoothly reorients the view.

Our solution has three main parts:

- A **Flask server** (`app.py`) that fetches ISS data, predicts ground targets, computes angles and range, and serves KML files for Google Earth along with JSON endpoints for GUIs.
- A **Tkinter GUI** (`simulation_gui.py`) to switch modes and display live heading rate, tilt rate, and estimated energy use.
- A **3D viewer** (`satellite_gui.py & satellite_gui2.py`) built with PySide6 and PyQtGraph that shows a satellite model rotating in real time according to the calculated angles.

This setup lets us watch the full system in action in Google Earth, in a simple dashboard, and in 3D, and understand how a satellite must move its camera to follow a target, how fast it rotates, and how much energy those movements consume.

Demonstration Video

You can watch the full simulation here:
before update
https://drive.google.com/file/d/1JhUz0Zecrq9HHuNc67pvaG3HO3MYRr7L/view

after update new version

https://drive.google.com/file/d/12TNvrS-UMboLWRBIGWjw_YkyLtWeyes8/view

What you'll see in the video:

1. **Google Earth View (track.kml)**
   - At the very start, the ISS path is loaded via our live `track.kml` network link.
   - The camera "looks" straight down (heading = 0°, tilt = 0°) so you see the satellite's ground track moving continuously along its orbit.
   - During this phase, the satellite is not focusing on any target, so the view remains fixed "north and down."
2. **Tkinter GUI (simulation_gui.py)**
   - A small window with three progress bars:
     - **Heading Rate (deg/s)** and **Tilt Rate (deg/s)** remain near zero initially.
     - **Energy Use (W)** stays at the idle baseline.
   - These values update in real time every 100 ms, reflecting the nearly zero angular motion in the static viewing mode.
3. **PySide6 3D Viewer (satellite_gui.py &**`satellite_gui2.py`**)**
   - A 3D mesh of our satellite model hovering above a grid representing Earth.

    - Angle readouts (“Heading” and “Tilt”) display until focus mode begins.
    - The camera position in the 3D view remains fixed, matching the Google Earth down-looking view.

4. **Activating Focus Mode**
    - After a few seconds, click **Start Focus Simulation** in the Tkinter GUI (or wait for the program’s automatic toggle).
    - The Flask server switches from `/orbit.kml` to `/live.kml`, and Google Earth begins to update its LookAt parameters toward the *nearest* ground target.

5. **Dynamic Target Tracking**
    - **Google Earth:**
        - You’ll see a red placemark for the current target.
        - The camera smoothly pans (heading) and tilts down to keep that target centered.
        - As the ISS moves, the script re-computes the closest target point every 5 s and updates the view accordingly.
    - **Tkinter GUI:**
        - **Heading Rate** and **Tilt Rate** bars jump to nonzero values, showing how many degrees per second the satellite must rotate.
        - **Energy Use** bar rises above baseline, reflecting the extra power draw for angular motion and focus calculations.
    - **3D Viewer:**
        - The satellite model rotates in real time: yaw around its vertical axis to match heading, pitch to match tilt.
        - Numeric labels update every 5 s with the current heading and tilt angles.

By the end of the clip you can observe how our full system, Python back end, KML network links, and two GUIs work together to fetch live ISS data, compute viewing angles, and visualize both the satellite’s orbit and its energy-informed attitude adjustments.

Code Structure

- **Flask /** `app.py` → data fetching, target prediction, KML/JSON serving
- `shared_state.py` → safe data sharing between threads
- `simulation_gui.py` → user control of focus and monitoring of rates/energy
- `satellite_gui.py` &`satellite_gui_2.py`→ 3D visualization of satellite attitude
- `networklink.kml` → ties everything into Google Earth's camera view

1. `static/networklink.kml`

- **What it is**: A simple KML file that you open in Google Earth Pro.
- **Role**: It points Google Earth at your Flask server's `dynamic.kml` endpoint every 5 seconds.
- **Key parts**:
  - `<refreshInterval>5</refreshInterval>` tells Earth to reload the KML link every 5 s.
  - `<href>http://192.168.68.129:5003/dynamic.kml</href>` is where your Flask app serves live tracking data.

