Kinect Chapter 12. Motion Detection Using OpenCV

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

[**Note**: this chapter is virtually identical to "NUI Chapter 3" at http://fivedots.coe.psu.ac.th/~ad/jg/nui03/, except that I'm using the Kinect sensor as the input device rather than a webcam.]

This chapter explains how to use the Kinect's camera to detect change or movement in a scene. Frames are grabbed from the camera and drawn rapidly onto a panel, using the code developed in the previous chapter. A movement detector analyzes consecutive frames and highlights any change/movement with a pair of crosshairs at the center-of-gravity of the motion. The application, called MotionDetector, is shown in Figure 1.
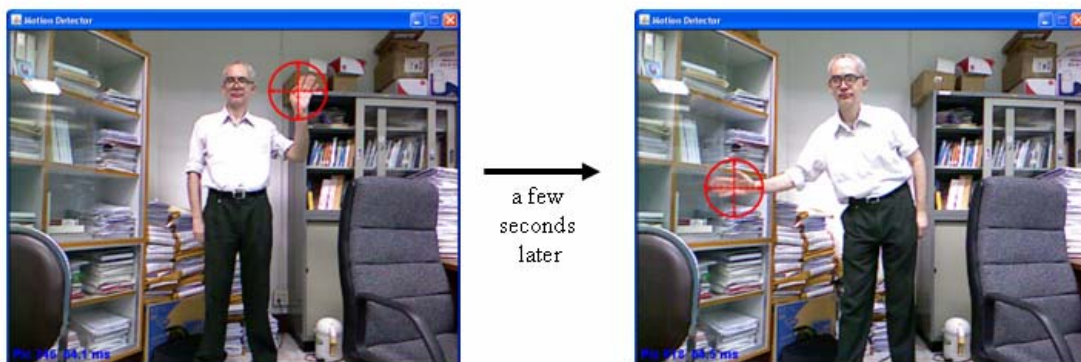

Figure 1. Motion Detection over Time.

The detection algorithm is implemented using the OpenCV computer vision library (http://opencv.willowgarage.com/). OpenCV is such an important tool that I'll spend a little time explaining its capabilities, and especially its Java interface (called JavaCV), before describing the detection application.

1. OpenCV

OpenCV is a real-time computer vision library with very extensive functionality (over 500 high-level and low-level functions). It was originally developed at Intel, starting in 1999 as a C library. Its 1.0 version was released in 2006, after going through five betas. OpenCV is open source, is available on all major platforms, and has API interfaces for many programming languages.

OpenCV 2.0 was released in October 2009, with a new C++ interface, and since 2008 has been supported by the Willow Garage company. The best starting point is its wiki

at http://opencv.willowgarage.com/wiki/, which has links to downloads, documentation, examples, and tutorials. The C API reference material is at http://opencv.willowgarage.com/documentation/c/, which will be useful when I describe the (lack of) documentation for the Java API.

There's an enormous amount of online tutorial material for OpenCV, and a wonderful textbook:

> *Learning OpenCV: Computer Vision with the OpenCV Library*
> Gary Bradski and Adrian Kaehler
> O'Reilly, September 2008
> http://oreilly.com/catalog/9780596516130

The date of publication is significant, because it indicates that the book (excellently) describes the features of OpenCV 1.0 using its C interface. There's nothing on the new C++ interface, or other language APIs such as Python, C#, or Java.

The Yahoo OpenCV group (http://tech.groups.yahoo.com/group/OpenCV/) is probably the most active forum for OpenCV.

I could enumerate the very long list of computer vision capabilities in OpenCV, but its easier just to say "everything". The C reference material has nine sections: core (core functionality), imgproc (image processing), features2d (feature detection), flann (multi-dimensional clustering), objdetect (object detection), video (video analysis), highgui (GUI and media I/O), calib3d (3D camera calibration), and ml (machine learning). These section names are useful to remember when we look at the JavaCV package and class structure.

