PSA 8: Bodies in Motion

Read all instructions in this document before starting any coding!

Due: 12/4/2017, 11:59pm

In this assignment you will be simulating multiple bodies in motion acting upon each other's forces in a 2-D plane and making a small movie of the movement. To do this, you will implement a class and use physics to create the simulation - but don't worry if you don't know physics! **You don't have to understand the physics to do this PSA!** We have explanations of what you need for each method. And if you *do* want to understand a bit of the physics behind the code, we have explanations for everything in this file!

For this assignment, you are encouraged to write any helper methods for the assigned methods to improve style. Of course, be sure to not change any of the method signatures that you are required to implement.

Simple Simulation
More Crazy Simulation

Helpful Information:

Online Communication: Using Piazza, Opening Regrade Requests

Getting Help from Tutor, TA, and Professor's Hours

Lab and Office Hours (Always refer to this calendar before posting.)

Academic Integrity: What You Can and Can't Do in CSE 8A

Remote Lab Access and File Transfer Guide or **Yingjun's SSH and VNC post**

Java.lang.Math Class Documentation

Picture documentation and **Sound Documentation**

Additional Explanation of Physics

Table of Contents:

How to get Started

Problem: **(25 points total)**

Part 0 - Overview

Part 1-1 - Introduction **(2 points)**

Part 1-2 - Calculating Forces (5 points)

Part 1-3 - Drawing the Bodies (4 points)

Part 2 - Writing NBody and the main Method (9 points)

Part 2A - Updating our System of Bodies

Part 2B - Drawing our System of Bodies

Explanation of how the methods interact

How you can test your code and how we will grade it

Program Description (3 points)

Comments and Style (2 points)

Part 3 - Star Point

How to turn in your assignment

How to get Started

If you are working on a **B230 lab machine,** it's available to copy from the public directory. This will copy it into your home directory:

```
cp -r ~/../public/psa8/ ~/.
```

You will also need your Picture.java file from PSA3 into the directory for this assignment you can copy it from your psa3 directory to your psa8 directory with the following command:

```
cp ~/psa3/Picture.java ~/psa8/
```

If you are working on **your own machine** grab all the starter code from StarterCode.zip here. Make sure that you also have the images folder and simulation folder (with the folders named exactly as 'images' and 'simulations' with no extra characters/numbers) in your PSA8 folder. Make sure to also to find and copy Picture.java from psa3.

If you have deleted your psa3, please leave a follow-up on the Piazza post for PSA retrieval. For this assignment, you will be using the `alphaBlending()` method you wrote in PSA3. Please fix any bugs or issues you may have had.

Problem:

Part 0 - Overview on Physics

This assignment requires you complete two classes: NBody and Body. The Body class represents the particles or planets in the simulation, and the NBody class conducts the simulation. The NBody class will orchestrate the simulation, and the Body class will be used to store the values of the bodies at each point in time and also will have helper methods that will be used by the NBody class.

Before writing code for the Body and NBody classes we'll look at some of the physics behind the simulations. You can refer back to this explanation when implementing methods in the Body class that are described after this overview section. We will try to put the method in the Body class in our explanations below so it makes sense to you about what each method should do.

Our Body objects will obey the laws of Newtonian physics. You won't need to fully understand the laws of physics used here, you will need to understand some basic geometry and how the simulation based on these laws works. In particular, planet objects in the simulation will be subject to:

1. **Pairwise Force**: *Newton's law of universal gravitation* asserts that the strength of the gravitational force between two particles is given by the product of their masses divided by the square of the distance between them, scaled by the gravitational constant *G* ($6.67 * 10^{-11}$ N-m^2 / kg^2). The gravitational force exerted on a particle is along the straight line between them (we are ignoring here strange effects like the curvature of space). Since we are using Cartesian coordinates to represent the position of a particle, it is convenient to break up the force into its x- and y-components (F_x, F_y). The relevant equations are shown below. We have not derived these equations, and you should just trust us.

- $F = G * m_1 * m_2 / r^2$
- $r^2 = \Delta x^2 + \Delta y^2$

(Note Δx is delta/difference between x-coordinates, similarly for Δy).

- $F_y = F * \Delta y / r$
- $F_x = F * \Delta x / r$

Note that force is a vector (i.e., it has direction). In particular, be aware that Δx and Δy are signed (positive or negative).

2. **Net Force**: The *principle of superposition* says that the net force acting on a particle in the x- or y-direction is the sum of the pairwise forces acting on the particle in that direction. That is, if 3 forces are acting on a Body, then the x component of the total force, F_x, on the Body will just be the the sum of the x components of the 3 forces.

