Solar

Assignment Out:	Thu, Jan 22
Help Session:	8:00pm Tue, Jan 27
Design Due By:	Fri, Jan 30, arranged with your design check TA
Code Due:	11:59pm Wed, Feb 4

1 Purpose

This assignment is designed to introduce you to a medium scale Java system, to test your ability to design and implement Java code, to test your ability to design object-oriented programs, to give you experience with appropriate Java programming and debugging tools, and to make a program that is slightly cool and possibly even useful.

2 Overview

Solar is a program that simulates the evolution of a solar system. A solar system here is defined as a large, massive star surrounded by many smaller planets orbiting the star. Each planet consists of a mass with an initial position and an initial velocity.

Determining where all of them will be in t seconds is known as the **N-Body Problem**. It's very challenging to calculate exactly how the planets should move, and it would take too many computer resources than we have to attempt to do it, so you're going to be approximating their trajectories as best you can. Given this, slightly different approximations could result in very different long-term results, so there is no *right* answer. Your simulation should be reasonably accurate - given a star with the sun's mass and a planet in earth's position, it should take around one simulated year to orbit the sun.

In addition to just moving around space, planets sometimes collide. Your simulation will have to handle this. However, it'll be done very simply - if two bodies collide, they turn into one object which has their combined mass and velocity.

3 Simulating the Solar System

The following is an overview of how to simulate the solar system. The physics behind it, especially the equations you have to use, is very important, so it has been placed in the Physics Appendix (Appendix A). Read through these steps first, then go to the Appendix to understand the equations that you'll need to implement the project.

The simulation works by calculating how all of the objects should move one step at a time by assuming that, for the duration of that step, the objects will move in a straight line. If your time step is small, like one second, then you will have a really accurate simulation. However, it would take forever to simulate even one day. If your time step is too large, like one year, then you'll get a wildly inaccurate simulation, since you'll assume that the earth would move in a straight line for one year, which isn't anything close to reality. A good time step to use is around 20 minutes, but you can change this to suit your needs.

Note: The sun acts just like any other object in the simulation - it, too, is affected by the gravity of the planets.

For each time step:

1. For each object, calculate the forces that all the other objects in the simulation exert on it, and add all of these forces together. Each object interacts with each other object once, through the force of gravity. See section A.1 of the appendix for details.

2. For each object, change its velocity and position based on the force being applied to it and the length of the time step. See section A.2 of the appendix for details.

3. Once all the objects positions have been updated, check if there are any collisions. If there are, each two colliding objects should combine to form one object. See section A.3 of the appendix for details.

Then, simply repeat!

4 The Initial State of the System

We'd like you to generate for us a solar system of 100 objects plus one star in the center. The system should be different each time you run your program, given the following constraints (note that this will make more sense once you understand the physics behind the project):

- The star is at $(0, 0, 0)$, and has the mass and radius of the sun. The mass is 1.989×10^{30} kg, and the radius is $695,000,000$ m.

- The masses of all 100 objects should be approximately $1.18 \times 10^{25} \pm 1\%$ kg. The easiest way is to give them all the same mass, but they can vary if you like.

- Each object should initially be in a stable orbit around your star with a radius greater than Mercury's orbit and less than Mars'. That means the distance between the star and each object should be between $57,910,000,000$ m and $227,940,000,000$ m. See A.4 for how to create a stable orbit.

- Space *is* three-dimensional. It is acceptable for all your z coordiantes to be zero, making your solar system, in effect, a flat disk. However, if you want a slightly more exciting simulation, you can add a random component to the z axis - you can vary it from around -60,000,000,000 meters to +60,000,000,000 meters. This would also be a good way to make sure that z coordinates are taken into account in your calculations.

Note that you might want to try your program on simpler cases before doing the full simulation. For example, a single object, or a pair of objects opposite each other, should yield a stable solution; setting an initial velocity of 0 for an object will test its gravitational attraction to the sun (it should fall straight in). It's also probably a good idea to use the same initial configuration when debugging rather than a different one each time you run.

