Chapter 20.5. When Worlds Collide

An important physics-related gaming problem is how to handle object collisions. It can be difficult to decide how a 3D object should rebound or bounce, especially when mass, linear and angular velocity, gravity, friction, and other forces are taken into account.

The solution is to utilize a physics API, such as ODE (http://www.ode.org/) to do the heavy-lifting for you. Create physics-based models of the objects, and let ODE calculate how they should move. Position and orientation details can be read from the models, and used to update the game's graphical entities.

Applications of this type utilize a *dual-model* approach, with (mostly) separate physics-based and graphical elements.

I'll explain this approach in more detail with a simple example: balls bouncing off each other and the walls of a box. Figure 1 shows the Balls3D application in action.

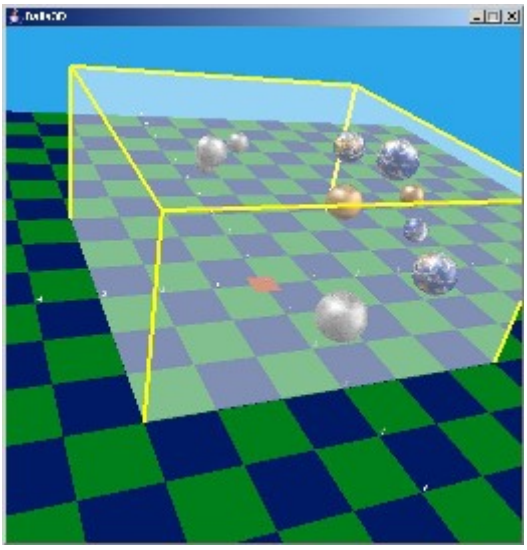

Figure 1. When Worlds Collide.

Figure 1 (and this chapter) are entitled "When Worlds Collide" because the balls are textured wrapped with images of the earth, the moon, and mars.

1. Odejava and ODE

I'll be using Odejava (http://odejava.org), a Java binding for the ODE physics API (http://www.ode.org/). ODE's application domain is the physics of articulated rigid bodies, such as a person's skeleton and muscles, or a car's suspension.

A rigid body's properties include its position, orientation, mass, and any applied forces, accelerations, and velocities (e.g. gravity and friction). Complex bodies are built by connecting simpler bodies together using different types of *joints*, such as hinges, balls and sockets, and pistons.

ODE bodies don't have a visual appearance; it's best to think of them as sets of equations applied to invisible, connected masses. An ODE system is executed by advancing time in discrete steps: at each time step, the equations are applied to the bodies, changing their position and orientation.

ODE also has a collision detection engine, based around *geoms* (geometries), which have shape and size attributes. Basic geoms include spheres, boxes, and meshes, with more complex geoms built by combining simpler ones. An ODE body can be involved in collisions if it's been assigned a geom.

As an ODE simulation progresses, body collisions are detected when their associated geoms intersect. A contact is represented by a temporary *contact joint*, which permits velocities and accelerations to be transferred between the bodies.

Although geoms have shape, size, position, and orientation, they don't have a visual representation. This can make debugging an ODE program somewhat difficult since there's no built-in view of the bodies and geoms. However, it also means that ODE can be utilized with a variety of visualization APIs. For example, Odejava has been employed with Java 3D, Xith3D, and jME (the jMonkey Engine).

It's possible to create geoms that have no corresponding body, which can't be affected by velocities or accelerations. Bodyless geoms are often used to represent walls and floors, which can be collided with, but can't move themselves.

A typical ODE (Odejava) program has two separate models: a graphical (perhaps 3D) scene, and a physics-based representation of (some of) the objects in that scene.

One advantage of this approach is that the two models can be developed independently. For example, the visual versions of the container and balls in Balls3D were developed first, without any physics elements. The physics was added after the graphical parts had been completed.

There's no requirement that the physics components model everything in the scene. In Balls3D only the container and the balls have Odejava representations, since they're the only things involved in collisions. Odejava isn't needed for the checkboard floor, the axis labels, or the blue background.

Even when an object does have both visual and physics-based components, they don't need to have the same complexity. For instance, a game monster may be represented by a large 3DS model but a comparatively simple ODE body made of 10-20 hinged parts.

Odejava is a Java wrapper over ODE's C/C++ API. Odejava offers both a low-level interface and a more object oriented API; I'll be using the latter.

1.1. Installing Odejava

The Odejava API can be obtained from https://odejava.dev.java.net/. The "Odejava snapshot" is available from the "2006-01-15_cvs (1)" folder, accessed via the website's "Documents & files" menu item.

The snapshot is a zipped file containing Odejava's platform-independent JARs (and lots more stuff as well). Extract odejava.jar from the odejava\odejava subdirectory,

and log4j-1.2.9.jar from odejava\odejava\lib. You may also want the vecmath.jar file, but Java 3D already contains that JAR, so I didn't need it.

The platform-dependent elements are a separate download from https://odejava.dev.java.net/. Grab the "Windows Binaries" zip file from the "004-10-30 natives (4)" folder, and extract the odejava.dll file from its windows\release subdirectory.

At this point, you should have three files: odejava.dll, odejava.jar, and log4j-1.2.9.jar. You may wonder why log4j logging is need, and so do I :). The two JARs should be copied to <JAVA HOME>\jre\lib\ext, and the DLL to <JAVA HOME>\jre\bin. On my machine, <JAVA HOME> is c:\Program Files\Java\jdk1.5.0_06.

If you've installed the JRE as well, then also copy the JARs and DLL into the corresponding directories below <JRE HOME>. On my machine, <JRE HOME> is c:\Program Files\Java\jre1.5.0_06.

One of Odejava's optional extras are classes which can semi-automatically map geoms to displayable entities in a particular graphics API. The main developer of Odejava, William Denniss, is also involved in Xith3D, and so the Odejava snapshot contains a Xith3D subdirectory holding classes for mapping geoms to Xith3D. A similar API for Java 3D, by Paul Byrne, can be found at https://odejava.dev.java.net/ in the "contrib (0)\java3d-binding (1)" folder.

These mapping classes are rather difficult to understand and use, and perhaps better suited for complex geometries. I won't be using the Java 3D binding, since Balls3D only utilizes box and spheres.

1.2. Documentation, Examples, and Online Help

The Odejava API documentation can be downloaded as a zip file from http://odejava.org/OdejavaDocs, or can be viewed online at http://odejava.org/javadoc/. There's also a version in the Odejava snapshot, but it's combined with Xith3D API information, which I don't need.

It's a good idea to download the ODE user guide, which explores ODE concepts and its API in detail. This is useful since much of the Odejava API is closely related to ODE. The user guide can be found at http://www.ode.org/ode-docs.html, in HTML or PDF formats.

