**Prometheus Simulator by Robert Sadler**

**Overview**

Simulator v1.0 was built using Java V8 and Java FX using Eclipse. JavaFX GUI was developed using Scene Builder. Any further changes to the GUI should be done using Scene Builder.

To use the simulation, launch the program and you will see a GUI. When pressing start, the simulation will run, and the main scene will show a robot, a ball, and two boxes. To drive the robot, use WASD keys and you can drive the robot. This is a simple form of control for the Robot, and it is built into the **Control** class. In the future, this should be developed so there is an API and commands are sent direction to the Robot when the **Control** class asks each AI element what it wants to do.

When the simulator is running, you can click on any entity (Obstacle or Robot) within the scene and the statistics for that object can be shown under “Entity Statistics”. Velocity for each **Obstacle** are usually separated into **VelocityX** and **VelocityY** for each axis, but the Velocity displayed in this area is the total velocity, which is the Pythagoras theorem on these two values.

Only “FPS” and “Entity #” are functional. Weather has not been implemented, and neither has the ability to save the data of the simulation to a file.

On the right-hand side, changing the tab from “Settings” to “Terrain Config” pauses the simulation. The “Save” and “Load” buttons currently do nothing, but should have the ability to load and save the terrain to a text file. Terrain can be represented by a large string that can be parsed, which will load a terrain as well as the obstacles, with their positions and velocities.

“Material” is a pane that contains buttons that can add things to the scene. Select a button and you can add that object to the scene. If the Button is a material, it will change the title of the terrain to that material, with the height of the value of the slider below it. You can place a “Fixed Obstacle” and a “Moveable Obstacle” to the scene, but currently they act the same. Placing a Robot in the scene does not work. It should be added in the future to have a way of configuring what these objects are, including size, mass and what Image they have.

The buttons in the menu bar currently do nothing, and were added as this was the original design of the GUI presented in the original Google Doc.

When the simulation works, collisions work (mostly – some bugs) as well as the sonar element for the robot. Walls are identified as a change between two terrain tiles which have a height difference of more than 1. This will eventually become a ramp, and will affect the obstacles with an increase or decrease of velocity.

The simulation aim is to run at 60 FPS. Frame rate should not affect simulation, but this was a personal goal for optimisation.

**Control**

Control is the class that interacts with the GUI, and any interactions with the GUI must be in this class. Control also contains the *mainSimulationLoop()* which first checks for any mouse Inputs, and then starts an *AnimationTimer()*, which creates a new thread for which the actual simulation loop is contained. The Loop runs the basic tasks:

* Update statistics of the simulation
* Draw the terrain – needs to be done first as things are drawn upon it
* Check physics. (See **Physics** for more detail)
* Draw the obstacles – each object is called to draw on top of the terrain at their now update position based upon what the **Physics** engine has calculated.

The other methods in the class are designed to support this main simulation loop, but are better explained by codes in the comment.

**Terrain**

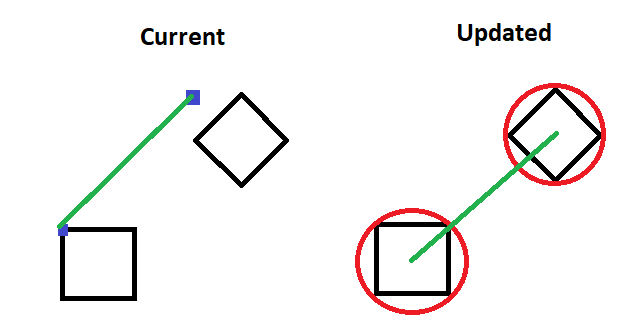
Terrain is a 2D array of Nodes. Each node has a material and a height. Each node also has a diameter. To gain a level of accuracy, each pixel in the simulation is equal to a cm in real life. Each Node is currently set to metre by metre squares. This scale should be maintained throughout the simulation for Obstacles as well, to maintain accuracy.

**Physics**

Physics, appropriately named, handles the Physics of the simulation. Currently it controls two main events; Collisions and Friction.

***Collisions***

Collisions can be thought of as two separate detection systems. Basic and advanced. Basic simply checks that for every obstacle in the scene, that they are checked against every other object that they’re not within a range of each other. *ToDo:* Currently basic detection relies on the distance between two objectsbased on a fixed threshold. However, this should be changed for two reasons. The position X and Y of an object are the top left corner, not the centre of the object. Instead, the check should be changed to work on the centre of Obstacles, and instead of some fixed threshold, it should be either the combined height or width (whichever is larger).