5 The Design

We'd like you to pitch your design to us much like you would at a medium-sized software company when the idea is in its initial small public presentation stage (e.g. presenting to your "team" of 3-6 people). This is less formal than a full description of exactly how the code will work, but it still contains a complete picture of the project which another person can easily understand. To be specific, we'd like to see:

- A complete class diagram, showing all classes and interfaces, showing run-time relationships (e.g. class A creates class B, and class C has a collection of class A instances, etc.)
- Inheritance and "implements" relationships among the classes and interfaces
- Method names and signatures (e.g. `void foo(Atype a, Btype b, int bar)`) with a short description for each (a sentence at most)
- Each class and interface should have a few sentences explaining its purpose and responsibilities
- Consistent, easily comprehensible symbols, which need not be perfect UML, or even like UML, but one should be able to understand what they mean without any additional explanation (use a legend if you think it would help). UML will come later.
- Descriptions of test cases you'd like to use to test various aspects of your code as you go and at the end. Be fairly specific and be sure to include the expected result with each test case.

There are also a lot of things we *don't* want to see. These include:

- A poor diagram, which you hope to make up for with a better verbal description
- Any filler, fluff, or verbosity that doesn't improve the diagram. If it takes four words to explain something really simple, only spend four. Save your words for the things which actually have complexity, but then spend them generously. We are not grading based on word count.
- Any other materials or additions. Your diagram should be awesome enough to stand on its own.
- Differences between your diagram and your "current design". Your diagram should be legible and up-to-date when you present it.

We really want you to get *a lot* out of this design check.

You'll present your design to a TA, and then receive interactive feedback. You can also feel free to ask questions. The total time for this design check is limited to 10 minutes. Your grade for this design check will be based on 2 things: the quality of your design and test cases coming into it, and how confident the TA is that your project is "a go", i.e. if the TA were your manager at a software company, and you pitched this design, would he or she be convinced to commit to doing it?

Details about when exactly you'll present will be posted to the website and/or mailing list. For now, just know that it will be on or after the date listed at the top.

6 Using the SolarDraw GUI

The goal of the project is to convince the TAs, as well as yourself, that your code works. Probably the easiest and most effective way to do this is by being able to physically see your solar system. To help with this, we've provided a GUI program, called the SolarDraw GUI, which will render your solar system for you using OpenGL.

6.1 Using SolarDraw in your code

Javadocs are available from the website for detailed info on all the functions SolarDraw supports, but here's all you should need to know to use it in your code.

First, to use the GUI, you must import this package:

```
import edu.brown.cs.cs032.solardraw.*;
```

You should make your top level class implement the SolarDraw.Control interface. This means you need to implement the following methods:

```
interface SolarDraw.Control
{
  double getTime();
  double getUPS();
  void saveCalled();
}
```

For the moment, you can ignore `saveCalled()`; you'll need it in Galaxy. For now, just define an empty function. `getTime()` should return the current time of your simulation in seconds. `getUPS()` is optional, see the next section for details.

Now, in your top-level object, create a `SolarDraw` instance like so:

```
SolarDraw solarDraw = SolarDraw.Factory.createSolarDraw(this);
```

You have to tell SolarDraw about all the objects that you want it to draw. You do this by registering them:

```
solarDraw.registerObject(myObject);
```

myObject is an object which has to implement the SolarDraw.Object interface, which is pretty self-explanatory:

```
interface SolarDraw.Object
{
  double getMass();
  double getRadius();
  double getX();
  double getY();
  double getZ();
}
```

The GUI will constantly call these functions whenever it re-draws them. Once you've registered an object, you don't have to worry about it anymore.

When you're done adding your objects, you want to start rendering your solar system. To make the GUI show up, call:

```
solarDraw.begin();
```

You should only call this method once!

To unregister an object (in other words, remove it), simply set its mass to zero, and the GUI will automatically unregister it. You will have to do this when two objects collide.

IMPORTANT NOTE: The SolarDraw GUI currently has a bug. If you've only registered one object with it, it's not going to display it; you're going to see a blank screen. When testing to see that the SolarDraw GUI works, make sure that you add at least two objects. This is due to a problem with its automatic zooming.

6.2 Displaying UPS (updates per second)

Speed is not an issue in Solar. We don't expect your simulation to be really fast. However, in the next project, Galaxy, you will improve your project's speed. Because of this, you might want to be able to see exactly how fast your program runs. The GUI has a place to display UPS (updates per second), which will tell you how many times the computer is recalculating the forces, velocities, and positions of the objects in your system.

