NUI Chapter 3. Motion Detection

[**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.]

This chapter explains how to use a webcam to detect change or movement in a scene. Since this is such a common requirement of video-based systems, I'll describe **three** different approaches. The first is based on the **image differencing** of consecutive video frames, the second utilizes **background/foreground segmentation**, and the third employs **optical flow**.

The image-difference detector highlights any movement between frames with a pair of crosshairs at the center-of-gravity (COG) of the motion. The application, called MotionDetector, is shown in Figure 1.

Figure 1. Image-difference Detection over Time.

I'll develop the application in two stages. First I'll focus on the detection problem by implementing code using a JavaCV-based test-rig. In the second step, I'll integrate the resulting detection class with the JavaCV grabber code from Chapter 2 that uses a JFrame and threaded JPanel. I'll also add the crosshairs graphic and code to make the tracking follow the user more smoothly.

The second technique, background substitution, utilizes a learning algorithm based on background and foreground clustering using Gaussian distributions. It's a more sophisticated detector of movement than image differencing, and is available in JavaCV as the class BackgroundSubtractorMOG2. Figure 2 shows the applications in action, with the detected foreground shown in white in the right-hand panel.

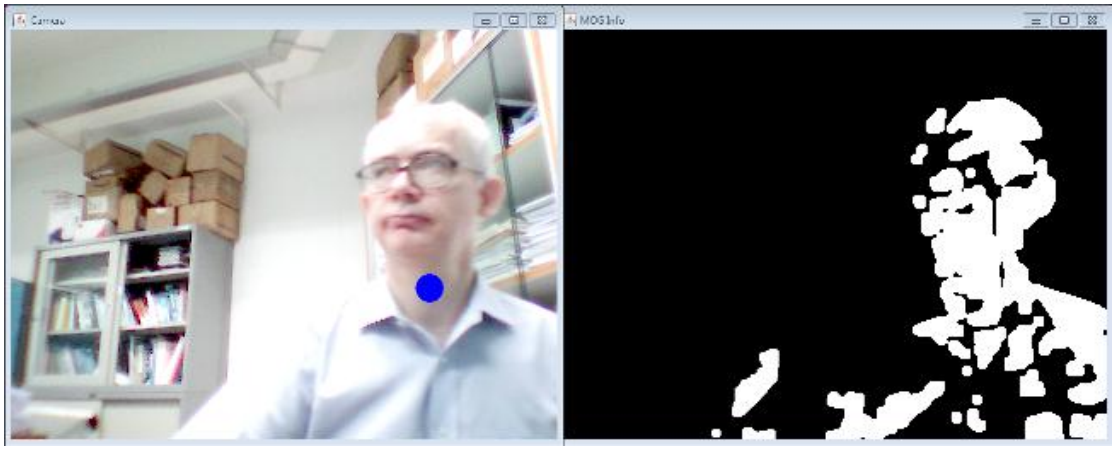

Figure 2. Foreground Detection.

The large blue circle drawn on the grabbed image in the left-hand panel represents the center-of-gravity (COG) of the foreground region. I'll explain the code in section 4.

The third movement processing technique I'll look at is (sparse) optical flow which matches up 'features' (pixels or small groups of pixels) that occur in consecutive video frames. My OpticalFlowMove application is illustrated in Figure 3. Each feature movement is shown as a blue arrow, and the overall COG of these direction vectors is denoted by a large red circle.

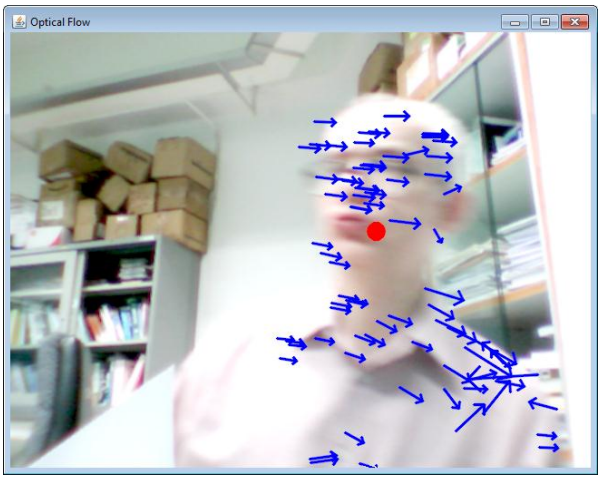

Figure 3. Optical Flow.

This optical flow code will be examined in section 5.

1. Image-differencing in JavaCV

The test-rig for movement detection based on comparing consecutive video frames is shown in Figure 4. It consists of two side-by-side JavaCV CanvasFrame displays.

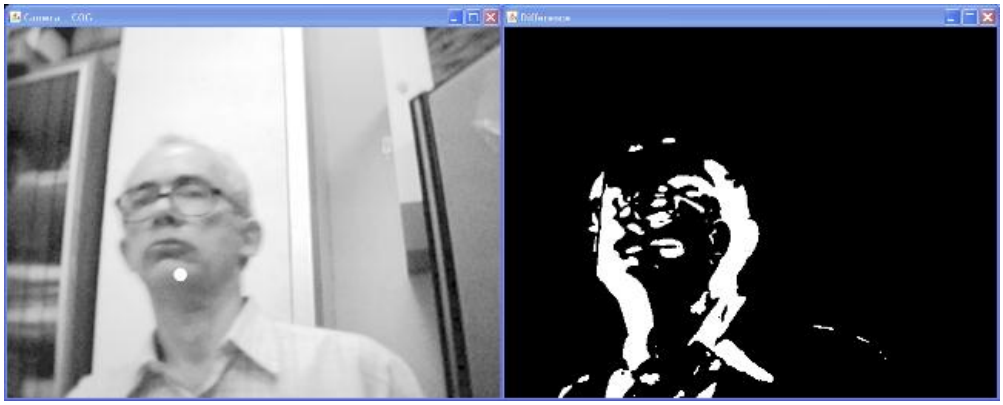

Figure 4. The Motion Detector Test-rig.

The left-hand display shows a grayscale version of the current webcam image with a white circle placed at the center-of-gravity of the detected motion (it's on my chin in Figure 4). The right-hand display shows the "threshold difference" between the current and previous webcam frames. The white areas denote a change or movement between the two images, while the black areas are where the two frames are the same.

When Figure 4 was generated, I was moving my head to the left, which is shown as the *two* blocks of white pixels in the difference image. There are two areas because the consecutive frames are different on both sides of my head as I move to the left. The position of the white dot in the grayscale image is calculated using the white areas in the difference image, as I'll explain later.

The test-rig starts by creating a JavaCV OpenCVFrameGrabber instance for accessing the webcam, and initializes an instance of my CVMotionDetector class. (OpenCVFrameGrabber is a concrete subclass of FrameGrabber which always uses the capture capabilities of OpenCV). The main() function then enters a loop which updates the two canvases until one of them is closed. The code:

```
public static void main(String[] args) throws Exception
{
  System.out.println("Initializing frame grabber...");
  OpenCVFrameGrabber grabber = new OpenCVFrameGrabber(CV_CAP_ANY);
  grabber.start();

  CVMotionDetector md = new CVMotionDetector( grabber.grab() );

  Dimension imgDim = md.getSize();
  IplImage imgWithCOG = IplImage.create(imgDim.width, imgDim.height,
                                        IPL_DEPTH_8U, 1);

  // two canvases for the image+COG and difference images
  // left-hand display
  CanvasFrame cogCanvas = new CanvasFrame("Camera + COG");
  cogCanvas.setLocation(0, 0);

  // right-hand display
  CanvasFrame diffCanvas = new CanvasFrame("Difference");
  diffCanvas.setLocation(imgDim.width+5, 0);

  // display grayscale+COG and diff images, until a window is closed
  while (cogCanvas.isVisible() && diffCanvas.isVisible())  {
```



```
    long startTime = System.currentTimeMillis();
    md.calcMove( grabber.grab() );

    // show current grayscale image with COG drawn onto it
    cvCopy(md.getCurrImg(), imgWithCOG);
    Point cogPoint = md.getCOG();
    if (cogPoint != null)
      cvCircle(imgWithCOG, cvPoint(cogPoint.x, cogPoint.y), 8,
               CvScalar.WHITE, CV_FILLED, CV_AA, 0);
    cogCanvas.showImage(imgWithCOG);

    diffCanvas.showImage(md.getDiffImg());  // update diff image

    System.out.println("Processing time: " +
                    (System.currentTimeMillis() - startTime));
  }

  grabber.stop();
  cogCanvas.dispose();
  diffCanvas.dispose();
}  // end of main()
```

The CVMotionDetector constructor is passed a webcam picture which becomes the initial 'previous' frame for use later on.

CVMotionDetector.getSize() returns the size of the webcam image, which is used to create an empty IplImage object called imgWithCOG, and two JavaCV CanvasFrames positioned side-by-side on the screen.

At the start of each iteration of the while-loop, CVMotionDetector.calcMove() is passed the current webcam image, which is treated as the 'current' frame, compared to the previous frame, resulting in a new difference image. CVMotionDetector also makes the current frame the new previous frame for use when the loop next iterates.

The CVMotionDetector.getCOG() call returns the difference image's center-of-gravity (COG) as a Java Point object.

A grayscale version of the current frame is retrieved from CVMotionDetector, and copied into the imgWithCOG image. A white circle can then be drawn onto the copy without affecting the original frame.