If the basic collision is true, then advanced collision should be checked. The may be replaced with something better, as the current operation is very computational expensive which is why the basic collision is required. Because objects can rotation, and because we have rounded obstacles (**Ball**), a simple bounding box rectangle would return true on obstacles even when the obstacles are not colliding. Instead, advanced collision works by create an area of the obstacles, and then calling an intersection of the areas. It is like the intersection of Sets, where the Intersection return is the area which is common to both. If there is anything at all in the Intersection, then the objects are overlapping.

There are two types of collisions we check for. **Obstacle** vs. **Wall** and **Obstacle** vs. **Obstacle**. *ToDo*: Make objects act with Fixed Obstacles the same way they act with **Walls**.

There is currently a bug where objects will pass through each other. Unsure as to what is causing this, and should be considered further. Possibly because Objects are moving quickly and are colliding between frames? May need **Collision** prediction to see if a collision happened between frames.

***Obstacle vs. Wall***

For each object in the scene, first you get the **Node** of the **Terrain** that the obstacle is currently inside. Walls are only temporary. This is an optimisation, as it would be very intensive to constantly have many obstacles within the scene, so walls are only generated on a “as needed” basis.

When the node is identified, it then check’s its neighbour’s nodes. If the difference of the height between the nodes is greater than 1, then a wall is generated along that edge. If you go to **Values** and enable *DRAWHITBOX*, you will be able to see the walls drawing when an Obstacle is close to it. If the advanced collision system detects a collision, it is then handled by the collision handling methods. Note, that these methods are very similar, except that the values of the wall are already known. I.e., they have no velocities before and after, and have a very large mass. Collision Handling is explained later.

***Obstacle vs. Obstacle***

Same as *vs. Wall* but does not generate walls (obviously). Instead, goes straight to collision handling methods, and both obstacles involved in the collision have velocities changed.

***Collision Handling***

Using Vector maths, we can figure out the collision direction between the two objects. The velocity along the normal is then calculated to figure out the total force of the collision. An optimisation here is that if the velocity along the normal is negative (i.e. they are moving away from each other), then it does nothing. This is so that an obstacle does not keep colliding over and over until they stop intersecting, which would cause weird behaviour.

The coefficient *e* is currently set manually. This should be updated in the future to either by programmed into the **Values** class or calculate on what the type of each obstacle is. Eplison *(e)* is a value between 0 or 1, which determines the elasticity of the collision (how much is bounces). This will need to be changed manually based on the type of materials, and maybe dependent on speed too. It is important to configure this properly so objects bounce or push appropriately.

***Friction***

Friction is very basic now. For every obstacle in the simulator, the velocities in each axis are multiplied by a percentage to decrease the velocity. Currently set at 99%. In the future, this should be updated to be dependent on mass, as well as what the current terrain is too.

Friction could be used to calculate what Epilison should be in *Collision Handling* to calculate how much energy would be lost in collision.

**Obstacles**

Obstacle is the super class for which all Obstacles in the scene inherit from. Everything entity in the simulation is an obstacle (including the robot). The subclasses, *Moveable* and *Driveable* inherit from this and have functionality that a Normal Obstacle (a fixed one) does not.

The position of the Obstacle is always set to the top left corner of the object, no matter what the orientation. This is because the *Canvas* element always draws from the top left corner.

A note about setting Velocities. There are three types of Velocities, X, Y and Velocity. Velocity is only used by driveable, as driveable always drives in the direction that it currently facing (*angle)*. Because off this, driveable has a different update method to the normal Obstacle class. It should be noticed that the *setVelocity* methods set the velocity instantly. It should be improved to consider acceleration, and instead of changing the velocity, it should add or decrease acceleration.

Also, worth noting, that the setVelocity methods take the argument in as metres, so this why it is multiplied by 100 to convert it to cm.

**Hitboxes**

Hitboxes are used for collision detection. They are expensive to compute and create and should be called as little as possible. Enabling *DRAWHITBOX* in **Values** draws these Hitboxes in the simulation. They rotation with the object, and fit as tightly as they can to enable accurate collision.

Ball has a different hitbox, as it is a perfect circle, and instead creates a circle object that represents the area that the ball takes up. The others create a polygon with a path that traces around the 4 main corners of the square.

**Driveable**

Driveable uses angles of a triangle to determine direction based on the velocity and the angle that it is currently facing. In order to “drive” the robot, the drive method is called. If no such method is called, then the robot is subject to collision handling as normal, where the velocities are set. This is needed as without the special case, the robot can drift and turn with the robot turning on the spot, which is not realistic.

