NUI Chapter 6.5. Topcodes and the Robot Arm

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/jg/??; only important fragments are described here.]

Chapter 6 ended with the robot arm being able to pick up and deliver items by utilizing 3D coordinate locations. This chapter adds in a webcam and the topcodes vision library (http://users.eecs.northwestern.edu/~mhorn/topcodes/) to enable things (and places) to be referred to by tag IDs. For example, the user can request that the arm moves the object with topcode ID 205 to the location marked by topcode ID 237.

The robot's working area is shown in Figure 1. The arm is positioned at the edge of the graph paper, with its base at (0, 0), facing along the positive y-axis, with the –x axis to its left, +x to the right.

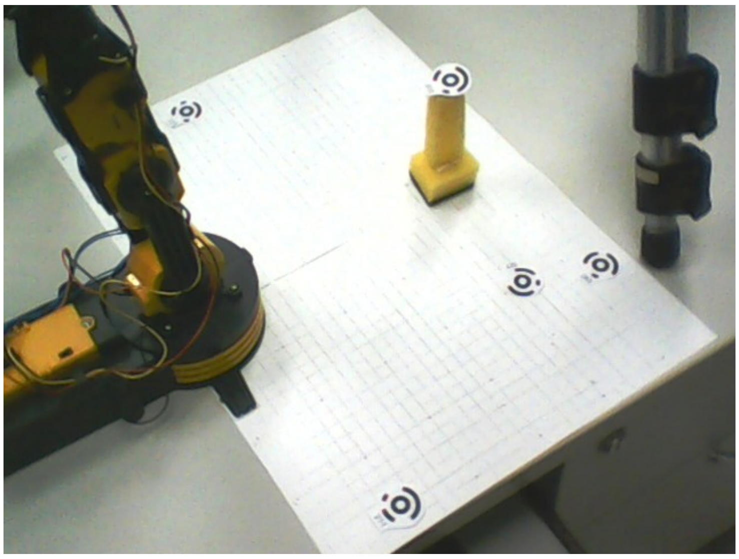

Figure 1. The Robot Arm Grid.

Figure 1 includes five topcodes – they're the circular black and white pieces of paper, one of which is stuck on the top of the sponge. Ninety nine different topcode graphics can be downloaded from http://users.eecs.northwestern.edu/~mhorn/topcodes/; Figure 2 shows a few of them.

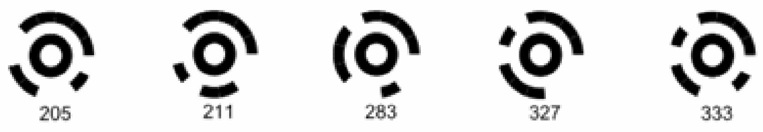

Figure 2. Some Topcodes.

The ID numbers below the codes don't need to be included when the topcodes are printed, the topcodes vision library identifies a symbol by its pattern of concentric circles.

The webcam is attached to a camera tripod, aimed straight down over the grid, as shown in Figure 3.

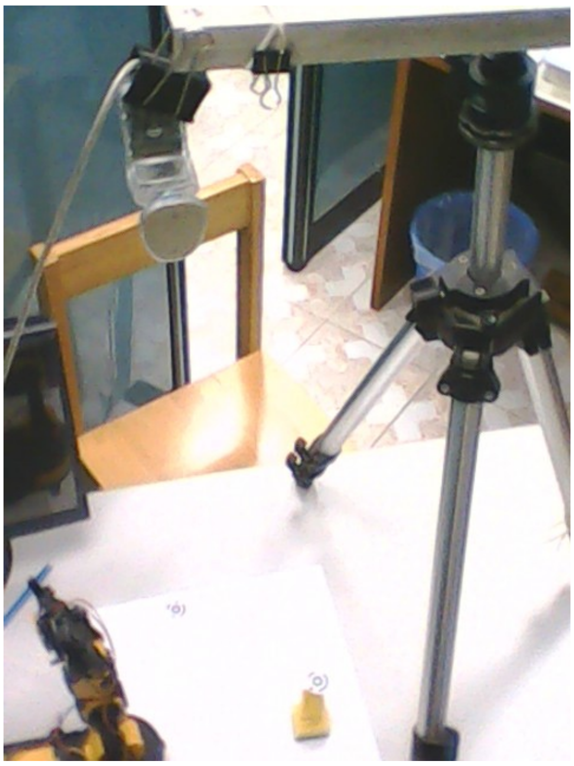

Figure 3. The Webcam and Tripod.

The webcam is positioned at right angles to the grid surface, a requirement for topcodes recognition. Figure 4 shows the webcam's view of the scene.

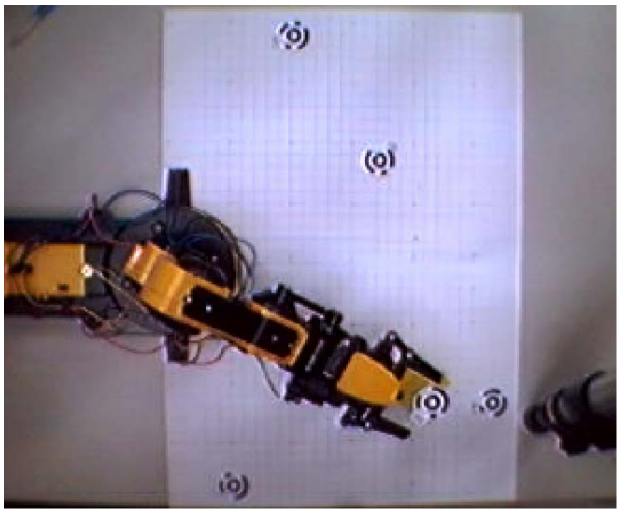

Figure 4. The Webcam View.

The topcodes library can recognize codes as small as 25 x 25 pixels in an image, under a variety of lighting conditions, without the need for camera calibration. For each detected code, the library returns its ID number, its image location, its angular orientation, and diameter.

Central to this chapter's application is the use of *two* coordinate systems, and the mapping from one to the other. One coordinate space is relative to the webcam image, the other to the grid, as illustrated by Figure 5.