In addition, all bodies have:

3. **Acceleration**: Newton's *second law of motion* says that the accelerations in the x- and y-directions are given by the equations below (derived from $F = ma$):

$a = F / m$

$a_x = F_x / m$

$a_y = F_y / m$

4. Combining formulas 1, 2, and 3, we can obtain a direct formula for acceleration calculation

$a = F / m = (G * m * m_2 / r^2)/m = G * m_2 / r^2$

$a_x = a * \Delta x / r$

$a_y = a * \Delta y / r$

Here m_2 is the mass of the body that applies the force. By directly using this formula, we can avoid using formulas 1, 2, and 3.

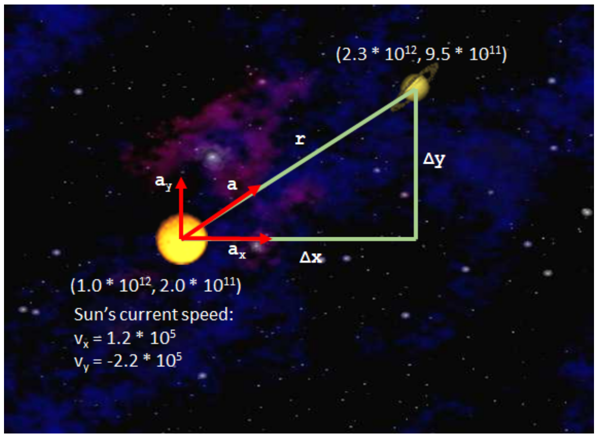
5. The **speed** of a body is changed by the following formula
$v_{x\_next} = v_{x\_current} + a_x * \Delta t$
$v_{y\_next} = v_{y\_current} + a_y * \Delta t$

6. The **distance** that a body will travel in Δt time is calculated as
$x_{next} = x_{current} + v_{x\_next} * \Delta t$
$y_{next} = y_{current} + v_{y\_next} * \Delta t$

Examples

Consider a small example consisting of two celestial objects: Saturn and the Sun. Suppose the Sun is at coordinates $(1.0 * 10^{12}, 2.0 * 10^{11})$ and Saturn is at coordinates $(2.3 * 10^{12}, 9.5 * 10^{11})$. Assume that the Sun's mass is $2.0 * 10^{30}$ Kg and Saturn's mass is $6.0 * 10^{26}$ Kg. The current speed of the sun in the x direction is 120 m/s and the y direction is -220 m/s. Assume our time interval for the system is a day (8.64 * 10^4 seconds)

Here's a diagram of this simple solar system:



Let's run through some sample calculations.

a) We'll begin by calculating r, which we've already expressed above in terms of Δx and Δy. Since we're calculating the force exerted by Saturn, Δx is Saturn's x-position minus Sun's x-position, which is $1.3 * 10^{12}$ meters. Similarly, Δy is $7.5 * 10^{11}$ meters.

   So, $r^2 = dx^2 + dy^2 = (1.3 * 10^{12} \text{ m})^2 + (7.5 * 10^{11} \text{ m})^2$. Solving for r gives us $1.5 * 10^{12}$ meters. Now that we have r, computation of other quantities is straightforward:

b) Acceleration of the Sun:
   $a = G * m_2 / r^2 = (6.67 * 10^{-11} \text{ N-m}^2 / \text{kg}^2) * 6.0 * 10^{26} \text{ kg} / (1.5 * 10^{12} \text{ m})^2 = 1.778 * 10^{-8} \text{ N/kg} = 1.778 * 10^{-8} \text{ m/s}^2$ (see here why N/kg is the same as m/s^2).
   $a_x = a * \Delta x / r = 1.778 * 10^{-8} * (1.3 * 10^{12} \text{ meters}) / 1.5 * 10^{12}$

meters = 1.5409 * 10^{-8} m/s^2
a_y = a * Δy / r = 1.778 * 10^{-8} * (7.5 * 10^{11} meters) / 1.5 * 10^{12} meters = 0.889 * 10^{-8} m/s^2

c) Update the speed of the Sun for the next day
v_{x_next} = $v_{x\_current}$ + a_x*Δt = 120 m/s + 1.5409 * 10^{-8} m/s * 8.64 * 10^4 s = 120.001 m/s
v_{y_next} = $v_{y\_current}$ + a_y*Δt = -220 m/s + 0.889 * 10^{-8} m/a * 8.64 * 10^4 s = -219.999 m/s

d) Calculate the new location of the Sun
x_{next} = $x_{current}$ + v_{x_next} * Δt = 1.0 * 10^{12} m + 120.001 m/s * 8.64 * 10^4 s = 1.000010368086 * 10^{12} m
y_{next} = $y_{current}$ + v_{y_next} * Δt = 2.0 * 10^{11} m + -219.999 m/s * 8.64 * 10^4 s = 1.99980992086 * 10^{11} m

