Chapter 28.8 (N10). Navigating a 3D Scene by Waving Your Arm

The BandObjView3D application creates a simple Java 3D world – a checkboard floor, stormy background, a giant humanoid, and assorted ground cover, mostly borrowed from the ObjView3D example in chapter 7 ??. Figure 1 shows a view of the scene.

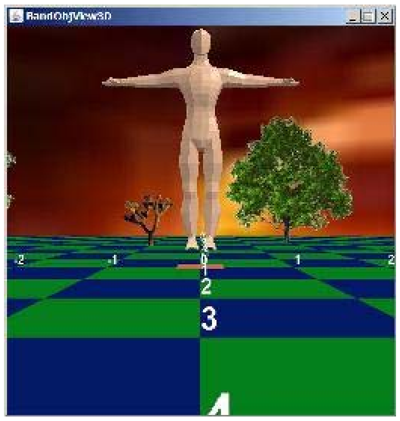

Figure 1. The BandObjView3D Scene.

The novelty is that navigation through the scene is achieved by the user moving and rotating their left arm; no keyboard or mouse manipulation is required.

This 'magic' is made possible by the user wearing a wrist strap containing three coloured bands (blue, yellow, and red), as shown in Figure 2.

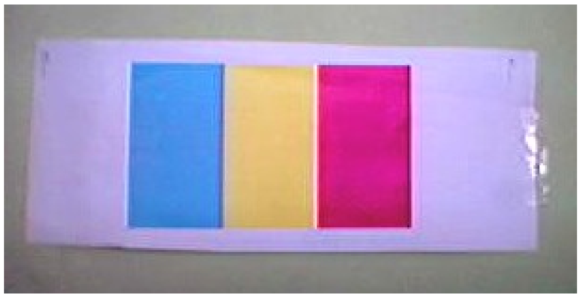

Figure 2. The Wrist Strap (blue, yellow, red).

Wrist strap is a rather fancy name for a sheet of paper with three coloured rectangles printed on it. It's taped around the user's wrist, as in Figure 3.

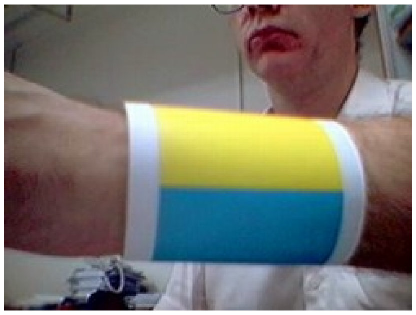
Figure 3. Wrist Strap in Place.

The blue band is the lower-half of the visible strap, yellow is at the top, and red is hidden on the other side of the user's arm.

A web camera is positioned close to the user on his left side, and takes pictures of the wrist strap (Figure 3 is such a snap). A new picture is processed every 0.1 second on average.

The largest visible band and its position in the image are converted into navigation commands inside the 3D scene. The yellow band triggers forward movement, the blue band turns the user's viewpoint to the right, and the red band to the left. The rotations match the way that the user's arm needs to turn so that the coloured band is in front of the camera.

If the center of the band in the current image is far enough to the right compared to the center of the band in the previous image, then forward motion is stopped. This corresponds to the user moving their arm backwards.

Figure 4 shows the user from a short distance away. The webcam is on the left of the monitor, partially hidden by the user's left arm.

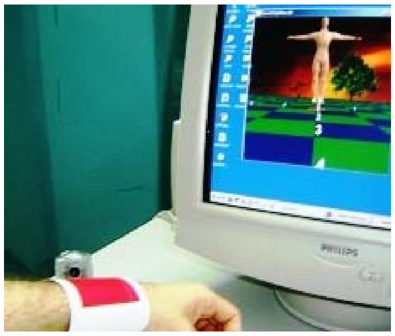
Figure 4. The Arm-Waving User.

1. Image Processing

The application utilizes a simple form of pattern recognition, which can be divided into three broad steps: segmentation, feature extraction, and classification, as illustrated by Figure 5.

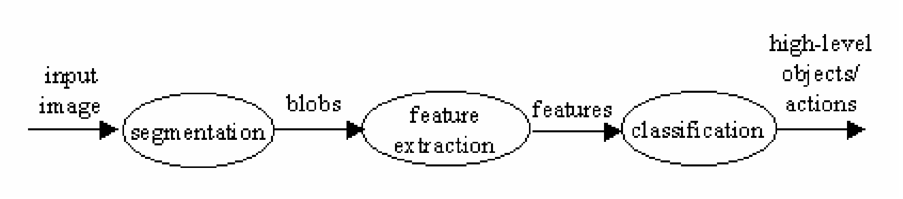

Figure 5. Pattern Matching Steps.

In general, segmentation involves finding interesting areas, regions, or *blobs* within the image. A membership function is applied to each pixel to allocate it to a particular blob.

Blob growing can be time-consuming, since it may require multiple passes over the blobs to combine/merge them into larger units. In the interests of processing speed, and coding simplicity, this application doesn't combine blobs together.

Feature extraction measures the blobs, which may include counting the number of member pixels, the colour spread, or the closeness to a prescribed shape. The resulting feature vectors are used to match the blobs with higher-level objects (e.g. the blob is really a car), or actions (e.g. a blob with those characteristics triggers an alarm).

The feature extraction and classification in this application is quite simple, based around converting the blob's center point and colour into translation and rotation commands in the 3D scene.

Image Processing Issues

Image processing is a complicated art, whose success greatly depends on the design of the imaging task. In our case, processing is simplified by positioning the camera just a few inches from the user's wrist, and only having to identify rectangular blobs of simple colours.

