7. Interaction

The I/O in most of the examples up to now has consists of reading from and writing to the command line. Two more graphical approaches are to use the mouse to 'pick' (i.e. select) objects in a scene, and have models interact when they collide. Both **picking** and **collision detection** are implemented using Java 3D's Behavior class, so if you're unfamiliar with that class, then you should read chapter 5 first. Collision detection relies on collision wakeup conditions, while picking uses Java 3D's PickMouseBehavior, a subclass of Behavior.

1. Picking

Simple3D includes a PickBehavior class, a subclass of PickMouseBehavior, which deals with the conversion of a mouse click on the 3D canvas into a pick ray projected into the scene. The first pickable shape that this hits is examined, and the shape's name and (possibly) the ray's intersection point with the shape are passed to a pick() method defined by the PickUpdater interface. These elements are illustrated in Figure 16.

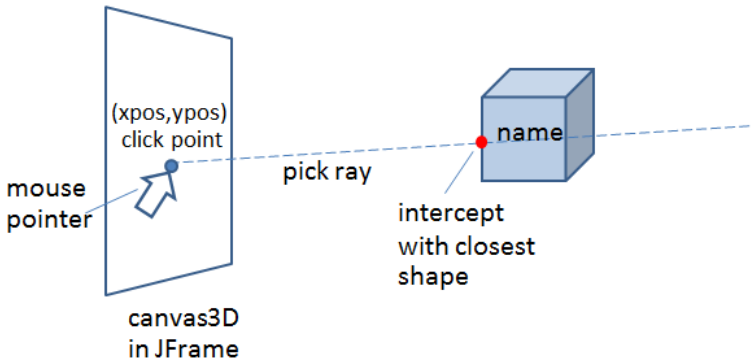

Figure 16. Picking a Shape with the Mouse.

PickBehavior is instantiated in a way a little unlike other Behavior classes because its PickMouseBehavior superclass requires a reference to the 3D canvas and scene graph:

```
public class PickBehavior extends PickMouseBehavior
{
  private PickUpdater pu;


  public PickBehavior(PickUpdater pu, Canvas3D canvas,
                                      BranchGroup sceneBG)
  {
    super(canvas, sceneBG, Simple3D.INFINITE_BOUNDS);
    this.pu = pu;
    setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
  }
```

```
  public void updateScene(int xpos, int ypos)
  {   ...  }


} // end of PickBehavior class
```

A class that extends PickMouseBehavior must implement updateScene(), which is passed the coordinates of the mouse click on the 3D canvas. My updateScene() fires a pick ray through those coordinates into the scene, and the name and intersection point of the closest shape are retrieved:

```
 public void updateScene(int xpos, int ypos)
 // in PickBehavior
 {
    pickCanvas.setShapeLocation(xpos, ypos);
         // register mouse pointer location on the screen (canvas)

    Point3d cameraPos = pickCanvas.getStartPosition();
            // get the camera's location

    PickResult pickResult = pickCanvas.pickClosest();
         // get the intersected shape closest to the camera

    if (pickResult != null) {  // get the shape's name
      String id = Simple3D.pickID(pickResult);

      // try to get the ray intersect point
      Point3d intercept = null;
      try {
        PickIntersection pi =
             pickResult.getClosestIntersection(cameraPos);
        intercept = pi.getPointCoordinatesVW();
      }
      catch(Exception e)  // intersection may fail
      { //System.out.println("No intercept found for " +  id);
      }

      pu.picked(id, intercept);
    }
  }  // end of updateScene()
```

Any Shape3D is pickable by default, but only the nodes below the scene graph supplied to the PickBehavior constructor will be considered. This will typically be a branch below Simple3D's rootBG, which will mean that Simple3D's floor and axes won't be examined. Even so, there may be a lot of matches, and so updateScene() only considers the first shape hit by the pick ray.

