Chapter NUI-2b. Webcam Snaps Using JavaCV

The webcam is at the heart of all the image processing and computer vision techniques that I'll be explaining in this book. This chapter is about implementing the basic webcam capabilities needed by those examples.

What's required is not a video stream coming from the camera, but a series of pictures or snaps, generated fairly infrequently (e.g. 10 images per second). Very often those pictures should be rendered as grayscales rather than in full color to help simplify the subsequent processing. It's also useful to have a way to store selected images to files.

In previous versions of this chapter, I implemented webcam snapping using JMF (the Java Media Framework). Unfortunately, JMF has become something of a dinosaur, that hasn't been updated since the last century. It can't play popular modern formats, such as MPEG-2, MPEG-4, Windows Media, RealMedia, or most QuickTime and Flash files. Its content editing functionality is feeble. More seriously for my snapping needs is that JMF's cross-platform support is becoming a problem – it appears to be impossible (or very, very difficult) to get JMF to run on the 64-bit version of Windows. There are forum postings explaining how to do it (see https://forums.oracle.com/forums/thread.jspa?threadID=2132405&tstart=8), and other posts claiming that the approach doesn't work!

It's time to move on from dear old JMF. Not unsurprisingly, there are a number of alternatives, conveniently listed on the JMF Wikipedia page (http://en.wikipedia.org/wiki/Java_Media_Framework). Since my key requirement is good support for image processing and computer vision techniques, I've decided to use JavaCV, a Java binding for the OpenCV vision library (http://code.google.com/p/javacv/).

The aim is to grab images at a reasonable speed of ten image per second, and display them in succession in a JPanel. The panel output includes the average time to take a snap and do any processing upon it, which will help me adjust the snapping rate in later examples. Figure 1 shows the JavaCV application in action.

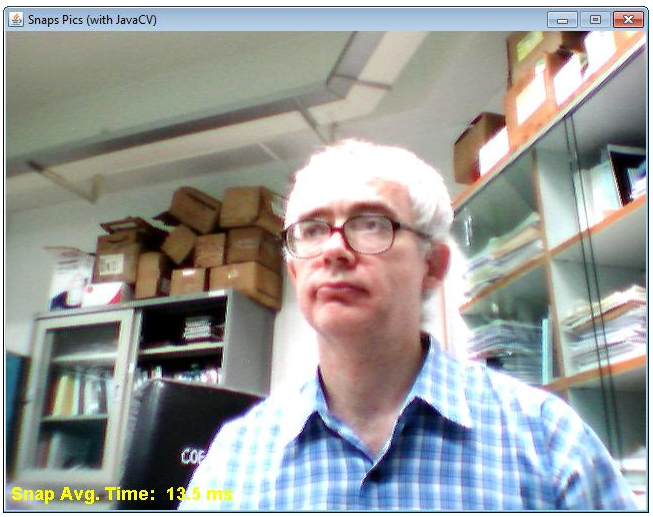

Figure 1. JavaCV Webcam Pictures.

The information about the picture is written in yellow in the bottom-left corner; in Figure 1 it says "Snap Avg. Time 13.5 ms". This shows that a frame is being processed in much less time than the 100 ms interval between redraws, which leaves plenty of time for additional image processing before the next picture arrives from the camera.

One of the irritations of working with a webcam is making sure that it's been correctly installed and identified by the operating system. Section 4 of this chapter describes three standalone tools (CommandCam, DevCon, and FFmpeg) that can check out a webcam independently of OpenCV and Java.

Although almost all of this book's examples employ OpenCV in some way, there are situations when it's massive capabilities (and size) aren't needed. The next chapter briefly puts OpenCV to one side, and implements webcam snapping using vlcj, the Java API for the popular VLC media player (http://code.google.com/p/vlcj/).

