Chapter N12. Game Pad Grabbers

Back in chapter 4 ??, I developed an example involving two multi-jointed arms with fingers (the *grabbers*). One drawback of the coding was the bewildering mix of key combinations needed to translate and rotate the arms.

This chapter connects a game pad to the grabbers, making them much easier to control, mainly because of the intuitive mapping of the grabbers operations to the pad's analog sticks, hat, and buttons. For example, the left stick controls the x- and y-axis rotations of the left arm, and the right stick handles the right arm.

Another advantage of using a game pad is that it becomes simpler for the code to process multiple inputs at the same time. For instance, the grabbers' base can rotate and move forward while the arms are turning. This is due to the use of polling, which collects information from multiple game pad components at once.

Other novel features of this example are:

- *Obstacles*. The obstacles are rendered as semi-transparent boxes, which can be placed anywhere on the floor. I've surrounded the scene's 3D models and ground shapes with them (as shown in Figure 1 below). The grabbers can't move through obstacles.
- *Collision Detection*. The original grabbers example detects collisions with the help of Java 3D wakeup criteria and a behavior. In this chapter I use a simpler mechanism, a *try-it-and-see* approach, which stops the grabber arms passing through each other, the floor, or traveling into the obstacles. Also, when a collision is detected, the game pad's rumbler is switched on to give the user some feedback.
- *Multiple Input Sources*. The grabbers are controlled via operations sent from the game pad *and* the keyboard, but with the input data converted into GrabberOp objects. This lets the grabbers execute without having to know the source of their operations, which makes it easy to connect other forms of input (such as a mouse).
- *Sound*. I develop a simple SoundsPlayer class which can continuously play music in the background, and short sound clips when the grabbers hit obstacles.
- *Platform Geometry*. The grabbers are attached to the camera viewpoint, so the arms translate and rotate in unison with the camera. This only requires minor adjustments to the grabber classes from chapter 4 ??. Most of the changes are reductions in the dimensions of the component cylinders and boxes because of their proximity to the viewpoint.

Figure 1 shows the ArmsPad3D example.

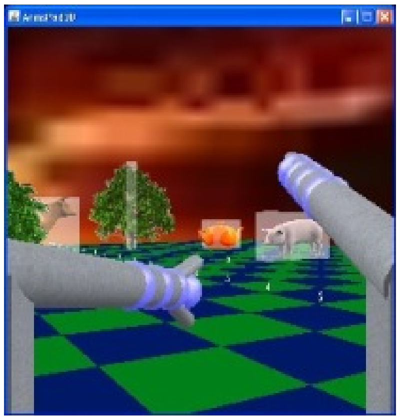

Figure 1. The ArmsPad3D Application.

The 3D models in the scene are OBJ files for a cow, pig, and teapot, loaded using the ModelLoader class from the ObjView3D example in chapter 7 ??. The trees are ground shapes created with the GroundShape class from the same example. The white boxes are obstacles.

The class diagrams for ArmsPad3D in Figure 2 use color-coding to indicate the amount of class reuse from previous examples. Only class names are shown to reduce the diagram's complexity.

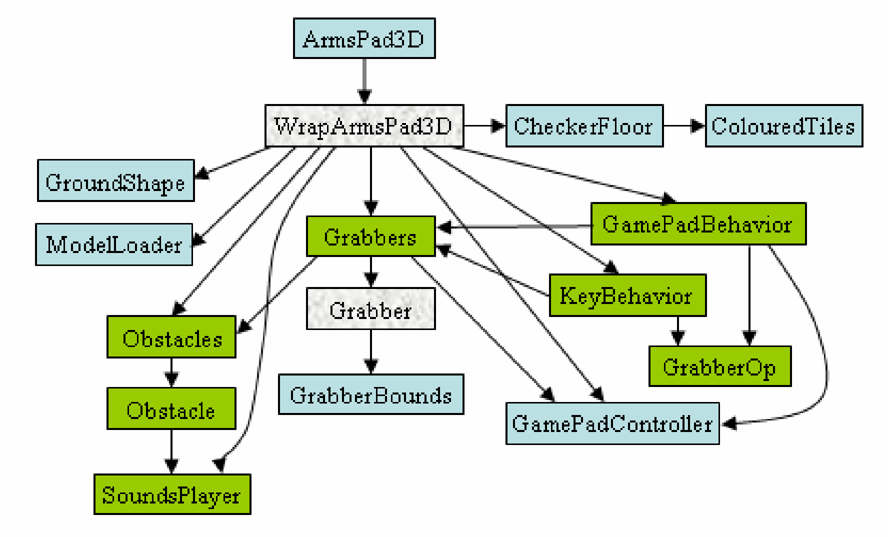

Figure 2. Class Diagrams for ArmsPad3D.

ArmsPad3D is the usual JFrame which utilizes WrapArmsPad3D to render the 3D scene inside a JPanel. Much of WrapArmsPad3D's functionality is familiar from ObjView3D (chapter 7 ??), using CheckerFloor and ColouredTiles to create the floor, GroundShape for the 2D tree billboards, and ModelLoader for the 3D models.

The obstacles are managed by an Obstacles instance, which stores a list of Obstacle objects.

SoundsPlayer plays background music and sound clips.

The grabbers are represented by a Grabbers instance, which delegates arm manipulation to two Grabber instances, and collision detection to two GrabberBounds objects.

Keyboard and game pad input arrive via the KeyBehavior and GamePadBehavior classes, which send operations to Grabbers in the form of GrabberOp instances. GamePadBehavior periodically polls the game pad via a GamePadController instance.

The green (dark gray) boxes in Figure 2 are new classes which I'll be explaining in this chapter (i.e. SoundsPlayer, Obstacles, Obstacle, KeyBehavior, GamePadBehavior, GrabberOp, and Grabbers). There was a Grabbers class in the Arms3D example of chapter 4 ??, but I've changed it so substantially that it qualifies as 'new'.

The light blue (light gray) boxes are for classes unchanged from previous examples, which I won't be looking at again:

- CheckerFloor and ColouredTiles were first used in chapter 4 ??
- GroundShape and ModelLoader were introduced in chapter 7 ??
- GrabberBounds is from chapter 4 ??
- GamePadController was explained in the last chapter.