Simple3d.pickID() use the pick result to dereference the shape's name, which should have been set earlier. It utilizes two similar functions, getPrimitiveID() and

getShapeID(), to examine Primitive and Shape3D object respectively. Simple3D.getPrimitiveID() is coded as:

```
  private static String getPrimitiveID(PickResult pickResult)
  // extract ID from a Primitive object in pickResult
  {
    Primitive pickedShape =
             (Primitive) pickResult.getNode(PickResult.PRIMITIVE);
    if (pickedShape == null)
      return null;
    else {
      String id = pickedShape.getName();
      if (id == null)
        return "primitive";
      else
        return id;
    }
  } // end of getPrimitiveID()
```

updateScene() also attempts to get the pick ray's intersection point with the shape, which is always successful if the shape is a Primitive, but may fail for Shape3Ds, especially loaded OBJ models.

1.1. Picking Some Shapes

PickCubes.java uses PickBehavior to let the user to click on three boxes and a car model to get the shape's name and intersection point. Figure 17 shows the scene.

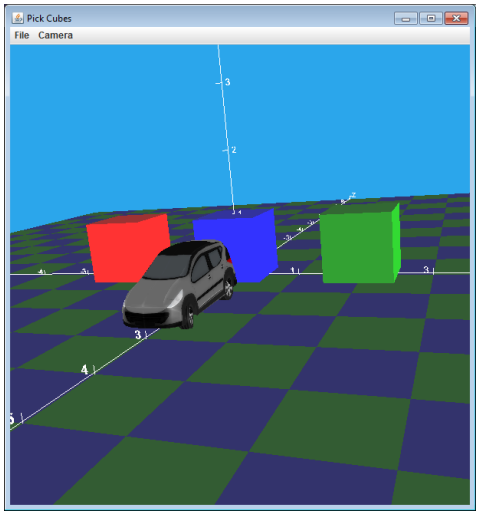

Figure 17. Picking Shapes with the Mouse.

Typical output:

```
green box picked at : ( 2.012, 0.468, 0.500 )
models/car/car.obj picked
red box picked at : ( -1.988, 0.644, 0.500 )
```

```
green box picked at : ( 1.568, 0.886, 0.500 )
```

The green box was clicked upon twice, but in slightly different places, and the picking of the car failed to return an intersection point. The shapes' names (e.g. "green box") were set when the scene was created in the PickCubes constructor.

The main() function creates a PickCubes object and a PickBehavior:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("Pick Cubes", true, true);

    PickCubes pc = new PickCubes();
    PickBehavior pb =  new PickBehavior(pc, j3d.canvas3D,
                                        pc.getScene());
    // build the scene graph
    BranchGroup bg = Simple3D.createBG();
    bg.addChild(pc.getScene());
    bg.addChild(pb);

    j3d.rootBG.addChild(bg);
  }  // end of main()
```

The PickCubes constructor creates the scene in the normal way, except for the allocation of names to the shapes:

```
  // global
  private BranchGroup sceneBG;


  public PickCubes()
  {
    /*   sceneBG --> pos1TG --> red box (pickable)
                 |
                 --> pos2TG --> blue box (pickable)
                 |
                 --> pos3TG --> green box (pickable)
                 |
                 --> pos4TG --> car (pickable)
    */
      : // standard code for shape creation ...

    Box greenBox = new Box(0.5f, 0.5f, 0.5f,
                           Simple3D.color("green"));
    greenBox.setName("green box");
    TransformGroup pos3TG = Simple3D.setTranslation(2,0.5,0);
    pos3TG.addChild(greenBox);

    BranchGroup modelBG = Simple3D.loadModel("models/car/car.obj");
    TransformGroup pos4TG = Simple3D.setTranslation(0,0,2);
    pos4TG.addChild(modelBG);
```

```
        // loadModel() calls setName() with the name of the file

    // build the scene graph
    sceneBG = Simple3D.createBG();
    sceneBG.addChild(pos1TG);
    sceneBG.addChild(pos2TG);
    sceneBG.addChild(pos3TG);
    sceneBG.addChild(pos4TG);
  }  // end of PickCubes()


  public BranchGroup getScene()
  {  return sceneBG;  }
```

It's not necessary to call setName() for the model since Simple3D.loadModel() does that, storing the filename.

PickCubes implements PickUpdater which means it must have a definition for the picked() method called by PickBehavior:

```
  public void picked(String id, Point3d pt)
  // this method is called by PickBehavior;
  // Note: the intercept may be null, as in the case of the car model
  {
    if (pt == null)
      System.out.println(id + " picked");
    else
      Simple3D.printTuple(id + " picked at ", pt);
  }  // end of picked()
```

1.2. Reacting to Picks

PickActs.java utilizes picking to trigger three different animations: the rotation of a red cube, and the shrinking/growing of a green sphere (the scene is shown in Figure 18).

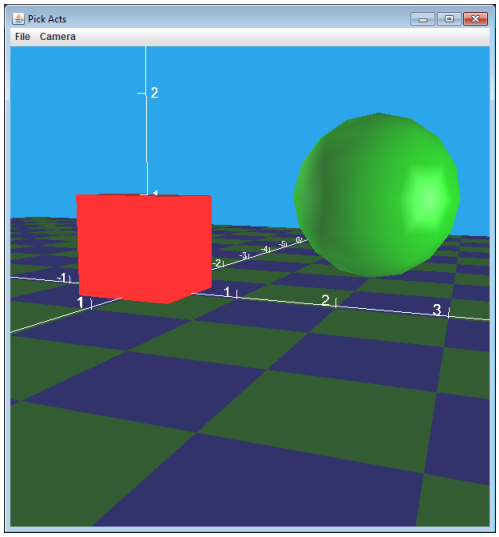

Figure 18. Picking an Animation.

The animations are implemented using Simple3D's FireAlpha (which was introduced back in section 2) and three interpolators. The sphere's scene graph is:

```
sceneBG --> posSTG -->  sphereTG --> sphere
                   |
                   ----> shrinking interpolator
                   |
                   ----> growing interpolator
```

The corresponding code in the PulseActs constructor is:

```
    Sphere sphere = new Sphere(1f, Simple3D.color("green"));
    sphere.setName("sphere");

    TransformGroup sphereTG = new TransformGroup();
    sphereTG.addChild(sphere);
    animateSphere(sphereTG);  // build both interpolators

    // position the sphere
    TransformGroup posSTG = Simple3D.setTranslation(2,1,-2);
    posSTG.addChild(sphereTG);


    // top-level of the scene graph
    sceneBG = Simple3D.createBG();
    sceneBG.addChild(posSTG);
```

The sphere is assigned a name, which is used to identify it at pick time.

The sphere's interpolators and their FireAlpha timers are created in animateSphere():

```
  // globals for animation
  private FireAlpha shrinkAlpha, growAlpha;


  private void animateSphere(TransformGroup sphereTG)
  /* make the sphere shrink or grow when picked using two
     ScaleInterpolators controlled by FireAlpha objects
  */
  {
    // shrinking animation: radius goes from 1 to 0.5
    shrinkAlpha = new FireAlpha(2000); // 2 sec duration
    ScaleInterpolator shrinking =
              new ScaleInterpolator(shrinkAlpha.getAlpha(), sphereTG,
                                    new Transform3D(), 1.0f, 0.5f);
                        // scaling applied to an identity matrix
    shrinking.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);

    sphereTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    sphereTG.addChild(shrinking);
```

```
    // growing animation: radius goes from 0.5 to 1
    growAlpha = new FireAlpha(2000); // 2 sec duration
    ScaleInterpolator growing =
              new ScaleInterpolator(growAlpha.getAlpha(), sphereTG,
                                    new Transform3D(), 0.5f, 1.0f);
    growing.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
    sphereTG.addChild(growing);
  }  // end of animateSphere()
```

The main() function creates a PickActs object and links it to PickBehavior in the same way as the main() function in PickCubes. When the user clicks on a shape, the behavior calls picked() in PickActs:

```
  // in PickActs
  // should the sphere shrink or grow?
  private boolean isShrinking = true;