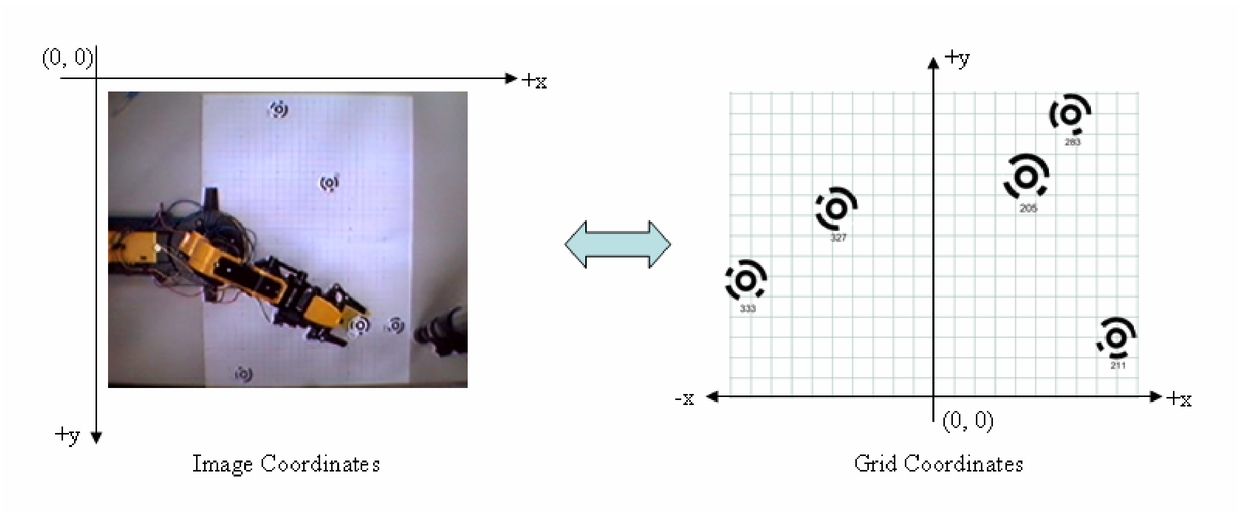

Figure 5. The Two Coordinate Systems.

Image coordinates for the topcodes are obtained by analyzing the webcam's picture, but the robot arm needs to be given *grid* coordinates in order to move correctly.

The mapping between the coordinate spaces can be derived from the relative positions of the topcodes. To align the codes in the two spaces, the image can be rotated counter-clockwise by 90 degrees, and enlarged.

My application utilizes topcodes in two different ways – as *landmarks* and as *trackers*. The landmark codes have fixed positions on the grid, which allow an image-to-grid mapping to be calculated quite easily. The tracker topcodes are moveable, and so are employed to label the sponge (see Figure 1) and arm destinations.

The landmarks in Figure 5 are the three topcodes around the edge of the grid, while the two trackers are towards the center.

1. The Topcodes Robot Arm Application

The classes used by this chapter's application (called TCRobotArm) are shown in Figure 6.

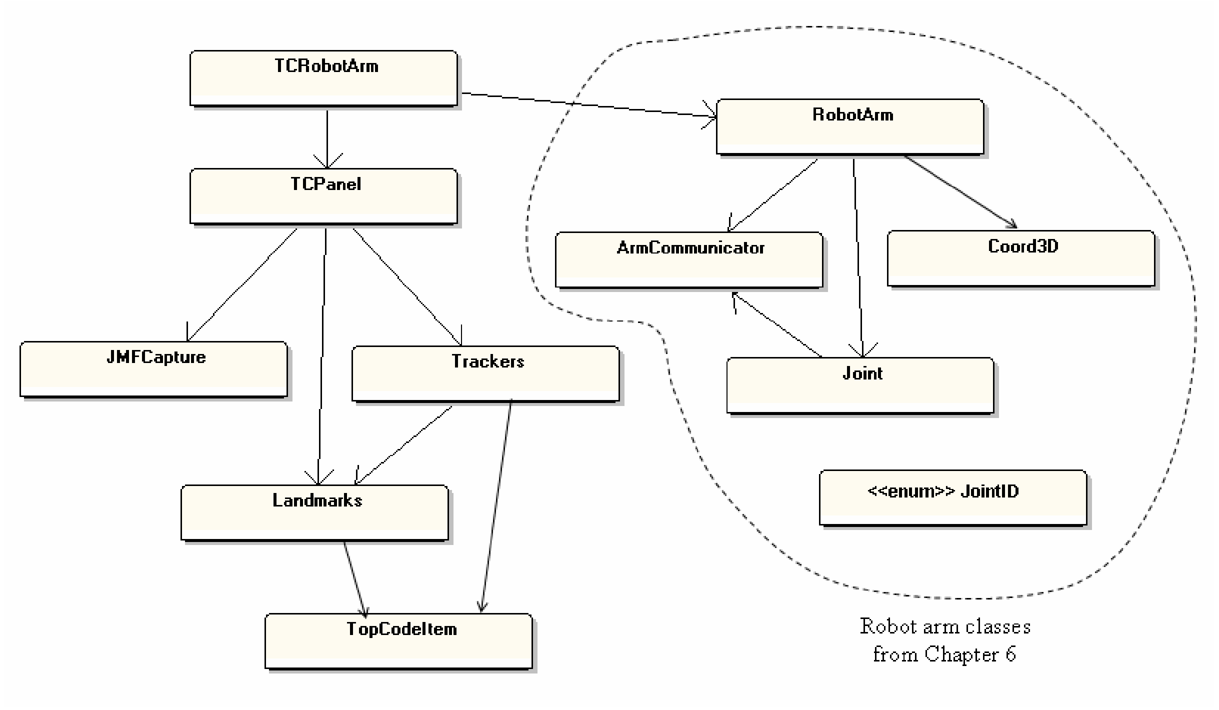

Figure 6. The Class Diagrams for the TCRobotArm Application.

The number of classes is a bit daunting, but we've seen most of them before. The classes that control the movement of the robot arm (surrounded by a dotted line in Figure 6) are unchanged from the previous chapter. I'll also be reusing the JMFCapture class to grab images from the webcam. TCRobotArm and TCPanel are subclasses of JFrame and JPanel, fairly similar to earlier GUI examples.

