Galaxy

Assignment Out:	Feb 01
Design Due By:	Feb 8
Code Due:	Feb 13 11:59pm

1 Purpose

The company that you are working for has liked your Solar design and ideas so much that they want to use to produce a system to sell the the government to predict asteroid collisions. Unfortunately, there are millions of potential asteroids. To accomplish this you will have to extend your Solar code (which you still must finish first) to accommodate.

For this project, you'll build a `Galaxy` simulator that will

- handle $1,000$ objects well, and up to $10,000$ objects to some degree
- save and load its status using an XML file
- balance the time vs. accuracy trade-off intelligently, using adaptive time stepping.

The goal here is to give you first-hand experience at extending code beyond its original specification. The difficulties you run into as you build upon your `Solar` project should give you a sense of the major issues in reusing code, so you can start thinking about how to avoid them. Ultimately, we want to help you write extensible code from the beginning.

2 What's New in Galaxy

2.1 New Force Calculation Algorithm

Simulating $1,000+$ objects requires a different algorithm than does simulating 100. Try running your `Solar` project on a huge number of masses to see for yourself.

You're free to use any approach you like to allow your `Galaxy` simulator to handle $10,000$ objects, whether you invent it yourself or find it through research. However, we strongly recommend that you use the **Barnes-Hut Algorithm**.

The intuition behind it is that to find the effect of a group of distant objects on a certain planet, we can treat the group of planets as a single planet without losing too much accuracy.

The algorithm actually consists of two components – an octtree implementation as well as an algorithm that is applied to the octtree. Rather than try to explain the algorithm here, we've decided to give you links to websites that do just that. These can be hard to read and understand,

so for that reason we're going to go over the algorithm thoroughly in the Galaxy help session. It's much easier to explain in person, and with examples, so if you have trouble understanding the websites, be sure to come to the help session!

That being said, check out:

`http://www.flipcode.com/archives/Introduction_To_Octrees.shtml`

for an explanation of octtrees. Note that, since your planets are constantly moving, you're going to have to recalculate your octtree every frame. You could try recalculating it only every few (2 to 4) frames for some extra speed, since if your time step is small enough it should still be reasonably accurate, but this isn't as accurate and is not required.

`http://www.cs.berkeley.edu/~demmel/cs267/lecture26/lecture26.html` provides a good explanation of the Barnes-Hut algorithm itself, as well as a small section on octtrees. Note that these links are available on the course website as well, in the Algorithms / Docs section.

If after the websites and the help session you still have trouble with the algorithm, come to TA hours and we'll be glad to help you understand it.

2.2 New Collision Detection Algorithm

Note: This section makes more sense once you understand the octtree algorithm.

Now that you have an octtree set up for the new force calculation, you can also make collision detection a lot more efficient. The reasoning behind collision detection is as follows: if a particluar node in the octtree does not intersect with a planet, then nothing in that node can possibly intersect with it either, so you can ignore the entire node.

So, intersect every planet in your solar system with the root octtree node. The collision between a planet p and a node is done recursively as follows:

- If the node is a leaf node (has no children octtree nodes), and it has no planets in it, then there's no collision.
- If the node is a leaf node and has a planet, then check collision between p and that planet in the same way as in Solar.
- If the node isn't a leaf node then, for every child node:
  - See if p intersects the child node's bounding box. You can do this by adding the planet's radius to every dimension of the child node's bounding box and then seeing if that box contains the planet's center.
  - If p intersects the child node, then recursively collide p with that child node. If any collisions happened, then return true. Otherwise, check the other child nodes.
  - If p doesn't intersect the child node, then just continue to the other child nodes.

2.3 XML

Your project should be able to read and write the state of the simulation to a file using our specified XML format. You're welcome to share your test input files (and their resulting output) with other CS32 students to test your code.

The specification for the XML we want you to read and write consists of only four tags.

SOLAR the root tag for the document (i.e. the outermost tag, which holds all the content inside). It has one required attribute:

- **TIME** The number of simulated seconds the simulation has run at the moment the XML file is saved. When you first create a galaxy, start with its `time` attribute as 0. If you save its XML later after simulating it, update the time.

