Chapter 8.7. Finger Tracking with TopCodes

This chapter describes how to implement finger tracking using TopCode markers (http://users.eecs.northwestern.edu/~mhorn/topcodes/). The objective is to use a webcam to track finger movements, employing them as the equivalents of mouse moves and button presses inside an application (see Figure 1).

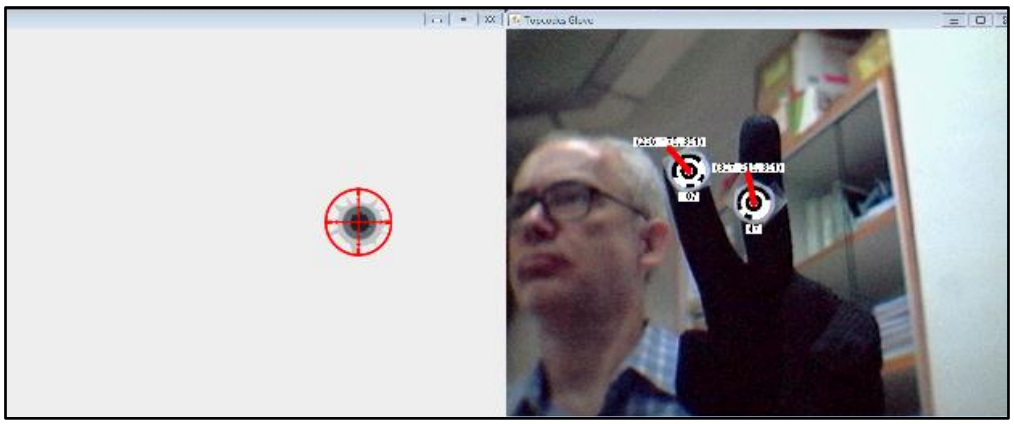

Figure 1. My Finger Tracking Application.

The movement of my fingers shown in the webcam image on the right of Figure 1 causes the crosshairs image to move and change in the panel on the left of the figure.

This doesn't use all the functionality offered by TopCodes, so I'll also spend some time explaining how to access and utilize those other features.

1. What's a TopCode?

TopCodes (Tangible Object Placement Codes) is a computer vision library for identifying a collection of tagged objects on a flat surface (http://users.eecs.northwestern.edu/~mhorn/topcodes/). It's possible to use up to 99 different tags, each one represented by a series of black-and-white nested rings. Two TopCodes are shown in Figure 2.

Figure 2. Example TopCodes.

Every TopCode has an inner black ring surrounded by a segmented outer ring which encodes a 13-bit number -- zero is represented by a black sector and one by white.

The library includes its own CRC-based error detection scheme, which allows a marker to be detected even when rotated, so long as it stays facing the camera.

A user attaches printed copies of the markers to points of interest in a scene (TopCode PDFs are available from their website). The library identifies the tags in a picture, and each one is assigned an XY position, diameter, angular orientation, and an ID number. For instance, the TopCodes in Figure 2 have IDs 107 and 47.

The software is designed to work well with low-resolution digital cameras with poor optics (i.e. my webcam), and contains its own adaptive thresholding code. This means it can handle a variety of lighting conditions, so I don't need to utilize JavaCV to clean up and enhance the image. The library is fast and accurate, being able to recognize markers as small as 25 x 25 pixels in an image. The library comes with an optional Windows webcam library, but I'll be using JavaCV's FrameGrabber in this chapter, as usual.

The main drawback is that a TopCode must be exactly facing the camera to be recognized. TopCodes are a simplified version of the SpotCode format (http://en.wikipedia.org/wiki/ShotCode) which does support angular rotation of a marker, identification at greater distances from the camera, and more codes (19,683). Unfortunately, that format was developed into a commercial product in the mid 2000's with no Java support (http://www.shotcode.com/).