The Odejava snapshot contains several example folders in the subdirectory odejava\odejava\src\org\odejava\test\. The simple\ folder is the place to start, and my Bouncer.java example (discussed in the next section) is a variant of its HighLevelApiExample.java. The car\ folder contains an example that models how a car moves over a bumpy terrain.

The best source for help on Odejava is the "Game Physics" forum at javaGaming.org (http://www.javagaming.org/forums/index.php?board=11.0).

For ODE advice, there's community page at the ODE site (http://www.ode.org/community.html), which includes a link to an ODE mailing list

archive at http://q12.org/pipermail/ode/. Unfortunately, there's no search feature at the q12.org site, but Google can be employed instead by including "site:q12.org" in search queries (e.g. type "xode site:q12.org" to scan the mailing list for references to xode).

Another ODE forum, this one directly searchable, is at http://ode.petrucci.ch.

The gamedev.net site has an informative physics forum (http://www.gamedev.net/community/forums/forum.asp?forum_id=20), and other general physics resources. Another good forum is at http://www.continuousphysics.com/Bullet/phpBB2/index2.php.

2. Bouncing a Ball

Bouncer.java is a small Odejava example that doesn't use any Java graphics (thereby keeping the code nicely simple). A sphere drops onto a floor, and its position and other information is printed out periodically. Whenever the sphere hits the floor, it bounces.

The simulation continues until 1000 steps have been carried out, then stops.

The following partial output shows what happens when the ball (which has a radius of 1) drops from a height of 4 meters, under a gravity of 0.2 m/s^2.

The "Pos" value is the position of the ball's center, and the "angle" is the ball's rotation around the y-axis. Information is printed every 10 simulation steps.

```
> java Bouncer
0    [main] INFO  odejava  - Odejava version 0.2.4
Step 10) Pos: (0, 4, 0), angle: 0
Step 20) Pos: (0, 3.9, 0), angle: 0
Step 30) Pos: (0, 3.8, 0), angle: 0
Step 40) Pos: (0, 3.6, 0), angle: 0
Step 50) Pos: (0, 3.4, 0), angle: 0
Step 60) Pos: (0, 3.1, 0), angle: 0
Step 70) Pos: (0, 2.8, 0), angle: 0
Step 80) Pos: (0, 2.4, 0), angle: 0
Step 90) Pos: (0, 2, 0), angle: 0
Step 100) Pos: (0, 1.5, 0), angle: 0
Step 110) Pos: (0, 1, 0), angle: 0
Step 120) Pos: (0, 1.4, 0), angle: 0
Step 130) Pos: (0, 1.8, 0), angle: 0
Step 140) Pos: (0, 2.1, 0), angle: 0
  :
Step 920) Pos: (0, 1, 0), angle: 0
Step 930) Pos: (0, 1, 0), angle: 0
Step 940) Pos: (0, 1, 0), angle: 0
Step 950) Pos: (0, 1, 0), angle: 0
Step 960) Pos: (0, 1, 0), angle: 0
Step 970) Pos: (0, 1, 0), angle: 0
Step 980) Pos: (0, 1, 0), angle: 0
Step 990) Pos: (0, 1, 0), angle: 0
```

The ball drops from (0,4,0) until it reaches the floor at step 110, and then bounces up to a reduced height. The bounces continue, but by step 990 the ball is at rest.

The Odejava API is simple enough, that it's easy to modify the ball's behaviour. In the second run shown below, the sphere now has a linear velocity of 2 m/s along the x-axis, and an angular velocity around the y-axis of 1 m/s. Friction is also included to slow the ball down.

```
> java Bouncer
0    [main] INFO  odejava  - Odejava version 0.2.4
Step 10) Pos: (1, 4, 0), angle: 28.6
Step 20) Pos: (2, 3.9, 0), angle: 57.3
Step 30) Pos: (3, 3.8, 0), angle: 85.9
Step 40) Pos: (4, 3.6, 0), angle: 114.6
Step 50) Pos: (5, 3.4, 0), angle: 143.2
Step 60) Pos: (6, 3.1, 0), angle: 171.9
Step 70) Pos: (7, 2.8, 0), angle: 200.5
Step 80) Pos: (8, 2.4, 0), angle: 229.2
Step 90) Pos: (9, 2, 0), angle: 257.8
Step 100) Pos: (10, 1.5, 0), angle: 286.5
Step 110) Pos: (11, 1, 0), angle: 315.1
Step 120) Pos: (11.7, 1.4, -0), angle: 318
Step 130) Pos: (12.4, 1.8, -0), angle: 281.5
Step 140) Pos: (13.2, 2.1, -0), angle: 236.8
Step 150) Pos: (13.9, 2.4, -0), angle: 190.5
Step 160) Pos: (14.6, 2.6, -0), angle: 143.9
  :
Step 920) Pos: (69.2, 1, -0.7), angle: 36.1
Step 930) Pos: (70, 1, -0.7), angle: 61.5
Step 940) Pos: (70.7, 1, -0.7), angle: 104.8
Step 950) Pos: (71.4, 1, -0.7), angle: 151.1
Step 960) Pos: (72.1, 1, -0.7), angle: 198.1
Step 970) Pos: (72.8, 1, -0.7), angle: 244.8
Step 980) Pos: (73.5, 1, -0.7), angle: 289.2
Step 990) Pos: (74.3, 1, -0.7), angle: 321.8
```

The ball bounces as before, and moves at a slowly decreasing speed to the right along the x-axis. The ball also drifts slightly down the negative -z-axis (ending at -0.7), due to its spin around the y-axis and the presence of friction.

The rotation angle increases until the ball bounces, then the change in the ball's direction causes the angle to start decreasing. This continues until the ball begins falling again, at which time the angle starts increasing again.

Much more tweaking is possible, such as changing the gravity, the bounce velocity, and the friction applied to the ball.

2.1. Three Stage Simulation

An Odejava simulation passes through three stages: set-up, execution, and clean up, which can be seen in the Bouncer() constructor.

```
private static final int MAX_STEPS = 1000;  // simulation steps
private DecimalFormat df;      // used for reporting


public Bouncer()
{
  df = new DecimalFormat("0.#");  // 1 dp
```

```
  // set-up
  Odejava.getInstance();
  initWorld();
  initStatics();
  initDynamics();

  simulate(MAX_STEPS);  // carry out the simulation

  cleanUp();
} // end of Bouncer()
```

The set-up is divided into three parts: the initialization of the rigid body and collision detection engines (in initWorld()), and the creation of static and dynamic objects. A static object is a geom without a body, so it can't move. A dynamic object is a body, so it can be affected by forces, accelerations, and velocities. A body may have an associated geom to let it get involved in collisions.

simulate() steps the simulation forward until MAX_STEPS (1000) steps have been carried out. The clean up phase closes down the ODE engines.