1. OpenCV and Java

OpenCV (http://opencv.org/) 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, 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 extensive online documentation at http://opencv.org/documentation.html, which has links to downloads, manuals, a wiki, examples, and tutorials. The current API reference material is at http://docs.opencv.org/2.4.5/modules/refman.html, which will be useful when I describe the (lack of) documentation for JavaCV.

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#, and 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.

1.1. JavaCV

JavaCV is a Java wrapper around OpenCV (http://code.google.com/p/javacv/). It's documentation is quite brief, mostly amounting to useful examples at http://code.google.com/p/javacv/#Quick_Start_for_OpenCV, and some others in the library download.

Unfortunately, there are no Java API documentation pages as yet. Probably the easiest way of finding out about a particular class, such as JavaCV's FrameGrabber used in this chapter, is via a Google Search for "JavaCV FrameGrabber filetype:java". This should bring up JavaCV's source code file for that class as the first hit, with the other links going to examples that use the class.

An alternative is to utilize a JAR browsing tool to examine the JAR files that make up the JavaCV download. javacv.jar contains the core classes, and is located in the javacv-bin\ directory. I use the JD-GUI browser (http://java.decompiler.free.fr/?q=jdgui) which decompiles the class files on-the-fly as you open them, and adds links between related classes. It can also permanently decompile the JAR, saving the source which you can text search with a grep-like tool such as "Windows Grep" (http://www.wingrep.com/).

Most JavaCV methods have similar names to their OpenCV counterparts, so a good way to find out about a particular method, such as JavaCV's cvCvtColor(), is to type "cvCvtColor opencv" into Google search, or look up the method in the index of the *Learning OpenCV* book.

An invaluable source is the JavaCV forum in Google groups (http://groups.google.com/group/javacv), which is actively monitored by the JavaCV developer. Some care must be taken when looking at old posts prior to 2011 because they refer to an out-of-date JNA-mapping of JavaCV to OpenCV.

1.2. Loading and Displaying an Image with JavaCV

The easiest way to introduce JavaCV, is through a simple example, ShowImage.java, which loads an image from a file and displays it in a window. ShowImage.java is the "hello world" of the OpenCV community, seemingly appearing in every tutorial for the library. For instance, it's the first example in *Learning OpenCV* (on p.17 of the first edition).

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 keypress 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("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);

    try {
      canvas.waitKey();   // wait for keypress on canvas
    }
    catch(InterruptedException e) {}

    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.

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 class.

1.3. Java OpenCV

Early in 2013, Willow Garage (the current maintainers of OpenCV) released a Java binding for the library (http://docs.opencv.org/2.4.4-beta/doc/tutorials/introduction/desktop_java/java_dev_intro.html), and also updated the Java API for Android. There are probably three advantages to using their "Java OpenCV" instead of JavaCV:

1. The API bindings are automatically updated when OpenCV is updated, so are always current.
2. The API is designed to match the C++ bindings for OpenCV, so are somewhat more object-oriented than the C-inspired JavaCV.
3. There's online Java OpenCV documentation at http://docs.opencv.org/java/2.4.5/

Perhaps the API's main drawback is its newness (I'm writing this in June 2013), which means that there's currently a lot more online examples and help (in the shape of forum posts) for JavaCV.

Since the majority of the coding for this book was finished before Java OpenCV was released, I've decided to stick with JavaCV. However, as a taster of the new API, section 3 briefly outlines how to convert the JavaCV webcam snapper into a Java OpenCV version.

2. Snapping Pictures with JavaCV

The class diagrams for the SnapPics application, with only the public methods visible, are shown in Figure 3.