Each topcode is represented by a TopCodeItem object, with the collection of landmark codes stored in the Landmarks class, and the trackers in Trackers.

The top-level TCRobotArm GUI is shown in Figure 7.

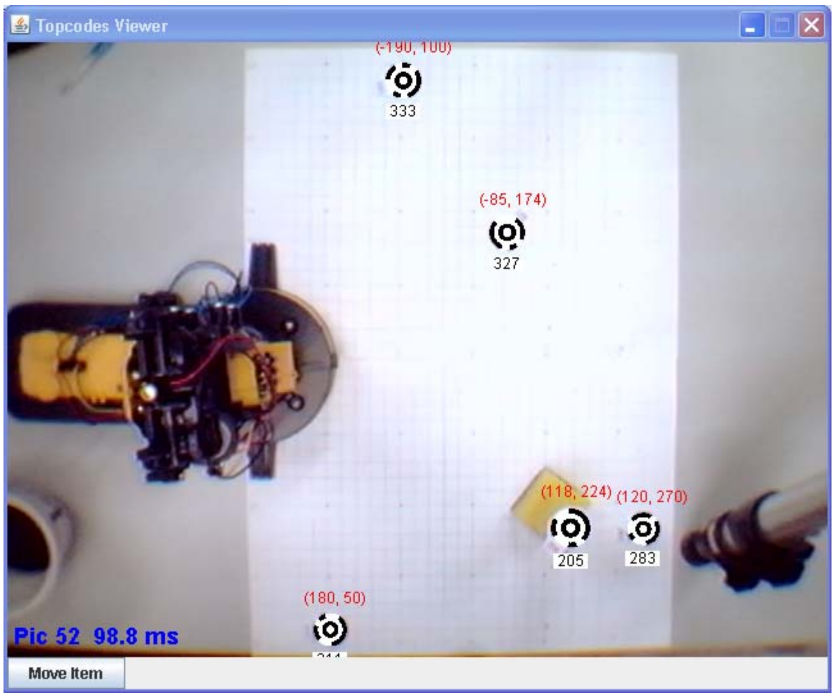

Figure 7. The TCRobotArm GUI.

The majority of the window is taken up with TCPanel's rendering of the current webcam image, but there's also a "Move Item" button at the bottom left. When the user presses it, the arm moves to wherever the sponge is located, picks it up, and carries it to the destination grid spot.

Crucially, the user doesn't have to enter coordinates for the sponge or destination; they're found by the application looking for tracker topcodes in the image, mapping them to grid positions, and passing those to the arm software.

Hardwired into my application is that the sponge is labeled by topcode ID 205, and the destination is topcode 327. Both are trackers (they may be located anywhere on the grid), but the other topcodes in Figure 7 (codes 333, 283, and 211) are landmarks with fixed positions.

The webcam image drawn by TCPanel highlights the identified topcodes, drawing black and white graphics for each, their IDs, and their grid positions.

After the sponge has been moved to the destination spot, the arm returns to its initial position, as in Figure 8.

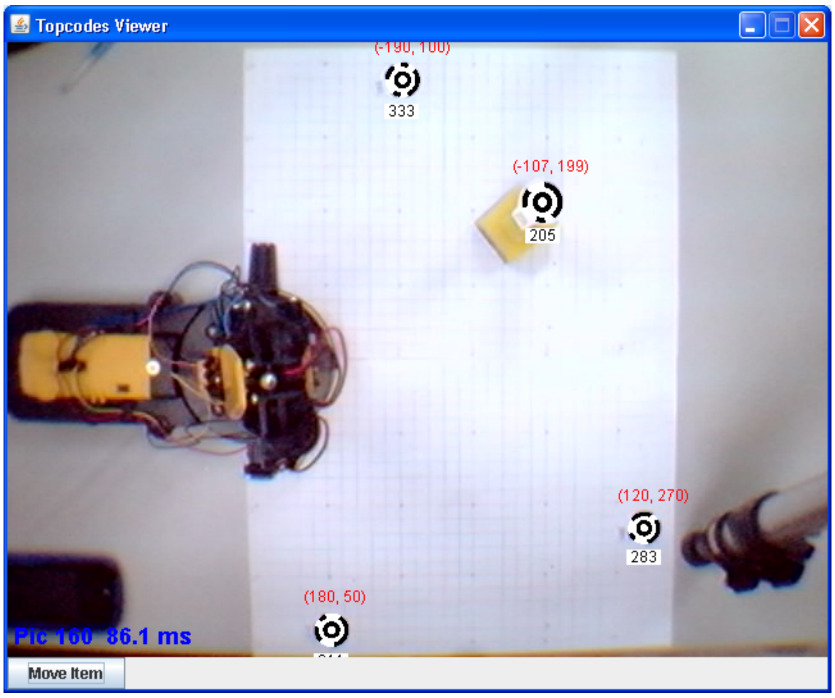

Figure 8. The TCRobotArm after Moving the Sponge.

Figure 8 shows that the sponge is now 'approximately' on top of the destination spot, the inaccuracy being due to the vagaries of the arm's gear wheels. Similarly, when the arm returns to being upright, it never manages to reach its original starting position.

2. Threaded Arm Movement

The most important coding design in the top-level TCRobotArm class is the use of a separate thread to execute arm movements. Since the arm takes several seconds to pick up and deliver an item, it would be a major mistake to execute it directly from the GUI thread. The GUI would 'lock up' (i.e. the webcam image would not be redrawn) until the arm had finished its task.

