Operation

It may be required to download the required libraries should the Customer wish to build the simulator themselves. The required libraries are

openGL/ Glut:
```
sudo apt-get install freeglut3 freeglut3-dev
sudo apt-get install binutils-gold
```

GLM:
```
sudo apt-get install libglm-dev
```

To run the Simulation from the command line, navigate to the $/GravitySimulator and run the executable GravSim. To build the GravSim executable, simply run make from within the GravitySimulator directory.

To run the Simulation with the GUI, navigate to $/GravSimUI2-build-desktop-Qt_4_8_1_in_PATH__System__Release and run the executable GravSimUI2

Regardless of whether you are running from UI or the Simulation from command line, make sure to download Glut and GLM, and build GravitySimulator code.

GUI

The graphical user interface for this project is based off of QT creator. The user can launch the fake space and the user can also add, edit or delete their own space objects.
When the user clicks **Launch the fake space** button, the user will be able to see the gravity simulation. If one wants to add a space object, the user can click on **Create new object**. Then the user must enter all the values, the X, Y and positions of the new object must not be the same as sun because then the new objects will collide into the sun and hence the light source will be lost. In addition, the values of positions cannot be the same as any other object and the values of mass and radius need to be bigger than the mass and radius of mercury. Similarly, the user can edit values for their space object by clicking on the **Edit space object** button. If the user wants to delete an object, they can click the **Delete space object** button. For both the edit and delete actions, the user must select the desired row before editing or deleting it. If the user wants to add a moon, they have to make sure that a name of a planet comes before the moon. For example, say that the user prefers to see a Jupiter moon, then they should name that object JupiterMoon1. Also since Jupiter is one of those planets that has many moons, it may come in handy to end the name with the number like JupiterMoon1, JupiterMoon2, etc.

Main Structure

Main is where the brunt of the work is done for the simulation. It starts off by reading in the Planet text file, and setting up openGL. The next step begins the loop that calculates the physics and draws the results. First a step is taken, and the Planets positions are calculated, this then calls the System Display function that handles the drawing of the planets, setting up the camera, and adding light to the system. The buffer is finally swapped and this loop repeats.

Solar System Class

This class will take in space objects retrieved by parser.
It will assign break out the space objects by planets and then assign
any moons to it. It will also house the information of the size of the solar system and have a reference to the sun, which is just another space object

- **SolarSystem(vector<SpaceObject> &spaceObjects)**
  - Constructor that will add the space objects to the system and will also assign them to their respective groups using helper functions

- **update(vector<SpaceObject> &spaceObjects)**
  - Function that will update the system with the given space objects

- **getPlanets()**
  - Function that will return the vector of planets in this system

- **getSun()**
  - Function that will return the sun for this solar system

- **getStars()**
  - Function that will return the stars of this solar system

- **getEntireSystem()**
  - Function that will return a vector of all the space objects in this system

Planet Class

This class will house the planet space object and any moons that are associated with it

- **Planet(SpaceObject spaceObject)**
  - Constructor that just stores the planet object

- **hasMoon()**
  - Function to check if planet has moon(s).
  - return: true if it contains a moon, false otherwise

- **addMoon(SpaceObject moon)**
  - Add moon to this planet

- **getSpaceObject()**
  - Function that will get the planet object in order to get object info
  - return: The planet space object

- **getMoons()**
  - Function that will get the moon(s) that are associated with this planet
  - return: A vector of space objects that represent a moon

Ship Class

This will house the coordinates needed in order to draw different elements of the ship. It will also have a method that can be called in order to have the its window rendered on the screen using openGL.

There are also inner classes that help make things easier to house different information.

Inner Classes

Vector Class

This class is used in order to hold the vectors for the points being used in the different classes.

- **Vector()**
  - Constructor that will take in the x, y, and z vectors
- **getX()**
  - Function that will return the x vector
- **getY()**
  - Function that will return the y vector
- **getZ()**
  - Function that will return the z vector

Side Class

This will take vectors and store them for a side that will be used in the ship class

- **Side()**
  - Constructor that will take in vectors to represent the points for the side
- **topLeftPoint()**
  - Will return the top left point of the side

    - **bottomLeftPoint()**
        - Will return the bottom left point of the side
    - **topRightPoint()**
        - Will return the top right point of the side
    - **bottomRightPoint()**
        - Will return the bottom right point of the side
- **Ship()**
    - Constructor that calls a helper method that will assign the coordinates needed in order to draw the window.

- **renderWindow()**
    - Function that will render the ship screen using openGL. It goes through and draws the polygons using the sides created in the constructor and also sets the color for the different sides.