One complication is the background – the user's hand, arm, shirt, and head (e.g. as in Figure 3). For example, in early tests a red blob kept being detected even when the wrist strap's yellow band was right in front of the camera. This was triggered by the user's red *shirt* which occupied a comparatively large part of the image.

Another problem is the lighting environment. Overhead lights can cause 'leeching out' of colour, caused by excessive reflection. This makes it more difficult to find the colour bands.

Different webcams generate slightly different images. For instance, the camera attached to my second test machine produces a slightly greener image. This makes green a bad choice for a band colour, since so much of the background has a similar hue.

Dealing with the Issues

Most image processing applications support a configuration/testing stage so the hardware and software parameters can be tweaked; I've employ a similar strategy here.

I'll first describe the FindBands application, which analyses webcam images without affecting a Java 3D scene. FindBands is simpler than BandObjView3D since the Java 3D elements are absent, and it also generates more information about its processing. This data can be very useful when the hardware and software are being configured.

Once image analysis is working satisfactorily in FindBands, it's a simple step to copy the configuration settings over to BandObjView3D. The two applications utilize almost identical image processing classes.

2. Finding the Bands

The FindBands application carries out the same image processing as BandObjView3D, but reports more about it. Figure 6 shows the FindBands window:

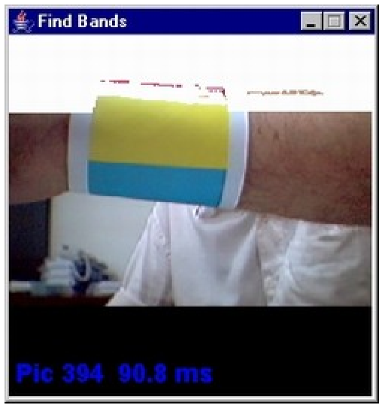

Figure 6. FindBands in Action.

FindBands doesn't use Java 3D; instead the images taken from the webcam are processed and displayed in rapid succession in the application window.

Each image will usually have a large block of white pixels at the top – white indicates that the original pixels were *not* added to any blobs. The pixels in the white block which retain their colour were added to blobs.

Figure 6 shows that blobs were created for parts of the red band in the wrist strap, a chunk of the yellow band, and parts of the user's left arm.

An important optimization in FindBands (and BandObjView3D) is that processing stops as soon as a large enough blob is created. The blob for the yellow band in Figure 6 grew to the necessary size (800 pixels) to stop the analysis, halting somewhere in the top part of the yellow band. The rest of the image wasn't examined, as indicated by the lack of white pixels.

This approach is obviously a little dangerous since it may mean that larger blobs, located further down the image, will be ignored. This problem can be alleviated to some degree by adjusting the large blob size constant; making the constant bigger means that more of the image will be examined.

The text in dark blue at the bottom of Figure 6 reads: "Pic 394 90.8 ms". The second number is the average processing time for the all the webcam images. In my tests, the average was always within the range 90-100 ms, including 30-40 ms for grabbing the image from the camera (as noted in chapter 9 ??). This indicates that the actual analysis only takes around 60ms. The first number in the text string is the total number of images displayed so far.

FindBands also prints translation and rotation command strings to standard output, such as:

```
    :
turn right
turn right
turn right
  forwards
  forwards
  forwards
No large blob
No large blob
  forwards
  forwards
  forwards
   :
```

Each line indicates how the large blob in the webcam image was classified (yellow as "forwards", blue is "turn right", and red is "turn left").

The "no large blob" string means that no suitably large blob was found in the image. This failure is also observable in the FindBands window, which will render the image almost completely white: no large blob means that processing will continue right to the end of the picture.

2.1. FindBands Class Diagrams

The class diagrams for the FindBands application are given in Figure 7; only the public methods are shown.