These calculations will be repeated until the end of the universe. :-). Also keep in mind that if we also have the Earth in the system, the Sun will be affected by both Earth and Saturn. We can calculate the how each planet affects the Sun's accelerations, and combine them in the x and y directions. Then use the combined acceleration to update the speed of the Sun and calculate the new location of the Sun.

You can read more information in this document.

Part 1 - The Body class

Part 1-1 - Introduction

The "Body" class represents an object of our system whose forces will interact with the other bodies in the system. For example, a "body" might be a planet in the solar system, moving and orbiting based on the other bodies in the system.

The Body class stores all of its information in six fields (member variables, or variables that belong to an instance of a class): `xPosition`, `yPosition`, `xVelocity`, `yVelocity`, `mass`, and `image`. `xPosition` and `yPosition` are doubles that hold the current location of the Body in the universe. `xVelocity` and `yVelocity` are doubles that indicate how fast (and in what direction) the Body is currently moving, and will be used to update the position of the Body as the simulation is run. `mass` is a double that measures how much matter makes up an object; a Body with more mass will exert a stronger pull on other Bodies. `image` is a Picture that will be used to draw the Body during the simulation.

Recall that static variables are shared by all objects of the "Body" and that final variables cannot be modified, which makes them appropriate for constants. The Body class has two static variables: `TIME_STEP`, the amount of time in seconds that each step in our simulation represents, and `UNIVERSE_SIZE`, which lets us keep of the overall scale of the universe. These are static because they will remain the same for all Bodies in a system. Additionally, there is a static final variable `GRAVITATIONAL_CONSTANT`, which is the same every time we run the program, and appears in some of the formulas we'll use for our simulation.

Now that you are familiar with the concept and variables inside the Body class, it's time to start writing the helper methods we need within the Body class. Implement these first:

`public Body(double xPosition, double yPosition, double xVelocity, double yVelocity, double mass, Picture image)` **(0.5 point)**: This is the constructor for the Body class. In this constructor, you should set all the private fields in the Body class to the values given in the parameters. Hint: you can distinguish between the parameters and private fields using the `this` keyword.

`public double getXPosition()` **(0.5 point)**: This method simply returns the current value of `xPosition`. We need this method because `xPosition` is private, we cannot get the `xPosition` of a Body in the code of other classes. (Note: do not make `xPosition` public!)

`public double getYPosition()` **(0.5 point)**: This method simply returns the current value of `yPosition`. We need this method because `yPosition` is private, we cannot get the `yPosition` of a Body in the code of other classes. (Note: do not make `yPosition` public!)

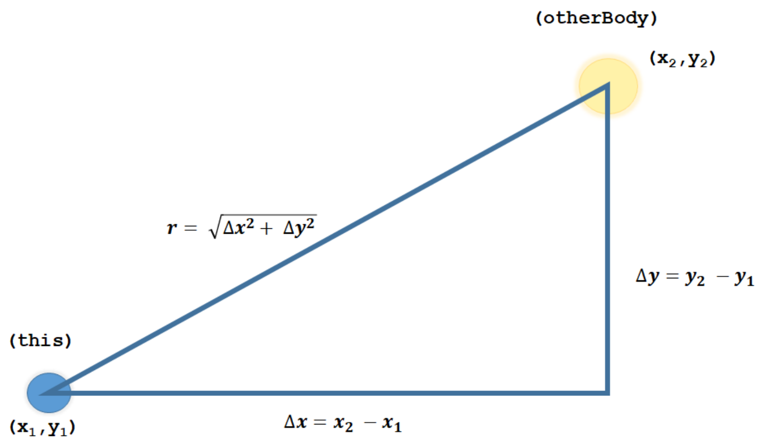
`public double getMass()` **(0.5 point)**: This method simply returns the current value of `mass`. We need this method because `mass` is private, we cannot get the `mass` of a Body in the code of other classes. (Note: do not make `mass` public!)