2. An Overview of the Application

I've implemented finger tracking by sticking two topcodes onto a black glove, as in Figure 3. TopCode 107 is attached to the tip of the index finger, and TopCode 47 is slightly lower down the middle finger.

Figure 3. A Glove with Attached TopCodes.

The aim is to use these two fingers to mimic the actions of mouse movement and button pressing. Translations are calculated from the (x, y) position of the index finger's TopCode. 'Button pressing' is triggered by the visibility of TopCode 47 on the middle finger. An example of this behavior is shown in Figure 4.

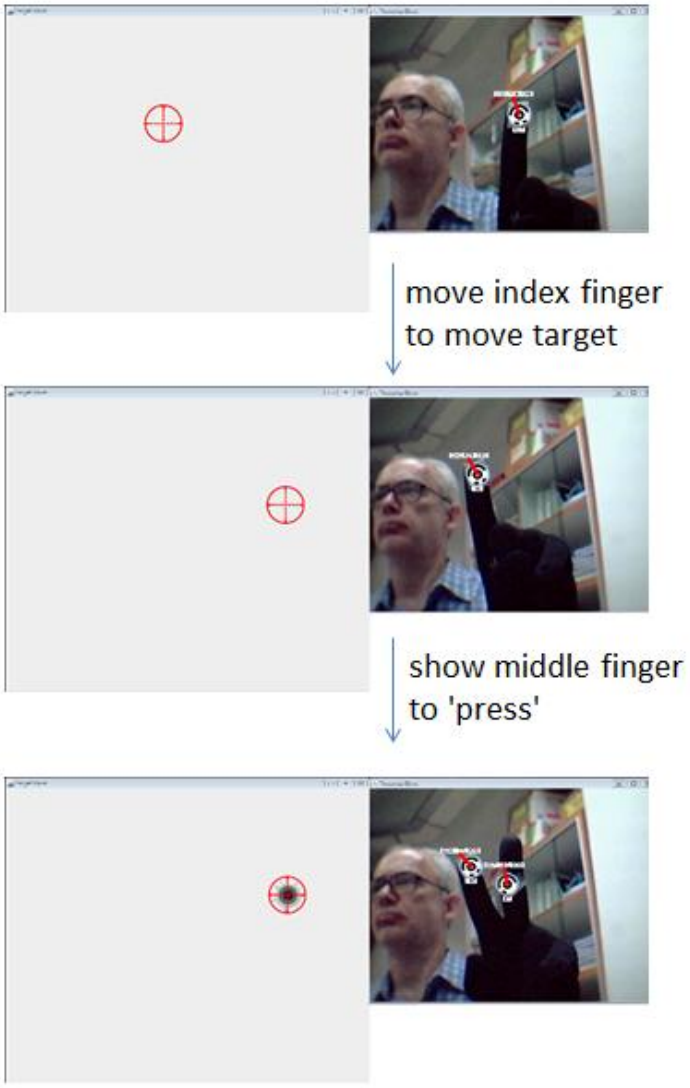

Figure 4. The TopCodesFingers Application in Action.

My TopCodesFingers application consists of two windows, a target window that contains a crosshairs image, and a webcam window that displays the user and additional TopCode information.

As the user moves his index finger (as shown in the first two screenshots of Figure 4), the target image moves. The target travels in the opposite x-direction from the finger in the webcam image, following the movement of the hand from the user's point of view. The translation is scaled so that a small horizontal or vertical change in front of the webcam is magnified in the target window.

The third screenshot of Figure 4 shows the user stretching out his middle finger, making TopCode 47 visible. This causes the target image to change to a 'fired' image to indicate a 'button press'. Figure 5 shows a close-up of the target and fired images.

Figure 5. The Target and Fired Images.

Figure 6 includes an enlarged fragment of the webcam window so the TopCodes information is easier to read.