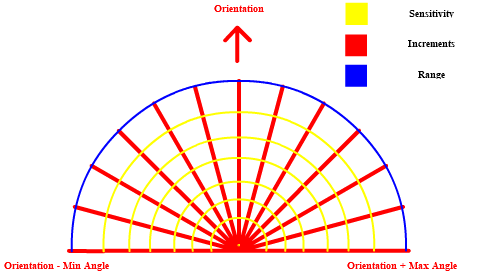
**Ball**

As stated before, Ball is perfectly round obstacle. It is drawn different as it is just a circle, as opposed to an Image on the screen. Its hitbox is also different.

**Sonar**

Sonar are attached to **Driveable** obstacles. They scan the area they are in based on the parameters inputted in the constructor, and return an array of distances to objects if there are any. You can see these in the Simulation by setting *DRAWSONAR* to true in Values. The diagram what the parameters of sonar do. One good thing to try with Sonar is to disable *DRAWOBSTACLES* with *DRAWSONAR* on, and seeing what the Robot will see.

Obviously, the higher the sensitivity, the more expensive it is to calculate. Optimisation can be done here so that hitboxes are not constantly calculated, and instead are saved. Obstacles only within range of the Robot should also be checked.



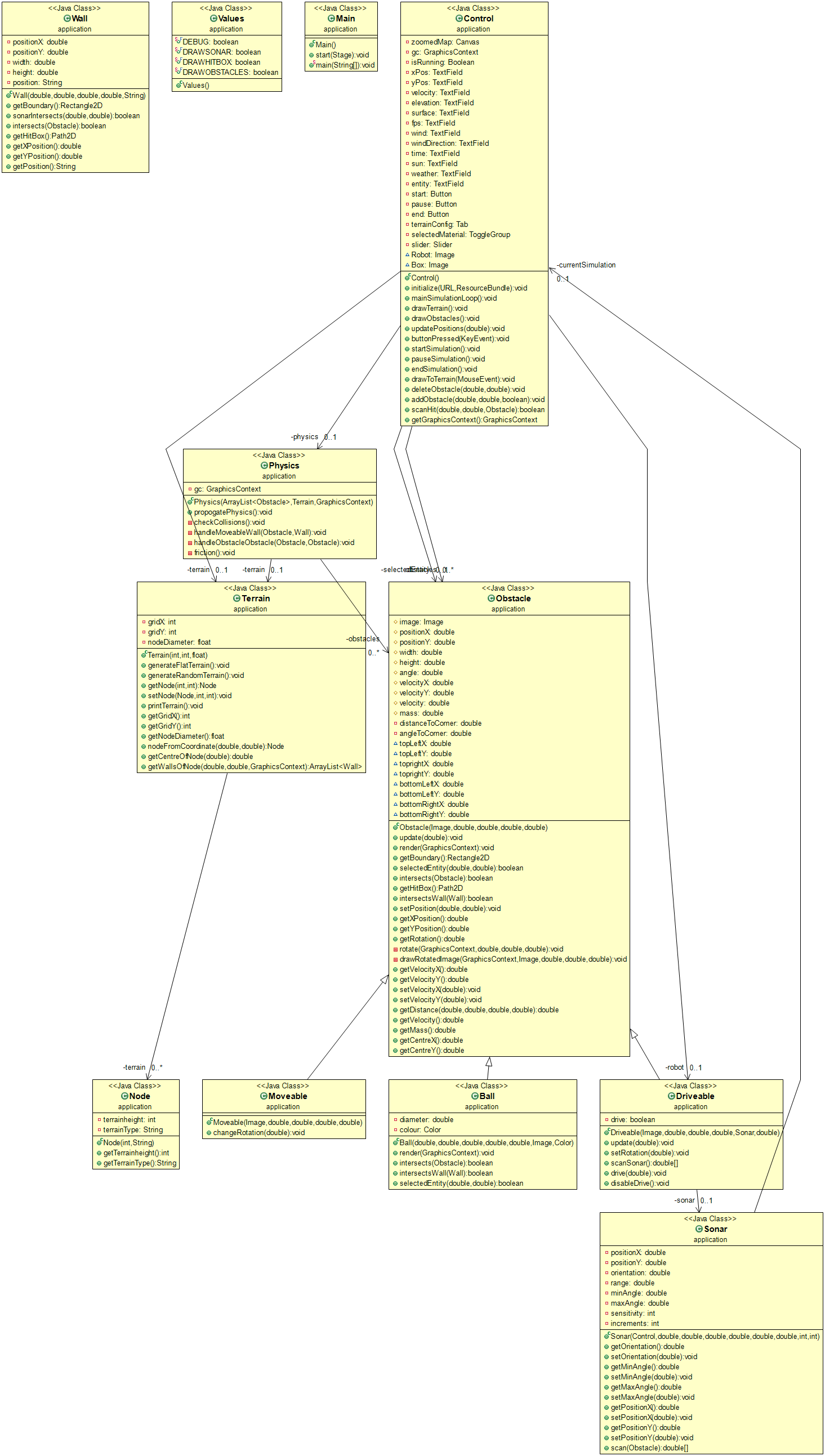
**Values**

Values is a static class used for making quick debugging changes to the simulator. Keeping one centralised class to control the simulation is beneficial. It has been referenced multiple times through this document for displaying debugging elements. It should also be used for storing Physics constants.

Would highly recommend building everything configurable for debugging around this class. Makes testing easier.

**Main**

Simply launches the GUI.

**Class Diagram**

**Known Bugs/Unimplemented Features**

* Rotation with collisions
  + When objects collide, add rotation velocity to the collision to “spin” off
* Friction
  + Friction is just multiplying by a constant value. The value should be dependent on Mass and the surface the obstacle on. Friction currently effects velocity, but, should affect acceleration. acceleration is not yet implemented, and velocity is set to instant values, no slowdown or speed up.
* Transitive collisions
  + If an object collides into a wall, then that object stops. If an object also collides into that object, or if the object cannot move and crashes into it, then that object can be pushed. A transitive check needs to be added
* Out of bounds
  + If an object goes out of bounds, have a wall around the entire perimeter of the world to keep objects in.
* Fixed Obstacles
  + Like walls, but with fixed Obstacles
* More Obstacle Images
  + Update the menu to add new obstacles, and have a dropdown obstacle selected.
* Obstacle configuration
  + Obstacles should be able to be placed and mass can be configured, as well as defining the type of Obstacle (Chair, table, box etc.)
* Save and Load Terrains
  + Have Terrains being able to save to a text file as a string, and then have that file loaded, parse the string, and create the world based upon that.
* CSV File saving
  + Recording when the simulation is running, save what happens
