Discovery Lab 1: Models and Simulation

CISC181 Spring 2014
Assigned: March 3
Due: March 9 at 11:55PM on Sakai
Text references: Chapters 4, 5, 8, 10, 11

Discovery labs should be completed by a pair of students. You may work alone (although not recommended), but you may not work in a group of 3.

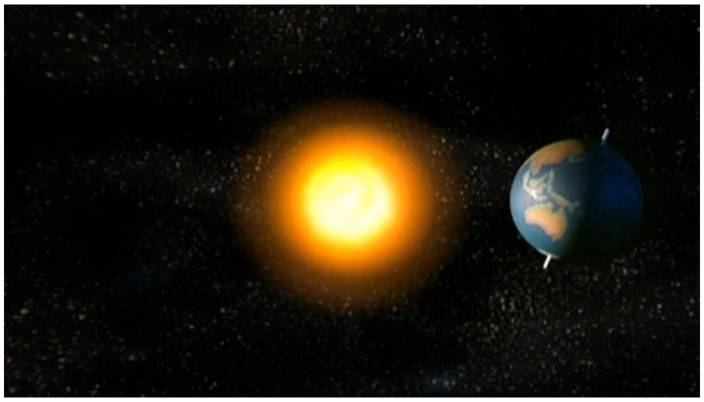
Discovery labs are designed to look at a particular area of Computer Science applications and research. In each lab you will develop a working prototype application that practices fundamental programming techniques. After completing the prototype, your group will apply these techniques to develop a checkpoint for your course project.

In this discovery lab you will explore the area of Models and Simulation.

Objectives:

1. Design, implement, test, and debug a program that uses: basic computation, standard conditional and iterative structures, and the definition of methods.
2. Choose appropriate conditional and iteration constructs for a given programming task.
3. Apply the techniques of structured (functional) decomposition to break a program into smaller pieces.
4. Create algorithms for solving simple problems.
5. Create structured data hierarchies
6. Write programs that use the following object oriented techniques: encapsulation, inheritance.
7. Encode mathematical models in object methods.

* Problems you need to solve and turn in are listed in green text.



One of the primary uses of computing power since the first computer has been the simulation of mathematical models of physical systems. The rapid increase in computing power expanded the importance of computational models and computer simulation. Many recent advancements in biology, physics, climate science, engineering, and financial analytics (to name a few) rely on sophisticated models and powerful amounts of computation. A model contains:

- Computational abstractions of system components
  - Not all information in a system is useful to represent in a computational model
  - We will need to carefully choose what data types and values to represent
- Interaction rules
  - These describe constraints and other functions that determine how system components interact with each other
  - Deterministic vs. Stochastic: some rules always produce the same result given the same input, but others may involve some amount of random outcome

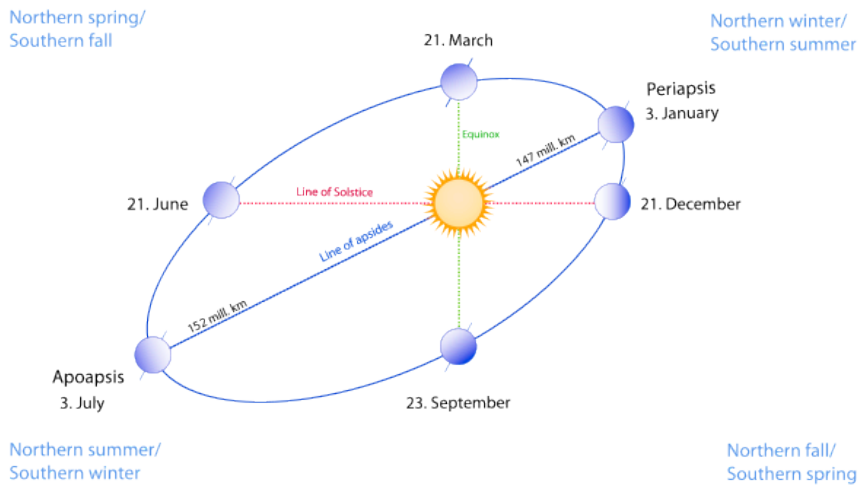
In addition to the internal representation and rules, a discrete event simulation requires a definition of external events that may affect the system.