The two canvases are updated, one with the imgWithCOG image, the other with the difference image (obtained from CVMotionDetector).

On my slow test machine, the processing time for each loop iteration averaged about 80 ms, which is acceptable.

1.1. Initializing CVMotionDetector

CVMotionDetector's constructor initializes two of the three IplImage objects used during motion detection:

```
// globals
private IplImage prevImg, currImg, diffImg;
               // grayscale images (diffImg is bi-level)
private Dimension imDim = null;       // image dimensions
```



```
public CVMotionDetector(IplImage firstFrame)
{
  if (firstFrame == null) {
    System.out.println("No frame to initialize motion detector");
    System.exit(1);
  }

  imDim = new Dimension( firstFrame.width(), firstFrame.height() );
  System.out.println("image dimensions: " + imDim);

  prevImg = convertFrame(firstFrame);
  currImg = null;
  diffImg = IplImage.create(prevImg.width(), prevImg.height(),
                                                  IPL_DEPTH_8U, 1);
}  // end of CVMotionDetector()
```

prevImg, which holds the 'previous' frame, is initialized with the image passed to the constructor after it has been modified by convertFrame(). diffImg, which holds the difference image, is set to be an empty grayscale.

convertFrame() applies three operations to an image: blurring, color conversion to grayscale, and equalization:

```
private IplImage convertFrame(IplImage img)
{
  // blur image to get reduce camera noise
  cvSmooth(img, img, CV_BLUR, 3);

  // convert to grayscale
  IplImage grayImg = IplImage.create(img.width(), img.height(),
                                            IPL_DEPTH_8U, 1);
  cvCvtColor(img, grayImg, CV_BGR2GRAY);

  cvEqualizeHist(grayImg, grayImg);   // spread grayscale range

  return grayImg;
}  // end of convertFrame()
```

The blurring reduces the noise added to the image by the poor quality webcam. The conversion to grayscale makes subsequent difference and moment calculations easier, and equalization spreads out the grayscale's range of grays, making it easier to differentiate between different shades.

1.2. Detecting Movement

CVMotionDetector.calcMove() is passed the 'current' frame (the current webcam image) which it compares with the previous frame, detects differences, and uses them to calculate a center-of-gravity point.

```
// globals
private static final int LOW_THRESHOLD = 64;

private IplImage prevImg, currImg, diffImg;
private Point cogPoint = null;    // center-of-gravity (COG) coord
```



```
public void calcMove(IplImage currFrame)
{
  if (currFrame == null) {
    System.out.println("Current frame is null");
    return;
  }

  if (currImg != null) // store old current as the previous image
    prevImg = currImg;

  currImg = convertFrame(currFrame);

  cvAbsDiff(currImg, prevImg, diffImg);
    // calculate absolute diff between curr & previous images;
    // large value means movement; small value means no movement

  /* threshold to convert grayscale --> two-level binary:
           small diffs (0 -- LOW_THRESHOLD) --> 0
           large diffs (LOW_THRESHOLD+1 -- 255) --> 255   */
  cvThreshold(diffImg, diffImg, LOW_THRESHOLD, 255,
                                        CV_THRESH_BINARY);

  Point pt = findCOG(diffImg);
  if (pt != null)     // update COG if there is a new point
    cogPoint = pt;
}  // end of calcMove()
```

The current frame is converted to a grayscale by convertFrame(), and then compared with the previous frame using JavaCV's cvAbsDiff() function. It calculates the absolute intensity difference between corresponding pixels in the two grayscales, storing the results in the difference image, diffImg.

If the intensities of corresponding pixels haven't changed much between the previous and current frame, then their subtraction will produce a small value. However, a pixel that has changed radically (e.g. from black to white), will register a large intensity difference.

The resulting difference image is a grayscale containing a wide range of values, but it's possible to simplify the data using thresholding. cvThreshold() maps a specified range of intensities (LOW_THRESHOLD+1 (65) to 255) to a single value 255, while the rest of the intensity range (0 to LOW_THRESHOLD (64)) goes to 0. The result is a "bi-level" grayscale image – one that only contains 0's and 255's. I'm employing thresholding as a simple form of clustering, and saying that intensities above 65 are important, while those below can be ignored.

1.3. A Moment Spent on Moments

Before I look at findCOG(), it's necessary to explain moments. I'll employ them to obtain a mathematical expression for the center of a shape. The shape isn't something regular like a circle or square, but a collection of pixels which may be widely dispersed (e.g. like a cloud).

In physics, the moment M expresses how a force F operates at some distance d along a rigid bar from a fixed fulcrum. The idea is illustrated by Figure 5.

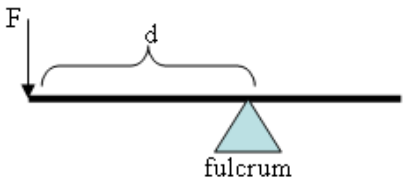

Figure 5. Force Acting at a Distance from a Fulcrum.

The equation for the moment is M = F*d

This concept can be generalized in numerous ways. For instance, rather than employ force, it's often convenient to talk about mass instead (denoted by a little m) so that gravity doesn't need to be considered. Also, the moment equation can be extended to multiple dimensions, to involve areas and volumes instead of just a one-dimensional bar.

The two-dimensional use of moments can be illustrated by the collection of *n* points shown in Figure 6.

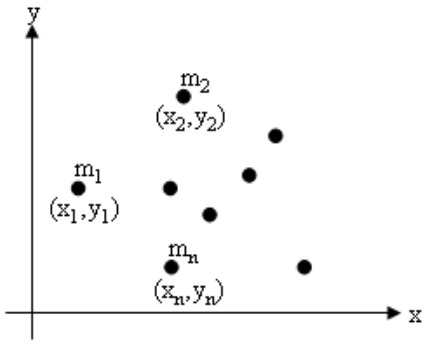

Figure 6. Points in Two-dimensions.

Each point has a mass (m_1, m_2,… , m_n) and an (x, y) coordinate ((x_1, y_1), (x_2, y_2),... , (x_n, y_n)). If we imagine that the points are attached by a rigid, weightless, frame to the x- and y- axes, then their moments can be calculated relative to those axes.

Considering the y-axis first, each point has a moment based on its mass*distance relative to the axis. For example, the moment of the point with mass m_1 with respect to the y-axis is m_1*x_1. The sum of all the points' moments around the y-axis is:

$$M_y = m_1*x_1 + m_2*x_2 + ... + m_n*x_n$$

The reasoning is the same for calculating the points' moments around the x-axis. The sum is:

$$M_x = m_1*y_1 + m_2*y_2 + ... + m_n*y_n$$

If we consider the collection of points, as a single 'system' then M_x and M_y can be read as the moments of the system around the x- and y-axes.

The total mass of the system is the sum of its point masses:

$m_{sys} = m_1 + m_2 + ... + m_n$

The basic moment equation is M = F*d in the one-dimensional case. This can be generalized to two-dimensions as:

$M_y = m_{sys} * \bar{x}$

and $M_x = m_{sys} * \bar{y}$

$\bar{x}$ and $\bar{y}$ can be viewed as the respective distances of the system from the x- and y-axes. In other words, the ($\bar{x}$, $\bar{y}$) coordinate is the system's 'center', which is often termed its *center-of-gravity* or *centroid*. Rearranging the above equations, gives us a way to calculate the center-of-gravity:

$$\bar{x} = \frac{M_y}{m_{sys}}$$

and

$$\bar{y} = \frac{M_x}{m_{sys}}$$

From Points to Pixels

The use of mass in the moment equations only makes sense when we're applying moments to real objects. When we use them in computer vision, the focus is on pixels rather than points, and the mass component can be replaced by a pixel function, such as its intensity (0-255 for a grayscale, 0-1 for a binary image).

Let's assume that each pixel has an intensity (I_1, I_2,… , I_n) and a (x, y) coordinate ((x_1, y_1), (x_2, y_2),... , (x_n, y_n)), as shown in Figure 7.

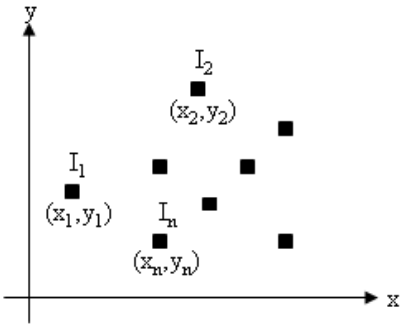

Figure 7. Pixels in Two-dimensions.

The sum of all the pixels' moments around the y-axis can be written as:

$M_y = I_1*x_1 + I_2*x_2 + ... + I_n*x_n$

The pixels' moments around the x-axis is:

$M_x = I_1*y_1 + I_2*y_2 + ... + I_n*y_n$

The 'system' is the collection of pixels, or the shape being studied.

The total intensity of the system (shape) is the sum of the intensities of its pixels:

$I_{sys} = I_1 + I_2 + ... + I_n$

Knowing I_{sys} and M_y allows us to obtain the distance of the shape from the y-axis:

$$\bar{x} = \frac{M_y}{I_{sys}}$$

In a similar way, the distance of the shape from the x-axis is:

$$\bar{y} = \frac{M_x}{I_{sys}}$$

The center-of-gravity point $(\bar{x}, \bar{y})$ is the shape's center.