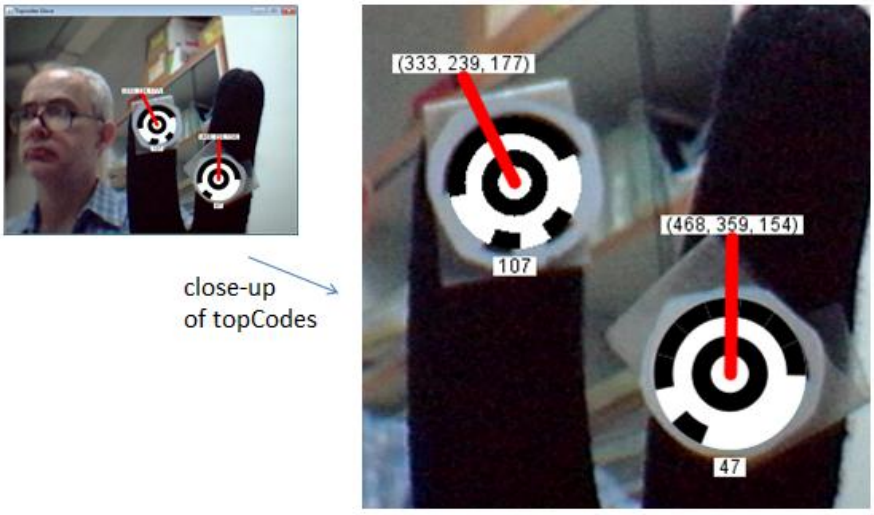

Figure 6. TopCodes Rendering in the Webcam Window.

Each tag is drawn as a black and white circular graphic, along with two white textboxes and a red line. The lower textbox gives the TopCode's ID, while the upper text is its (x, y, z) coordinate. The (x, y) part is in image pixels, as measured from the top-left corner of the webcam picture. The z coordinate is the distance in millimeters of the marker from the camera. This is calculated from the pixel diameter of the tag with the help of some calibration information supplied by the programmer, which I'll explain later. In Figure 6, the index finger is 177 mm from the camera, and the middle finger 154 mm.

The red line denotes the orientation of the TopCode relative to the vertical. TopCode 107 is rotated slightly to the left because I stuck it to the glove at an angle.

Figure 7 has another close-up of the webcam image after I'd raised my hand, turned it to the left, and moved it further from the camera.

Figure 6. Later TopCodes Rendering.

The coordinates of the TopCodes reflect the movement up, to the left, and away from the webcam (e.g. the index finger is now 309 mm from the camera). The rotation is signaled by the red lines being turned counterclockwise.

Most of this TopCodes information is ***not*** used in my finger tracking application. It's included to show the capabilities of TopCodes.

The class diagrams for TopCodesFingers are presented in Figure 7.

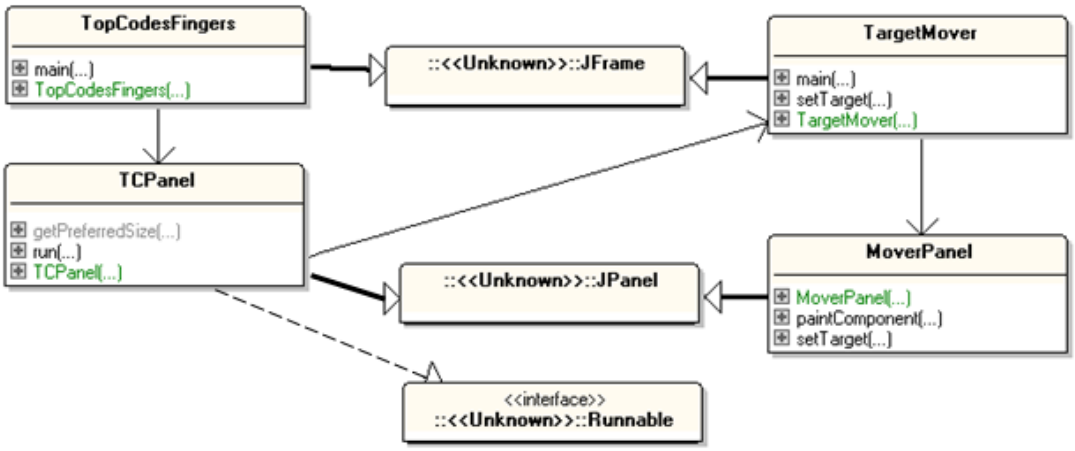

Figure 7. TopCodesFingers Class Diagrams.