- One of the most common external events is the advancement of **time**.
  - Since computation takes actual time, most simulations do not occur **in real time**.
  - A unit of time is chosen for the **tickLength** and all updates to the model for one **tick** of the simulation use this value to compute updates.
  - The frequency of calls to update the model for one **tick** is up to the programmer. In scientific simulations this often happens as soon as the computing resource has finished the previous update. In games and other real time simulations a controlling process typically monitors how much time has elapsed since the previous update and will wait to update the model until a designated update rate.
- Other common events include human interaction with the simulation
  - In a games, human input through keyboard/mouse/touchscreen/other controller generates a discrete event that forces an update to the model

In this assignment you will be developing a model and simulation for a 2-body celestial motion system (namely the Earth and the Sun). While our model for this is not particularly complex, it could be extended to allow for a more realistic simulation of N-body problems. It also introduces many concepts we need to develop a good design for an Android game application.

Familiarize yourself with the basic model for our solar system:
http://en.wikipedia.org/wiki/Solar_System

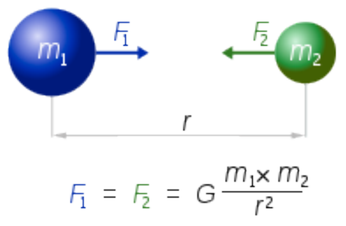
and Earth's orbit:
http://en.wikipedia.org/wiki/Earth's_orbit



We will make the following simplifications for our model:

- We only need to represent the Earth and the Sun (no other interactions will be modeled)
- The Sun will be considered to be stationary and we will ignore the small difference between the center of the Sun and the true center of Earth's orbit
- Earth's rotation around its axis will not be modeled

Hence we are left with a model that can simply compute Newton's Law of Universal Gravitation:
http://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation



$$F = G\frac{m_1 m_2}{r^2} \qquad F_1 = F_2 = G\frac{m_1 \times m_2}{r^2}$$

We'll begin by noting that an object in Java is a model. It uses properties to represent data types and values. It uses methods to encode interaction rules. By using objects of different types, we can design a complex object model that represents a real system.

- I have provided 3 basic classes that will help with this assignment and your project. Look through the code for Component3d, Position, and DirectionVector. When you feel you have an understanding of what data they represent and how they work, complete the move(DirectionVector v, double scaling) method on Position. Make sure that your code passes the test in DL1Tests.

- Place all of the classes you create for this assignment in the DL1 package
- Create 3 classes to represent a small class hierarchy:
    - CelestialBody: has a Position, DirectionVector, and mass (double). Add a constructor that takes all 3 and getters for each property.
    - Sun: extends CelestialBody and passes default values for all of the superclass properties. Position and Direction should both be 0, 0 and the mass of the Sun in kilograms is approximately 1.9891E30 (this is how you write 1.9891x10^30 in Java).
    - Planet: extends CelestialBody and takes all 3 superclass parameters as arguments to the Planet constructor and passes them up to the superclass constructor
    - Make sure to uncomment the hierarchy tests in DL1Tests and run them

- Add the following method to the Planet class. This computes the update to the DirectionVector on a Planet from the force of the Sun's gravity

```
public void updateDirectionFromSun(Sun s, double t) {
    double dx = s.getPosition().getX() - getPosition().getX();
    double dy = s.getPosition().getY() - getPosition().getY();
    double dist = Math.sqrt(dx*dx + dy*dy);

    // Newton's gravitational constant in N m^2 kg^-2
    double force = 6.67384E-11 * s.getMass() / (dist*dist*dist);

    getDirection().setX(getDirection().getX() + t * force * dx );
    getDirection().setY(getDirection().getY() + t * force * dy );
}
```

- Add the method, updatePosition(double t), to the Planet class such that after calling it the Planet's position is updated according to its DirectionVector and the given time, t. Hint: you should use your Position move method from earlier.

  Make sure both of the update methods pass the tests in DL1Tests.

- Create a class, SolarModel, that will contain a Sun, a Planet, and a tick length. In addition to getters for each property, it should have an onTick() method that updates the planet using the model's Sun and tick length for time.

  Make sure your SolarModel passes the tests in DL1Tests.

- Run the Simulation.java program to see the full model simulation. If you expand Eclipse's Console window size you will see the output as a "flip-book" style animation.

- Modify your SolarModel to allow for two Planets instead of one. Do not worry about Planet to Planet gravitational pull.

- Modify the Simulation.java program to create a SolarModel with both planets. Make a new constructor that takes the other Planet and make your old constructor pass null for the other Planet. Fix your onTick logic to update the other Planet if it is not null.

You should look up the statistics on Venus (mass in kilograms, distance from Sun in meters, and orbital velocity in meters per second). Start Venus with position x=distance from Sun, y = 0 with the DirectionVector x=0 y=velocity in meters per second.

Uncomment the lines in the printSolarModel and run the simulation again to see that Venus also orbits around the Sun.

Please export your modified DL1 project as an archive for submission to Sakai. Make sure to include both student names (if working in a group) in a comment in Simulation.java.