OBJECT the only tag allowed directly within a `solar` tag. Every object in the system is described by an `object` tag. Its attributes are:

- **NAME** some name you give the object
- **MASS** the object's mass
- **RADIUS** its radius

Within the `object` tag there must be exactly two other tags, `position` and `velocity`.

POSITION always found nested in an `object` tag, it specifies an object's position, with the following attributes:

- **X** object position's X-coordinate.
- **Y** object position's Y-coordinate.
- **Z** object position's Z-coordinate.

VELOCITY always found nested in an `object` tag, it specifies an object's velocity, with the following attributes:

- **X** object velocity's X-coordinate.
- **Y** object velocity's Y-coordinate.
- **Z** object velocity's Z-coordinate.

Here's a sample XML document your program should be able to handle.

```
<SOLAR TIME='0.0'>
  <OBJECT NAME='Object_0' MASS='6.93802901065139E23' RADIUS='1271511.650378473'>
    <POSITION X='-2.966473042307841E9' Y='7.289090278627592E8' Z='772.2514875' />
    <VELOCITY X='-161.88780633549223' Y='-658.841906212116' Z='0.0' />
  </OBJECT>
  <OBJECT NAME='Object_1' MASS='5.7352863630113265E23' RADIUS='1193328.48932346'>
    <POSITION X='-1.8524168835642315E10' Y='4.746119148457207E9' Z='33685.441' />
    <VELOCITY X='-97.34967716226407' Y='-379.95713959052785' Z='0.0' />
  </OBJECT>
</SOLAR>
```

For everything you need to know to do XML Saving and Loading, take a look at appendix A.

2.4 Adaptive Time-Stepping

Having a fixed time step, as in Solar, leads to many problems. The time step could be fine for one solar system, but way too small for a very expanded galaxy, and way too large if you have a very small solar system. Furthermore, objects will tend to go through each other if they get very close, and have inaccurate orbits.

The choice of the proper time step is dependent on a number of factors such as the speed of the objects and their distance from one another. For every step of the simulation, the following criteria should be met:

- To make sure the simulation is accurate, any one planet should not move more than 1/100th of the average distance between two objects.
- To make sure that collisions are detected, any one planet should not not move more than half of the way through any other planet.

You have to calculate a time step every frame that makes sure every planet satisfies these criteria. To do this you must limit your time step in two ways.

- You must keep track of the average distance between all objects as well as the maximum velocity of any object. You then must make sure that the fastest object will not travel more than 1/100 of this distance. So,

  $$t_{max} = d_{avg}/100/v_{max}$$

  The timestep cannot exceed t_{max}, or your simulation will be too inaccurate.

- For every pair of objects, you have to find out how much time it would take for them to go halfway through each other. An object going at velocity $\overrightarrow{v}$ travels $\overrightarrow{v}t$ meters per second. So, for every pair of planets you look at, see how long it would take them to travel $d - \frac{r_1+r_2}{2}$ meters, where d is the distance between them and r_1 and r_2 are their radii. You can either:

– Find the magnitude $|\overrightarrow{v}|$ of each of the planets velocities:

$$|\overrightarrow{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

add them together, and divide the distance between them by this amount.

$$t_{max} = \frac{d - \frac{r_1 + r_2}{2}}{|\overrightarrow{v}_1| + |\overrightarrow{v}_2|}$$

– The previous method isn't too accurate, since the two planets are not necessarily always moving directly towards each other. You will always err on the side of caution, so it is acceptable, but to be more efficient you'll have to calculate the **projection** of each planet's velocity onto the vector representing the distance between them. This isn't necessary, but if you're up for it:

First, find the distance vector between them:

$$\overrightarrow{d} = \overrightarrow{p_2} - \overrightarrow{p_1}$$
$$(d_x, d_y, d_z) = (p_{2x} - p_{1x}, p_{2y} - p_{1y}, p_{2z} - p_{1z})$$

where p is a planet's position.

