7 Homework Assignment

Solar system

In this exercise you will build an animation of (part of) our solar system with planets circling around a sun and moons circling around some of the planets. The animation is started and paused with two buttons. A slider controls the speed of the animation. Celestial bodies are drawn as colored circles. Each celestial body revolves in circular orbits around another body, except for the sun, which is static in the center of the panel.

Before read the design hints and Approach

7.1 Design hints

1. All celestial bodies are objects of the class CelestialBody or of its subclass CelestialBodyExtra. The latter one represents celestial bodies that have one or more objects in orbit.

2. The position of, e.g., a moon is quite complex to calculate directly, since it involves the also moving position of the planet it circles around. This can be solved by describing the position of a celestial body always *relative* to its center of rotation. This then can be best done by polar coordinates, using an angle (measured from the positive x-axis) and a radius (distance to its center of rotation). For the movement, angular speed is used, i.e., the change in angle per time unit (e.g., radians per day).

   Only one celestial body does not have such a center: the sun. The sun is positioned in the middle of the canvas and does not move.

3. To make this work, somehow a celestial body must be be connected to its direct satellites. The key idea is to define a class `CelestialBodyExtra`, using inheritance to have all the features of class `CelestialBody` and also an array of satellites.

4. All calculations are performed in doubles. Only at the actual drawing, the `double`s are converted to `int`s. This can be performed with `double Math.round(double x)`, which rounds a double to the nearest whole number (but still a `double`). Then a so-called *cast* converts the `double` to an `int`: `(int) Math.round(...)`.

5. Because Java's `Graphics` objects uses cartesian (rectangular) coordinates, the relative polar coordinates need to be translated to absolute rectangular coordinates at drawing. If (x_c, y_c) is the position of the center of rotation, then the position of a point with distance d and angle α has rectangular coordinates $(x_c + d\cos\alpha, y_c + d\sin\alpha)$.

6. For our own solar system, circles are not a bad approximation. Elliptic orbits are an exercise for the dedicated programmer.

7. Each CelestialBody has a size, a color, a distance to its center of rotation, an angular velocity (omega), and an angle thst determines its position (see above).

8. The constructor for CelestialBodyExtra can best first call the constructor of CelestialBody with `super(size, distance,...)`. Add an extra parameter to this constructor that is an array of CelstialBodies and represents the satellites of the CelestialBodyExtra under construction.

9. Note that it is not necessary to show the complete orbits on the panel. The draw methods of `Graphics` will take any int arguments and only show what is inside the panel.

7.1.1 Time and animation

The passing of time is modeled by a `Timer` object that issues ActionEvents at regular intervals. Each of these events causes an update of position of every celestial body, corresponding to the amount of time that has passed. This update is implemented by a method `void step(double time)`. The first object on which `step` is called is the sun. From there the `step` methods of all its satellites are called. These call the step methods of their satellites and so on.

In `step` the angular distance traveled is computed from the angular speed and the size of the timestep.

To see something move on the screen, you have to accelerate time. E.g., in one second of viewing time, the bodies move along their orbits as if one day has passed. This relation between viewing time and simulated time is determined by the value of the parameter `time` of `step` and the delay setting of the Timer object. The slider should determine this relationship.

7.1.2 Drawing

When `paintComponent` of the panel is called, the solar system has to be drawn. This results in a cascade of calls, as with the `step` method above, to the method `void draw(Graphics g, double pivot_x, double pivot_y)` in each celstial body. The Graphics object `g` is the parameter of the original `paintComponent` that is just passed through. The parameters `pivot_x` and `pivot_y` are the coordinates of the rotation center of this celestial body.

7.2 Approach

First write a program for a simple solar system with the sun and Mercury. Some starter code is provided. It contains the GUI components and some, often partially implemented, methods.

Then add more planets. How many is up to you.

Then make the slider work.

To see how flexible this setup is, add afterwards a small spaceship circling the moon of the earth and an astronaut circling this spaceship.

7.3 Improvements and embellishments

The application described above will be graded with an 8 maximally. Adding to it will give you extra points. Some ideas follow here.

Make the sun draggable. With the sun, the whole solar system will move.

Use two scale factors, one for the orbits and one for the size of the celestial bodies and draw the solar system truthfully to reality on these scales.

Show the ratio between simulation time (real viewing time) and simulated time, for instance as simulated days per viewed seconds.

7.4 Anything goes

If you like, add extras of your own choice and find out what is possible. Suggestions: Add a button to move step by step; add a button to go back a step (for example to look again at an interesting configuration that flashed by). Add the option to add a planet, e.g., by clicking with the mouse on the place where you want it, giving the parameters via a dialog box.

If you still feel creatively challenged, redesign the whole thing using the laws of Newtonian mechanics rather than the predefined circles. :-)