We've given you a class, UpdateCounter, to help you calculate UPS. To use it, create an instance of UpdateCounter, then call update() on the UpdateCounter once for every pass through your main calculation loop. Then, to actually display the UPS, define the following method in your top-level object:

```
public double getUPS() {
    return myUpdateCounter.getUPS();
}
```

where myUpdateCounter is your instance of UpdateCounter.

Again, speed is not an issue in Solar, so feel free to ignore `UpdateCounter` completely. However, to keep Java happy, you will still need to define `getUPS()`. Simply have it return a dummy number if you don't care about UPS.

That being said, there is one very important optimization that will make your code run about five times faster - take a look at section 7.

6.3 The Interface

The GUI has a few features which are useful. These are the controls that it supports:

Page Up	Zoom in
Page Down	Zoom out
Arrow Keys	Rotate by 30 degrees
Home	Reset the GUI
Hold Right Mouse Button	Rotate the screen
Shift + Hold Right Mouse Button	Pan the screen

A few notes:

- There is no key or button to close the SolarDraw window. Just use your window manager's standard close button.
- The SolarDraw GUI likes to recenter its view, sometimes in a sudden way, so don't be alarmed if things are a little jerky.
- The GUI doesn't draw things necessarily to scale. If you zoom in and see your planets colliding, but your collision code isn't verifying that, they might actually be far apart enough to not collide, but the GUI is exagerrating the size of the spheres to make them easier to see. So, when checking collision detection code, don't rely on the GUI, but use debugging print statements instead.

If you have any problems with the GUI, please report them to `cs032tas@brown.edu`. If you have any suggestions as to how to make the interface more informative or more helpful, please tell us – we are looking for suggestions.

As we mentioned before, we won't be grading for speed. However, there are a few important tips that won't take much effort to implement and can get your Solar to run at up to 1000 UPS for 100 objects.

Make sure to tell us whichever one of these you've done in your `README`. If you've done any other optimizations which you think improved your project's speed a lot, be sure to mention those as well.

7 Java Math

We've said that speed doesn't matter. However, there is an important thing to know when doing your project - **do not use the Math.pow function to square numbers**. In other words, if you have a `double x`, the following code:

```
Math.pow(x, 2.0);
```

is about **five to six times slower** than doing

```
x * x
```

Whenever you square a number, such as in distance calculations, make sure you don't use Math.pow. Since this is done several times in the force calculation, which gets executed thousands of times a second, this will be a gigantic speed upgrade.

The reasoning behind this is that Math.pow is a very generalized function which can handle any exponents, including fractional and negative ones. It doesn't check the special cases of squaring or cubing a number, so it'll take the same amount of time to do that as to raise a number to the 3.43284th power, which is a very inefficient calculation.

8 Demo

To run the demo, use this command:

```
/course/cs032/demos/solar [planets]
```

The demo takes a single argument, the number of objects that you want in the solar system. If you do not specify the number of objects you want, the demo will default to 100.

9 Getting Started

We encourage you to use Eclipse to develop this project. Here are the steps you need to take to get all set up:

1. Copy the stencil code to your home directory:

   `cs032_install solar (or /course/cs032/bin/cs032_install if cs032 is not in your path)`

2. Run eclipse: `eclipse` . If it says file not found, type in the full path: `/contrib/bin/eclipse`

3. Create a workspace if you haven't already.

4. Go to File → New Project. Choose Java Project. Give it a name.

5. In the resulting dialog box, select "Create Project from Existing Source". Navigate to the solar folder which you created in Step 1. Click Next.

6. Now we have to add libraries to get the SolarDraw GUI to work. Go to the "Libraries" tab. Click "Add External Jar." In the file selection dialog box, choose:

   `/course/cs032/lib/solardraw.jar` .

7. Add another external jar: `/pro/java/linux/software/jogl/jogl-1_0_0/lib/jogl.jar` .

8. Click Finish. The project will now be created and will show up on the left.
9. Right-click on this project. Go to Run As → Run Configurations. Go to the ”Arguments” tab. Under ”VM Arguments,” enter:

   `-Djava.library.path=/pro/java/Linux/software/jogl/jogl-1_0_0/lib`
10. Click Close. The project should now be ready to run. You can run it by right clicking on SolarMain.java located by in the javasrc $->$ edu.brown.cs032.solar package.