Basic image processing features include filtering, edge detection, corner detection, sampling and interpolation, and color conversion. The GUI capabilities mean that an OpenCV program can be written without the use of OS-specific windowing features, although in JavaCV it's just as portable to use Swing/AWT.

2. JavaCV

JavaCV is a Java wrapper around OpenCV (http://code.google.com/p/javacv/), which started out using JNA to create an interface to OpenCV's C API. However, JavaCV switched to using JavaCPP to map Java onto C++ in 2011. This mapping incurs less overheads than JNA, and makes it easier to utilize features from the OpenCV 2.0 C++ API.

JavaCV comes with support wrappers for a number of other libraries, such as FFmpeg, ARToolKitPlus, and OpenKinect. JavaCV's freenect wrapper makes it possible to read the Kinect's depth and camera images. Unfortunately, there's no OpenNI support at the moment, which is the reason for my developing KinectCapture in the last chapter.

The JavaCV documentation is quite brief, but gradually growing. Places to look are the useful examples at http://code.google.com/p/javacv/#Quick_Start_for_OpenCV, links to other people's examples in http://code.google.com/p/javacv/wiki/OtherResources, and programs in the download.

There are no Java API documentation pages, so JD_GUI (http://java.decompiler.free.fr/?q=jdgui) is invaluable for examining the contents of javacv.jar in the download. Fortunately, JavaCV parallels OpenCV so closely that I was able to find the necessary C function (either in the "Learning OpenCV" book or online), and then do a search through the JAR file. Another approach I used was to decompile the contents of the JAR, and use grep (actually "Windows Grep", http://www.wingrep.com/) to search the source for the same-named function.

An invaluable source is the JavaCV forum in Google groups (http://groups.google.com/group/javacv), which is actively monitored by the JavaCV developer.

OpenCV / JavaCV Small Examples

The easiest way to introduce OpenCV, and its Java API, is through some simple examples. I'll describe two in the following sections: ShowImage.java loads an image from a file and displays it in a window, and CameraCapture.java grabs a series of images from the local webcam and displays them in a window.

ShowImage.java is the "hello world" example of the OpenCV community, seemingly appearing in every tutorial for the library. For instance, it's the first example in the "Learning OpenCV" textbook (on p.17 of the first edition). CameraCapture.java shows how webcam functionality is employed in OpenCV/JavaCV.

3. Loading and Displaying an Image

ShowImage.java reads a filename from the command line, loads it into a JavaCV window (a CanvasFrame object), and then waits for the user to press a key while the focus is inside that window. When the key press is detected, the application exits. The execution of ShowImage.java is shown in Figure 2.

Figure 2. The ShowImage Example.

The code for ShowImage.java:

```
import com.googlecode.javacv.*;
import static com.googlecode.javacv.cpp.opencv_core.*;
import static com.googlecode.javacv.cpp.opencv_highgui.*;
```

```
public class ShowImage
{
  public static void main(String[] args)
  {
    if (args.length != 1) {
      System.out.println("Usage: run ShowImage <input-file>");
      return;
    }

    System.out.println("OpenCV: Loading image from " +
                                      args[0] + "...");
    IplImage img = cvLoadImage(args[0]);
    System.out.println("Size of image: (" + img.width() +
                                ", " + img.height() + ")");

    // display image in canvas
    CanvasFrame canvas = new CanvasFrame(args[0]);
    canvas.setDefaultCloseOperation(CanvasFrame.DO_NOTHING_ON_CLOSE);

    canvas.showImage(img);

    canvas.waitKey();    // wait for keypress on canvas
    canvas.dispose();
  }  // end of main()

}  // end of ShowImage class
```

The listing above includes import lines, which I usually leave out. They illustrate an important JavaCV coding style – the *static* import of the classes containing the required OpenCV native functions. In this case, ShowImage utilizes functions from com.googlecode.javacv.cpp.opencv_core and com.googlecode.javacv.cpp.opencv_highgui. These JavaCV package names are derived from the OpenCV C API documentation section names, core and highgui, at http://opencv.willowgarage.com/documentation/c/.