I'll also skip ArmsPad3D since it only places the WrapArmsPad3D JPanel into a JFrame.

The two boxes with mottled black and white texturing (WrapArmsPad3D and Grabber) have a few new features which I'll be describing, but most of their code is reused. WrapArmsPad3D is very similar to WrapObjView3D from chapter 7 ??, and Grabber is from chapter 4 ??.

In summary, I'll be reusing code from chapters 4, 7, and 11 ??, so it's a good idea if you read those chapters first.

1. Playing Sounds

A SoundsPlayer object acts as a storehouse (jukebox) for sounds, and a sound is played by supplying its name to the SoundsPlayer.

SoundsPlayer is utilized in two ways in ArmsPad3D – to continuously play a MIDI sequence as background music while the application is running, and to play sound clips (short WAV files) when the grabbers hits obstacles.

Java comes with an extensive Java Sound API which supports the recording, playback, and synthesis of sampled audio, such as WAV files, and MIDI. But it's power isn't needed here, since basic playback is possible with the much simpler AudioClip class. For my needs AudioClip is sufficient, with one limitation which can be coded around.

SoundsPlayer stores the MIDI and WAV files as AudioClip objects in a HashMap, soundsMap, who keys are their filenames. It also stores the running time of the clips in a separate soundLensMap HashMap.

```
private HashMap<String, AudioClip> soundsMap;
private HashMap<String, Integer> soundLensMap;
```

The running time is utilized by the Obstacle class, as I'll explain when I get to that code. AudioClip's limitation is that it doesn't offer a method for accessing a clip's running time, so I supply the value as part of SoundsPlayer's load() method.

```
// global
private final static String SOUND_DIR = "Sounds/";


public boolean load(String fnm, int soundLen)
/* Load the sound file and store it as an AudioClip
   in the hash map. Store its length in the soundLens map.
*/
{
  if (soundsMap.get(fnm) != null) {
    System.out.println(SOUND_DIR + fnm + " already loaded");
    return true;
  }

  AudioClip audioClip = Applet.newAudioClip(
            getClass().getResource(SOUND_DIR + fnm) );
  if (audioClip == null) {
    System.out.println("Problem loading " +  SOUND_DIR + fnm);
    return false;
  }

  System.out.println("Loaded " +  SOUND_DIR + fnm);
  soundsMap.put(fnm, audioClip);
  soundLensMap.put(fnm, soundLen);
  return true;
}  // end of load()
```

The clip is loaded with Applet.newAudioClip(), then stored in the soundsMap HashMap.

A sound's running time can be discovered with the help of any decent audio editing tool. On Windows, I use WavePad (http://nch.com.au/wavepad), which offers a great mix of features, and is free.

I only need the running time for the sound clips, so the MIDI file is loaded with the single argument version of load().

```
public boolean load(String fnm)
{  return load(fnm, -1);  }
```

AudioClips can be played once with AudioClip.play(), repeatedly with loop(), or stopped with stop(). These method calls are wrapped up in SoundsPlayer methods which search the soundsMap HashMap for the specified AudioClip. For example:

```
public void play(String fnm)
// retrieve the relevant AudioClip, and start it playing
{
  AudioClip audioClip = (AudioClip) soundsMap.get(fnm);
  if (audioClip == null) {
    System.out.println("Could not find " +  SOUND_DIR + fnm);
    return;
  }
  audioClip.play();
}  // end of play()
```

1.1. Background Music with SoundsPlayer

The SoundsPlayer object is created in WrapArmsPad3D, which also loads and starts the MIDI sequence.

```
// globals in WrapArmsPad3D
private static final String BG_MUSIC = "Mission_Impossible_TV.mid";
private SoundsPlayer soundsPlayer;


// in the WrapArmsPad3D constructor
soundsPlayer = new SoundsPlayer();
soundsPlayer.load(BG_MUSIC);   // load the background MIDI sequence
soundsPlayer.playLoop(BG_MUSIC);  // play it continuously
```

This code snippet should start the pulse-pounding TV theme tune from the 1960's show "Mission Impossible".

Unfortunately, a bug in (some versions of) Java SE 5 and 6 means that nothing may be heard. This is due to a problem with the creation of the MIDI transmitter for playing the sound, a bug that's also present in the Java Sound API. There's a report, no. 6483856, in Sun's bug database at http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6483856. Please vote for it to be fixed.

A key benefit of the MIDI format is that it represents musical data in an extremely efficient way, leading to drastic reductions in file sizes compared to sampled audio (such as WAV files). For instance, files containing high quality stereo sampled audio require about 10 MB per minute of sound, while a typical MIDI sequence may need less than 10 KB.

Even so, other music formats could be used for the background music. One of AudioClip's great strengths is the large number of formats it supports: Windows Wave files, Sun Audio (AU files), Mac AIFF files, different kinds of MIDI (type 0 and type 1), and RMF (Rich Media Format). Data can be 8- or 16- bit, mono or stereo, with sample rates between 8,000 and 48,000 Hz.

Another helpful feature of AudioClip is that different clips can play at the same time, letting the sounds be layered together. However, only a single copy of a clip can be playing at once. This can be too restrictive for some sound effects, such as explosions, where it's common practice to play several instances of a sound so they overlap.

The standard trick is to create multiple copies of the required audio, and load the files as separate AudioClip objects; these 'different' objects can be played simultaneously.

1.2. Obstacle Noises with SoundsPlayer

WrapArmsPad3D loads its 3D models and ground shapes by reusing the ModelLoader and GroundShape classes from chapter 7 ?? At the same time, the obstacles and collision noises are created. The following code fragment from addModels() loads the cow model, its obstacle box, and a "cow.wav" clip:

```
// from addModels() in WrapArmsPad3D
// a cow model
t3d.setTranslation( new Vector3d(-2,0,-4));   // move
t3d.setScale(0.7); // shrink
TransformGroup tg1 = new TransformGroup(t3d);
tg1.addChild( ml.getModel("cow.obj", 1.3) );
sceneBG.addChild(tg1);

soundsPlayer.load("cow.wav", 1500);
obstacles.add("cow.wav", -1.75,-4, 2.55,1.6,1);
```

SoundsPlayer.load()'s numerical argument is the running time of "cow.wav" – 1500 ms (1.5 secs).

