Practical 4: Physics World

The Physics World application consists of four classes that are well distinguished.

public class PhysicsWorld

This class uses the JBox2D physics engine that simulates a number of objects with material properties and masses that may act under the force of gravity, and a few other optional forces (including joints, discussed a bit later). The objects move about in a predefined space, and are seen to collide and rotate with each other and a bounding-box aligned with the edges of the screen (as in figure 1). The objects in the PhysicsWorld class are simulated independently of visual rendering (cleanly separated in the class PhysicsView below).

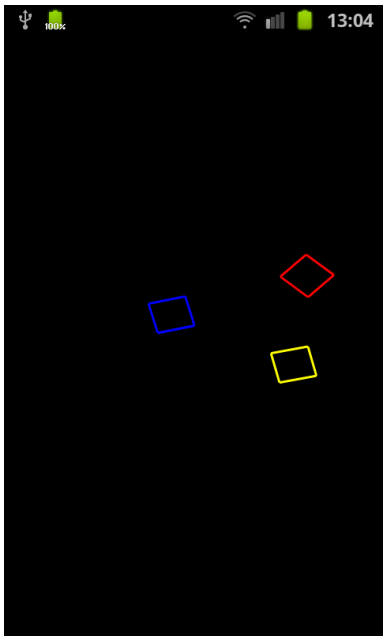

Figure 1: The Physics World application simulates three square-like objects.

The object provided by the JBox2D library that takes care of the simulation is called World. My implementation, PhysicsWorld, instantiates the World class and adds to it several (I've chosen three square-like objects) predefined objects using two principle definitions. These are the FixtureDef (fixture definition) that describes properties of a body's dynamics including their size and material properties (friction, density and restitution) that are relevant to determine what happens on collisions. The other is the BodyDef (body definition) that describes their localisation properties, chiefly position, initial velocity and angular velocity. The World class is a factory class that take the definitions and returns a reference to the instantiated (and thereby simulated) object. The objects in the PhysicsWorld are set in motion by the method world.step(timeStep, velocityIterations, positionIterations). timeStep is taken as the frame-rate (1/60s in my case). The velocity and position iterations relate to number of computations we desire between each frame, higher values give better simulation but is computationally more expensive. I've set these to recommended (by the JBox2D manual) value of 6 and 4 respectively. I've chosen a bounding-box to have the following world-coordinates that are roughly the same aspect ratio as the screen (units can be thought of as in metres):

$0 < x < 10, 0 < y < 20$

public class WorldThread **extends** Thread

The PhysicsWorld operates in discrete time-steps, so motion is controlled externally within a separate Thread. WorldThread is a very light class that simply keeps track of time and tries to increment the simulation at regular intervals. The thread loops indefinitely within its run() method, given in the implementation I have

```
sleep(SLEEP_TIME - time_d);
```

where time_d is measured as the time-delay it takes for the simulation to perform the incremental processing. SLEEP_TIME is given in millisecond, by some trial-and-error a value of 30 was seen to work well. I observed that with the subtraction of time_d to the sleep-time the simulation was significantly more fluent than when performed without. Noting also the values of time_d were usually somewhere in the ranges of 5-20 ms; the computational step is large compared to the chosen delay.

public class PhysicsView **extends** SurfaceView

PhysicsView sub-classes SurfaceView which is specifically designed for demanding graphics. It is called on each completion of the World time-step. It accesses the world-coordinates of the objects by calling getWorldPoint() on each of the vertices. The world-coordinates are transformed to screen-coordinates (in pixels) using linear transformations. Colours and PaintBrush properties are defined here, and I chose to give the three squares the colour blue, red and yellow. Considering game-engines

in general, it is in a class like this one would add any special effects like sparks or glowing lights on collision. The Physics World only provides basic mathematical points, but there is a lot of visual potential one could add on top of it.

SurfaceView works in general by obtaining a drawable canvas by calling

```
Canvas canvas = surface_holder.lockCanvas();
```

the canvas is later released. In between the lock and release no screen updates will occurs. The graphical rendering happens in-between, on release the frame-buffers are allowed to swap which produce smooth animations free from visual interference.

I implemented MotionEvent listener in the PhysicsView class that will create a MouseJoint between the cursor point and the blue square on ACTION_DOWN. The MouseJoint target updates on the coordinate position (translated to world coordinates) of ACTION_MOVE, and is released when the touch dis-engages. This has the effect of being able to fling the object around. Unfortunately, the implementation is seen to cause a null-pointer exception at times. I wondered if it could have something to do with synchronisation issues, but an attempt at implementing a Handler and message queueing wasn't successful.

```
public class PhysicsActivity extends Activity implements SensorEventListener
```

The PhysicsActivity is the main class that instantiates and brings together all the other classes. It sets the PhysicsView as the entire content view and so is not dependent on any UI specification. The SensorEventListener is implemented to find the orientation of the phone with respect to the horizontal. The onSensorChanged event handler changes the direction and the force of the PhysicsWorld gravity by multiplying by the sine of each of the x,y orientations. This produces a force relationship to the orientation, as if one imagines the objects in the Physics-World to be sliding on top of the phone as if it was a horizontal plane.