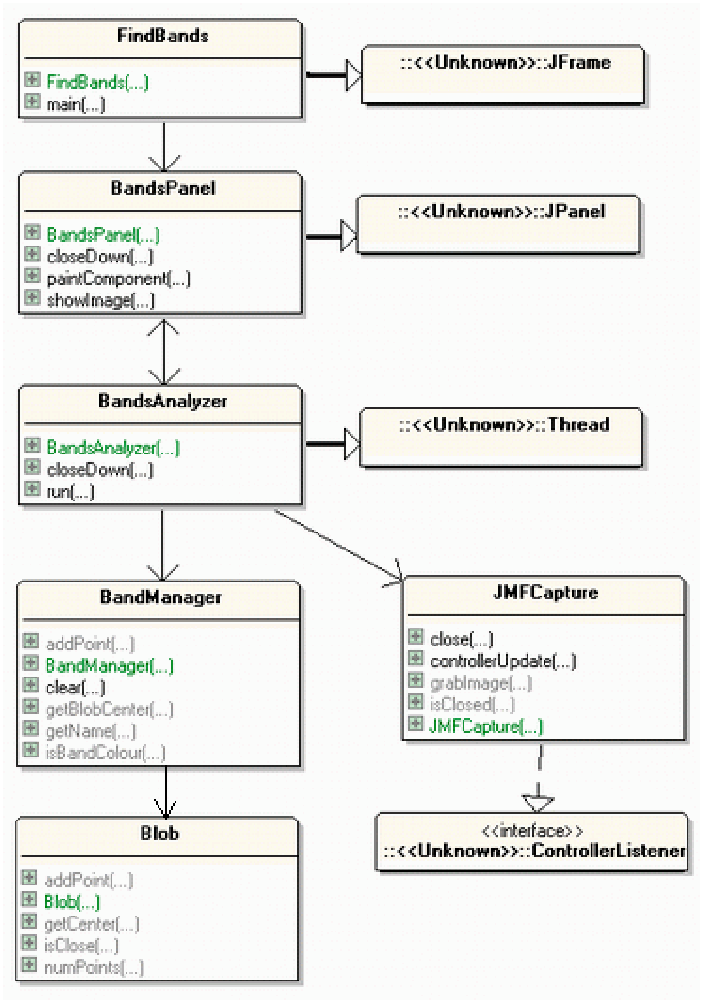

Figure 7. FindBands Class Diagrams

The FindBands and BandsPanel classes implement the application window and its drawing pane. Image processing is handled by BandsAnalyzer, which uses the JMFCapture class from chapter 9 ?? to take snaps (the class is unchanged from there, so I won't be discussing it again).

A BandManager object is created for each colour band in the wrist strap (three altogether, for red, yellow, and blue). Each BandManager stores a collection of Blob objects, one Blob object for each blob of colour found in the image.

2.2. Image Processing Overview

All the image processing tasks in FindBands are carried out by BandsAnalyzer, which repeatedly takes a snap, analyzes it, and displays the result in BandsPanel.

Figure 8 shows the main stages in the processing of a single image:

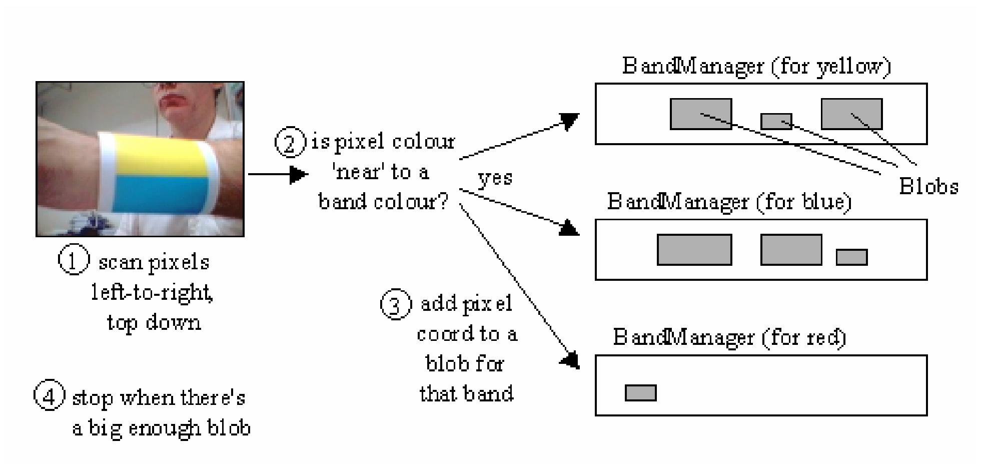
BandManager (for yellow)
1 scan pixels left-to-right, top down
2 is pixel colour 'near' to a band colour?
yes
Blobs
BandManager (for blue)
3 add pixel coord to a blob for that band
BandManager (for red)
4 stop when there's a big enough blob

Figure 8. Image Processing in FindBands.

The image is examined in a left-to-right, top-down manner, a pixel at a time. If the pixel's colour is close enough to one of the band colours, then the pixel's coordinates are passed to the relevant BandManager object for that band. The Blob objects already stored in the BandManager are examined to see if any of them are near to the pixel's coordinate, in which case the coordinate is added to the closest blob. If no neighborhood blob is found, then a new one is created, starting with the pixel coordinate.

Figure 8 corresponds to the "segmentation" stage in Figure 5. When a blob grows to a sufficiently large size, segmentation stops, and the blob is passed to the feature extraction stage. Only a single feature is calculated, the average center point for the blob. The center and the blob's colour are used to generate translation and rotation command strings during the classification stage.

2.3. The Bands Analyzer

The analysis begins by creating three BandManager objects for the three band colours:

```
// globals
private static final int NUM_BANDS = 3;   // no. of bands colours

private static final Color BAND1 = new Color(205, 16, 80);   // red
private static final Color BAND2 = new Color(200, 200, 85); //yellow
private static final Color BAND3 = new Color(50, 150, 200);  // blue

private BandManager bandMans[];    // band managers for the bands


// in the constructor...
bandMans = new BandManager[NUM_BANDS];

// initialize the band managers; there should be NUM_BANDS of them
bandMans[0] = new BandManager("red1", BAND1);
bandMans[1] = new BandManager("yellow", BAND2);
bandMans[2] = new BandManager("blue", BAND3);
```

An obvious question is how I decided on the values for the band colours: BAND1, BAND2, and BAND3. It was mostly a matter of trial and error, guided by a need for colours that were very different from each other, and from the image background.

Green was tried, and rejected, since one of my webcams makes the background look slightly greenish, which confuses the analysis. Red is also problematic, since my skin appears quite red in some webcam pictures. This seems to be due to the fluorescent lights in my office, because the problem doesn't arise when the images are generated in natural light.

The RGB values for each band colour come from examining a webcam snap (such as Figure 3) in a photo application (e.g. Adobe Photoshop, The Gimp). Several values were measured at different places in a band to calculate an average.

I examined a webcam image so that environmental factors, such as camera quality and lighting conditions, were factored into the numbers. The downside is that when FindBands is employed with a different camera, or on a different machine, then the colour constants usually need to be adjusted.