1.4. Moments in JavaCV

JavaCV calculates different types of moments using the parameterized equation:

$m(p, q) = \sum_{i=1}^{n} I(x, y)\, x^p\, y^q$

The m() moments function takes two arguments, p and q, which are used as powers for x and y. The I() function is a generalization of my intensity notation, where the intensity for a pixel is defined by its (x, y) coordinate. *n* is the number of pixels that make up the shape.

The m() function is sufficient for calculating the center-of-gravity point $(\bar{x}, \bar{y})$ for a shape. Recall that $\bar{x} = \frac{M_y}{I_{sys}}$ and $\bar{y} = \frac{M_x}{I_{sys}}$, so the three sums we need are:

$M_y = I_1*x_1 + I_2*x_2 + ... + I_n*x_n$

$M_x = I_1*y_1 + I_2*y_2 + ... + I_n*y_n$

$I_{sys} = I_1 + I_2 + ... + I_n$

These can be expressed as versions of m() with different p and q values:

$M_y = m(1, 0)$

$M_x = m(0, 1)$

$I_{sys} = m(0, 0)$

This means that $(\bar{x}, \bar{y})$ can be expressed as:

$$\bar{x} = \frac{m(1,0)}{m(0,0)}$$

and

$$\bar{y} = \frac{m(0,1)}{m(0,0)}$$

findCOG() uses JavaCV's m() function to calculate the center-of-gravity point $(\bar{x}, \bar{y})$, which it returns as a Point object.

All the moments are calculated at once by a call to JavaCV's cvMoments() function, which stores them inside a CvMoments object. The ones needed for the center-of-gravity calculation (m(0,0), m(1,0), m(0,1)) are retrieved by calling cvGetSpatialMoment() with the necessary p and q values.


```
// globals
private static final int MIN_PIXELS = 100;
     // min number of non-black pixels for COG calculation


private Point findCOG(IplImage diffImg)
{
  Point pt = null;

  int numPixels = cvCountNonZero(diffImg);
  if (numPixels > MIN_PIXELS) {
    CvMoments moments = new CvMoments();
    cvMoments(diffImg, moments, 1);
          // 1 == treat image as binary (0,255) --> (0,1)
    double m00 = cvGetSpatialMoment(moments, 0, 0) ;
    double m10 = cvGetSpatialMoment(moments, 1, 0) ;
    double m01 = cvGetSpatialMoment(moments, 0, 1);

    if (m00 != 0) {   // create COG Point
      int xCenter = (int) Math.round(m10/m00);
      int yCenter = (int) Math.round(m01/m00);
      pt = new Point(xCenter, yCenter);
    }
  }
  return pt;
}  // end of findCOG()
```

cvMoments() calculates many types of moments (many more than I've explained), and one optimization is to simplify the intensity relation it uses. Since the difference image, diffImg, is a bi-level grayscale, it can be treated as a binary image with intensities of 0 and 1. This is signaled to JavaCV by calling cvMoments() with its third argument set to 1:

```
cvMoments(diffImg, moments, 1);  // treat image as a binary
```

Another optimization is to check the number of non-zero pixels in diffImg before calculating the moments. A difference is denoted by an intensity of 255, and no change by 0, so the counting of non-zeros is a way to gauge the amount of difference in diffImg. If the difference is too small (less than MIN_PIXELS are non-zero) then the moments calculation is skipped.

2. The GUI-based Motion Detector

The class structure of the MotionDetector application is summarized in Figure 8.

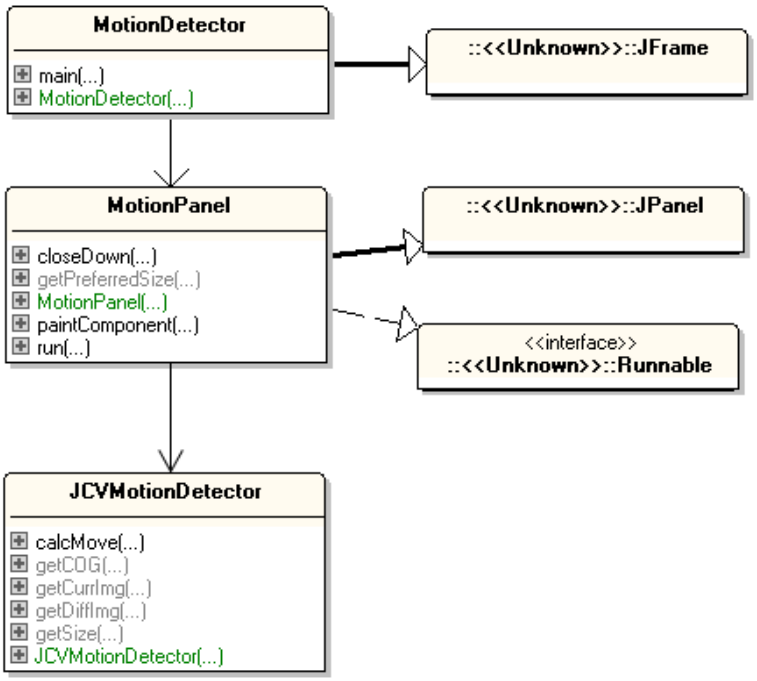

Figure 8. Class Diagrams for the MotionDetector Application.

I won't bother explaining the top-level MotionDetector class – it's just like the SnapPics JFrame developed in Chapter 2, but simpler since I've removed the code for saving a snap when the user presses numpad-5, space, or enter.

MotionPanel is similar to the Chapter 2's PicsPanel class, in that it spends much of it's time inside a threaded loop repeatedly grabbing an image from the webcam (with JavaCV's FrameGrabber) and drawing it onto the panel until the window is closed. Consequently, I won't explain the grabber-related methods again.

MotionPanel also draws crosshairs, centered on the current center-of-gravity point of the detected motion. This position is calculated using a slightly modified version of CVMotionDetector from the previous section (now called JCVMotionDetector).

2.1. The Webcam Processing Loop

MotionPanel executes the webcam display loop inside run(), just as in Chapter 2 except for the use of JCVMotionDetector. A JCVMotionDetector object is created before the loop starts, and JCVMotionDetector.calcMove() and JCVMotionDetector.getCOG() are called during each loop iteration:

```
// in MotionPanel
// globals
private static final int DELAY = 100;
                  // time (ms) between redraws of the panel
```



```
private static final int CAMERA_ID = 0;

private IplImage snapIm = null;  // current webcam snap
private volatile boolean isRunning;

// used for the average ms snap time information
private int imageCount = 0;
private long totalTime = 0;

// previous and current center-of-gravity points
private Point prevCogPoint = null;
private Point cogPoint = null;


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

  snapIm = picGrab(grabber, CAMERA_ID);
  JCVMotionDetector md = new JCVMotionDetector(snapIm);

  Point pt;
  long duration;
  isRunning = true;

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

    snapIm = picGrab(grabber, CAMERA_ID);

    md.calcMove(snapIm);    // update detector with new image
    if ((pt = md.getCOG()) != null) {    // get new COG
      prevCogPoint = cogPoint;
      cogPoint = pt;
      reportCOGChanges(cogPoint, prevCogPoint);
    }

    imageCount++;
    repaint();

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

The center-of-gravity point is stored in the cogPoint global, and the previous value is backed-up in prevCogPoint. Both these objects are passed to reportCOGChanges() so that changes in the center-of-gravity can be reported.

2.2. Reporting on the Center-of-gravity

The reportCOGChanges() method prints out the current center-of-gravity if it has moved sufficiently far from its previous position, and prints the distance moved (in pixels) and its angle (in degrees) relative to the old point. Some typical output:

```
COG: (614, 371)
  Dist moved: 39; angle: 94
COG: (612, 341)
  Dist moved: 30; angle: 94
COG: (614, 315)
  Dist moved: 26; angle: 86
COG: (614, 303)
  Dist moved: 12; angle: 90
COG: (609, 319)
  Dist moved: 16; angle: -105
```

The reportCOGChanges() code:

```
// global
private static final int MIN_MOVE_REPORT = 3;


private void reportCOGChanges(Point cogPoint, Point prevCogPoint)
{
  if (prevCogPoint == null)
    return;

  // calculate the distance moved and the angle (in degrees)
  int xStep = cogPoint.x - prevCogPoint.x;
  int yStep = -1 *(cogPoint.y - prevCogPoint.y);
                                  // so + y-axis is up screen

  int distMoved = (int) Math.round(
                 Math.sqrt( (xStep*xStep) + (yStep*yStep)) );
  int angle = (int) Math.round( Math.toDegrees(
                                  Math.atan2(yStep, xStep)) );

  if (distMoved > MIN_MOVE_REPORT) {
    System.out.println("COG: (" + cogPoint.x + ", " +
                                   cogPoint.y + ")");
    System.out.println("  Dist moved: " + distMoved +
                             "; angle: " + angle);
  }
}  // end of reportCOGChanges()
```

2.3. Rendering Motion Detection

Figure 1 shows that the panel only contains three elements: the webcam image in the background, a crosshairs image, and statistics written in yellow at the bottom left corner.

All rendering is done through calls to the panel's paintComponent():

```
// globals
private IplImage snapIm;  // current webcam snap
```



```
private Point cogPoint;