The two-window approach is implemented using two JFrame subclasses called TopCodesFingers (for the Webcam screen) and TargetMover (for the target window). The important work is done by their JPanel subclasses, TCPanel and MoverPanel. TCPanel executes a threaded loop for capturing the webcam picture, processing it, drawing, sleeping, and then repeating. The main topics of this chapter will be the processing and drawing phases in TCPanel. The MoverPanel class is passed an (x, y) coordinate and an isPressed boolean which it uses to draw the correct crosshairs image inside the panel.

3. The Processing Loop

TCPanel's processing loop is located in its run() method. One difference from previous examples is the absence of JavaCV code for converting the webcam image to a grayscale, applying smoothing and thresholding. Those tasks are done automatically by the TopCodes Scanner.scan() method, which returns a list of TopCode objects extracted from the current picture.

```
// globals
private static final int DELAY = 50;  // time (ms) between redraws
private static final int CAMERA_ID = 0;

private BufferedImage im = null;
private volatile boolean isRunning;

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


public void run()
{
  FrameGrabber grabber = initGrabber(CAMERA_ID);
  if (grabber == null)
    return;

  IplImage snapIm;
  long duration;
  isRunning = true;

  while (isRunning) {
    long startTime = System.currentTimeMillis();

    snapIm = picGrab(grabber, CAMERA_ID);
    im = snapIm.getBufferedImage();
    topCodes = scanner.scan(im);  // find topcodes in the image
    trackFingers(topCodes);
    repaint();

    duration = System.currentTimeMillis() - startTime;
    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);
      }
      catch (Exception ex) {}
    }
  }
  closeGrabber(grabber, CAMERA_ID);
}  // end of run()
```

The scanner object used in run() is instantiated in TCPanel's constructor:

```
scanner = new topcodes.Scanner();
```


The DELAY period between updates is fairly short (50 ms) because the TopCode scanner processes an image quite quickly. The shorter delay means more finger updates per second, so the application becomes more responsive.

4. Extracting Tracking Information

trackFingers() looks for the TopCode IDs 107 and 47, assuming that tag 107 is attached to the user's index finger, and tag 47 is on the middle finger.

The (x, y) coordinate of TopCode 107 is stored globally, and the boolean isPressed is set when the middle finger tag is detected. The coordinate is be used when redrawing the panel, and is also passed to the TargetMover window for positioning the crosshairs.

```
// globals
// dimensions of panel == dimensions of webcam image
private static final int WIDTH = 640;
private static final int HEIGHT = 480;

// topcode IDs used for the fingers
private static final int INDEX_FINGER = 107;
private static final int MIDDLE_FINGER = 47;

private static final double FINGER_SCALE = 3;
  // for increasing finger movement relative to webcam center


// current relative finger position inside the panel
private double xFinger = 0.5;   // at the center
private double yFinger = 0.5;

private TargetMover targetFrame;
  // the window whose target is moved (and 'pressed')
  // by finger movement


private void trackFingers(java.util.List<TopCode> topCodes)
{
  boolean isPressed = false;
  for (TopCode tc : topCodes) {
    int id = tc.getCode();
    if (id == INDEX_FINGER) {  // topcode 107 found
      // calculate dist of finger point from center of webcam image
      int xDist = (int)tc.getCenterX() - WIDTH/2;
      int yDist = (int)tc.getCenterY() - HEIGHT/2;

      /* scale distances, and convert to % positions inside the
         image (the values may be > 1 due to the scaling)  */
      xFinger = ((double)(WIDTH/2 + xDist*FINGER_SCALE))/WIDTH;
      yFinger = ((double)(HEIGHT/2 + yDist*FINGER_SCALE))/HEIGHT;
    }
    else if (id == MIDDLE_FINGER)   // topcode 47 is visible
      isPressed = true;
  }

  // move target using finger's relative position inside panel
```



```
  // and the isPressed boolean
  targetFrame.setTarget(xFinger, yFinger, isPressed);
}  // end of trackFingers()
```