The WAV filename is passed to Obstacles.add() so the sound can be played later.

2. Managing Obstacles

The Obstacles class is mostly just a wrapper around an ArrayList of Obstacle objects. It can create a new Obstacle object and add it to the list, and offers several intersection methods.

Obstacles also stores a reference to the SoundsPlayer object, and the Java 3D BranchGroup for the scene, sceneBG. It passes these references to each Obstacle instance so the obstacle can play a sound and attach a semi-transparent box to the scene.

Obstacles' constructor creates the ArrayList and stores the references:

```
// globals
private BranchGroup sceneBG;
private SoundsPlayer soundsPlayer;
private ArrayList<Obstacle> obs;


public Obstacles(BranchGroup sceneBG, SoundsPlayer sp)
{
  this.sceneBG = sceneBG;
  soundsPlayer = sp;
  obs = new ArrayList<Obstacle>();
} // end of Obstacles()
```

The add() method adds an Obstacle instance to the list.

```
public void add(String sndfnm, double x, double z,
                      double xLen, double yLen, double zLen)
{  obs.add( new Obstacle(sceneBG, soundsPlayer, sndfnm,
                    x, z, xLen, yLen, zLen) );  }
```

The long sequence of input arguments are required to position the obstacle inside the scene.

A typical intersection method cycles through the ArrayList, testing a supplied Bounds object against each Obstacle.

```
public boolean intersect(Bounds b)
{
  Obstacle ob;
  for(int i=0; i < obs.size(); i++) {
    ob = obs.get(i);
    if (ob.intersect(b))
      return true;
  }
  return false;
} // end of intersect()
```

3. Making an Obstacle

An obstacle is composed from two things: a Java 3D bounding box for detecting collisions with the grabbers, and a semi-transparent Java 3D box shape located in the same space in the scene. Figure 3 shows a close-up of the box around the cow model (with the pig in the background).

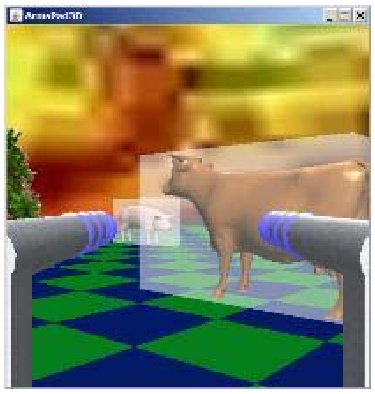

Figure 3. The Cow inside an Obstacle Box.

The main purpose of the visible box is to help me while testing and debugging the application. It could be removed without affecting the collision detection functionality.

Obstacle's dual nature can be seen in its constructor:

```
// globals
private SoundsPlayer soundsPlayer;
private String soundFnm;
```

```
public Obstacle(BranchGroup sceneBG, SoundsPlayer sp,
            String sndfnm, double x, double z,
            double xLen, double yLen, double zLen)
{
  soundsPlayer = sp;
  soundFnm = sndfnm;

  makeBoundBox(x, z, xLen, yLen, zLen);
  if (sceneBG != null)
    addVisibleBox(sceneBG, x, z, xLen, yLen, zLen);
}  // end of Obstacle()
```

makeBoundBox() handles the creation of the bounding box, while the visible box is made by addVisibleBox(). This latter call should be commented out to make the obstacle invisible.

The (x, z) input arguments are used to center the box; no y value is given since the code always rests the obstacle on the ground. The xLen, yLen, and zLen triple are the lengths of the box's sides.

3.1. Making the Boxes

The bounding box is created with the help of Java 3D's BoundingBox class. There are also bounds classes for spheres and more general polyhedra.

```
// globals
private BoundingBox boundBox;


private void makeBoundBox(double x, double z,
                  double xLen, double yLen, double zLen)
{
  Point3d lower = new Point3d(x-xLen/2, 0, z-zLen/2);
  Point3d upper = new Point3d(x+xLen/2, yLen, z+zLen/2);
  boundBox = new BoundingBox(lower, upper);
}  // end of makeBoundBox
```

Making the visible box takes a bit more work, but utilizes Java 3D's Box class for the geometric aspects. It occupies the same space as the bounding box: centered at (x, z) on the XZ plane, with sides of xLen, yLen, and zLen. It's appearance is a default white, but translucent.

```
private void addVisibleBox(BranchGroup sceneBG, double x, double z,
                          double xLen, double yLen, double zLen)
{
  Appearance app = new Appearance();    // default white colour

  // 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);

  Box obsBox = new Box( (float)xLen/2, (float)yLen/2,
                              (float)zLen/2, app);
  // fix box position
  Transform3D t3d = new Transform3D();
  t3d.setTranslation( new Vector3d(x, yLen/2+0.01, z) );
                    // bit above ground

  TransformGroup tg = new TransformGroup();
  tg.setTransform(t3d);
  tg.addChild(obsBox);

  sceneBG.addChild(tg);
}  // end of addVisibleBox()
```

Face culling is switched off so the user can see the inside walls of a box when they look inside an obstacle.

The box is positioned a bit above the floor so that its rendering won't interact with the box's base. It's attached to the scene via the sceneBG BranchGroup.

3.2. Collision Detection

Obstacle contains two versions of an intersect() method: one for Bounds input, one for a point. The methods are quite similar; here's the Bounds version:

```
public boolean intersect(Bounds b)
/* If b intersects with the bounding box for this object
   then play a sound. */
{
  boolean isIntersecting = boundBox.intersect(b);
  if (isIntersecting)
    playSound();
  return isIntersecting;
}  // end of intersect()
```

3.3. Playing a Collision Sound

There's a tricky aspect to playing a collision sound – what to do when multiple collisions occur in rapid succession?

Rapid player movement is achieved by the user holding down a key, or a button on the game pad. When a collision occurs, the user will take a few milliseconds to release the key/button, and in that time the collision will be detected multiple times.