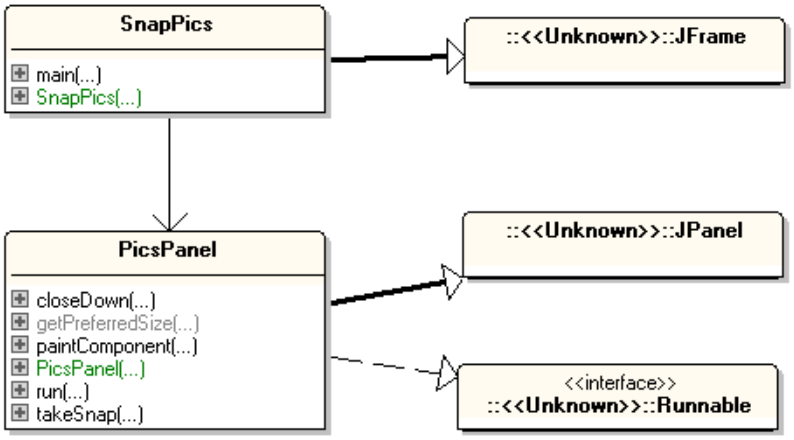

Figure 3. Class Diagrams for SnapPics.

The SnapPics class is in charge of creating the PicsPanel display panel that utilizes the JavaCV FrameGrabber class. SnapPics does two things of note: clicking its close box triggers a call to closeDown() in PicsPanel, and it listens for a keypress to trigger a call to takeSnap() in PicsPanel. Both of these are coded in the SnapPics constructor:

```
// global
private PicsPanel pp;

public SnapPics()
{
  super( "Snaps Pics (with JavaCV)" );
  Container c = getContentPane();
  c.setLayout( new BorderLayout() );

  // Preload the opencv_objdetect module to work around a known bug.
  Loader.load(opencv_objdetect.class);

  pp = new PicsPanel();
  c.add(pp, BorderLayout.CENTER);

  addKeyListener( new KeyAdapter() {
```



```
    public void keyPressed(KeyEvent e)
    {
      int keyCode = e.getKeyCode();
      if ((keyCode == KeyEvent.VK_NUMPAD5) ||
          (keyCode == KeyEvent.VK_ENTER) ||
          (keyCode == KeyEvent.VK_SPACE))
        pp.takeSnap();
    }
   });

  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { pp.closeDown();    // stop snapping pics
      System.exit(0);
    }
  });

  setResizable(false);
  pack();
  setLocationRelativeTo(null);
  setVisible(true);
} // end of SnapPics()
```

The keypress listener responds to the user pressing 5 on the number pad, enter, or space. In response, PicsPanel saves a grayscale version of the current webcam image to a file in the pics/ subdirectory, like the one in Figure 4.

Figure 4. A Saved Grayscale Image.

A bug in JavaCV requires SnapPics to preload its opencv_objdetect module; this isn't needed for every application, but introduces no execution overhead so is included in all my examples.

2.1. Snapping a Picture Again and Again and ...

PicsPanel is threaded so it can keep repeatedly calling FrameGrabber.grab() without causing the GUI parts of the panel to block. The thread is executed by PicsPanel's run() method:

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



```
private static final int CAMERA_ID = 0;


private volatile boolean isRunning;
private volatile boolean takeSnap = false;

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

private IplImage snapIm = null;


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

  long duration;
  int snapCount = 0;
  isRunning = true;

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

    snapIm = picGrab(grabber, CAMERA_ID);

    if (takeSnap) {   // save the current images
      saveImage(snapIm, PIC_FNM, snapCount);
      snapCount++;
      takeSnap = false;
    }

    imageCount++;
    repaint();

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

Each iteration of the loop is meant to take DELAY (100) milliseconds. The snap processing time is stored in the duration variable, and used to modify the loop's sleep period. If the snap duration exceeds the DELAY time, then the loop doesn't sleep at all.

A DELAY value of 100 ms makes the webcam picture update at about 10 frames/second (10 FPS), which is adequate for most computer vision application.

The camera source must be manipulated from a single thread, which includes the initialization of the device, grabbing pictures, and closing the link at the end. If this rule isn't followed, then the application may crash, and can cause Windows to start

acting strangely. Consequently, all interactions with the device are localized inside run()'s thread.

Initializing the Frame Grabber

JavaCV's FrameGrabber is an abstract class which supports a wide range of different types of webcam and camera software via its concrete subclasses DC1394FrameGrabber, FlyCaptureFrameGrabber, OpenKinectFrameGrabber, PS3EyeFrameGrabber, VideoInputFrameGrabber, and FFmpegFrameGrabber.

A FrameGrabber instance cycles through these classes until it finds one that can access the camera (the current order is "DC1394", "FlyCapture", "OpenKinect", "PS3Eye", "VideoInput", "OpenCV", "FFmpeg"). The last three are probably the most commonly used : "VideoInput" (i.e. the VideoInputFrameGrabber class) works with DirectShow on Windows, "OpenCV" makes use of the OpenCV's CvCapture functionality, and "FFmpeg" employs the FFmpeg package (http://www.ffmpeg.org/).

The OpenCV download comes with parts of the FFmpeg libraries (in C:\opencv\3rdparty\ffmpeg\opencv_ffmpeg.dll), including it's capture functions; if they're insufficient then you'll have to download the rest of FFmpeg.

The JavaCV FrameGrabber class hides all of this complexity, and only requires the user to supply the ID number of the webcam that is to be accessed. This ID is almost always 0, as used in my initGrabber() method:

```
// globals
// image size == panel size
private static final int WIDTH = 640;
private static final int HEIGHT = 480;

private static final int CAMERA_ID = 0;

private FrameGrabber initGrabber(int ID)
{
  FrameGrabber grabber = null;
  System.out.println("Initializing grabber for " +
                       videoInput.getDeviceName(ID) + " ...");
  try {
    grabber = FrameGrabber.createDefault(ID);
    grabber.setFormat("dshow");       // use DirectShow
    grabber.setImageWidth(WIDTH);
                       // default resolution is 320x240
    grabber.setImageHeight(HEIGHT);
    grabber.start();
  }
  catch(Exception e)
  {  System.out.println("Could not start grabber");
     System.out.println(e);
     System.exit(1);
  }
  return grabber;
}  // end of initGrabber()
```

The FrameGrabber class contains useful methods for getting and setting camera attributes such as the format and image dimensions. Probably the easiest way of

finding out what these methods are is to do a Google search for "JavaCV FrameGrabber filetype:java", as I mentioned above.

PicsPanel assumes that the webcam has the ID 0; what if it doesn't? On Windows, it's easy to get a list of webcam IDs with JavaCV's videoInput class (which utilizes DirectShow behind the scenes). The ListDevices class shows how:

```
public class ListDevices
{
  public static void main( String args[] )
  {
    int numDevs = videoInput.listDevices();
    System.out.println("No of video input devices: " + numDevs);
    for (int i = 0; i < numDevs; i++)
      System.out.println(" " + i + ": " +
                         videoInput.getDeviceName(i));
  }  // end of main()

} // end of ListDevices class
```

On my test PC, ListDevices prints the following:

```
No of video input devices: 5
 0: USB2.0 Camera
 1: Kinect Virtual Camera : Depth
 2: Kinect Virtual Camera : Image
 3: Kinect Virtual Camera : SmartCam
 4: Video Blaster WebCam 3/WebCam Plus (VFW)
```

The webcam I'm using is the USB camera, which has ID 0.

Section 4 talks about three standalone tools for testing the webcam outside of JavaCV.

Taking a Picture

The call to FrameGrabber.grab() is wrapped up in some exception handling code in PicsPanel.picGrab():

```
private IplImage picGrab(FrameGrabber grabber, int ID)
{
  IplImage im = null;
  try {
    im = grabber.grab();  // take a snap
  }
  catch(Exception e)
  {  System.out.println("Problem grabbing image for " + ID);  }
  return im;
}  // end of picGrab()
```

PicGrab returns an instance of JavaCV's main image class, IplImage, which isn't the same as Java's BufferedImage, but includes methods for converting between the two.

Terminating the Grabber

When the user presses the close box in the JFrame, PicsPanel.closeDown() sets the isRunning boolean to false. This will eventually cause the loop inside PicsPanel.run() to finish, and closeGrabber() is executed:

```
private void closeGrabber(FrameGrabber grabber, int ID)
{
  try {
    grabber.stop();
    grabber.release();
  }
  catch(Exception e)
  {  System.out.println("Problem stopping camera " + ID);  }
}  // end of closeGrabber()
```

The call to FrameGrabber.release() isn't strictly necessary since JavaCV will automatically release the camera at garbage collection time.

2.2. Painting the Panel

The paintComponent() method draws the webcam picture in the panel, and writes the average snap time at the bottom-left.

```
// globals
private static final int WIDTH = 640;
private static final int HEIGHT = 480;

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

private IplImage snapIm = null;


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

  // draw the image and stats
  if (snapIm != null) {
    g.setColor(Color.YELLOW);
    g.drawImage(snapIm.getBufferedImage(), 0, 0, this);
    String statsMsg = String.format("Snap Avg. Time:  %.1f ms",
                      ((double) totalTime / imageCount));
    g.drawString(statsMsg, 5, HEIGHT-10);
                     // write stats in bottom-left corner
  }
  else  { // no image yet
    g.setColor(Color.BLUE);
    g.drawString("Loading from camera " + CAMERA_ID +
                                      "...", 5, HEIGHT-10);
  }
} // end of paintComponent()
```


paintComponent() is called when the application is first made visible, which occurs before an image has been retrieved from the camera. In that case, paintComponent() draws the string "Loading from camera" and the ID at the bottom left of the panel.

When there is an image, it is the JavaCV image format IplImage, and so needs to be converted to a BufferedImage before being drawn. 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();
```

2.3. Saving a Snap

Being able to save a snap is useful in several of the later examples for debugging the image processing outcomes. The current image is saved when the user presses 5 on the number pad, space, or enter, which triggers a call to PicsPanel.takeSnap(). It sets a global boolean takeSnap to true, which is detected inside the loop in run(); the relevant code fragment:

```
// code fragment in PicsPanel.run()
if (takeSnap) {   // save the current images
  saveImage(snapIm, PIC_FNM, snapCount);
  snapCount++;
  takeSnap = false;
}
```

saveImage() saves a grayscale version of the image as a JPG file whose name includes a digit which is incremented after each save:

```
// globals
private static final int WIDTH = 640;
private static final int HEIGHT = 480;

// directory and filenames used to save images
private static final String SAVE_DIR = "pics/";


private void saveImage(IplImage snapIm, String saveFnm,
                                                int snapCount)
{ if (snapIm == null) {
    System.out.println("Not saving a null image");
    return;
  }

  IplImage grayImage  = IplImage.create(WIDTH, HEIGHT,
                                                IPL_DEPTH_8U, 1);
  cvCvtColor(snapIm, grayImage, CV_BGR2GRAY);

  String fnm = (snapCount < 10) ?
              SAVE_DIR + saveFnm + "0" + snapCount +".jpg" :
              SAVE_DIR + saveFnm + snapCount +".jpg";
```



```
  System.out.println("Saving image " + fnm);
  cvSaveImage(fnm, grayImage);
}  // end of saveImage()
```

When cvCvtColor() is called with the CV_BGR2GRAY constant, a grayscale version of the image is written to the IplImage object created with a single 8-bit channel. JavaCV's cvSaveImage() is utilized rather than Java's ImageIO.write() so there's no need to convert IplImage into BufferedImage. Figure 4 shows an example of what's saved.

3. Java OpenCV Grabber

Just a few months ago (as I write this in June 2013), OpenCV finally got it's very own Java binding. What changes are needed to convert SnapPics from JavaCV into "Java OpenCV"?