Then, find the **scalar projection** s of each planet's velocity onto the distance vector:

$$s = \frac{\overrightarrow{d} \cdot \overrightarrow{v}}{|\overrightarrow{d}|}$$

where

$$\overrightarrow{d} \cdot \overrightarrow{v} = d_x v_x + d_y v_y + d_z v_z$$

This will give you exactly how much each object will move in the direction given by $\overrightarrow{d}$. It will be the same as the previous method if the object is moving in that direction, smaller if it's in a different direction, and negative if it's in the opposite direction.

Compute s_1 and s_2 and use them in place of $|\overrightarrow{v}_1|$ and $|\overrightarrow{v}_2|$ in the previous method to calculate t_{max}.

Note: You will have to use $\overrightarrow{p_1} - \overrightarrow{p_2}$ for $\overrightarrow{d}$ when calculating s_2, because you want to see how fast the second planet is moving towards the first, so you want your direction to point accordingly. You don't have to completely recalculate $\overrightarrow{d}$ though – you can just negate the one you used before, or, more simply, use the same $\overrightarrow{d}$ but negate s_2.

t_{max} now represents the maximum timestep in your simulation to ensure collision between this pair of planets.

You will end up with many different values for t_{max}: one to ensure accuracy, and one for each pair of planets to ensure collisions. Pick the smallest of these in order to satisfy all the criteria.

Note: Since you'll implement an octtree, you won't actually be iterating between all pairs of objects anymore. Doing so just for the sake of picking a time-step will slow the simulation down an unacceptable amount. Instead, you should be doing these calculations as you're doing your force calculations. If you end up doing a force calculation between an object and a leaf node, then you can do these calculations between the object and the object in the leaf node directly. If, however, your program decides that a node is far enough away to be considered as a single object, then:

- For the average distance calculation, use the node's center of gravity to calculate the distance, and weigh it by the number of objects in that node.
- For the collision ensuring calculation, if you skip an entire node, then your object is too far to be at risk with colliding with anything in that node, so you don't have to limit the time-step at all.

The second part of these calculations seems a bit inefficient – you can have 1000 objects all millions of kilometers from each other, but then you have another two which are just a few thousand kilometers away. This will result in the entire simulation slowing down for just those two objects. There is a way around this.

1. Pick a timestep based only on the average distance calculation.
2. If two objects are too close to each other, and are in danger of missing a collision, then do multiple force calculations for just those two objects. For example, you might split the timestep into two. You would first do a force calculation, update the two object's new positions and velocities, then do another force calculation using their new positions and velocities and update them again. If they are close enough together, this will be considerably more accurate. Note that you should probably do collision detection after each interval.

2.5 A few more things

- `Galaxy` is 3-dimensional. When you set up your scenes, make sure you spread things out in the z-dimension as well. Feel free to be creative: try simulating two galaxies colliding, a galaxy with a huge black-hole (i.e. a massive object) in the center, or perhaps two black-holes orbiting each other in the center of a galaxy. Of course, when you test initially, it's a good idea to set up some simple situations whose behavior you can predict by hand. Also make sure you have a test case which clearly demonstrates your adaptive time stepping.
- If an object 'escapes the system' (that is, gets very far away from everything else, say 10 times farther away than the initial maximum distance between two objects), you're allowed to delete it and forget about it.
- Collisions happen the same way as in Solar – when two objects collide, they glue together and form one object.
- Your program should take command line arguments. It should at least be able to take the name of the XML file containing the initial configuration. You can add other arguments if

you want (e.g. parameters that control your simulation and thus allow you to experiment without recompiling).

- While displaying Updates Per Second was not required for Solar, speed does matter for Galaxy, so make sure you get that working. See the Solar handout for details on how to do this.