2. `src/app.py` (main Flask server + background logic)

This script does three jobs at once:

1. **Fetch and update ISS data**
    - **`fetch_iss_tle()`** downloads the latest Two-Line Element set from Celestrak.
    - A background **`satellite_updater()`** thread runs every `UPDATE_INTERVAL_S` (5 s), calls `get_sat_position()`, and appends `(lat, lon, alt)` to `positions_history`.
2. **Precompute ground targets**
    - At startup it calls `precompute_shifted_targets()`, which fills `target_points` with predicted ground-track points (one per minute for 90 minutes by default), optionally shifting some sideways for variety.
3. **Serve KML and JSON endpoints**
    - `/orbit.kml` → camera looks straight down at the ISS path.
    - `/live.kml` → camera tracks the *nearest* target point, updating heading, tilt, and range.
    - `/dynamic.kml` → chooses between orbit-only and live KML based on `focus_mod`.
    - `/state` and `/angles` → return JSON for the Tkinter GUI (`simulation_gui.py`) to read focus flag plus angular rates, and for the 3D viewer (`satellite_gui.py &satellite_gui_2.py`) to read the current heading/tilt angles.
    - `/set_state` → lets the Tkinter GUI toggle `focus_mod` on and off via POST.

Finally, in the `if __name__ == "__main__":` block it:

- Starts the **Tkinter GUI** thread (`start_simulation_gui`).

- Fetches the initial TLE and precomputes targets.
- Starts the **satellite_updater** thread.
- Launches the Flask server on port 5003.

3. `src/shared_state.py`

- A simple thread-safe container (class `SharedState`) for sharing data between threads:
    - `focus_mod` (bool)
    - `heading`, `tilt` (float)
    - `heading_rate`, `tilt_rate` (float)
- All getters and setters are protected by a `threading.Lock` to avoid race conditions.
- You import the single instance `state` wherever you need to read or write these values.

4. `src/simulation_gui.py`

- **Purpose**: Let you toggle focus mode and watch live angular rates + energy use.
- **Key features**:
    1. **Start/Stop Focus** button calls `/set_state` to flip `focus_mod`.
    2. **Progress bars** for:
        - Heading rate (deg/s)
        - Tilt rate (deg/s)
        - Energy use (computed locally as a quadratic function of the two rates plus idle/focus offsets)
    3. A loop (`update_gui`) runs every 100 ms:

        - GET `/state`
        - Update button text, rate labels, bar values.

This GUI starts automatically when you run `app.py`.

5. `src/satellite_gui.py`

- **Purpose**: Show a 3D satellite model rotating in real time to match heading/tilt.
- **Workflow**:
    1. Load an STL mesh (`pod_box.stl`) and center/rotate it upright.
    2. Set up a `GLViewWidget` (PyQtGraph + PySide6) with a ground grid at z = 500 km.
    3. Every `POLL_INTERVAL` (5 s), call GET `/angles` → receive JSON `{ heading, tilt }`.
    4. Convert those angles to yaw/pitch, reset the mesh transform, and apply the new orientation.
    5. A fast 100 ms timer updates the on-screen camera parameters in the status bar.

You launch this separately (after `app.py`) by running `satellite_gui.py`.

Algorithms

4.1 Angle Calculations

1. **Bearing (heading):**
   The function `bearing_deg` calculates the **heading angle** (also known as **initial bearing**) between two points on the Earth's surface. It gives the **direction you need to face** when traveling from the starting point `(lat1, lon1)` to the destination `(lat2, lon2)` along the shortest path on the globe (a great-circle route).

How the algorithm works:

```
φ1, φ2 = math.radians(lat1), math.radians(lat2)
Δλ = math.radians(lon2 - lon1)
```

Convert the latitudes and the longitude difference into **radians**, since trigonometric functions in Python expect radians.

```
x = math.sin(Δλ) * math.cos(φ2)
y = math.cos(φ1) * math.sin(φ2) - math.sin(φ1) * math.cos(φ2)
* math.cos(Δλ)
```

These are the **components of the direction vector** from the starting point to the destination: `x` is the eastward component, `y` is the northward component.

```
θ = math.atan2(x, y)
```

Compute the angle between the direction vector and geographic north using `atan2`, which gives the result in radians in the range .

```
return (math.degrees(θ) + 360) % 360
```

Convert the result to **degrees** and normalize it to the range [0°,360°)[0°, 360°)[0°,360°), so it can be interpreted as a compass direction:
`0°` → North, `90°` → East, `180°` → South, `270°` → West

The final value is the **compass bearing** from the starting point to the destination, following the shortest path over the Earth's surface. It's the direction you would face if standing at the satellite's location and looking toward the target.