The ActionListener for the "Move Item" button calls TCRobotArm.moveItem():

```
// globals
/* tracker topcodes IDs used by the robot arm: the arm will move
   the SPONGE_TOPCODE to the location of the DEST_TOPCODE */
private static final int SPONGE_TOPCODE = 205;
       // ID of the topcode stuck to the top of the sponge

private static final int DEST_TOPCODE = 327;
      // ID of the topcode located somewhere on the grid


private RobotArm robotArm;
private ExecutorService robotExecutor;
         // for executing the robot arm code in a separate thread

private JButton moveBut;     // the "Move Item" button
private void moveItem()
/* Move the robot as a separate thread task;
   called by the "Move Item" button's ActionListener */
{
  robotExecutor.execute( new Runnable() {
    public void run()
    {
      moveBut.setEnabled(false);  // disable button during move

      // convert topcode IDs into grid coordinates
      Coord3D fromPt = getCoord(SPONGE_TOPCODE);
      Coord3D toPt = getCoord(DEST_TOPCODE);

      if ((fromPt != null) && (toPt != null)) {
        robotArm.moveItem(fromPt, toPt);
        System.out.println("Coord: " + robotArm.getCoord() );
        robotArm.showAngles();
        robotArm.moveToZero();
      }

      moveBut.setEnabled(true);
    } // end of run()
  });
}  // end of moveItem()
```

The time-consuming operations are the calls to RobotArm.moveItem() and RobotArm. moveToZero(), so the easiest thing is to place all the code in the hands of an ExecutorService which runs it in a separate thread. moveItem() immediately returns, letting the GUI thread continue. However, I disable the "Move Item" button so the user can't press it again until the robot has finished the curent move.

The topcode IDs for the sponge and destination spot are stored as constants, but their current coordinates on the grid are determined at runtime by calling getCoord().

```
// globals
private static final int SPONGE_HEIGHT = 65;   // in mm

private TCPanel topCodesPanel;


private Coord3D getCoord(int trackerID)
{
  TopCodeItem item = topCodesPanel.findTracker(trackerID);
  if (item == null) {
    System.out.println("Item " + trackerID + " is unknown");
    return null;
  }
  else if (!item.isLocatedOnGrid()) {
    System.out.println("Item  " + trackerID + " not on grid");
    return null;
  }
  // get the tracker's grid position
  int xCoord = item.getXGrid();
  int yCoord = item.getYGrid();
  return new Coord3D(xCoord, yCoord, SPONGE_HEIGHT);
}  // end of getCoord()
```

getCoord() uses the supplied ID to access the relevant TopCodeItem in the Trackers object maintained by TCPanel. The TopCodeItem is employed to check if the topcode is on the grid, and to retrieve its current position.

I'll explain the details of the TopCodeItem class later in this chapter.

3. The Rendering Panel

TCPanel contains the usual run() method for displaying the current webcam image. In addition, it extracts a list of topcodes from the image, and renders information about them on top of the picture (as shown in Figure 8).

```
// globals
private BufferedImage image = null;  // current webcam snap
private JMFCapture camera;
private volatile boolean isRunning;

private topcodes.Scanner scanner;
private java.util.List<TopCode> topCodes = null;


public void run()
{ initDisplay();

  BufferedImage im;
  long duration;
  isRunning = true;
  while (isRunning) {
    long startTime = System.currentTimeMillis();

    im = camera.getImage();  // take a snap
    if (im == null) {
      System.out.println("Problem loading image " + (imageCount+1));
      duration = System.currentTimeMillis() - startTime;
    }
    else {
      image = im;   // only update image if it contains something
      imageCount++;

      topCodes = scanner.scan( enhanceImage(image) );
                               // find topcodes in the image
      locateItems(topCodes);   // use topcodes to locate items

      duration = System.currentTimeMillis() - startTime;
      totalTime += duration;
      repaint();
    }

    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);
      }
      catch (Exception ex) {}
    }
  }

  camera.close();    // close down the camera
}  // end of run()
```

The main topcodes library method is Scanner.scan() which is passed an image and returns a list of TopCode objects found in that image.

In tests, the image supplied by my old webcam was often too blurry for the scan() method to find all the topcodes. enhanceImage() employs several OpenCV techniques to improve the results: image sharpening, smoothing for noise reduction, and adaptive thresholding, applied to a grayscale version of the picture.

```
// global JavaCV variables
private CvMat lapKernel;  // Laplace kernel for image sharpening
private IplImage grayImg;


private BufferedImage enhanceImage(BufferedImage im)
{
  IplImage img = IplImage.createFrom(im);
  cvCvtColor(img, grayImg, CV_BGR2GRAY);   // convert to grayscale

  cvSmooth(grayImg, grayImg, CV_GAUSSIAN, 7, 7, 0, 0);

  cvAdaptiveThreshold(grayImg, grayImg, 255,
          CV_ADAPTIVE_THRESH_MEAN_C, CV_THRESH_BINARY, 5, 2);

  cvFilter2D(grayImg, grayImg, lapKernel, cvPoint(-1,-1)); // sharpen

  return grayImg.getBufferedImage();
}  // end of enhanceImage()
```

The smoothing removes some of the graininess from the image, and the adaptive thresholding maps the image to black and white, taking into account varying light intensity. The final step is to sharpen the image using a Laplace filter, which is created in TCPanel.initOpenCV():

```
// in initOpenCV(): make Laplace kernel
lapKernel = CvMat.create(3, 3);
lapKernel.put(1, 1, 5.0);
lapKernel.put(0, 1, -1.0);
lapKernel.put(2, 1, -1.0);
lapKernel.put(1, 0, -1.0);
lapKernel.put(1, 2, -1.0);
```

The transformation carried out by enhanceImage() is illustrated in Figure 9.

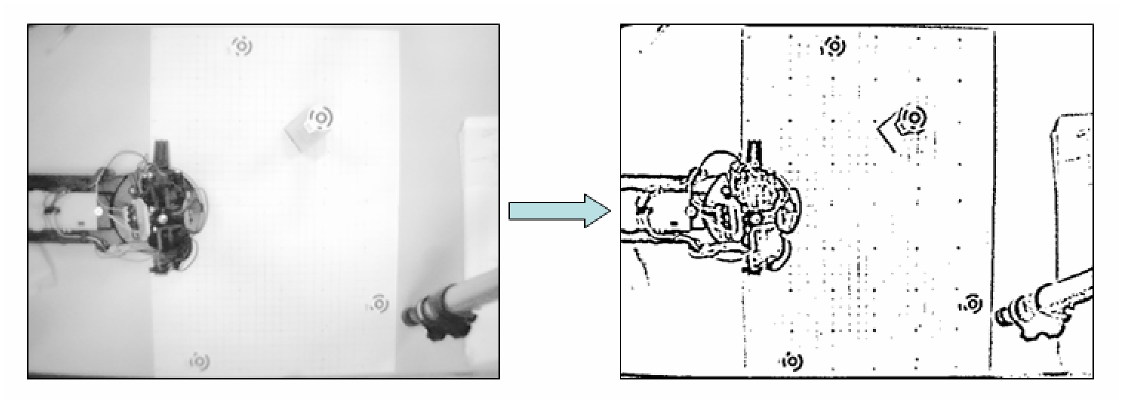

Figure 9. The Effects of enhanceImage()

enhanceImage() makes the topcode symbols more distinct, allowing Scanner.scan() to reliably find all of them at runtime.