3 Design & Testing Plan

Please have a design and testing plan ready for your design meeting. Look out for a post on the website about how to sign up for a design check. Unlike the Solar design check, we expect you to come with an electronically generated UML diagram. We recommend using Argo, which you can launch by using the following script:

- `/course/cs032/pub/cs032argo`

We also expect to see a write-up of test cases similar to what you did for checkpoint 1 in lab01. We want to see JUNIT test cases (pseudocode please) for all the functions that generate output, including any supporting math or physics functions that you plan to write. Also, include in your test plan test cases that test the visual output of your program.

This is worth 10% of your grade.

4 Demo

The demo can be run in one of two ways:

- `/course/cs032/demos/galaxy -random <numplanets> [optional timestep]`
  This will create a galaxy with a given number of randomly placed planets. If an optional timestep is specified, it will run using that timestep instead of doing adaptive timestepping.
- `/course/cs032/demos/galaxy <xmlfile> [optional timestep]`
  This will load the given xml file. `optional timestep` works the same way.

5 Getting Started

Galaxy builds directly off of your Solar code. So, once you've completed Solar, copy all your solar code to another directory and follow the same steps as in Section 8 of the Solar hand-out to set up eclipse.

6 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.

Handin with the following command, which you should run from the directory which contains `build.xml` so we'll get everything.

```
/course/cs032/bin/cs032_handin galaxy
```

A XML Primer

Here's a quick overview on how to get started with XML in java. In this section, we're going to write code to load and save files in the following format:

```
<World name="Quuxo">
  <Country name="Footrania">
    <State name="Barabara" size="14" />
  </Country>
  <Country name="Bazonia">
    <State name="Quxa" size="199" />
  </Country>
</World>
```

A.1 Theory

XML is represented as a tree structure. It consists of nodes – things like `World`, `Country`, and `State`.

Each document has one root node – in this case, the `World` node. Each node has zero or more children – `World` has two children nodes, the two `Country` nodes. Each `Country` node has one `State` node as a child, and each `State` node has no children.

Each node has one or more attributes – `World` nodes, in this example, have a `name` attribute, while `State` nodes have both `name` and `size` attributes. Note that each attribute is just a String – even though `size` represents a number, the XML file doesn't know this, so your code must convert it to an Integer or a Double.

A.2 Imports

First, to make sure you have everything, import the following things in the class file that will handle your saving and loading:

```
import org.w3c.dom.*;
import org.xml.sax.*;
import java.io.*;
import java.text.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
```

A.3 Loading

Loading is done using DOM, which stands for **Document Object Model**[1] DOM parses the entire XML file at once and provides ways to access the internal tree structure.

A.3.1 Loading the File

First, you want to load your entire XML file into a `Document` variable. This is done as follows:

```
Document document;
String XMLFileName = "galaxyfile.xml";
try {
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  DocumentBuilder builder = factory.newDocumentBuilder();
  document = builder.parse( new File(XMLFileName) );
} catch (Exception e) {
  //Some error occurred -- print it out and exit
}
```

It's rather convoluted, and it isn't too important to know exactly what each of those lines does or exactly what exceptions they return, so you can just copy that straight into your loading code. The end result is that `document` contains your parsed XML file.

A.3.2 Getting the Root Node

Each node in the tree structure is represented by the `Node` class. To get the root node – in our case, the `World` node, you do:

```
Node world = document.getDocumentElement();
```

A.3.3 Getting Node Names

To get a node's name, do `Node.getNodeName()`. You probably want to make sure that each node you are accessing has the right name; if it doesn't, the XML file you're reading is an invalid format, and you should return an error.

```
if (!world.getNodeName().equals("World")); //Error - do something
```

A.3.4 Getting Node Attributes

You can get a node's attributes as a map by doing `Node.getAttributes()`. This returns a `NamedNodeMap`. To access an attribute by name, you do `NamedNodeMap.getNamedItem()`. This actually returns a `Node`, so you have to call the `Node.getNodeValue()` function to get the actual attribute as a String. Note that node names and attributes are case-sensitive.