* Weather
  + Implement Weather and lighting
* Sensors
  + Other sensors can be implemented to measure other things, but heavily linked with weather. This can be implemented to have a Super of Sensor, and then Sonar refactored to be a sub type of Sensor. Each *Driveable* will then have a list of Sensors which get called in Update and send that information to the AI
* Sonar bug
  + Make sonar extend forever when we are looking over a “cliff” edge
* Ramps
  + Add ramps. Increase velocity going down, decrease when going up. Changing the Sonar too so when going down the ramp, it “cuts off” what can be seen.
* Menu Bar features
  + Fill in the buttons of the menu bar
* Wall collisions
  + sometimes objects can go through a wall or another object. Need’s considering for fixing. Collision detection is working, I believe it is a problem with the resolution.
* Place Robot
  + Place a robot using the editor menu. Not currently implemented.
* Interface for AI
  + Build the API for an AI to communicate with the simulation. Should be simply as it should just be communication with the Control class.
* End Simulation
  + Ending the simulation should create a blank world to begin with
* Generate Random Terrain
  + The ability to create “natural” terrains randomly generated with a theme. Using something like Perlin noise to create natural looking world for AI to work in.
* Add Agents
  + Agents are a subtype of Obstacles. They are like Moveable, but they are programmed to follow a route. It would be implemented with an array of co-ordinates, and the object would cycle through the points. Can be thought of like a person pacing. Each update frame will check if it’s at the target yet, else move towards it. Once at the target, get the next target, and repeat.
* Refactor Constructors
  + Make sure all obstacles follow the same constructor layout for easier writing. Currently the elements in the constructor are in different order due to being developed at different times. Should be refactored to restore consistency. I.e. Position, Size and Mass should all be in the same position as each other.
* MiniMap
  + A small canvas that shows the entire world. It’d be like drawing to the main canvas, but with a scaled factor
* Checkbox – Follow Entity
  + When an entity is selected, the simulator scrolls bars move to keep the entity centre frame.
* Replaying Speed
  + Simulator should have the ability to play at different speeds. It can be frame by frame, or half the speed etc. This can be done by changing what the timer is. If you take the value and divide it by the factor of slowdown, everything will be slower. Frame by Frame may help with debugging other bugs too. A simply button clicking “Next Frame” which moves the simulation along by one tick, and then pauses again.