Chapter N4. The Colliding Grabbers

This chapter looks at how to use Java 3D's built-in Box, Cylinder, and Sphere shapes to create *articulated models*. The shapes are linked together by TransformGroups acting as joints, which can be rotated and moved. This functionality is essential if you want to build humanoid figures with operational limbs, or machinery with working parts.

The Arms3D example consist of two arms (grabbers). Each grabber can rotate at its 'elbow' around the x-, y-, and z- axes. A grabber's forearm ends with two 'fingers' which can open and close. The grabbers can slide in unison over the floor forwards, backwards, left or right, but the two arms always stay the same distance apart.

Figures 1 and 2 show the grabbers in different poses.

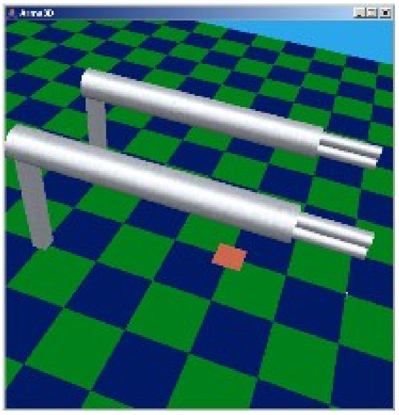

Figure 1. The Grabbers' Initial Pose.

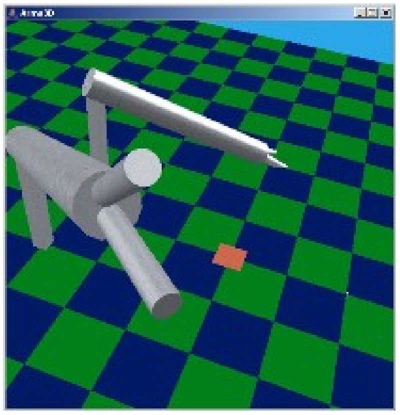

Figure 2. After Moving the Grabbers.

Figure 3 highlights a grabber's elbow and finger joints.

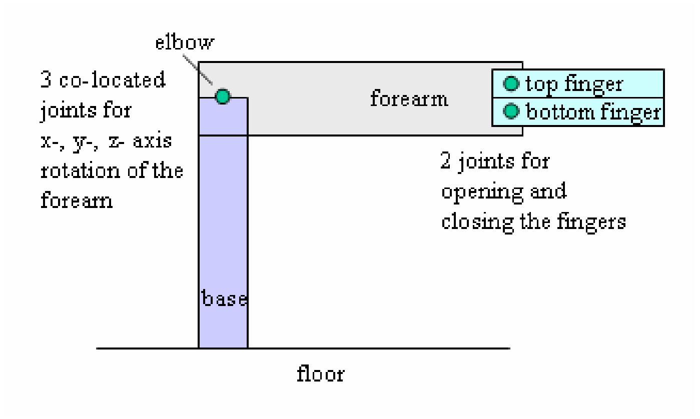
elbow
3 co-located joints for x-, y-, z- axis rotation of the forearm
forearm
top finger
bottom finger
2 joints for opening and closing the fingers
base
floor

Figure 3. A Grabber's Joints.

Articulated models also need *collision detection* abilities, to prevent them from moving through objects in the scene. Arms3D demonstrates a simple form of collision detection and recovery, which stops the grabbers from rotating through each other.

The detection code also reports when a grabber's fingers touch or pass through the floor (but doesn't stop the fingers).

Rotation and translation commands entered at the keyboard are caught by a Java 3D Behavior. Another Behavior subclass is used to monitor the grabbers' joints, and trigger collision checking.

The grabbers' appearance is a combination of a shiny metallic Java 3D Material object and a texture (a GIF image) to add extra detail.

The x- and z- axes of the checkboard floor are labeled with 2D text, which is useful when positioning the grabbers. I'll be reusing the floor code in several later examples.

Arms3D employs the same lighting scheme and background colour as in Life3D. Java 3D's OrbitBehavior class is again utilized to move the camera around the scene.

1. Class Diagrams for Arms3D

Figure 4 shows the class diagrams for the Arms3D application. Only the public methods are shown for each class.

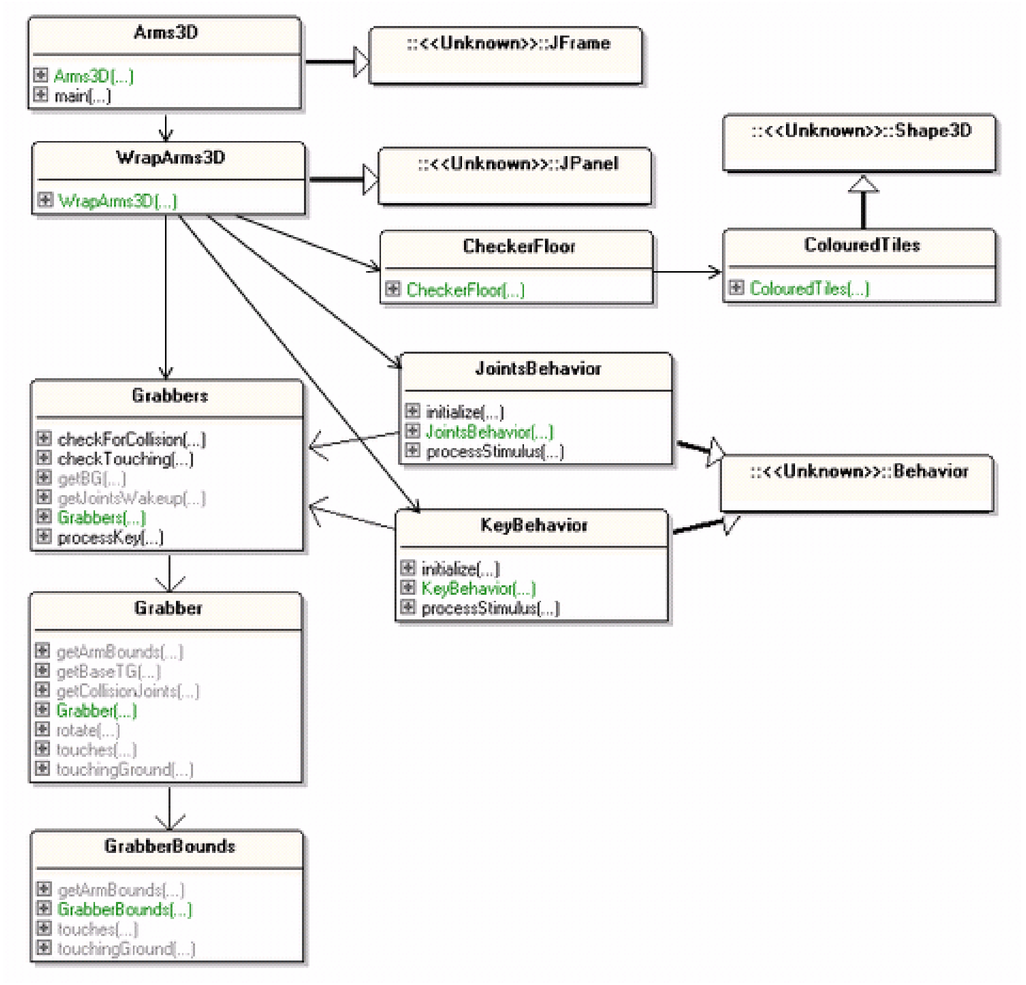

Figure 4. Class Diagrams for Arms3D.

Arms3D creates the application's JFrame, and calls WrapArms3D to render the 3D scene in a JPanel. The checkboard floor is built by CheckerFloor, with the help of ColouredTiles to make (and group together) floor tiles of the same colour.

The Grabbers class manages the grabbers, and carries out tasks common to both, such as moving them about the floor. It creates two Grabber instances to handle grabber-specific work, such as rotating a particular grabber's elbow or fingers. Collision detection is supported by a GrabberBounds object assigned to each grabber.

The KeyBehavior object passes the user's key presses to the Grabbers class for processing, while JointsBehavior monitors the joints of both grabbers, and triggers collision detection and recovery code in GrabberBounds.

2. Creating the Application Window.

Arms3D creates a non-resizable JFrame to hold the JPanel where the 3D scene is rendered.

```
public Arms3D()
{
  super("Arms3D");
  Container c = getContentPane();
  c.setLayout( new BorderLayout() );
  WrapArms3D w3d = new WrapArms3D();  // panel holding 3D canvas
  c.add(w3d, BorderLayout.CENTER);

  setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  pack();
  setResizable(false);    // fixed size display
  setVisible(true);
} // end of Arms3D()
```

If the application needed additional GUI elements, such as buttons or menus, they could be added to Arms3D.

3. Drawing the 3D Scene

WrapArms3D wraps Java 3D's heavyweight Canvas3D component inside a lightweight Swing JPanel, thereby protecting the rest of the application's GUI.

```
// globals
private static final int PWIDTH = 512;   // size of panel
private static final int PHEIGHT = 512;

private SimpleUniverse su;


public WrapArms3D()
{
  setLayout( new BorderLayout() );
  setOpaque( false );
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  GraphicsConfiguration config =
                SimpleUniverse.getPreferredConfiguration();
  Canvas3D canvas3D = new Canvas3D(config);
  add("Center", canvas3D);
  canvas3D.setFocusable(true);    // give focus to the canvas
  canvas3D.requestFocus();

  su = new SimpleUniverse(canvas3D);

  createSceneGraph();
  initUserPosition();        // set user's viewpoint
  orbitControls(canvas3D);   // controls for moving the viewpoint

  su.addBranchGraph( sceneBG );
} // end of WrapArms3D()
```

The WrapArms3D constructor sets up the user's viewpoint and the Java 3D OrbitBehavior in a similar way to WrapLife3D. Unlike WrapLife3D, there's no KeyListener for capturing key presses. Key strokes are still sent to the Canvas3D instance because of the setFocusable() call, but they're dealt with by the KeyBehavior instance.