If each collision triggers sound output, then the result is a rapid noisy stutter, caused by the clip restarting repeatedly. It's restarted due to AudioClip's restriction that multiple instances of the same clip can't be played concurrently.

This stutter may not be a drawback for some applications, but I'd like to avoid it in ArmsPad3D. If there's still a collision after the sound's end, then it can be played again then.

Another aspect of sound playing is that it shouldn't 'freeze' the rest of the application; a sound effect may last for 1-2 seconds, which is much too long for the grabbers to be unresponsive.

The solution to these issues is to use a thread to play the sound, with a boolean to control when the thread is executed.

```
// globals
private String soundFnm;
private SoundsPlayer soundsPlayer;
private boolean isSoundPlaying = false;


private void playSound()
{
  if ((soundFnm != null) && !isSoundPlaying) {
    Thread player = new Thread() {
      public void run()
      {
        isSoundPlaying = true;
        int sndLen = soundsPlayer.getLength(soundFnm);
        if (sndLen == -1)
          sndLen = 1000;   // reasonable waiting time (1 sec)
        soundsPlayer.play(soundFnm);
        try {
          sleep(sndLen);   // wait for sound to finish
        }
        catch(InterruptedException ex) {}
        isSoundPlaying = false;
      }
    };
    player.start();
  }
}  // end of playSound()
```

playSound() plays the collision sound in a separate thread, thereby allowing the rest of the application to continue executing. The isSoundPlaying boolean permits the method to ignore new sound playing requests if the sound is already playing – the call returns without starting a thread.

This approach introduces a new problem: *how long* should the boolean be set to true while the sound plays to its end? Since the AudioClip class has no running time method, it falls to me to code one up inside SoundsPlayer. The retrieved time is used to make the thread sleep for as long as the sound is playing, after which the boolean is set back to false. If there isn't a time available, then the thread 'wings it' by sleeping for a second.

The code might suffer from a race condition since there's a very short interval between the if-test at the start of playSound() and the setting of isSoundPlaying at the beginning of the thread. This might be enough time for a second call to playSound() to get past the test and create a second thread. But it's quite unlikely because the interval between user inputs (key presses or button presses) is in the 10's of milliseconds which is normally much longer than the time to create a thread. Even if the race condition occurs, there's no serious consequence except a small amount of stutter as the sound starts playing.

4. Sending Input to the Grabbers

Movement requests are sent to the grabbers either from the keyboard or the game pad. Instead of having the grabbers deal with their data directly, the information is converted into GrabberOp instances. This enforces a clean separation between the input sources (the keyboard and game pad) and the input's processing in Grabbers. The Grabbers code becomes much less convoluted, and it's straightforward to add new input sources, such as the mouse or a data glove.

The keyboard, game pad, and the grabbers are connected in addGrabbers() in WrapArmsPad3D. The relevant code fragment is shown below:

```
// in addGrabbers() in WrapArmsPad3D

GamePadController gamePad = new GamePadController();

Grabbers grabbers = new Grabbers(. . .);
   // Grabbers input arguments will be explained later

// set up game pad behavior to catch game pad input
GamePadBehavior gamePadBeh = new GamePadBehavior(grabbers, gamePad);
gamePadBeh.setSchedulingBounds(bounds);
sceneBG.addChild(gamePadBeh);

// set up keyboard controls to catch keypresses
KeyBehavior keyBeh = new KeyBehavior(grabbers);
keyBeh.setSchedulingBounds(bounds);
sceneBG.addChild(keyBeh);
```

KeyBehavior sends key presses to the grabbers as GrabberOp instances, while GamePadBehavior does the same for game pad input.

4.1. Processing Keyboard Input

The version of KeyBehavior in ArmsPad3D differs from other KeyBehaviors I've explained (e.g. KeyBehavior in ObjView3D). They contain methods which convert the keyboard data into translations and rotations applied to TransformGroups. This KeyBehavior class passes the data to a GrabberOp instance to initialize it, then sends it to the Grabber instance for processing:

```
// global
private Grabbers grabbers;


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

  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 ) {
          gop = new GrabberOp( (KeyEvent)event[i] );
                      // make a GrabberOp
          if (!gop.isOp(GrabberOp.NONE))
            grabbers.processOp(gop);
              // send it to Grabbers for processing
        }
      }
    }
  }
  wakeupOn( keyPress );
} // end of processStimulus()
```

The call to GrabberOp.isOp() tests if the generated operation is a no-op (one that does nothing), in which case processOp() isn't called.

4.2. Building a Grabber Operation for Keypresses

GrabberOp represents a grabber operation using 4 integer variables:

- opVal: it holds the grabber operation name (as an integer);
- partVal: it holds the integer name for the grabber part being used by this operation;
- jointVal: it holds the integer name of the joint being used by this operation;
- rotVal: holds the rotation direction (positive or negative) for joint operations.

```
// global state vars in GrabberOp
private int opVal, partVal, jointVal, rotVal;
```

Their values come from a very large set of public integer constants. They're public so they can be used by other classes (e.g. Grabbers and Grabber).

A GrabberOp object is created either from keystrokes (sent by KeyBehavior) or from game pad input (sent by GamePadBehavior), so there are three constructors: a default, simple version, one for key input, and one for game pad input. I'll explain the key-based constructor and its related methods here, and return to the game pad methods after explaining GamePadBehavior.

The key-based constructor calls set() methods which examine the input to decide how to assign values to opVal, partVal, jointVal, and rotVal.

```
// globals
// better names for the translation/rotation keys
// for translating the grabbers
private final static int forwardKey = KeyEvent.VK_UP;
private final static int backKey = KeyEvent.VK_DOWN;
private final static int leftKey = KeyEvent.VK_LEFT;
private final static int rightKey = KeyEvent.VK_RIGHT;


public GrabberOp()
{ // default values
  opVal = NONE;
  partVal = NO_PART;
  jointVal = NOT_JOINT;
  rotVal = NO_ROT;
} // end of GrabberOp();


public GrabberOp(KeyEvent eventKey)
// create a GrabberOp from keystrokes (sent by KeyBehavior)
{
  this();   // assign default values

  // the shift and alt keys are used as modifiers
  int keyCode = eventKey.getKeyCode();
  boolean isShift = eventKey.isShiftDown();
  boolean isAlt = eventKey.isAltDown();

  if((keyCode == forwardKey) || (keyCode == backKey) ||
     (keyCode == leftKey) || (keyCode == rightKey))
    setBaseOp(keyCode, isAlt);
  else {  // grabbers
    if (isShift)    // right grabber
      setRightGrabberOp(keyCode, isAlt);
    else  // left grabber
      setLeftGrabberOp(keyCode, isAlt);
  }
}  // end of GrabberOp() the keyboard
```

The grabbers respond to the arrow keys, and 'x', 'y', 'z', and 'f', which can be combined with the shift and alt keys to affect their meaning. The pressed keys are identified at the start of the GrabberOp() constructor.