2.2. Initializing the Engines

The rigid body engine is accessed through the World class, while collision detection utilizes HashSpace, JavaCollision, and Contact.

```
// globals
private World world;

// for collisions
private HashSpace collSpace;      // holds collision info
private JavaCollision collCalcs;  // calculates collisions
private Contact contactInfo;      // for accessing contact details


private void initWorld()
{
  world = new World();
  world.setGravity(0f, -0.2f, 0);  // down y-axis (9.8 is too fast)

  // max interactions per step (bigger is more accurate, but slower)
  world.setStepInteractions(10);

  // set step size (smaller is more accurate, but slower)
  world.setStepSize(0.05f);

  // create a collision space for the world's geoms
  collSpace = new HashSpace();

  collCalcs = new JavaCollision(world);   // collision calculations
  contactInfo = new Contact( collCalcs.getContactIntBuffer(),
                             collCalcs.getContactFloatBuffer());
}  // end of initWorld()
```

The World object acts as an environment for the bodies and joints, and manages gravity and step-related parameters.

The HashSpace object is a collision space for geoms. Collision testing involves examining pairs of geoms in the space to see if they intersect. It's possible to create multiple collision spaces, which improves collision testing performance since testing is only done between geoms within the same space.

JavaCollision performs the collision testing on geoms in a particular space. The tests generate a list of contact points, and the details relating to a particular point can be manipulated with the Contact object, contactInfo.

2.3. Initializing Static Objects

Static objects are geoms with no bodies, so the simulation can't change the objects' positions or orientations. Static objects are often employed for modeling boundaries in a scene, such as walls and the floor – objects which can be collided with, but can't move themselves.

The only static entity in Bouncer is the floor, which is represented by a GeomPlane object with its normal facing up the y-axis.

```
private void initStatics()
{
  // the floor, facing upward
  GeomPlane groundGeom = new GeomPlane(0, 1.0f, 0, 0);
  collSpace.add(groundGeom);
}
```

Since the floor plays a part in the collision calculations, it's added to the collision space, collSpace.

2.4. Initializing Dynamic Objects

Dynamic objects have bodies, so the simulation can move them. If a body also has a geom, then it can collide with other objects. The geom defines the object's shape, while the body specifies the object's mass, position, orientation, and the forces, velocities, and accelerations being applied to it.

initDynamics() creates a body and a geom for the ball. Its body has a mass, position, and linear and angular velocities. Odejava has other methods for specifying force and torque, which I haven't used here.

```
// globals for the sphere's body and geom
private Body sphere;
private GeomSphere sphereGeom;


private void initDynamics()
{
  // a sphere of radius 1, mass 1
  sphereGeom = new GeomSphere(1.0f);
  sphere = new Body("sphere", world, sphereGeom);
  sphere.adjustMass(1.0f);

  sphere.setPosition(0, 4.0f, 0);    // starts 4 unit above the floor
  sphere.setLinearVel(2.0f, 0, 0);   // moving to the right
```

```
  sphere.setAngularVel(0, 1.0f, 0);  // velocity around y-axis

  collSpace.addBodyGeoms(sphere);
} // end of initDynamics()
```

The sphere is added to the collision space, since it'll be involved in collisions with the floor.

2.5. Executing the Simulation

simulate() repeatedly calls step() to advance the simulation. Information about the sphere is printed every 10th simulation step.

```
// globals for holding sphere info
private Vector3f pos = new Vector3f();
private AxisAngle4f axisAng = new AxisAngle4f();


private void simulate(int maxSteps)
{
  int step = 1;
  while (stepCount < maxSteps) {
    step();
    // print sphere's details every 10th step
    if ((stepCount % 10) == 0) {
      pos = sphere.getPosition();
      sphere.getAxisAngle(axisAng);
      System.out.println("Step " + stepCount + ") Pos: (" +
               df.format(pos.x) + ", " + df.format(pos.y) + ", " +
               df.format(pos.z) + "), angle: " +
               df.format( Math.toDegrees(axisAng.angle)) );
               // ", quat: " + sphere.getQuaternion() );
    }
    stepCount++;
/*
    // sleep a bit
    try {
      Thread.sleep(50);   // ms
    }
    catch(Exception e) {}
*/
  }
} // end of simulate()
```

The Odejava Body class has numerous get methods, and three are shown in simulate(). Body.getPosition() returns the sphere's current position as a Vector3f object. Body.getAxisAngle() returns the axis angle (a rotation in radians) about a direction vector. Body.getQuaternion() is another way of obtaining the body's rotation, as a quaternion.

The pos and axisAng objects are global so that new, temporary Vector3f and AxisAngle4f objects don't need to be created in every iteration of the simulation loop.

The simulation step (0.05 ms) defined in initWorld() is an elapsed time within the simulation, and has no effect on the running time of the Bouncer application. A

simple way of slowing down the simulate() loop is to call Thread.sleep(), as shown in the commented code.

2.6. Performing a Simulation Step

step() detects collisions, creates contact joints, and advances the simulation.

```
private void step()
{
  collCalcs.collide(collSpace);   // find collisions
  examineContacts();              // examine contact points
  collCalcs.applyContacts();      // apply contacts joint

  world.stepFast();               // make a step
}
```

JavaCollision.collide() examines the geoms in the collision space to determine which ones are touching each other, and their contact points are collected.

examineContacts() converts the interesting contact *points* into contact *joints*. These joints act as a temporary links between the bodies attached to the geoms.

JavaCollision.applyContacts() applies the forces, acceleration, and velocity equations of the newly formed contact joints to their associated bodies, and then the joints are deleted.

Finally the simulation is stepped forward, moving the bodies and their geoms.

2.7. Examining the Contact Points

In Bouncer, the only contact point is the one made by the sphere when it touches the floor. examineContacts() finds the contact point, and converts it into a 'bouncy' contact joint.

```
private void examineContacts()
{
  for (int i = 0; i < collCalcs.getContactCount(); i++) {
    contactInfo.setIndex(i);    // look at the ith contact point

    // if the contact involves the sphere, then make it bounce
    if ((contactInfo.getGeom1() == sphereGeom) ||
        (contactInfo.getGeom2() == sphereGeom)) {
      contactInfo.setMode(Ode.dContactBounce);
      contactInfo.setBounce(0.82f);      // 1 is max bounciness
      contactInfo.setBounceVel(0.1f);    // min velocity for a bounce
      contactInfo.setMu(100.0f);         // 0 is friction-less
    }
  }
} // end of examineContacts()
```

The contact point in the list is accessed via the Contact object, contactInfo.

A contact point involves a pair of geoms, referenced via Contact.getGeom1() and Contact.getGeom2(). If either one is the sphere, then a bouncing contact joint is created. The joint's mode is set to be Ode.dCountBounce, and the amount of

bounciness is specified. If the sphere's velocity is below 0.1 m/s then no bounce will occur, and friction is added in to slow the sphere down.