3.1. Creating the Scene Graph

createSceneGraph() sets up a bounding sphere for the environment nodes, initializes the top-level BranchGroup, sceneBG, and compiles the scene, all in a similar manner to Life3D. However, the scene graph created below sceneBG is quite different.

```
// 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
  addGrabbers();        // add grabbers and behaviours

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

Figure 5 shows the scene graph constructed by the calls to lightScene(), addBackground(), the CheckerFloor() instance, and addGrabbers().

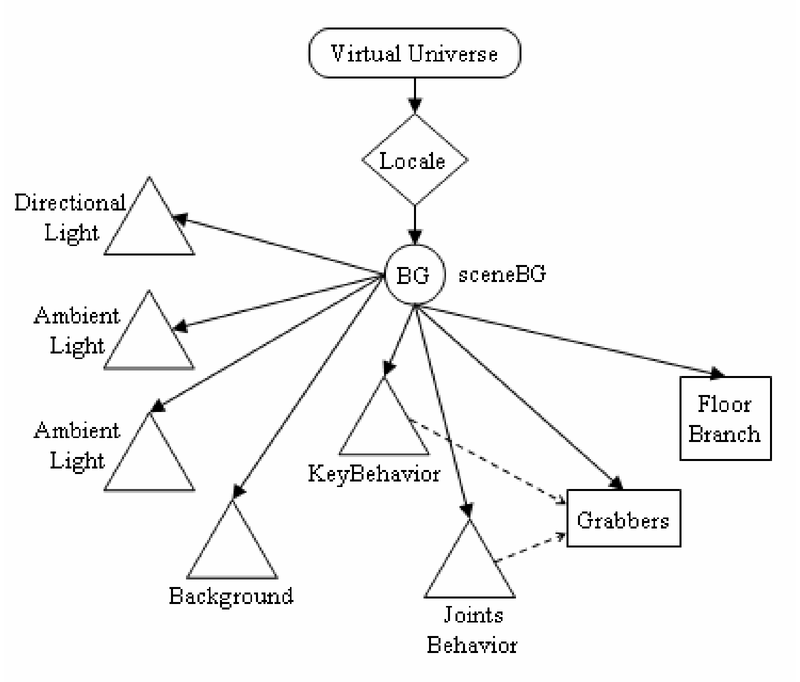

Figure 5. The Scene Graph for Arms3D.

lightScene() creates an ambient light and two directional sources, and addBackground() adds a light blue sky.

The "Floor Branch" box hides the details of floor creation, which I'll explain when I get to the CheckerFloor class.

The two behaviours and the "Grabbers" box are created in addGrabbers():

```
private void addGrabbers()
{
  Grabbers grabbers = new Grabbers( new Vector3f(0,0,3), 1.0f);
        /* Supply the position of the grabbers center point, and
           each grabber's x-axis offset from that point. */
  sceneBG.addChild( grabbers.getBG() );

  // the keyboard controls
  KeyBehavior kb = new KeyBehavior(grabbers);
  kb.setSchedulingBounds(bounds);
  sceneBG.addChild(kb);

  // the joints collision detection behaviour
  JointsBehavior jb = new JointsBehavior(grabbers);
  jb.setSchedulingBounds(bounds);
  sceneBG.addChild(jb);
}  // end of addGrabbers()
```

The "Grabbers" box in Figure 5 hides the complicated scene branch constructed for the two grabbers.

The Vector3f argument of the Grabbers constructor can be any (x,y,z) position, so it's entirely possible to place the grabbers in mid-air, or below the floor's surface. But once they're in place, they can only move parallel to the XZ plane, not up or down. Also, the grabbers always stay facing along the -z axis; I'll explain how to rotate them when I revisit Arms3D in chapter N10??.

The second constructor argument is the x-axis offset of each grabber from the midpoint between them. In this example, the grabbers will be 2 units apart.

4. Processing the Keys

KeyBehavior waits for a key to be pressed, then sends it's details to Grabbers. The WakeupCondition is an AWT key press, specified in the constructor and registered in initialize():

```
// globals
private WakeupOnAWTEvent keyPress;
private Grabbers grabbers;


public KeyBehavior(Grabbers gs)
{ grabbers = gs;
  keyPress = new WakeupOnAWTEvent( KeyEvent.KEY_PRESSED );
}

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

processStimulus() checks that the waking criteria is an AWT event and responds to a key press:

```
public void processStimulus(Enumeration criteria)
{
  WakeupCriterion wakeup;
  AWTEvent[] event;

  while( criteria.hasMoreElements() ) {
    wakeup = (WakeupCriterion) criteria.nextElement();
    if( wakeup instanceof WakeupOnAWTEvent ) {
      event = ((WakeupOnAWTEvent)wakeup).getAWTEvent();
      for( int i = 0; i < event.length; i++ ) {
        if( event[i].getID() == KeyEvent.KEY_PRESSED )
          processKeyEvent((KeyEvent)event[i]);
      }
    }
  }
  wakeupOn( keyPress );
} // end of processStimulus()
```

All the testing and iteration through the event[] array leads to a call to processKeyEvent():

```
private void processKeyEvent(KeyEvent eventKey)
{
  grabbers.processKey(eventKey.getKeyCode(),
                      eventKey.isShiftDown(),
                      eventKey.isAltDown() );
}
```

The key code, and whether the 'alt' and 'shift' keys are pressed, are delivered to processKey() in Grabbers.

5. Monitoring Grabber Joints

JointsBehavior monitors TransformGroups which implement grabber joints. Not every joint is watched, only those that may cause collisions when they change.

Grabbers creates the wakeup condition, a Java 3D WakeupOr object, which contains a list of WakeupOnTransformChange criteria. JointsBehavior begins by retrieving it from Grabbers, then registers it.

```
// globals
private WakeupOr jointsWakeup;
private Grabbers grabbers;


public JointsBehavior(Grabbers gs)
{ grabbers = gs;
  jointsWakeup = grabbers.getJointsWakeup();
}

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

processStimulus() is called when any of the TransformGroups in the wakeup condition change (i.e. a joint is rotated). The rotation *may* cause a collision between the grabbers, or between a grabber and the floor. These possibilities are checked by calls to checkForCollision() and checkTouching() in Grabbers.

```
public void processStimulus(Enumeration criteria)
{
  WakeupCriterion wakeup;
  TransformGroup tg;

  while( criteria.hasMoreElements() ) {
    wakeup = (WakeupCriterion) criteria.nextElement();
    if( wakeup instanceof WakeupOnTransformChange ) {
      // reportTG(wakeup);
      grabbers.checkForCollision();
      grabbers.checkTouching();
    }
  }
  wakeupOn( jointsWakeup );
} // end of processStimulus()
```

JointsBehavior doesn't need to know which TransformGroup woke it up, but it isn't hard to find out. reportTG() prints the name of the joint.

```
private void reportTG(WakeupCriterion wakeup)
{
  TransformGroup jointTG =
          ((WakeupOnTransformChange)wakeup).getTransformGroup();
  String name = (String) jointTG.getUserData();
  if (name == null)
    System.out.println("Joint has no name");
  else
    System.out.println(name + " moved");
} // end of reportTG()
```

A reference to the TransformGroup is obtained by calling WakeupOnTransformChange.getTransformGroup(). Only printing the object's reference isn't very informative, so in Grabber I label each TransformGroup with SceneGraphObject.setUserData(). reportTG() reads a label with getUserData().

There are specialized versions of these set/get methods for Strings only (setName() and getName()), introduced in Java 3D 1.4.

In a more complex application, the TransformGroup labels could be passed to checkForCollision() and checkTouching(), to allow them to focus collision detection upon a particular joint.

6. Managing the Grabbers

The Grabbers class does four main jobs:

- It creates two Grabber objects, and ties their scene branches to the rest of the scene graph.
- It processes keyboard commands. Translation commands are handled by Grabbers itself, while joint rotation requests are passed to the relevant Grabber instance.
- It asks each Grabber instance for a collection of TransformGroup joints which may potential cause collisions when they're rotated. It combines the two collections into a single Wakeup condition and makes it available to JointsBehavior.
- Grabbers method are called by JointsBehavior to check if a collision has occurred, and to fix it.

6.1. Adding the Grabbers to the Scene Graph

Figure 6 supplies the contents of the "Grabbers" box in Figure 5.

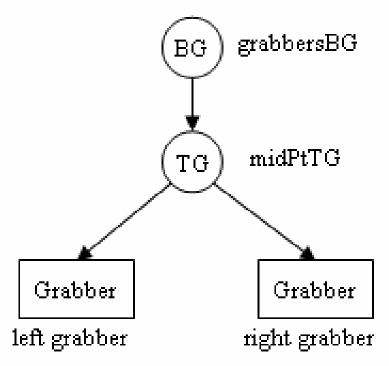

Figure 6. The Scene Branch for Grabbers.

midPtTG in Figure 6 is the position of the midpoint between the grabbers.

I'll explain the scene graph branches made by each Grabber instance when I consider the Grabber class

The code that generates Figure 6 is located in the Grabbers constructor:

```
// globals
private final static String TEX_FNM = "images/steel.jpg";

// reusable objects for calculations
private Transform3D t3d, toMove;
private Vector3f moveVec;

// scene graph elements
private BranchGroup grabbersBG;
private TransformGroup midPtTG;
private Grabber leftGrabber, rightGrabber;


public Grabbers(Vector3f posnVec, float grabOffset)
{
  t3d = new Transform3D();
  toMove = new Transform3D();
  moveVec = new Vector3f();

  grabbersBG = new BranchGroup();

  // position the grabbers midpoint at posnVec
  t3d.set(posnVec);
  midPtTG = new TransformGroup(t3d);
  midPtTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  midPtTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    // let midPtTG be read and changed at runtime
  grabbersBG.addChild(midPtTG);

  Texture2D tex = loadTexture(TEX_FNM);  // used by both grabbers

  // add the grabber left of the midpoint
  leftGrabber = new Grabber("left", tex, -grabOffset);
  midPtTG.addChild( leftGrabber.getBaseTG() );

  // add the grabber right of the midpoint
  rightGrabber = new Grabber("right", tex, grabOffset);
  midPtTG.addChild( rightGrabber.getBaseTG() );

  buildWakeUps();
}  // end of Grabbers()
```

The t3d, toMove, and moveVec objects are used repeatedly in Grabbers to hold translation information. Reusing objects isn't difficult, and avoids the garbage collection overheads associated with creating many short-lived temporary objects.

Grabbers moves the grabbers by applying translations to the midPtTG TransformGroup, which is the midpoint between the grabbers, and the parent node for both grabber scene branches. In order for midPtTG to be read and changed at runtime, its ALLOW_TRANSFORM_READ and ALLOW_TRANSFORM_WRITE capability bits must be set.

Using Textures

A texture is made in two stages. First, a TextureLoader object is created for the file holding the texture image, then the texture is extracted from that object.