10 Handin

Your grade will be based on your design check and your handin. Your handin should be very easy to compile and run. Your code should be easy to read.

Make a `README` file which explains any quirks in your code, as well as any special instructions about running your program. You should also mention any design decisions you made which you found troubling. The file *must* be called `README` without any extensions, or else the handin script will not work!

Handin with the following command from your solar directory:

```
/course/cs032/bin/cs032_handin solar
```

A Physics Appendix

This contains all the equations you'll need to get your project running. **Important note:** All of this math will be done in kilograms, meters, seconds, and Newtons. The equations will only work with these units, so make sure that you are consistent in your code with how you represent them.

A.1 Calculating Forces

A force is made up of the magnitude of the force, or how powerful it is, as well as the direction the force is acting in. For the force of gravity, the magnitude is determined by this formula:

$$F = \frac{Gm_1m_2}{d^2}$$

where m_1 is the mass of the first object, m_2 the mass of the second, d the distance between the two, and G the **universal gravitational constant**, which is

$$G = 6.674 \times 10^{-11}\frac{Nm^2}{kg^2}$$

.

The distance d is determined in a similar way to that in two-dimensional space, except you have to account for the z coordinates as well:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Now you have a magnitude. We need a way to represent the direction of our force. However, rather than thinking of the force as one magnitude and one direction, it's much easier to think of a force as three separate forces - one acting only on the x-axis, one on the y, and one on the z. If we split our force in this way, then combining forces becomes very easy - you simply add the components together separately:

$$\begin{aligned} F_1 &= (F_{x_1}, F_{y_1}, F_{z_1}) \\ F_2 &= (F_{x_2}, F_{y_2}, F_{z_2}) \\ F_1 + F_2 &= (F_{x_1} + F_{x_2}, F_{y_1} + F_{y_2}, F_{z_1} + F_{z_2}) \end{aligned}$$

This is vital, since you'll be summing hundreds of forces together.

What we need is a way to break apart a force into its three components, one on each axis. This is done using the following formulas:

$$F_x = F\frac{x_2 - x_1}{d}, F_y = F\frac{y_2 - y_1}{d}, F_z = F\frac{z_2 - z_1}{d}$$

Make sure you keep the order of the subtraction straight. If you accidentally subtract the second object from the first, then they will repel each other, and you'll get a very strange-acting solar system.

So, to calculate all the forces acting on an object at a given time, you:

1. Calculate the force that each object is exerting on it separately.
2. Split this force into its three components.
3. Add all of the forces together by summing their components separately.

To see how to affect your object based on the force acting on it, see the next section.

A.2 Approximating the Motion

For each object, you now have three forces acting on it, one for each axis. Using these forces, we can update each dimension separately.

A force acts on an object over a period of time. The length of this time period is the time step that you've decided to use for your simulation. Make sure that if, for example, you decide to use a time step of 20 minutes, you plug 1200 into all these equations, since they all assume you're using standard units, which, for times, are seconds.

First, we have to determine the acceleration of the object over this time. You probably remember the equation

$$F = ma$$

where F is the force along one of your axes calculated from the previous section, and m is the mass of the object. We just re-write this to solve for a:

$$a = \frac{F}{m}$$

This is where the approximation part comes in: a should actually be a continuous function during the time period, but we're going to pretend it's constant, which makes everything a lot simpler.

The simplest method to find the new position and velocity is called **Euler Integration**. Using this method, the new velocity is

$$v = v_0 + at$$

and the new position is

$$p = p_0 + v_0 t + \frac{1}{2}at^2$$

where v_0 and p_0 are the object's old velocity and position and t is the length of the time step.

You have to perform these calculations three times - one for each of the forces acting on your object. Once you have the new position and velocity, simply update the object with them, and you're done!

Note: Make sure that you don't move an object until all of the forces on all of the objects have been calculated. All these forces are supposed to be applied *simultaneously*, and if you move an object prematurely all the force calculations involving it are going to be off.

