Practical 4 – Sprite Collision Detection + Tile Maps

This practical sheet is focussed on collision detection and tile maps and continues on from the previous practical sheet with the same code that you were working on before. We will start off by looking at sprite to sprite collision detection then add a tile map. We will then consider sprite to tile map collision detection and a possible resolution.

Collision Detection
We will start by trying out a simple form of sprite to sprite collision detection. In order to do this, we need something to collide with. Since we already have the ability to create a rock sprite from our previous practical sheet, we can just create another rock sprite and set it on a collision course with our first sprite. Before you do this, make sure that you reset the scale of your original rock back to 1x1.

1. In the `init` method of GCTester.java use the example of rock as a guide and create a second sprite called *boulder* with the same animation as *rock*, and set its initial position as 500,500 and its velocity as -0.1,-0.1, as follows:

```
boulder = new Sprite(anim);
boulder.setPosition(500,500);
boulder.setVelocity(-0.1f,-0.1f);
```

   To draw it, just use `boulder.draw(g);` in the *draw* method and don't forget to put a call to `boulder.update(elapsed);` and `checkScreenEdge(boulder);` in the GCTester.update method. Search for all the places in GCTester than you use *rock* and make sure you perform an equivalent action for *boulder*. Once you have done this, compile and test your code. You should see *boulder* and *rock* glide gracefully through each other.

2. Now we are going to add a collision detector based on the simple bounding box concept discussed in the lectures. Add the following method to the bottom of your GCTester.java file:

```
public boolean boundingBoxCollision(Sprite s1, Sprite s2)
{
  return ((s1.getX() + s1.getImage().getWidth(null) > s2.getX()) &&
        (s1.getX() < (s2.getX() + s2.getImage().getWidth(null))) &&
        ((s1.getY() + s1.getImage().getHeight(null) > s2.getY()) &&
        (s1.getY() < s2.getY() + s2.getImage().getHeight(null))));

}
```

   This method will return true if Sprite s1 is deemed to have collided with Sprite s2. To use this method, we need to make a call to it in our *update* method and then take some appropriate action. In our case we are just going to stop both sprites and see where the algorithm thinks they have collided. In order to do this, put the following code at the end of your *update* method.

```
if (boundingBoxCollision(rock,boulder))
{
   rock.stop();
   boulder.stop();
}
```

   Compile and test this code. You should observe that the sprites are deemed to have collided even though they are not actually touching. This is because the boxes that define their outer limits have collided.

Practical 4 – Sprite Collision Detection + Tile Maps

3. You can double check this by adding the following code to the 'draw' method in GCTester:

```
g.setColor(Color.yellow);
boulder.drawBoundingBox(g);
rock.drawBoundingBox(g);
```

This will draw a yellow rectangle highlighting the bounding box of each sprite. You should see that a collision has been detected when the two corners of the sprites have met. You may find it useful to draw bounding boxes around your sprites when trying to debug your assignment code. For example, it is often quite useful to see where the invisible edges of your sprites are when working out collisions with tile maps. You could put the above code in a section of the draw method that is only called if a debug value is set to true. The debug state can then be flipped using a key event which would cause the action `debug = !debug`.

4. The last thing to try is to vary the behaviour of the sprites in the collision. Instead of stopping them in their tracks, we are going to get them to bounce off each other by reversing their X velocity. This effect is not very realistic but it does demonstrate the principal. To do this, try changing the action from stopping the two sprites to the following:

```
rock.setVelocityX(-rock.getVelocityX());
boulder.setVelocityX(-boulder.getVelocityX());
```

5. You will want to think about trying to add a more advanced form of collision detection to the above such as the bounding circle method. A `drawBoundingCircle` method has been added to the Sprite class to help you visualise this and it can be called in a similar way to `drawBoundingBox`. When you are writing your bounding circle collision code, ensure you are making your calculations from the centre of the sprite rather than the top left x and y coordinates that the sprite reports as its position.

Practical 4 – Sprite Collision Detection + Tile Maps

Tile Maps

1. We have recently covered the use of tile maps in lectures and you should find that the relevant tile map code is provided in the game2D package as Tile.java and TileMap.java. We will now add a tile map to our current setup and explore how to use it. The first thing we need to do is to declare a reference to the tile map we are going to use and assign a new TileMap object to it. Add the following declaration at the top of the GCTester code after the declaration of the rock and boulder sprites to achieve this:

```
TileMap tmap = new TileMap();
```

2. We need to load a tile map into our TileMap object and the best place to do this for now will be in our init method. Add the following code at the end of GCTester.init to load the relevant map:

```
tmap.loadMap("maps", "example-map.txt");
```

3. In order for the user to see the tile map associated with tmap, we need to draw it so add the following line to GCTester.draw:

```
tmap.draw(g,0,0);
```

   The 3 parameters for the `TileMap.draw` method are the graphics object to draw to and an x and y offset to shift the relative drawing position of the tile map. These x and y parameters will be useful later on when you wish to have a game world that is bigger than the window size you are using.

4. Compile and run this code. If all goes according to plan, you should see orange blocks around the edge of the screen corresponding to the 'b' characters in the tile map and a couple of green circles that correspond to the 'c' character. Try editing the `example-map.txt` file in the maps directory by adding a couple of extra green circles to the tile map and check that they render correctly when you run the game again.

Practical 4 – Sprite Collision Detection + Tile Maps

5. You will have probably noticed very quickly that the rock and boulder sprites that we were working with in the previous section are not colliding with the tile map and sail straight through the tiles. The principles required to correct this are covered in the lecture notes but the following sample code will help you start along this process:

```
public void checkTileCollision(Sprite s, TileMap tmap)
{
    // Take a note of a sprite's current position
    float sx = s.getX();
    float sy = s.getY();

    // Find out how wide and how tall a tile is
    float tileWidth = tmap.getTileWidth();
    float tileHeight = tmap.getTileHeight();

    // Divide the sprite's x coordinate by the width of a tile, to get
    // the number of tiles across the x axis that the sprite is at

    int xtile = (int)(sx / tileWidth);
    // The same applies to the y coordinate
    int ytile = (int)(sy / tileHeight);

    // What tile character is at the top left of the sprite s?
    char ch = tmap.getTileChar(xtile, ytile);

    if (ch != '.') // If it's not a dot (empty space), handle it
    {
        // This just stops the sprite where the collision was
        // detected but you will probably need to move it back
        // to a position where there is no collision or hide it.
        s.stop();
    }
}
```

Add the above method to the GCTester class, put in calls to this method at the end of the GCTester.update method using the code that follows and run the code to see what happens:

```
checkTileCollision(rock,tmap);
checkTileCollision(boulder,tmap);
```

6. The above code just looks for a collision with the top left corner of the sprite. Your code for the assignment will provide a further example of a collision in the bottom left of the sprite. You will need to consider all four corners of the sprite and decide what to do about a collision. In this case we stop the sprite but you might want to make it bounce and perhaps remove the tile it collided with. If you want to try this, replace `s.stop()` with the following lines and rerun the program:

```
s.setVelocityX(-s.getVelocityX()); // Reverse velocity
tmap.setTileChar('.',xtile,ytile); // Remove the tile we hit
```

The physics are not quite right here but it should give you an idea of what is possible and how to change characters in the tile map within the program. With the addition of sound, you should now be in a position to create your own game world using sprites and tile maps. You will need to work on improving collision detection and deciding what to do when a sprite collides with a tile or another sprite but the core elements have now been covered.