[1] There is an alternate way to load XML files, called SAX, but DOM is simpler to implement and is all that will be covered here.

```
NamedNodeMap attributes = world.getAttributes();
String worldName = attributes.getNamedItem("name").getNodeValue();
```

A.3.5 Getting Node Children

Now that we have the world name, we want to see what countries are in the world. We do this by getting the world node's children nodes and iterating through them:

```
NodeList countryNodes = world.getChildNodes();
for (int i = 0; i < countryNodes.getLength(); ++i)
{
  Node country = countryNodes.item(i);
  System.out.println("Country name = " + country.getNodeName());
}
```

A.3.6 Text Nodes

An important thing to note about XML – whitespace does matter. That means that there is a difference between

```
<World name="Quuxo">
  <Country name="Footrania" />
</World>
```

and

```
<World name="Quuxo"><Country name="Footrania" /></World>
```

The first one has text nodes added in. So the children in the `World` node in the first example are actually:

1. A `Text` node whose value is `"\n  "`
2. A `Node` whose name is `"Country"`
3. Another `Text` node whose value is `"\n"`

It's easy to test whether a node is a text node or a real node:

- `Node.getNodeName()` will return `"#text"` if the node is a text node.
- `Node.getNodeType()` will return `Node.TEXT_NODE` if the node is a text node.

You should check for this in either way and ignore text nodes if they come up.

A.3.7 Example

Now you can do all the things with `country` that you can do with `world`, including iterating through its children. For example, if you want to get the first country's first state's name and size, you would do:

```
NodeList countryNodes = world.getChildNodes();
Node firstCountry = countryNodes.item(1); //skip the text node
NodeList stateNodes = firstCountry.getChildNodes();
Node firstState = stateNodes.item(1); //skip the text node
NamedNodeMap stateAttrs = firstState.getAttributes();

String stateName = stateAttrs.getNamedItem("name").getNodeValue();
int stateSize = Integer.parseInteger(
  stateAttrs.getNamedITem("size").getNodeValue());
```

Note that `NodeList.item` returns `null` if the index is out of bounds. Also note that to get `stateSize` as an `int`, we had to call `Integer.parseInteger()` on the `String` that `getNodeValue()` returned.

A.4 Saving

Saving is done by creating a `Document` element, creating a root `Node`, and adding children to that root node one by one.

A.4.1 Creating the Document

Creating the `Document` is done as follows:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.newDocument();
```

Remember to wrap all that in a try/catch, since they can each throw various exceptions.

A.4.2 Creating Nodes

You have to use the `document` to create any nodes in the XML file. This is done using `Document.createElement()`:

```
Element world = document.createElement("World");
```

`Element` is a subclass of `Node` which allows for attributes to be set.

A.4.3 Setting Node Attributes

This is done by using `Element.setAttribute()`:

```
world.setAttribute("name", "Quuxo");
```

where the first parameter is the attribute name and the second is the value.

A.4.4 Appending Children

Both `Documents` and `Elements` can have children appended to them. A `Document` can have only one child – this is the root node. A `Element` can have any number of children.

So, the code to create the XML file we've been using as an example would be:

```
Element country = document.createElement("Country");
country.setAttribute("name", "Footrania");
Element state = document.createElement("State");
state.setAttribute("name", "Barabara");
state.setAttribute("size", "14"); //note that 14 is a String, not a number.
country.appendChild(state);
world.appendChild(country);
/* .
   . Similar code for the other country, "Bazonia".
   .
 */
document.appendChild(world);
```

A.4.5 Saving the Document

Now that you have a `Document` representing your desired XML file, you have to actually save this to a file on disk. This is also done in a very convoluted way, much like loading an XML file, so you can just copy the following code and use it as-is:

```
String fileName = "output.xml";

TransformerFactory tFactory =  TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer();
transformer.setOutputProperty(OutputKeys.INDENT, "yes");

DOMSource source = new DOMSource(document);
StreamResult result = new StreamResult(new File(fileName));
transformer.transform(source, result);
```

This will save the `document` to `"output.xml"`. This can also throw some nasty exceptions, so be sure to wrap it in a try/catch.