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

  // draw the image, crosshairs, and stats
  if (snapIm != null) {
    g.drawImage(snapIm.getBufferedImage(), 0, 0, this);

    if (cogPoint != null)
      drawCrosshairs(g, cogPoint.x, cogPoint.y);
                                    // image is centered at COG
    g.setColor(Color.YELLOW);
    String statsMsg = String.format("Snap Avg. Time:  %.1f ms",
                         ((double) totalTime / imageCount));
    g.drawString(statsMsg, 5, HEIGHT-10);
  }
  else  {// no image yet
    g.setColor(Color.BLUE);
    g.drawString("Loading from camera " + CAMERA_ID +
                                    "...", 5, HEIGHT-10);
  }
} // end of paintComponent()
```

drawCrosshairs() draws a pre-loaded PNG image (see Figure 9) so it's centered at the center-of-gravity coordinates.

Figure 9. The Crosshairs Image.

3. Smoothing the Motion Detection

JCVMotionDetector differs from CVMotionDetector in only one way: the addition of smoothing to the center-of-gravity point. In the first version of calcMove(), a point was stored in a global called cogPoint, and retrieved by calls to getCOG(). In JCVMotionDetector, calcMove() adds the new point to an *array*. This array includes the last few center-of-gravity points as well as the current one:

```
// in JCVMotionDetector
// globals
private static final int LOW_THRESHOLD = 64;
private static final int MAX_PTS = 5;  //size of smoothing array

private IplImage prevImg, currImg, diffImg;
private Point[] cogPoints;   // array for smoothing COG point
private int ptIdx, totalPts;

// in the JCVMotionDetector constructor:
```



```
cogPoints = new Point[MAX_PTS];
ptIdx = 0;
totalPts = 0;


public void calcMove(IplImage currFrame)
// smoothing version
{
  if (currFrame == null) {
    System.out.println("Current frame is null");
    return;
  }

  if (currImg != null)  // store old current as the previous image
    prevImg = currImg;

  currImg = convertFrame(currFrame);
  cvAbsDiff(currImg, prevImg, diffImg);
  cvThreshold(diffImg, diffImg, LOW_THRESHOLD, 255,
                                        CV_THRESH_BINARY);
  Point cogPoint = findCOG(diffImg);

  if (cogPoint != null) {    // store in points array
    cogPoints[ptIdx] = cogPoint;
    ptIdx = (ptIdx+1)%MAX_PTS;   // index cycles around array
    if (totalPts < MAX_PTS)
      totalPts++;
  }
}  // end of calcMove()
```

A new point is added to a fixed-size array (5 elements in my code), so there's an upper-bound on the number of 'old' points which can be stored. When the array is full, a new point replaces the oldest one.

getCOG() is also changed – instead of returning a cogPoint global, it calculates an average point from the values stored in the array:

```
public Point getCOG()
{
  if (totalPts == 0)
    return null;

  int xTot = 0;
  int yTot = 0;
  for(int i=0; i < totalPts; i++) {
    xTot += cogPoints[i].x;
    yTot += cogPoints[i].y;
  }
  return new Point( (int)(xTot/totalPts), (int)(yTot/totalPts));
}  // end of getCOG()
```

This code implements a form of smoothing since it averages the current and previous centers-of-gravity, which reduces variations in its position over time. Unfortunately, this also means that if the user moves quickly, then the crosshairs movement will lag behind. This lag time can be shortened by reducing the size of the smoothing array, thereby reducing the number of old points that affect the current one.

4. Background/Foreground Segmentation

OpenCV supports background/foreground segmentation via two Mixture of Gaussians (MOG) implementations, a widely used form of background modeling that allows a static camera to detect moving objects. The APIs for the BackgroundSubtractorMOG class, and the somewhat faster BackgroundSubtractorMOG2, can be found at http://docs.opencv.org/modules/video/doc/motion_analysis_and_object_tracking.htm. The documentation includes links to the papers that proposed the algorithms. Another source of information are the comments in the C++ header file for these functions, opencv\build\include\opencv2\video\background_segm.hpp, which comes with the OpenCV download.

The essential idea of both algorithms is to use a history of video frames to segment the scene into several Gaussian distributions (or clusters) representing the background, foreground, and shadows. The darkest pixel areas are labeled as shadow, the regions that are changing the most are added to the foreground Gaussian cluster (or clusters), and the areas changing the least are assigned to the background.

This description highlights the somewhat misleading use of the words 'foreground' and 'background' in the algorithm, which suggests a depth-measuring aspect which isn't there. A pixel area is added to the 'foreground' if it has changed between frames, which typically denotes movement in the scene. Conversely, 'background' areas are those which remain unchanged between frames over a period of time.

Figure 10 shows my running MogCog.java application, which utilizes two CanvasFrame displays in a similar way to the motion detector test-rig back in Figure 2.

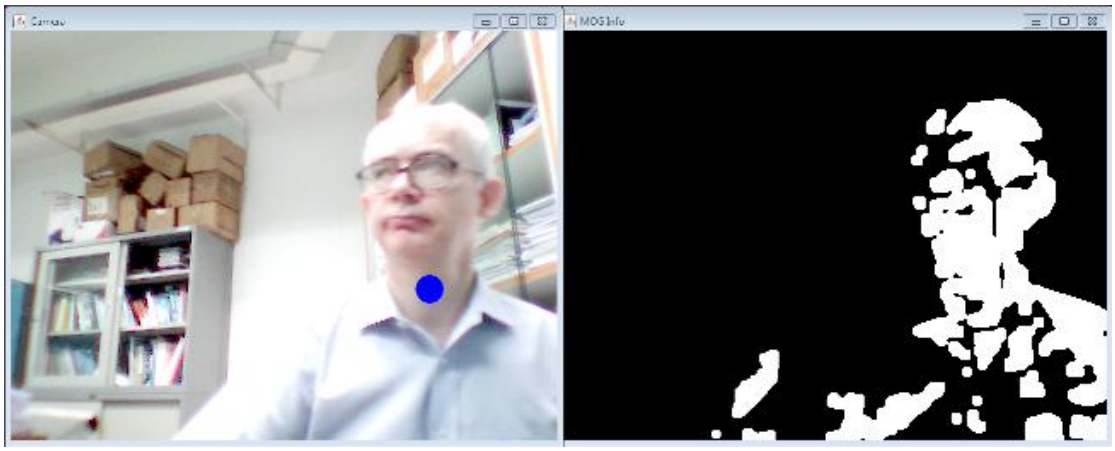

Figure 10. The MogCog Detector.

The left-hand canvas shows the current grabbed image, and adds a large blue circle at the center-of-gravity (COG) of the foreground (i.e. moving) region. The right-hand canvas displays the foreground mask generated using the BackgroundSubtractorMOG2 class.

One difference between the image difference display in Figure 2 and the foreground mask in Figure 10 is that the mask covers more of the head while the difference image

only highlights it's left and right edges. This is achieved by adjusting the parameters supplied to the BackgroundSubtractorMOG2 object, which would otherwise only display similar edge information.

The main() method for MogCog spends a little time initializing the BackgroundSubtractorMOG2 instance, and then enters a timed loop which repeatedly grabs a camera image, and extracts a COG from the foreground mask.

```
// globals
private static final int DELAY = 100;    // ms

private static CvMemStorage contourStorage;



public static void main(String[] args) throws Exception
{
  // Preload the opencv_objdetect module to work around a known
  Loader.load(opencv_objdetect.class);

  contourStorage = CvMemStorage.create();

  System.out.println("Initializing frame grabber...");
  OpenCVFrameGrabber grabber = new OpenCVFrameGrabber(CV_CAP_ANY);
  grabber.start();

  IplImage grab = grabber.grab();
  int width  = grab.width();
  int height = grab.height();
  IplImage fgMask = IplImage.create(width, height, IPL_DEPTH_8U, 1);
  IplImage background = IplImage.create(width, height,
                                        IPL_DEPTH_8U, 3);

  CanvasFrame grabCanvas = new CanvasFrame("Camera");
  grabCanvas.setLocation(0, 0);
  CanvasFrame mogCanvas = new CanvasFrame("MOG Info");
  mogCanvas.setLocation(width+5, 0);

  BackgroundSubtractorMOG2 mog =
                 new BackgroundSubtractorMOG2(300, 16, false);
  mog.set("nmixtures", 3);    // was 5
  System.out.println("Num. mixtures: " + mog.getInt("nmixtures"));
  System.out.println("Shadow detect: " +
                                  mog.getBool("detectShadows"));
  try {
    System.out.println("Background ratio: " +
                               mog.getDouble("backgroundRatio"));
  }
  catch (RuntimeException e)
  {  System.out.println(e);  }

  // process the grabbed camera image
  while (grabCanvas.isVisible() && mogCanvas.isVisible()) {
    long startTime = System.currentTimeMillis();
    grab = grabber.grab();
    if (grab == null) {
      System.out.println("Image grab failed");
      break;
    }
```



```
    mog.apply(grab, fgMask, 0.005);  // get foreground mask
    mog.getBackgroundImage(background);

    // reduce noise in mask
    cvErode(fgMask, fgMask, null, 5);
    cvDilate(fgMask, fgMask, null, 5);
    cvSmooth(fgMask, fgMask, CV_BLUR, 5);
    cvThreshold(fgMask, fgMask, 128, 255, CV_THRESH_BINARY);

    mogCanvas.showImage(fgMask);         // show foreground mask
    // mogCanvas.showImage(background);  // show background