The static imports mean that class names from these Java packages don't need to be prefixed to their method names. For instance, in ShowImage.java, I can write cvLoadImage(args[0]) without having to add the class name before cvLoadImage(). This makes the code very like its C version:

```
IplImage* img = cvLoadImage( argv[1] );   // C code, not Java
```

The CanvasFrame class is a JavaCV original, which implements OpenCV windowing functions such as cvNamedWindow() and cvShowImage() as CanvasFrame's constructor and the showImage() method respectively. CanvasFrame.waitKey() parallels OpenCV's cvWaitKey() which waits for a key press.

CanvasFrame is implemented as a subclass of Java's JFrame, and so it's possible to dispense with OpenCV's key-waiting coding style. Instead, we can write:

```
canvas.setDefaultCloseOperation(CanvasFrame.EXIT_ON_CLOSE);
```

which will cause the application to exit when the close box is pressed. We should, at the very least, include the line:

```
canvas.setDefaultCloseOperation(CanvasFrame.DO_NOTHING_ON_CLOSE);
```

This disables the close box on the CanvasFrame so it's not possible to make the window disappear without terminating the application. CanvasFrame's maximize

window box also doesn't redraw the window contents correctly without some extra coding.

In my opinion, OpenCV's highGUI functions are good for prototyping and debugging image processing code, but an application should utilize Java's Swing for its finished user interface.

Another important element of this example is the IplImage class, which corresponds to the IplImage struct in OpenCV. JavaCV uses this data structure for image storage, not Java's BufferedImage.

4. Grabbing Pictures from a WebCam

OpenCV contains useful support for accessing webcams and video files, which are treated as sequences of image frames, each of which can be manipulated. The relevant OpenCV function for accessing a webcam is cvCreateCameraCapture(), but JavaCV uses a more object-oriented approach – the OpenCVFrameGrabber class (a subclass of FrameGrabber).
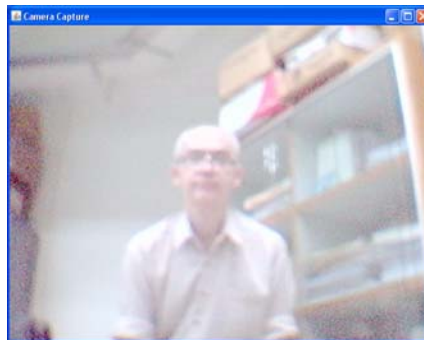
The CameraCapture.java example is shown in Figure 3.

Figure 3. Capturing Images from a Webcam with JavaCV.

The code for CameraCapture.java:

```
import java.io.*;
import java.awt.event.*;

import com.googlecode.javacv.*;
import static com.googlecode.javacv.cpp.opencv_core.*;
import static com.googlecode.javacv.cpp.opencv_highgui.*;


public class CameraCapture
{
  private static volatile boolean isRunning = true;


  public static void main(String[] args)
  {
    System.out.println("Starting OpenCV...");
    try {
      CanvasFrame canvas = new CanvasFrame("Camera Capture");
```

```
      canvas.addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent e)
        { isRunning = false; }
      });

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

      IplImage frame;
      while(isRunning) {
        if ((frame = grabber.grab()) == null)
          break;
        canvas.showImage(frame);
      }
      grabber.stop();
      canvas.dispose();
    }
    catch(Exception e)
    {  System.out.println(e);  }
  }  // end of main()

} // end of CameraCapture class
```

The code spends most of its time in a loop which keeps calling OpenCVFrameGrabber.grab() to grab an image as an IplImage object. The image is quickly pasted into a CanvasFrame with CanvasFrame.showImage().

The loop is controlled by a boolean, which is set to false when the user presses the window's close box.

The execution of OpenCVFrameGrabber was very reliable across my test machines, although the OpenCVFrameGrabber can take several seconds to start up.