2.8. Cleaning Up

At the end of the simulation (after 1000 steps have been executed), cleanUp() is called from Bouncer():

```
private void cleanUp()
{
  collSpace.delete();
  collCalcs.delete();
  world.delete();
  Ode.dCloseODE();
}
```

It switches off the rigid body and collection detection engines, and terminates ODE.

3. Visualizing Balls in a Box

Figure 2 shows a snapshot of the Balls3D application:

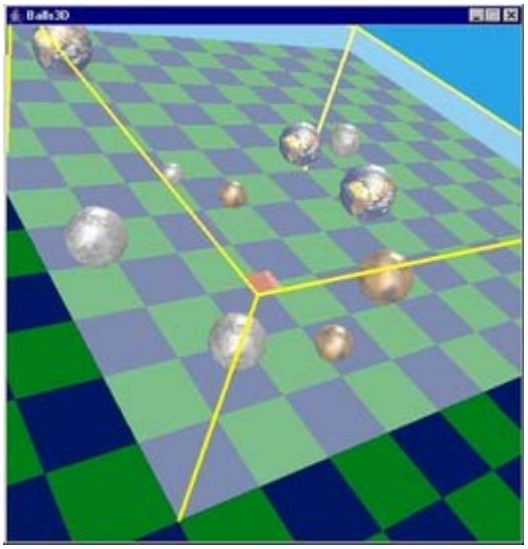

Figure 2. More Ball Collisions.

Most of the visual elements in Balls3D are taken from the Checkers3D example in Chapter 15, including the checkboard floor, the numbered axes, the blue background, the two directional lights, and the OrbitBehavior for moving the user's viewpoint around the scene. I won't bother explaining them again, so please look back at Chapter 15 if that material is unfamiliar to you.

The new elements are the translucent box with yellow edges, and the ten textured bouncing spheres. The box and the spheres have Odejava models, as does the floor, so that collision detection can be carried out.

The class diagrams for Balls3D are shown in Figure 3; only the public methods are listed for each class.

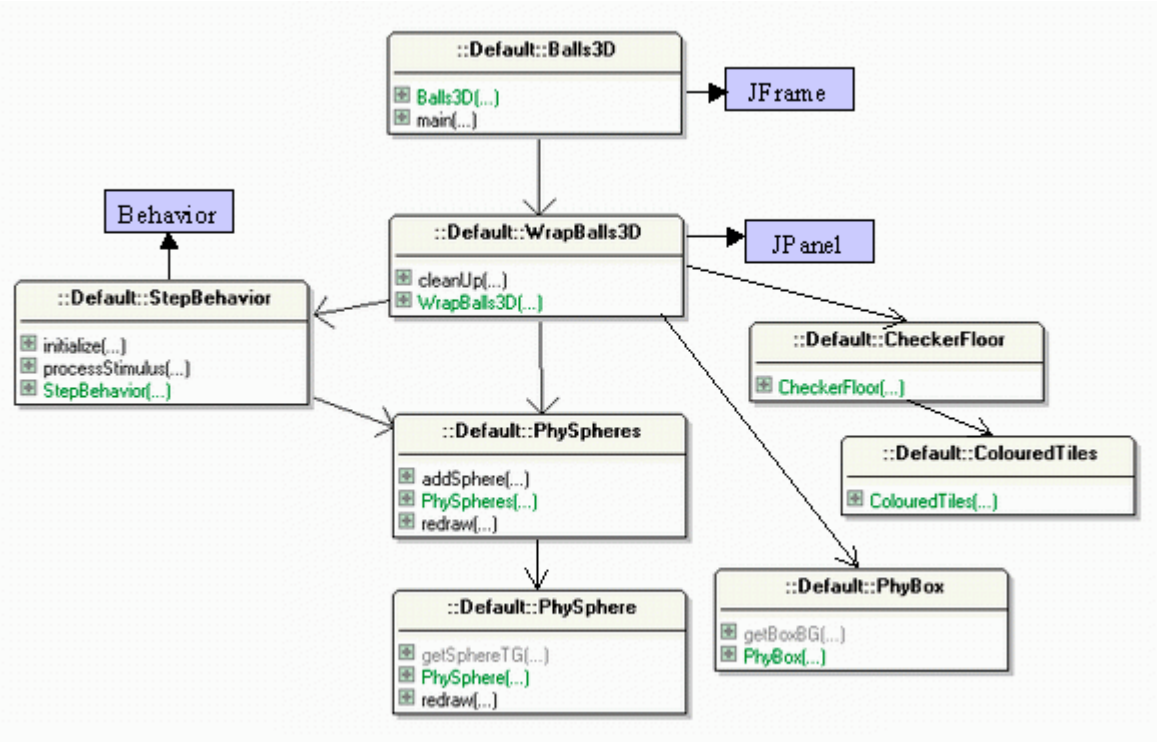

Figure 3. Class Diagrams for Balls3D.

Balls3D is the top-level JFrame, while WrapBalls3D does the work of creating the 3D scene. This includes setting up the graphical elements borrowed from Checkers3D, such as the floor, background, and lighting. The floor is created with the help of the CheckerFloor and ColouredTiles classes taken from Checkers3D.

Each sphere has a graphical and physics component, which are wrapped up in the PhySphere class. PhySpheres manages the PhySphere objects.

The Java 3D and Odejava aspects of the translucent box are created by PhyBox.

The simulation is executed by a Behavior subclass, StepBehavior, which is triggered every 30 milliseconds to perform a simulation step.

3.1. Creating the Scene

WrapBalls3D initializes the physics system, creates the box and spheres, and starts StepBehavior. These tasks are carried out from createSceneGraph()

```
// globals
private BranchGroup sceneBG;
private BoundingSphere bounds;   // for environment nodes


private void createSceneGraph()
{
  sceneBG = new BranchGroup();
  bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);
```

```
  lightScene();          // add the lights
  addBackground();       // add the sky
  sceneBG.addChild( new CheckerFloor().getBG() );  // add the floor

  initPhysWorld();    // start the physics engines
  addPhysObjects();   // add the box and spheres
  addStepper();       // step behaviour for simulation

  sceneBG.compile();   // fix the scene
} // end of createSceneGraph()
```

3.1.1. Initializing the Engines

initPhysWorld() starts Odejava's rigid body and collision detection engines.

```
// global physics objects
private World world;
private HashSpace collSpace;      // holds collision info
private JavaCollision collCalcs;  // calculates collisions


private void initPhysWorld()
{
  Odejava.getInstance();

  world = new World();
  // world.setGravity(0f, -0.2f, 0);
  world.setStepInteractions(10);
  world.setStepSize(0.05f);

  // create a collision space for the world's box and spheres
  collSpace = new HashSpace();

  collCalcs = new JavaCollision(world);  // for collision calcs
}
```

The gravity setting has been commented out so the spheres won't fall to the bottom of the box.