Part 1-2 - Calculating Forces

In order to get started with the physics simulation, begin with the following helper methods in the Body class:

`public double getDistance(Body otherBody)` **(1 point)**: Calculates the distance between the current Body and another Body using each body's x and y coordinates. You will use the pythagorean theorem in getting the distance between the two Bodies. This will be a helper method for the next method, `updateVelocity()`, as the distance between the bodies is important to finding the next acceleration.

- You will need to use the Java Math class to calculate the distance between the two bodies. If you don't already know what you can use, look through the linked documentation to find helper methods you can use and their descriptions. (You cannot use '`x ^ 2`' to calculate x^2.)
- You can refer to the picture below to visualize how you are to get the distance.



`public double getAcceleration(Body otherBody)` **(1 point)**: This helper method calculates the acceleration from the current Body toward one other Body, by applying the formula:

$$a = G\frac{m}{r^2}$$

where G is the gravitational constant, m is the mass of the other Body (not being accelerated), and r is the distance between them.

`public void updateVelocity(Body otherBody)` **(2 point)**: This method uses the acceleration towards another Body and updates the (calling object) Body's velocity, breaking it down into x and y components. It does this by modifying the values of `xVelocity` and `yVelocity` in the calling object.

- You need to find and update the Body's pairwise acceleration towards the other Body in the x component, $a_x$and in the y component, a_y. Use the `getAcceleration()` method to calculate the total magnitude (*a*) of the acceleration towards the other body. You can then separate that acceleration into x and y components by $a_x = a\frac{x_2 - x_1}{r}$and $a_y = a\frac{y_2 - y_1}{r}$ where x_2-x_1 and y_2-y_1 are the signed difference in x and y from the other Body to this Body, and r is the distance between the two bodies. Use the helper method `getDistance()` you wrote in order to get r.
  - Make sure that when you calculate the signed difference between the coordinates of this body to the other body, you are subtracting the current body's coordinate from the other body's coordinate, ie. `otherBody.getXPosition() - this.getXPosition()`.
- After you've calculated acceleration (in the x and y directions) use the acceleration to calculate the new velocity (x and y). The formula for the velocity in the x direction is $v_x(t+1) = v_x(t)\ +\ a_x{\times}timestep$ where $v_x(t+1)$ is the new velocity, updated for the updated time, and $v_x(t)$ is the previous velocity from one `TIME_STEP` ago, and a_x is the acceleration in the *x* component that was calculated in the previous step of the method.
  - The current $v_x(t)$ is the current value of the Body's `xVelocity`, and the body's new velocity $v_x(t+1)$will be the updated value of this same field.
  - Similarly for the *y* component, $v_y(t+1) = v_y(t)\ +\ a_y{\times}timestep$.

`public void move()`(1 point): In this method you will finally update the locations of the bodies, using the current location and the newly calculated velocity. The equations: $x = x_0 + v_x \Delta t$ and $y = y_0 + v_y \Delta t$ should be applied here. In this method, for the x component, this would translate into code as xPosition = xPosition + xVelocity*TIME_STEP.