The TopCode's (x, y) coordinate (returned by tc.getCenterX() and tc.getCenterY()) is scaled relative to its distance from the center of the webcam window. This means that the user only has to move their index finger a small amount in order to shift the crosshairs a large distance from the center of the target window.

The conversion stages for the coordinate are illustrated by Figure 8.

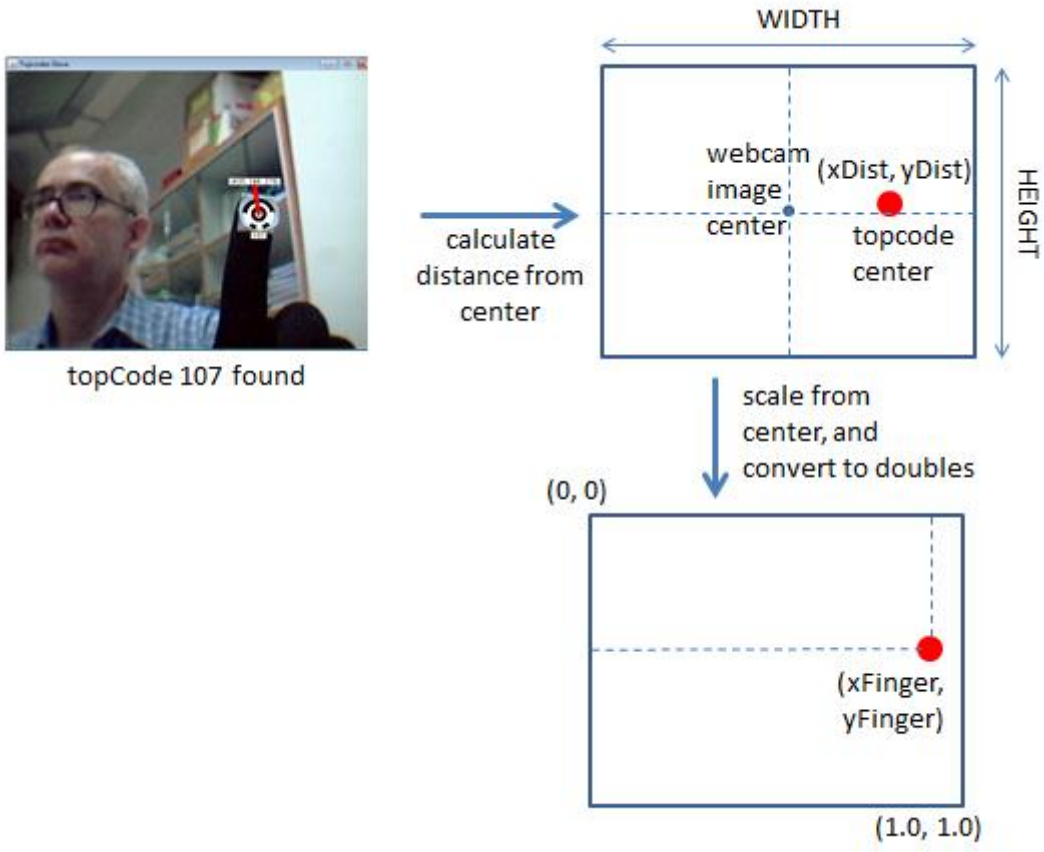

Figure 8. Converting a Finger Coordinate to a Target Position.