2.3.1. The Analysis Loop

BandAnalyzer performs its analysis in a loop in its run() method, processing a new webcam image in each iteration.

```
// globals
private static final int SNAP_INTERVAL = 120;
                            // ms; the time between snaps

private JMFCapture camera;  // JMF webcam object
private int imSize;         // size of captured image
private BandsPanel bandsPanel;
       // for displaying the snapped (and modified) images
private boolean running;
```

```
public void run()
{
  camera = new JMFCapture(imSize);   // initialize the webcam
  System.out.println("**Camera Ready**");

  long duration;
  BufferedImage im = null;
  running = true;

  while (running) {
    long startTime = System.currentTimeMillis();
    im = camera.grabImage();  // take a snap
    analyzeImage(im);         // analyze the image
    applyBigBlob();      // convert blob info to trans/rot cmds

    duration = System.currentTimeMillis() - startTime;

    if (im == null)
      System.out.println("Problem loading image");
    else
      bandsPanel.showImage(im, duration);  // show image in JPanel

    if (duration < SNAP_INTERVAL) {
      try {
        Thread.sleep(SNAP_INTERVAL-duration);
        // wait until interval has passed
      }
      catch (Exception ex) {}
    }
  }
  camera.close();    // close down the camera
}  // end of run()
```

The loop uses a JMFCapture object to take a picture every SNAP_INTERVAL ms. analyzeImage() implements the segmentation illustrated in Figure 8. applyBigBlob() uses the large blob (if it exists) to generate a translation or rotation command string.

The resulting image, which will have been modified by analyzeImage(), is displayed by bandsPanel (the JPanel) via a call to BandsPanel.showImage().

My value for SNAP_INTERVAL is based on the average processing time reported by FindBands (e.g. as seen in Figure 6). SNAP_INTERVAL is set at 120 ms to let the BandsAnalyzer thread sleep for a short period at the end of each iteration (about 20 ms); this gives Java some time to render the image to the screen.

2.3.2. Analyzing an Image

analyzeImage() implements the segmentation shown in Figure 8.

```
// constant
private static final int WHITE = 0xffffff;  // white pixel colour


private void analyzeImage(BufferedImage im)
{
  if (im == null) {
    System.out.println("Input image is null");
    return;
  }

  reset();   // reset the bands analyzer

  // extract pixels from the image into an array
  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  int [] pixels = new int[imWidth * imHeight];
  im.getRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);

  int xc = 0;    // store the current pixel coordinates
  int yc = 0;
  int bandId;    // id of band manager
  boolean madeBigBlob = false;

  // examine the pixels
  int i = 0;
  while((i < pixels.length) && (!madeBigBlob)) {
    bandId = isBandColour(pixels[i]);
    if (bandId != -1)   // pixel colour is a band colour
      madeBigBlob = addPointToBand(bandId, xc, yc);
    else      // pixel colour isn't a band colour
      pixels[i] = WHITE;   // clear the pixel to white

    // move to next coordinate in image
    xc++;
    if (xc == imWidth) {    // at end of row
      xc = 0;   // start next row
      yc++;
    }
    i++;
  }

  // update the image with the new pixels data
  im.setRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth);
}  // end of analyzeImage()
```

The image is manipulated as a pixel array, by calling Bufferedmage.getRGB(). This makes it very simple to iterate over the pixels, using (xc, yc) to store the coordinates of the current pixel.

Each pixel is compared to the band colours with isBandColour():

```
private int isBandColour(int pixel)
{
  // extract RGB components from the current pixel as integers
  int redVal = (pixel>>16)&255;
  int greenVal = (pixel>>8)&255;
  int blueVal = pixel&255;

  for(int i=0; i < NUM_BANDS; i++)
    if (bandMans[i].isBandColour(redVal, greenVal, blueVal))
      return i;
  return -1;
}  // end of isBandColour()
```

The bit manipulation at the start of isBandColour() relies on the pixel using the ARGB format. This format is specified when JMFCapture creates a snap using the BufferedImage.TYPE_INT_RGB constant.

Each pixel is a 32-bit word, with 8 bits for each colour component. The extracted RGB colours will have integer values between 0 and 255, with 255 meaning full-on.

The returned value from isBandColour() is the index of the relevant band manager in the bandMans[] array, or –1 if the pixel isn't close enough to any of the band colours.

Back in analyzeImage(), a pixel is added to a blob via the band manager for the matched colour. Otherwise the pixel is switched to white to indicate its uselessness:

```
  if (bandId != -1)   // pixel colour is a band colour
    madeBigBlob = addPointToBand(bandId, xc, yc);
  else      // pixel colour isn't a band colour
    pixels[i] = WHITE;   // clear the pixel to white
```

addPointToBand() asks the relevant BandManager to store the pixel in a blob:

```
// globals for band manager and blob indices
private int bandIdx, blobIdx;


private boolean addPointToBand(int bandId, int x, int y)
{
  boolean madeBigBlob = false;

  if ((bandId < 0) || (bandId >= NUM_BANDS))
    System.out.println("Band ID out of range: " + bandId);
  else {
    int blobId = bandMans[bandId].addPoint(x,y);
    if (blobId != -1) {   // made a large-enough blob
      madeBigBlob = true;
      bandIdx = bandId;   // store indices for large blob
      blobIdx = blobId;
    }
  }
  return madeBigBlob;
}  // end of addPointToBand()
```

If the resulting blob has become large enough to halt the processing then BandManager.addPoint() returns its index, and the indices of the band manager and blob are recorded in bandIdx and blobIdx. These indices allow the blob to be quickly accessed later by applyBigBlob().