The grabbers consist of three parts: the base, and the left and right arms. The base can translate and rotate, while each arm has x-, y-, and z- axis joints and fingers that can open and close. Based on the keys, setBaseOp(), setLeftGrabber(), or setRightGrabber() are called in order to assign values to the GrabberOp state variables (opVal, partVal, jointVal, and rotVal).

setBaseOp() gives a flavor of the coding of these set methods:

```
private void setBaseOp(int keycode, boolean isAlt)
/* Make grabbers move forward or backward;
   rotate left or right */
{
  partVal = BASE;
  if(isAlt) {   // key + <alt>
    if(keycode == forwardKey)
      opVal = BASE_UP;
    else if(keycode == backKey)
      opVal = BASE_DOWN;
    else if(keycode == leftKey)
      opVal = BASE_LEFT;
    else if(keycode == rightKey)
      opVal = BASE_RIGHT;
  }
  else {  // just <key>
    if(keycode == forwardKey)
      opVal = BASE_FWD;
    else if(keycode == backKey)
      opVal = BASE_BACK;
    else if(keycode == leftKey)
      opVal = BASE_ROT_LEFT;
    else if(keycode == rightKey)
      opVal = BASE_ROT_RIGHT;
  }
} // end of setBaseOp()
```

For example, if the up-arrow (forwardKey) and alt keys are pressed together, then the resulting GrabberOp operation will specify that the grabbers base be lifted up. The values assigned to the state variables are:

- `opVal == BASE_UP`
- `partVal == BASE`
- `jointVal == NONE`
- `rotVal == NONE`

jointVal and rotVal are unaffected since they refer to the arm joints which aren't used in base-related operations.

Each of the four state variables has its own get, set, and is-test methods which makes it simple to query the GrabberOp states. For example, opVal is accessed with:

```
public int getOp()
{  return opVal;  }

public boolean isOp(int op)
{  return opVal == op;  }

public void setOp(int op)
{  opVal = op;  }  // no checking of op
```

setOp() isn't generally needed since the constructor performs all the state initialization.

4.3. Processing Game Pad Input

GamePadBehavior periodically polls the game pad controller, and converts the component settings into new GrabberOp objects. These are past to the Grabbers object for processing.

The constructor creates an ArrayList to hold the GrabberOp objects, and initializes the time-based wakeup condition.

```
// globals
private static final int DELAY = 75;   // ms (polling interval)

private Grabbers grabbers;
private ArrayList<GrabberOp> gops;
               // storage for the created GrabberOps
private GamePadController gamePad;
private WakeupCondition wakeUpCond;


public GamePadBehavior(Grabbers gs, GamePadController gp)
{
  grabbers = gs;
  gamePad = gp;

  gops = new ArrayList<GrabberOp>();
  wakeUpCond = new WakeupOnElapsedTime(DELAY);
} // end of GamePadBehavior()
```

processStimulus() is called every DELAY milliseconds, polls GamePadController to refresh its state information, then examines the left and right sticks, the hat, and buttons. The resulting list of GrabberOps is sent over to Grabbers for processing.

```
public void processStimulus(Enumeration criteria)
{
  gops.clear();     // empty the GrabberOps list

  gamePad.poll();   // update the game pad's components

  /* create GrabberOps for the game pad components:
        left/right sticks, a hat, and buttons
  */
  processLeftStick( gamePad.getXYStickDir() );
  processRightStick( gamePad.getZRZStickDir() );
  processHat( gamePad.getHatDir() );
  processButtons( gamePad.getButtons() );

  // send the GrabberOps to Grabbers
  for(int i=0; i < gops.size(); i++)
    grabbers.processOp( gops.get(i) );

  wakeupOn(wakeUpCond);       // make sure we are notified again
} // end of processStimulus()
```

The 'process' methods in GamePadBehavior test for the various possible states of the game pad components, and build GrabberOps in response. All the methods are quite similar; for example, processLeftStick() converts the eight compass directions of the left stick into GrabberOps.

```
private void processLeftStick(int dir)
{
  if ((dir == GamePadController.NORTH) ||
      (dir == GamePadController.SOUTH) ||
      (dir == GamePadController.EAST) ||
      (dir == GamePadController.WEST))
      gops.add( new GrabberOp(GamePadController.LEFT_STICK, dir) );
  else if (dir == GamePadController.NE) {
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.NORTH) );
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.EAST) );
  }
  else if (dir == GamePadController.NW) {
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.NORTH) );
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.WEST) );
  }
  else if (dir == GamePadController.SE) {
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.SOUTH) );
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.EAST) );
  }
  else if (dir == GamePadController.SW) {
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.SOUTH) );
    gops.add( new GrabberOp(GamePadController.LEFT_STICK,
                            GamePadController.WEST) );
  }
}  // end of processLeftStick()
```

The diagonal directions are converted into two operations each, which means that only the GamePadController constants NORTH, SOUTH, EAST, and WEST need to be processed over in GrabberOp.

The translation is complicated by the large range of input types and output operations. The two sticks and hat can be assigned eight different compass directions each, and there are 12 buttons. There are 24 different grabber operations. The complexity is reduced by using public constants rather than a bevy of anonymous numbers and booleans.

