3D Practical 2 – Animation

In this practical we are going to modify the Checkers3D application from the first practical to add both keyboard control and time-based animation of the sphere position. You will learn how to construct Behavior classes to do this.

MoveSphere3D

Copy **‘K:\CSC9N6\Practicals\MoveSphere3D’** (or from \\**wsv****CSCU9N6****code****3D**\) to a sensible folder on your ‘H’ drive. Now create an Eclipse or BlueJ project in your copy of this folder, add all the files in the folder to your project and set the file **Checkers3D.java** as the ‘main’ file. This is the same application as for the first practical, but with the addition of two classes: **TouristControls** and **TimeBehavior**. Initially neither of these classes is used, so we will add code to make use of them.

Keyboard Control

Firstly we will change the sphere size and initial position. Then we will add keyboard control so that we can manually move the sphere around the checkerboard floor. Take a look at **TouristControls.java** to see how its code is structured. The initialize() method sets up the behaviour activation (wakeup) criterion, which will be any key press. Pressing a key results in the processStimulus() method being called, which extracts all current key presses and passes them to processKeyEvent() for processing. Note that processStimulus() reregisters the wakeup criterion before exiting (otherwise it would never be called again! It is a common mistake to forget to do this.) Now do the following.

1. Edit WrapCheckers3D and change the sphere size to a radius of 1.0 metre and change its initial position so that it just sits on the floor at the (0,0) x-z position. Compile and run to check you have achieved this.
2. To add some keyboard control of the sphere position, add the following two lines of code to the floatingSphere() method, after the Transform Group tg has been constructed.

```
TouristControls tc = new TouristControls(tg);
tc.setSchedulingBounds( bounds );
```

3. Add tc as a child of the sceneBG. Compile and run to see if it does anything! (Examine the code in TouristControls.java to find out what keys have been implemented so far.)
4. Edit TouristControls.java and modify the processKeyEvent method so that you can move the sphere to the left and right, and forwards and backwards using the arrow keys. You will find the necessary key codes have already been defined as global variables. You will need to work out what vector to pass to the doMove() method in each case.
5. Examine the doMove() method to find out how it works. Basically it is passed a vector (theMove) which is the incremental change in position that is required. This needs to be concatenated with the current translation transform of the sphere. This translation is read into the t3d object and theMove is used to create the translation transform toMove, which is then multiplied with the existing transform in t3d to complete the move. Note that objectTG points to the

translation transform group for the sphere (initialised in the TouristControls constructor.)

Adding an Obstacle

To make things more interesting we will add another object to our world. This will eventually act as an obstacle that the sphere cannot pass through.

6. Edit WrapCheckers3D and make a copy of the floatingSphere() method, calling it floatingWall().
7. In floatingWall(), change the declaration of the sphere into a Box object, giving it a suitable name e.g. myWall.
8. Change the creation of the sphere to be the creation of a box, using a width of 5, a height of 1 and a depth of 0.5 (actually you will get a box of twice this size due to a slightly strange way the boxes are created!)
9. Add your box to the transform group tg (instead of the sphere in this method). Also, alter the translation held in tg so that the box sits on the floor and is situated 3 metres along the z-axis from the origin away from the viewer.
10. Change the material so that the wall is coloured red.
11. Remove the construction of the TouristControls object from this method (our wall will be static.)
12. Add a call to floatingWall() just before the call to floatingSphere() in createSceneGraph().
13. Compile and run your code. Your scene should now have a sphere and a wall. Play with moving the sphere. You will see it can pass right through the wall!

Collision Detection

To stop the sphere going through the wall we will add collision detection based on suitable bounding regions and modify the doMove() method in TouristControls so that it detects and reacts to collisions between the sphere and the box (wall).