  public void picked(String id, Point3d pt)
  // this method is called by PickBehavior
  {
     if (id == null)
       return;
    if (id.equals("box"))
      boxAlpha.fire();   // rotate the cube
    else if (id.equals("sphere")) {
      // grow or shrink the cube (when previous change has finished)
      if (isShrinking && !growAlpha.isRunning()) {
        shrinkAlpha.fire();
        isShrinking = false;  // Next time the sphere should grow
      }
      else if (!isShrinking && !shrinkAlpha.isRunning()) {
        growAlpha.fire();
        isShrinking = true; // Next time the sphere should shrink
      }
    }
  }  // end of picked()
```

A mouse click on the sphere causes the second branch of the second if-statement to be executed which, depending on the current value of the isShrinking global, will either call the FireAlpha for shrinking or growing.

Note that FireAlpha.fire() is protected by tests of FireAlpha.isRunning() which ensure that an interpolator is only started when it's finished an earlier interpolation. This means that shrinking or growing cannot be interrupted by another animation.

1.3. Picking for Movement

Using picking to move an object around the scene, by dragging it or rotating it with the mouse, is such a common task that the Java 3D utilities package contains three

Pick behaviors for that purpose – PickTranslateBehavior, PicRotateBehavior, and PickZoomBehavior. These in turn rely on three built-in mouse behaviors, MouseTranslate, MouseRotate, and MouseZoom. These behaviors are woken by a mouse dragging event and then examine the `alt` and `meta` keys to decide how to respond, which can be problematic.

This approach works well on Mac OS and Linux, but fails on Windows 10. The issue is that that Java maps `meta` to the `windows` key on Windows (i.e. the key on the left of the space bar on most keyboards), and in Windows 10 this key's state is no longer passed to Java. In other words, any code relying on the pressing/clicking of the `meta` key is never going to execute.

The simplest solution for Java 3D mouse behaviors is to alter their code to test for another key. For example, calls to MouseEvent.isMetaDown() can be replaced by isShiftDown() or isControlDown(). Incidentally, MouseEvent is a Java class, not a part of the Java 3D API.