4.4. Building a Grabber Operation for the Game Pad

The GrabberOp constructor for the game pad must map all the possible input types to GrabberOp operations. It does this by dividing the input into four groups: data from the left stick, the right stick, the hat, and buttons.

```
public GrabberOp(int gamePadComponent, int val)
// create a GrabberOp from the game pad input
{
  this();

  // input may be from left or right stick, hat, or buttons
  if (gamePadComponent == GamePadController.LEFT_STICK)
    setLeftStick(val);
  else if (gamePadComponent == GamePadController.RIGHT_STICK)
    setRightStick(val);
  else if (gamePadComponent == GamePadController.HAT)
    setHat(val);
  else if (gamePadComponent == GamePadController.BUTTONS)
    setButton(val);
  else
    System.out.println("Do not recognize game pad component: " +
                            gamePadComponent);
}  // end of GrabberOp() for the game pad
```

The 'set' operations for game pad input have the same structure as the keyboard-based ones I described above: a series of if-tests decide what values to assign to opVal, partVal, jointVal, and rotVal. setLeftStick() is a typical method (it creates a GrabberOp for affecting the grabber's left arm):

```
private void setLeftStick(int dir)
/* Deal with input from the game pad's left stick.
```

```
   The left stick maps to the left grabber. */
{
  partVal = LEFT_GRABBER;

  if (dir == GamePadController.NORTH) {   // y up
    opVal = LEFT_Y_POS;
    jointVal = Y_JOINT;  rotVal = POS;
  }
  else if (dir == GamePadController.SOUTH) {  // y down
    opVal = LEFT_Y_NEG;
    jointVal = Y_JOINT;  rotVal = NEG;
  }
  else if (dir == GamePadController.EAST) {  // x right
    opVal = LEFT_X_NEG;
    jointVal = X_JOINT;  rotVal = NEG;
  }
  else if (dir == GamePadController.WEST) {  // x left
    opVal = LEFT_X_POS;
    jointVal = X_JOINT;  rotVal = POS;
  }
}  // end of setLeftStick()
```

Each arm has 4 joints (x-, y-, z-, and the fingers), and each joint can be rotated in a positive or negative direction. For instance, if the user presses the left stick upwards (which is denoted by GamePadController.NORTH), then the arm's y-axis joint should be rotated in a positive direction to move it up. The GrabberOp states are set to:

- `opVal == LEFT_Y_POS`
- `partVal == LEFT_GRABBER`
- `jointVal == Y_JOINT`
- `rotVal == POS`

It may seem that opVal contains all the necessary information (left, y, pos), which is needlessly duplicated by partVal, jointVal, and rotVal. The duplication is useful because the part, joint and rotation information become easier to access from Grabbers and Grabber. The alternative would be additional code to extract the component attributes from opVal.

5. The Grabbers

The grabbers consists of a base, and left and right arms, just as they did in chapter 4 ??. However, the base can be translated *and* rotated (the old version could only translate). Also, the grabbers are attached to the camera viewpoint via a PlatformGeometry instance, so the base is moved by affecting the viewpoint's targetTG TransformGroup. Movement commands may come from the keyboard or the game pad, arriving as GrabberOp objects.

Collision processing is more elaborate. An operation is tried out, and if the resulting base or arm positions intersect with each other, the obstacles, or the floor, then the operation is undone. This *try-it-and-see* technique is employed before the scene is rendered, so a move followed by an undo won't be displayed. The user only notices that the grabbers don't move. This is quite different from the approach in chapter 4 ??

where the collision was detected after the offending operation had been rendered. A behavior was triggered which executed a reverse operation to undo the move. To the user this looks like the base or arm 'bouncing' back from its move.

A collision with an obstacle triggers the playing of a sound in the Obstacle object, and the game pad rumbles.

5.1. Connecting the Grabbers to the Scene

The grabbers are attached to the camera viewpoint in addGrabbers() in WrapArmsPad3D. The relevant code fragment:

```
// in addGrabbers() in WrapArmsPad3D...

// get targetTG, the TG for the viewpoint
ViewingPlatform vp = su.getViewingPlatform();
TransformGroup targetTG = vp.getViewPlatformTransform();

// create the grabbers
Vector3d posnVec = new Vector3d(0, -0.5, -0.9);
Grabbers grabbers = new Grabbers(posnVec, 0.4f,
                           targetTG, obstacles, gamePad);
  // supply grabbers position and each arm's x-axis offset

// add grabbers to the viewpoint
PlatformGeometry pg = new PlatformGeometry();
pg.addChild( grabbers.getTG() );
vp.setPlatformGeometry(pg);
```

The PlatformGeometry object is essentially a BranchGroup attached to the targetTG TransformGroup which moves the viewpoint. Figure 4 shows the view branch subgraph.

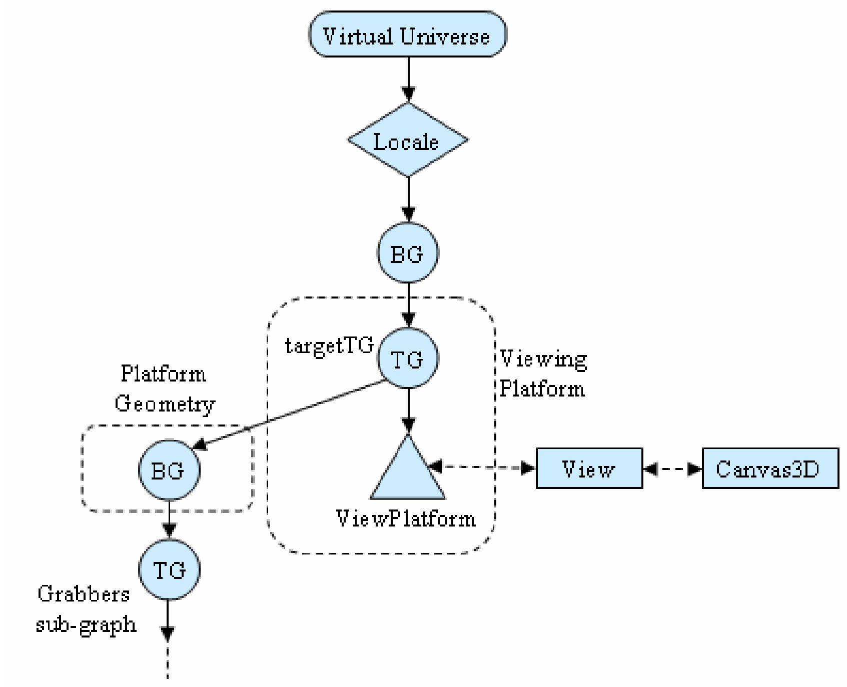

Figure 4. The View Branch Subgraph.