14. Note that this version of WrapCheckers3D has a global variable boxBnds of type BoundingBox. In your floatingWall() method, create an instance of boxBnds that creates a BoundingBox that fits exactly around your box object. You will need to refer to the Java 3D API to work out how to call the BoundingBox constructor – you will add a call of the form:

    ```
    boxBnds = new BoundingBox(lower, upper);
    ```

    where “lower” and “upper” are the lower left-hand back and upper right-hand front corners of the box, respectively (you will need to work out their coordinates.) Unfortunately this BoundingBox is invisible, so you may not know if you have exactly succeeded until we have finished the collision detection!
15. To implement collision detection we need to tell our TouristControls object about the box and the sphere. To do this, change the TouristControls constructor so that you can pass a BoundingBox and the radius of your sphere as extra

arguments in the constructor call. Note to make things general, use Bounds (which is the parent class of BoundingBox) as the type of that argument.

16. Alter the constructor code to store the new arguments in the global variables collRad and obsBounds.
17. Now, back in floatingSphere(), alter your call to the TouristControls constructor to match the new specification.
18. Now we need to add some code to doMove() in TouristControls to detect when the sphere will run into the wall (box). We need to first generate a BoundingSphere that surrounds our sphere, then check whether this intersects with the BoundingBox. We do this by getting the predicted new position (translation) of our sphere from its transform group (stored here in t3d). We then use this to generate the appropriate BoundingSphere. Add the following code just before the last line of doMove() (ie just after the line “t3d.mul(toMove);”) to achieve this (and check the Java 3D API to work out what the code is doing!):

```
Vector3d trans = new Vector3d();
t3d.get( trans ); // get predicted translation of sphere
Point3d newLoc = new Point3d( trans.x, trans.y, trans.z);
BoundingSphere testBnds = new BoundingSphere(newLoc, collRad);
```

19. Add the following test so that a movement will only be made if it will not result in a collision (ie add the “if” test before the final line of doMove()):

```
// only allow move if it does not intersect with obstacle
if (!obsBounds.intersect(testBnds))
    objectTG.setTransform(t3d);
```

20. Compile and run your code. Move the sphere around with the arrow keys and see what happens now when it reaches the wall. (Using the mouse to move the viewer position to a “bird’s eye view” will give you a clearer picture of when the sphere collides with the wall.)

Animated Movement

For the final part of this practical, we will animate the movement of the sphere, in addition to having it under keyboard control. We will simply make the sphere move backwards and forwards along the Z-axis, bouncing off the wall then coming towards the viewer and returning towards the wall before it reaches the edge of the floor. To do this we will need to implement another behaviour. Skeleton code for this behaviour is in TimeBehavior.java.

21. Examine the current code in TimeBehavior.java. The constructor is very similar to the constructor we created for TouristControls, with the addition of the timeDelay parameter which specifies how frequently (in milliseconds) TimeBehavior is activated – see the wakeup condition timeOut. If you look at the initialize() and processStimulus() methods this wakeup condition is set so that TimeBehavior is activated after the required time delay (rather than on a keyboard press, as with TouristControls.)

22. Each activation of TimeBehavior will result in a movement of the sphere, and if we chose timeDelay suitably we will get a smooth animation. At the moment TimeBehavior does nothing, so we will need to add some code to get it to work. Firstly note that the current incremental movement is stored locally in currentMove, which is initialised to a small movement along the Z-axis away from the viewer ie towards the wall.
23. The doMove() method is essentially identical to the final version of doMove() you implemented in TouristControls. One difference is that it now returns the incremental movement. So when doMove() is called we both pass it the current value of currentMove and store the returned value in currentMove (see processStimulus().) This is because we will add code to doMove() so that it will make the sphere change direction when necessary.
24. So we haven't done anything yet! To see what TimeBehavior does at the moment, in WrapCheckers3D() add a constructor for a TimeBehavior object, just after the constructor for the TouristControls object in floatingSphere(). Use a time delay of 100msecs. Don't forget to add its scheduling bounds and to add it to sceneBG. Now compile and run to see what happens.
25. You should see the sphere move towards the wall and stop when it gets there. You can move it back a bit with the arrow keys but it will still return to the wall!
26. OK, here goes! – add code to the doMove() method in TimeBehavior so that the sphere automatically reverses direction when it hits the wall (so it comes back towards the viewer along the Z-axis) and automatically reverses direction back towards the wall before it reaches the edge of the checkered floor. To do this you will need more complicated logic in the "if" test and also and "else" branch. Note that Vector3d objects have a negate() method which will produce a vector in the opposite direction.