    Point pt = findCOG(fgMask);  // same method as earlier
    if (pt != null)   // draw the COG as a blue circle
      cvCircle(grab, cvPoint(pt.x, pt.y), 16,
                     CvScalar.BLUE, CV_FILLED, CV_AA, 0);
    grabCanvas.showImage(grab);

    long duration = System.currentTimeMillis() - startTime;
    System.out.println("Processing time: " + duration);
    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY - duration);
      }
      catch(InterruptedException e) {}
    }
  }

  grabber.stop();
  grabCanvas.dispose();
  mogCanvas.dispose();
}  // end of main()
```

The COG is calculated using findCOG() which is unchanged from the code in section 1.4. The only difference is that it utilizes the foreground mask (i.e. the black and white image in the right-hand canvas of Figure 10) rather than the difference image (the black and white image in the right-hand canvas of Figure 2). Consequentially, I won't explain the mathematics behind the moments calculations in findCOG() again.

The initialization of the BackgroundSubtractorMOG2 object is handled by:

```
BackgroundSubtractorMOG2 mog =
                 new BackgroundSubtractorMOG2(300, 16, false);
mog.set("nmixtures", 3);
```

The three parameters in the constructor are the length of the frame history, a distance measure for the size of the each Gaussian cluster, and a boolean indicating whether shadows should be collected as a cluster.

The large frame history means that the Gaussian cluster for the background will be based on a lot of historical information (assuming that the background is fairly static), and so it should be easier for the algorithm to distinguish movement in the scene.

The number of mixtures (the nmixtures parameter) sets the number of clusters used to distinguish the background and foreground elements. I've reduced the value from the default of 5 to 3 since I'm not collecting shadow information.

In OpenCV version 2.4.2 only the detectShadows, history, and nmixtures parameters can be set/get, although this was fixed in v.2.4.6. Just to be on the safe side, I surround the get() call for the background ratio in a try-catch block to deal with a potential runtime exception.

The assignment of the pixels in an image to different Gaussian clusters is carried out by the apply() method, which is called inside the while-loop after a new image has been grabbed:

```
mog.apply(grab, fgMask, 0.005);
```

The third parameter (0.005) is a learning rate, which is often set to -1, indicating a default rate at which objects that have stopped moving in the foreground are reassigned to the background cluster. I've set the learning rate close to 0, to slow down this reassignment.

Combined with this low learning rate, I also ensure that the while-loop is provided with several seconds of views of the background with no foreground objects. This supplies the algorithm with a history of video frames for it to learn the background. This process can be seen in action by observing changes to the right-hand display canvas which shows the foreground mask. When the application starts, it is all white, indicating that everything is considered to be in the foreground. After a few seconds the mask turns completely black, signifying that all of the scene has been reassigned to the background cluster.

The image assigned to fgMask is a 8-bit binary image which often contains quite a bit of noise, and so the calls to cvErode(), cvDilate(), cvSmooth(), and cvThreshold() in main() are intended to remove those small specks and combine adjacent white regions into larger blobs.

main() contains code for accessing the background image generated by BackgroundSubtractorMOG2:

```
IplImage background =
       IplImage.create(width, height, IPL_DEPTH_8U, 3);
  :
mog.getBackgroundImage(background);
  :
mogCanvas.showImage(fgMask);          // show foreground mask
// mogCanvas.showImage(background);  // show background
```

If the display code for the right-hand canvas is replaced to show the background image rather than the foreground mask, then the MogCog application will look something like Figure 11.

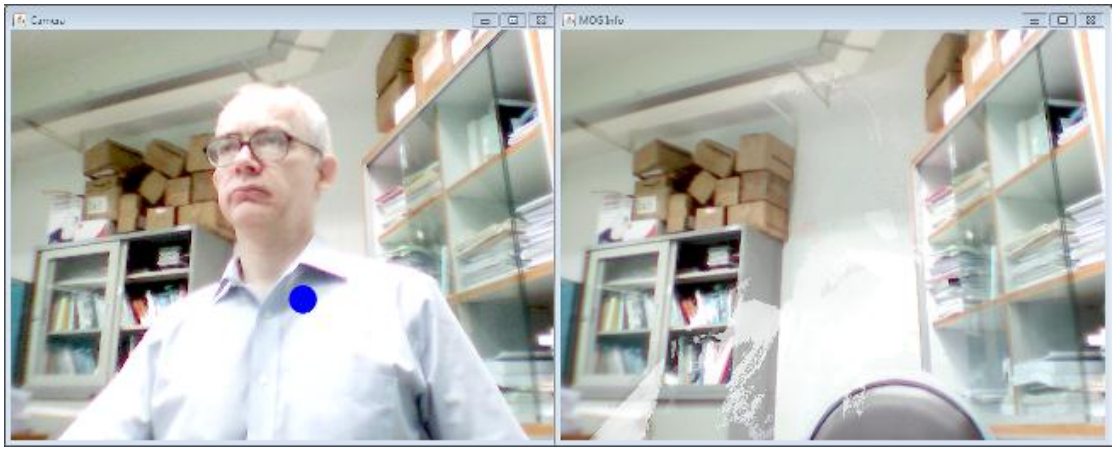

Figure 11. MogCog Showing the Background Image.

The ghostly white parts of the right-hand image demonstrate how parts of my white shirt are gradually being reassigned from the foreground to the background.

Another difference from the difference image test-rig described at the start of this chapter is the use of Thread.sleep() inside main()'s while-loop. This implements a simple timed loop which ensures that each image grab occurs after 100 ms or more. This reduces the looping speed since a single iteration without any delay takes around 80 ms to complete on my slow test machine.

5. Movement Detection Using Optical Flow

Optical flow tracks the movement of pixel-level elements across consecutive frames in a video stream. In *dense* optical flow, the tracking tries to follow every pixel, which can be rather tricky if the picture lacks detail (e.g. a polar bear dancing in a snow storm). *Sparse* optical flow concentrates on following pixels that are easy to identify (also called features or corners) due to their difference in intensity, texturing, or color from neighboring pixels.

OpenCV supports a number of techniques for findings corners; I'll be using an algorithm due to Shi and Tomasi, implemented as the function cvGoodFeaturesToTrack(), which selects pixels surrounded by rapidly changing intensities.

I'll obtain the optical flow of these corners using the cvCalcOpticalFlowPyrLK() function. It implements the pyramidal Lucas-Kanade (LK) method, which uses level-of-detail pyramids to detect varying amounts of movement between consecutive frames.

Chapter 10 ("Tracking and Motion") of *Learning OpenCV* by Bradski and Kaehler contains a great introduction to these methods, including details on the maths behind the LK method. My code is partly based on their example and also code from http://www.ccs.neu.edu/course/cs7380/f10/HW2/examples/c/lkdemo.c. A briefer, slide introduction to these techniques by David Stavens can be found at http://ai.stanford.edu/~dstavens/cs223b/, along with source code.

My OpticalFlowMove application is shown in action in Figure 12.

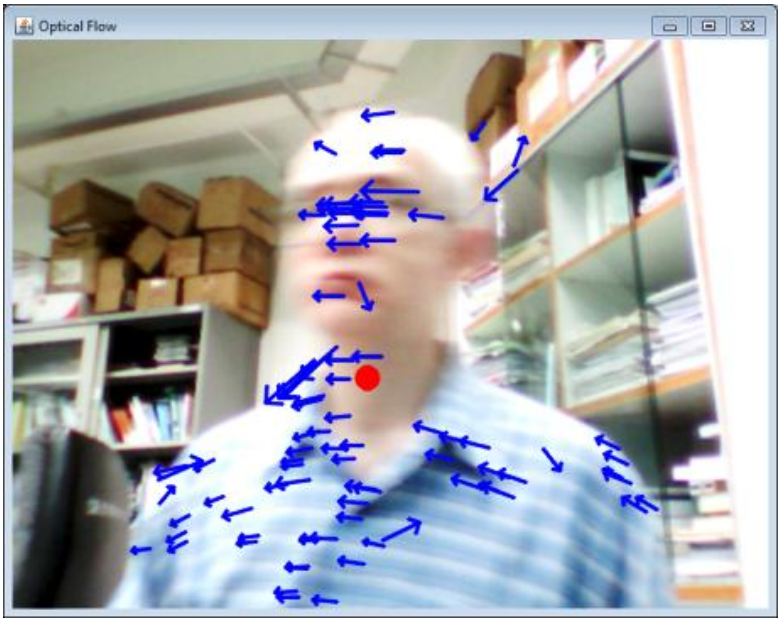

Figure 12. Displaying Optical Flow.

The movement of the selected corners between the previous frame and the current one are drawn as blue arrows.

Unlike the movement tracking approaches from earlier in this chapter, optical flow isn't represented by a single blob of pixels but as multiple individual directions (vectors). This presents a number of problems when trying to determine an average center-of-gravity (COG) for the movement. For example, although Figure 12 shows that the user is moving right-to-left, the directions span a wide range of angles. Also, there are quite a few directions which are 'incorrect', due to the misidentification of corners.

My solution is to store the directions in an array of lists, with each list containing directions ranging over a small number of angles. My code uses a 20 degrees range for each list, resulting in an array of at most 18 lists. The largest list is used to calculate the COG, and the smaller lists are ignored. This approach is satisfactory if the scene contains a single overriding direction of movement (e.g. as in Figure 12). It's less ideal if there are multiple sources of movement, such as two people walking past each other.

OpticalFlowMove.java consists of a lengthy main() function and several support methods. main() starts by initializing a large number of data structures required by the OpenCV methods, but its heart is a loop which can be summarized like so:

```
prevGray = grab an image;

while (display window is visible) {
  grayGrab = grab an image;

  cornersA = select corners from prevGray using
             cvGoodFeaturesToTrack();

  cornersB = find those same corners in grayGrab using
```



```
            cvCalcOpticalFlowPyrLK();

  draw cornersA-to-cornersB directions as blue arrows;
  prevGray = grayGrab;
}
```

The data structure initialization, the creation of a FrameGrabber, and the initial grabbing of an image is coded like so:

```
// globals
private static final int MAX_CORNERS = 300;

// group directions into lists based on their angles
private static final int ANGLE_RANGE = 20;
                                    // degree range for each list
private static final int NUM_ANGLE_LISTS = 360/ANGLE_RANGE;
                                    // number of lists

private static int width, height;   // of a grabbed image


public static void main(String[] args) throws Exception
{
  // work around a known JavaCV bug
  Loader.load(opencv_objdetect.class);

  System.out.println("Initializing frame grabber...");
  OpenCVFrameGrabber grabber = new OpenCVFrameGrabber(CV_CAP_ANY);
  grabber.start();

  CanvasFrame grabCanvas = new CanvasFrame("Optical Flow");

  IplImage grab = grabber.grab();
      // grab an image for the initial 'previous' frame
  width = grab.width();
  height = grab.height();
  grabCanvas.showImage(grab);

  IplImage prevGray = toGray(grab);

  // (image) buffers for corner selection
  IplImage eigenIm = IplImage.create(width, height,
                                     IPL_DEPTH_32F, 1);
  IplImage tempIm = IplImage.create(width, height,
                                     IPL_DEPTH_32F, 1);

  // (image) buffers for the level-of-detail pyramids
  IplImage pyramidA = IplImage.create(width+8, height/3,
                                     IPL_DEPTH_32F, 1);
  IplImage pyramidB = IplImage.create(width+8, height/3,
                                     IPL_DEPTH_32F, 1);
  // corners arrays
  int[] cornerCount = { MAX_CORNERS };
  CvPoint2D32f cornersA = new CvPoint2D32f(MAX_CORNERS);
  CvPoint2D32f cornersB = new CvPoint2D32f(MAX_CORNERS);
  boolean findCorners = true;

  // selected corners info arrays
  byte[] cornersFound = new byte[MAX_CORNERS];
```



```
  float[] trackErrs = new float[MAX_CORNERS];

  int plk_flags = 0;   // Lucas-Kanade method flags

  CvPoint cogBall = null;
     // null means that no COG will be displayed

  // direction lists array
  @SuppressWarnings("unchecked")
            // avoid warnings about using generics in arrays
  ArrayList<Direction>[] angledDirs = new ArrayList[NUM_ANGLE_LISTS];
  for (int i=0; i < NUM_ANGLE_LISTS; i++)
    angledDirs[i] = new ArrayList<Direction>();


  // enter the processing loop; see below
  //    :
  }  // end of main()
```

The angledDirs[] array holds the lists of Direction objects; each Direction instance is created from a pair of corresponding corners found in the previous and current images.

The selection of corners uses intensities differences between a pixel and its neighbors, so each grabbed image is converted into an equalized grayscale by toGray() before the processing begins:

```
private static IplImage toGray(IplImage img)
{
  // blur image to get reduce camera noise
  cvSmooth(img, img, CV_BLUR, 3);

  // convert to grayscale
  IplImage grayImg = IplImage.create(img.width(), img.height(),
                                     IPL_DEPTH_8U, 1);
  cvCvtColor(img, grayImg, CV_BGR2GRAY);

  cvEqualizeHist(grayImg, grayImg);
                         // spread out the grayscale range
  return grayImg;
}  // end of toGray()
```

The second half of main() contains the processing loop, shown below:

```
// globals
private static final int DELAY = 100;    // ms
private static final int WIN_SIZE = 10;   // size of search window


public static void main(String[] args) throws Exception
{
  // initialization of data structures; see above
  //  :

  // process the grabbed camera image
  while (grabCanvas.isVisible() &&
         ((grab = grabber.grab()) != null)) {
    long startTime = System.currentTimeMillis();
```



```
IplImage grayGrab = toGray(grab);

// find corners
cornersA.position(0);            // reset position in array
if (findCorners)
  cvGoodFeaturesToTrack(prevGray, eigenIm, tempIm,
           cornersA, cornerCount, 0.01, 5, null, 3, 0, 0.04);

cornersA.position(0);   // reset position in arrays
cornersB.position(0);
/* calculate new positions of the corners (in cornersB)
   based on how the corners in cornersA move between
   the previous image (prevGray) and the
   current image (grayGrab)
*/
cvCalcOpticalFlowPyrLK(prevGray, grayGrab,
    pyramidA, pyramidB,
    cornersA, cornersB, cornerCount[0],
    cvSize(WIN_SIZE,WIN_SIZE), 5,
    cornersFound, trackErrs,
    cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 20, 0.3),
    plk_flags);

// store the corner pairs (A --> B) as 'directions'
int numDirs = storeDirs(cornersA, cornersB, cornerCount[0],
                        cornersFound, angledDirs, grab);

cogBall = updateCOG(cogBall, angledDirs, numDirs);
if (cogBall != null)
  cvCircle(grab, cogBall, 10, CvScalar.RED, CV_FILLED, CV_AA, 0);

grabCanvas.showImage(grab);
prevGray = grayGrab;
      // save image as previous for next time around

if (numDirs > 0) {
  /* swap corners and pyramids data to speed up
     the next call to cvCalcOpticalFlowPyrLK() */
  CvPoint2D32f swapCorners = cornersA;
  cornersA = cornersB;
      // no need to recalculate cornersA next time around
  cornersB = swapCorners;
  findCorners = false;

  IplImage swapPyramid = pyramidA;
  pyramidA = pyramidB;
       // use pyramidB as the 'A' pyramid next time around
  pyramidB = pyramidA;
  plk_flags |= CV_LKFLOW_PYR_A_READY;  // pyramid 'A' initialized
}
else {  // no directions found
  // call cvGoodFeaturesToTrack() next time to improve matters
  findCorners = true;
  cornerCount[0] = MAX_CORNERS;
  plk_flags = 0;
}

long duration = System.currentTimeMillis() - startTime;
System.out.println("  Processing time: " + duration);
if (duration < DELAY) {
  try {
```



```
        Thread.sleep(DELAY - duration);
      }
      catch(InterruptedException e) {}
    }
  }  // end of processing loop

  grabber.stop();
   grabCanvas.dispose();
}  // end of main()
```

The call to cvGoodFeaturesToTrack() is:

```
cvGoodFeaturesToTrack(prevGray, eigenIm, tempIm,
    cornersA, cornerCount, 0.01, 5, null, 3, 0, 0.04);
```

The grayscale image in prevGray is examined, and suitable corners are stored in the cornersA array, along with the number of those corners in cornerCount.

The two arguments of cvGoodFeaturesToTrack() which most effect the subsequent speed and accuracy of the optical flow calculation are the minimum quality level for selecting a corner (which is set to 0.01 in the code above) and the minimum distance between corners (set to 5 pixels above).

The smaller the quality level, the smaller the intensity change necessary to classify a pixel as a corner. This will increase the total number of corners, which will slow down the subsequent flow processing.

One way to limit the number of corners is to reduce the MAX_CORNERS constant (currently set at 300) which puts an upper bound on the number of corners that are stored in the corners arrays.

If cvGoodFeaturesToTrack()'s minimum pixel distance is increased (e.g. from 5 to 10), then there will be more space between the selected corners, which will reduce their total number.

A full description of all the arguments for cvGoodFeaturesToTrack() can be found on the OpenCV API documentation page http://docs.opencv.org/modules/video/doc/motion_analysis_and_object_tracking.html , or on p.318 of the first edition of *Learning OpenCV* by Bradski and Kaehler.

The call to the cvCalcOpticalFlowPyrLK() is:

```
cvCalcOpticalFlowPyrLK(prevGray, grayGrab,
    pyramidA, pyramidB,
    cornersA, cornersB, cornerCount[0],
    cvSize(WIN_SIZE,WIN_SIZE), 5,
    cornersFound, trackErrs,
    cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 20, 0.3),
    plk_flags);
```

The algorithm calculates the new positions of the previous image's (prevGray) corners relative to the current image (grayGrab). The previous image corners are read from the cornersA array, and the newly calculated positions are stored in cornersB.

A side-effect is the creation of two level-of-detail pyramids (pyramidA and pyramidB) which are used when calculating the movement of the corners. The success (or failure) of matching the corners in cornersA to new positions in cornersB is recorded in the cornersFound and trackErrs arrays. cornersFound will contain a sequence of 1's and 0's denoting success or failure for tracking each corner, while trackErrs gives a numerical value for the distance between the paired corners (smaller is better).

The cvTermCriteria term in cvCalcOpticalFlowPyrLK() specifies termination criteria for ending the flow calculation, including the maximum number of iterations of the algorithm (20) and an epsilon value (0.3) representing the required accuracy of the results.

The easiest way to affect cvCalcOpticalFlowPyrLK()'s speed is to adjust the window size (WIN_SIZE is 10 in the above code) which represents the number of neighboring pixels considered. Another parameter to adjust is the number of pyramid levels (5). Reducing either will make the algorithm faster, at the expense of accuracy.

A full description of all the arguments for cvCalcOpticalFlowPyrLK() can be found on the OpenCV API documentation page http://docs.opencv.org/modules/video/doc/motion_analysis_and_object_tracking.html , or on p.330-331 of the first edition of *Learning OpenCV*.

5.1. Reusing Corners Information

At the end of the processing loop, the current image is assigned to the previous image variable (prevGray), ready for the next iteration. It's also possible to reuse corners and pyramid data in a similar way.

If the findCorners boolean is set to true (as it is in the first iteration of the processing loop), then cvGoodFeaturesToTrack() is called to assign corners to the cornersA array. This is not necessary for subsequent iterations since a side-effect of cvCalcOpticalFlowPyrLK() is to generate corners for the current image, stored in cornersB. This data can be reassigned to the cornersA variable at the same time that the current image becomes the new previous picture. This means that it is unnecessary to call cvGoodFeaturesToTrack() during the next loop, thereby saving quite a lot of processing time.

A similar trick can be applied to the pyramid data employed in cvCalcOpticalFlowPyrLK(). For the first iteration, the method needs to calculate level-of-detail pyramids for both the previous and current images, which are stored as pyramidA and pyramidB. However, on subsequent iterations, pyramidA can be assigned the pyramidB data from the previous loop. cvCalcOpticalFlowPyrLK() is informed on this optimization by having its plk_flags argument set to CV_LKFLOW_PYR_A_READY.

After applying both optimizations, the average execution time for an iteration of the processing loop drops from around 80 ms to about 40 ms on my slow test machine.

However, care must be taken when reusing the corners and pyramid data since it means that any poor corners tracking by cvCalcOpticalFlowPyrLK() will be retained in subsequent loops, leading to a gradual worsening of the corners results. Visually this will result in fewer direction arrows being drawn on the image as time progresses. One fix is to monitor the number of direction vectors (via the numDirs variable), and

if it drops to 0 then the processing loop should 'reinstate' calls to cvGoodFeaturesToTrack() and have cvCalcOpticalFlowPyrLK() create both pyramids.

The numDirs variable is set when my code creates direction vectors from the corners information. If your application doesn't do this, then it's possible to monitor the number of corners, through the cornerCount variable. But this is a less reliable measure since the number of corners hardly ever drops to 0 because noise in the images ensures that a few 'fake' corners are always detected.

5.2. From Corners to Directions

My storeDirs() method uses each corresponding corner coordinate in cornersA and cornersB to create a Direction object, and adds the directions to the relevant lists in the angledDirs[] array.

```
// globals
private static final int ANGLE_RANGE = 20;
                              // degree range for each list
private static final int NUM_ANGLE_LISTS = 360/ANGLE_RANGE;
                              // number of lists

private static final int MIN_DIR_LENGTH = 15;


private static int storeDirs(CvPoint2D32f cornersA,
                             CvPoint2D32f cornersB,
               int cornerCount, byte[] cornersFound,
               ArrayList<Direction>[] angledDirs, IplImage grab)
{
  for (int i=0; i < NUM_ANGLE_LISTS; i++)    // clear array's lists
    angledDirs[i].clear();
  int numDirs = 0;

  cornersA.position(0);   // reset position in corner arrays
  cornersB.position(0);
  int numErrs = 0;
  int numLongs = 0;
  int numShorts = 0;

  // save corners as directions
  for(int i = 0; i < cornerCount; i++) {
    if (cornersFound[i] == 0)    // not found
      numErrs++;
    else {
      cornersA.position(i);
      cornersB.position(i);
      Direction dir = new Direction(cornersA, cornersB);

      /* if a direction's length is within a 'good' range,
         then store and draw it */
      double lenDir = dir.getLength();
      if (lenDir > width/8)
        numLongs++;
      else if (lenDir < MIN_DIR_LENGTH)
        numShorts++;
      else {
        dir.drawArrow(grab); // draw direction on grabbed image
```



```
          addDir(angledDirs, dir);
          numDirs++;
        }
      }
    }
/*   // useful during debugging
  if (numErrs > 0)
    System.out.println("No. of feature errors: " + numErrs);
  if (numLongs > 0)
    System.out.println("No. of too long dirs: " + numLongs);
  if (numShorts > 0)
    System.out.println("No. of too short dirs: " + numShorts);
*/
  return numDirs;
}  // end of storeDirs()
```

storeDirs() filters out a significant number of corners, thereby reducing the number of directions. If the cornersFound[] value for a corner is 0, then cvCalcOpticalFlowPyrLK() was unable to track that corner between the two frames, and so no Direction object is made. Additionally, the length of each Direction instance is tested to see if it is either too short or too long. The assumption is that directions with 'abnormal' lengths are caused by noise or incorrect tracking, and so shouldn't be added to the angledDirs[] array or drawn on the image.

The angledDirs[]'s direction lists are divided into angle ranges of 20 degrees. Consequently, addDir() method uses the direction's angle to determine which list should be used:

```
//globals
private static final int ANGLE_RANGE = 20;
private static final int NUM_ANGLE_LISTS = 360/ANGLE_RANGE;


private static void addDir(ArrayList<Direction>[] angledDirs,
                                                  Direction dir)
{ int angleIndex = (dir.getAngle()+180)/ANGLE_RANGE;
  if (angleIndex == NUM_ANGLE_LISTS)
    angleIndex = 0;
  angledDirs[angleIndex].add(dir);
}  // end of addDir()
```

Direction.getAngle() returns an integer degree value between -180 and 180, which is mapped to a positive range (0 to 360) before being scaled using ANGLE_RANGE. The result is used as an index into the array to determine which list to utilize.

5.3. Representing a Direction

The Direction class stores the two corners supplied to its constructor as CvPoint objects, and calculates the length and angle between those points.

```
// globals
private CvPoint p0, p1;
private double length;
private double angle;   // in radians
```



```
public Direction(CvPoint2D32f cornersA, CvPoint2D32f cornersB)
{
  p0 = cvPoint(Math.round(cornersA.x()), Math.round(cornersA.y()));
  p1 = cvPoint(Math.round(cornersB.x()), Math.round(cornersB.y()));

  double xDist = p1.x() - p0.x();
  double yDist = p1.y() - p0.y();

  length = Math.sqrt((xDist*xDist) + (yDist*yDist));
  angle = Math.atan2(yDist, xDist);
}  // end of Direction()
```

Direction.drawArrow() draws a blue arrow representing the direction onto the supplied image. The code uses three calls to cvLine() – one for the arrow body, and two shorter lines for the branches of the arrow head:

```
// global
private static final int THICKNESS = 2;   // of drawn line

public void drawArrow(IplImage im)
{
  // draw the direction line
  cvLine(im, p0, p1, CvScalar.BLUE, THICKNESS, CV_AA, 0);

  int arrowHeadLen = (int)Math.round(length/4);
  CvPoint arrowEnd = new CvPoint();

  // compute coords of end of first segment of arrow head
  arrowEnd.x( (int)Math.round(p1.x() -
               arrowHeadLen * Math.cos(angle + Math.PI/4)) );
  arrowEnd.y( (int)Math.round(p1.y() -
               arrowHeadLen * Math.sin(angle + Math.PI/4)) );

  // draw the first segment
  cvLine(im, arrowEnd, p1, CvScalar.BLUE, THICKNESS, CV_AA, 0);

  // compute coords of end of second segment
  arrowEnd.x( (int)Math.round(p1.x() -
               arrowHeadLen * Math.cos(angle - Math.PI/4)) );
  arrowEnd.y( (int)Math.round(p1.y() -
               arrowHeadLen * Math.sin(angle - Math.PI/4)) );

  // draw the second segment
  cvLine(im, arrowEnd, p1, CvScalar.BLUE, THICKNESS, CV_AA, 0);
}  // end of drawArrow()
```

5.4. Calculating a Center-of-gravity (COG)

Back in main() in the OpticalFlowMove class, the COG is represented by a CvPoint which is updated at the end of each processing loop by updateCOG(). The main problem is how to summarize hundreds of direction vectors as a single position. My solution is to employ angledDirs[] – the mean position is calculated using only the directions from the biggest list, which means it's based on the most common direction.

The details are delegated to findMeanPos(), while updateCOG() mostly deals with deciding whether the COG point should be drawn on the grabbed image. The code for

updateCOG():

```
// globals
private static final int MAX_WITHOUT_DIRS = 30;
                // num of iterations before COG disappears
private static int noDirsCount = 0;


private static CvPoint updateCOG(CvPoint cogBall,
             ArrayList<Direction>[] angledDirs, int numDirs)
{ if (numDirs == 0) {
    noDirsCount++;
    if (noDirsCount > MAX_WITHOUT_DIRS) {
      cogBall = null;    // means that COG will not be drawn
      noDirsCount = 0;
    }
  }
  else {  // there are some directions
    CvPoint meanPos = findMeanPos(angledDirs, numDirs);
    if (meanPos != null)
      cogBall = meanPos;
  }
  return cogBall;
}  // end of updateCOG()
```

Each time that updateCOG() receives an empty angledDirs[] array, a global counter noDirsCount is incremented. When it reaches a prescribed maximum (MAX_WITHOUT_DIRS), the COG is set to null causing the point to disappear from the image.

findMeanPos() calculates a mean direction position in two steps: first the biggest list is found in angledDirs[]. The mean midpoint of the directions in that list is calculated, and returned as a new position for the COG.