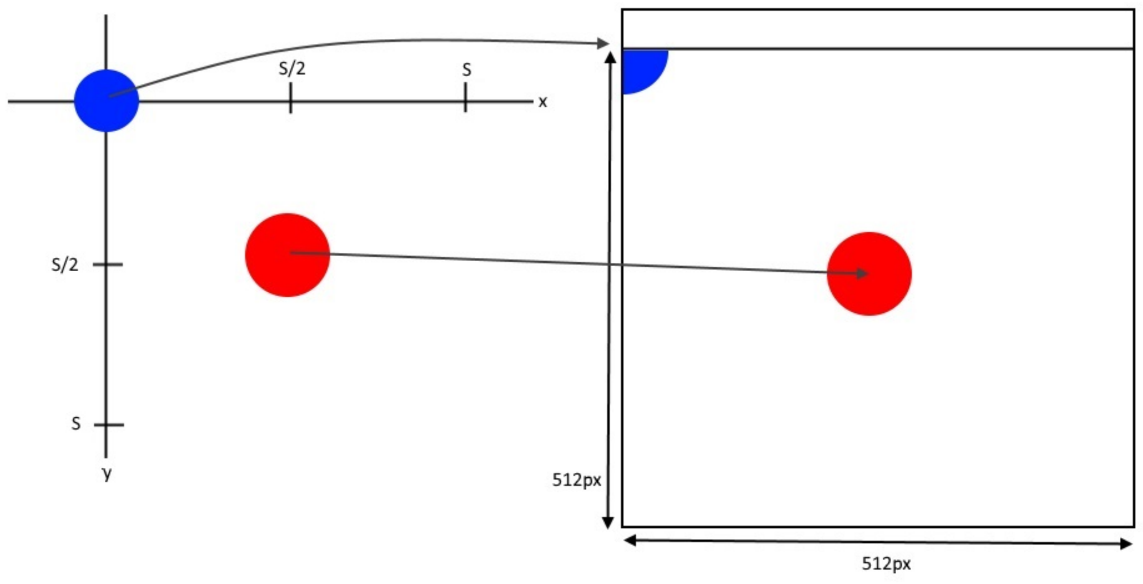
Part 1-3 - Drawing the Bodies

At this point you should have a Body class that can store the information for the planets and calculate the physics between them. Now, we will need to write some methods in the Body class that will help us to draw the Bodies on a Picture. You will be using and modifying the `alphaBlending()` method you wrote in PSA3 to blend the the images of the Bodies on the Picture for the universe. You will later show this Picture of universe to the user. The `draw()` method (in the Body class) will call `centerXToPixelPosition()` and `centerYToPixelPosition()` to get the pixel coordinates, then call alphaBlending (Picture method) to actually put the body picture on the system picture.

`public int centerXToPixelPosition()`(0.5 point): This method should map the `xPosition` of a Body in the system to the x coordinate of the corresponding pixel, based on the size of the universe Picture, which is the Body class's static field UNIVERSE_SIZE. You may assume that the the width of the universe Picture is going to be a constant 512.

- Since the x location of a planet could be something like 2.5e15, you need to translate that to fit in the picture.
- For this method, round down (truncate) for the pixel coordinates.
- For example, if the size of our universe is 6.7e13 and the body has xPosition of 2.3e12, the x pixel location would be $\frac{2.3\times10^{12}}{6.7\times10^{13}}\times512 = 17.57$, but since we round down this would be 17.

`public int centerYToPixelPosition()`(0.5 point): This method should map the `yPosition` of a Body to a y coordinate given the size of the Picture. You may assume that the the width of the universe Picture is going to be a constant 512.



Body locations should be scaled so that a body at (x, y) = (0, 0) should center in the top left, (S/2, S/2) should map to the exact center, and (S, S) should map to the bottom left, where S is the universe size.

`public void alphaBlending(int x, int y, Picture background)` **(1 point)**: This method is **part of the Picture class**. You should have your Picture class from PSA3. You will need to modify this method in order to avoid an IndexOutOfBoundsException. Here's what you need to change: before the method modifies a Pixel in background, you will need to check that you are not attempting to modify a Pixel at a location that is outside of the bounds of the picture (you need to check that you haven't gone beyond the background height or width, but **also** that you haven't gone to a negative x or y coordinate). If some of the Pixel locations are inside the universe Picture, but others do not, this method should still modify the Pixels at locations inside.

`public void draw(Picture universe)` **(2 point)**: This method (part of the Body class) draws the body onto the picture that is passed into the method. To do this, you will need to call the modified alphaBlending() method on the Picture given in the Body's constructor.

- The helper methods `centerXToPixelPosition()` and `centerYToPixelPosition()` will give you the pixel locations of the **center of the planet**. You will have to do a bit of thinking and a bit of math to center the image of the planet on its pixel location in the universe picture (Hint: rethink about PSA 4 where you have to find the upper left based on the center).
- As in PSA4, if the width/height of the image of the body is even, use the right/bottom middle pixel.
- The draw method should be short, since most of the work will be done by the methods it calls.

You have now completed the Body class. Make sure that it passes all tests **by following the testing instructions here.**

Part 2 - Writing NBody.java and Main Loop

The main method for your program is a loop that runs for the total time of the simulation. You need to play background music to make your motion picture more exciting, so you should start playing it at the beginning so it will continue to play as your universe is updated and each image is repainted. We must continuously update and redraw the Bodies in our system in order to show the user. There is also a method written for you in the NBody class called `pause()` that you should call after repaint to ensure that your animation will play smoothly.