If you feel that you want to try a more accurate motion approximation for some extra credit, see section B.2 for some info on methods you can look up.

A.3 Collisions

To make things simpler, all of your objects are going to be spheres. So, in addition to a position, your objects have a radius. When running your simulation, you have to check whether two objects are actually touching each other. This is a simple check; two objects are colliding when:

$$d < r_1 + r_2$$

where d is the distance between them and r_i is the radius of each object. This checks if they are touching exactly. However, the simulation is just an approximation, and if objects are moving very quickly and your time step is too large, they can actually skip over each other when they should be colliding. To attempt to account for this, you should scale the right side by a factor k:

$$d < k(r_1 + r_2)$$

This effectively treats the spheres as if they were bigger, which increases the chance of them colliding. Make sure that k is not too large, though. If it is, objects which aren't actually colliding will behave as if they are. A k somewhere between 5 and 20 seems to work well, but try different values yourself and see what works well.

When two objects collide, they should become one object. The new object should have a mass which is the sum of the masses of the original two objects:

$$m_{new} = m_1 + m_2$$

The object's new position should be the center of the collision, which is just the average of their positions:

$$x_{new} = \frac{x_1 + x_2}{2}, y_{new} = \frac{y_1 + y_2}{2}, z_{new} = \frac{z_1 + z_2}{2}$$

The object's velocity should be the **weighted average** of the two original velocities, weighted by the objects' masses. The idea is that the heavier object will influence the new object's velocity more.

$$v_{x_{new}} = \frac{v_{x_1} m_1 + v_{x_2} m_2}{m_1 + m_2}$$

$$v_{y_{new}} = \frac{v_{y_1} m_1 + v_{y_2} m_2}{m_1 + m_2}$$

$$v_{z_{new}} = \frac{v_{z_1} m_1 + v_{z_2} m_2}{m_1 + m_2}$$

And that's all you need to set the new object on it's way.

A.4 Establishing a Stable Orbit

Once you've selected a random initial position for an object, as well as a mass, you need to initialize its velocity so that it doesn't simply fall straight into the sun. **Note:** These calculations assume the sun is at position $(0, 0, 0)$.

First, calculate the force of attraction between the object and the star. This is done the same way as in A.1. The speed of the object for a stable orbit is going to be

$$v = \sqrt{\frac{Fd}{m}}$$

where F is the force of attraction, d is the distance between the object and the sun and m is the object's mass.

The velocity components are going to be:

$$v_x = \frac{vy}{d}, v_y = -\frac{vx}{d}, v_z = 0$$

Note that x and y are switched in the equations.

To make the simulation more interesting, you should perturb the v_x and v_y values at random by a factor up to 10% of its value. This will give you slightly elliptical orbits and encourage collisions between objects.

For some extra credit, you can add moons to orbit the planets in your solar system - see section B.1 for more details.

B Extra Credit

B.1 Moons (3 points)

Once your solar system is running, you might think that it is a bit unexciting - a hundred planets, and no moons at all? For extra credit, add any number of moons to some or all of your planets. A hint: this can be done in a very similar way to the way that you set the planets themselves around the orbit of the sun.

If you implement moons, you're allowed to have less than 100 planets - just make sure that the number of planets plus the number of moons is at least 100.

B.2 Better Motion Approximation (5 to 8 points)

Once you've got Euler Integration working and everything looks good, if you're feeling brave, you can try replacing Euler Integration with some other method which is more accurate. For example, you might try **Verlet Integration**, which will reduce the error in our approximation, but adds the complexity of needing to store 2 positions, a current and a previous, and omitting the velocity altogether. If you're interested, check out the Wikipedia article on the subject (it's quite good). One caveat: avoid the so-called "Velocity Verlet" method, as it requires you to make multiple passes over all objects to calculate acceleration twice, and frankly, you won't have the CPU cycles to do that.

Another more accurate approximation method is **Runge-Kutta**. It will take a bit of research, but for a really accurate simulation, **RK4** is a premium choice. You're also free to implement any other well-known approximation method, provided its results are at least as good as Euler Integration for this type of problem.

If you choose one not mentioned here, consult with a TA first, and be sure to mention it in your `README` along with a pointer to a good source on the topic.