3.1. Locating Landmarks and Trackers

locateItems() passes the topcodes' image coordinates to the Landmarks and Trackers objects.

```
// globals
private int imageHeight = 0;
private Landmarks landmarks;    // the two kinds of topcodes
private Trackers trackers;


private void locateItems(java.util.List<TopCode> topCodes)
{
  if ((topCodes == null) || (topCodes.size() == 0))  // no topcodes
    return;

  for (TopCode tc : topCodes) {
    int id = tc.getCode();
    int xImage = (int) tc.getCenterX();
    int yImage = imageHeight - (int) tc.getCenterY();
            // so y-axis goes up screen

    // store the image coordinates
    if (landmarks.isLandmark(id)) {
      if (!landmarks.canMapCoords())
        /* only keep adding landmark info until
           a mapping is obtained */
        landmarks.setImageCoord(id, xImage, yImage);
    }
    else
      trackers.setImageCoord(id, xImage, yImage);
  }

  if (landmarks.canMapCoords())
    trackers.calcGridCoords();
}  // end of locateItems()
```

Once the Landmarks class has been passed image coordinates for at least two of its topcodes, it can calculate an image-to-grid coordinates mapping (I'll explain the details when I describe the Landmarks class). From then on, Landmarks.canMapCoords() will return true.

The Trackers object employs this mapping in calcGridCoords() to convert its topcodes' image coordinates to grid positions. calcGridCoords() is called repeatedly because trackers can move, so their changing image coordinates will require re-mapping to the grid.

The topcodes vision library plots its codes using the standard image coordinate system where the top-left of the image is at (0,0). However, the image-to-grid mapping is somewhat easier to define if the image's origin is at the bottom-left. This is achieved by subtracting the topcode's y value from the height of the webcam image:

```
int yImage = imageHeight - (int) tc.getCenterY();
```

imageHeight is assigned a value in TCPanel.initDisplay() at the same time that the panel's dimensions are set.

3.2. Displaying a TopCode

TCPanel.paintComponent() displays the usual webcam image and statistics, and also calls drawTopCodes() to render the topcodes. Each code is represented by three elements: a circular black-and-white graphic, its ID, and a grid coordinate, as shown in Figure 10.

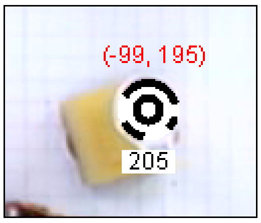

Figure 10, A Rendered Topcode.

The drawTopCodes() method:

```
// globals
private java.util.List<TopCode> topCodes = null;
private Landmarks landmarks;    // the two kinds of topcodes
private Trackers trackers;


private void drawTopCodes(Graphics2D g2)
{
  if ((topCodes == null) || (topCodes.size() == 0)) // no topcodes
    return;

  for (TopCode tc : topCodes) {
    tc.draw(g2);    // draw a topcode image at its image location

    // access the topcode's attributes
    int id = tc.getCode();
    String idStr = String.valueOf(id);
    int dia = (int) tc.getDiameter();
    int xCenter = (int) tc.getCenterX();
    int yCenter = (int) tc.getCenterY();
    int strWidth = g2.getFontMetrics().stringWidth(idStr);

    // write ID in black inside a white box below the topcode image
    g2.setColor(Color.WHITE);
    g2.fillRect( (int)(xCenter - strWidth/2 - 3),
                 (int)(yCenter + dia/2 + 6),   strWidth+6, 12);
    g2.setColor(Color.BLACK);
    g2.drawString(idStr, xCenter - strWidth/2, yCenter + dia/2 + 16);

    // is the ID for a tracker or a landmark?
    TopCodeItem item = trackers.findTracker(id);
    if (item == null)
      item = landmarks.findLandmark(id);

    // write grid coordinates in red above the topcode image
    if ((item != null) && item.isLocatedOnGrid()) {
      g2.setColor(Color.RED);
      g2.drawString("(" + item.getXGrid() + ", " + item.getYGrid() +
                  ")", xCenter - dia/2 - 8, yCenter - dia/2 - 8);
    }
  }
}  // end of drawTopCodes()
```

The method calls TopCode.draw() to render the graphics, but the printing of the grid coordinates requires TopCodeItem objects from the landmarks and trackers.

4. The TopCodeItem Class

The Topcode library's TopCode class stores a code's ID, its location in the image, its angular orientation, and diameter. Unfortunately, this isn't quite enough because I also need each code's position on the grid paper. My TopCodeItem class maintains both coordinates for a code, and offers support methods for calculating the mapping between them.

```
// globals in the TopCodeItem class
private int id;
private int xGrid, yGrid;     // position on the grid (in mm)
private int xIm, yIm;         // position on the image
private boolean isOnGrid, isOnImage;


public TopCodeItem(int id)
{
  this.id = id;
  isOnGrid = false;   // no coords assigned to code yet
  isOnImage = false;
}  // end of TopCodeItem()


public void setGridCoord(int x, int y)
{  xGrid = x;
   yGrid = y;
   isOnGrid = true;
}

public void setImageCoord(int x, int y)
{ xIm = x;
  yIm = y;
  isOnImage = true;
}
```

The mapping calculation carried out by the Landmarks class needs the distance and angle between two topcodes, in both the image and grid coordinate systems.

The grid distance and angle are calculated using a bit of trigonometry, as illustrated by Figure 11, where the values are obtained for the topcodes 333 and 283, with the angle relative to topcode 333.