3.1.2. The Physics Objects

The details of the spheres and the box are hidden away inside their own objects, which addPhysObjects() create.

```
// globals
private static final int NUM_SPHERES = 10;
private PhySpheres spheres;     // manages the bouncing spheres


private void addPhysObjects()
{
  PhyBox box = new PhyBox(6.0f, 3.0f, 6.0f, collSpace);
  sceneBG.addChild( box.getBoxBG() );   // add the box to the scene
```

```
  // create the spheres
  spheres = new PhySpheres(sceneBG, world, collSpace);
  for(int i=0; i < NUM_SPHERES; i++)
    spheres.addSphere();
}
```

The first three arguments of PhyBox() are the box's width, height, and depth. The box's visual representation (the translucent space with yellow edges) is returned via PhyBox.getBoxBG(), and added to the scene graph.

NUM_SPHERES (10) spheres are created at random places inside the box, by repeatedly calling PhySpheres.addSphere(). The Java 3D spheres are attached to the scene graph inside PhySpheres, which is passed sceneBG.

3.1.3. Staring the Simulation

The simulation is driven by a StepBehavior object, created in addStepper().

```
// global
private StepBehavior stepBeh;


private void addStepper()
{
  stepBeh =
    new StepBehavior(30, spheres, world, collSpace, collCalcs);
       // it will be triggered every 30ms (== 33 frames/sec)

  stepBeh.setSchedulingBounds( bounds );
  sceneBG.addChild( stepBeh );
}
```

The behaviour is triggered every 30 ms, according to the first argument of its constructor. StepBehavior also takes a reference to the PhySpheres object, spheres, so it can request that the spheres' visual components be redrawn.

3.2. The Box

The PhyBox class manages the graphical and physics elements of a box whose dimensions are specified by width, height, and depth values supplied in the PhyBox() constructor. The box is centered at the origin, with its base resting on the XZ plane.

The graphical box is translucent, and its edges are highlighted with thick yellow lines (apart from those edges resting on the floor.) See Figures 1 and 2 for screenshots.

The physics box is defined by a geom plane on the XZ plane and five geom boxes for the walls and ceiling.

The dual nature of PhyBox is highlighted in its constructor:

```
public PhyBox(float width, float height, float depth,
                                          HashSpace collSpace)
{ makeBox(width, height, depth);        // makes the graphical parts
```

```
  makeBoxGeom(width, height, depth, collSpace);  // physics parts
}
```

makeBox() handles the Java 3D graphics, while makeBoxGeom() generates the Odejava geoms.

3.2.1. The Graphical Box

WrapBalls3D supplies the dimensions (6, 3, 6) for the box's width, height, and depth, and makeBox() creates a Java 3D Box object like the one in Figure 4.

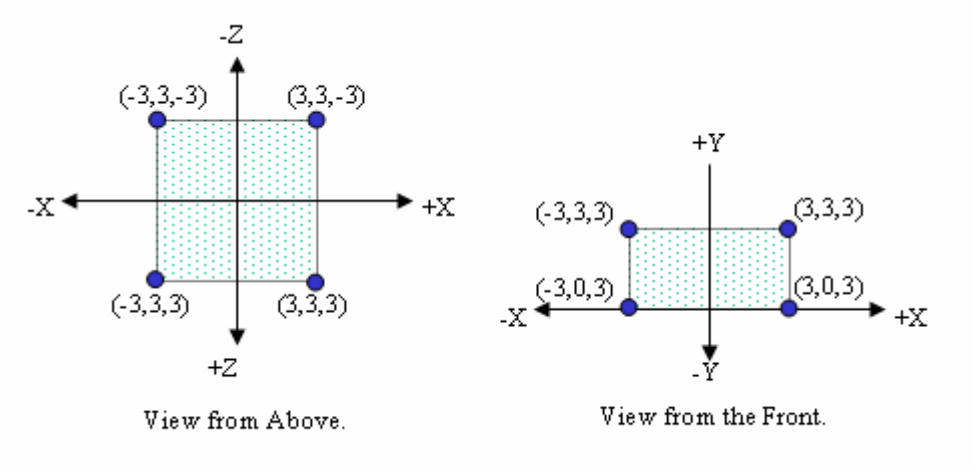

Figure 4. The Box's Dimensions.

By default, a Java 3D Box object is centered at the origin. makeBox() must move the box 1.5 units up the y-axis to make its base sit on the XZ plane. The box also needs to be made semi-transparent.

The makeBox() method:

```
// global
private BranchGroup boxBG;  // for holding the box's graphical parts


private void makeBox(float width, float height, float depth)
{
  float xDim = width/2.0f;
  float yDim = height/2.0f;
  float zDim = depth/2.0f;

  Appearance app = new Appearance();

  // switch off face culling
  PolygonAttributes pa = new PolygonAttributes();
  pa.setCullFace(PolygonAttributes.CULL_NONE);
  app.setPolygonAttributes(pa);

  // semi-transparent appearance
  TransparencyAttributes ta = new TransparencyAttributes();
  ta.setTransparencyMode( TransparencyAttributes.BLENDED );
  ta.setTransparency(0.7f);     // 1.0f is totally transparent
  app.setTransparencyAttributes(ta);
```

```
  // position the box: centered, sitting on the XZ plane
  Transform3D t3d = new Transform3D();
  t3d.set( new Vector3f(0, yDim+0.01f,0));
    /* the box is a bit above the floor, so it doesn't visual
       interact with the floor. */
  TransformGroup boxTG = new TransformGroup(t3d);
  boxTG.addChild(
     new com.sun.j3d.utils.geometry.Box(xDim, yDim, zDim, app));
           // set the box's dimensions and appearance

  Shape3D edgesShape =  makeBoxEdges(xDim, height, zDim); // edges

  // collect the box and edges together under a single BranchGroup
  boxBG = new BranchGroup();
  boxBG.addChild(boxTG);
  boxBG.addChild(edgesShape);
}  // end of makeBox()
```

Aside from making the Box's Appearance node use transparency, culling is also switched off. Then if the user moves the camera inside the box, the box's sides will still appear translucent.

The translation upwards is by height/2 (yDim), plus a small amount (0.01), so the box's base doesn't overlap with the checkboard floor.

makeBoxEdges() creates the yellow box edges by utilizing a Shape3D object. The Box, the box's TransformGroup, and edges shape are collected together under a single BranchGroup, resulting in the scene graph branch shown in Figure 5.