A reference to targetTG is passed into Grabbers since the grabbers move by changing the viewpoint. Grabbers also receives a position vector (posnVec), which is used to place it a little way in front of (and below) the viewpoint. I decided on (0, -0.5, -0.9) by experimentation.

5.2. Constructing the Grabbers

The scene graph for the grabbers is a TransformGroup (grabbersTG) parent with two Grabber subgraphs for the left and right arms.

```
// globals
// scene graph elements
private TransformGroup targetTG;
            // the ViewPlatform's transform group
private TransformGroup grabbersTG;
private Grabber leftGrabber, rightGrabber;

private Obstacles obs;    // for collision detection
private GamePadController gamePad;   // used to switch on rumbling


public Grabbers(Vector3d posnVec, float grabOffset,
                  TransformGroup targetTG,
                  Obstacles obs, GamePadController gp)
{
  this.targetTG = targetTG;
  this.obs = obs;
  gamePad = gp;

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

  // position the grabbers
  t3d.set(posnVec);
  grabbersTG = new TransformGroup(t3d);

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

  // add the right grabber
  rightGrabber = new Grabber("right", tex, grabOffset);
  grabbersTG.addChild( rightGrabber.getBaseTG() );
}  // end of Grabbers()


public TransformGroup getTG()
// called by WrapArms3D
{  return grabbersTG;  }
```

5.3. Processing an Operation

processOp() tries out the requested GrabberOp operation in doOp(). A resulting collision causes the operation to be undone, and Grabbers notifies the user by rumbling the game pad.

```
synchronized public void processOp(GrabberOp gop)
{
  if (doOp(gop))
    if (isColliding()) {
      rumblePad();
      undoOp(gop);
    }
} // end of processOp()
```

processOp() is synchronized since it can be called by two behaviors (KeyBehavior and GamePadBehavior) which may be running concurrently.

The "try-it-and-see" approach puts the control logic in one class, called from processOp(). By comparison, the behavior triggering employed in chapter 4 ?? spreads the collision processing over several classes, and connects it with rather opaque event handling.

Doing an Operation

An operation may either affect the grabbers base, or one of the arms.

```
private boolean doOp(GrabberOp gop)
{
  if (gop.isOp(GrabberOp.NONE))
    return false;
  boolean done = false;
  if (gop.isPart(GrabberOp.BASE))
    done = affectBase(gop);
  else if (gop.isPart(GrabberOp.LEFT_GRABBER))
    done = leftGrabber.rotate(gop);
  else if (gop.isPart(GrabberOp.RIGHT_GRABBER))
    done = rightGrabber.rotate(gop);
  return done;
}  // end of doOp()
```

The code uses GrabberOp.isPart() to examine the partVal state in the GrabberOp instance. The processing for the arms is carried out by the Grabber instances, but base operations are left to Grabbers.

The base can be translated or rotated, and the choice comes down to an examination of the opVal state in the GrabberOp object.

```
// globals
// used when rotating the base
private static final double ROT_AMT = Math.PI / 36.0; // 5 degrees
private static final double MOVE_STEP = 0.2;

// hardwired movement vectors used when translating the base
private static final Vector3d FWD = new Vector3d(0,0,-MOVE_STEP);
private static final Vector3d BACK = new Vector3d(0,0,MOVE_STEP);
private static final Vector3d LEFT = new Vector3d(-MOVE_STEP,0,0);
private static final Vector3d RIGHT = new Vector3d(MOVE_STEP,0,0);
private static final Vector3d UP = new Vector3d(0,MOVE_STEP,0);
private static final Vector3d DOWN = new Vector3d(0,-MOVE_STEP,0);

private int upMoves = 0;


private boolean affectBase(GrabberOp gop)
{
  if (gop.isOp(GrabberOp.BASE_FWD))
    doMove(FWD);
  else if (gop.isOp(GrabberOp.BASE_BACK))
    doMove(BACK);
  else if (gop.isOp(GrabberOp.BASE_ROT_LEFT))
    rotateY(ROT_AMT);
  else if (gop.isOp(GrabberOp.BASE_ROT_RIGHT))
    rotateY(-ROT_AMT);
  else if (gop.isOp(GrabberOp.BASE_UP)) {
    upMoves++;
    doMove(UP);
  }
  else if (gop.isOp(GrabberOp.BASE_DOWN)) {
    if (upMoves > 0) {  // don't drop below start height
      upMoves--;
      doMove(DOWN);
    }
    else
      return false;   // since doing nothing
  }
  else if (gop.isOp(GrabberOp.BASE_LEFT))
    doMove(LEFT);
  else if (gop.isOp(GrabberOp.BASE_RIGHT))
    doMove(RIGHT);
  return true;
}  // end of affectBase()
```

This code should look familiar from previous chapters. doMove() and rotateY() apply translations and rotations to the targetTG TransformGroup, thereby moving the camera viewpoint, and the grabbers attached to it.

```
// globals
// used for repeated calcs
private Transform3D t3d = new Transform3D();
private Transform3D toMove = new Transform3D();
private Transform3D toRot = new Transform3D();


private void doMove(Vector3d theMove)
// move targetTG by the amount in theMove
{
  targetTG.getTransform(t3d);
  toMove.setTranslation(theMove);
  t3d.mul(toMove);
  targetTG.setTransform(t3d);
} // end of doMove()


private void rotateY(double radians)
// rotate about the y-axis of targetTG by radians
{ targetTG.getTransform(t3d);
  toRot.rotY(radians);
  t3d.mul(toRot);
  targetTG.setTransform(t3d);
} // end of rotateY()
```

Detecting a Collision

After the operation has affected the grabbers' TransformGroups, it's time to see if a collision has occurred. isColliding() deals with five cases:

- the arms may be touching each other;
- the left or right arm might be touching an obstacle;
- the left or right arm might be touching the ground.

```
// globals
private Grabber leftGrabber, rightGrabber;
private Obstacles obs;


private boolean isColliding()
{
  BoundingSphere[] bs = rightGrabber.getArmBounds();
  if (leftGrabber.touches(bs))   // arms touching each other?
    return true;

  // check the right arm against the obstacles
  if (obs.intersects(bs))
```

```
    return true;

  // check the left arm against obstacles
  bs = leftGrabber.getArmBounds();
  if (obs.intersects(bs))
    return true;

  // are either arms touching the ground?
  if (leftGrabber.touchingGround())
    return true;

  if (rightGrabber.touchingGround())
    return true;

  return false;
}  // end of isColliding()
```

The Grabber methods, getArmBounds(), touches(), and touchingGround() are the same as in chapter 4 ??. The Obstacles method, intersects(), was described earlier.

Undoing an Operation

If a collision is detected then the operation (the GrabberOp object, gop) has to be reversed. This is easy to do in this application since the operation is either a translation or rotation.

There are three cases to consider: whether the original operation was applied to the base, or to one of the arms. A base operation is reversed inside Grabbers, while GrabberOp and the relevant Grabber instance deals with the reversal of an arm rotation.

```
private void undoOp(GrabberOp gop)
{
  if (gop.isPart(GrabberOp.BASE))  // is a base op
    undoBase(gop);
  else {
    GrabberOp revGOP = gop.reverse();  // reverse the rotation op
    if (revGOP.isPart(GrabberOp.LEFT_GRABBER))
      leftGrabber.rotate(revGOP);
    else  // must be right grabber
      rightGrabber.rotate(revGOP);
  }
}  // end of undoOp()
```

undoBase() checks the opVal state in the original GrabberOp, and then carries out the reverse translation or rotation.

```
private void undoBase(GrabberOp baseGOP)
{
  switch (baseGOP.getOp()) {
    case GrabberOp.NONE: break;     // do nothing

    case GrabberOp.BASE_FWD: doMove(BACK); break;
    case GrabberOp.BASE_BACK: doMove(FWD); break;
    case GrabberOp.BASE_LEFT: doMove(RIGHT); break;
    case GrabberOp.BASE_RIGHT: doMove(LEFT); break;
    case GrabberOp.BASE_ROT_LEFT: rotateY(-ROT_AMT); break;
    case GrabberOp.BASE_ROT_RIGHT: rotateY(ROT_AMT); break;
    case GrabberOp.BASE_UP: doMove(DOWN); break;
    case GrabberOp.BASE_DOWN: doMove(UP); break;

    default: System.out.println("Not a base grabber op"); break;
  }
} // end of undoBase()
```

All the arm operations are joint rotations, so reversing one is just a matter of reversing the rotation direction, positive to negative, and vice versa. This is done by GrabberOp.reverse() which generates a new GrabberOp instance; it's a copy of the original except for the rotation direction.

```
// in the GrabberOp class
public GrabberOp reverse()
{
  // copy the op, part, and joint values for this object
  GrabberOp gop = new GrabberOp();
  gop.setOp(opVal);
  gop.setPart(partVal);
  gop.setJoint(jointVal);

  // reverse the rotation (if possible)
  if (rotVal == NO_ROT) {
    System.out.println("Cannot reverse since no rotation found");
    gop.setRotation(rotVal);
  }
  else if (rotVal == POS)
    gop.setRotation(NEG);
  else   // must be NEG
    gop.setRotation(POS);

  return gop;
}  // end of reverse()
```

Once the reversed GrabberOp object has been generated, it can be processed by the Grabber instance in the same way as other operations.

Alerting the User

processOp() calls rumblePad() when it detects a collision, causing the game pad to rumble for a short time. This is somewhat trickier to implement correctly than it may at first seem.

The rumbler is turned on and off by two separate calls to GamePadController.setRumbler(), which means that Grabbers must wait for a short time between the calls to give the pad time to rumble.Having Grabbers go to sleep is unacceptable since it would prevent the grabbers from responding to other user input. The answer is to use a thread (as when playing a sound in Obstacle).

Another issue is how to deal with rapid multiple calls to setRumbler(), caused when the user holds down a key (or button) that triggers a collision.

For my game pad, calls from multiple threads make the rumbler perform intermittently, and sometimes not rumble at all. This can be avoided by employing another idea from the sound playing code in Obstacle: a boolean, isRumbling, prevents a new thread from starting until the existing one has finished.

```
// globals
private GamePadController gamePad;
private boolean isRumbling = false;


private void rumblePad()
/* Make the pad rumble for 0.5 secs, but
   only if it's not already rumbling. */
{
  if (!isRumbling) {
    Thread rumbler = new Thread() {
      public void run()
      {
        isRumbling = true;
        gamePad.setRumbler(true);
        try {
          sleep(500);   // wait for 0.5 secs
        }
        catch(InterruptedException ex) {}
        gamePad.setRumbler(false);
        isRumbling = false;
      }
    };
    rumbler.start();
  }
}  // end of rumblePad()
```

6. The Grabber Arms

The Grabber class is essentially the same one that's in Arms3D in chapter 4 ??. A grabber arm can turn left or right, up or down, and spin around it's long axis; it's two fingers can open and close. All these operations are implemented as joint rotations: there are three joints at the start of the arm for the x-, y-, and z- axes, and a joint for each finger.

There are three areas where the Grabber class in this chapter is different from the one in Arms3D:

1. *Arm Size*. Due to the proximity to the camera, the dimensions of the arm and fingers have been reduced, to roughly a quarter of the size of the originals.
2. *Input Data*. The rotate() method processes a GrabberOp input argument rather than a collection of integers. The code is changed to examine the GrabberOp values, but the same rotations are carried out.
3. *Collision Detection*. Grabber no longer needs to create a list of 'collision' joints, since Grabbers no longer creates wakeup criteria using those joints. However, the collision *checking* methods, getArmBounds(), touches(), and touchingGround() are unchanged because they utilize the GrabberBounds object, which is the same as in Arms3D.