***NWEN 303 Project 1 Report***

**Task 1: How the model.java class works**

Beginning with the fields at the top of the class, 'size' represents the size of the GUI window in width and height. It's default value is 900, i.e therefore a 900 x 900 window is created using this field.

The fields gravitionalConstant, lightSpeed and timeFrame are all used in the logic of the particle in the 'interact' method.

The gravitionalConstant constant field represents the attraction that particles have to other particles. It is a constant acceleration force.

The lightspeed constant how quickly the particles move in a certain direction accelerated by the gravitionalConstant. Thus these two work in a counter balance effect.

The timeFrame constant controls the speed of the particles, i.e how far each particle can move in one given timeFrame. The higher the value the shorter the timestep.

P is the list of Particles in the model simulation. The particles themselves contain business logic that is used in the model to determine how they move, act and collide with other particles.

pDraw is a list of DrawableParticles. Each entry in this list is equivalent to the same indexed Particle in list 'p'. Each DrawableParticle has unique visual attributes relevant to the GUI but not relevant to the model. This list is listed as volatile, this is because of the structure of the mainLoop which calls step() when pDraw is updated. Essentially because there are 2 threads running the simulation, one for the GUI and one for the model, we need to make sure that the data that the GUI is using to draw is not cached. This is why the pDraw is atomically updated at the end of updateGraphicalRepresentation() too to avoid timeSlicing issues where the data drawn and the data stored in 'p' do not match.

The step() method operates like a tick method. It is called via the MainLoop class inside the GUI class and runs on its own thread. It has four purposes. In order, it first calculates new speed and direction values for each particle (including collision information). It then merges those particles that are now colliding in the model. It then moves (adjusts their position) of all particles based upon their speed and direction in a timeStep and finally updates the list of pDraw particles to reflect the changed data in 'p' so when pDraw particles are drawn, they reflect the state of the model data (p).

**How the algorithm works for combining particles that collided:**

It works by making each particle interact with every other particle (not inc. itself) in the simulation via a nested loop (second loop contained in particle class). Going deeper into this, if we look at the interact() method we can see new speeds and directions etc are calculated for each particle. One thing that is important that is calculated is if any other particles are impacting with us (our particle), if they are, we add them to our own personal impacting list for our particle.

Stepping back out to the step() method, we merge the particles. In the previous section of step() every particle touching any other particle was added to their impacting list, thus this meant they were impacting something. In mergeParticles() we look through every particle and look at their impacting list, if it's not empty they're impacting with another so we add it to a 'deadPs' stack and after iteration is complete we remove all of these from 'p', the main list of particles in the simulation model.

We then call getSingleChunk() on every element in deadPs, this returns a set of Particles to which any element that is touching the popped element is returned. Also any element that is touching any element that is touching the popped element is also returned etc. These particles are then removed from the 'deadPS' set of dead particles. Once removed, we have all the particles that need to be combined. We called the mergeParticles() method which takes into account masses and speeds of all the particles and then we add this combined particle to ‘p’, our main list of particles.

\_\_\_\_\_\_\_

Task 2: How does GUI.java work?

Gui.java contains an inner private class called MainLoop too. This description will include both of their workings.

What does it do?

It splits the task of displaying the View part of the MVC and the Model into two separate threads. Thus no contention between threads occcurs as the first thread's (gui) only purpose is to redraw the panel and the second (MainLoop) is to update the model.

Gui is a JFrame class that implements runnable. This means that it can be run on a thread defined by 'scheduler', the 'newScheduledThreadPool'. This Gui extending JFrame means it can support Swing views and JPanels so things can visually be drawn to a window. This is done by creating a 'canvas' class which is just a Swing JPanel, which has a simple paint method to iterate through the DrawableParticles of the model and draw ovals in their positions. Then repaint() is called on a 'scheduleAtFixedRate' applied via the scheduler. Schedule at fixed rate works by executing it every n time whether or not the last repaint has finish or not taking into account how long the last repaint took.

MainLoop is a little different. It implements Runnable only so has the ability to be run on a separate thread. It's running on the same ExecutorService that the 'gui' class is running on but on a different thread. MainLoop has a reference to the model (which holds the simulations data). The run method is run forever on that thread. The run method handles the timing for updating the model data so that this takes the same amount of time each time it loops. For instance, it takes the current time, then calls model.step() stepsForFrame times. It then calculates the time elapsed doing this. If the length of each frame minus the time taken to update the model is > 1 (i.e updating model took less time than the constant time required for each frame), then we sleep the thread for that long. This loops forever.

\_\_\_

What data is contended and what is the contention pattern?

Contention is simply when two threads try to access the same resource and as a result one of the contending threads runs slower than the other. This instance of a contention pattern is a race condition between shared data in 2 threads.

The first thread (that's purposed with drawing the simulation) has to read pDraw's list values to draw them. At the same time the other thread tasked with the model updating has to edit the contents of pDraw. pDraw is made volatile to make sure no cacheing takes place which is good but timeslicing could still take place which would mean that the GUI gets an old version of the data that isn't correct in comparison to the models updated data as there contention that slows down the Gui thread.

\_\_\_

Is this parallelism properly implemented?

It seems like a good way to separate GUI handling and model updates. For instance the O(n^2) complexity for interactions shouldn't effect the speed of the GUI updates, putting this on a new thread solves this nicely. Having one for each works simply. It seems like it's properly implemented, but access to shared resources should be carefully managed when extending the model and Gui threads as race conditions will occur. Synchronisation and locks should be used. But then of course there lies chances for lock contentions etc... I digress. Yep, seems like an OK implementation.