2.3.3. Using the Large Blob

If analyzeImage() has found a large blob, then the two global indices, bandIdx and blobIdx, will have values. These are utilized in applyBigBlob() to extract the relevant band and blob features, and then to generate translation and rotation command strings.

```
// global
```

```
private Point prevCenterPt = null;
      // center of blob in previous image


private void applyBigBlob()
{
  if (bandIdx != -1) {  // there is a large blob
    BandManager bm = bandMans[bandIdx];
    Point pt = bm.getBlobCenter(blobIdx);   // blob's center pt

    if (prevCenterPt != null) {   // there is a previous center point
      rotateView( bm.getName() );
      translateView(pt, prevCenterPt);
    }
    prevCenterPt = pt;  // save for next analysis
  }
  else
    System.out.println("No large blob");
  }  // end of applyBigBlob()
```

The previous image's blob center (prevCenterPt) is compared with the current center in translateView() in order to determine a translation.

From Colour to Rotation

The colour name is mapped to a left (or right) rotation string. In BandObjView3D, this method will call navigation code in the Java 3D scene, but here it just prints a string.

```
private void rotateView(String bmName)
{
  if (bmName.equals("red1"))
    System.out.println("turn left");
  else if (bmName.equals("blue"))
    System.out.println("turn right");
}
```

From Points to Translation

The x-axis difference between the current center point and the one for the previous image is used to generate a translation string.

```
// globals
// min. distance for blob center to move to trigger a translation
private static final int MIN_MOVE_DIST = 10;

private boolean movingFwd;


private void translateView(Point currPt, Point prevPt)
{
  int xChg = currPt.x - prevPt.x;

  if (xChg < -MIN_MOVE_DIST)
    movingFwd = true;
  else if (xChg > (MIN_MOVE_DIST*2))
    // stopping requires more of a change
    movingFwd = false;

  if (movingFwd)
    System.out.println("  forwards");
}  // end of translateView()
```

When the user starts travelling forward in the 3D world, he will keep going until he requests a stop. This is implemented by setting a movingFwd boolean to true when the wrist strap is moved forward. It's only set to false when the strap is moved backwards by a large amount. The value for MIN_MOVE_DIST was arrived at by a process of trial-and-error, and will need to be adjusted depending on the distance of the camera from the user's arm.

In this code, the movingFwd boolean only triggers a System.out.println() call, but in BandsObjView3D it will translate the user's viewing position.

One tricky aspect of the mapping is the assumption that the movement of a blob to the left in the image means that the user has moved their arm forwards. This is not the case for some webcams which flip their images; in that case, a leftwards movement occurs when the user moves their arm backwards! This is easily corrected by switching the if-tests in translateView().

I don't use the y-coordinates of the center points, partly because I don't need to, and also because deliberate vertical change is hard to differentiate from random arm movement.

2.4. The Band Manager

A BandManager object manages all the blobs created for a given band colour. It offers isBandColour() to check if a pixel is close to the band manager's colour. addPoint() adds a pixel coordinate to a blob, and reports if a large enough blob has been made. BandManager also has methods to return the band colour and a given blob's center point.

BandManager is initialized with a name and the colour it represents, and creates an empty ArrayList for future Blob objects:

```
// globals
private String bandName;

// RGB components for this band manager's colour
private int redCol, greenCol, blueCol;

private ArrayList<Blob> blobs;   // of Blob objects
private int currBlobIdx;  // index of last blob that was updated


public BandManager(String nm, Color c)
{
  bandName = nm;

  redCol = c.getRed();
  greenCol = c.getGreen();
  blueCol = c.getBlue();
  blobs = new ArrayList<Blob>();
  currBlobIdx = -1;
}  // end of BandManager()
```

isBandColour() calculates the Euclidean distance between the pixel's colour and the band colour, and tests if it is less than a certain amount.

```
// max distance*distance to the band colour
private static final int MAX_DIST2 = 5000;


public boolean isBandColour(int r, int g, int b)
/* is (r,g,b) close enough to the band colour? */
{
  int redDiff = r - redCol;
  int greenDiff = g - greenCol;
  int blueDiff = b - blueCol;

  return (((redDiff * redDiff) +
           (greenDiff * greenDiff) +
           (blueDiff * blueDiff)) < MAX_DIST2);
}  // end of isBlobColour()
```

isBandColour() actually calculates the *square* of the distance between the colours, skipping the expensive square root operation.

The MAX_DIST2 value was decided on by experimentation: the square root of MAX_DIST2 is around 71, or about 24 for each colour component (a component's value can range between 0 and 255).

2.4.1. Adding a Pixel

Adding a pixel to a blob involves first cycling though the existing blobs to check if the pixel is close to one of them. findCloseBlob() returns the index of the blob, or –1 if nothing suitable was found. The index is employed in a Blob.addPoint() call to add the pixel's coordinates to the right blob.

```
// globals
private int currBlobIdx;  // index of last blob that was updated

public int addPoint(int x, int y)
{
  int largeIdx = -1;   // index of Blob with enough points

  int blobIndex = findCloseBlob(x,y);
  if (blobIndex != -1) {   // found a blob close to (x,y)
    Blob b = blobs.get(blobIndex);
    boolean isLarge = b.addPoint(x,y);
    currBlobIdx = blobIndex;
    if (isLarge)   // created a large enough blob
      largeIdx = blobIndex;
  }
  else {   // no close blob, so create a new one
    Blob b = new Blob();
    b.addPoint(x,y);
    blobs.add(b);
    currBlobIdx = blobs.size() - 1;
  }
  return largeIdx;
}  // end of addPoint()
```

currBlobIdx is used to record the ArrayList index position of the blob that was just updated.