However, I gallantly decided to make more drastic changes. My versions of MouseTranslate and MouseRotate (the cleverly named MyMouseTranslate and MyMouseRotate) still wake up when the mouse is dragged, but now check whether a particular button on the mouse has being pressed. The left button causes MyMouseTranslate to start translating a shape, while the right button triggers rotation in MyMouseRotate.

MyMouseTranslate and MyMouseRotate are included as part of Simple3D.jar, so I'll spare you a detailed examination of their code since the source is available inside the JAR file.

MouseTranslate (and MyMouseTranslate) only translates a shape in the XY plane. For movement along the z-axis (which Java 3D calls 'zooming') the MouseZoom class needs to be employed. This also suffers from the `meta` key problem, but Java 3D fortunately includes an alternative class called MouseWheelZoom which triggers movement along the z-axis when the mouse wheel is turned. It pairs nicely with MyMouseTranslate and MyMouseRotate.

As I mentioned before, Java 3D's PickTranslateBehavior, PicRotateBehavior, and PickZoomBehavior work closely with MouseTranslate, MouseRotate, and MouseZoom, so their functionality needs to be recoded. I decided to combine the three picking behaviors, which are closely related, into a single class called PickMoverBeh. which will be described in this section.

All these changes are summarized in Figure 19.

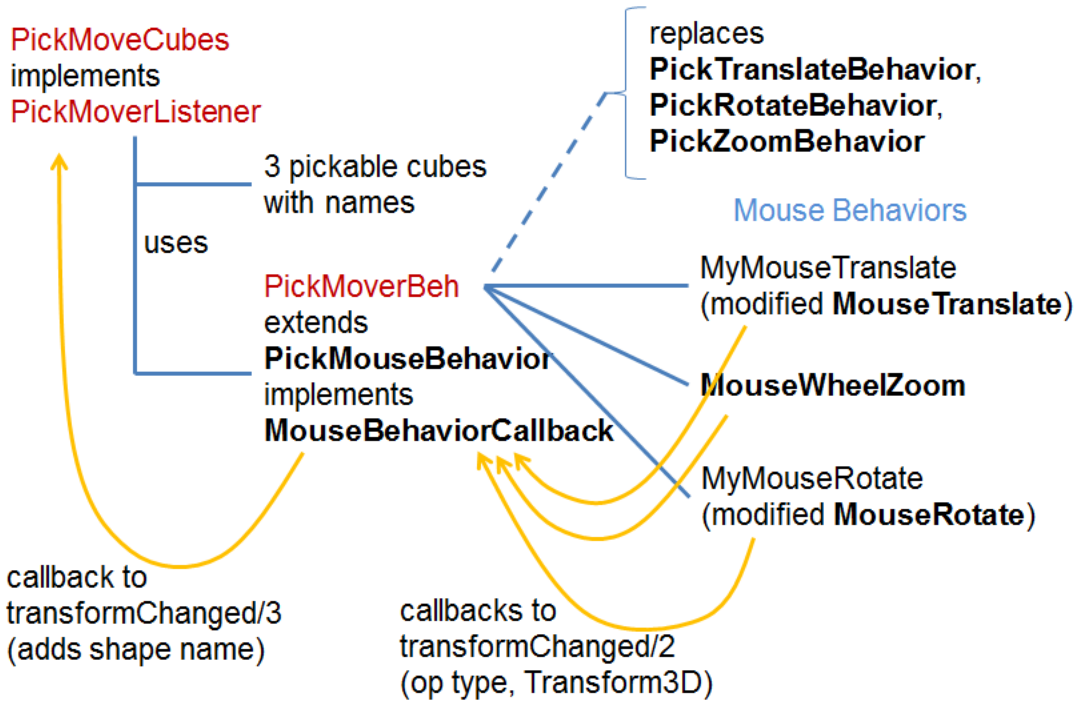

Figure 19. The Classes Used by PickMoveCubes.

The classes highlighted in bold are Java 3D utility classes. My classes are shown in red.

The PickMoveCubes application supports the translation and rotation of the three cubes shown in Figure 20. The left-hand screenshot shows the cubes' initial orientation, and the right-hand picture was taken after the cubes were moved around.

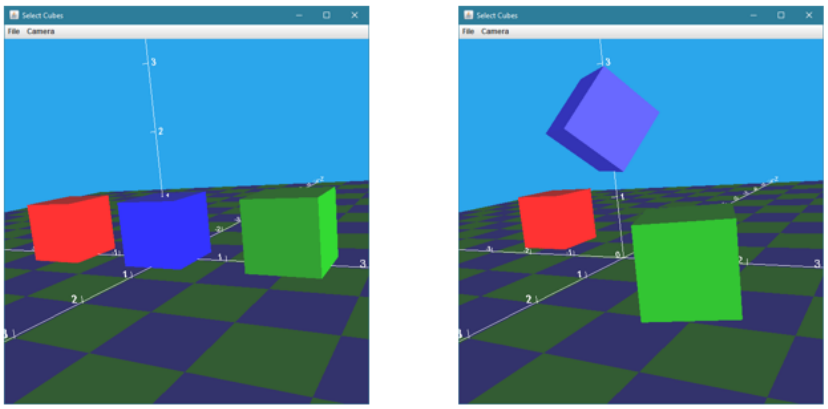

Figure 20. The Cubes in PickMoveCubes.

1.3.1. Setting up the Scene

The trick to moving a picked object is to utilize a 'pick' TransformGroup linked to each shape. When the shape is picked at run time, the associated TransformGroup is used to translate or rotate the shape below it.

PickMoveCubes creates a scene containing three cubes spaced out along the x-axis. Each one has its own 'pick' TransformGroup, which means that the scene graph will look something like:

```
sceneBG --> 3 pickable boxes
      |
      | --> posTG1 --> pickTG1 --> red box
      |
      | --> posTG2 --> pickTG2 --> blue box
      |
      | --> posTG3 --> pickTG3 --> green box
      |
      ----> PickMoverBeh
```

The corresponding code in PickMoveCubes:

```
public PickMoveCubes()
{
  Simple3D j3d = new Simple3D("Select Cubes", true, true);
  /*
       sceneBG --> 3 pickable boxes
            |
            ----> PickMoverBeh
  */
  // build the scene graph
  BranchGroup sceneBG = Simple3D.createBG();
  sceneBG.addChild( makeBox(-2, 0, "red"));    //(x,z) pos on floor
  sceneBG.addChild( makeBox(0, 0, "blue"));
  sceneBG.addChild( makeBox(2, 0, "green"));

  j3d.cameraMouseEnable(false);
        // disables mouse effects on the camera

  PickMoverBeh pickMove =
             new PickMoverBeh(j3d.canvas3D, sceneBG, this);
  sceneBG.addChild(pickMove);

  j3d.rootBG.addChild(sceneBG);
}  // end of PickMoveCubes()
```

Each scene graph branch for a cube is built by a call to makeBox():

```
private TransformGroup makeBox(double x, double z, String colorStr)
/*
    posTG --> pickTG --> box (pickable)
*/
{
  Box b = new Box(0.5f, 0.5f, 0.5f, Simple3D.color(colorStr));
  b.setName(colorStr + " box");  // ID returned during picking
```

```
    // position the box on the XZ plane
    TransformGroup posTG = Simple3D.setTranslation(x,0.5,z);

    // Create a second transform group node
    // which can report its picking changes.
    TransformGroup pickTG = Simple3D.createTG();
    pickTG.setCapability(TransformGroup.ENABLE_PICK_REPORTING);

    // build scene graph branch
    pickTG.addChild(b);
    posTG.addChild(pickTG);
    return posTG;
  }  // end of makeBox()
```

The "pos" TransformGroup in each branch is employed to position a cube along the x-axis. When PickMoverBeh is woken up by a cube being picked, it passes the associated "pick" TransformGroup to MyMouseTranslate, MouseWheelZoom, or MyMouseRotate for modification.

Although the changes to the 'pick TransformGroups could be performed without affecting the top-level application, it's more useful to have some notification mechanism in place. That's implemented as a chain of callbacks (shown as orange lines in Figure 19) which send information from the mouse behaviors to PickMoverBeh and then to PickMoveCubes.

PickMoveCubes implements PickMoverListener which defines transformChanged():

```
  public void transformChanged(String id, int type, Transform3D t3d)
  // called by PickMoverBeh when there is a move
  {
    if (id != null) {
      if (type != PickMoverListener.NO_PICK) {
        System.out.println(id + ": " + getPickType(type));
        if (t3d != null)
          Simple3D.printMatrix(t3d);
        else
          System.out.println("Transform is null");
      }
    }
  }  // end of transformChanged()
```

PickMoveListener defines four constants (ROTATE, TRANSLATE, ZOOM, and NO_PICK) which are passed as the value of the type argument. The Transform3D argument is the transformation just applied to the 'pick' TransformGroup by the mouse behavior. The id variable contains the name of the affected shape.

A crucial line in the constructor for PickMoveCubes is:

```
    j3d.cameraMouseEnable(false);
```

It disables Simple3D's camera mouse controls which would otherwise also be woken when the user dragged the mouse inside the scene. However, Simple3D also supports

keyboard-based controls for the camera, which are unaffected, so they can still be employed to navigate through the scene.

1.3.2 The Picking Behavior

PickMoverBeh is a subclass of PickMouseBehavior in the same way as the earlier PickBehavior example. This means that it must implement updateScene() which is called when an object is picked. PickMoverBeh then utilizes one of the mouse behaviors (MyMouseTranslate, MouseWheelZoom, or MyMouseRotate) to modify the relevant 'pick' TransformGroup node above the picked object.

The mouse behaviors are initialized in the PickMoverBeh() constructor:

```
  public PickMoverBeh(Canvas3D canvas, BranchGroup root,
                                          PickMoverListener pml)
  // Initialize mouse behaviors for moving a picked object
  {
    super(canvas, root, Simple3D.INFINITE_BOUNDS);
    pmListener = pml;

    mouseTrans = new MyMouseTranslate(MouseBehavior.MANUAL_WAKEUP);
    // mouseTrans.setControl(true);   // use <CTRL> key
    mouseTrans.setTransformGroup(currGrp);
    mouseTrans.setupCallback(this);
    currGrp.addChild(mouseTrans);

    mouseZoom = new MouseWheelZoom(MouseBehavior.MANUAL_WAKEUP);
    mouseZoom.setTransformGroup(currGrp);
    mouseZoom.setupCallback(this);
    currGrp.addChild(mouseZoom);

    mouseRot = new MyMouseRotate(MyMouseRotate.MANUAL_WAKEUP);
    mouseRot.setControl(true);   // use <CTRL> key
    mouseRot.setTransformGroup(currGrp);
    mouseRot.setupCallback(this);
    currGrp.addChild(mouseRot);

    mouseTrans.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
    mouseZoom.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
    mouseRot.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);