```
// globals
private static final int ANGLE_RANGE = 20;
private static final int NUM_ANGLE_LISTS = 360/ANGLE_RANGE;
private static final int MIN_DIRS_IN_LIST = 5;

private static CvPoint findMeanPos(
            ArrayList<Direction>[] angledDirs, int numDirs)
{
  CvPoint meanPos = null;
  if (numDirs > 0) {
    // find the largest group of directions
    int maxListSize = 0;
    int largestListIdx = -1;
    for (int i=0; i < NUM_ANGLE_LISTS; i++) {
      if (angledDirs[i].size() > maxListSize) {
        maxListSize = angledDirs[i].size();
        largestListIdx = i;
      }
    }

    if (largestListIdx != -1) {
      // find the mean midpoint in the largest list
      ArrayList<Direction> popDirs = angledDirs[largestListIdx];
      int listSize = popDirs.size();
      // group must be at least MIN_DIRS_IN_LIST in size
```



```
      if (listSize > MIN_DIRS_IN_LIST) {
        int xTot = 0;
        int yTot = 0;
        for (Direction d : popDirs) {
          CvPoint midpt = d.getMid();
          xTot += midpt.x();
          yTot += midpt.y();
        }
        meanPos = new CvPoint(xTot/listSize, yTot/listSize);
      }
    }
  }
  return meanPos;
}  // end of findMeanPos()
```

findMeanPos()'s result may be null if no directions are found, or the largest list isn't long enough (i.e. greater than MIN_DIRS_IN_LIST). A null result will cause updateCOG() to leave the COG position unchanged.

5.5. More Accurate Corners

The processing loop inside main() relies on two OpenCV functions highlighted in the code fragment below:

```
// inside main()
CvPoint2D32f cornersA = new CvPoint2D32f(MAX_CORNERS);
CvPoint2D32f cornersB = new CvPoint2D32f(MAX_CORNERS);
                                    // used as arrays in JavaCV
   :
// find corners
cornersA.position(0);
if (findCorners)
  cvGoodFeaturesToTrack(prevGray, eigenIm, tempIm,
    cornersA, cornerCount, 0.01, 5, null, 3, 0, 0.04);

cornersA.position(0);
cornersB.position(0);
cvCalcOpticalFlowPyrLK(prevGray, grayGrab,
    pyramidA, pyramidB,
    cornersA, cornersB, cornerCount[0],
    cvSize(WIN_SIZE,WIN_SIZE), 5,
    cornersFound, trackErrs,
    cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 20, 0.3),
    plk_flags);
```

The corners array, cornersA, filled by cvGoodFeaturesToTrack() will consist of integer pixel locations, and cvCalcOpticalFlowPyrLK() will track those corners in the current image, filling cornersB with their new locations.

Integer coordinates are good enough for this application, but occasionally it's useful to pinpoint a corner more exactly inside a pixel. Subpixel corners, which utilize floating point accuracy, are generally necessary if fine-grained measurements or velocities need to be calculated, or the task involves camera calibration. This explains why corners array are typically defined as a sequence of floats (e.g. the CvPoint2D32f type in JavaCV).

Subpixel locations can be calculated using the cvFindCornerSubPix() function which replaces the integer pixel values in the supplied corners array with floating point positions based on extrapolating from the intensities gradients of neighboring pixels in the image.

This approach would result in the code fragment above being changed to:

```
// inside main()
CvPoint2D32f cornersA = new CvPoint2D32f(MAX_CORNERS);
CvPoint2D32f cornersB = new CvPoint2D32f(MAX_CORNERS);
                                    // used as arrays in JavaCV
   :
// find corners
cornersA.position(0);
if (findCorners) {
  cvGoodFeaturesToTrack(prevGray, eigenIm, tempIm,
        cornersA, cornerCount, 0.01, 5, null, 3, 0, 0.04);

  cvFindCornerSubPix(prevGray, cornersA, cornerCount[0],
        cvSize(WIN_SIZE, WIN_SIZE),  cvSize(-1, -1),
        cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 20, 0.3));
}

cornersA.position(0);
cornersB.position(0);
cvCalcOpticalFlowPyrLK(prevGray, grayGrab,
    pyramidA, pyramidB,
    cornersA, cornersB, cornerCount[0],
    cvSize(WIN_SIZE,WIN_SIZE), 5,
    cornersFound, trackErrs,
    cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 20, 0.3),
    plk_flags);
```

The key change is the inclusion of a call to cvFindCornerSubPix() between cvGoodFeaturesToTrack() and cvCalcOpticalFlowPyrLK().

cvFindCornerSubPix() uses termination criteria in the same way as cvCalcOpticalFlowPyrLK(), and the same arguments can be used for both. cvFindCornerSubPix() utilizes a window size (WIN_SIZE) to determine how many neighboring pixels to examine for gradient information, and varying that value will affect the execution time. In my tests, the above method added about 20 ms to each iteration.

A full description of all the arguments for cvFindCornerSubPix() can be found on the OpenCV API documentation page http://docs.opencv.org/modules/video/doc/motion_analysis_and_object_tracking.html , or on p.319-321 of the first edition of *Learning OpenCV*.

6. Comparing the Movement Detection Approaches

In this chapter, I've looked at three ways of detecting movement:

- image differencing of consecutive video frames
- background/foreground segmentation

- optical flow

Movement detection using image differences and moments is simple to implement and executes quickly. However, it also have some limitations, the main one being that it isn't really detecting movement, only change. For example, if the scene being monitored contains a flashing light, then that will be detected.

The same advantages and disadvantages apply to background/foreground segmentation using BackgroundSubtractorMOG2, with the additional confusion of the technique's name. It's quite easy for users to be misled into thinking that depth (distance from the camera) is somehow employed in the segmentation process. As we've seen, this is not the case, with the foreground being distinguished only by its rate of change (i.e. movement) relative to a static background. Indeed, if elements close to the camera are stationary, then they'll be placed in the background, while more distant, moving objects will be added to the foreground mask.

The use of moments assumes that all the change (i.e. the white pixels in the difference image or foreground mask) form a single shape, and so a single center-of-gravity is returned. Of course, in a busy scene, such as a traffic intersection, there will be many distinct shapes (i.e. cars) moving about. This code will position the crosshairs at the 'center' of all this movement, which may not be of much use. Another example of this problem is shown in Figure 13.

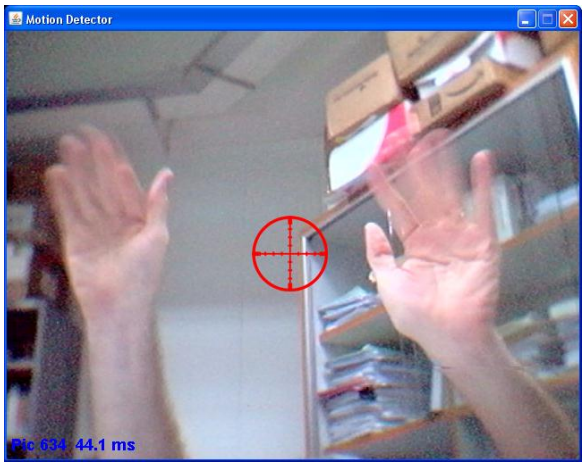

Figure 13. Waving Hands and the Crosshairs.

My waving hands in Figure 13 are treated as a single shape, so the crosshairs are positioned between them, in thin air!

Unlike image differencing and background/foreground segmentation, optical flow tracks multiple interesting features (i.e. corners) in the image, and so can distinguish between multiple sources of movement, as in Figure 14.

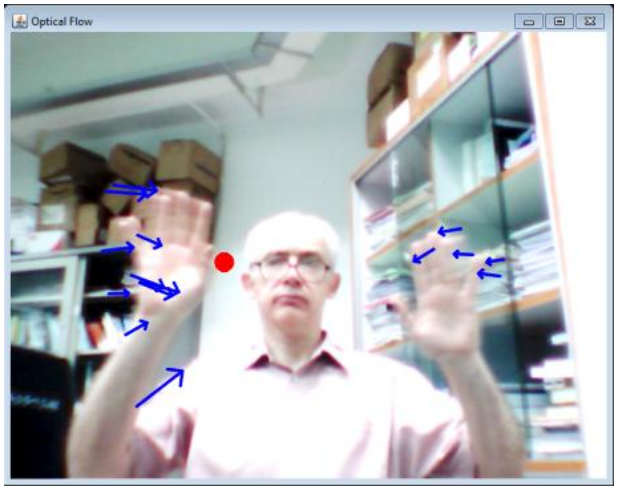

Figure 14. Optical Flow of Two Hands.

However, my optical flow application makes the assumption that there is a single most important direction of movement in the scene, and calculates a position for the red COG circle based on that simplification.

7. Using Motion to Shoot Missiles

In an earlier draft of this book, I included a chapter using the image difference motion detector code to control a toy missile launcher. The *Dream Cheeky* launcher, shown in Figure 15, has a base that can be rotated left, right, up, or down via USB commands to point at a target, and includes an all-important 'fire' instruction for shooting.

Figure 15. The *Dream Cheeky* Missile Launcher.

The attachment on top of the missile turret is a webcam, which can be used as the image source rather than the usual camera. Since it's attached to the launcher, it always points in the same direction as the missiles.

Regretfully, shooting stuff doesn't have much to do with vision-based user interfaces, and so it's not included here. An online version is available at http://fivedots.coe.psu.ac.th/~ad/jg/nui04/.