```
private Texture2D loadTexture(String fn)
{
  TextureLoader texLoader = new TextureLoader(fn, null);
  Texture2D texture = (Texture2D) texLoader.getTexture();
  if (texture == null)
    System.out.println("Cannot load texture from " + fn);
  else {
    System.out.println("Loaded texture from " + fn);
    texture.setEnable(true);
  }
  return texture;
}  // end of loadTexture()
```

TextureLoader can handle JPEGs and GIFs (which are useful if transparency is required), and it can be employed in conjunction with JAI (Java Advanced Imaging) to load other formats, such as BMP, PNG, and TIFF files. The loader can include various flags, such as one for creating textures at various levels of resolution for rendering onto small areas.

The if-test in loadTexture() checks if the texture was successfully created. A common reason for the texture being null is the source image's dimensions being invalid. The image must be square, with its dimensions a power of 2. Keeping this in mind, I made the "steel.gif" image's size 128 by 128 pixels. Figure 7 shows the contents of the file.

Figure 7. The steel.gif File.

The call to Texture2D.setEnable() switches on texture mapping, which allows the texture to be wrapped around a shape. The wrapping will be carried out in the Grabber class when 'limbs' are created from Java 3D's Box and Cylinder classes.

6.2. Processing Keyboard Commands

An unusual aspect of this application is the large number of different input commands. A grabber can rotate its elbow around the x-, y-, and z- axes, and open and close its fingers, which requires eight commands since the joints and fingers can rotate in both the positive and negative direction; this doubles to 16 when both grabbers are considered. Adding grabbers translation forwards, backwards, left, and right, takes the total to 20.

I use the letter keys 'x', 'y', and 'z', for positive rotations of the x-, y-, and z- axis joints for the left grabber, and 'f' to open its fingers. 'f' is enough for both finger joints since they open and close together.

If 'alt' is pressed as well, the meaning switches to a negative rotation. If 'shift' is included, the commands refer to the right grabber.

The keyboard's arrow keys are utilized for forwards, backwards, left, and right movements of both grabbers.

Admittedly, all these key combinations are somewhat confusing, and is one reason why I revisit Arms3D in chapter N10?? to replace the keyboard with a gamepad.

As explained earlier, KeyBehavior obtains the letter (or arrow) key, sets booleans if the 'alt' and shift' keys are pressed, and calls Grabbers' processKey():

```
// global
private boolean moveNeedsChecking = false;


public void processKey(int keyCode, boolean isShift, boolean isAlt)
{ if (!moveMidPt(keyCode) && !moveNeedsChecking)
    rotateJoint(keyCode, isShift, isAlt);
}
```

moveMidPt() tries to translate the grabbers. If the supplied keycode isn't an arrow key then moveMidPt() returns false, and rotateJoint() is called. It attempts to rotate a joint, and looks at the keycode, and the 'alt' and 'shift' booleans, to determine the grabber, joint, and direction.

The moveNeedsChecking boolean is set to true when the previous command rotated a joint which *may* have caused a collision. Until the situation has been checked, moveNeedsChecking will be true, thereby preventing any new joint rotations being carried out by rotateJoint().

6.3. Translating the Grabbers

moveMidPt() contains a series of if-tests to determine which arrow key was pressed and to respond accordingly.

```
// globals: the translation key codes
private final static int forwardKey = KeyEvent.VK_DOWN;
private final static int backKey = KeyEvent.VK_UP;
private final static int leftKey = KeyEvent.VK_LEFT;
private final static int rightKey = KeyEvent.VK_RIGHT;

// step size for moving the base
private final static float STEP = 0.1f;


private boolean moveMidPt(int keyCode)
// move the grabbers midpoint if the keyCode is the right type
{
  if(keyCode == forwardKey) {
    moveBy(0, STEP);
    return true;
  }
  if(keyCode == backKey) {
    moveBy(0, -STEP);
    return true;
  }
  if(keyCode == leftKey) {
    moveBy(-STEP, 0);
    return true;
  }
  if(keyCode == rightKey) {
    moveBy(STEP, 0);
    return true;
  }
  // not a move keyCode
  return false;
}  // end of moveMidPt()
```

moveBy() moves the grabbers midpoint position by an (x,z) step.

```
// reusable global objects for calculations
private Transform3D t3d, toMove;
private Vector3f moveVec;


private void moveBy(float x, float z)
// add (x,z) to the grabbers midpoint position
{
  moveVec.set(x,0,z);
  midPtTG.getTransform(t3d);
  toMove.setTranslation(moveVec);
  t3d.mul(toMove);  // t3d = t3d * toMove
  midPtTG.setTransform(t3d);
}
```

midPtTG's current transformation (its translation from the origin) is copied into t3d. The x- and z- step values are stored in a vector (with the y- value == 0), and from there are copied into toMove.

The effect of multiply toMove to t3d is to 'add' the (x,z) step to t3d's translation. The changed t3d is put back into midPtTG, making it move by that (x,z) step when the scene is next rendered.

6.4. Rotating a Grabber Joint

If the keycode isn't an arrow key, then moveMidPt() returns false, and rotateJoint() is called (if moveNeedsChecking is false). rotateJoint() decides which grabber is to be affected, and the direction of the rotation.

```
// globals that store the keyboard info for the last joint command
private int keyCode;
private boolean isShift, isAlt;


private void rotateJoint(int keyCode, boolean isShift, boolean isAlt)
{
  // store keyboard info for the joint command
  this.keyCode = keyCode;
  this.isShift = isShift;
  this.isAlt = isAlt;

  // SHIFT means the right grabber
  Grabber grabber = (isShift) ? rightGrabber : leftGrabber;

  if(isAlt)   // ALT means a negative rotation
    rotJoint(grabber, keyCode, Grabber.NEG);
  else
    rotJoint(grabber, keyCode, Grabber.POS);
}  // end of rotateJoint()
```

rotateJoint() also stores the keyboard information for the joint command, which will come in useful later if a collision needs to be undone.

rotJoint() decides which of the joints of the selected grabber should be rotated based on the supplied keycode.

```
// globals
// keys for rotating a grabber's joints
private final static int rotXKey = KeyEvent.VK_X;
private final static int rotYKey = KeyEvent.VK_Y;
private final static int rotZKey = KeyEvent.VK_Z;

// key for rotating a grabber's fingers
private final static int rotfinKey = KeyEvent.VK_F;


private void rotJoint(Grabber grabber, int keycode, int dir)
{
  boolean done;
  if(keycode == rotXKey) {
    done = grabber.rotate(Grabber.X_JOINT, dir);
    if (done)
      moveNeedsChecking = true;  // move needs checking
  }
  else if(keycode == rotYKey) {
    done = grabber.rotate(Grabber.Y_JOINT, dir);
    if (done)
      moveNeedsChecking = true;
  }
  else if(keycode == rotZKey)  // no checking for z- and finger
    grabber.rotate(Grabber.Z_JOINT, dir);
  else if(keycode == rotfinKey)
    grabber.rotate(Grabber.FING_JOINT, dir);
} // end of rotJoint()
```

The four joints are referred to using the Grabber constants X_JOINT, Y_JOINT, Z_JOINT, and FING_JOINT. If either X_JOINT or Y_JOINT are rotated, then moveNeedsChecking is set to true. This prevents further joint command keys from being processed until the rotation has been checked by JointsBehavior calling checkForCollision().

X_JOINT and Y_JOINT are singled out since X_JOINT swings a grabber's forearm left and right, and Y_JOINT moves the forearm up and down. These movements may bring a grabber in contact with the other grabber or the floor.

Z_JOINT and FING_JOINT aren't considered since Z_JOINT rotates the forearm around its major axis, and FING_JOINT opens and closes the fingers. It's unlikely that either will cause collisions.

As you'll see later, this same distinction is applied when collecting collision joints for JointsBehavior. The joints for X_JOINT and Y_JOINT are included, but not Y_JOINT and FING_JOINT. This reduces the number of times that JointsBehavior is woken up, reducing the overhead of including collision detection in Arms3D.

6.5. Collecting the Collision Joints

The Grabbers constructor finishes with a call to buildWakeUps(). References to the collision-causing TransformGroups are converted into a series of "wake up" criteria, and packed into a "wake up" condition for JointsBehavior.

```
// global
private WakeupOr jointsWakeup;


private void buildWakeUps()
{
  TransformGroup[] leftJoints = leftGrabber.getCollisionJoints();
  TransformGroup[] rightJoints = rightGrabber.getCollisionJoints();

  WakeupOnTransformChange[] jointCriteria =
      new WakeupOnTransformChange[leftJoints.length +
                                  rightJoints.length];

  // fill the criteria array with the TGs from both grabbers
  int i = 0;
  for (int j=0; j < leftJoints.length; j++) {
    jointCriteria[i] = new WakeupOnTransformChange( leftJoints[j]);
    i++;
  }
  for (int j=0; j < rightJoints.length; j++) {
    jointCriteria[i] = new WakeupOnTransformChange( rightJoints[j]);
    i++;
  }
  jointsWakeup = new WakeupOr(jointCriteria);  // make the condition
}  // end of buildWakeUps()
```

JointsBehavior should be triggered when any of the TransformGroups is rotated, so the WakeupOnTransformChange objects are wrapped in a WakeupOr instance.

The condition is retrieved by JointsBehavior calling Grabbers' getJointsWakeup() method:

```
public WakeupOr getJointsWakeup()
{ return jointsWakeup; }
```

6.6. Collision Detection and Recovery

When JointsBehavior is woken up, it calls checkForCollision() and checkTouching() in Grabbers.

checkForCollision() generates bounds information for the grabbers, and checks if they intersect. If there's a collision, then it's undone by 'reversing' the joint rotation command that caused it. The keyboard information for that command is utilized to create a new command that reverses the original's rotation direction.

```
// globals: last key command for moving a joint
private int keyCode;
private boolean isShift, isAlt;

private int touchCounter = 1;


public void checkForCollision()
{
  // SHIFT means the right grabber
  Grabber grabber = (isShift) ? rightGrabber : leftGrabber;

  // check if the grabber's bounding spheres intersect
  BoundingSphere[] bs = rightGrabber.getArmBounds();
  if (leftGrabber.touches(bs)) {
    System.out.println((touchCounter++) + ") Arms are touching");

    // reverse the previous joint rotation command
    if(isAlt)
      rotJoint(grabber, keyCode, Grabber.POS);   // was NEG
    else
      rotJoint(grabber, keyCode, Grabber.NEG);   // was POS
  }

  moveNeedsChecking = false;  // since move has been checked
}  // end of checkForCollision()
```

6.7. Why Bother with JointsBehavior?

At first glance, JointsBehavior seems like a needless complication. Since Grabbers processes the rotation command in rotateJoint(), why can't it just call checkForCollision() itself at the end of that method?

The reason is *frame delay*: it takes the Java 3D system two scene redraws (two frame renders) to update the application's TransformGroups. The rotation operation called by rotateJoint() doesn't change a joint immediately, and so rotateJoint() cannot immediately call checkForCollision(). The joints must be monitored by separate code which waits until they are updated, which is the role of JointsBehavior.

6.8. Touching the Floor

JointsBehavior also calls checkTouching() in Grabbers. It examines each grabber to see if its fingers are touching the floor.