The scaling may result in (xFinger, yFinger) being located outside the (0,0) – (1, 1) x- and y- axis ranges, but that issue is handled by the setTarget() method in the target window.

TargetMover.setTarget() sends the position and boolean to the setTarget() method in the MoverPanel class, which converts the doubles into a panel coordinate.

```
// MoverPanel class globals
private int xCenter, yCenter;    // loc of the target image
private int pWidth, pHeight;     // dimensions of panel
private boolean isPressed = false;
   // used to decide whether to display the 'target' or 'fired' image
```



```
public void setTarget(double x, double y, boolean b)
{
  isPressed = b;   // used when painting

  xCenter = (int) Math.round(x*pWidth);    // convert to panel coords
  yCenter = (int) Math.round(y*pHeight);

  // keep the target visible on-screen
  if (xCenter < 0)
    xCenter = 0;
  else if (xCenter >= pWidth)
    xCenter = pWidth-1;

  // reverse xCenter so left-of-center <--> right-of-center
  xCenter = pWidth - xCenter;

  if (yCenter < 0)
    yCenter = 0;
  else if (yCenter >= pHeight)
    yCenter = pHeight-1;

  repaint();
}  // end of setTarget()
```

The coordinate is constrained so the target image stays visible inside the panel.

Another change is to reverse the x-axis value so that the target moves in the same direction as a finger from the point of view of the user. This is necessary because the webcam is facing the user, so left and right are reversed in its recorded images.

paintComponent() inside MoverPanel decides which crosshairs image to draw (see Figure 5) based on the isPressed boolean, and positions it at (xCenter, yCenter).

```
// globals
private BufferedImage targetIm, firedIm;  // taget/fired image
private int imWidth, imHeight;   // dimensions of both images

private int xCenter, yCenter;
private boolean isPressed = false;


public void paintComponent(Graphics g)
{
  super.paintComponent(g);

  // draw the target or fired image
  BufferedImage im = (isPressed) ? firedIm : targetIm;
  g.drawImage(im, xCenter-imWidth/2, yCenter-imHeight/2, null);
} // end of paintComponent()
```

5. Drawing the TopCodes

The paintComponent() method back in TCPanel is a little complicated since it draws TopCodes information on top of the webcam image (as illustrated in Figure 9).

Figure 9. TopCodes Rendering.

```
// inside TCPanel
// globals
private BufferedImage im = null;  // current webcam snap
private java.util.List<TopCode> topCodes = null;


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;
  g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                      RenderingHints.VALUE_INTERPOLATION_BILINEAR);

  if (im != null) {
    g2.drawImage(im, 0, 0, this);
    drawTopCodes(g2, topCodes);
  }
  else
    g2.drawString("Initializing, please wait...", 20, HEIGHT/2);
} // end of paintComponent()
```

The TopCodes rendering is carried out by drawTopCodes() which loops through the detected tags, drawing each one.

```
private void drawTopCodes(Graphics2D g2,
                      java.util.List<TopCode> topCodes)
{
  if ((topCodes == null) || (topCodes.size() == 0))   // no topcodes
    return;

  for (TopCode tc : topCodes) {
    tc.draw(g2);    // draw topcode at its location on the image
    drawID(g2, tc);
    drawPos(g2, tc);   // draw (x,y,z) and orientation line
```

```
  }
}  // end of drawTopCodes()
```

Four things are drawn for each TopCode: a black and white rings graphic, the tag's ID, its (x, y, z) position, and a red line indicating its orientation. The good news is that the TopCode class has a draw() method for painting the marker's rings at the current position, suitably scaled and rotated.