There's no changes at the top-level, in the SnapPics class, since all the grabber functionality is located inside PicsPanel. Almost all the important adjustments occur in the new version of PicsPanel.run():

```
// Java OpenCV Version
private static final int DELAY = 100;  // ms
private static final int CAMERA_ID = 0;

private Mat snapIm = null;


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

  long duration;
  int snapCount = 0;
  isRunning = true;

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

    snapIm = new Mat();
    grabber.read(snapIm);

    if (takeSnap) {   // save the current images
      saveImage(snapIm, PIC_FNM, snapCount);
      snapCount++;
      takeSnap = false;
    }

    imageCount++;
    repaint();

    duration = System.currentTimeMillis() - startTime;
    totalTime += duration;
    if (duration < DELAY) {
```



```
      try {
        Thread.sleep(DELAY-duration);
      }
      catch (Exception ex) {}
    }
  }
  grabber.release();
}  // end of run()
```

Java OpenCV has a VideoCapture class (see http://docs.opencv.org/java/2.4.5/), which is fairly similar to JavaCV's FrameGrabber.

The set-up of the VideoCapture object is done inside a new version of initGrabber():

```
// Java OpenCV Version
private static final int WIDTH = 640;
private static final int HEIGHT = 480;


private VideoCapture initGrabber(int ID)
{
  VideoCapture grabber = new VideoCapture(ID);
  if ((grabber == null) || (!grabber.isOpened())) {
    System.out.println("Cannot connect to webcam: " + ID);
    System.exit(1);
  }
  else {
    System.out.println("Connected to webcam: " + ID);
    grabber.set(Highgui.CV_CAP_PROP_FRAME_WIDTH, WIDTH);
    grabber.set(Highgui.CV_CAP_PROP_FRAME_HEIGHT, HEIGHT);
        // make sure of image size
  }
  return grabber;
}  // end of initGrabber()
```

The VideoCapture constructor uses a camera ID in the same way as FrameGrabber, and allows the width and height of the grabbed image to be set. However, there's no way to specify the DirectShow format, which fits in with OpenCV's aim of being OS-independent.

There's also no equivalent of JavaCV's videoInput class, which makes it impossible to code a ListDevices class to print ID details. Platform-specific information about a camera requires standalone tools outside of OpenCV, as explained in section 4.

Since VideoCapture.read() and VideoCapture.release() can not raise exceptions, they are called directly in the run() method above. In fact, VideoCapture.read() returns a boolean, which will be false if the image couldn't be retrieved, but I don't bother testing for it in run().

There's no image class in Java OpenCV, whereas JavaCV has IplImage. Instead, OpenCV saves the image returned from a webcam read as a matrix of type Mat. This only affects the coding when the 'image' has to be converted into a Java BufferedImage in paintComponent():

```
// Java OpenCV Version
public void paintComponent(Graphics g)
{
```



```
  super.paintComponent(g);
  g.setFont(msgFont);

  // draw the image and stats on the panel
  if (snapIm != null) {
    g.setColor(Color.YELLOW);
    g.drawImage( matToImage(snapIm), 0, 0, this);
    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()
```

matToImage() converts the matrix into JPG byte format, extracts the raw bytes, and fills a BufferedImage object:

```
private BufferedImage matToImage(Mat snapIm)
{
  MatOfByte matOfByte = new MatOfByte();
  Highgui.imencode(".jpg", snapIm, matOfByte);
  byte[] byteArray = matOfByte.toArray();

  BufferedImage bufImage = null;
  try {
    InputStream in = new ByteArrayInputStream(byteArray);
    bufImage = ImageIO.read(in);
  }
  catch (Exception e) {
    System.out.println("Could not convert matrix
                                      to a BufferedImage");
  }
  return bufImage;
}  // end of matToImage()
```