2. **Elevation & Tilt:**

   The **tilt** angle describes how much the satellite's camera (or sensor) must look **downward** from the local vertical direction in order to point toward the target. A tilt of `0°` means the camera is pointing straight down (nadir), while a tilt of `90°` means it's looking along the horizon.

the algorithm works:

```
elev_deg = math.degrees(math.atan2(sat_alt_km, dist_km)) if
dist_km else 90.0
```

This computes the **elevation angle**, which is the angle above the horizontal from the **target** to the **satellite**.
It uses the inverse tangent of the ratio between:

`sat_alt_km`: the altitude of the satellite above the Earth's surface,

`dist_km`: the ground (horizontal) distance between the satellite and the target (computed separately using the Haversine formula).

The result is converted from radians to degrees.
If the distance is zero (i.e., the satellite is directly above the target), the elevation is set to `90°`.

```
tilt = max(0.0, min(90.0, 90.0 - elev_deg))
```

The **tilt** is calculated by subtracting the elevation angle from `90°`, because:

- A higher elevation means a **lower tilt** (closer to looking straight down).
- A lower elevation means a **higher tilt** (closer to the horizon).

We also **clamp** the value to stay within the valid range `[0°, 90°]`.

Interpretation:

`tilt = 0°`: the satellite is looking **directly downward** (nadir).
`tilt = 45°`: the satellite is looking at the target at a moderate slant.
`tilt = 90°`: the satellite is looking **horizontally**, i.e., the target is on the horizon from the satellite's perspective.

3. **3D Range for LookAt:**

We calculate the true slant distance between satellite and target in 3D by converting each geodetic point to cartesian (x,y,z) and computing the Euclidean distance. Then add a fixed offset (e.g., 3 km) when setting the LookAt `range` in KML.

First we calculate the real distance:

The following function calculates the **3D (straight-line) distance in kilometers** between a satellite and a ground target, taking into account their **latitude, longitude, and altitude**.

```
def calculate_3d_distance_km(sat_lat, sat_lon, sat_alt_km,
tgt_lat, tgt_lon, tgt_alt_km):
```

The function receives the **latitude, longitude, and altitude (in km)** of:

- The **satellite**: `sat_lat`, `sat_lon`, `sat_alt_km`
- The **target** on Earth: `tgt_lat`, `tgt_lon`, `tgt_alt_km`

Set the Earth's radius:

```
R_earth = 6371.0
```

Earth's average radius in kilometers.

Convert geographic coordinates to 3D Cartesian coordinates:

```
def to_cartesian(lat, lon, alt, R_base):

    radius = R_base + alt

    lat_rad = math.radians(lat)

    lon_rad = math.radians(lon)

    x = radius * math.cos(lat_rad) * math.cos(lon_rad)

    y = radius * math.cos(lat_rad) * math.sin(lon_rad)

    z = radius * math.sin(lat_rad)

    return x, y, z
```

This helper function Converts spherical coordinates (latitude, longitude, altitude) to **3D Cartesian coordinates** (x,y,z)(x, y, z)(x,y,z).

It Adds the altitude to Earth's radius to get the full radial distance from Earth's center. Converts lat/lon to radians using standard formulas.

Convert both satellite and target positions:

```
x1, y1, z1 = to_cartesian(sat_lat, sat_lon, sat_alt_km,
R_earth)
```

```
x2, y2, z2 = to_cartesian(tgt_lat, tgt_lon, tgt_alt_km,
R_earth)
```

These lines compute the Cartesian coordinates of the satellite and the ground target.

Compute 3D Euclidean distance:

```
distance = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2 + (z2 -
z1) ** 2)
```

This is the standard 3D Euclidean distance formula.

```
return distance
```

Returns the final distance in kilometers between the satellite and the target in space.

In the LookAt the range is set to be distance * 1000 (m).

4. **Angular Rates:**

Measure changes in heading and tilt over each update interval (5 s) in degrees per second. This code computes how fast the **heading** and **tilt** angles are changing over time i.e., their **angular velocities -** and stores the values in a shared state.

```
delta_t = now - prev_time
```

This calculates the time difference between the current moment (`now`) and the previous update (`prev_time`).

```
delta_heading = abs(heading - prev_heading)
```

```
delta_tilt = abs(tilt - prev_tilt)
```

These lines compute the **absolute change** in heading and tilt since the previous update.

`heading` and `tilt` are the **current values**.
`prev_heading` and `prev_tilt` are the **previous values**.

Taking the absolute value ensures the result is always positive (magnitude of change).

```
heading_rate = delta_heading / delta_t

tilt_rate = delta_tilt / delta_t
```

These lines calculate the **rate of change** (i.e., angular velocity) for heading and tilt:

- `heading_rate`: how many degrees per second the heading has changed.
- `tilt_rate`: how many degrees per second the tilt has changed.