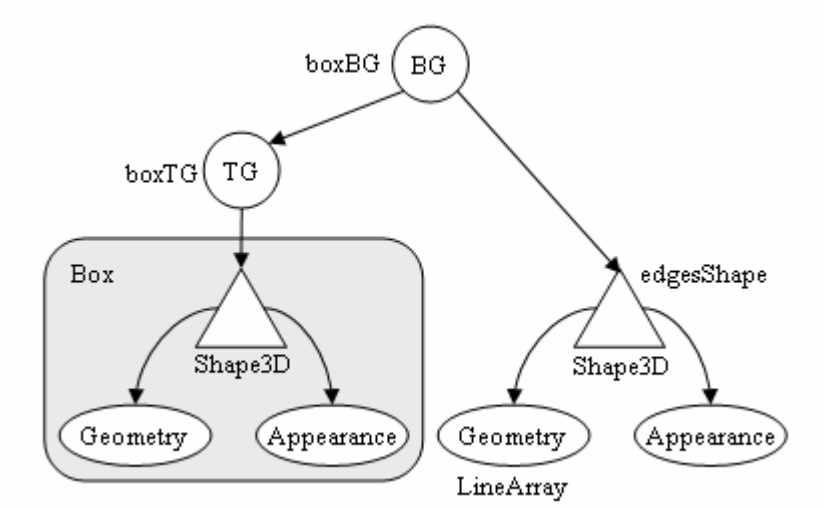

Figure 5. The Scene Graph Branch for the On-screen Box.

A close look at Figures 1 and 2 shows that the yellow highlighting only appears on the eight box edges above the floor. The simplest way of creating these edges is as lines in a Java 3D LineArray. This data structure can be used to initialize the Shape3D node.

Eight lines requires 16 points in the LineArray.

```
private Shape3D makeBoxEdges(float x, float y, float z)
```

```
{
  LineArray edges = new LineArray(16, LineArray.COORDINATES |
                                      LineArray.COLOR_3);

  Point3f pts[] = new Point3f[16];
  // front edges
  pts[0] = new Point3f(-x, 0, z);   // edge 1 (left)
  pts[1] = new Point3f(-x, y, z);

  pts[2] = new Point3f(-x, y, z);   // edge 2 (top)
  pts[3] = new Point3f( x, y, z);

  pts[4] = new Point3f( x, y, z);   // edge 3 (right)
  pts[5] = new Point3f( x, 0, z);

  // back edges
  pts[6] = new Point3f(-x, 0,-z);   // edge 4 (left)
  pts[7] = new Point3f(-x, y,-z);

  pts[8] = new Point3f(-x, y,-z);   // edge 5 (top)
  pts[9] = new Point3f( x, y,-z);

  pts[10] = new Point3f( x, y,-z);   // edge 6 (right)
  pts[11] = new Point3f( x, 0,-z);

  // top edges, running front to back
  pts[12] = new Point3f(-x, y, z);   // edge 7 (left)
  pts[13] = new Point3f(-x, y,-z);

  pts[14] = new Point3f( x, y, z);   // edge 8 (right)
  pts[15] = new Point3f( x, y,-z);

  edges.setCoordinates(0, pts);

  // set the edges colour to yellow
  for(int i = 0; i < 16; i++)
    edges.setColor(i, new Color3f(1, 1, 0));

  Shape3D edgesShape = new Shape3D(edges);

  // make the edges (lines) thicker
  Appearance app = new Appearance();
  LineAttributes la = new LineAttributes();
  la.setLineWidth(4);
  app.setLineAttributes(la);
  edgesShape.setAppearance(app);

  return edgesShape;
}  // end of makeBoxEdges()
```

The colour of the lines is set as part of the geometry, while their thickness is achieved with a LineAttributes object in the shape's appearance.

3.2.2. The Physics-based Box

The box geometry manufactured in makeBoxGeom() consists of a geom plane for the floor, four geom boxes for the walls, and a fifth one for the ceiling.

The geom plane coincides with the XZ plane, while the boxes surround the space occupied by the translucent Java 3D box. This notion is illustrated in Figure 6.

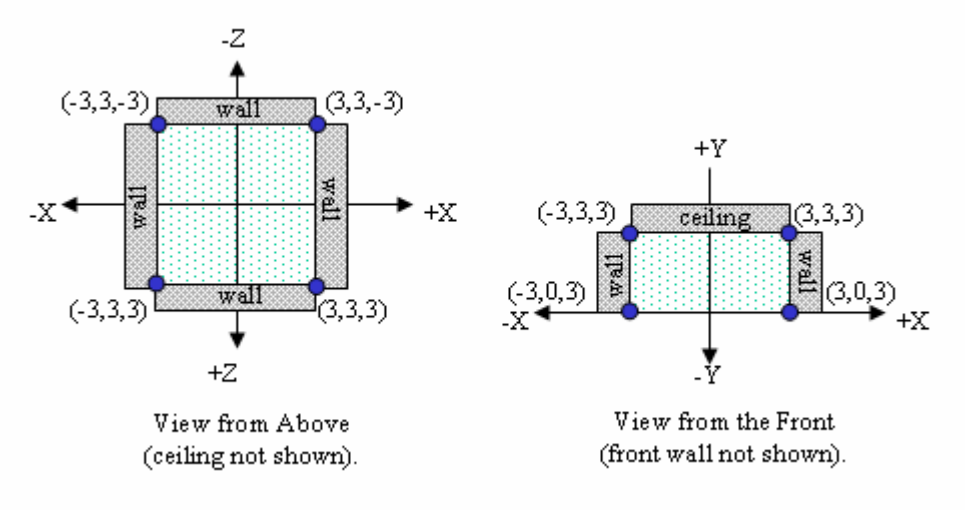

Figure 6. The Placing of the Geoms Around the Box.

The trickiest aspect is positioning the geom boxes correctly. By default, each box is centered at the origin, so needs to be repositioned. Each box is THICKNESS (1.0) units thick, an arbitrary value.

```
private static final float THICKNESS = 1.0f;

private void makeBoxGeom(float width, float height, float depth,
                                                  HashSpace collSpace)
{
  float xDim = width/2.0f;
  float yDim = height/2.0f;
  float zDim = depth/2.0f;
  float midWall = THICKNESS/2.0f;

  collSpace.add( new GeomPlane(0, 1.0f, 0, 0));   // floor

  // the four walls
  GeomBox rightWall = new GeomBox(THICKNESS, height, depth);
  rightWall.setPosition(xDim+midWall, yDim, 0);
  collSpace.add(rightWall);

  GeomBox leftWall = new GeomBox(THICKNESS, height, depth);
  leftWall.setPosition(-(xDim+midWall), yDim, 0);
  collSpace.add(leftWall);

  GeomBox frontWall = new GeomBox(width, height, THICKNESS);
  frontWall.setPosition(0, yDim, zDim+midWall);
  collSpace.add(frontWall);

  GeomBox backWall = new GeomBox(width, height, THICKNESS);
  backWall.setPosition(0, yDim, -(zDim+midWall));
  collSpace.add(backWall);

  // the ceiling
  GeomBox ceiling = new GeomBox(width, THICKNESS, depth);
  ceiling.setPosition(0, height+midWall, 0);
```

```
  collSpace.add(ceiling);
}  // end of makeBoxGeom()
```

Since the floor, walls, and ceiling will be involved in collision detection, they're added to the collision space, collSpace.