Your main method should do the following **(3 point)**:

1. Load the background Picture and create a universe Picture.
2. Make the user pick a simulation with FileChooser.
3. Create a PlanetReader object using the returned filename to load the simulation.
4. Get the universe's size and the timestep using the `getUniverseSize()` and `getTimestep()` (in PlanetReader). Set the Body class's static variables `TIME_STEP` and `UNIVERSE_SIZE`.
5. Get the array of Bodies with `getPlanets()` and the total time to play the simulation with `getTotalTime()`.
6. Load and play the 2001 theme.
7. Draw the current state of the universe with `drawBodies()`, and show it.
8. From the current time (you will need to make a variable to hold the current time of the simulation, and it must start at 0), in steps of timestep, until the current time reaches the total simulation time:
    a. Allow a small pause to occur (call the `pause()` helper method given in the starter code).
    b. Update the velocity and positions of every Body by calling `moveBodies()`.
    c. Draw the current state of the universe, remembering to clear/reset the background by calling `drawBodies()`.
    d. Repaint the universe so the user can see any changes.

Note: when you run NBody.java, sometimes you will want to exit the simulation early. If you try to close the window, it will not work. Instead, click the "Reset" button in Dr. Java (next to "Compile"), which will stop the execution of your program, and close the window displaying the simulation.

Part A - Updating our System of Bodies

In Part 2 you made methods to calculate the pairwise force and the next position of the bodies after each `TIME_STEP`, and in this part we're going to use those methods to actually move all of the bodies.

`public static void moveBodies(Body[] planets)` **(3 point)**: This is a helper method in NBody that will update all of the bodies of the universe after one `TIME_STEP`. You need to update the velocities of all of the bodies with `updateVelocity()` for current frame *and then* update their positions with `move()`. If you do this incorrectly, you will be updating the acceleration of some Bodies based on the *next* positions rather than their current ones.

- Write a loop that that goes through all Bodies in the system and calls `updateVelocity()` several times, once with each other Body in the system, to update the velocities
- Write *another* loop through the Bodies and updates their positions with `move()`.

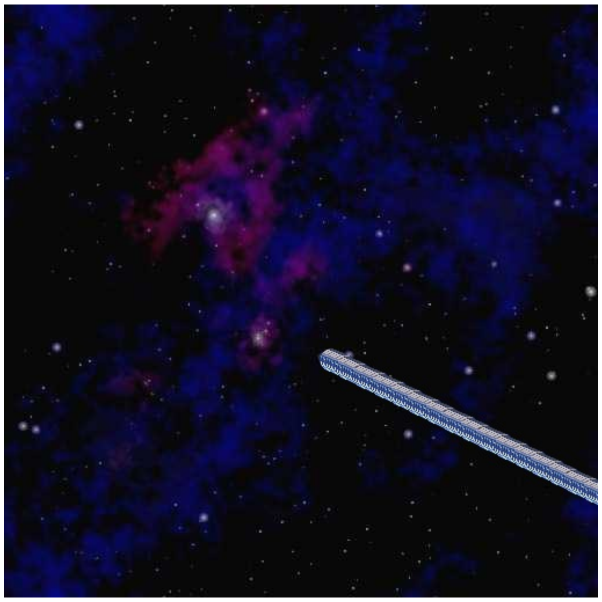
Hint: Don't try to calculate the acceleration between a Body and itself! You will likely get an Arithmetic Exception because you are dividing by the distance between them, which in this case will be zero. So in your loop for calculating the forces for each Body, check whether the two have the exact same x and y coordinates before accelerating them towards one another.

Part B - Drawing our System of Bodies

After Part 3, our Body class was capable of drawing itself on a Picture of the universe. In your main method, you created a Picture of the universe. You must now write a helper method that will use the draw method for each Body. Once this is done, you will display the universe Picture to the user by calling `repaint()` on it. You will need to reset the Picture, draw each planet, and display the Picture to the user. By using `repaint()` you will effectively show a video of the movement of the bodies. (Implementation details are below.)

Reset

What happens if we simply draw the bodies on the universe Picture every time they've moved? Well...



... it leaves an interesting trail behind it, but this is not exactly what we're looking for. You'll have to reset every pixel to its original background color before you draw any planets. In order to do this you can create a separate Picture object to hold the background, and copy the color of each pixel over to the universe Picture. You may assume that the background Picture (from "starfield.jpg") has the same dimensions (512x512) as the universe Picture.

Note: Copying a 512x512 image means copying more than 250,000 pixel values every time the bodies move. If this causes the simulation to run slowly on your computer, you may use the following line to reset the universe to black while working on the assignment (in `drawBodies()`, before drawing any of the planets):

```
universePic.getGraphics().clearRect(0, 0, CANVAS_WIDTH,
CANVAS_HEIGHT);
```

However, when you submit your assignment, it must reset the universe to the background image.