My drawID() method draws the TopCode's ID number in a text box below the rings graphic.

```
private void drawID(Graphics2D g2, TopCode tc)
{
  String idStr = String.valueOf( tc.getCode() );
  int y = (int)(tc.getCenterY() + tc.getDiameter()*0.7 + 8);
  drawTextBox(g2, idStr, (int)tc.getCenterX(), y);
}  // end of drawID()


private void drawTextBox(Graphics2D g2, String msg, int x, int y)
// draw the msg in black inside a white box centered at (x,y)
{
  int strWidth = g2.getFontMetrics().stringWidth(msg);

  g2.setColor(Color.WHITE);    // white box
  g2.fillRect(x - strWidth/2 - 3,  y - 16, strWidth + 6, 16);

  g2.setColor(Color.BLACK);    // black text
  g2.drawString(msg, x - strWidth/2, y - 4);
}  // end of drawTextBox()
```

The position of the textbox (i.e. the (x, y) coordinate in the call to drawTextBox()) is calculated using the TopCode's diameter (TopCode.getDiameter()), which varies based on the tag's proximity to the camera.

6. TopCode Position and Orientation

Calculating the x- and y- parts of the (x, y, z) coordinate of a TopCode is easy – just call TopCode.getCenterX() and TopCode.getCenterY() to obtain an (x, y) pixel coordinate inside the image. The z- value is slightly harder to generate, relying on the *inverse* relationship between the TopCode's pixel diameter and its distance from the camera. This means that as a tag's distance from the camera increases then its on-screen pixel diameter decreases. This relationship can be used to convert a diameter (obtained from TopCode.getDiameter()) into a z- distance, so long as the programmer hardwires some conversion data into the code. Figure 10 illustrates the idea:

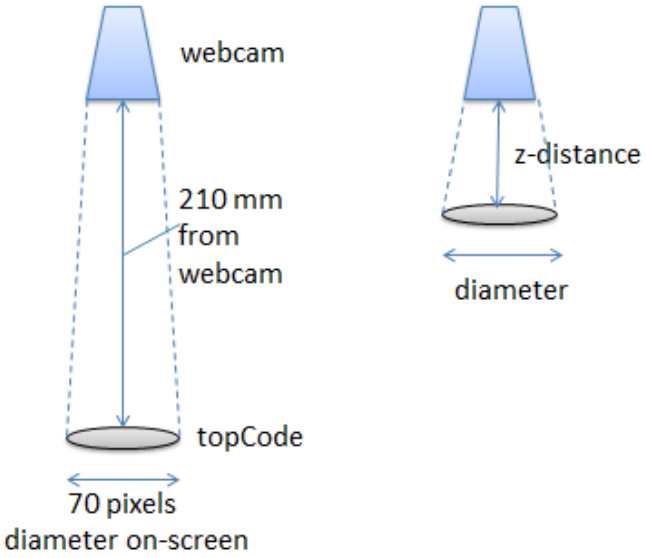

Figure 10. Converting TopCode Diameter to z-distance.

I added a simple println to the application which reported the current pixel diameter of a TopCode, and used a ruler to measure the corresponding distance between the marker (stuck on my glove) and the webcam lens. I obtained the data shown on the left of Figure 10, which shows that 210 mm $\propto$ 1/70 pixels, or that 210 = k/70, so k = 70 *210. This allows me to express the equality z-distance = k/diameter, which is implemented in the drawPos() method as:

```
int zDist = -1;
float dia = tc.getDiameter();   // diameter in pixels
if (dia > 0)
  zDist = (int)(DIST_DIA / dia);   // DIST_DIA = k = 210 * 70
```

The DIST_DIA value is defined as a constant, byt will need to be adjusted to match the characteristics of the webcam being used.

A tag's orientation around the z-axis is obtained by calling TopCode.getOrientation(), which returns a value in radians. In terms of degrees, the value is rather strange, varying between -360 degrees when the TopCode is pointing straight up, increasing to around +3 degrees as the tag is rotated clockwise by one full turn. The positive values may suddenly toggle to their equivalent negative values (e.g. 3 degrees can become -357). This behavior is easily simplified with the aid of some math:

```
int angle = 360 + (int) Math.toDegrees(tc.getOrientation());
angle = angle % 360;
```

This ensures that the angle varies between 0 and 359 degrees, with 0 indicating that the TopCode is pointing straight up, and the angle increasing in a clockwise direction, as in Figure 11.