```
public void checkTouching()
{
  if (leftGrabber.touchingGround())
    System.out.println("Left Grabber's fingers are
                                  touching the ground");
  if (rightGrabber.touchingGround())
    System.out.println("Right Grabber's fingers are
                                  touching the ground");
}  // end of checkTouching()
```

7. The Grabber

Figures 1 and 2 show the grabbers in action, and a grabber's joints are highlighted in Figure 3.

A grabber has a vertical base, forearm and two fingers. It's forearm points along the -z axis, and can turn left/right, up/down, and spin around it's main axis. It's two fingers are vertically aligned, and can open and close. There are three joints at the elbow between the base and forearm for rotating the forearm around the x-, y-, and z- axes, and one joint for each finger.

The Grabber class performs three tasks:

- It builds the grabber's scene graph, which includes TransformGroups for the joints, Java 3D Cylinders and a Box for its shape, and a shiny metallic material and texture for its appearance.
- It processes rotation commands, moving specified joints, but only within hardwired limits. For example, a finger can only rotate open to a maximum of 20 degrees.
- It supports bounds detection for its forearm and fingers. However, most of the hard work is done by the GrabberBounds object.

7.1. Making the Grabber's Appearance

The constructor sets the grabber's appearance first, then builds the scene graph.

```
// global
private Appearance grabberApp;
```

```
public Grabber(String name, Texture2D tex, float xPos)
{ makeAppearance(tex);
  makeGrabber(name, xPos);
}


private void makeAppearance(Texture2D tex)
// shiny metallic appearance, combining a material and texture
{
  grabberApp = new Appearance();

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

  // shiny metal material
  Color3f alumDiffuse = new Color3f(0.37f, 0.37f, 0.37f);
  Color3f black = new Color3f(0.0f, 0.0f, 0.0f);
  Color3f alumSpecular = new Color3f(0.89f, 0.89f, 0.89f);

  Material armMat =
     new Material(alumDiffuse, black, alumDiffuse, alumSpecular, 17);
             // sets ambient, emissive, diffuse, specular, shininess
  armMat.setLightingEnable(true);
  grabberApp.setMaterial(armMat);

  // apply texture to the shape
  if (tex != null)
    grabberApp.setTexture(tex);
}  // end of makeAppearance();
```

The material and texture are combined with the TextureAttributes.MODULATE mode, which merges the material's lighting effects with the texture's detailing. The best effects are achieved when the material's ambient and diffuse colours are close to the main colour of the texture (in this case, grey).

Figure 8 shows the grabber's appearance when the texture (shown in Figure 7) is applied without a material component.

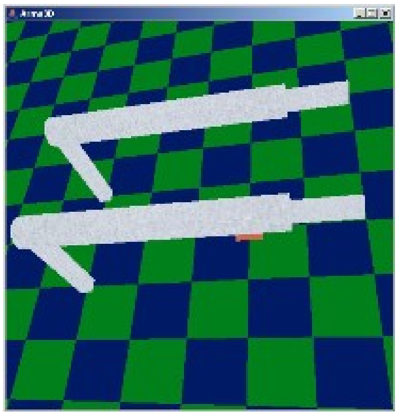

Figure 8. Textured Grabbers without Material.

Figure 9 shows the grabbers with the metallic material but no texturing.

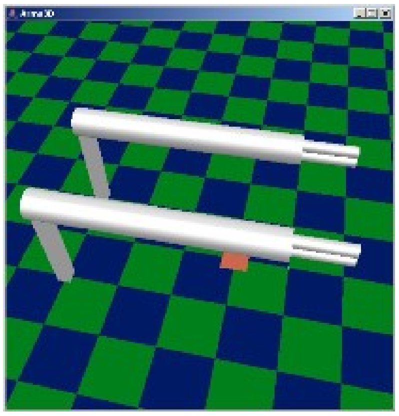
Figure 9. Grabbers with a Material but no Texture.

Figures 1 and 2 show the grabbers covered with the material and texture.

7.2. The Grabber Shape

The grabber is built from four shapes: a tall, skinny box-like base, a cylindrical forearm, and two fingers also made from cylinders. They're linked together by several TransformGroups, with the main ones shown in Figure 10.

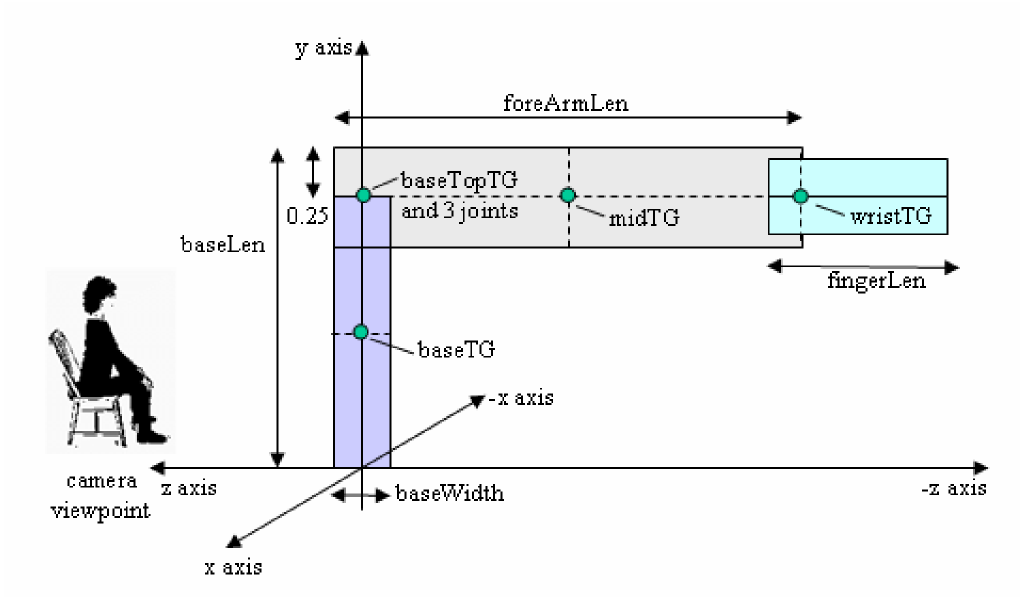

Figure 10. The Elements of a Grabber.

When Arms3D starts, the camera is facing down the -z axis, but Figure 10 shows the view from the right, facing along the -x axis.

The baseTG TransformGroup positions the base so it's resting on the XZ plane. baseTopTG connects the base and forearm, and is the parent of the forearm's three elbow joints (TransformGroups). midTG positions the forearm cylinder so it extends from the left side of the base. wristTG connects the grabber fingers to the forearm, although the fingers use other TransformGroups to rotate (which aren't shown in Figure 10).

The scene graph corresponding to Figure 10 is given in Figure 11. Copies of it occupy the "Grabber" boxes in Figure 6.

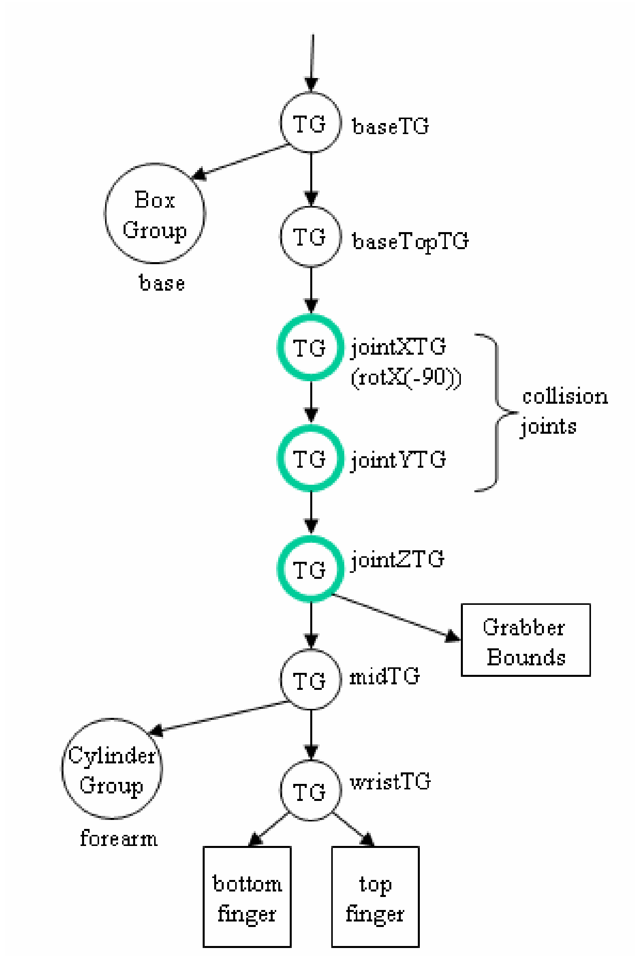

Figure 11. Scene Graph for the Grabber.

The three TransformGroups for rotating the forearm are jointXTG, jointYTG, and jointZTG, and the first two are monitored by JointsBehavior.

The "Grabber Bounds" box refers to the scene graph branch made by the GrabberBounds object, which handles collision detection. The "bottom finger" and "top finger" boxes hide the details of how the fingers are represented.

The graph in Figure 11 is built by makeGrabber().