`public static void drawBodies(Body[] planets, Picture universe, Picture background)` **(3 point)**: This is the helper method that modifies the universe Picture to show the current state of the universe. You will need to reset the background picture onto the universe picture by copying each pixel's color from the background picture to the universe picture. After that, you will have to draw every Body by using the `draw` method you wrote in part 3.

Finally, you will need to call the `repaint()` method on the the finished picture. The `repaint()` method will close the window that shows the picture object as it previously was and replaces it with the Picture object as it is now.

If you have completed every method up to this point in the write-up, congratulations! You have finished writing the NBody program. You should now test your entire program by running a simulation **as described here.**

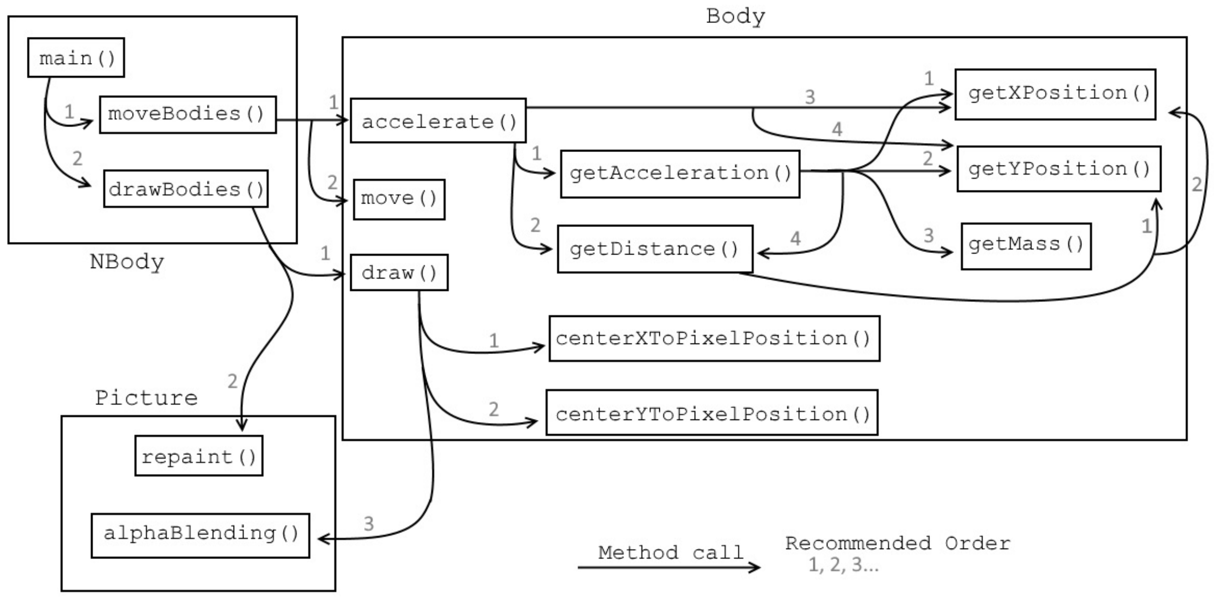
Explanation of How the Methods Interact

There are a lot of methods, and in the beginning it is confusing how they interact with each other. Here are some details for the classes you will be implementing.

```
public class Body
      // Fields/instance variables
      private double xPosition;
      private double yPosition;
      private double xVelocity;
      private double yVelocity;
      private double mass;
      private Picture image;
      // Class Methods
      public Body(double xPosition, double yPosition, double xVelocity,
            double yVelocity, double mass, Picture image);
      public double getXPosition();
      public double getYPosition();
      public double getMass();
      public double getDistance(Body otherBody);
      public double getAcceleration(Body otherBody);
      public void updateVelocity(Body otherBody);
      public void move();
      public int centerXToPixelPosition();
      public int centerYToPixelPosition();
      public void draw(Picture universe);

public class Picture //already implemented in PSA3
      public void alphaBlending(int x, int y, Picture background)

public class NBody
      public static void main(String[] args);
      public static void moveBodies(Body[] planets);
      public static void drawBodies(Body[] planets, Picture universe,
            Picture background);
```



This diagram shows the relationships of all methods from these classes.

Here is the overview how the code will run altogether: when the NBody program is run, it will update the bodies and repaint the picture of the system for each increment of time. To update the bodies each time, it will use the `moveBodies()` method to move all of the bodies. The `moveBodies()` method will use each Body's `updateVelocity()` and `move()` to determine the new positions for the Bodies. The `drawBodies()` method "resets" (see part 4) the picture of the system and calls each Body's `draw()` to draw the Body at its new position.