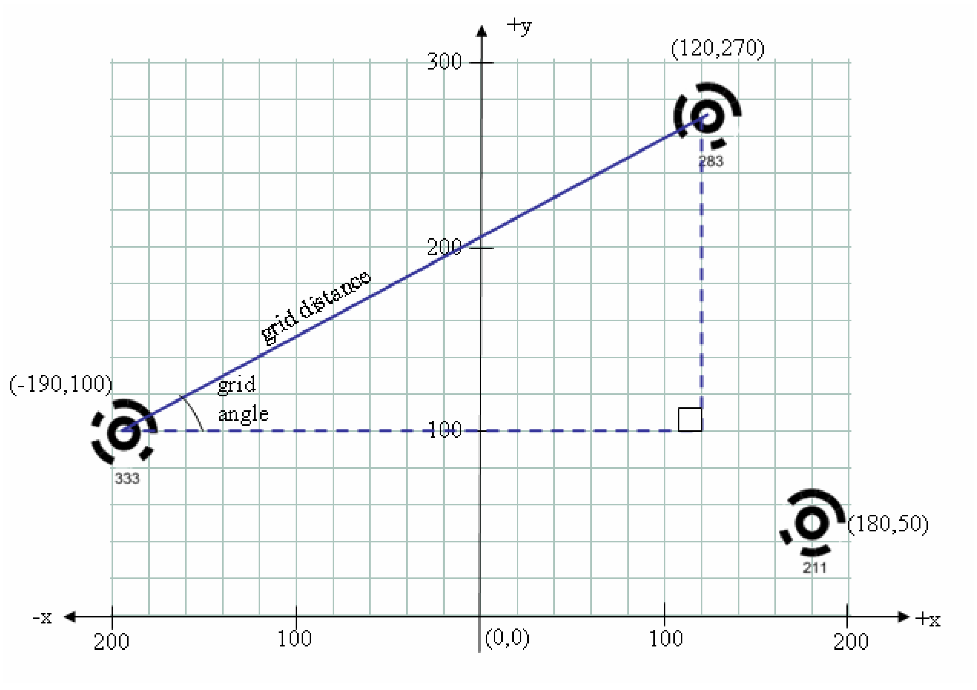

Figure 11. The Grid Distance and Angle.

The distance and angle are returned by the gridDistTo() and gridAngleTo() methods in TopCodeItem:

```
public double gridDistTo(TopCodeItem item)
// calculate grid distance to item
{
  if (!isOnGrid) {
    System.out.println("" + id + " not found on grid");
    return 0;
  }
  if (!item.isLocatedOnGrid()) {
    System.out.println("" + item.getID() + " not found on grid");
    return 0;
  }
  int x = item.getXGrid();
  int y = item.getYGrid();
  return Math.sqrt( (xGrid-x)*(xGrid-x) + (yGrid-y)*(yGrid-y));
}  // end of gridDistTo()


public int gridAngleTo(TopCodeItem item)
// calculate grid degree angle to item
{
  if (!isOnGrid) {
    System.out.println("" + id + " not found on grid");
    return 0;
  }
  if (!item.isLocatedOnGrid()) {
    System.out.println("" + item.getID() + " not found on grid");
    return 0;
  }
  int x = item.getXGrid();
  int y = item.getYGrid();
  double radAngle = Math.atan2( (double)(y-yGrid),
                                (double)(x-xGrid) );
  int angle = (int) Math.round( Math.toDegrees(radAngle) );
  return angle;
}  // end of gridAngleTo()
```

The image distance and angle are obtained in the same way but by employing the image coordinates of the topcodes (see Figure 12).

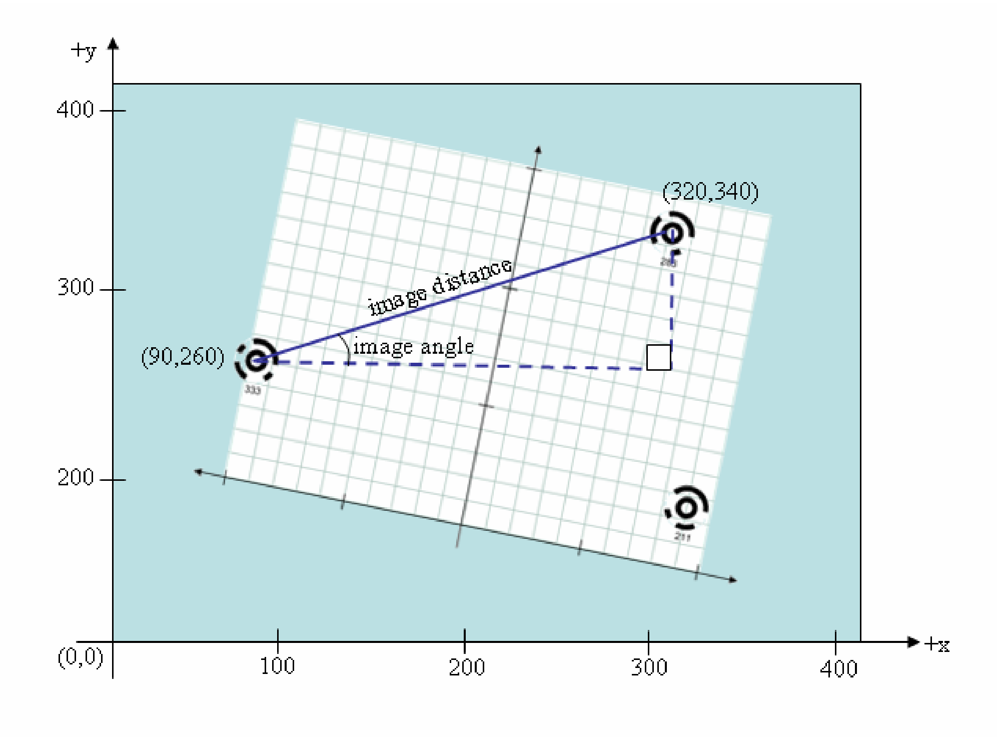

Figure 12. The Image Distance and Angle.