If the blob has grown sufficiently large, then addPoint() returns its index, causing BandAnalyzer to cut short any further image processing.

findCloseBlob() first checks if the blob that was previously updated can be updated again, before it falls back to cycling through the blobs. The hope is that successive pixels in an image often end up joining the same blob.

```
private int findCloseBlob(int x, int y)
/* find a blob that's close to (x,y) */
{
  Blob blob;

  // try current blob first
  if (currBlobIdx != -1) {
    blob = blobs.get(currBlobIdx);
    if (blob.isClose(x,y))
      return currBlobIdx;
  }

  // otherwise try the others
  for(int i=0; i < blobs.size(); i++) {
    if (i != currBlobIdx) {
      blob = blobs.get(i);
      if (blob.isClose(x,y))
        return i;
    }
  }
  return -1;   // didn't find a close blob
}  // end of findCloseBlob()
```

2.5. Representing a Blob

A blob is a collection of pixel coordinates which are close to each other in the image, and have a similar colour.

A blob facilitates feature extraction by collecting related image data together in one place. FindBands only utilizes a single blob 'feature', the blob's center, which is calculated by dividing the sums of the blob's x- and y- coordinates by the number of those coordinates.

```
// globals
private ArrayList<Point> points;   // of Point objects
private int numPoints;
private int xSum, ySum;   // sums of (x,y) coords
```

```
public Blob()
{  points = new ArrayList<Points>();
   numPoints = 0;
   xSum = 0; ySum = 0;
} // end of Blob()
```

A pixel's proximity to a blob is determined by isClose():

```
// constant
private static final int PROXIMITY = 4;


public boolean isClose(int x, int y)
/* is (x,y) close to a point in the blob? */
{
  Point p;
  for(int i=0; i < numPoints; i++) {
    p = points.get(i);
    if ((Math.abs(x - p.x) < PROXIMITY) &&
        (Math.abs(y - p.y) < PROXIMITY))
      return true;
  }
  return false;
}  // end of isClose()
```

PROXIMITY is another 'magic' number, decided on by testing. The larger the value, the easier it is for a point to join the blob.

Adding a pixel coordinate to the blob involves adding a Point object to the blob's ArrayList. addPoint() also checks if the blob has enough points to be classified as large.

```
// no. of points necessary to make a large blob
private static final int LARGE_BLOB_SIZE = 800;


public boolean addPoint(int x, int y)
/* add (x,y) to the blob's points, and report if the
   blob is now 'large' */
{
  points.add( new Point(x,y) );
  numPoints++;
  xSum += x; ySum += y;
  return (numPoints > LARGE_BLOB_SIZE);
}  // end of addPoint()
```

Increasing the LARGE_BLOB_SIZE value will lengthen the image processing time, since more points will need to be added to the blobs before one grows sufficiently large. A larger LARGE_BLOB_SIZE makes it less certain that a blob will attain the necessary size, increasing the chance of the image classification failing. However, if a blob is found, then it's more likely to be the largest one in the image.

3. Arm Navigation

The main purpose of FindBands is to help us test the image processing part of the code without Java 3D complicating things. FindBands reports details about the analysis that can be used to fine-tune the hardware, environment, and software. When everything is working satisfactorily, the move to BandObjView3D is relatively simple.

BandObjView3D is a combination of the ObjView3D example from chapter 7 ??, and FindBands. This is evident in its class diagrams, shown in Figure 9. To reduce clutter, I've left out private methods and data, and superclasses and interfaces.