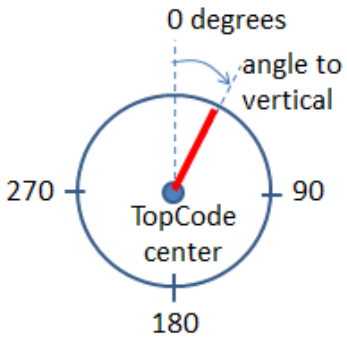

Figure 11. TopCode Rotational Range.

Converting the angle into a line is straightforward since the center point and diameter of the TopCode are available. The end point for the line is calculated like so:

```
int xEnd = xc + (int)( 0.9 * dia * Math.sin(Math.toRadians(angle)));
int yEnd = yc - (int)( 0.9 * dia * Math.cos(Math.toRadians(angle)));
```

The 0.9 scale factor for the line length was chosen at random. The calculation is illustrated in Figure 12.

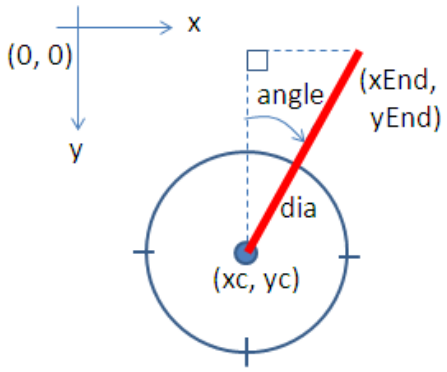

Figure 12. Calculating an End Point.

drawPos() combines the calculation of the z-distance, the angle, the line's end point, and rendering:

```
// globals
private static final double DIST_DIA = 210 * 70;
               // change this constant for different webcams


private void drawPos(Graphics2D g2, TopCode tc)
{
  int xc = (int) tc.getCenterX();     // in pixels
  int yc = (int) tc.getCenterY();

  int zDist = -1;
  float dia = tc.getDiameter();    // diameter in pixels
  if (dia > 0)
```



```
    zDist = (int)(DIST_DIA / dia);   // z-distance in mm from camera

  int angle = 360 + (int) Math.toDegrees(tc.getOrientation());
           // since topcodes orientation varies from -360 to 3
  angle = angle % 360;

  // calculate a rotated point using the angle and (xc, yc)
  int xEnd = xc + (int)( 0.9 * dia *
                         Math.sin(Math.toRadians(angle)));
  int yEnd = yc - (int)( 0.9 * dia *
                         Math.cos(Math.toRadians(angle)));

  // draw topcode coordinate in a text box
  String coord = String.valueOf("(" + xc + ", " + yc + ", " +
                                                    zDist + ")");
  drawTextBox(g2, coord, xEnd, yEnd);

  // draw a angled line going to the box
  g2.setColor(Color.RED);  // rounded, thick red line
  g2.setStroke(new BasicStroke(8, BasicStroke.CAP_ROUND,
                               BasicStroke.JOIN_ROUND));
  g2.drawLine(xc, yc, xEnd, yEnd);
}  // end of drawPos()
```

7. TopCodes Summarized

TopCodes are a simple but powerful way of obtaining 3D coordinate and z-axis orientation information for multiple points in a scene. The library is designed to work well in poor light, and with webcams of less than stellar quality. Tags can be detected over a range of distances, including at close quarters to the camera, which is useful for vision-based UI applications.

The main drawback of TopCodes is the need for a marker to be directly facing the webcam. Another possible drawback is their visibility; in my application a user must wear a glove with TopCodes stuck to it, which may not be possible in some use-case scenarios.