3.3. Managing the Spheres

PhySpheres main task is to create a PhySphere using a randomly generated radius, position, and linear velocity. Each sphere is also assigned a texture, chosen at random.

The textures are loaded by PhySpheres at initialization time, so each PhySphere object can use one immediately, without the overhead of loading it.

PhySpheres has a redraw() method for all its spheres, which the StepBehavior class calls.

3.3.1. Initializing PhySpheres

The constructor creates an empty array list for holding future PhySphere objects, and uses loadTextures() to load the "earth", "moon", and "mars" textures, storing them as Texture2D objects in an array.

```
// globals
private BranchGroup sceneBG; // the scene graph
private World world;         // physics elements
private HashSpace collSpace;

private ArrayList<PhySphere> spheres;
private Random rand;


public PhySpheres(BranchGroup sg, World w, HashSpace cs)
{
  sceneBG = sg;
  world = w;
  collSpace = cs;

  spheres = new ArrayList<PhySphere>();
  rand = new Random();

  loadTextures();
}  // end of PhySpheres()
```

3.3.2. Adding a Sphere

addSphere() (called from WrapBalls3D) utilizes random numbers to set the texture, radius, position, and velocity for a new sphere.

```
// globals
private Texture2D[] textures;   // holds the loaded textures
```

```
public void addSphere()
{
  Texture2D planTex = textures[ rand.nextInt(textures.length) ];
  float radius = rand.nextFloat()/4.0f + 0.2f; // between 0.2 & 0.45

  PhySphere s = new PhySphere(world, collSpace, "planet "+counter,
                      planTex, radius, randomPos(), randomVel());

  sceneBG.addChild( s.getSphereTG() );
  spheres.add(s);  // add to ArrayList
  counter++;
}  // end of addSphere()
```

PhySphere creates a Java 3D sphere, which is accessed via PhySphere.getSphereTG(), and linked to the scene graph.

randomPos() and randomVel() generate random position and velocity vectors within prescribed ranges.

```
private Vector3f randomPos()
{ Vector3f pos = new Vector3f();
  pos.x = rand.nextFloat()*5.0f - 2.5f;   // -2.5 to 2.5
  pos.y = rand.nextFloat()*2.0f + 0.5f;   // 0.5 to 2.5
  pos.z = rand.nextFloat()*5.0f - 2.5f;   // -2.5 to 2.5
  return pos;
}  // end of randomPos()

private Vector3f randomVel()
{ Vector3f vel = new Vector3f();
  vel.x = rand.nextFloat()*6.0f - 3.0f;   // -3.0 to 3.0
  vel.y = rand.nextFloat()*6.0f - 3.0f;
  vel.z = rand.nextFloat()*6.0f - 3.0f;
  return vel;
}  // end of randomVel()
```

The position values are hardwired to be somewhere within the translucent space, while the velocity numbers produce a reasonable speed in most cases.

3.3.3. Helping StepBehavior

StepBehavior periodically needs to redraw the visual components of the spheres. It does this with PhySpheres' redraw() method.

```
public void redraw()
{ for(PhySphere ps: spheres)
    ps.redraw();
}
```

3.4. A Sphere

A sphere has two parts: a Java 3D visualization and an Odejava body. This is highlighted by the method calls in the constructor, which build them.

```
// globals
private Transform3D t3d;    // used for accessing a TG's transform
private DecimalFormat df;   // for printing data


public PhySphere(World world, HashSpace collSpace, String name,
                      Texture2D tex, float radius,
                      Vector3f posVec, Vector3f velVec)
{ t3d = new Transform3D();
  df = new DecimalFormat("0.##");  // 2 dp

  makeSphere3D(tex, radius, posVec);     // makes the graphical part
  makeSphereBody(world, collSpace, name, radius, posVec, velVec);
                                         // physics part
}
```

3.4.1. The Graphical Sphere

The graphical sphere is textured and lit, and hangs below two TransformGroups (TGs) which control its position and orientation. The configuration is shown in Figure 7.

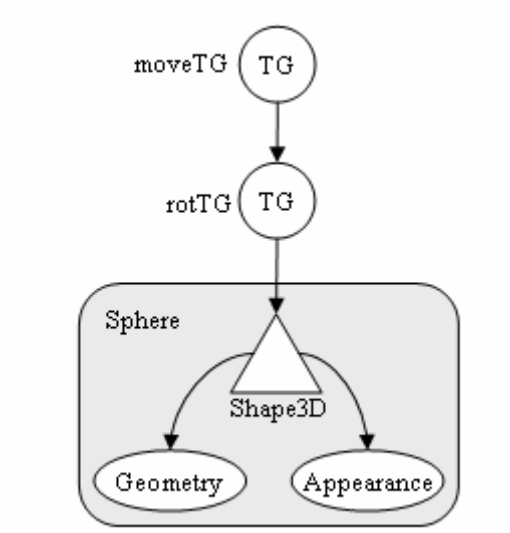

Figure 7. The Scene Graph Branch for an On-screen Sphere.

The moveTG TransformGroup is used to translate the sphere, and rotTG rotates it. I could have used a single TransformGroup instead, but separating the two operations makes them easier to understand.

makeSphere3D() also constructs the Java 3D Sphere, and sets its appearance.

```
// globals
// sphere colours
private static final Color3f BLACK = new Color3f(0.0f, 0.0f, 0.0f);
```

```
private static final Color3f GRAY = new Color3f(0.6f, 0.6f, 0.6f);
private static final Color3f WHITE = new Color3f(0.9f, 0.9f, 0.9f);

// TGs which the sphere hangs off:
private TransformGroup moveTG, rotTG;


private void makeSphere3D(Texture2D tex, float radius,
                                                  Vector3f posVec)
{
  Appearance app = new Appearance();

  // combine texture with material and lighting of underlying surface
  TextureAttributes ta = new TextureAttributes();
  ta.setTextureMode( TextureAttributes.MODULATE );
  app.setTextureAttributes( ta );

  // assign gray material with lighting
  Material mat= new Material(GRAY, BLACK, GRAY, WHITE, 25.0f);
     // sets ambient, emissive, diffuse, specular, shininess
  mat.setLightingEnable(true);
  app.setMaterial(mat);

  // apply texture to shape
  if (tex != null)
    app.setTexture(tex);

  // make the sphere with normals for lighting, and texture support
  Sphere sphere = new Sphere(radius,
                         Sphere.GENERATE_NORMALS |
                         Sphere.GENERATE_TEXTURE_COORDS,
                         15, app);   // default divs == 15

  // create a transform group for rotating the sphere
  rotTG = new TransformGroup();
  rotTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  rotTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  rotTG.addChild(sphere);

  // create a transform group for moving the sphere
  t3d.set(posVec);
  moveTG = new TransformGroup(t3d);
  moveTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  moveTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  moveTG.addChild(rotTG);
}  // end of makeSphere3D()
```

3.4.2. The Physics-based Sphere

The Odejava sphere combines GeomSphere and Body objects. The GeomSphere is for detecting collisions with other spheres and the box. The Body object stores the mass, position, and the linear and angular velocities.