Note that the image coordinate system has the y-axis pointing upwards. The TopCodeItem imageDistTo() and imageAngleTo() methods are:

```
public double imageDistTo(TopCodeItem item)
// calculate image distance to item
{
  if (!isOnImage) {
    System.out.println("" + id + " not found on image");
    return 0;
  }
  if (!item.isLocatedOnImage()) {
    System.out.println("" + item.getID() + " not found on image");
    return 0;
  }
  int x = item.getXImage();
  int y = item.getYImage();
  return Math.sqrt( (x-xIm)*(x-xIm) + (y-yIm)*(y-yIm));
}  // end of imageDistTo()


public int imageAngleTo(TopCodeItem item)
// calculate image  angle to item
{
  if (!isOnImage) {
    System.out.println("" + id + " not found on image");
    return 0;
  }
  if (!item.isLocatedOnImage()) {
    System.out.println("" + item.getID() + " not found on image");
    return 0;
  }
  int x = item.getXImage();
  int y = item.getYImage();
  double radAngle = Math.atan2( (double)(y-yIm), (double)(x-xIm) );
  int angle = (int) Math.round( Math.toDegrees(radAngle) );
  return angle;
}  // end of imageAngleTo()
```

5. Finding Landmarks

The Landmarks object is assigned topcodes with fixed grid positions (which come from a landmarks.txt file) but their image positions are initially unknown. TCPanel repeatedly calls Landmarks.setImageCoord() to assign image coordinates to those codes. When there are at least two landmark topcodes with both grid and image coordinates, an image-to-grid mapping can be calculated.

Once the mapping exists, the Trackers object can start asking Landmarks to calculate the grid positions of its topcodes.

My application uses three landmarks (seen around the edge of the grid in Figure 11). Their grid positions are specified in landmarks.txt:

```
// landmarks for large graph
// id xGrid yGrid (in mm)

333 -190 100
283 120 270
211 180 50
```

The three topcodes have IDs 333, 283, and 211. The Landmarks.readInfo() method creates a TopCodeItem object for each data line from the file, and stored them in an

ArrayList called landmarks. The Landmarks() constructor also creates an empty array called located[]:

```
// globals
private static final String LANDMARKS_FNM = "landmarks.txt";

private static final int MIN_LANDMARKS = 2;
     /* minimum of landmarks that need to be 'located'
       (i.e. have  both grid and image coords). */


private ArrayList<TopCodeItem> landmarks;      // all the landmarks
private TopCodeItem[] located;
           // landmarks with both grid and image coords
private int numLocated;
          // no. of landmarks currently located on the image


public Landmarks()
{
  landmarks = new ArrayList<TopCodeItem>();
  readInfo(landmarks, LANDMARKS_FNM);

  located = new TopCodeItem[MIN_LANDMARKS];
  numLocated = 0;
}  // end of Landmarks()
```

The located[] array hold references to the topcodes which have both grid and image coordinates. MIN_LANDMARKS (2) such codes are needed before the image-to-grid mapping can be calculated.

Topcodes in the Landmarks object are assigned image coordinates by TCPanel calling Landmarks.setImageCoord():

```
// globals
private boolean canMapCoords = false;
                    // is the mapping possible yet?


public void setImageCoord(int id, int xImage, int yImage)
// set landmark id's image coordinate
{
  if (canMapCoords){
    System.out.println("No need for " + id +
                "; enough landmarks have been located");
    return;
  }

  TopCodeItem item = findLandmark(id);    // find id landmark
  if (item == null) {
    System.out.println("" + id + " not a landmark");
    return;
  }

  if (item.isLocatedOnImage()) {
    System.out.println("" + id + " already located on the image");
    return;
  }
  // record image coordinate
  item.setImageCoord(xImage, yImage);
  located[numLocated] = item;
  numLocated++;
  if (numLocated == MIN_LANDMARKS) {
    calcMapping();
    canMapCoords = true;
        // trackers image coords can now be mapped to grid coords
  }
}  // end of setImageCoord()
```

Once MIN_LANDMARKS codes have been issued image coordinates, calcMapping() is called to calculate the mapping from image to grid coordinates. This is defined using two parameters – a scale factor from image to grid distances, and a rotation offset for moving from images to grid orientations.

```
private void calcMapping()
{
  // distance scale factor (image --> grid)
  double gridDist = located[0].gridDistTo(located[1]);
  double imageDist = located[0].imageDistTo(located[1]);
  distanceScaleFactor = gridDist/imageDist;

  // angle difference (image --> grid)
  int gridAngle = located[0].gridAngleTo(located[1]);
  int imageAngle = located[0].imageAngleTo(located[1]);
  angleDifference = gridAngle - imageAngle;
}  // end of calcMapping()
```

Figures 11 and 12 show typical values for the grid distance, grid angle, image distance, and image angle between two topcodes.

The mapping only requires two parameters since both coordinate systems are flat, allowing the transformation between them to be *affine*.