Saving the image isn't nearly as tricky since OpenCV has matrix-saving functions. There's no need to bother with BufferedImage or Java's ImageIO.write():

```
// Java OpenCV Version
private void saveImage(Mat snapIm, String saveFnm, int snapCount)
{
  if (snapIm == null) {
    System.out.println("Not saving a null image");
    return;
  }

  Mat grayImage = new Mat(WIDTH, HEIGHT, CvType.CV_8UC1);
  Imgproc.cvtColor(snapIm, grayImage, Imgproc.COLOR_BGR2GRAY);

  String fnm = (snapCount < 10) ?
              SAVE_DIR + saveFnm + "0" + snapCount +".jpg" :
              SAVE_DIR + saveFnm + snapCount +".jpg";
  System.out.println("Saving image " + fnm);
  Highgui.imwrite(fnm, grayImage);
```



```
}  // end of saveImage()
```

The conversion of the image to a grayscale uses the same approach as in JavaCV, but with slightly differently named methods and constants.

4. Checking the Camera

A problematic aspect of webcam snapping is making sure that the camera is working. Of course, one solution is to simply plug the camera in and fire up the SnapPics application. But what's to be done when nothing appears in the JPanel?

Although SnapPics is quite small, it rests atop a large number of libraries (i.e. JavaCV, OpenCV, OS imaging support, such as DirectShow, and the camera driver). What part of that stack of software is causing the non-appearance of the webcam picture?

A related issue is how to discover the camera ID that's required by JavaCV's FrameGrabber and Java OpenCV's VideoCapture classes. Almost always, an ID of 0 is the right guess, but what if the ID is something else?

I find it useful to have tools for testing and examining a webcam that are independent of JavaCV and OpenCV. I'll briefly outline three: CommandCam, DevCon and FFmpeg

4.1. CommandCam

CommandCam is a command line webcam image grabber for Windows utilizing DirectShow (http://batchloaf.wordpress.com/commandcam/). It can provide details about all the cameras connected to the device, and allows you to snap a single image and store it as a bitmap.

For example, on my test machine:

```
> commandcam /devlist
```

produces:

```
Available capture devices:
  Device name: USB2.0 Camera
  Device name: Kinect Virtual Camera : Depth
  Device name: Kinect Virtual Camera : Image
  Device name: Kinect Virtual Camera : SmartCam
  Device name: Video Blaster WebCam 3/WebCam Plus (VFW)
```

This information doesn't include ID numbers, but the devices are listed in the same order used by FrameGrabber and CaptureDevice, so you can guess the ID.

A more detailed list of information is provided by the "/devlistdetail" argument:

```
> commandcam /devlistdetail
```

Available capture devices:

```
Capture device 1:
```



```
  Device name: USB2.0 Camera
  Device path: \\?\usb#vid_1e4e&pid_0102&mi_00#6&244eccde&0&0000#
{65e8773d-8f56-11d0-a3b00a0c9223196}\global

Capture device 2:
  Device name: Kinect Virtual Camera : Depth
  Device path: Kinect Virtual Camera : Depth

Capture device 3:
  Device name: Kinect Virtual Camera : Image
  Device path: Kinect Virtual Camera : Image

Capture device 4:
  Device name: Kinect Virtual Camera : SmartCam
  Device path: Kinect Virtual Camera : SmartCam

Capture device 5:
  Device name: Video Blaster WebCam 3/WebCam Plus (VFW)
  Device path: Video Blaster WebCam 3/WebCam Plus (VFW)
```

The numbering of the devices in the output is misleading since OpenCV starts counting from 0 not from 1. The device path information for the USB camera includes its vendor ID (1e4e) and product ID (0102), which can be useful when manipulating the camera as a USB device.

Taking a picture is done with the "/preview" argument, and an optional device number ("/devnum") or device name ("/devname") to specify a camera other than the first. For example:

```
> commandcam /preview /devnum 1
Capture device: USB2.0 Camera
Capture resolution: 640x480
Captured image to image.bmp
```

On one of my Windows 7 test machine, the preview window always remains black (or momentarily shows an image before turning black), but a snap is correctly saved to image.bmp.