I'll utilize KinectCapture in the rest of this chapter so I can grab images from the Kinect. As I mentioned at the start, this chapter is a thinly revised version of NUI Chapter 3 (online at http://fivedots.coe.psu.ac.th/~ad/jg/) which uses a webcam via my JMFCapture class. KinectCapture and JMFCapture have almost identical interfaces which means that I only needed to change two lines of code to convert my webcam-based motion detector into a Kinect-based version.

You may be put off using OpenCVFrameGrabber because its grab() method returns an IplImage object, while the rest of Java utilizes BufferedImage instances. In fact, it's easy to switch between the two formats with IplImage.createFrom() and IplImage. getBufferedImage():

```
// BufferedImage to IplImage
BufferedImage im = /* a Java BufferedImage */
IplImage cvImg = IplImage.createFrom(im);

// IplImage to BufferedImage
BufferedImage im2 = cvImg.getBufferedImage();
```

5. Motion/Change Detection in OpenCV

The motion detector class (called CVMotionDetector) described in this section will be utilized by my GUI Motion detector application later in the chapter. It also includes a fairly simple test-rig in its main() function using OpenCV's OpenCVFrameGrabber and CanvasFrame classes. I'll explain the test-rig first, then look at the internals of the class.

5.1. The CVMotionDetector Test-rig

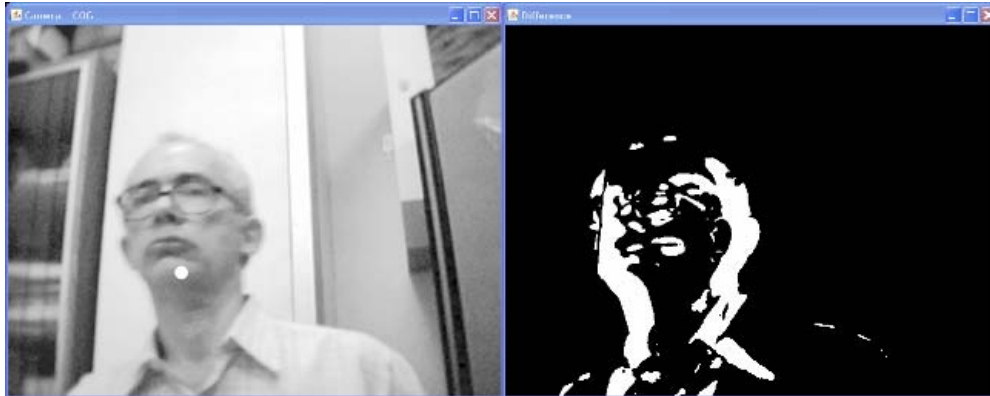The test-rig keeps updating two CanvasFrame displays, which are shown in Figure 4.

Figure 4. The CVMotionDetector 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. 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 OpenCVFrameGrabber instance for accessing the webcam, and initializes a CVMotionDetector object. It 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("OpenCV: 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
  CanvasFrame cogCanvas = new CanvasFrame("Camera + COG");
  cogCanvas.setLocation(0, 0);

  CanvasFrame diffCanvas = new CanvasFrame("Difference");
  diffCanvas.setLocation(imgDim.width+5, 0);  // on the right

  // 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 frame 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 CanvasFrames positioned side-by-side.

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 240 ms, which is quite lengthy. Fortunately this drops quite considerably when CVMotionDetector is employed in the GUI motion detection of Figure 1.

© Andrew Davison 2011

5.2. 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.

5.3. 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 OpenCV'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. Thresholding is being used as a simple form of clustering, which says that intensities above 65 are important, while those below can be ignored.

© Andrew Davison 2011

5.4. 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.
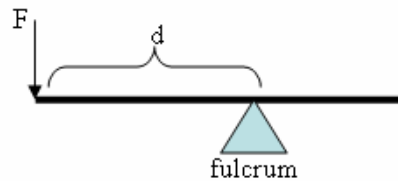

Figure 5. Force Acting at a Distance from a Fulcrum.

