Developer Guide: Solar System Simulator

---

1. Overview and Architecture

The Solar System Simulator is built as a *single-file application* using the Racket programming language, specifically the *Intermediate Student with lambda (ISL+)* language level. It relies on the *big-bang* system from the *2htdp/universe* library to manage the simulation loop, state updates, and rendering.

The application state is centralized in a single structure (SimState), which is updated every clock tick to simulate physics and rendered every frame to display the simulation.

Source Files

- *solarnew.rkt:* The entire source code is contained within this single file. It includes data definitions, constant declarations, logic helpers, the physics engine, the rendering engine, and input handlers.

External Libraries

- *2htdp/image:* Used for all graphical operations. It handles loading external assets (e.g., *bitmap/file* for planet images), creating geometric primitives (e.g., *circle*, *rectangle* for UI), and compositing the scene using *place-image* and *overlay*.
- *2htdp/universe:* Provides the runtime engine. It manages the simulation loop via *big-bang*, linking the clock ticks (*on-tick*), rendering (*to-draw*), and input events (*on-key*, *on-mouse*).

2. Core Data Structures (The Model)

The simulation creates a virtual universe defined by the following key structures:

- *SimState:* The root structure representing the world at any specific moment. It holds the list of celestial bodies, the current camera settings (*zoom*, *locked-body-name*), user interface state (*active-tab*, *gallery-index*), the *day-counter*, and transient objects like the *asteroid-belt* and *active-comets*.
- *CelestialBody:* Represents a major planet. It contains physical properties (radius, orbit distance) and dynamic properties (current *angle*, *angular-speed*). Uniquely, it is a recursive structure that contains a list of *Satellite* structures (moons).

- *Location:* A helper structure used to map polar coordinates (orbit angle/distance) into Cartesian coordinates (x,y) for rendering.

3. Top-Level Functions

A. The Physics Engine: next-world

This function serves as the *State Updater*. Triggered every 1/100th of a second, it transforms the current *SimState* into the next valid state.

- *Time Scaling:* It retrieves the current simulation speed from the *SPEED-LEVELS* list using *speed-index*. This multiplier (*speed-mult*) allows the user to pause or accelerate time. Moreover, it updates the *day-counter* field to synchronize the calendar date with the movement of the planets.
- *Orbital Mechanics:* It calls the helper *update-bodies*, which recursively applies *next-body-state*. This increments the *angle* of every planet and moon based on their specific angular speeds.
- *Dynamic Entities:*
  → *update-asteroids:* Updates the orbital position of the asteroid belt.
  → *filter-comets* & *maybe-spawn-comet:* Updates the linear position of comets (*x+vx*), removes those that drift off-screen, and probabilistically spawns new ones.

B. The Rendering Engine: draw-world

This function is responsible for *Displaying the State*. It consumes the *SimState* and produces a single, composited *Image*.

- *Camera Logic:* The simulator does not use a static viewpoint.
  1. It searches for the currently locked planet using *find-body-by-name*.
  2. It converts that planet's angle and orbit radius into Cartesian coordinates (*target-loc*) using *angle-to-location*.
  3. All other objects are drawn relative to this *target-loc*, ensuring the locked planet remains fixed at the screen center (*CENTER-X*, *CENTER-Y*).
- *Layering:* The scene is built from back to front:
  1. *Space Scene:* Comets, asteroids, and planets are drawn using *draw-all-bodies*. This helper handles the scaling of bitmaps based on the *zoom* factor and draws orbital paths.
  2. *Interface:* The *draw-interface* function overlays the HUD, Legend, and Info Panel on top of the space scene. It handles the logic for displaying the correct text and gallery images based on *active-tab, and* includes logic to convert the *day-*

counter into a readable *Year* and *Day* format (using floor and modulo arithmetic) for the Heads-Up Display.

C. Input Handling:

→ *handle-mouse*

This is the most complex control function, managing two distinct interaction layers simultaneously:

1. *UI Interaction:* It detects clicks within the bounds of the Info Panel buttons (*PANEL-X*, *PANEL-Y*). If a button is clicked, it updates the *active-tab* or *gallery-index*.
2. *World Interaction:* If the click is outside the UI, it calculates the distance between the mouse cursor and every planet's current screen position using the Euclidean distance formula. If a planet is clicked, it updates *locked-body-name*, triggering the camera to pan to that object.

→ *handle-key*

This function manages global simulation variables:

1. *Zooming:* Maps the Up/Down and =/- keys to increment or decrement the *zoom* field.
2. *Time Control:* Maps the *'i'* and *'d'* keys to modify the *speed-index*, altering the flow of time.
3. *Simulation Reset:* Maps the *'r'* key to reload the INITIAL-STATE, effectively resetting the planets and the calendar to their starting positions.

4. Helper Utilities

Several utility functions support the main logic:

- *angle-to-location:* A math helper that converts the polar coordinates used for physics (r,β) into Cartesian coordinates used for drawing (x,y) using trigonometric functions (*cos*, *sin*).
- *get-planet-info:* A lookup function that returns static data (mass, diameter, description) for the UI panel based on the planet name.
- *find-body-by-name:* A recursive list search used by the camera system to locate the target planet's data.