```
// globals
private TransformGroup baseTG;  // top-level TG

// for collision detection
private GrabberBounds grabberBounds;
private TransformGroup[] collisionJoints;
```

```
       // joint TGs that may cause collisions

// grabber's joints
private TransformGroup jointXTG, jointYTG, jointZTG,
                       topFingTG, botFingTG;


// reusable objects for calculations
private Transform3D t3d = new Transform3D();
private Transform3D toRot = new Transform3D();


private void makeGrabber(String name, float xPos)
{
  // limb dimensions
  float baseWidth = 0.24f;
  float baseLen = 2.0f;
  float foreArmLen = 4.0f;

  /* position baseTG at the base limb's midpoint, so the limb
     will rest on the floor */
  t3d.set( new Vector3f(xPos, baseLen/2, 0));
  baseTG = new TransformGroup(t3d);

  // make base limb
  com.sun.j3d.utils.geometry.Box baseLimb =
      new com.sun.j3d.utils.geometry.Box(baseWidth/2,
                   baseLen/2, baseWidth/2,
                   Primitive.GENERATE_NORMALS |
                   Primitive.GENERATE_TEXTURE_COORDS, grabberApp);
  baseTG.addChild(baseLimb);

  // move to top of base limb
  t3d.set( new Vector3f(0, baseLen/2, 0));
  TransformGroup baseTopTG = new TransformGroup(t3d);
  baseTG.addChild(baseTopTG);

  /* add jointXTG for left/right swings, start it rotated
     by 90 degrees so the rest of the grabber points along
     the -z axis */
  t3d.rotX(Math.toRadians(-90));
  jointXTG = new TransformGroup(t3d);
  jointXTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  jointXTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  jointXTG.setUserData(name + "-X");   // identify the joint
  baseTopTG.addChild(jointXTG);

  // add jointYTG for up/down swings
  jointYTG = new TransformGroup();
  jointYTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  jointYTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  jointYTG.setUserData(name + "-Y");  // identify the joint
  jointXTG.addChild(jointYTG);

  // add jointZTG for spinning the forearm
  jointZTG = new TransformGroup();
  jointZTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  jointZTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  jointYTG.addChild(jointZTG);

  // set up bounds checking for the grabbers forearm and fingers
  grabberBounds = new GrabberBounds(jointZTG,
                                 (1.0f - baseWidth/2), 0.25, 7);

  /* Move to the middle of forearm limb - half width of baseLimb.
     This ensures that the end of the forearm will coincide with the
     left side of the base limb when drawn. */
  t3d.set( new Vector3f(0, (foreArmLen/2 - baseWidth/2), 0));
  TransformGroup midTG = new TransformGroup(t3d);
  jointZTG.addChild(midTG);

  // make forearm limb
  Cylinder foreArmLimb = new Cylinder(0.25f, foreArmLen,
                   Primitive.GENERATE_NORMALS |
                   Primitive.GENERATE_TEXTURE_COORDS, grabberApp);
  midTG.addChild(foreArmLimb);

  // move to end of forearm
  t3d.set( new Vector3f(0, foreArmLen/2, 0));
  TransformGroup wristTG = new TransformGroup(t3d);
  midTG.addChild(wristTG);

  float fingerRadius = 0.1f;

  // create topFingTG for up/down rotation of the top finger
  topFingTG = new TransformGroup();
  makeFinger(fingerRadius, wristTG, topFingTG);

  // create botFingTG for down/up rotation of the bottom finger
  botFingTG = new TransformGroup();
  makeFinger(-fingerRadius, wristTG, botFingTG);

  // store references to the joints that _may_ cause collisions
  collisionJoints = new TransformGroup[2];
  collisionJoints[0] = jointXTG;
  collisionJoints[1] = jointYTG;
    // don't bother with jointZTG and the finger joints
}  // end of makeGrabber()
```

makeGrabber() makes repeated use of the t3d Transform3D object to hold the translations used in the different TransformGroups. This avoids the creation of many temporary objects, which Java would then have to spend time garbage collecting.

The joint TransformGroups (jointXTG, jointYTG, and jointZTG) have their ALLOW_TRANSFORM_READ and ALLOW_TRANSFORM_WRITE capability bits switched on so their positions can be read and changed at runtime.

jointXTG and jointYTG have strings stored with them as 'user data' via calls to setUserData(). This data is read by JointsBehavior to identify the joint that woke it.

The base, the forearm cylinder, and the fingers, are initialized with GENERATE_TEXTURE_COORDS. The generated texture coordinates are used to wrap the steel texture over the shapes.

The GrabberBounds object that deals with collision detection will be explained later. Another part of collision processing is the storage of references to jointXTG and jointYTG in the collisionJoints[] array. This array is retrieved by the Grabbers object, and used to create the WakeupOnTransformChange conditions for JointsBehavior.

7.3. Local and Global Coordinate Systems

The positioning of the TransformGroups and shapes relies on the scene graph nodes having both local and world coordinates.

A node's local coordinate system always has the x-axis running left-to-right, the y-axis straight up, and the z-axis pointing out of the page, as in Figure 12. A new node is positioned at (0,0,0) by default. Translations, rotations, or scaling affect the node according to this local system.

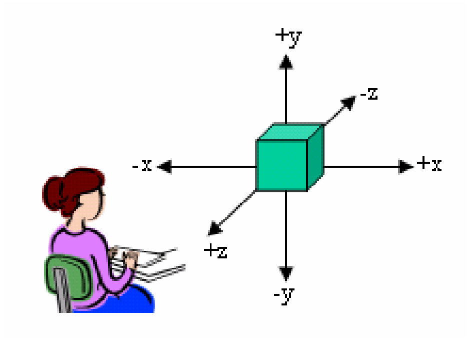

Figure 12. Local Coordinate Axes for a Node.

The world coordinates for a node give it's position in the coordinates system for the entire 3D scene, and depend not only on the node's own transformations but on those applied to its ancestors in the scene graph. For instance, baseTopTG is moved up the y-axis by half the base's height in its local coordinates system:

```
t3d.set( new Vector3f(0, baseLen/2, 0));
TransformGroup baseTopTG = new TransformGroup(t3d);
baseTG.addChild(baseTopTG);
```

In world coordinates, its position also includes the translation of its parent TransformGroup, midTG (shown in Figure 11), which is moved up by half the base height as well. This means that baseTopTG is actually located at (0, baseLen, 0) in the world coordinates system.

The effects of ancestor TransformGroups also include rotations and scaling. For example, jointXTG is initialized with the rotation:

```
  t3d.rotX(Math.toRadians(-90));
  jointXTG = new TransformGroup(t3d);
```

The "right hand" rule can be used to determine the direction of the -90 degree rotation. Clench your right hand, and point your thumb along the +x axis (in Figure 10, that's pointing straight out of the page). The direction of your fingers will be the direction of a positive rotation: to the left in Figure 10. Therefore a negative rotation is to the right.

This rotation not only affects jointXTG, but all the other nodes below it in the Figure 11 scene graph. It means that the forearm and fingers shapes, and the child TransformGroups, are rotated to the right in the world coordinates system.

The local x-, y, and z- axes are rotated so they are orientated as in Figure 13.

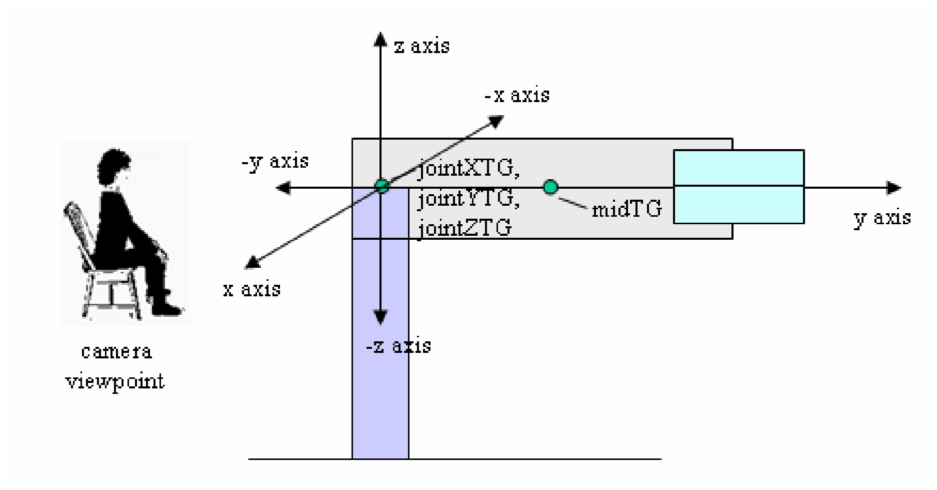

Figure 13. Axes after jointXTG's Rotation.

This explains why the translation from jointZTG to midTG is along the y-axis:

```
// y-axis translation from jointZTG to midTG
t3d.set( new Vector3f(0, (foreArmLen/2 - baseWidth/2), 0));
TransformGroup midTG = new TransformGroup(t3d);
jointZTG.addChild(midTG);
```

jointZTG's positive y-axis is pointing to the right, because of the rotation of its grandparent, jointXTG.

7.4. Making the Fingers

Each finger is created using makeFinger(), and connected to the forearm as in Figure 14. Figure 14 is a close-up of the fingers part of Figure 10.

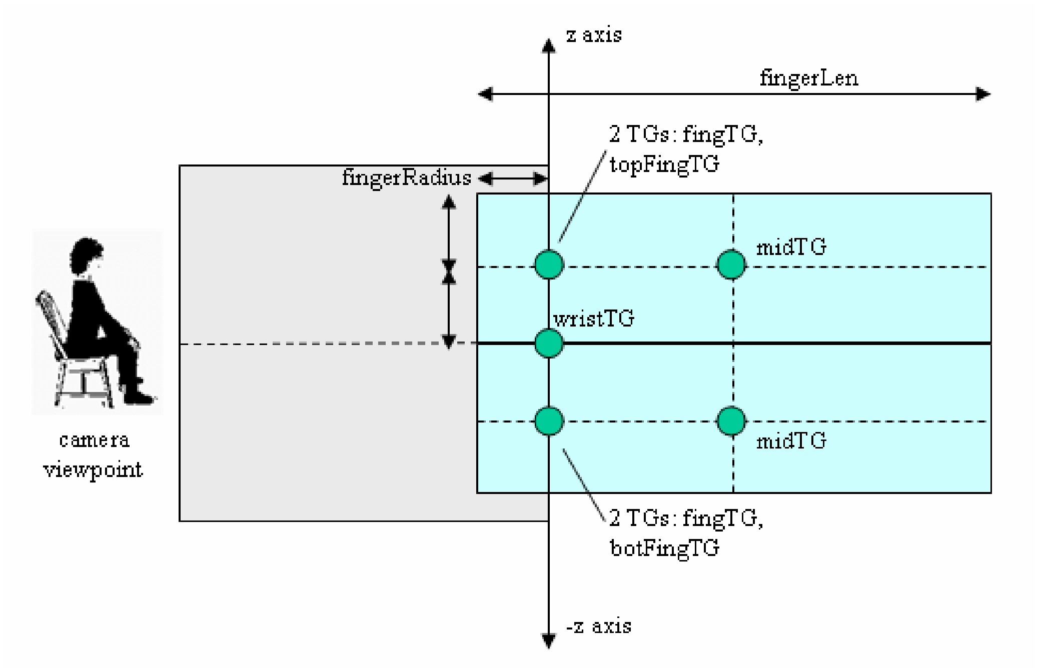

Figure 14. Connecting the Fingers to the Forearm.

wristTG connects the fingers to the forearm, but each finger uses a separate rotation TransformGroup, topFingTG and botFingTG.

Each finger is linked to midTG (which is locally declared in makeFinger()). midTG is positioned so that the left end of the finger overlaps the forearm by fingerRadius units. This ensures that when the finger rotates around topFingTG (or botFingTG) that its left edge doesn't come into view.

The z-axis is aligned vertically because of the rotation in jointXTG.

The scene graph for both fingers is shown in Figure 15; it replaces the "bottom finger" and "top finger" boxes in Figure 11.

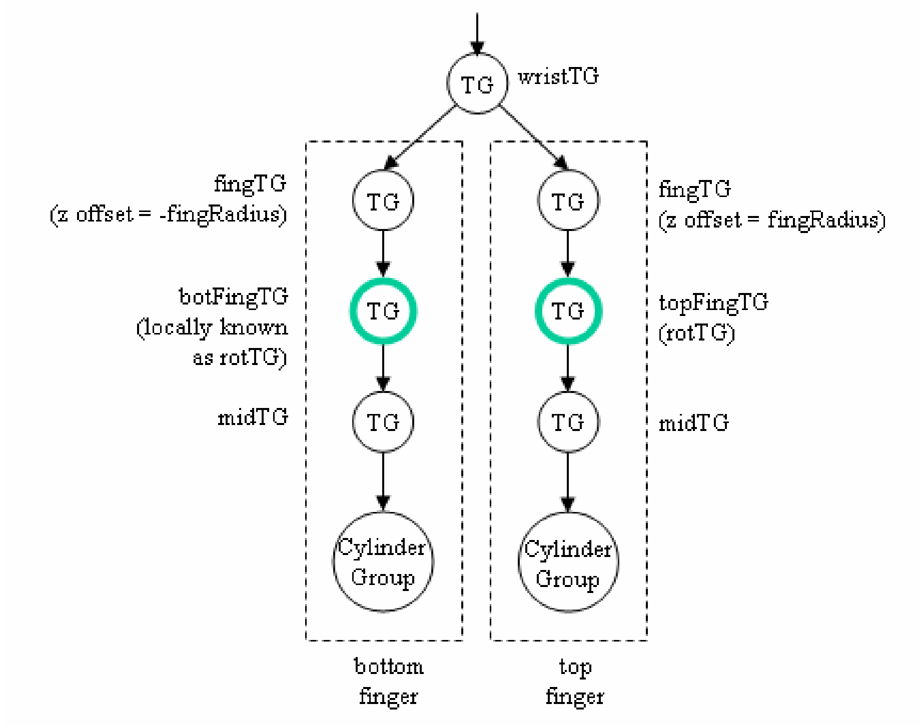

Figure 15. Scene Graph for the Fingers.

Two calls to makeFinger() in makeGrabber() build the scene graph in Figure 15.

```
private void makeFinger(float offset, TransformGroup wristTG,
                                            TransformGroup rotTG)
{
  // finger dimensions
  float fingerRadius = Math.abs(offset);
  float fingerLen = 1.0f;

  // move finger by offset towards the forearm's edge
  t3d.set( new Vector3f(0,0,offset));  // along local z-axis
  TransformGroup fingTG = new TransformGroup(t3d);
  wristTG.addChild(fingTG);

  // add in finger rotation TG
  rotTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  rotTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  fingTG.addChild(rotTG);

  /* forward to middle of finger - radius of finger.
     This will mean that the finger will be drawn with one end
     embedded in the forearm. */
  t3d.set( new Vector3f(0, (fingerLen/2 - fingerRadius), 0));
  TransformGroup midTG = new TransformGroup(t3d);
  rotTG.addChild(midTG);

  // make finger limb
  Cylinder finger = new Cylinder(fingerRadius, fingerLen,
                   Primitive.GENERATE_NORMALS |
                   Primitive.GENERATE_TEXTURE_COORDS, grabberApp);
  midTG.addChild(finger);
}  // end of makeFinger()
```

7.5. Handling Rotation Commands

The Grabbers object passes rotation commands to the Grabber instances. Each command is sent to the relevant grabber, and includes the joint name (X_JOINT, Y_JOINT, Z_JOINT, or FING_JOINT) and the rotation direction (POS or NEG). Grabber's rotate() method converts the direction to an angle (in both degrees and radians), tests if the request is within the rotation limits of the joint, and then carries it out.

```
// globals
// joint rotation amount
private final static int ROTATE_DEG = 2;  // degrees
private final static double ROTATE_AMT = Math.toRadians(ROTATE_DEG);

// rotation joint names (used by Grabbers)
public static final int X_JOINT = 0;
public static final int Y_JOINT = 1;
public static final int Z_JOINT = 2;
public static final int FING_JOINT = 3;

// rotation directions (used by Grabbers)
public static final int POS = 0;
public static final int NEG = 1;

// rotation joints current angle (in degrees)
int xCurrAngle, yCurrAngle, zCurrAngle, fingCurrAngle;


public boolean rotate(int rotType, int dir)
{
  int degAngle;   // angle in degrees
  double angle;   // angle in radians

  // test if the rotation is possible
  if (dir == POS) {
    degAngle = ROTATE_DEG;
    if (!withinRange(rotType, degAngle))
      return false;  // do nothing
    angle = ROTATE_AMT;
  }
  else {  // dir == NEG
    degAngle = -ROTATE_DEG;
    if (!withinRange(rotType, degAngle))
      return false;  // do nothing
    angle = -ROTATE_AMT;
  }

  // carry out the rotation; store the new joint angle
  if (rotType == X_JOINT) {       // left/right movement
    doRotate(jointXTG, X_JOINT, angle);
    xCurrAngle += degAngle;
  }
  else if (rotType == Y_JOINT) {  // up/down movement
    doRotate(jointYTG, Y_JOINT, angle);
    yCurrAngle += degAngle;
  }
  else if (rotType == Z_JOINT) {  // spinning around forearm
    doRotate(jointZTG, Z_JOINT, angle);
    zCurrAngle += degAngle;
  }
  else if (rotType == FING_JOINT) {   // up/down of fingers
    doRotate(topFingTG, Y_JOINT, angle); //up (or down) of top finger
    doRotate(botFingTG, Y_JOINT, -angle);//down (up) of bottom finger
    fingCurrAngle += degAngle;
  }
  else {
    System.out.println("Did not recognise rotation type: "+ rotType);
    return false;
  }

  return true;
}  // end of rotate()
```

Representing the angle as an integer degree makes the coding of withinRange() simpler, while the radian value is used by the Java 3D built-in methods.

FING_JOINT is mapped to two rotations, since the fingers use two TransformGroups (topFingTG, botFingTG) as joints.

Checking the Rotation

withinRange() employs hardwired numerical ranges to determine if a rotation should be carried out. The intended rotation is added to the current joint angle to see if the range is exceeded.

```
private boolean withinRange(int rotType, int degAngle)
{
  int nextAngle;

  if (rotType == X_JOINT) {
    nextAngle = xCurrAngle + degAngle;
    return ((nextAngle >= -45) && (nextAngle <= 45));
  }
  else if (rotType == Y_JOINT) {
    nextAngle = yCurrAngle + degAngle;
    return ((nextAngle >= -45) && (nextAngle <= 45));
  }
  else if (rotType == Z_JOINT) {
    nextAngle = zCurrAngle + degAngle;
    return ((nextAngle >= -90) && (nextAngle <= 90));
  }
  else if (rotType == FING_JOINT) {
    nextAngle = fingCurrAngle + degAngle;
    return ((nextAngle >= 0) && (nextAngle <= 20));
  }

  System.out.println("Did not recognise rotation type: " + rotType);
  return false;
}  // end of withinRange()
```

Doing the Rotation

The joint name is mapped to an axis, and the rotation applied around that axis.

```
// global reusable objects for calculations
private Transform3D t3d = new Transform3D();
private Transform3D toRot = new Transform3D();

private void doRotate(TransformGroup tg, int rotType, double angle)
// rotate the tg joint by angle radians
{
  tg.getTransform(t3d);

  if (rotType == X_JOINT)
    toRot.rotZ(angle);    // left/right == rotation around z-axis
  else if (rotType == Y_JOINT)
    toRot.rotX(angle);    // up/down == rotation around x-axis
  else   // must be Z_JOINT
    toRot.rotY(angle);    // spin == rotation around y-axis

  t3d.mul(toRot);     // t3d = t3d * toRot, which adds the rotation
  tg.setTransform(t3d);
}
```

X_JOINT is the jointXTG joint, which swings the forearm left and right. A quick look at Figure 13 shows that this means a rotation about the z-axis. The right hand rule means that a positive rotation is a swing to the left (from the camera's viewpoint). The same reasoning can be used to understand the Y_JOINT and Z_JOINT rotation choices in doRotate().

To save on temporary objects, the rotation is stored in an existing toRot Transform3D, and then multiplied to the joint's current Transform3D. Once this is stored back in the joint's TransformGroup, the effect is to 'add' the rotation to the joint's current orientation.

7.6. Collision Detection

Grabber delegates the collision detection processing to its GrabberBounds object, which it created in makeGrabber(). The following methods are called by Grabbers:

```
// global
private GrabberBounds grabberBounds;


public BoundingSphere[] getArmBounds()
{  return grabberBounds.getArmBounds();  }

public boolean touches(BoundingSphere[] bs)
{  return grabberBounds.touches(bs); }

public boolean touchingGround()
{  return grabberBounds.touchingGround();  }
```

8. Implementing Collision Detection

GrabberBounds generates an array of TransformGroups which extend from the Grabber's forearm to its fingers. Whenever the grabber moves, the world coordinates for these TransformGroups are recalculated, and used to initialize an array of Java 3D bounding spheres. These spheres are employed for collision detection with bounding spheres in the other grabber.

The last TransformGroup in the array is near the tip of the fingers, so is also used to detect if the fingers are touching the ground.

The positioning of the TransformGroups is shown in Figure 16.

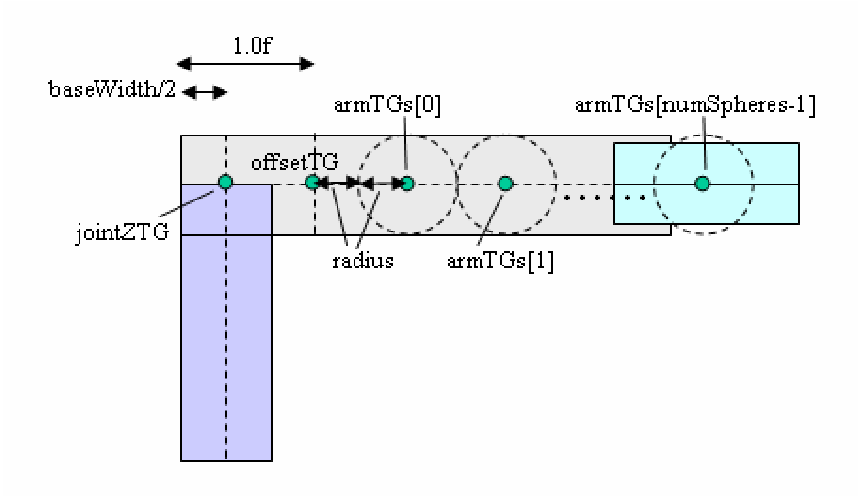

Figure 16. GrabberBounds' TransformGroups.

armTGs[] contains the TransformGroups positioned down the forearm and fingers, starting at offsetTG.

The Java 3D bounding spheres created for each TransformGroup are shown as dotted circles in Figure 16. The TransformGroups are spaced radius*2 units apart, so the spheres just touch each other.

The scene graph made by a GrabberBounds instance is shown in Figure 17; it occupies the "GrabberBounds" box in Figure 11.

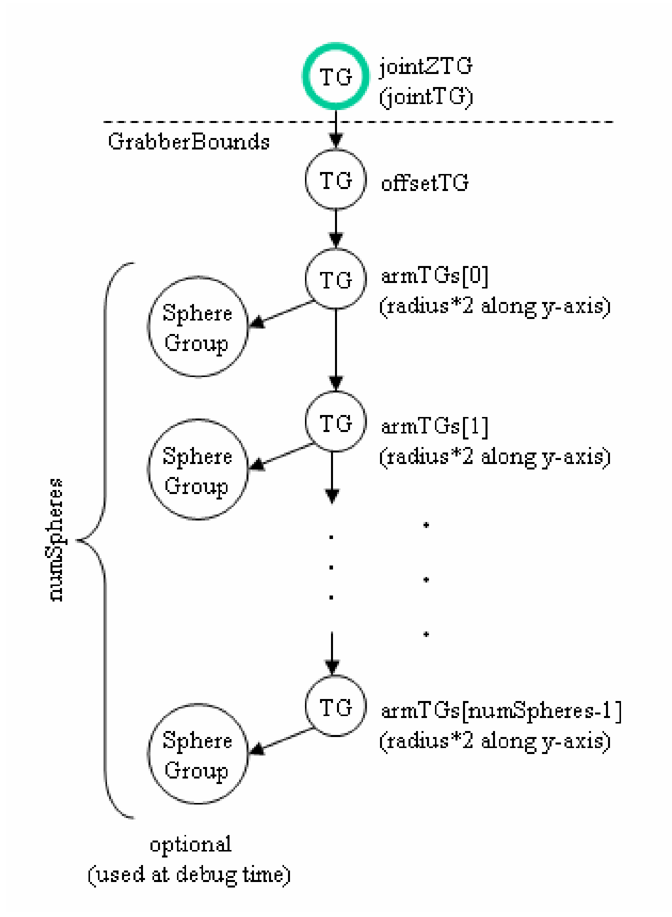

Figure 16. The GrabberBounds Scene Graph.

The Sphere Groups in Figure 16 are *not* Java 3D bounding spheres, but sphere shapes. I use them as visual representations of their parent TransformGroups when debugging the collision detection code. Sphere creation is commented out in the final version of Arms3D.

Each TransformGroup is moved radius*2 units along the y-axis due to the jointXTG rotation shown in Figure 13.

Figure 16 is constructed by makeArmTGs():

```
// globals
private double radius;   // of the bounding spheres
private int numSpheres;

// reusable object for calculations
private Transform3D t3d;

// TGs used for generating the bounding spheres
private TransformGroup offsetTG;
private TransformGroup[] armTGs;
private void makeArmTGs(TransformGroup jointTG, float offset)
{
  Appearance blueApp = makeSphereApp();  // sphere's blue appearance

  // move to offset position, along the arm
  t3d.set( new Vector3f(0, offset, 0));
  offsetTG = new TransformGroup(t3d);
  jointTG.addChild(offsetTG)


  /* Create TGs chain after offsetTG. Start at the last one
     and move backwards towards the offsetTG. This makes it easier
     to attach a child TG to its parent. */
  armTGs = new TransformGroup[numSpheres];
  for (int i=(numSpheres-1); i >= 0; i--) {
    t3d.set( new Vector3f(0, (float)(radius*2), 0));
    armTGs[i] = new TransformGroup(t3d);
    armTGs[i].setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    armTGs[i].setCapability(
                       TransformGroup.ALLOW_LOCAL_TO_VWORLD_READ);
    if (i != numSpheres-1)
      armTGs[i].addChild( armTGs[i+1] );
    // armTGs[i].addChild( new Sphere((float)radius, blueApp) );
             // optionally add a sphere for TG visibility
  }

  offsetTG.addChild(armTGs[0]);
}  // end of makeArmTGs()
```

offsetTG is connected to jointTG (jointZTG in Grabber), the last joint at the start of the forearm. offsetTG is positioned 'offset' units along the forearm after jointTG.

The commented-out line:

```
  armTGs[i].addChild( new Sphere((float)radius, blueApp) );
```

adds a blue sphere to each TransformGroup. Figure 17 shows the grabbers with these spheres (I've also reduced the radius of the forearm cylinders so the spheres are more visible).

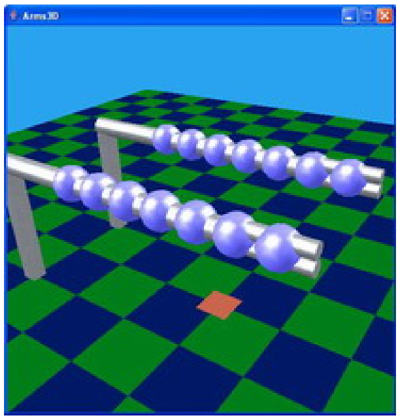

Figure 17. Grabber with Spheres.

8.1. Initializing the Bounding Spheres

For efficiency reasons, the bounds calculations reuse existing bounding spheres, which are initialized by the GrabberBounds constructor calling initBS():

```
// globals
private BoundingSphere[] armBounds;

private void initBS()
{
  armBounds = new BoundingSphere[numSpheres];
  for (int i=0; i < numSpheres; i++) {
    armBounds[i] = new BoundingSphere();
    armBounds[i].setRadius(radius);
  }
}  // end of initBS()
```

The array of bounding spheres are give a specified radius (which will always stay the same), and are positioned at (0,0,0) by default. Before collision detection is performed, their positions are updated with the current locations of the armTGs[] TransformGroups.

8.2. Positioning the Bounding Spheres

When a joint moves, JointsBehavior calls checkForCollision() in Grabbers. It calls getArmBounds() in the right-hand grabber and tests the returned bounds against those in the left-hand grabber with touches():

```
// in checkForCollisions() in Grabbers
BoundingSphere[] bs = rightGrabber.getArmBounds();
if (leftGrabber.touches(bs)) {
  System.out.println((touchCounter++) + ") Arms are touching");
```

There's no particular reason for calling getArmsBounds() on the right-hand grabber and touches() on the left-hand one. The test result would be the same if the grabbers were reversed.

The getArmBounds() and touches() calls pass through Grabber to be processed by GrabberBounds:

```
// in Grabber
public BoundingSphere[] getArmBounds()
{  return grabberBounds.getArmBounds();  }

public boolean touches(BoundingSphere[] bs)
{  return grabberBounds.touches(bs); }
```

getArmBounds() in GrabberBounds updates the positions of the bounding spheres with the locations of the TransformGroups.

```
// global
private Point3d wCoord;  // world coordinate
```

```
public BoundingSphere[] getArmBounds()
{
  for (int i=0; i < numSpheres; i++) {
    setWCoord(armTGs[i]);  // wCoord gets this TG's center point
    armBounds[i].setCenter(wCoord);
  }
  return armBounds;
}
```

setWCoord() calculates the given TransformGroup's position in world coordinates, and stores it in wCoord.

```
// globals
// reusable objects for the TG --> world coordinate calculations
private Transform3D t3d, t3d1;
private Vector3f posnVec;
private Point3d wCoord;


private void setWCoord(TransformGroup tg)
{
  tg.getLocalToVworld(t3d);
        // position, not including this node's transform
  tg.getTransform(t3d1);      // get this TG's transform
  t3d.mul(t3d1);              // 'add' it in

  t3d.get(posnVec);     // extract the position
  wCoord.set(posnVec);  // store it in wCoord
}
```

Java 3D's Node.getLocalToVWorld() returns the world coordinates transformation for the specified node. This combines all the transformations (i.e. translations, rotations, scaling) made by the node's ancestors into a single Transform3D object. Rather confusingly, it doesn't include the transformations applied to the node itself. They have to be obtained with TransformGroup.getTransform(), and then multiplied to the first Transform3D to get the complete transformation. The position component is extracted with Transform3D.get(), and assigned to wCoord.

touches() tests if one of the spheres in the supplied array intersects with one of the object's own bounding spheres.