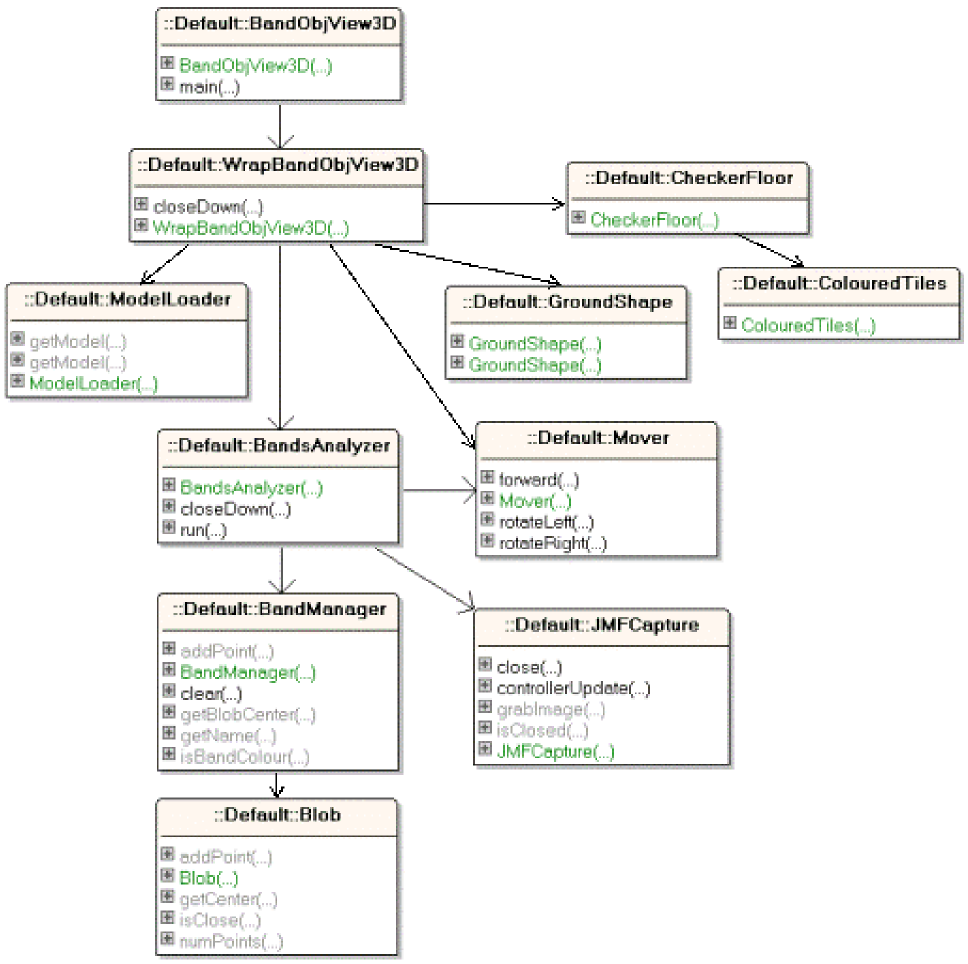

Figure 9. Class Diagrams for BandObjView3D.

BandObjView3D is the application JFrame, and does little more than create the WrapBandObjView3D JPanel for rendering the 3D scene.

WrapBandObjView3D is similar to WrapObjView3D in ObjView3D – it creates a checkboard floor, with a red center square, and labelled XZ axes, using the CheckerFloor and ColouredTiles classes.

A humanoid model and ground shapes are standing on the floor, the model loaded by the ModelLoader class, the ground shapes with GroundShape. The lighting and background are also done by WrapBandObjView3D.

WrapBandObjView3D employs a new Mover class to affect the user's viewpoint; in WrapObjView3D, movement was achieved with my KeyBehavior class.

BandsAnalyzer performs almost the same image analysis as in FindBands, using BandManager, Blob, and JMFCapture unchanged. BandsAnalyzer differs in that it calls translation and rotation methods in Mover rather than rendering images to a JPanel and printing command strings to standard output.

I won't discuss BandManager, Blob, and JMFCapture again, since they're the same as before. I'll also skip most of the 3D scene creation code in WrapBandObjView3D, since it's been borrowed without modification from ObjView3D in chapter 7 ??.

I'll concentrate on explaining two features: how Mover affects the user's 3D viewpoint, and how BandsAnalyzer communicates with Mover.

3.1. Creating the 3D Scene

WrapBandObjView3D contains the following lines in its constructor:

```
// global
private BandsAnalyzer ba;   // to analyze the user's arm movements


// in the constructor...
createSceneGraph();
Mover mover = createMover();
su.addBranchGraph(sceneBG);

ba = new BandsAnalyzer(mover);
ba.start();   // start responding to user arm movements
```

createSceneGraph() initializes most parts of the 3D scene, including its lighting, sky, floor, and the floating sphere.

createMover() creates a Mover object:

```
private Mover createMover()
{
  ViewingPlatform vp = su.getViewingPlatform();
  TransformGroup targetTG = vp.getViewPlatformTransform();
         // viewer transform group

  return new Mover(targetTG);
     /* Mover translates/rotates the user's viewpoint
        depending on the user's arm movements */
}  // end of createMover()
```

The targetTG reference passed to Mover is the user's viewing platform TransformGroup (TG), a node in the view branch part of the application's scene graph (see Figure 10). targetTG controls the movement of the user's viewpoint around the 3D scene.

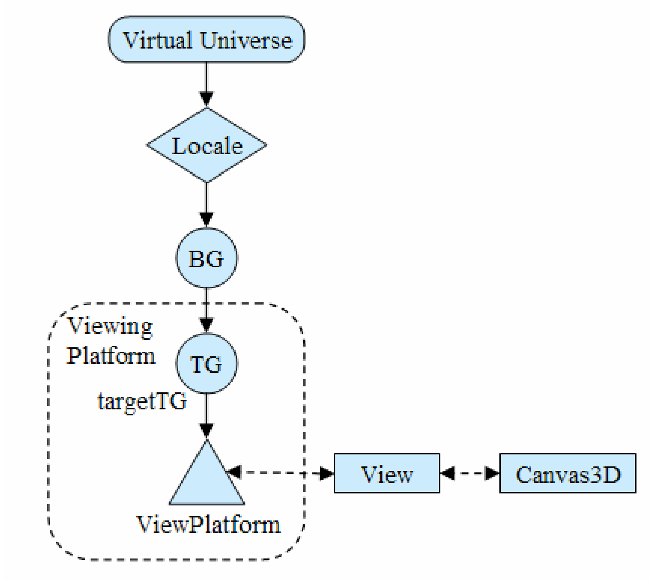

Figure 10. The View Branch Graph.

KeyBehavior in the original ObjView3D application used targetTG as well, but gained access to it by being a subclass of Java 3D's ViewPlatformBehavior.

Back in WrapBandObjView3D's constructor, the Mover reference is passed to BandsAnalyzer. This allows BandsAnalyzer to call Mover's translation and rotation methods, which affect the targetTG node.

3.2. Moving the User's Viewpoint

Mover only permits the viewpoint to move forward, and to rotate left or right. The meaning of forward depends on the viewpoint's current orientation.