Once the mapping parameters have been calculated, the Trackers object can start sending topcodes to Landmarks.mapToGridCoord() to obtain grid values.

```
// globals
// landmarks mappings from (image --> grid) coordinates
private double distanceScaleFactor;
private int angleDifference;
private boolean canMapCoords;   // is the mapping possible yet?


public boolean mapToGridCoord(TopCodeItem trackerItem)
{
  if (!canMapCoords) {
    System.out.println("Insufficent landmarks");
    return false;
  }

  if (!trackerItem.isLocatedOnImage()) {
    System.out.println("Item has no image coords");
    return false;
  }
  // store the old grid coordinates for this tracker;
  // will be 0 if the tracker has no previous grid coords
  int xPrev = trackerItem.getXGrid();
  int yPrev = trackerItem.getYGrid();

  // map item to grid distance and angle from located[0]
  double gItemDist = located[0].imageDistTo(trackerItem) *
                                 distanceScaleFactor;
  double gItemAngle = located[0].imageAngleTo(trackerItem) +
                                 angleDifference;
  double ang = Math.toRadians(gItemAngle);

  // calculate grid (x,y) position of item
  double xItem = located[0].getXGrid() +
                      (gItemDist * Math.cos(ang));
  int x = (int)Math.round(xItem);
  double yItem = located[0].getYGrid() +
                      (gItemDist * Math.sin(ang));
  int y = (int)Math.round(yItem);

  if ((x != xPrev) || (y != yPrev))  // update if changed
    trackerItem.setGridCoord(x, y);

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

mapToGridCoord() starts by obtaining the image distance and angle between the first landmark and the supplied tracker (see Figure 13).

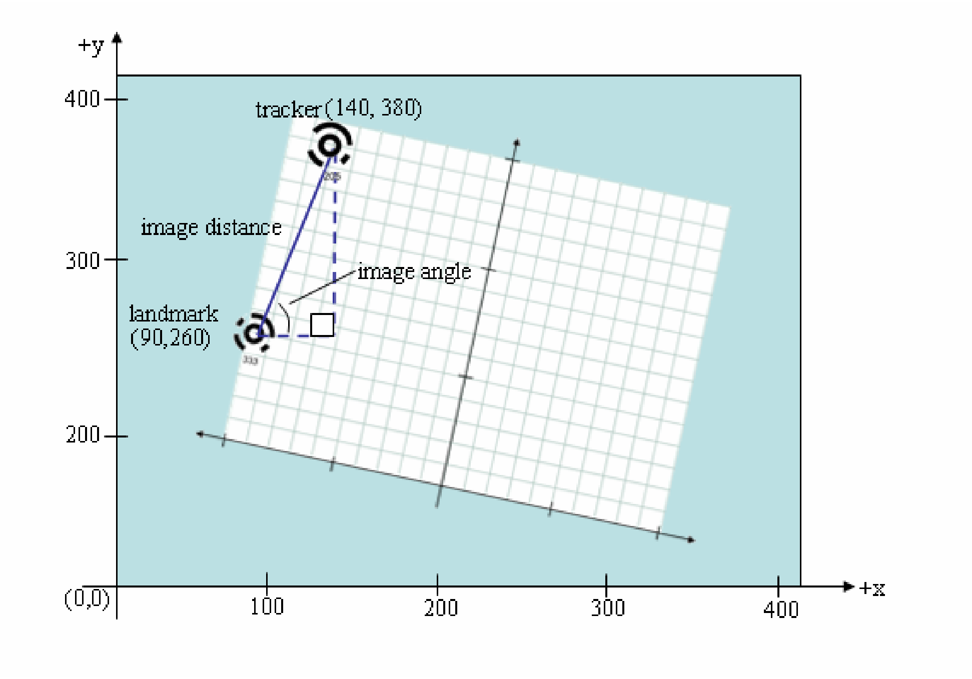

Figure 13. Image Position of Tracker Relative to Landmark.

The mapping converts these values into a *grid* distance and angle between the two topcodes (see Figure 14).

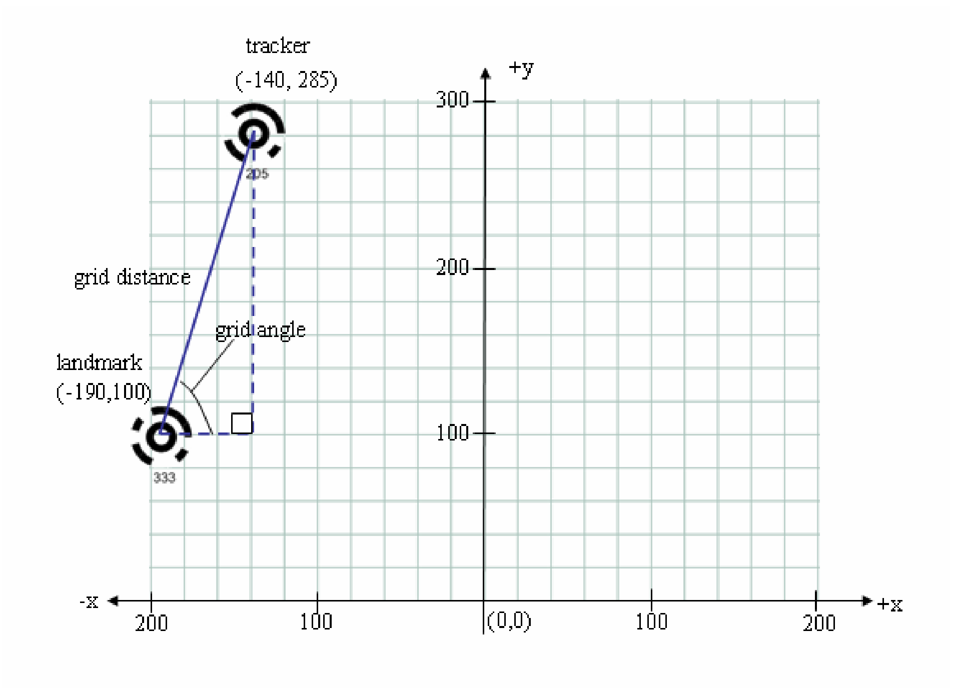

Figure 14. Grid Position of Tracker Relative to Landmark.

The Cartesian (x, y) grid coordinates of the tracker is obtained by combining the landmark's position with the x- and y- offset of the tracker.

6. Tracking Moving Topcodes

The Trackers object stores topcodes that can move, which mean their image coordinates may change over time. Modifications are done by TCPanel calling Trackers.setImageCoord():

```
// globals
private ArrayList<TopCodeItem> trackers;      // all the trackers


public void setImageCoord(int id, int xImage, int yImage)
{
  TopCodeItem item = findTracker(id);
  if (item != null)   // tracker already exists
    item.setImageCoord(xImage, yImage);   // update position
  else {
    TopCodeItem newItem = new TopCodeItem(id);
    newItem.setImageCoord(xImage, yImage);
    trackers.add(newItem);
  }
}  // end of setImageCoord()
```

Trackers.setImageCoord() is called from TCPanel.locateItems(). locateItems() also calls Trackers.calcGridCoords() to recalculate of the tracker codes' grid coordinates:

```
// global
```

```
private ArrayList<TopCodeItem> trackers;      // all the trackers
private Landmarks landmarks;    // for grid coord calculations


public void calcGridCoords()
{
  if (!landmarks.canMapCoords())
    System.out.println("Insufficent landmarks");
  else {
    for(TopCodeItem item : trackers)
      landmarks.mapToGridCoord(item);
  }
}  // end of calcGridCoords()
```

The Trackers object deals with the recalculation of its codes' grid positions by sending the codes over to the Landmarks object for updating.