    this.setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
  }  // end of PickMoverBeh()
```

One of the new elements of MyMouseTranslate and MyMouseRotate is a setControl() method which causes the behaviors to examine if the `ctrl` key is being pressed when the mouse is dragged.

Normally, MyMouseTranslate treats a mouse drag as a translation across the XY plane, with the shape's x- and y-axis positions changing at the same time. This can make it rather hard to place a shape exactly, and so if setControl() is set to true then a

mouse drag only affects the x-axis position. However, if the `ctrl` key is pressed during a drag then only the y-axis location is changed.

A similar modification can be applied to MyMouseRotate. Normally, a mouse drag will affect both the x- and y-axis orientations at the same time. If setControl() is set to true (as in the code above) then a mouse drag only rotates the shape around the y-axis, and the `ctrl` key must be pressed to change that to an x-axis revolution.

PickMoverBeh.updateScene() extracts the picked shape's ID and its 'pick' TransformGroup, and calls the relevant mouse behavior to modify that TransformGroup:

```
  @Override
  public void updateScene(int xpos, int ypos)
  /* Examine the mouse to decide whether to use
     the mouse behavior for translation, zooming, or rotation
     on the picked TransformGroup.
  */
  {
    TransformGroup tg = null;

    pickCanvas.setShapeLocation(xpos, ypos);
    PickResult pr = pickCanvas.pickClosest();
    currID = Simple3D.pickID(pr);     // get ID of picked object

    if ((pr != null) &&
        ((tg = (TransformGroup)pr.getNode(
                          PickResult.TRANSFORM_GROUP)) != null) &&
         (tg.getCapability(TransformGroup.ALLOW_TRANSFORM_READ)) &&
         (tg.getCapability(TransformGroup.ALLOW_TRANSFORM_WRITE))) {
              // get the pickable TransformGroup of the picked shape

      if ((mevent.getModifiersEx() &
           InputEvent.BUTTON1_DOWN_MASK) != 0) {
        // left button pressed
        mouseTrans.setTransformGroup(tg);
        mouseTrans.wakeup();
      }

      else if ((mevent.getModifiersEx() &
                InputEvent.BUTTON2_DOWN_MASK) != 0) {
        // middle button (the mouse wheel) pressed
        mouseZoom.setTransformGroup(tg);
        mouseZoom.wakeup();
      }

      else if ((mevent.getModifiersEx() &
                InputEvent.BUTTON3_DOWN_MASK) != 0) {
        // right button pressed
        mouseRot.setTransformGroup(tg);
        mouseRot.wakeup();
      }
    }
    else if (pmListener != null)
```

```
      pmListener.transformChanged(currID,
                                  PickingCallback.NO_PICK, null);
  } /* updateScene */
```

When the mouse behaviors were initialized in PickMoverBeh(), they were configured to call the two argument transformChanged() in PickMoverBeh after they had finished a transformation:

```
  @Override
  public void transformChanged(int type, Transform3D t3d)
  // called by the mouse behaviors;
  // this method calls the 3-arg version of transformChanged()
  {  pmListener.transformChanged(currID, type, t3d);  }
```

PickMoverBeh adds the ID of the picked shape, and calls the listener function in PickMoveCubes.

2. Collision Detection

Collision detection is implemented using a Behavior class that's triggered by one or more collision wakeup conditions – WakeupOnCollisionEntry, WakeupOnCollisionExit, or WakeupOnCollisionMovement (see Figure 10).

The determination of what shapes have collided relies on the examination of a SceneGraphPath object that's supplied as part of the trigger event. It also helps to assign names to the shapes, using a similar approach to in picking.

The HitSpheres example involves three colored spheres orbiting the y-axis at the origin, each at a distance of 3 units, moving at different speeds, and starting from different points in the orbit. A 'sun' is located at the origin just to make things look more astronomical; it doesn't play any role in the collision code. The setup is shown in Figure 21.

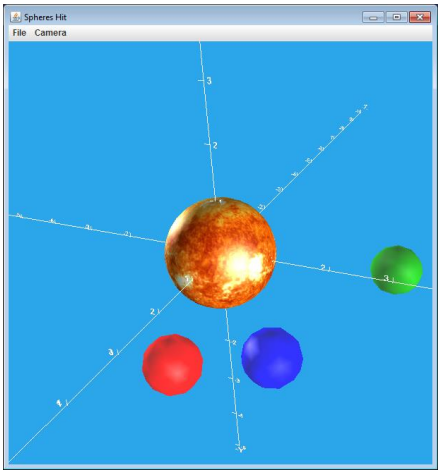

Figure 21. Orbiting Spheres on a Collision Course.

A CollisionBeh behavior object is assigned to the red sphere and another to the blue sphere, which report when their sphere collides with another. Typical output is:

```
red hit green at ( 1.036, 0.000, -2.815 )
  red left green
red hit blue at ( 2.159, 0.000, 2.083 )
blue hit red at ( 1.294, 0.000, 2.706 )
  blue left red
  red left blue
red hit green at ( 1.038, 0.000, -2.815 )
  red left green
red hit blue at ( 2.159, 0.000, 2.083 )
blue hit red at ( 1.290, 0.000, 2.708 )
  blue left red
  red left blue
```

Inside HitSpheres.java, the code for the red sphere is:

```
    // orbiting red sphere
    Sphere redSphere = new Sphere(0.4f, Simple3D.color("red"));
    redSphere.getShape().setName("red");   // collision naming
    TransformGroup pos1TG = Simple3D.setTranslation(0,0,-3);
    pos1TG.addChild(redSphere);
    TransformGroup rot1TG = Simple3D.orbit(pos1TG, 4000);

    // detect when the red sphere collides with something
    CollisionBeh cbRed =  new CollisionBeh(redSphere);
```

The CollisionBeh constructor sets up the collision boundary for its shape, which can use a BoundarySphere object in this case:

```
  // globals
  private Sphere sphere;
  private String name;   // of this sphere


  public CollisionBeh(Sphere s)
  {
    sphere = s;
    sphere.setCollisionBounds(
          new BoundingSphere( new Point3d(0,0,0),
                              sphere.getRadius()));
    name = sphere.getShape().getName();
           // store the name stored in the sphere's Shape3D
    setSchedulingBounds(Simple3D.INFINITE_BOUNDS);
  }
```

initialize() configures a trigger that wakes up for either a collision entry or an exit:

```
  // global
```

```
  private WakeupCondition wakes;

  public void initialize()
  {
    WakeupCriterion[] crits = new WakeupCriterion[2];
    // collision entry and exit
    crits[0] = new WakeupOnCollisionEntry(sphere);
    crits[1] = new WakeupOnCollisionExit(sphere);
    wakes = new WakeupOr(crits);
    wakeupOn(wakes);
  }
```

processStimulus() determines which of the two conditions caused it to fire, and examines the trigger information:

```
  public void processStimulus(Iterator<WakeupCriterion> wCrits)
  {
    WakeupCriterion wakeUp;

    while(wCrits.hasNext()) {
      wakeUp = wCrits.next();
      if (wakeUp instanceof WakeupOnCollisionEntry) {
       // collision entry
        SceneGraphPath sgp = ((WakeupOnCollisionEntry) wakeUp).
                                              getTriggeringPath();
        Node hitNode = sgp.getObject();
        if (hitNode.getName() != null) {  // ignore unnamed shapes
          Transform3D t3d = sgp.getTransform();
          Vector3d trans = new Vector3d();
          if (t3d != null)
            t3d.get(trans);
          System.out.println(name + " hit " +
                             hitNode.getName() +
                             " at " + Simple3D.tupleStr(trans));
        }
      }
      else if (wakeUp instanceof WakeupOnCollisionExit) {
        // collision exit
        SceneGraphPath sgp = ((WakeupOnCollisionExit) wakeUp).
                                              getTriggeringPath();
        Node hitNode = sgp.getObject();
        if (hitNode.getName() != null)
          System.out.println("  " + name + " left " +
                                      hitNode.getName());
            // coordinates not reported on collision exit
      }
      wakeupOn(wakes);
    }
  }  // end of processStimulus()
```

getTriggeringPath() retrieves a scene graph path for the object that was hit. The information will vary depending on what object was encountered, and so the simplest

thing is to ensure that every object that might be involved in a collision is assigned a name. This also means that objects in the scene graph path with no names can be ignored.

Upon collision entry, processStimulus() prints the names the shapes involved and the transform associated with the shape (which may be null). If a transform is available then it will be specified in world coordinates, which can be accessed with a get() call.