The constructor specifies a starting position for the viewpoint by calling doMove():

```
// global
private TransformGroup targetTG;  // the viewpoint TG

public Mover(TransformGroup vTG)
{ targetTG = vTG;
  doMove( new Vector3d(0, 0.5, 5.0) );   // starting position
}
```

doMove() multiplies the translation vector to the Transform3D component of the viewpoint. This 'adds' the translation to the viewpoint position.

```
// globals for repeated calculations
private Transform3D t3d = new Transform3D();
private Transform3D toMove = new Transform3D();
private void doMove(Vector3d theMove)
{ targetTG.getTransform(t3d);
  toMove.setTranslation(theMove);
  t3d.mul(toMove);
  targetTG.setTransform(t3d);
} // end of doMove()
```

The initial, default position of the user's viewpoint is at (0,0,0), with forward being along the negative z-axis. The call to doMove() moves the viewpoint to (0, 0.5, 5), and keeps it facing along the –z-axis.

Translating and Rotating

The public face of moving forwards, is the forward() method, which is called by BandsAnalyzer:

```
// forward movement constants
private static final double MOVE_STEP = 0.2;
private static final Vector3d FWD = new Vector3d(0,0,-MOVE_STEP);

public void forward()
{  doMove(FWD);  }
```

The translation is along the negative z-axis, which always means forward, even after the viewpoint has been rotated.

The public face of rotation is represented by two methods:

```
private static final double ROT_AMT = Math.PI / 36.0;   // 5 degrees

public void rotateLeft()
{  rotateY(ROT_AMT);  }

public void rotateRight()
{  rotateY(-ROT_AMT);  }
```

rotateY() multiplies the rotation (in radians) to the Transform3D component of the viewpoint TransformGroup, which effectively 'adds' the rotation to the old orientation.

```
// global for repeated calculations
private Transform3D toRot = new Transform3D();

private void rotateY(double radians)
{ targetTG.getTransform(t3d);
  toRot.rotY(radians);
  t3d.mul(toRot);
  targetTG.setTransform(t3d);
}
```

3.3. From Analysis to Action

The BandsAnalyzer class is largely unchanged from its FindBands version, since the image processing involving BandManager, Blob, and JMFCapture is the same. However, rotateView() and translateView() are different:

```
// global
private Mover mover;     // for moving the 3D viewpoint (camera)


private void rotateView(String bmName)
// blob colour --> rotation
{
  if (bmName.equals("red1"))
    mover.rotateLeft();
  else if (bmName.equals("blue"))
    mover.rotateRight();
}

private void translateView(Point currPt, Point prevPt)
/* The change in the x values of the current center point
   (currPt) and the previous center point (prevPt) triggers
   a viewpoint translation forward.
*/
{ int xChg = currPt.x - prevPt.x;

  if (xChg < -MIN_MOVE_DIST)
    movingFwd = true;
  else if (xChg > (MIN_MOVE_DIST*2))
    movingFwd = false;

  if (movingFwd)
    mover.forward();
}  // end of translateView()
```

The changes involve replacing the System.out.println() calls in the FindBands versions (described in section 2.3.3) by calls to Mover.

4. Other Approaches

Another example of this approach can be found in the online chapter 28.85, "Waving a Magic Wand", at my *Killer Game Programming in Java* website (http://fivedots.coe.psu.ac.th/~ad/jg). The user waves a 'magic wand' in front of the webcam, and an on-screen wand moves in a corresponding manner, and emits magic bolts as a added bonus. The image processing employs a variant of the BandsAnalyzer class, and captures images using JMFCapture.

I've implemented my own, very simple, image processing operations in these examples, based around region (blob) building. The code uses basic Java 2D features for accessing the internals of a BufferedImage (e.g. BufferedImage.getRGB()). More serious programming probably calls for more advanced APIs, focussing specifically on image analysis.

JAI. Java Advanced Imaging (JAI) offers extended image processing capabilities beyond those found in Java 2D (http://java.sun.com/products/java-media/jai/). For example, geometric operations include translation, rotation, scaling, shearing, and transposition and warping. Pixel-based operations utilize lookup tables and rescaling equations, but can be applied to multiple sources, then combined to get a single outcome. Modifications can be restricted to regions in the source, statistical operations are available (e.g. mean and median), and frequency domains can be employed.

Sun offers an online book on JAI (dating from 1999) at http://java.sun.com/products/java-media/jai/forDevelopers/jai1_0_1guide-unc/JAITOC.fm.html. Image analysis is covered in chapter 9.

ImageJ. For complex image processing, I like ImageJ (http://rsb.info.nih.gov/ij/), which can be utilized either as a standalone application, or as an image processing library.

ImageJ supports standard image processing functions, such as contrast manipulation, sharpening, smoothing, edge detection, and median filtering. It also offers geometric transformations, such as scaling, rotation, and flips.

Its analysis capabilities include the calculation of area and pixel value statistics for user-defined selections, and it can create density histograms and line profile plots.

ImageJ can be easily extended with macros and plugins; there are hundreds available at the ImageJ website.

Books. Two books on image processing, using Java as an implementation language:

- *Digital Image Processing: A Practical Introduction Using Java*
  Nick Efford, Addison Wesley, 2000
  http://www.comp.leeds.ac.uk/nde/book/
- *Image Processing in Java*
  Douglas A. Lyon, Prentice Hall, 1999
  http://www.docjava.com/

Both texts use the standard Java libraries (e.g. Java 2D); they don't utilize JAI or ImageJ.