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.
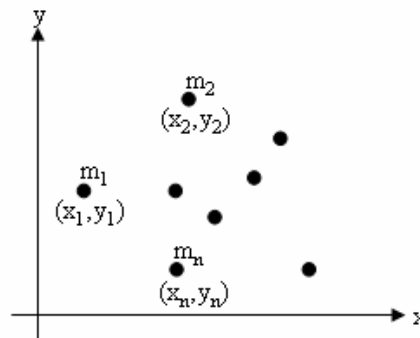

Figure 6. Points in Two-dimensions.

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 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:

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

and $$\bar{y} \quad \frac{M_x}{m_{sys}}$$

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.
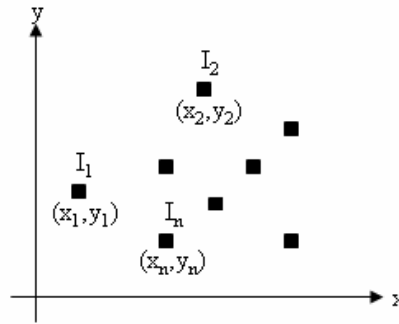
Figure 7. Pixels in Two-dimensions.

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.

5.5. Moments in OpenCV

OpenCV 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:

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

and

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

findCOG() uses OpenCV'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 OpenCV'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 OpenCV 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.

6. The GUI-based Motion Detector
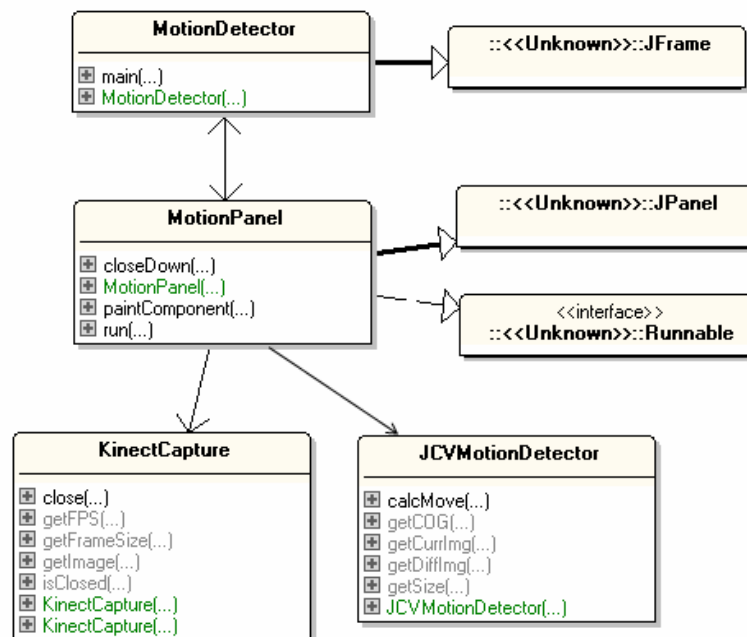
The applications class are summarized by Figure 8.

Figure 8. Class Diagrams for the MotionDetector Application.

The GUI uses Swing, not JavaCV's CanvasFrame, so MotionDetector and MotionPanel subclass JFrame and JPanel respectively. I won't bother explaining the top-level MotionDetector class – it's a standard JFrame which creates a MotionPanel object for rendering the webcam and crosshairs images.

MotionPanel is very similar to the PicsPanel class of the previous chapter, in that it spends much of it's time inside a threaded loop repeatedly grabbing an image from the Kinect (with KinectCapture) and drawing it onto the panel until the window is closed.

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).