The position and linear velocity come from PhySpheres, the mass is derived from the sphere's radius, and the angular velocity is hardwired to produce a clockwise spin around the y-axis.

```
// globals
```

```
// radius --> mass conversion
private static final float MASS_FACTOR = 5.0f;

private Body sphereBody;


private void makeSphereBody(World world, HashSpace collSpace,
                            String name, float radius,
                            Vector3f posVec, Vector3f velVec)
{
  sphereBody = new Body(name, world, new GeomSphere(radius));

  sphereBody.adjustMass(MASS_FACTOR*radius);
  sphereBody.setPosition(posVec);  // same as graphical sphere
  sphereBody.setLinearVel(velVec);
  sphereBody.setAngularVel(0, 2.0f, 0);  // clockwise around y-axis

  collSpace.addBodyGeoms( sphereBody );  // add to collision space
} // end of makeSphereBody()
```

Care must be taken that the Odejava and Java 3D spheres start at the same position. Also, the Odejava sphere must be added to the collision space so it can take part in collision detection.

3.4.3. Redrawing the Sphere

redraw() is where changes in the Odejava sphere affect the Java 3D sphere.

StepBehavior calls PhySpheres' redraw() at the end of its simulation step, after the physics-based spheres have been moved.

PhySpheres calls redraw() in each PhySphere to update the position and orientation of the Java 3D sphere with the position and orientation of the corresponding Odejava sphere.

```
public void redraw()
{
  // get position and orientation from the physics sphere
  Vector3f posVec = sphereBody.getPosition();
  Quat4f quat = sphereBody.getQuaternion();

  // update the TGs in the graphical sphere
  t3d.set(posVec);
  moveTG.setTransform(t3d);   // translate the sphere

  t3d.set(quat);
  rotTG.setTransform(t3d);    // rotate the sphere
}  // end of redraw()
```

The mapping from Odejava to Java 3D is quite straightforward. The position and orientation details of the Odejava sphere are extracted as a Vector3f vector and a Quat4f quaternion, and applied to the moveTG and rotTG TransformGroups of the Java 3D sphere.

An important reason for the simplicity is that Odejava and Java 3D use the same vector and matrix classes, so data can be readily shared.

3.5. I'm Steppin Out…

StepBehavior advances the simulation a step at a time, once every timeDelay ms.

The simulation step performs collision detection, creates contact joints, and redraws the Java 3D spheres.

3.5.1. Initializing the Behavior

The StepBehavior constructor stores references to the PhySpheres object, the rigid body and collision engines, and the Java 3D WakeupCondition object, which will periodically trigger the behaviour.

```
// globals
private WakeupCondition timeOut;
private PhySpheres spheres;

private World world;
private HashSpace collSpace;      // holds collision info
private JavaCollision collCalcs;  // calculates collisions
private Contact contactInfo;      // for accessing contact details


public StepBehavior(int timeDelay, PhySpheres ps,
                 World w, HashSpace cs, JavaCollision cc)
{
  timeOut = new WakeupOnElapsedTime(timeDelay);
  spheres = ps;
  world = w;
  collSpace = cs;
  collCalcs = cc;
  contactInfo = new Contact( collCalcs.getContactIntBuffer(),
                             collCalcs.getContactFloatBuffer());
}
```

Java 3D calls the behaviour's initialize() method to set it waiting until the specified time has elapsed.

```
public void initialize()
{ wakeupOn( timeOut ); }
```

3.5.2. Responding to a Wake-up Call

The behaviour is woken by Java 3D calling its processStimulus() method, where the step execution code is located.

```
public void processStimulus( Enumeration criteria )
{
  // step through the simulation
  collCalcs.collide(collSpace);   // find collisions
  examineContacts();              // examine contact points
  collCalcs.applyContacts();
                    // add contacts to contactInfo jointGroup
```

```
  world.stepFast();              // advance the simulation

  spheres.redraw();   // redraw the graphical spheres

  wakeupOn( timeOut );  // wait for the next wake-up
} // end of processStimulus()
```

The contact points are found, converted to joints, and the physics simulation is advanced, thereby changing the position and orientation of the physics spheres. redraw() is called in PhySpheres to use those changes to modify the graphical sphere's position and orientation.

examineContacts() loops through the contact points and converts those involving spheres into bouncing contact joints.

```
private void examineContacts()
{
  for (int i = 0; i < collCalcs.getContactCount(); i++) {
    contactInfo.setIndex(i);     // look at the ith contact point

    // if contact involves a sphere, then make the contact bounce
      if ((contactInfo.getGeom1() instanceof GeomSphere) ||
          (contactInfo.getGeom2() instanceof GeomSphere)) {
      contactInfo.setMode(Ode.dContactBounce);
      contactInfo.setBounce(1.0f);     // 1 is max bounciness
      contactInfo.setBounceVel(0.1f);  // min velocity for a bounce
      contactInfo.setMu(0);            // 0 is friction-less
    }
  }
} // end of examineContacts()
```

The contact point details in contactInfo include references to the geoms involved in the collision. StepBehavior checks if either of them is a GeomSphere before configuring the joint to be very bouncy, and frictionless.

Another, slightly more general, way of identifying a geom is to examine its name, using Geom.getName(). In Balls3D, all the sphere names start with "Planet", followed by a number, so are easy to recognize. makeSphereBody() in PhySphere assigns the name to the sphere's Body object, which gets taken up by its associated geom.

4. A Note of Application Development

A useful way of structuring the development of a Java 3D/Odejava application is to split it into two stages. The first stage involves the creation of the visual elements only – the Java 3D scene *without* the physics.

In the case of Balls3D, I implemented the PhyBox and PhySphere classes without their Odejava methods (makeBoxGeom() and makeSphereBody()). I also left out the StepBehavior and PhySpheres classes.

This approach allows the graphical features to be tested and debugged without the extra complexity of physics simulation.

The second stage adds in the physics elements. Since the Odejava bodies and geoms are closely linked to their Java 3D counterparts, it's quite easy to decide which classes need augmenting.

The basic StepBehavior class is always the same: a behaviour using an elapsed time WakeupCondition. The code in processStimulus() is fairly standard, although the details of examineContacts() will vary depending on what geoms are of interest.

Another advantage of developing the graphical side of the application first, is that it can show how things look when various parameters, such as gravity, friction, and the amount of bounce, are tweaked.