4.2 Energy Consumption Model

Constants and What They Mean

- **P_idle = 5 W** (defined inside `update_energy_use()` in **`simulation_gui.py`**)
  This is the "floor" power that the satellite bus always consumes power for avionics, thermal control, and other baseline systems even when no attitude maneuvers take place.

- **k_h = 0.03** and **k_t = 0.04** (also in `update_energy_use()`)
  These are the coefficients that convert squared angular rates into extra watts. Physically, they capture how reaction wheels or control-moment gyros require more torque (and therefore more electrical power) as rotation speed increases; the quadratic relationship (ω2\omega^2) reflects the nonlinear motor torque power curve. The heading coefficient khk_{h} applies to rotation about the vertical axis; the tilt coefficient ktk_{t} applies to rotation about the lateral axis.

- **P_focus = 10 W when focus mode is active, otherwise 0** (again in `update_energy_use()`)

This fixed overhead models the additional load of running target-tracking algorithms and activating high-rate sensor hardware (such as a camera) whenever the satellite is in “focus” mode.

Algorithm Overview

1. **Read current state**
   The GUI calls the function `update_energy_use()` (in **`simulation_gui.py`**) once every 100 ms. That function retrieves three values from the shared state:

   - A boolean flag (`focus`) indicating whether focus mode is on.
   - The current heading-rate (`ω_h`) in degrees per second.
   - The current tilt-rate (`ω_t`) in degrees per second.
     These rates are themselves computed on the server side by the Flask endpoints `stream_kml()` and `stream_kml_orbit_only()` (in your main application file), which measure the change in heading and tilt between successive time stamps and divide by the elapsed time.

2. **Compute each power contribution**

   - **Baseline:** Always include **P_idle**.
   - **Rotation cost:** Multiply khk_h by ωh2\omega_h^2 and ktk_t by ωt2\omega_t^2.
   - **Focus overhead:** Add **P_focus** if focus is active.

3. **Sum all terms**
   The function sums these four contributions to yield the instantaneous power draw PP (in watts).

Why This Model Is Correct

- **Quadratic dependence on rotation rate**
  Reaction wheels or control moment gyros (CMGs) draw power approximately proportional to the square of their angular velocity ($\Delta P \propto \omega^2$), highlighting that faster slews demand disproportionately more electrical power.

- **Baseline consumption**
  Even without motion, the satellite must run its avionics, maintain thermal balance,

and communicate, so no realistic power model can drop below a nonzero idle floor.

- **Fixed focus overhead**
  Engaging tracking algorithms and camera sensors imposes a clear, constant extra load; modeling it as a single 10 W term cleanly separates the cost of attitude control from the cost of payload operation.

How It Appears in Code

- **`simulation_gui.py:update_energy_use()`** defines and uses **P_idle**, **k_h**, **k_t**, and **P_focus** exactly as described, and returns their sum every GUI update cycle.

- **`main.py:stream_kml()`** and **`stream_kml_orbit_only()`** compute the heading and tilt rates by storing `prev_heading`, `prev_tilt`, and `prev_time`, then dividing the angular difference by elapsed seconds passing the results into `state.set_values()` so the GUI can read them.

Conclusion

We have successfully created a clear, end-to-end simulation that tracks the ISS in real time and shows how a satellite would aim its camera at ground targets. Key achievements include:

- **Live Data Integration**: Every five seconds, the system retrieves the latest ISS position using Skyfield and updates all components automatically.
- **Accurate Angle Computations**: We convert geographic coordinates into heading and tilt angles, ensuring the camera view points precisely at the chosen target.
- **Flexible Viewing Modes**:
    - *Orbit Mode* keeps the camera fixed straight down to display the ISS ground track.
    - *Focus Mode* dynamically selects the nearest target and smoothly pans and tilts the view to follow it.
- **Multi-Platform Visualization**:
    - Google Earth Pro shows the live KML network link.
    - A Tkinter dashboard displays real-time rotation rates and estimated power consumption.
    - A PySide6/QtGraph 3D viewer animates the satellite model's attitude changes.
- **Energy Insight**: By measuring angular rates and applying a simple quadratic power model, we can see how much additional energy is needed when the satellite moves its camera.

Through this project, we gained hands-on experience with orbital data, geographic-to-cartesian math, KML generation, web services, and multiple GUI frameworks. The result is a modular, easy-to-understand system that demonstrates both the theory and practice of real-time satellite tracking and camera control.