4.2. DevCon

Microsoft's DevCon command-line utility is as an alternative to Window's GUI Device Manager which allows you enable, disable, restart, update, remove, and query devices (http://support.microsoft.com/kb/311272). It gives more information on a webcam's status than CommandCam, but doesn't support picture snapping. All the commands I need use the "=image" argument to focus on imaging devices only. The "status" command lists details about connected devices:

```
>devcon status =image

USB\VID_1E4E&PID_0102&MI_00\6&244ECCDE&0&0000
    Name: USB2.0 Camera
    Driver is running.
1 matching device(s) found.
```


This identifies the single camera which is currently plugged into my PC.

The "driverfiles" argument provides information on the device driver:

```
>devcon driverfiles =image

USB\VID_1E4E&PID_0102&MI_00\6&244ECCDE&0&0000
    Name: USB2.0 Camera
    Driver installed from C:\Windows\INF\usbvideo.inf [USBVideo]. 1
file(s) used by driver: C:\Windows\system32\drivers\usbvideo.sys
1 matching device(s) found.
```

4.3. FFmpeg

A drawback of the two previous tools is their limitation to Windows. FFmpeg is a cross-platform collection of software and libraries which supports the recording, conversion and streaming of audio and video (http://www.ffmpeg.org/). I only need the ffmpeg.exe command line tool to grab and encode video.

The FFmpeg site contains lots of documentation, with image capture explained at http://ffmpeg.org/trac/ffmpeg/wiki/How%20to%20capture%20a%20webcam%20input

The following command lists DirectShow supported devices:

```
> ffmpeg -list_devices true -f dshow -i dummy
```

After a large amount of help information has been printed by ffmpeg, the useful stuff is:

```
DirectShow video devices
  "USB2.0 Camera"
  "Kinect Virtual Camera : Depth"
  "Kinect Virtual Camera : Image"
  "Kinect Virtual Camera : SmartCam"
  "Video Blaster WebCam 3/WebCam Plus (VFW)"
DirectShow audio devices
  "Rear Mic (SigmaTel High Definit"
```

The ordering of the output indicates the numbering that should be used by the camera ID in FrameGrabber and CaptureDevice; my USB camera is ID 0.

More details about a particular device can be obtained with the "-list_options" parameter:

```
> ffmpeg -list_options true -f dshow -i video="USB2.0 Camera"
```

The relevant part of the output is:

```
DirectShow video device options
Pin "Capture"
  pixel_format=yuyv422  min s=640x480 fps=30 max s=640x480 fps=30
  pixel_format=yuyv422  min s=640x480 fps=30 max s=640x480 fps=30
  pixel_format=yuyv422  min s=352x288 fps=30 max s=352x288 fps=30
  pixel_format=yuyv422  min s=352x288 fps=30 max s=352x288 fps=30
  pixel_format=yuyv422  min s=320x240 fps=30 max s=320x240 fps=30
  pixel_format=yuyv422  min s=320x240 fps=30 max s=320x240 fps=30
  pixel_format=yuyv422  min s=176x144 fps=30 max s=176x144 fps=30
```



```
  pixel_format=yuyv422  min s=176x144 fps=30 max s=176x144 fps=30
  pixel_format=yuyv422  min s=160x120 fps=30 max s=160x120 fps=30
  pixel_format=yuyv422  min s=160x120 fps=30 max s=160x120 fps=30
```

This indicates that the camera is able to record at 640x480 resolution, with a frame rate of 30 fps. In SnapPics, the DELAY interval is 100 ms which translates to a frame rate of 1000/100 = 10 fps, so is within the capabilities of the device.

Testing the camera is done by recording some video:

```
> ffmpeg -f dshow -s 640x480 -i video="USB2.0 Camera" webcam.flv
```

This uses DirectShow to record video and audio into the webcam.flv file at a resolution of 640x480 pixels. A lot of extra information is printed to the command line during the recording, including the on-going frame rate, which settles down to 30 fps soon after the recording has started.