```
public boolean touches(BoundingSphere[] bs)
{
  BoundingSphere[] myBS = getArmBounds();

  for (int i=(bs.length-1); i >= 0; i--)
    for (int j=(myBS.length-1); j >=0; j--)
      if (bs[i].intersect(myBS[j]))
        return true;
  return false;
}
```

A minor optimization is that the spheres are checked in reverse order in the arrays, which means the spheres nearer the fingers are checked first. This will often find a collision sooner, since it's more likely that the grabbers will touch lower down their arms.

8.3. Touching the Ground

JointsBehavior calls checkTouching() in Grabbers:

```
// in Grabbers
public void checkTouching()
{
  if (leftGrabber.touchingGround())
    System.out.println("Left Grabber's fingers are
                                         touching the ground");
  if (rightGrabber.touchingGround())
    System.out.println("Right Grabber's fingers are
                                         touching the ground");
}
```

Grabber's touchingGround() calls GrabberBounds's touchingGround():

```
// in Grabber
public boolean touchingGround()
{  return grabberBounds.touchingGround();  }
```

GrabberBounds's touchingGround():

```
public boolean touchingGround()
{
  setWCoord(armTGs[numSpheres-1]);  // set wCoord for last TG

  boolean touchingGround = (wCoord.y-radius <= 0);
  if (touchingGround)
    printTuple("last ball touching ground; center: ", wCoord);
  return touchingGround;
}
```

touchingGround() calculates the world coordinates for the last TransformGroup in the armTGs[] array, and checks if its y-value is less than or equal to 0, which means that it's center is on or below the floor.

This code is less complex than a calculation involving the floor's geometry. But it's also less accurate since the center of the last TransformGroup is several units back from the grabber's fingertips, as can be seen in Figure 17.

9. The Floor

The floor is made up of multiple blue and green tiles, and a single red tile, created with my ColouredTiles class, and axis labels made with the Java 3D Text2D class. A close-up of the floor is shown in Figure 18.

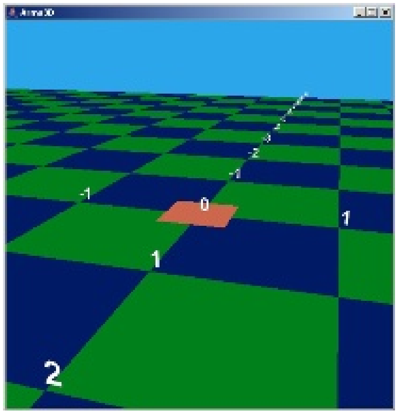

Figure 18. Close to the Floor.

Figure 19 shows the floor branch, previously hidden inside the "Floor Branch" box in Figure 5.

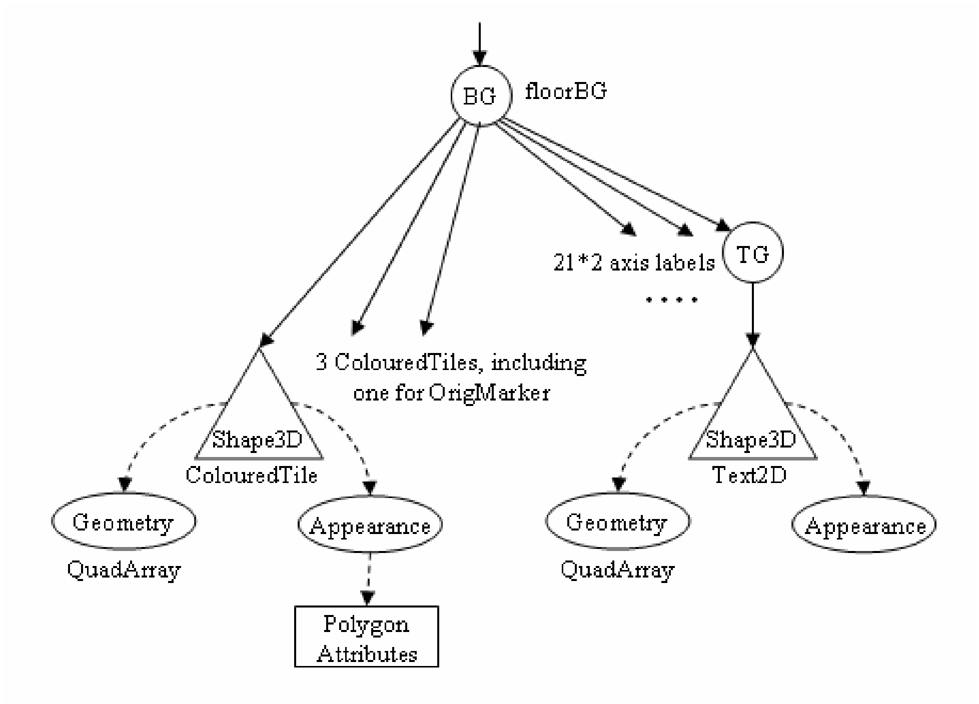

Figure 19. Floor Branch of the Scene Graph.

The floor subgraph is constructed with an instance of my CheckerFloor class, and made available via the getBG() method:

```
// in createSceneGraph() in WrapArms3D
```

```
sceneBG.addChild( new CheckerFloor().getBG() );  // add the floor
```

The CheckerFloor() constructor uses nested for-loops to initialize two ArrayLists. The blueCoords list contains all the coordinates for the blue tiles, and greenCoords holds the coordinates for the green tiles. Once the ArrayLists are filled, they're passed to ColouredTiles objects, along with the colour that should be used to render the tiles. A ColouredTiles object is a subclass of Shape3D, so can be added directly to the floor's graph:

```
floorBG.addChild( new ColouredTiles(blueCoords, blue) );
floorBG.addChild( new ColouredTiles(greenCoords, green) );
```

The red square at the origin (visible in Figure 18) is made in a similar way:

```
Point3f p1 = new Point3f(-0.25f, 0.01f, 0.25f);
Point3f p2 = new Point3f(0.25f, 0.01f, 0.25f);
Point3f p3 = new Point3f(0.25f, 0.01f, -0.25f);
Point3f p4 = new Point3f(-0.25f, 0.01f, -0.25f);

ArrayList<Point3f> oCoords = new ArrayList<Point3f>();
oCoords.add(p1); oCoords.add(p2);
oCoords.add(p3); oCoords.add(p4);

floorBG.addChild( new ColouredTiles(oCoords, medRed) );
```

The square is centered at (0, 0) on the XZ plane, and raised a little above the y-axis (0.01 units) so that it's visible above the tiles.

Each side of the square is of length 0.5 units. The four Point3f points in the ArrayList are stored in a counter-clockwise order. This is also true for each group of four points in blueCoords and greenCoords. Figure 20 shows the ordering of the square's points.

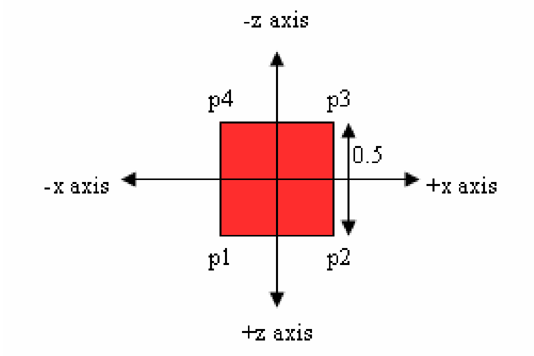

Figure 20. OrigMarker, Viewed from Above.

10. The Coloured Tiles

My ColouredTiles class extends Shape3D, and defines the geometry and appearance of tiles with the same colour. The geometry uses a Java 3D QuadArray to represent the tiles as a series of quadrilaterals (quads). The constructor is:

```
QuadArray(int vertexCount, int vertexFormat);
```

The vertex format is an OR'ed collection of static integers which specify the different aspects of the quad to be initialized later, such as its coordinates, colour, and normals. In ColouredTiles, the QuadArray plane is created using this line of code:

```
plane = new QuadArray(coords.size(),
                GeometryArray.COORDINATES | GeometryArray.COLOR_3 );
```

The size() method returns the number of coordinates in the supplied ArrayList. The coordinate and colour data is supplied in createGeometry():

```
int numPoints = coords.size();
Point3f[] points = new Point3f[numPoints];
coords.toArray( points );   // ArrayList-->array
plane.setCoordinates(0, points);

Color3f cols[] = new Color3f[numPoints];
for(int i=0; i < numPoints; i++)
  cols[i] = col;
plane.setColors(0, cols);
```

The order in which a quad’s coordinates are specified is significant; the front of a polygon is the face where the vertices form a counter-clockwise loop. Knowing front from back is important for lighting and hidden face culling and, by default, only the front face of a polygon will be visible in a scene. In this application, the tiles are oriented so their fronts are facing upwards along the y-axis.

It's also necessary to make sure that the points of each quad from a convex, planar polygon, or rendering may be compromised. However, each quad in the coordinates array doesn't need to be connected or adjacent to the other quads, which is the case for these tiles.

Since a quad's geometry doesn't include normals information, a Material node component can't be used to specify the quad's colour when lit. I could use a ColoringAttributes, but a third alternative is to set the colour in the geometry, as done here (plane.setColors(0, cols);). This colour will be constant, unaffected by the scene lighting.

Once finalized, the Shape3D’s geometry is set with:

```
setGeometry(plane);
```

The shape’s appearance is handled by createAppearance(), which uses a Java 3D PolygonAttribute component to switch off the culling of the back face. PolygonAttribute can also be employed to render polygons in point or line form (i.e. as wire frames), and to flip the normals of back facing shapes:

```
Appearance app = new Appearance();
PolygonAttributes pa = new PolygonAttributes();
pa.setCullFace(PolygonAttributes.CULL_NONE);
app.setPolygonAttributes(pa);
```

Once the appearance has been fully specified, it's fixed in the shape with:

```
setAppearance(app);
```

11. The Floor's Axes Labels

The floor’s axis labels are generated with the labelAxes() and makeText() methods in CheckerFloor(). labelAxes() uses two loops to create labels along the x- and z- axes. Each label is constructed by makeText(), and then added to the floor’s BranchGroup (see Figure 19):

```
floorBG.addChild( makeText(pt,""+i) );
```

makeText() uses the Text2D utility class to create a 2D string of a specified colour, font, point size, and font style:

```
Text2D message = new Text2D(text, white, "SansSerif", 36, Font.BOLD);
                        // 36 point bold Sans Serif
```

A Text2D object is a Shape3D object with a quad geometry (a rectangle), and appearance given by a texture map (image) of the string, placed on the front face. By default, the back face is culled; if the user moves behind an axis label, the object becomes invisible.

The point size is converted to world units by dividing by 256. Generally, it's a bad idea to use too large a point size in the Text2D() constructor since the text may be rendered incorrectly. Instead, a TransformGroup should be placed above the shape and used to scale it to the necessary size.

The positioning of each label is done by a TransformGroup above the shape:

```
TransformGroup tg = new TransformGroup();
Transform3D t3d = new Transform3D();
t3d.setTranslation(vertex);   // the position for the label
tg.setTransform(t3d);
tg.addChild(message);
```

setTranslation() only affects the position of the shape. The tg TransformGroup is added to the floor scene graph.