How you can test your code and how we will grade it

This assignment has many parts that rely on each other so it will be difficult to test the final product for this assignment until everything is complete. Therefore, we have provided a tester file to help you ensure the correctness of your individual methods. In order to run these tests, open BodyTester.java file and compile and run the program. The results of the test will be in the "Interactions" pane, with messages summarizing what went wrong. Please ensure that you pass these tests as you complete each method. This strategy will help you check your progress as your code gets more and more complicated.

Please note that **passing the tester does not guarantee full points on the assignment**!

In order to help with the correctness for the assignment, we will be providing unit tests for some of the methods in the Body class. In order to test your program, open BodyTester.java and press "Run." This will run all unit tests in the BodyTester.java file. The results of the unit tests will be output in the "Interactions" pane. These tests will not catch all errors in your program, but may help with identifying problems.

Once you believe that your code is correct and it passes all tests, then you should run some simulations to see if the resulting physics is correct. You should be able to run NBody and pick a simulation to run. You can compare your NBody program to some of the sample videos for the files (note that the simulation for planets.txt has been extended in the video and will be shorter when you run it):

planets.txt (extended)
armageddon.txt
3-body.txt
planets-elliptical.txt

You can also call the `toString()` method defined for the Body class in order to get a more full understanding of what your program is doing - this will give you the current state of a Body with all of its fields.

Program Description (3 points)

Describe what your program does as if it was intended for a 5 year old or your grandmother. Do not assume your reader is a computer science major. The programs you should comment on in this segment include Body.java and NBody.java. Be sure to describe the concept of a class and how it applies to your program as if to someone with no knowledge of computer science. Also include information on how you made sure your program is correct. **Write this as comments at top of NBody file.**

Comments and Style (2 points)

Remember to include file header comments with your and your partner's name, login ID, and date in *all* your files.

We will be looking for the following additional comment/style points in your code:

- Meaningful variable names (i.e. don't name a pixel "n", "i", or "bob" or "hello". Name them something pertinent to their role, such as "sourcePixel" or "targetPixel")
- Proper code indentation. Every subsequent coding/loop block must be further indented properly. Every single code example in your textbook follows this convention, so see your book for examples
    - In Dr. Java, you can highlight everything and press tab -- it will automatically indent everything for you! Remember for future projects (cse8b, etc.), you won't have Dr. Java as your editor, and will have to do this indenting manually!
- Method header comments at beginning of methods indicating what they do at a high level.

    - Example: "*//method flips the image upside down*" or "*//method mirrors the image along the center vertical line*".
    - Include a description of the parameters and return values in the header comments (google javadoc styling if you're not sure).
- **Each code line should be no longer than ~80 characters.**
- **Do not copy-paste formatted text into your code! Apostrophes and certain characters being inside of your code will break the code! We will take points off if your code does not compile because of this!**

Part 3 - Star Point (Optional)

Creating an interesting system.

For this star point, you can create your own simulation file with an interesting system. There is a strict format in which the reader reads a simulation file. Open some of the simulation files with a text editor to see the information they contain. Each file contains a header with the following, each on its own line:

1. The number of Bodies in the system.
2. The radius of the universe (The size of the universe is 2 * radius).
3. The timestep to run through the simulation at.
4. The total time the simulation should run for.

Then, for each Body in the simulation there is a line that specifies the following information (in order):

1. The x coordinate of the Body (see note).
2. The y coordinate of the Body (see note).
3. The x velocity of the Body.
4. The y velocity of the Body.
5. The mass of the Body.
6. The path to the image that represents the Body.

Note: when the reader reads the x and y positions of a Body, it adds the radius of the universe. (0, 0) in a simulation file will be the center of the universe.

In order to get the star point, you must create your own simulation (in the same directory as your NBody.java) called starpoint.txt that impresses your grader.

How to Turn in Your Homework

1. Place your psa8 folder in the home directory of your student account (cs8afxx).
2. Ensure that the psa8 folder contains at least all the following files: **Body.java, BodyTester.java, NBody.java, Picture.java, and PlanetReader.java.** If you completed the star point, ensure that your psa8 folder also contains **starpoint.txt** and any images that were not provided.
3. Run the command: `cse8aturnin psa8`
4. Follow the prompts.
5. Verify that your homework was turned in with the command: `cse8averify psa8`

Refer back to psa0 turnin instructions for additional details.