6.1. The Webcam Display Loop

MotionPanel executes the camera display loop inside run(). The code is almost identical to that in PicsPanel in the last chapter 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 JFrame top;
private BufferedImage image = null;  // current webcam snap
private KinectCapture camera;        // changed from webcam version
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()
{
  camera = new KinectCapture();    // changed from webcam version

  BufferedImage im = camera.getImage();
  JCVMotionDetector md = new JCVMotionDetector(im);
      // create motion detector (uses OpenCV so is slow to start)

  // update panel and window sizes to fit video's frame size
  Dimension frameSize = camera.getFrameSize();
  if (frameSize != null) {
    setPreferredSize(frameSize);
    top.pack();   // resize and center JFrame
    top.setLocationRelativeTo(null);
  }

  Point pt;
  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;
    }
```

© Andrew Davison 2011

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

      image = im;    // only update image if im contains something
      imageCount++;
      duration = System.currentTimeMillis() - startTime;
      totalTime += duration;
      repaint();
    }

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

The two highlighted lines in the code:

```
private KinectCapture camera;
```

and

```
camera = new KinectCapture();
```

are the only changes needed to convert this application from using a webcam to the Kinect. In the original version (in NUI Chapter 3, http://fivedots.coe.psu.ac.th/~ad/jg/nui03/), the class name was JMFCapture on the two lines.

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.

At run time, the timing code indicates that one iteration (which includes Kinect camera capture, rendering, and motion detection) takes about 40-50 ms, and so the delay constant for redrawing (DELAY) was increased to 100 ms.

These timing results were initially quite surprising since the JavaCV test-rig for CVMotionDetector takes an average of 240 ms to complete one rendering cycle. The reasons for the difference can be seen by comparing the while-loop in the test-rig with the display loop in run(). The JavaCV code has to access the current image and the difference image in order to render them onto the canvases, and the current image must be copied so that a white circle can be drawn onto it. None of these tasks needs to be done in the MotionPanel display loop.

6.2. 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()
```

6.3. Rendering Motion Detection

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

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

```
// global
private BufferedImage image = null;   // current webcam snap
```

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

  if (image != null)
    g2.drawImage(image, 0, 0, this);

  if (cogPoint != null)
    drawCrosshairs(g2, cogPoint.x, cogPoint.y);
          // crosshairs are positioned at COG point
  writeStats(g2);
} // end of paintComponent()
```

writeStats() is unchanged from the last chapter, so I won't explain it again.

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.

The image is loaded in MotionPanel's constructor:

```
// global
private static final String CROSSHAIRS_FNM = "crosshairs.png";

/* circle and cross-hairs dimensions
   (only used if crosshairs image cannot be loaded)  */
private static final int CIRCLE_SIZE = 40;
private static final int LINES_LEN = 60;


// in the MotionPanel() constructor
// load the crosshairs image (a transparent PNG)
crosshairs = null;
try {
  crosshairs = ImageIO.read(new File(CROSSHAIRS_FNM));
}
catch (IOException e)
{  System.out.println("Could not find " + CROSSHAIRS_FNM);  }
```

The drawCrosshairs() method:

```
private void drawCrosshairs(Graphics2D g2, int xCenter, int yCenter)
// draw crosshairs graphic or make one from lines and a circle
{
  if (crosshairs != null)
    g2.drawImage(crosshairs, xCenter - crosshairs.getWidth()/2,
                          yCenter - crosshairs.getHeight()/2, this);
  else {
```

```
      // draw thick red circle and cross-hairs in center
      g2.setColor(Color.RED);
      g2.drawOval(xCenter - CIRCLE_SIZE/2, yCenter - CIRCLE_SIZE/2,
                                  CIRCLE_SIZE, CIRCLE_SIZE);
      g2.drawLine(xCenter, yCenter - LINES_LEN/2,
                  xCenter, yCenter + LINES_LEN/2);    // vertical line
      g2.drawLine(xCenter - LINES_LEN/2, yCenter,
                  xCenter + LINES_LEN/2, yCenter);    // horiz line
  }
}  // end of drawCrosshairs()
```

A surprising absence from OpenCV is support for partially transparent or translucent images; OpenCV can load a PNG file, but any alpha channel in the image is stripped away. Also, OpenCV can only perform simple weighted blending of two images, whereas Java 2D offers a full range of Porter-Duff alpha compositing rules (eight are directly available, and the other four, less commonly used ones, can be fairly easily implemented).

drawCrosshairs() would only be 1-2 lines long except for backup code which draws a red circle, and two crossed lines if the image isn't available.

7. Modifying the OpenCV Motion Detection

MotionPanel uses the JCVMotionDetector class, which is different from CVMotionDetector in two ways.

A minor change is that the constructor and calcMove() methods are passed BufferedImage objects rather than IplImage instances. Each image is still processed by convertFrame() for blurring, grayscaling, and equalization, so it was easy to have the method also change the image's type. The new version of convertFrame() adds a single line that employs IplImage.createFrom() to convert the BufferedImage into a IplImage:

```
// in JCVMotionDetector

private IplImage convertFrame(BufferedImage buffIm)
{
  IplImage img = IplImage.createFrom(buffIm);

  // 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()
```

A more significant change in JCVMotionDetector is the addition of a simple form of smoothing for the center-of-gravity point. In the first version of calcMove(), a point

© Andrew Davison 2011

was stored in a global called cogPoint, and retrieved by calls to getCOG(). In JCVMotionDetector, calcMove() stores the new point in 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(BufferedImage 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);
  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 center-of-gravities, 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 affects the current one.

8. Problems with this Approach to Movement Detection

On the plus side, movement detection using image differences and moments is simple to implement and fast. But it also have some serious 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 use of moments assumes that all the change (i.e. the white pixels in the difference image) 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 10.
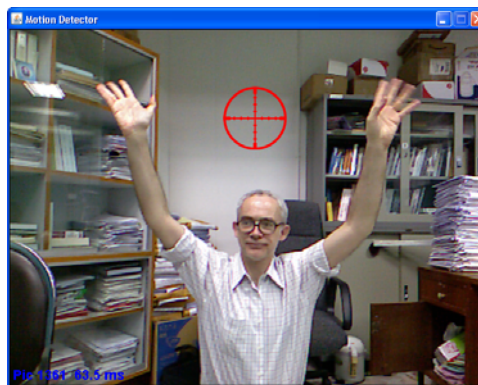

Figure 10. Waving Hands and the Crosshairs.

My waving hands in Figure 10 are treated as a single shape, so the crosshairs are positioned between them, in thin air!

Perhaps the most serious drawback of MotionDetector is that it's not utilizing all the capabilities of the input device. For instance, the Kinect's depth map could be employed to separate the background from the foreground. It is also easier to rely on OpenNI's skeleton tracking and/or hand points information if a human figure is being tracked.

Despite these issues, this chapter's approach to motion detection is useful when the scene's background is unchanging, and the detector is looking for a single moving

shape which doesn't have to be a human. It also acts as a simple introduction to combining the OpenNI and OpenCV.

9. More OpenCV Examples (and Beyond)

Unfortunately, one chapter is all the time I can spend on the Kinect and OpenCV in this book, but there's several more "related" OpenCV chapters online at my website http://fivedots.coe.psu.ac.th/~ad/jg/, which utilize a webcam as the input device. As with the original version of this chapter (NUI Chapter 3, http://fivedots.coe.psu.ac.th/~ad/jg/nui03/), the webcam is accessed via a JMFCapture class to deliver the video frames. The examples can be easily converted to use the Kinect by changing all the references to JMFCapture to KinectCapture.

The chapter topics:

A Motion-tracking Missile Launcher
(http://fivedots.coe.psu.ac.th/~ad/jg/nui04/). This example combines the motion detector code of this chapter with a toy missile launcher.

Blobs Drumming (http://fivedots.coe.psu.ac.th/~ad/jg/nui05/). I employ colored blob detection to play a variety of percussion sounds.

Face Detection and Tracking (http://fivedots.coe.psu.ac.th/~ad/jg/nui07/). OpenCV's pre-trained Haar classifier is used to detect faces in a scene.

Face Recognition (http://fivedots.coe.psu.ac.th/~ad/jg/nui08/). This application extends face detection to recognition, using a technique called eigenfaces.